Architectures et Systèmes des Calculateurs Parallèles

François PELLEGRINI
ENSEIRB
pelegrin@enseirb.fr

3 octobre 2003

Avant-propos

La loi de Moore, énoncée en 1975, et toujours vérifiée depuis, stipule que la puissance des ordinateurs, à prix égal, double en moyenne tous les 14 mois. Cette amélioration constante en puissance ne peut s'expliquer par la simple augmentation de la fréquence des processeurs, car celle-ci n'a pas suivi la même évolution. Elle est le fruit d'une intense recherche qui porte tout à la fois sur l'architecture générale du processeur, l'optimisation du câblage de ses opérations, les stratégies efficaces de prédiction de branchement, la définition de hiérarchies mémoire, les techniques de compilation avancées, l'optimisation des ressources disques, et l'amélioration des systèmes d'exploitation.

L'objectif de ce cours est de faire un tour d'ensemble des techniques matérielles et logicielles mises en œuvre au sein des architectures des processeurs hautes performances, afin d'en tirer parti au maximum lors de l'écriture de programmes faisant un usage intensif du processeur et de la mémoire.

Ouvrages de référence

- *Highly Parallel Computing – Second edition*, G. S. Almasi et A. Gottlieb. Benjamin Cummings.
- *Advanced Computer Architecture : Parallelism, Scalability, Programmability*, K. Hwang. McGraw-Hill.
- *Designing and Building Parallel Programs*, I. Foster. Addison-Wesley, `http ://www.mcs.anl.gov/dbpp/`.
- *Practical Parallel Computing*, H. S. Morse. AP Professional.
- *Algorithmes et Architectures Parallèles*,M. Cosnard et D. Trystram.InterÉditions.
- *CPU Info Center*, `http ://infopad.eecs.berkeley.edu/CIC/`.
- *Journal of Parallel and Distributed Computing*, ...

Chapitre 1

Introduction

1.1 Un aperçu du parallélisme

Depuis les débuts de l'informatique s'est posée la question de résoudre rapidement des problèmes (le plus souvent numériques) coûteux en temps de calcul : simulations numériques, cryptographie, imagerie, S.G.B.D., etc.

Pour résoudre plus rapidement un problème donné, une idée naturelle consiste à faire coopérer simultanément plusieurs agents à sa résolution, qui travailleront donc *en parallèle*.

À titre d'illustration, on peut se représenter le travail d'un maçon en train de monter un mur de briques. S'il est seul, il procède rangée par rangée (figure 1.1).

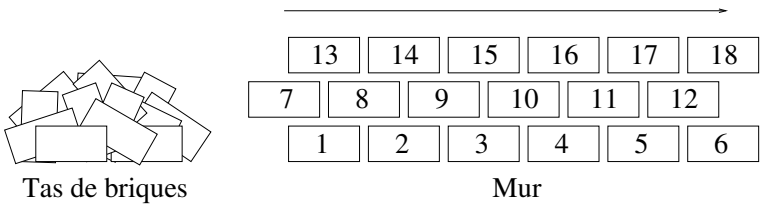


FIG. 1.1 – Séquencement du travail d'un maçon travaillant seul.

Si l'on veut monter le mur plus rapidement, on peut faire appel à deux maçons, qui peuvent organiser leur travail de plusieurs manières différentes.

a) Chacun pose une brique, l'un après l'autre (figure 1.2). Dans ce cas, ils risquent de se gêner mutuellement, tant pour prendre les briques dans le tas que pour les mettre en place.

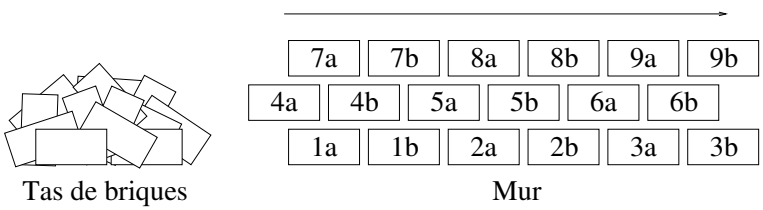


FIG. 1.2 – Séquencement du travail de deux maçons travaillant brique par brique.

b) Chacun s'attribue une portion de mur pour travailler (figure 1.3). Ils ne se gênent plus, mais le maçon le plus éloigné du tas a plus de chemin à faire, et sa partie du mur avancera moins vite. Remarquons également qu'ils se gênent toujours pour prendre les briques.

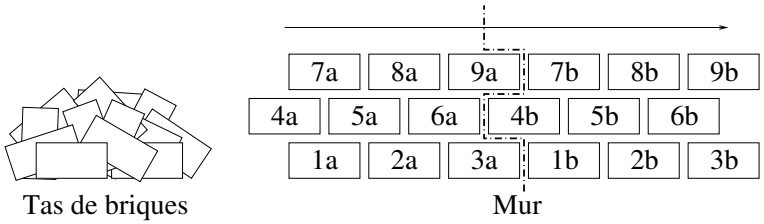


FIG. 1.3 – Séquencement du travail de deux maçons travaillant sur deux portions de mur séparées.

c) Chacun s'attribue une portion de mur pour travailler, mais le maçon le plus près du tas lance une brique à l'autre chaque fois qu'il en prend une pour lui. Dans ce cas, ils ne se gênent plus ni pour prendre les briques, ni dans leur travail. Cependant, ils doivent bien savoir viser et attraper...

Le montage du mur en parallèle est plus rapide que le montage par un seul maçon, mais la quantité totale de travail est nécessairement plus importante, car il faut que les maçons s'organisent entre eux.

Cet exemple impose plusieurs réflexions.

- Pour que la résolution parallèle soit possible, il faut que le problème puisse être décomposé en sous-problèmes suffisamment indépendants les uns des autres pour que chaque agent puisse travailler sans perturber les autres.
- Il faut pouvoir organiser efficacement le travail à répartir. En plus du coût de calcul intrinsèque du problème, on génère un surcoût dû aux calculs annexes et à la communication entre agents de l'information nécessaire à sa résolution.

Les problèmes réels sont parallélisables à des degrés différents. Parfois, il est même plus intéressant d'éviter le parallélisme si le surcoût engendré par celui-ci est trop important. C'est tout-à-fait regrettable, mais il existe des algorithmes intrinsèquement séquentiels.

L'obtention d'une version parallèle efficace d'un algorithme peut conduire à une formulation très différente de l'algorithme séquentiel équivalent. En fait, un problème a souvent plusieurs formulations parallèles différentes, dont les performances peuvent elles aussi être très différentes.

1.2 Le parallélisme est-il nécessaire ?

La puissance des ordinateurs séquentiels augmentant de manière régulière (en gros, elle est multipliée par deux tous les quatorze mois), on pourrait croire qu'elles sera toujours suffisante, et que les machines parallèles (ordinateurs multi-processeurs) sont inutiles. C'est faux, pour plusieurs raisons.

- Plus on en a, plus on en veut. À mesure que la puissance des machines augmente, on introduit l'outil informatique dans des disciplines où il ne pouvait jusqu'alors pénétrer, et on cherche à intégrer de plus en plus de paramètres dans les modèles numériques : météorologie, synthèse et reconstruction d'images, simulations numériques, etc.
  Un certains nombre d'applications « sensibles » ont été classées « *Grand Challenge* », et font l'objet de recherches intensives, tant au niveau du matériel que

du logiciel. Elles sont également appelées « applications 3T », parce qu'elles nécessitent pour leur exécution :
- 1 Téra[1] flops (« *floating operation per second* ») ;
- 1 Téra octet de mémoire centrale ;
- 1 Téra octet par seconde de bande passante pour produire les résultats.

À l'heure actuelle, ces applications ne peuvent être réalisées qu'en ayant recours au parallélisme massif, sur des machines à plus de 8000 processeurs hautes performances [13].

- La vitesse de la lumière est (actuellement) une limitation intrinsèque à la vitesse des processeurs. Supposons en effet que l'on veuille construire une machine entièrement séquentielle disposant d'une puissance de 1 Tflops et de 1 To de mémoire.
  Soit d la distance maximale entre la mémoire et le micro-processeur. Cette distance doit pouvoir être parcourue 10^{12} fois par seconde à la vitesse de la lumière, $c \approx 3.10^8 m.s^{-1}$, d'où :

$$d \leq \frac{3.10^8}{10^{12}} = 0,3mm \ .$$

  L'ordinateur devrait donc tenir dans une sphère de $0,3$ mm de rayon. Avec cette contrainte de distance, si l'on considère la mémoire comme une grille carrée de $10^6 \times 10^6$ octets, alors chaque octet doit occuper une cellule de 3Å de côté, c'est à dire la surface occupée par un petit atome. On ne tient ici pas compte de l'espace nécessaire à l'acheminement de l'information et de l'énergie, ainsi qu'à l'extraction de la chaleur.
  Cette argumentation est biaisée en ce que la mise en œuvre d'une hiérarchie mémoire (voir section 4.1) permet d'augmenter la distance entre la mémoire de masse et l'unité de traitement. Néanmoins, elle reste globalement valable.

1.3 La recherche en parallélisme

L'utilisation efficace de machines parallèles nécessite de travailler sur :
- l'architecture des machines. Il faut assurer que :
  - la machine est extensible (« *scalable* ») : on peut (facilement) augmenter la taille de la machine sans que les performances s'écroulent ;
  - les échanges de données entre processeurs sont rapides, pour éviter leur famine ;
  - les entrées/sorties ne sont pas pénalisantes.
- les modèles d'expression du parallélisme : chaque algorithme possède un modèle de parallélisme avec lequel il s'exprime mieux ;
- les langages parallèles : il faut choisir le langage le plus adapté au problème ;
- l'algorithmique proprement dite : de nombreux problèmes pour lesquels il existe un algorithme séquentiel optimal ne possèdent encore pas de contrepartie parallèle efficace ;
- l'environnement de développement : débogueurs, profileurs, bibliothèques portables, etc. ;
- la parallélisation automatique : compilateurs « *data-parallèles* » ou « *multithreads* » avec directives dans le cas de HPF [6] et d'OpenMP [9], parallélisation automatique de boucles, etc.

[1] 1 Téra = 10^3 Giga = 10^6 Méga = 10^9 Kilo = 10^{12}.

Chapitre 2

Modèles de calculateurs parallèles

Afin de définir et comparer les architectures de machines, plusieurs classifications ont été développées.

2.1 Classification de Flynn

La classification la plus connue est celle de Flynn [2], qui caractérise les machines suivant leurs flots de données et d'instructions, ce qui donne quatre catégories :

– SISD (« *Single Instruction stream, Single Data stream* »). Cette catégorie correspond aux machines séquentielles conventionnelles, pour lesquelles chaque opération s'effectue sur une donnée à la fois (figure 2.1) ;

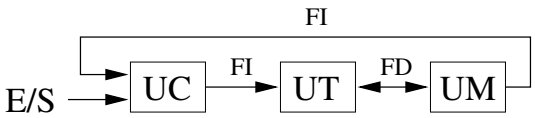


Fig. 2.1 – Architecture SISD. L'unité de contrôle (UC), recevant son flot d'instructions (FI) de l'unité mémoire (UM), envoie les instructions à l'unité de traitement (UT), qui effectue ses opérations sur le flot de données (FD) provenant de l'unité mémoire.

– MISD (« *Multiple Instruction stream, Single Data stream* »). Cette catégorie regroupe les machines spécialisées de type « systolique », dont les processeurs, arrangés selon une topologie fixe, sont fortement synchronisés (figure 2.2) ;

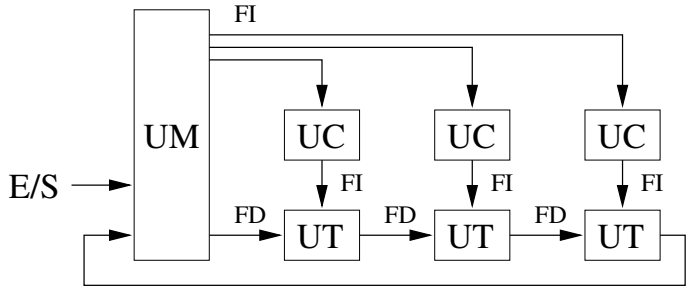


Fig. 2.2 – Architecture MISD.

– SIMD (« *Single Instruction stream, Multiple Data stream* »). Dans cette classe d'architectures, les processeurs sont fortement synchronisés, et exécutent au même instant la même instruction, chacun sur des données différentes (figure 2.3). Des informations de contexte (bits de masquage) permettent d'inhiber l'exécution d'une instruction sur une partie des processeurs.

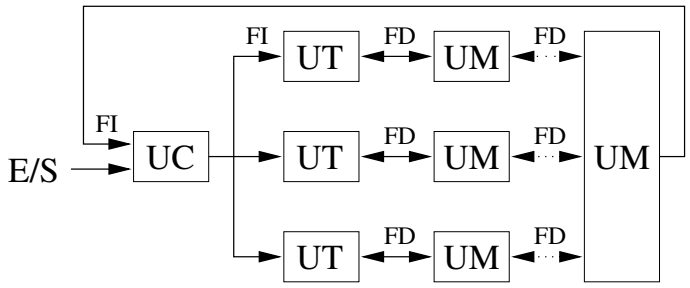


FIG. 2.3 – Architecture SIMD.

Ces machines sont adaptées aux traitements réguliers, comme le calcul matriciel sur matrices pleines ou le traitement d'images. Elles perdent en revanche toute efficacité lorsque les traitements à effectuer sont irréguliers et dépendent fortement des données locales, car dans ce cas les processeurs sont inactifs la majorité du temps.
Ainsi, pour exécuter une instruction conditionnelle de type `if...then...else` (figure 2.4), l'ensemble des instructions des deux branches doit être présenté aux processeurs, qui décident ou non de les exécuter en fonction de leur bit local d'activité, positionné en fonction des valeurs de leurs variables locales. Chacun des processeurs n'exécutera effectivement que les instructions de l'une des branches.

Code source	Code compilé	Exécution, `cond`=VRAI	Exécution, `cond`=FAUX

`blocA`
`if (cond)`
`  blocV;`
`else`
`  blocF;`
 	
`blocB`
	`blocA;`
	`ACTIF = (cond);`
	`blocV;`
	`ACTIF = ~ACTIF;`
	`blocF;`
	`ACTIF = VRAI`
	`blocB`
		`blocA;`
		`ACTIF = (cond);`
		`blocV;`
		`ACTIF = ~ACTIF;`
		`--`
		`ACTIF = VRAI`
		`blocB`
			`blocA;`
			`ACTIF = (cond);`
			`--`
			`ACTIF = ~ACTIF;`
			`blocF;`
			`ACTIF = VRAI`
			`blocB`

FIG. 2.4 – Exécution d'une expression conditionnelle `if...then...else` sur une architecture SIMD.

– MIMD (« *Multiple Instruction stream, Multiple Data stream* »). Cette classe comprend les machines multi-processeurs, où chaque processeur exécute son propre code de manière asynchrone et indépendante. On distingue habituellement deux sous-classes, selon que les processeurs de la machine ont accès à une mémoire commune (on parle alors de MIMD à mémoire partagée, « *multiprocessor* », figure 2.5), ou disposent chacun d'une mémoire propre (MIMD à mémoire distribuée, « *multicomputer* », figure 2.6). Dans ce dernier cas, un réseau d'interconnexion est nécessaire pour échanger les informations entre processeurs.

Cette classification est trop simple, car elle ne prend en compte ni les machines vectorielles (qu'il faut ranger dans la catégorie SISD et non pas SIMD, car elles ne

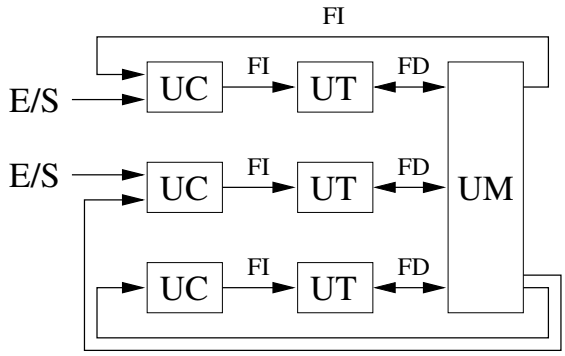


FIG. 2.5 – Architecture MIMD à mémoire partagée.

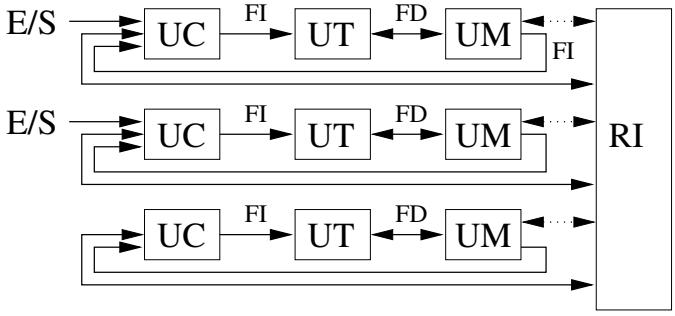


FIG. 2.6 – Architecture MIMD à mémoire distribuée.

disposent que d'un seul flot mémoire, voir section 3.7), ni les différences d'architecture mémoire.

2.2 Classification de Raina

Une sous-classification étendue des machines MIMD, due à Raina [11], et illustrée en figure 2.7, permet de prendre en compte de manière fine les architectures mémoire, selon deux critères :

- l'organisation de l'espace d'adressage :
  - SASM (« Single Address space, Shared Memory ») : mémoire partagée ;
  - DADM (« Distributed Address space, Distributed Memory ») : mémoire distribuée, sans accès aux données distantes. L'échange de données entre processeurs s'effectue nécessairement par passage de messages, au moyen d'un réseau de communication ;
  - SADM (« Single Address space, Distributed Memory ») : mémoire distribuée, avec espace d'adressage global, autorisant éventuellement l'accès aux données situées sur d'autres processeurs.
- le type d'accès mémoire mis en œuvre :
  - NORMA (« *No Remote Memory Access* ») : pas de moyen d'accès aux données distantes, nécessitant le passage de messages ;
  - UMA (« *Uniform Memory Access* ») : accès symétrique à la mémoire, de coût identique pour tous les processeurs ;
  - NUMA (« *Non-Uniform Memory Access* ») : les performances d'accès dépendent de la localisation des données ;

– CC-NUMA (« *Cache-Coherent NUMA* ») : type d'architecture NUMA intégrant des caches ;
– OSMA (« *Operating System Memory Access* ») : les accès aux données distantes sont gérées par le système d'exploitation, qui traite les défauts de page au niveau logiciel et gère les requêtes d'envoi/copie de pages distantes ;
– COMA (« *Cache Only Memory Access* ») : les mémoires locales se comportent comme des caches, de telle sorte qu'une donnée n'a pas de processeur propriétaire ni d'emplacement déterminé en mémoire.

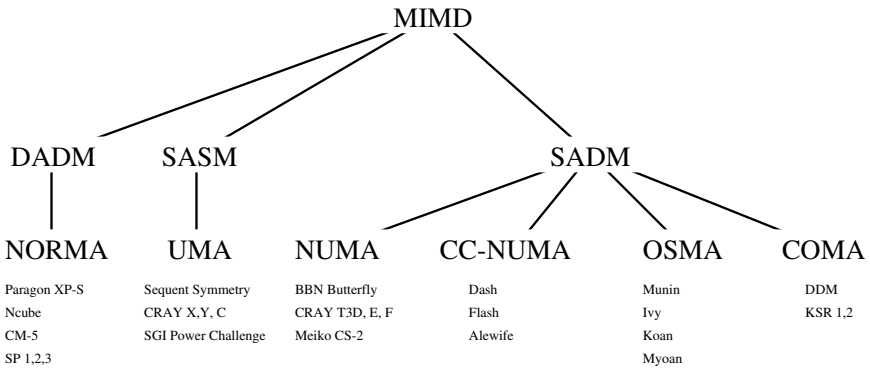


FIG. 2.7 – Classification MIMD de Raina.

Chapitre 3

Architecture des processeurs

L'obtention de performances élevées sur une architecture parallèle nécessite l'obtention de performances de calcul élevées sur chacun des nœuds qui la composent. Pour ce faire, il est nécessaire de connaître les principes architecturaux mis en œuvre dans les processeurs haut-de-gamme actuels, afin d'en tirer pleinement parti lors de l'écriture des logiciels, tout en conservant à ceux-ci une portabilité maximale.

3.1 Horloge

La vitesse d'un processeur dépend en premier lieu de la durée de son cycle d'horloge, qui cadence le système. Plus cette période est courte, plus le processeur est rapide. Cependant, disposer d'un processeur rapide ne sert à rien si les composants annexes (bus, mémoire) sont trop lents : le processeur passera son temps à les attendre.

La fréquence d'horloge est le nombre de cycles d'horloge par seconde, mesurée en Hertz (Hz). La relation entre le cycle d'horloge τ et la fréquence d'horloge f est donnée par la relation $f = \frac{1}{\tau}$.

Actuellement, selon la technologie utilisée (bipolaire, CMOS, etc.), les temps de cycle vont de 250 ps à 60 ns, ce qui correspond à des fréquences de 16 MHz à 4 GHz.

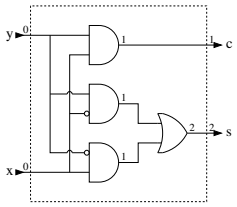
On ne peut augmenter la fréquence à l'infini, et l'on pense que les limites de la technologie actuelle seront bientôt atteintes, aux alentours de 10 GHz. Il faut également noter qu'il y a un lien direct entre la fréquence d'horloge et le prix du processeur (c'est d'ailleurs un argument commercial). Le parallélisme semble donc intéressant en ce qu'il permet de tirer parti de processeurs moins puissants, mais plus nombreux.

3.2 Câblage

Un premier moyen d'accélérer le traitement des opérations par le processeur consiste à exhiber le parallélisme au niveau des bits. Nous allons illustrer cette approche en étudiant un additionneur et un multiplicateur en arithmétique entière.

3.2.1 Additionneur

On peut réaliser l'addition de deux bits au moyen du circuit présenté en figure 3.1, appelé « demi-additionneur » (« *half-adder* », ou HA), et constitué de deux niveaux de portes logiques « et » et « ou ». Le bit s correspond à la somme, et c à la retenue.



x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$s_i = x_i \oplus y_i = x_i\overline{y_i} + \overline{x_i}y_i$$
$$c_i = x_i y_i$$

FIG. 3.1 – Schéma, table, et équations logiques d'un demi-additionneur binaire (HA).

En combinant deux HA, on peut réaliser une tranche d'additionneur complet (« *full adder* », ou FA), représentée en figure 3.2.

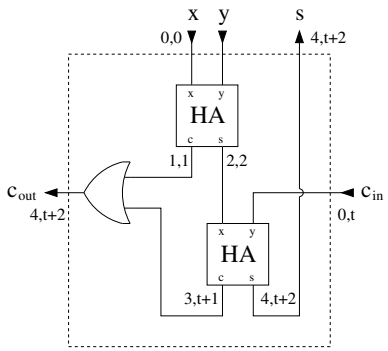


FIG. 3.2 – Schéma d'une tranche d'additionneur binaire (FA).

En chaînant ensemble des FA, on peut alors construire un additionneur par propagation de retenue (« *Ripple Carry Adder* », ou RCA), montré en figure 3.3.

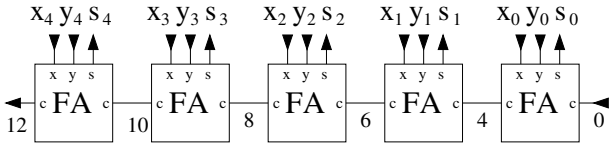


FIG. 3.3 – Schéma d'un additionneur binaire à propagation de retenue (RCA).

Le temps de calcul d'une addition sur n bits est donc :

$$t_{\mathrm{RCA}}(n) = 2n + 2 \ .$$

L'additionneur ci-dessus présente une très forte séquentialité, qui dérive de la nécessité de connaître la retenue c_{in} du bit i pour calculer celle du bit $i+1$. Plus la valeur des retenues partielles sera connue tôt, et plus le calcul des bits de la somme pourra être accéléré. Le circuit FA peut nous apporter des informations, au prix d'une légère modification.
Le circuit modifié FA' de la figure 3.4 possède deux sorties supplémentaires, g et p, qui indiquent respectivement si une retenue a été générée au sein de l'additionneur, et si une retenue c_{in} éventuelle sera propagée.

Si l'on considère deux additionneurs FA', et que l'on cherche à calculer les valeurs globales de g et p, on trouve :

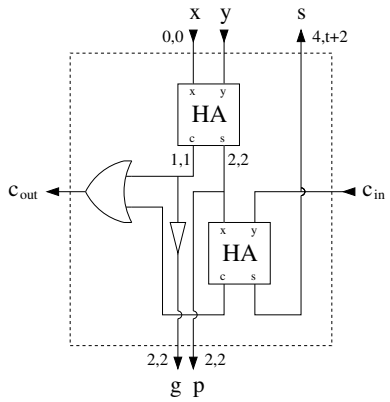
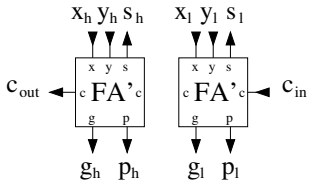


FIG. 3.4 – Schéma d'une tranche d'additionneur binaire modifiée (FA').



$$\begin{aligned} g &= g_h + g_l p_h \\ p &= p_h p_l \\ c_{out} &= p c_{in} + g \end{aligned}$$

Les formules logiques de p, g, et c_{out} peuvent être câblées dans le circuit combineur de retenue (« *Carry Merger* », ou CM) présenté en figure 3.5, page 17.

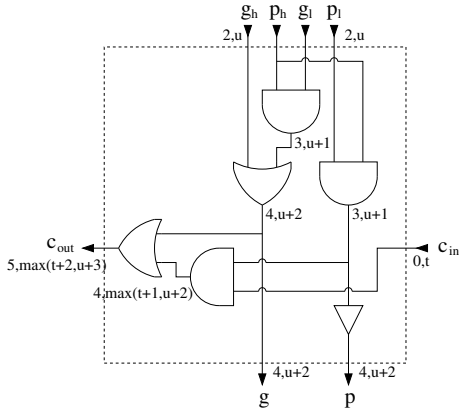


FIG. 3.5 – Schéma d'un combineur de retenues (CM).

On peut alors réaliser un additionneur complet par pré-calcul de retenue (« *Carry Lookahead Adder* », ou CLA) selon le schéma de la figure 3.6, page 18.

Le temps de calcul sur $n = 2^k$ bits de cet additionneur est donc :

$$\begin{aligned} t_{\mathrm{CLA}}(n) &= \underbrace{2}_{\text{descente FA'}} + \underbrace{2\left(\log_2(n) - 2\right)}_{\text{descente}} + \underbrace{3}_{\text{virage}} + \underbrace{2\left(\log_2(n) - 2\right)}_{\text{remontée}} + \underbrace{2+2}_{\text{remontée FA'}} \\ &= 4\left\lceil \log_2(n) \right\rceil + 1 \end{aligned}$$

Le tableau ci-dessous permet de constater les gains réalisés.

n	8	16	32	64	128
RCA	18	34	66	130	258
CLA	13	17	21	25	29

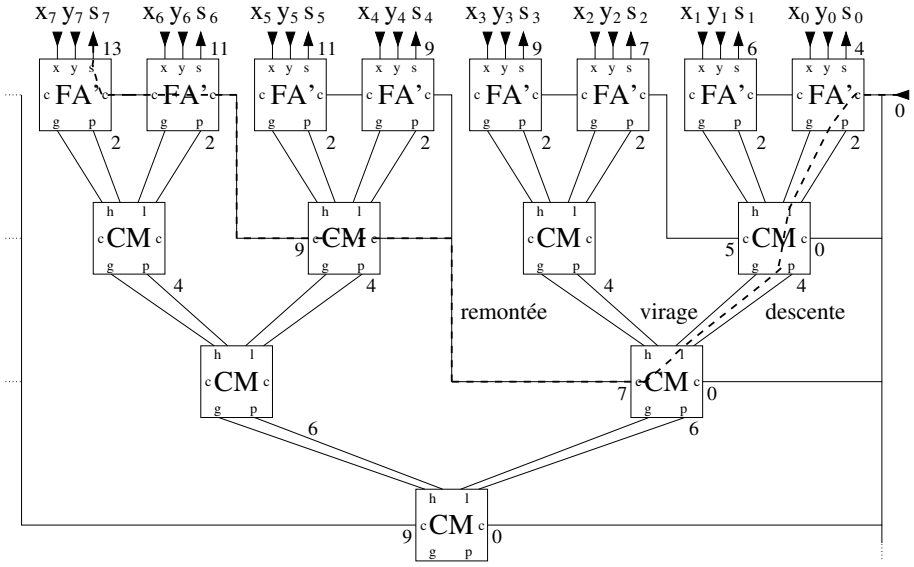


FIG. 3.6 – Schéma d'un additionneur binaire à pré-calcul de retenue (CLA).

3.2.2 Multiplicateur

En arithmétique binaire, la multiplication s'exprime simplement au moyen de décalages et d'additions. Par exemple, si l'on considère la multiplication de deux nombres A et B codés chacun sur 8 bits :

	A	=							1	0	1	1	0	0	0	1
$*$	B	=							1	1	0	1	0	0	1	1
									1	0	1	1	0	0	0	1
$+$								1	0	1	1	0	0	0	1	
$+$							0	0	0	0	0	0	0	0		
$+$						0	0	0	0	0	0	0	0			
$+$					1	0	1	1	0	0	0	1				
$+$				0	0	0	0	0	0	0	0					
$+$			1	0	1	1	0	0	0	1						
$+$		1	0	1	1	0	0	0	1							
	1	0	0	1	0	0	0	1	1	1	1	0	0	0	1	1

multiplier A par B revient à sommer les huit produits partiels obtenus en décalant de i bits le mot obtenu par un « et » logique entre chacun des bits de A et le $i^{ème}$ bit de B.

La réalisation d'un multiplicateur efficace est bien plus délicate que celle d'un additionneur, en ce que l'opération de multiplication nécessite de nombreuses additions. Le goulet d'étranglement d'un additionneur étant la propagation de la retenue, qui sérialise les calculs, il faut limiter leur nombre autant que possible dans le multiplicateur afin d'obtenir un parallélisme maximum.

L'additionneur « *à conservation de retenue* » (« *Carry Save Adder* », ou CSA) permet d'effectuer l'addition de trois nombres binaires, en conservant les retenues de chaque bit dans un vecteur auxiliaire, Ainsi, si X, Y, et Z sont trois nombres

codés sur 8 bits, on aura :

$$\begin{array}{rrcccccccc} & X & = & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ + & Y & = & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ + & Z & = & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ \hline & S^b & = & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ & C^b & = & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \end{array}$$

avec

$$\begin{array}{rcl} s_i^b & = & x_i \oplus y_i \oplus z_i \;=\; x_i y_i z_i + x_i \overline{y_i z_i} + \overline{x_i} y_i \overline{z_i} + \overline{x_i y_i} z_i \\ c_i^b & = & x_{i-1} y_{i-1} + x_{i-1} z_{i-1} + y_{i-1} z_{i-1} \;\;, \end{array}$$

où S^b est le vecteur somme bit à bit, toujours sur 8 bits, et C^b est le vecteur retenue bit à bit, sur neuf bits mais tel que le bit de poids le plus faible soit toujours zéro. Le résultat produit vérifie bien $S^b + C^b = X + Y + Z$, et tous les bits de S^b et C^b ont été calculés en parallèle.

En combinant entre eux les additionneurs CSA pour former un arbre de Wallace, et en intercalant des tampons (« *latches* »), on construit un multiplicateur pipeliné (voir section 3.4 pour de plus amples informations sur les pipe-lines), présenté en figure 3.7, dont le dernier étage est un additionneur à propagation de retenue (« *Carry Propagate Adder* », ou CPA) comme l'additionneur CLA décrit plus haut.

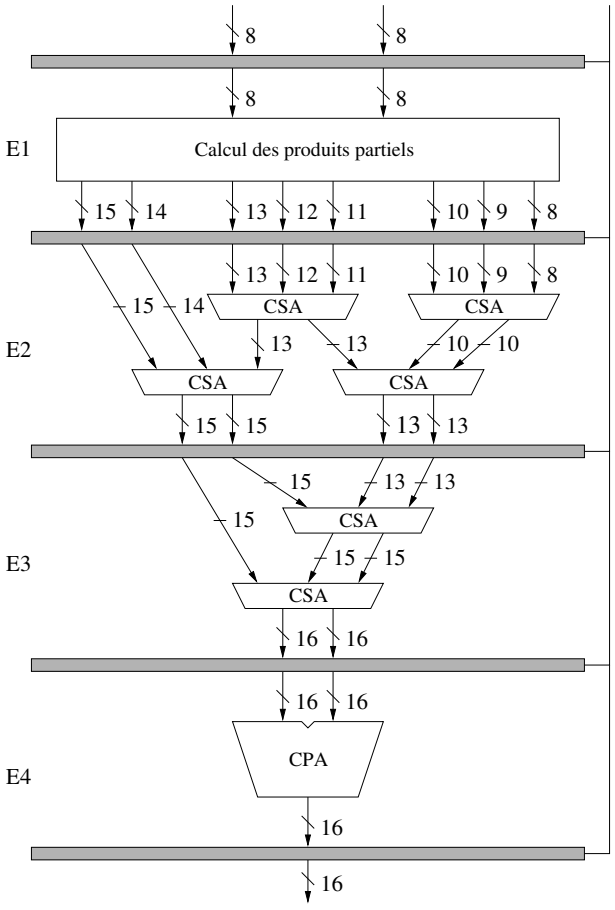


FIG. 3.7 – Schéma d'un multiplicateur binaire.

En utilisant autant que possible des additionneurs CSA, on a considérablement réduit le nombre de propagations de retenues à calculer (il n'en reste qu'une seule,

inévitable). Pour un résultat sur 16 bits, l'additionneur CLA a une profondeur de quatre niveaux, chaque CSA nécessite deux niveaux de portes logiques, et la génération des produits partiels (au moyen de « et » logiques) nécessite également deux niveaux. Le pipe-line est donc relativement équilibré.

Dans les deux exemples d'opérateurs arithmétiques présentés dans cette section, à chaque fois, on a gagné en temps au prix d'un surcoût de calcul (mesuré ici en nombre de portes logiques).

3.3 Jeu d'instructions

On oppose conceptuellement deux types de jeux d'instructions :
- RISC (« *Reduced Instruction Set Computer* ») : jeu d'instructions réduit ;
- CISC (« *Complex Instruction Set Computer* ») : jeu d'instructions complexe.

Dans les faits, ces concepts impliquent d'autres choix technologiques, que nous allons présenter dans cette section.

Les premiers processeurs étaient RISC par nature, puisqu'ils possédaient un jeu d'instructions très réduit. Dans les années 1960-1970, on s'est orienté vers une complexification des jeux d'instructions, afin de simplifier l'écriture des compilateurs et d'économiser la mémoire en réduisant la taille des programmes. L'amélioration des techniques d'intégration et l'augmentation des fréquences d'horloge compensaient largement le surcoût induit par cette complexification. Le plus souvent, les instructions complexes n'étaient pas câblées, mais micro-codées.

À titre d'exemple, le jeu d'instructions de la famille VAX de DEC possédait 304 instructions, dont certaines étaient de très haut niveau : l'instruction POLY servait à évaluer les polynômes !

Des études statistiques ont alors montré que la plupart des instructions n'étaient en fait pas utilisées, car trop spécialisées et de trop haut niveau pour qu'un compilateur puisse les générer à partir d'un code source. On s'est donc naturellement orienté vers une simplification des jeux d'instructions, marquée par la naissance en 1972 du premier processeur délibérément RISC, le RISC I de l'Université de Berkeley, qui ne possédait que 32 instructions. On peut remarquer que cette date charnière coïncide avec celle du développement de la théorie de la compilation, qui a permis la réalisation de compilateurs efficaces.

Réduire la taille du jeu d'instructions permet de gagner :
- en temps de décodage des instructions, du fait de la plus grande simplicité de celles-ci, ce qui permet de réduire le nombre de niveaux de portes logiques à traverser pour exécuter une instruction ;
- en surface d'intégration, de par la réduction de la circuiterie de décodage et l'absence de la gestion du micro-code, ce qui diminue la longueur maximale des pistes à l'intérieur du processeur.

La combinaison de ces deux gains, en termes de nombre de niveaux logiques et de longueur de pistes, permet une augmentation significative de la fréquence d'horloge dans les architectures RISC, par rapport aux processeurs CISC. Un rapport de deux à quatre est courant à l'heure actuelle.

Les caractéristiques généralement admises des architectures RISC sont les suivantes :
- toutes les instructions ont le même format et la même taille. Ceci simplifie leur décodage, mais aussi les accès mémoire, car dans la plupart des processeurs RISC les instructions doivent être alignées sur la taille d'un mot machine.

Cette caractéristique est également essentielle pour l'optimisation des architectures pipe-linées ;
- le jeu d'instruction est de type « *load-store* ». Les seules instructions pouvant accéder à la mémoire sont les opérations « *load* » et « *store* », les autres opérations n'opérant que sur les registres du processeur. Les processeurs RISC disposent donc d'un grand nombre de registres, afin de stocker les valeurs des opérandes qui ne peuvent plus être directement accédées en mémoire ;
- l'architecture est « orthogonale ». Chaque instruction peut utiliser indifféremment toutes les opérandes des types autorisés. On n'a donc pas de registres spécialisés, comme c'est par exemple le cas pour la famille des 80x86, où seul le registre AX sert aux opérations arithmétiques, le registre CX comme compteur de répétition, etc. ;
- la plupart des instructions s'exécutent en un cycle d'horloge. Dans les architectures RISC « pures », toutes les instructions s'exécutent en un cycle d'horloge, à part les accès mémoire qui peuvent prendre plus de temps. Cependant, pour des raisons d'efficacité, on tend actuellement à inclure dans des processeurs dits RISC des instructions (le plus souvent arithmétiques : multiplication, division) s'exécutant en quelques cycles d'horloge, bien plus efficaces que leur contrepartie logicielle car câblées de façon optimisée ; ces processeurs évolués sont parfois appelés CRISC (« *Complexified RISC* »). Dans tous les cas cependant, on n'a jamais recours à un micro-code ;
- le jeu d'instructions est limité uniquement aux instructions nécessaires en terme de compromis performance/place/pipe-line. Ainsi, les premiers processeurs SPARC ne possédaient pas de multiplication câblée (ce qui a été rajouté assez rapidement, il est vrai...).

Ces choix architecturaux ont un effet certain sur l'écriture des compilateurs. Si leur écriture peut sembler plus compliquée au premier abord (nécessité d'émuler les modes d'adressage étendus, les opérations arithmétiques complexes, etc.), on peut gagner par rapport aux processeurs CISC pour lesquels :
- les instructions de trop haut niveau (comme le POLY du VAX) ne peuvent être générées à partir d'un langage de bas niveau (comme le C) ;
- l'architecture non-orthogonale complique la gestion des registres (sauvegardes et restauration perpétuelles).

3.4 Pipe-line

3.4.1 Principe

Le pipe-line est une technique permettant d'exploiter le parallélisme induit par l'exécution répétée d'une même opération sur des données distinctes. On peut très simplement comparer un pipe-line à une chaîne de montage dans une usine. On décompose l'unité de traitement de l'opération en sous-unités indépendantes, qui travaillent en parallèle sur les données qui se présentent séquentiellement à elles, chaque sous-unité travaillant à un instant donné sur une donnée différente.

Trois conditions sont nécessaires à l'élaboration d'un pipe-line :
- une opération de base doit être répétée dans le temps ;
- cette opération doit pouvoir être décomposée en étapes (étages) indépendants ;
- la complexité de ces étages doit être à peut près la même. Si ce n'est pas le cas, on peut multiplexer les étages les plus coûteux pour augmenter le débit du pipe-line, comme illustré en figure 3.8.

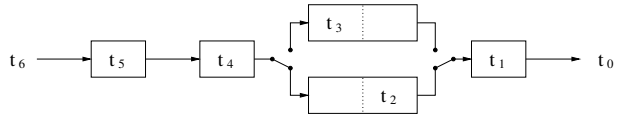


FIG. 3.8 – Utilisation du multiplexage pour augmenter le débit d'un pipe-line. Les commutateurs basculent à chaque cycle.

Ainsi, typiquement, le traitement des instructions par le processeur peut se décomposer en cinq étapes :
- « *fetch* » : recherche de la prochaine instruction à exécuter ;
- « *decode* » : décodage de l'instruction, avec calculs éventuels des adresses ;
- « *read* » : chargement des opérandes dans l'unité d'exécution, par lecture à partir des registres ou de la mémoire ;
- « *execute* » : exécution proprement dite de l'instruction ;
- « *write* » : écriture du résultat vers les registres ou la mémoire.

Le nombre d'étages composant un pipe-line est appelé profondeur du pipe-line. Dans le cas général, un pipe-line de profondeur p peut exécuter n opérations en $n + p$ étapes, s'il n'y a pas de bulles. Sans pipe-line, le temps d'exécution serait de $n\,p$, d'où un facteur d'accélération de $\frac{np}{n+p}$. Quand $n \gg p$, $n + p \simeq n$, ce qui donne une accélération de p, ce qui suggère d'augmenter le nombre d'étages afin de bénéficier de l'accélération la plus grande possible. Cependant, plus le nombre d'étages augmente, et plus le risque d'apparition de bulles augmente, ce qui réduit l'efficacité du pipe-line.

En règle générale, le nombre d'étages des pipe-lines d'instructions est donc compris entre 5 et 15. Ainsi, si le Pentium d'Intel avait 5 étages, les Pentium II et III en ont 12, et le Pentium IV en a 20, ce qui lui a permis d'augmenter considérablement sa fréquence de fonctionnement par rapport aux précédents.

Les figures 3.9 et 3.10 représentent l'exécution dans le temps d'une séquence d'instructions. Sans pipe-line, l'exécution de chaque instruction ne peut se faire que lorsque la précédente a été entièrement traitée. Avec pipe-line, les différents étages de traitement peuvent travailler en parallèle, si les instructions ne présentent pas de dépendences. Sinon, des bulles peuvent apparaître à l'exécution.

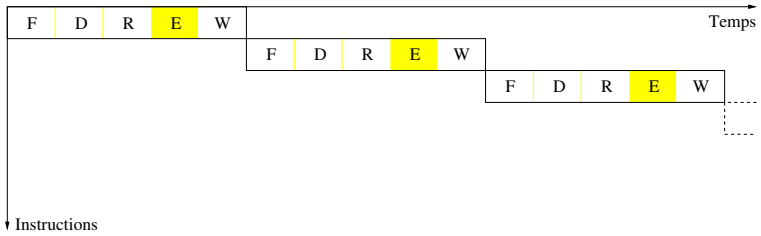


FIG. 3.9 – Exécution d'une séquence d'instructions sur une machine non pipelinée.

Un autre domaine d'application classique des pipe-line est l'unité arithmétique et logique, dont les nombreuses fonctions sont susceptibles d'être pipe-linées. Le calcul d'une addition en virgule flottante peut ainsi se décomposer en cinq étapes :
- comparaison des exposants et calcul de leur différence (soustraction entière) ;
- alignement des mantisses en conséquence (décalage) ;
- addition des mantisses (addition entière) ;

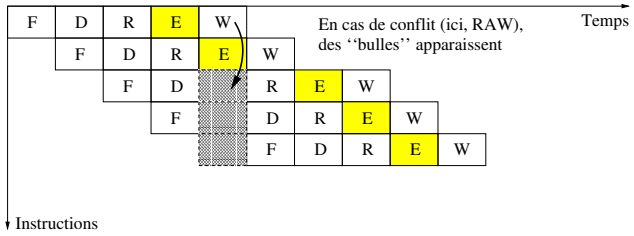


FIG. 3.10 – Exécution d'une séquence d'instructions sur une machine pipelinée à cinq étages. La troisième instruction nécessitant le résultat calculé par la première (conflit « *Read After Write* », ou « RAW »), une bulle apparaît dans le pipe-line.

– calcul du facteur de renormalisation (comptage de bits à zéro) ;
– normalisation du résultat (décalage).

3.4.2 Pipe-lines non linéaires

Afin d'implémenter des opérations complexes tout en économisant de la place, il est possible de câbler plusieurs fonctions au sein de la même unité pipe-line. Dans ce cas, en plus des liens directs entre étages voisins, on trouvra des connexions avant (« *feedforward* ») et arrière (« *feedback* »), ainsi que plusieurs sorties, qui seront activées ou non suivant la configuration dynamique du pipe-line, comme illustré en figure 3.11.

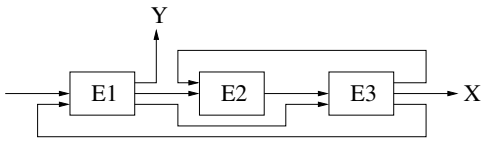


FIG. 3.11 – Exemple de pipe-line non linéaire.

Ces connexions non-linéaires compliquent beaucoup l'utilisation du pipe-line, et en particulier la réservation des différents étages en fonction des opérations demandées. L'enchainement des opérations dans le pipe-line est habituellement représenté au moyen d'une table de réservation (« *reservation table* »), dont les lignes représentent les étages du pipe-line, et les colonnes les pas de temps nécessaires à l'évaluation de la fonction associée. Dans le cas d'un pipe-line linéaire, cette table est triviale, puisque les étages sont traversés dans l'ordre. Dans le cas d'un pipe-line non-linéaire, les tables sont plus complexes, et à une structure de pipe-line donnée peuvent correspondre plusieurs tables, définissant chacune une fonction différente, comme par exemples celles définies en figure 3.12.

Le fonctionnement du pipe-line peut lui aussi être représenté sous forme de table, avec un format dérivant de celui des tables de réservation. Les cases non-vides de la table d'exécution sont alors indicées par le numéro d'instance de la fonction en train de s'exécuter à partir du temps de référence, comme illustré en figure 3.13.

Le nombre de pas de temps séparant deux exécutions d'une fonction dans le pipe-line est appelé « latence ». Lorsque deux instances de fonctions nécessitent un même étage du pipe-line en même temps, il y a collision. Les collisions se produisent pour des valeurs de latence particulières, qui sont appelées « latences interdites ».

E1	X					X	
E2		X		X		X	
E3			X		X		X

E1	Y		Y			Y
E2				Y		
E3		Y			Y	

FIG. 3.12 – Tables de réservation pour le pipe-line non linéaire de la figure 3.11.

E1	X_1			X_2		X_1			X_2	X_3					X_3	
E2		X_1		X_1	X_2	X_1	X_2		X_2		X_3		X_3		X_3	
E3			X_1		X_1	X_2	X_1	X_2		X_2		X_3		X_3		X_3

FIG. 3.13 – Table d'utilisation du pipe-line non linéaire de la figure 3.11 pour calculer la fonction X. L'exécution de X_2 est lancée trois cycles après celle de X_1, et celle de X_3 est lancée 6 cycles après celle de X_2.

Ainsi, pour la fonction X définie en figure 3.12, les latences 2, 4, et 5 sont-elles interdites.

E1	X_1				X_2	X_1				X_2	
E2		X_1		X_1		X_1X_2		X_2		X_2	
E3			X_1		X_1		X_1X_2		X_2		X_2

FIG. 3.14 – Collision entre X_1 et X_2 lorsque l'exécution de celle-ci est lancée 4 cycles après celle de X_1.

Les latences interdites se déduisent simplement des tables de réservation. Elles correspondent aux distances entre cases occupées appartenant aux mêmes lignes.

Une séquence de latences est une séquence de latences autorisées entre exécutions successives. Un cycle de latences est une séquence de latences répétant indéfiniment le même motif. Il peut y en avoir plusieurs, comme illustré en figures 3.15 et 3.16.

E1	X_1	X_2				X_1	X_2	X_3	X_4				X_3	X_4	X_5	X_6
E2		X_1	X_2	X_1	X_2	X_1	X_2		X_3	X_4	X_3	X_4	X_3	X_4		X_5
E3			X_1	X_2	X_1	X_2	X_1	X_2		X_3	X_4	X_3	X_4	X_3	X_4	

FIG. 3.15 – Table d'utilisation du pipe-line pour calculer la fonction X, avec le cycle de latences { 1, 6 }.

La latence moyenne d'un cycle de latences est obtenue en divisant la somme de toutes les latences du cycle par le nombre de latences contenues dans le cycle. Un cycle de latences constant est un cycle ne contenant qu'une unique valeur de latence. Du point de vue de l'efficacité, on souhaite déterminer le cycle donnant le débit le plus élevé, c'est-à-dire correspondant ła latence moyenne minimale (« *Minimal Average Latency* », ou MAL).

En examinant la table de réservation, il est possible de déterminer l'ensemble des latences autorisées, à partir des latences interdites. Si p est le nombre de colonnes

E1	X_1			X_2		X_1	X_3		X_2	X_4		X_3	X_5		X_4	X_6
E2		X_1		X_1	X_2	X_1	X_2	X_3	X_2	X_3	X_4	X_3	X_4	X_5	X_4	X_5
E3			X_1		X_1	X_2	X_1	X_2	X_3	X_2	X_3	X_4	X_3	X_4	X_5	X_4

FIG. 3.16 – Table d'utilisation du pipe-line pour calculer la fonction X, avec le cycle de latences { 3 }.

de la table de réservation, et m la plus grande latence interdite, avec $m < p$, on souhaite déterminer la plus petite latence autorisée a, dans le domaine $1 \leq a < m$.

L'ensemble décrivant les états autorisés et interdits peut être représenté sous la forme d'un vecteur de collisions, qui est un vecteur binaire $C = (c_m c_{m-1} \dots c_1)$ sur m bits, où la valeur de c_i est 1 si une latence de i provoque une collision et 0 si elle est autorisée.

À partir du vecteur de collisions, il est possible de créer un diagramme d'états spécifiant les transitions d'états autorisées entre exécutions successives dans le pipeline. Le vecteur de collisions C correspond à l'état initial du pipe-line au temps 1, et est donc appelé vecteur initial de collision. Si a est une latence autorisée à partir d'un état E, l'état obtenu à partir de E par lancement d'une exécution après une latence de a cycles s'obtient en décalant le vecteur de collision de E de a bits sur la droite, et en additionnant C au vecteur décalé. Quand la latence est supérieure à $m + 1$, toutes les transitions sont redirigées vers l'état initial, et la transition est notée « $(m + 1)+$ ». Ainsi, le diagramme d'état de la fonction X est représenté en figure 3.17.

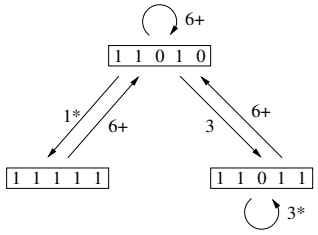


FIG. 3.17 – Diagramme d'état de la fonction X.

À partir du diagramme d'états, il est possible de déterminer les cycles permettant d'obtenir la MAL. Il existe une infinité de cycles obtensibles à partir d'un état donné du diagramme d'états. Cependant, seuls les cycles simples, c'est-à-dire les cycles ne passant qu'une fois au plus par un état donné, sont intéressants.

Certains cycles simples sont dits « gloutons ». Ce sont les cycles tels que les arêtes empruntées pour sortir de chaque état traversé du cycle ont les plus petites étiquettes possibles. Le cycle donnant la MAL est le cycle glouton dont la latence moyenne est inférieure à celle de tous les autres cycles gloutons.

3.4.3 Dépendances

Dans les pipe-line d'instructions des processeurs, les dépendances entre instructions constituent la plus grande source de bulles, qui peuvent parfois être évitées par une réorganisation du code, qui est réalisée par les compilateurs. À titre d'exemple, supposons que l'on veuille calculer l'expression de la figure 3.18, sous sa forme langage machine.

Si les instructions sont séquencées dans cet ordre sur une machine pipe-linée à cinq étages, le temps d'exécution est de $t = 14$ cycles, comme illustré en figure 3.19,

```
A = B * C + D * E;          mul     r1, [B], [C]
F = G * H + I * J;          mul     r2, [D], [E]
                            add     [A], r1, r2
                            mul     r1, [G], [H]
                            mul     r2, [I], [J]
                            add     [F], r1, r2
```

FIG. 3.18 – Fragment de langage machine (à droite) correspondant directement à l'expression à calculer (à gauche).

ce qui est deux fois plus rapide que sur une machine non pipe-linée, qui aurait nécessité $6 \times 5 = 30$ cycles élémentaires.

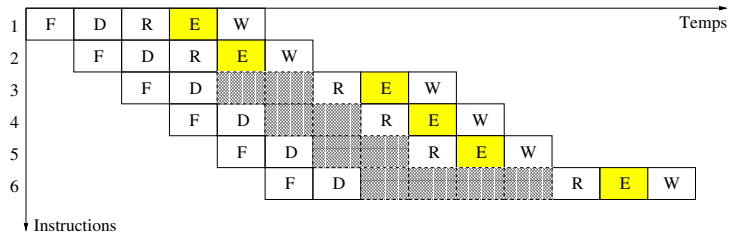


FIG. 3.19 – Exécution sur une machine pipe-linée à cinq étages du fragment de langage machine de la figure 3.18.

Si l'on réordonne les instructions en fonction du pipe-line selon le schéma de la figure 3.20, le nombre de cycles nécessaire tombe à 11, comme illsutré en figure 3.21. Cette diminution demande une augmentation du nombre de registres mis en œuvre, ce qui est une caractéristique des architectures RISC.

```
mul     r1, [B], [C]
mul     r2, [D], [E]
mul     r3, [G], [H]
mul     r4, [I], [J]
add     [A], r1, r2
add     [F], r3, r4
```

FIG. 3.20 – Fragment de langage machine réordonné, sémantiquement équivalent à celui de la figure 3.18

On distingue habituellement dans la littérature quatre types de dépendances, illustrées en figure 3.22. Certaines sont réelles, et reflètent le schéma d'exécution ; d'autres sont de fausses dépendances, qui résultent d'accidents dans la génération du code ou du manque d'informations sur le schéma d'exécution. Deux instructions ont une dépendance réelle de données si le résultat de la première est un opérande de la seconde. Deux instructions sont anti-dépendantes si la première utilise la valeur d'un opérande qui est modifié par la deuxième. Deux instructions ont une dépendance de résultat si toutes deux modifient le même opérande. Enfin, il existe une dépendance de contrôle entre un branchement et une instruction dont l'exécution est conditionnée par ce branchement.

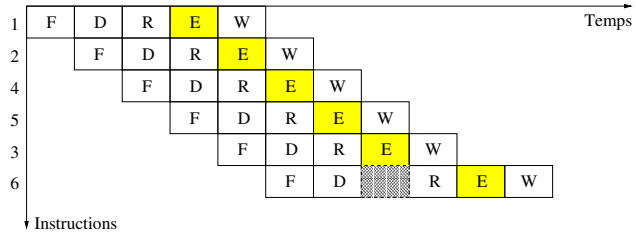


FIG. 3.21 – Exécution sur une machine pipe-linée à cinq étages du fragment de langage machine de la figure 3.20.

```
mov    r1, [A]
...
...
...
add    r3, r1, r2
```

Dépendance réelle

```
add    r2, r1, r4
...
...
...
mov    r1, [A]
```

Anti-dépendance

```
add    r1, r2, r3
...
...
...
mov    r1, [A]
```

Dépendance de résultat

```
bz     r4, L
...
div    r1, r1, r4
...
L:
```

Dépendance de contrôle

FIG. 3.22 – Exemples des quatre types de dépendances.

3.4.4 Branchements conditionnels

Une autre source très importante de bulles est l'existence des branchements, qui sont principalement les branchements conditionnels, dus aux boucles : dans le cas où le branchement est pris (« *branch taken* »), il faut vidanger le pipe-line, qui s'était automatiquement rempli avec les instructions situées directement après le branchement. De nombreuses techniques ont été développées afin de réduire l'impact des branchements sur le pipe-line d'instructions.

Déroulage de boucle

Le déroulage de boucle (« *loop unrolling* ») consiste en la recopie en plusieurs exemplaires du corps de la boucle, qui permet de supprimer les branchements intermédiaires et par là même d'éviter la vidange du pipe-line d'instructions à chaque tour de boucle.

Lorsque le nombre d'itérations initial n'est pas connu ou n'est pas un multiple de la valeur de déroulage, une copie de la boucle originale est ajoutée avant ou après la boucle déroulée, afin d'exécuter les itérations restantes, comme représenté en figure 3.23.

À titre d'exemple, la figure 3.24, page 29, présente un fragment de code machine Intel 80x86 correspondant au déroulage de la boucle d'un programme C de sommation des valeurs d'un tableau. Le nombre d'itérations étant inconnu, le com-

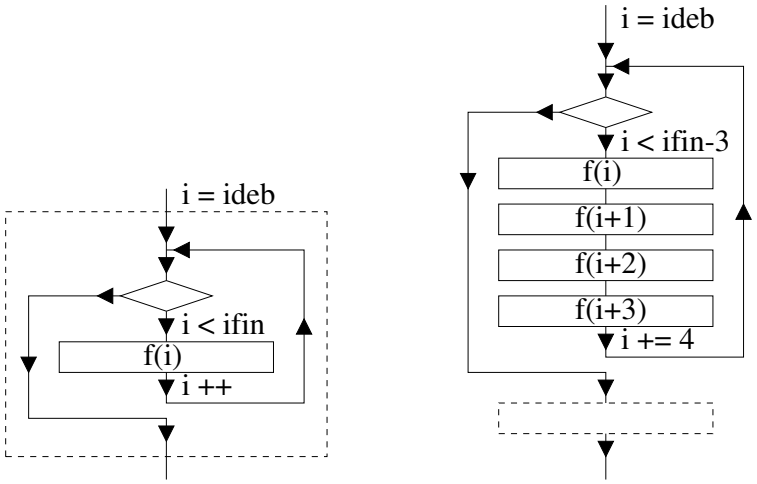


FIG. 3.23 – Déroulage d'ordre 4 d'un corps de boucle à nombre d'itérations inconnu.

pilateur prend tous les cas possibles en compte. Il déroule la boucle quatre fois, et fait précéder la portion déroulée d'un pré-traitement destiné à traiter les itérations résiduelles, dont le nombre est compris entre zéro et trois. Ce code fonctionne de la façon suivante :

- si le nombre d'itérations est inférieur ou égal à zéro, on évite la boucle ;
- si le nombre d'itérations, modulo 4, est 1, on va à la portion du code permettant de réaliser une itération, puis d'effectuer un test de terminaison, avant d'entrer dans la boucle déroulée ;
- si le nombre d'itérations, modulo 4, est 2, on va à la portion du code permettant de réaliser une itération, puis de continuer avec le code précédent, qui réalise la deuxième itération avant d'effectuer le test de terminaison et d'entrer dans la boucle déroulée ;
- si le nombre d'itérations, modulo 4, est 3 (cas restant), on charge directement la somme avec la première valeur du tableau (le compilateur a donc pris en compte que la valeur de la somme était nulle avant le début de la boucle), et on met le compteur à 1, avant d'entrer dans le code précédent, qui réalise les deux itérations suivantes.

Lorsque le nombre d'itérations est connu à la compilation, le compilateur déroule la boucle d'un ordre qui divise exactement le nombre d'itérations, ou bien fait précéder le corps de la boucle déroulée d'un nombre d'instances du code égal au modulo du nombre d'itérations par l'ordre de déroulage.

Le compilateur peut tirer parti du déroulage des boucles pour factoriser le code déroulé et calculer des optimisations inter-itérations, au prix d'une augmentation de la taille du code généré et du nombre de registres mis en œuvre. Cependant, cette technique peut diminuer l'efficacité d'autres techniques d'optimisation, comme le renommage dynamique de registres ou la prédiction de branchement, ce qui peut parfois conduire à une diminution de la concurrence des programmes [14, page 24] ;

Prédiction de branchement

La prédiction de branchement (« *branch prediction* ») a pour but d'augmenter la probabilité de suivre la bonne branche, et ainsi de ne pas rompre le flot du pipe-line.

```
int                 t[1000], n, s, i;

n = f ();                          // f() renvoie 1000 mais le
for (i = 0, s = 0; i < n; i ++)    // compilateur ne le sait pas
  s += t[i];
```

```
        ...
        xorl %edx,%edx             // Met la somme à zéro
        movl %eax,%esi             // Met le nombre dans esi
        movl %edx,%eax             // Met le compteur à zéro
        cmpl %esi,%edx             // Valeur de fin atteinte ?
        jge .L45                   // Si oui, rien à faire
        movl %esi,%ecx             // Copie le compteur dans ecx
        leal -4000(%ebp),%ebx      // Adresse du tableau dans ebx
        andl $3,%ecx               // Si compteur multiple de 4
        je .L47                    // Va à la boucle déroulée
        cmpl $1,%ecx               // Si valeur modulo 4 est 1
        jle .L59                   // Fait un tour et déroule
        cmpl $2,%ecx               // Si valeur modulo 4 est 2
        jle .L60                   // Fait deux tours et déroule
        movl -4000(%ebp),%edx      // Charge la première valeur
        movl $1,%eax               // Un tour fait, reste deux
.L60:
        addl (%ebx,%eax,4),%edx    // Fait un tour de boucle
        incl %eax                  // Incrémente le compteur
.L59:
        addl (%ebx,%eax,4),%edx    // Fait un tour de boucle
        incl %eax                  // Incrémente le compteur
        cmpl %esi,%eax             // Valeur de fin atteinte ?
        jge .L45                   // Termine si c'est le cas
        .align 4                   // Alignement pour cache
.L47:                              // Corps de boucle déroulée
        addl (%ebx,%eax,4),%edx    // Déroulage d'ordre 4
        addl 4(%ebx,%eax,4),%edx
        addl 8(%ebx,%eax,4),%edx
        addl 12(%ebx,%eax,4),%edx
        addl $4,%eax               // Ajoute 4 au compteur
        cmpl %esi,%eax             // Valeur de fin atteinte ?
        jl .L47                    // Reboucle si non atteinte
.L45:   ...
```

FIG. 3.24 – Exemple de déroulage de boucle à nombre d'itérations inconnu réalisé par `gcc-2.8.1` sur une architecture Intel 80x86. La boucle est déroulée quatre fois, et est précédée d'un code destiné à traiter les itérations résiduelles, dont le nombre est compris entre zéro et trois.

En l'absence de tout mécanisme de prédiction, les études statistiques effectuées sur de très nombreux codes indiquent que les meilleurs résultats sont obtenus lorsqu'on suppose que tous les branchements sont pris. Cependant, cette technique n'est pas efficace dans tous les cas.

Les techniques statiques utilisent uniquement l'information contenue dans le code pour effectuer la prédiction. Cette information peut être un bit de l'instruction de branchement, qui indique si le branchement est considéré comme pris ou non, et qui est positionné par le compilateur en fonction de son analyse du code (si la probabilité estimée que le branchement soit pris est supérieure à $\frac{1}{2}$ ou non). Elle peut aussi être constituée de l'adresse du branchement elle-même : si l'adresse de destination est inférieure à l'adresse courante (branchement remontant), le branchement est considéré comme pris, et si elle est supérieure (branchement descendant), non. Ces postulats correspondent à la manière dont les compilateurs codent les boucles, qui sont les plus grosses consommatrices de branchements conditionnels : test de `do...while` dans le cas remontant, et test de sortie de `for` ou de `while` dans le cas descendant.

Sur un ensemble représentatif de programmes, il a été montré [14, page 10] que la prédiction statique suivant le signe du déplacement (remontant ou descendant) donne le bon résultat dans 55% des cas, alors que supposer que le branchement est toujours pris est efficace dans 63% des cas, et qu'une pré-détermination par inspection du code à la compilation peut amener un taux de réussite moyen de 90%.

Un des problèmes de la prédiction statique de branchement est que la plupart des branchements conditionnels sont biaisés : la probabilité qu'ils soient pris évolue au cours du temps, même dans le cas de données quelconques (ce qui est d'ailleurs rarement le cas). Ainsi, pour le fragment de code suivant :

```
L1:  max = a[0];
L2:  for (i = 1; i < N; i ++)
L3:    if (a[i] > max)
L4:      max = a[i];
```

et en supposant que les `a[i]` sont aléatoires, la probabilité que `(a[i] > max)` pour un `i` donné est la probabilité que `a[i]` soit plus grand que les i valeurs précédentes déjà rencontrées (partant de 0), c'est-à-dire la plus grande de $i+1$ valeurs aléatoires, et est donc égale à $\frac{1}{i+1}$. La probabilité que le branchement du `if` soit pris diminue dans le temps. Les occurences successives du branchement sont donc corrélées dans le temps, et leur traitement efficace nécessite donc un mécanisme de prédiction utilisant un historique.

Les techniques dynamiques utilisent l'historique des décisions prises sur un ou plusieurs branchements précédents pour prédire la décision suivante. Pour des raisons d'efficacité, seul entre en compte actuellement l'historique récent (les deux ou trois derniers branchements).

Afin de prendre en compte l'historique des décisions précédentes, on peut conserver dans le cache d'instructions, dans le code de chaque instruction de branchement, un bit d'historique qui est mis à jour lorsque la condition est réévaluée (ce bit peut avoir été pré-initialisé par le compilateur, au cours d'une phase d'analyse du code ; notez également que le segment de code reste en lecture seule). On dispose alors d'un prédicteur dynamique à un bit d'historique.

De même, Lee et Smith [7] ont proposé d'utiliser un tampon de destination de branchement (« *Branch Target Buffer* », ou BTB) pour réaliser une prédiction de branchement. Ce dispositif est constitué comme un cache associatif adressé par l'adresse précédant l'adresse du branchement (ce qui permet d'anticiper la

prédiction, et d'augmenter l'effet d'historique selon la manière dont on arrive à l'instruction de branchement), et contenant les informations d'historique ainsi que l'adresse de destination prédite. L'information contenue sera mise à jour en fonction de la destination effective du branchement, lorsqu'elle sera connue. Le plus souvent, la prédiction est effectuée en fonction d'un diagramme d'état, similaire à celui de la figure 3.25, pour conserver l'historique des deux décisions précédentes.

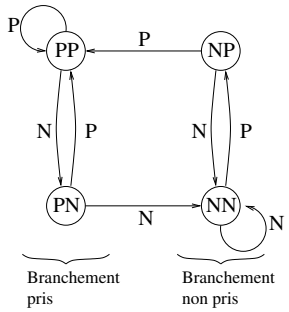


FIG. 3.25 – Diagramme d'état d'un prédicteur à deux bits.

Un cache de destination de branchement (« *Branch Target Cache* », ou BTC) peut être associé au BTB, et a pour but de contenir les quelques instructions situées à l'adresse de destination du branchement, afin de les charger rapidement dans le pipe-line d'instructions.

L'avantage de disposer d'au moins deux bits de prédiction apparaît clairement pour les boucles. Si l'on considère le fragment de code suivant :

```
L1:  for (i = 0; i < 100; i ++) {
L2:    for (j = 0; j < 4; j ++) {
         ...
```

exécuté en régime stationnaire, alors avec un prédicteur à un bit, la prédiction de L2 est fausse lors de la première itération de la boucle (on y reste, alors qu'on en était sorti la dernière fois), et de la dernière (on en sort, alors qu'on avait bouclé les fois précédentes). Avec un prédicteur à deux bits, en régime stationnaire, on considère toujours que la boucle est prise, puisqu'au moins deux itérations « pris » mettent le prédicteur dans l'état « PP », et que l'itération de sortie ne le met que dans l'état « PN ». Le branchement L2 n'est donc mal prédit que pour l'itération de sortie. Le prédicteur à deux bits est donc bien plus efficace pour traiter les boucles que le prédicteur à un bit, car il génère deux fois moins de mauvaises prédictions (seulement en sortie de boucle, mais pas en entrée).

Cependant, ce type d'historique, local à chaque branchement, peut ne pas être efficace. En effet, si l'on considère le fragment de code suivant :

```
L1:  if (cond1) action1;
L2:  if ((cond1) || (cond2)) action2;
```

et si l'on suppose que les deux conditions `cond1` et `cond2` sont aléatoires et non corrélées entre elles, alors la probabilité que L1 soit pris est de $\frac{1}{2}$ et la probabilité que L2 soit pris est de $1 - \frac{1}{2} \times \frac{1}{2} = \frac{3}{4}$.

Avec un prédicteur local à un bit (mais les résultats restent valables pour un plus grand nombre de bits), la probabilité qu'un branchement soit bien prédit est égale à la somme, pour chaque branche du test, de la probabilité que cette branche soit prise multipliée par la probabilité que cette branche soit bien prédite qui est, avec un prédicteur à un bit, la probabilité qu'elle ait également été prise la fois

précédente. C'est donc la somme des carrés des probabilités que chaque branche soit prise. La probabilité que L1 soit bien prédite est donc de $\left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2 = \frac{1}{2}$, et la probabilité que L2 soit bien prédite est de $\left(\frac{3}{4}\right)^2 + \left(\frac{1}{4}\right)^2 = \frac{5}{8}$. Avec un prédicteur de type *toujours pris*, la probabilité que L2 soit bien prédit est égale à la probabilité que L2 soit pris, c'est-à-dire de $\frac{3}{4} = \frac{6}{8}$. Dans cet exemple, la probabilité de bonne prédiction avec un prédicteur dynamique est *inférieure* à la probabilité de bonne prédiction avec un prédicteur statique, ce qui peut sembler étrange. Ceci est dû au fait que, L2 étant très fortement biaisé vers « pris », considérer qu'il sera non-pris à nouveau est très pénalisant. Il faut donc rendre le prédicteur plus stable par rapport aux cas exceptionnels, en utilisant par exemple un prédicteur à deux bits au lieu d'un seul, mais le problème de tels branchements corrélés demeure.

À titre d'exemple, si l'on pouvait prédire L2 en fonction du résultat de `cond1`, la probabilité que L2 soit pris si L1 est pris serait de 1, et la probabilité que L2 soit pris si L1 n'est pas pris serait de $\frac{1}{2}$. De même, la probabilité que L2 soit bien prédit si L1 est pris serait de 1, et la probabilité que L2 soit bien prédit si L1 est non-pris serait de $\left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2 = \frac{1}{2}$. La probabilité que L2 soit bien prédit connaîssant L1 serait donc de $\frac{1}{2} \times 1 + \frac{1}{2} \times \frac{1}{2} = \frac{3}{4}$. L'utilisation d'un historique global, prenant en compte les résultats des autres branchements, peut donc être plus efficace que de ne considérer que l'historique local.

Pour cela, on peut utiliser un prédicteur à deux niveaux [8]. Le premier est constitué d'une fonction de hachage prenant comme arguments l'adresse du branchement et l'historique global, sur b bits, des décisions de branchement les plus récentes, codé comme un registre à décalage de b bits (« 1 » pour « pris » et « 0 » pour « non pris »). Le deuxième est constitué d'un tableau de prédicteurs à deux bits indexé par cette fonction de hachage. Le schéma d'un tel mécanisme est donné en figure 3.26.

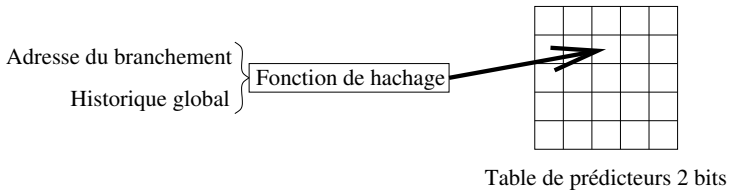


FIG. 3.26 – Schéma d'un prédicteur à historique global à deux niveaux. L'adresse du branchement et la chaîne de bits de l'historique global courant servent d'index dans une table de prédicteurs à deux bits.

Le prédicteur à historique global permet de traiter plus efficacement les branchements corrélés. Ainsi, dans l'exemple précédent, lorsque le branchement L1 est pris, le registre à décalage global vaut $xxx1$ lors de la prédiction de L2, et adresse des prédicteurs à deux bits différents de ceux utilisés lorsque le registre à décalage vaut $xxx0$. Si l'on ne considère pas les risques d'interférences dus à la fonction de hachage, la probabilité de bonne prédiction est donc celle calculée précédemment, c'est-à-dire $\frac{3}{4}$;

On peut également remarquer que les mécanismes de prédiction de branchement et de branchement retardé sont antagonistes : une prédiction de branchement parfaitement efficace rend inutile les branchements retardés, et réciproquement. Cependant, comme il est impossible de prédire totalement les branchements, les architectures récentes implémentent souvent au moins un niveau de branchement retardé, pour amortir la pénalité des branchements mal prédits ;

Exécution spéculative

L'exécution spéculative (« *speculative execution* ») utilise les performances des architectures superscalaires pour exécuter concurrement les deux flots d'instructions associés à chacune des branches, dont les résultats sont temporairement stockés dans des registres fantômes (« *shadow registers* »). Dès que la condition de branchement est effectivement évaluée, les registres fantômes correspondant à la branche prise sont renommés pour devenir les registres visibles par l'utilisateur ; ceux correspondant à l'autre branche sont simplement libérés, pour resservir lors d'un prochain branchement conditionnel.

L'exécution spéculative nécessite des mécanismes très évolués pour retarder les effets de bord tels que les écritures en mémoire ou les exceptions tant que la branche qui les génère n'est pas encore définitivement acceptée ou rejetée. En pratique, les processeurs actuels n'autorisent qu'au plus quatre niveaux simultanés d'exécution spéculative. En fait, il est plus économique d'implémenter des mécanismes efficaces de prédicition de branchement que d'augmenter la profondeur d'exécution spéculative. Les deux mécansimes sont antagonistes, et disposer d'un mécanisme parfaitement efficace pour réaliser l'une rend inutile la présence de l'autre [14, page 27]. Une approche proposée consiste à utiliser l'exécution spéculative si elle est possible, et ensuite la prédiction de branchement lorsque la profondeur maximale a été atteinte.

3.5 Parallélisme d'instructions

3.5.1 Superscalarité

Les processeurs classiques, dits « scalaires » (par opposition aux processeurs vectoriels, voir section 3.7), n'exécutent au plus qu'une instruction par cycle : en l'absence de dépendances, une seule instruction est introduite à chaque cycle dans le pipe-line d'instructions.

Les processeurs superscalaires, eux, disposent de plusieurs pipe-lines d'instructions qui leur permettent de traiter plusieurs instructions par cycle, et ainsi d'exploiter le parallélisme existant entre instructions consécutives. Le nombre de pipe-lines d'instructions d'un processeur superscalaire est appelé « degré » du processeur.

Typiquement, sur un code sans déroulement de boucles, le nombre d'instructions consécutives sans dépendances est proche de deux. De fait, les processeurs superscalaires actuels ont un degré compris entre trois et cinq.

Il existe différents modèles d'exécution superscalaires, que l'on peut classer suivant la façon dont l'allocation et le réordonnancement des instructions sont effectués. On entend par allocation la technique qui permet d'affecter une instruction à une unité de calcul, et par réordonnancement la technique qui permet d'exécuter une instruction avant ou après une autre.

Ces deux techniques peuvent être traitées soit de façon statique au niveau du compilateur (celui-ci devra alors, au moyen des techniques de déroulage de boucle, d'entrelacement de code, et de prédiction de branchement, fournir un code dont les instructions consécutives soient les plus indépendantes possible), soit de façon dynamique au niveau du processeur.

Parmi les processeurs superscalaires à allocation dynamique, on trouve tous les processeurs récents, tels le Pentium IV, l'Alpha 21164, le Power, etc. L'avantage de l'allocation dynamique est que le code exécutable d'un tel processeur ne dépend pas du nombre d'unités fonctionnelles qu'il possède, et est donc identique à celui

des architectures traditionnelles, garantissant la compatibilité au sein d'une même famille, comme par exemple la famille Power d'IBM.

3.5.2 VLIW (« *Very Long Instruction Word* »)

Les processeurs VLIW sont les représentants des processeurs superscalaires a allocation statique. Les instructions des processeurs VLIW, de grande taille (jusqu'à 1024 bits), permettent de coder dans leurs différents champs les opérandes de toutes les unités fonctionnelles du processeur, qui peuvent toutes travailler en parallèle. Les programmes écrits au moyen d'instructions courtes doivent être réarrangés pour former des instructions VLIW. Ceci doit être fait par le compilateur, qui doit mettre en œuvre des stratégies très élaborées.

Les différences entre les processeurs VLIW et les processeurs à allocation dynamique sont que :

- la densité du code est souvent meilleure pour les architectures à allocation dynamique, de nombreuses unités fonctionnelles du processeur VLIW étant inhibées en cas de dépendances ;
- le décodage des instructions VLIW est très simple, puisqu'il ne concerne que les opérandes ;
- les processeurs VLIW n'ont pas besoin de circuiterie de gestion des dépendances et de reséquencement des instructions, puisque cette tâche est dévolue au compilateur.

3.5.3 LIW (« *Long Instruction Word* »)

Actuellement, on s'oriente vers une voie hybride, avec des processeurs de type LIW. Ainsi, la nouvelle architecture Itanium développée conjointement par Intel et HP est basée sur une architecture LIW où une instruction longue (ou ¡¡ paquet ¿¿) de 64 bits code trois instructions courtes qui peuvent être exécutées concurremment. Le Crusoe de Transmeta est un autre exemple de ce modèle. Ici encore, on supprime la circuiterie de réordonnancement et de gestion des dépendances, à charge pour le compilateur de réarranger le code en paquets de trois instructions courtes.

L'avantage des architectures LIW par rapport aux VLIW est qu'elles garantissent une compatibilité ascendante lorsque le nombre d'unités fonctionnelles augmente : si le nouveau processeur de la famille possède deux unités fonctionnelles d'addition au lieu d'une seule, les instructions LIW du plus vieux processeur seront cependant toujours légales sur le nouveau. En revanche, les instructions LIW du nouveau processeur pourront contenir deux instructions d'addition par paquet, ce qui n'est pas légal pour le vieux processeur.

L'ensemble des techniques permettant de mettre en œuvre le parallélisme explicite au niveau des instructions (VLIW et LIW), est désignée en anglais par l'acronyme EPIC, pour « *Explicit Parallel Instruction Computing* ».

3.6 Application à la programmation

Les techniques matérielles et logicielles d'optimisation décrites ci-dessus ont un impact majeur sur la performance des programmes, selon que ceux-ci en tirent parti ou non.

À titre d'exemple, considérons le problème suivant : on dispose d'un tableau d'entiers de très grande taille, ne contenant que des 0, des 1, et des 2, et l'on souhaite connaître le nombre de 0, de 1, et de 2 contenus dans le tableau. Pour ce faire, on peut écrire au moins trois programmes différents :

– le premier programme (P1), présenté en figure 3.27, est conçu pour tirer parti de l'exécution spéculative : à chaque tour de boucle, un ou deux tests sont effectués, qui conditionnent l'incrémentation de deux compteurs, la valeur du troisième étant déduite à la fin par soustraction ;

```
c0 = c1 = 0;
for (i = 0; i < n; i ++) {
  if (t[i] == 0)
    c0 ++
  else if (t[i] == 1)
    c1 ++;
}
c2 = n - c0 - c1;
```

FIG. 3.27 – Programme de comptage des 0, 1, et 2 d'un tableau, basé sur l'exécution spéculative.

– le deuxième programme (P2), présenté en figure 3.28, est basé sur l'indexation d'un tableau de compteurs, dont le contenu est incrémenté en fonction des valeurs du tableau initial. Ce programme ne requiert aucun branchement conditionnel au sein du corps de boucle, mais nécessite obligatoirement des accès mémoire (même si elles ils se feront à priori que dans le cache de premier niveau), puisque les cases à incrémenter ne peuvent pas être connues à la compilation, et donc ne peuvent être affectées à des registres.

```
c[0] = c[1] = c[2] = 0;
for (i = 0; i < n; i ++)
  c[t[i]] ++;
```

FIG. 3.28 – Programme de comptage des 0, 1, et 2 d'un tableau, basé sur l'indexation d'un tableau de compteurs.

– le troisième programme (P3), présenté en figure 3.29, diffère du précédent en ce que l'on remplace l'accès au tableau indexé de compteurs par une mise à jour conjointe de deux compteurs par des valeurs obtenues par masquage binaire des valeurs du tableau. Cette version ne contient aucun test interne, et peut être compilée uniquement avec des registres, mais nécessite plus d'opérations arithmétiques par tour de boucle.

```
c1 = c2 = 0;
for (i = 0; i < n; i ++) {
  c1 += (t[i] & 1);
  c2 += (t[i] & 2);
}
c2 >>= 1;
c0   = n - c1 - c2;
```

FIG. 3.29 – Programme de comptage des 0, 1, et 2 d'un tableau, basé sur la mise à jour conjointe de deux compteurs par des valeurs masquées.

Le comportement des deuxième et troisième programme ne dépend pas de la distribution des données contenues dans le tableau. En revanche, pour le premier, l'historique de prédiction de branchement au sein du corps de boucle dépendra fortement des proportions relatives de 0, 1, et 2, ainsi que de leur placement dans le

tableau (de grandes plages de valeurs identiques seront préférables à une répartition aléatoire des valeurs).

Processeur	`t[i] = { 0, 1, 2 }`			`t[i] = 0`		
	P1	P2	P3	P1	P2	P3
Celeron 300A 450Mhz	9860	6650	7160	5730	6650	7170
Pentium III 700Mhz	9070	6660	7290	6090	6670	7280
Duron 700Mhz	7860	5510	5830	5340	5490	5830
Athlon XP 1.4Ghz	2850	2030	2110	2000	1990	2110
PowerPC 60Mhz	13550	12220	11810	12280	12270	11810
PowerWhwk2+ 375Mhz	5640	3960	2160	3480	6190	2190
U.SPARC II 450Mhz	10330	8880	8050	8800	9670	7940
U.SPARC IIe 400Mhz	11650	9960	8710	9670	10770	9170
U.SPARC IIe 500Mhz	8030	7290	6370	6610	8080	6360
Alpha EV67 667Mhz	7678	6002	4555	4728	6003	4553
HP PA-8700 750MHz	4480	3540	3310	4370	4870	3320

TAB. 3.1 – Temps d'exécution des programmes présentés en figures 3.27, 3.28, et 3.29, mesurés sur différentes architectures, en mode 32 bits, pour un tableau rempli de façon aléatoire ou constitué uniquement de zéros.

Le tableau 3.1 donne les temps d'exécution des trois versions sur de nombreuses architectures[1], pour un tableau de données rempli soit de façon aléatoire, soit uniquement avec des zéros. Il n'est pas très pertinent de comparer quantitativement les temps obtenus pour deux architectures différentes ; seule l'étude qualitative, ligne par ligne, nous intéresse ici. On peut en dégager les renseignements suivants :

- quelle que soit la distribution des valeurs du tableau, la version P3 donne des temps identiques sur toutes les architectures, et est la plus efficace sur les architectures Power, Alpha, et HP PA-RISC, fortement superscalaires (pour plus d'informations sur la superscalarité, voir la section 3.5.1) ;
- quelle que soit la distribution des valeurs du tableau, la version P2 donne elle aussi des temps équivalents pour toutes les architectures, sauf sur le Power WinterHawk2+, où les temps explosent lorsque le tableau n'est rempli qu'avec des zéros. Ceci est extrêmement surprenant, vu le niveau « haut de gamme » du processeur, et doit provenir de conflits d'accès au cache lors de demandes de lectures-écritures multiples à la même case mémoire. Le HP PA semble également sensible, bien que dans une moindre mesure, au même phénomène ;
- la version P2 est la plus performante sur les architectures Intel lorsque la prédiction de branchement est inopérante, car celles-ci possèdent un cache de premier niveau peu coûteux et ne sont pas très superscalaires, rendant la version P3 moins efficace ;
- la version P1 est en revanche la plus efficace, sur les architectures Intel uniquement, lorsque la prédiction de branchement donne des résultats optimaux. Intel, limité en superscalarité par son jeu d'instructions CISC, a en revanche fait des efforts importants pour disposer d'une unité d'exécution spéculative performante, qui réduit d'un tiers le temps d'exécution. Celle du HP PA semble aussi très performante, puisque le temps de la version P1 dépend relativement peu de la distribution des données.

Ces résultats montrent bien qu'il est important d'écrire des tests biaisés le plus possible, et que, sur les architectures fortement superscalaires, il est préférable de remplacer des tests potentiellement générateurs de ruptures de pipe-line par quelques

[1] Ces résultats ont été fournis par Baptiste Malguy, ENSEIRB Info PRCD, promotion 2001, et complétés par Christophe Giaume, promotion 2003.

opérations supplémentaires.

Certains processeurs disposent d'instructions à prédicats (« *predicated instructions* »), qui ne sont effectivement exécutées que si une certaine condition (valeur de registre) est vérifiée, et permettent d'éviter de recourir à des instructions de branchement pour exécuter ou non des blocs conditionnels de petite taille. Cependant, pour traiter au plus vite ces instructions sans attendre le résultat du calcul de la condition, il faut disposer de capacités importantes d'exécution spéculative, dont la complexité limite les gains en performance [1].

3.7 Processeurs vectoriels

De nombreux problèmes scientifiques sont intrinsèquement vectoriels, c'est-à-dire qu'ils opèrent sur des vecteurs unidimensionnels de données. Pour les traiter efficacement, certaines architectures (dont les CRAY ont été les plus célèbres représentants) disposent d'instructions vectorielles, qui s'appliquent à des tableaux unidimensionnels de données de même nature (le plus souvent des nombres en virgule flottante), élément après élément.

Quand l'unité de contrôle décode et exécute une instruction vectorielle, le premier élément du (ou des) vecteur(s) impliqué(s) est soumis à l'unité de traitement considérée. Après un certain nombre de cycles, le second élément est soumis, et ainsi de suite, jusqu'à ce que toutes les opérandes du (ou des) vecteur(s) aient été traités.

Cette technique permet de remplacer une séquence d'instructions scalaires par une instruction vectorielle qui ne sera décodée qu'une seule fois, mais surtout permet d'utiliser à plein les pipe-lines des unités de traitement en virgule flottante (addition, multiplication, inverse) auxquelles les instructions vectorielles sont le plus souvent associées.

Afin d'accélérer encore plus les calculs, il est possible de chaîner plusieurs opérations vectorielles entre elles (« *pipeline chaining* »). Le résultat d'une unité de traitement pipe-linée est alors soumis en entrée d'une autre, sans attendre que la première opération vectorielle ait terminé. Ainsi, le chaînage de deux opérations vectorielles ne coûte que le temps d'initilisation du pipe-line de la deuxième unité de traitement, en plus du coût d'exécution de la première instruction vectorielle. Ceci revient, du point de vue de l'efficacité, à augmenter la profondeur du pipe-line de traitement.

À titre d'exemple, on peut étudier la manière dont on calcule l'inverse de nombres flottants sur le CRAY 1. Cette machine ancienne (1976), dont une version simplifiée de l'architecture fonctionnelle est présentée en figure 3.30, est basée sur une architecture vectorielle pipe-linée, avec un temps de cycle $\tau = 12,5$ ns, une profondeur de pipe-line de 6 cycles pour l'addition, 7 cycles pour la multiplication, et 14 cycles pour l'approximation réciproque (« *reciprocal approximation* », ou RA), qui permet d'obtenir une approximation à deux bits près de l'inverse d'un nombre flottant. Le surcoût d'utilisation des registres vectoriels est d'un cycle pour la lecture et d'un cycle pour l'écriture, et le surcoût de chaînage entre deux unités est de deux cycles. Sur le CRAY 1, on calcule l'inverse $x = \frac{1}{a}$ d'un nombre a par la méthode de Newton, c'est-à-dire la recherche du zéro de la fonction $f(x) = a - x^{-1}$, avec $f'(x) = x^{-2}$, en itérant :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = (2 - a \cdot x_n)x_n \ .$$

Comme la fonction RA donne une bonne approximation de $\frac{1}{a}$, une seule itération de la méthode de Newton est nécessaire pour déterminer les deux derniers bits de

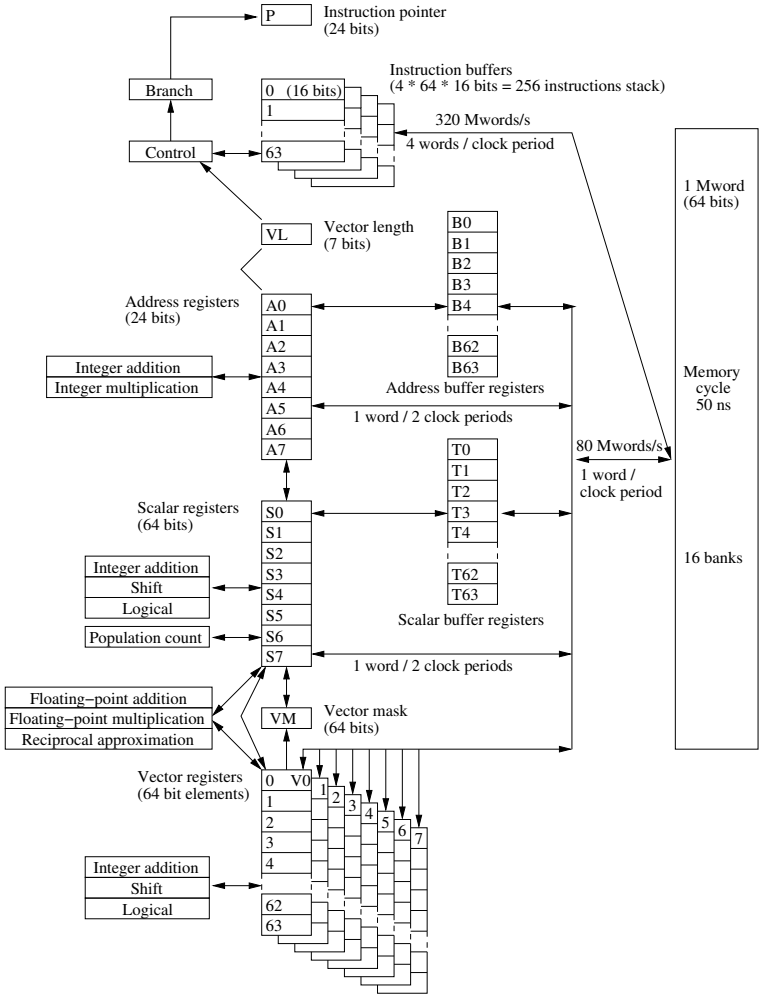


FIG. 3.30 – Schéma simplifié de l'architecture fonctionnelle du CRAY 1. Extrait de [4, page 73].

la mantisse de $\frac{1}{a}$, et donc

$$x = (2 - a \cdot \mathrm{RA}(a))\mathrm{RA}(a) \ .$$

Si l'on effectue le calcul S6 = S1 / S2 au moyen des registres scalaires, selon la séquence de la figure 3.31, on obtient une puissance de $\frac{1}{29\tau} \approx 2,76$ Mflop/s. Si l'on réalise maintenant la division de façon vectorielle, avec des vecteurs de taille $l \geq 9$, selon la séquence de la figure 3.32, illustrée par la figure 3.33, on obtient une puissance de $\frac{l}{(24+3\,l)\tau} \approx 23,7$ Mflop/s pour des vecteurs de taille $l = 64$. Remarquons que, dans l'algorithme vectoriel, on a inversé l'ordre des deux instructions du milieu, car sinon les deux instructions chaînées accèderaient en même temps en lecture au vecteur V2, ce qui n'est pas possible sur le CRAY 1.

Dans le cas de la multiplication vectorielle V3 = V1 * V2, le temps de calcul pour un vecteur de taille l est $1+7+1+(l-1) = 8+l$, ce qui donne une puissance de 70 Mflop/s pour des vecteurs de taille $l = 64$.

Il est à noter que la limitation d'accès à la mémoire a fait l'objet d'améliorations très précoces. Dès la génération X-MP, les CRAY se sont vus dotés de trois pipe-line d'accès à la mémoire : deux pour la lecture, et un pour l'écriture. Ainsi,

Instruction	Unité	Début	Fin	
S3 = RA(S2)	RA	0	14	
S4 = (2 - S3 * S2)	mul (!)	14	21	
S5 = S1 * S3	mul	15	22	Pipe-line
S6 = S4 * S5	mul	22	29	

FIG. 3.31 – Séquencement des instructions scalaires nécessaires au calcul de S6 = S1 /S2.

Instruction	Unité	Début	Fin	
V3 = RA(V2)	RA	0	16 + (l - 1)	
V5 = V1 * V3	mul	1 + 14 + 1	25 + (l - 1)	Chaînage
V4 = (2 - V3 * V2)	mul	17 + (l - 1)	26 + 2 (l - 1)	Pipe-line
V6 = V4 * V5	mul	17 + 1 + 2 (l - 1)	27 + 3 (l - 1)	Pipe-line

FIG. 3.32 – Séquencement des instructions vectorielles nécessaires au calcul de V6 = V1 /V2.

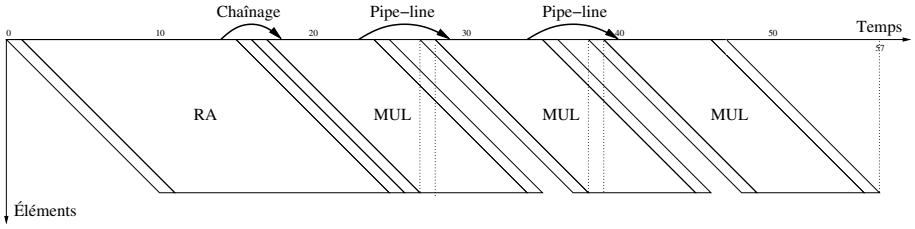


FIG. 3.33 – Séquencement des calculs vectoriels de la figure 3.32.

l'exécution de la fonction BLAS-1 SAXPY, qui réalise l'opération $Y = aX + Y$ sur deux vecteurs X et Y, avec a scalaire, s'effectue-t-elle au moyen de trois chaînes sur le CRAY 1, mais seulement avec une sur le CRAY X-MP, comme illustré en figure 3.34.

La vectorisation automatique a été un sujet de recherche très actif au cours de la dernière décennie, qui a permis d'offrir aux utilisateurs des compilateurs vectoriseurs efficaces. Ceux-ci utilisent plusieurs techniques :

- le déroulage de boucles (« *loop unrolling* »), pour transformer les opérations scalaires de plusieurs itérations en opérations vectorielles plus efficaces ;
- la segmentation de tableaux (« *strip-mining* »), pour convertir des opérations logiques sur des vecteurs de grandes tailles en instructions opérant sur les registres vectoriels de la machine, qui sont de taille fixe ;
- la transformtion des boucles (« *loop transformation* »), qui permet de modifier l'ordre dans lequel l'espace des itérations d'un nid de boucle est parcouru afin de maximiser la localité des données dans les boucles les plus internes, comme illustré en figure 3.35.

3.8 Évaluation des performances des processeurs

La fréquence d'horloge est un paramètre déterminant de la puissance des processeurs. Cependant, de nombreux autres critères architecturaux doivent être pris en compte, sans parler de l'environnement du processeur (la hiérarchie mémoire, en particulier). De fait, l'augmentation de la fréquence d'horloge n'est significative par

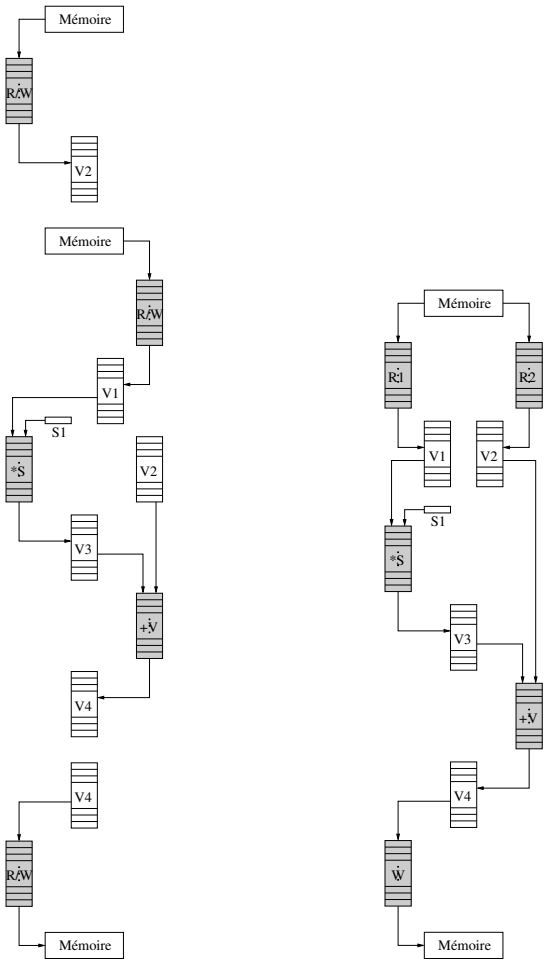


FIG. 3.34 – Chaînages des instructions vectorielles nécessaires à la réalisation de l'opération BLAS-1 SAXPY, sur le CRAY 1 et sur le CRAY X-MP.

elle même qu'au sein d'une famille de processeurs donnée.

Pour comparer les performances de deux machines différentes (architecture, type de jeu d'instructions, etc.), il faut décomposer le temps total d'exécution des programmes en leurs constituants. Le temps mis pour exécuter un programme donné est le produit du nombre de cycles nécessaires par le temps de cycle :

$$T = c\,\tau \ .$$

Le nombre de cycles peut être quant à lui réécrit comme le nombre d'instructions exécutées multiplié par le nombre moyen de cycles par instructions :

$$T = i\,\left(\frac{c}{i}\right)\,\tau \ .$$

Le nombre d'instructions dépend de facteurs logiciels (algorithme choisi, compilateur), mais aussi du type de jeu d'instructions utilisé (CISC ou RISC). Le fait d'avoir un jeu d'instructions plus complexe n'accélère pas forcément l'exécution, car ces instructions nécessitent plus de cycles pour s'exécuter.

```
      DO 20 I = 2, N                    DO 20 J = 2, N
        DO 10 J = 2, I                    DO 10 I = 2, J
          A(I, J) = A(I, J - 1)             A(I, J) = A(I, J - 1)
     *            + A(I - 1, J)        *            + A(I - 1, J)
10      CONTINUE                  10      CONTINUE
20    CONTINUE                    20    CONTINUE
```

FIG. 3.35 – Example de transformation de boucles. L'échange des boucles en `I` et `J` permet d'accéder au tableau `A` par colonnes, qui est l'ordre naturel en FORTRAN, et donc de le charger en mémoire par des instructions vectorielles.

Le nombre moyen de cycles par instruction ne mesure pas seulement la complexité du jeu d'instructions, et dépend également de critères architecturaux : superscalarité, pipe-lines, etc.

Le temps de cycle du processeur dépend de la technologie et des matériaux utilisés, mais aussi de l'architecture du processeur. Un jeu d'instructions petit et une circuiterie simple nécessitent une surface de silicium moins importante, d'où un temps de parcours de l'information plus petit.

La mesure effective des performances des machines s'effectue en mesurant le temps d'exécution de programmes de complexité connue : calcul matriciel (LinPACK 100×100 ou 1000×1000), etc. Cette méthode seule permet de prendre en compte l'intégralité des phénomènes mis en jeu, tant du point de vue matériel que logiciel. Elle n'est cependant valide que pour une application donnée (si celle-ci est plutôt vectorielle ou superscalaire, on pourra avoir des performances très variables sur des architectures différentes).

Chapitre 4

Architecture des mémoires

4.1 Hiérarchie mémoire

L'efficacité des processeurs dépend très fortement du temps d'accès aux informations stockées en mémoire. Cependant, pour des raisons techniques (vitesse de la lumière) autant que financières, il n'est pas possible de réaliser une mémoire de grande capacité ayant un temps d'accès compatible avec les fréquences de cadencement des processeurs actuels (4 GHz, soit 250 ps de latence).

Dans tout programme, il est possible de mettre en évidence un phénomène de localité des accès mémoire, exprimé en termes de :

- localité temporelle : plus une zone mémoire a été accédée récemment, et plus sa probabilité de ré-acces est élevée ;
- localité spatiale : plus une zone mémoire est proche de la dernière zone mémoire accédée, et plus la probabilité qu'elle soit à son tour accédée est importante.

Ceci est vrai :

- pour les instructions. C'est le cas du déroulement normal d'un programme séquentiel sans branchements, dont on tire également partie dans les pipelines d'instructions ;
- pour les données. C'est le cas lors des mises à jour de variables, de l'accès à des données structurées, du parcours séquentiel de tableaux, etc.

On s'appuie sur ce principe pour mettre en place une hiérarchie de la mémoire, entre mémoires rapides de faible capacité et mémoires de grande capacité aux temps d'accès plus longs, afin que les informations les plus fréquemment utilisées soient disponibles le plus rapidement possible ; cette structure pyramidale est illustrée en figure 4.1.
Le transfert des informations entre zones lentes et zones rapides s'effectue soit de façon logicielle (registres, zones cache utilisateur, va-et-vient (« *swap* ») disque), soit de façon matérielle (cache).

4.2 Registres

Les registres sont des mémoires très rapides (temps d'accès de l'ordre des cent pico-secondes), situés le plus souvent sur le processeur lui-même.

Afin de mettre en œuvre efficacement les techniques pipe-line et superscalaires, et de réduire le nombre d'accès à la mémoire, les processeurs actuels possèdent de plus en plus de registres. Les architectures les plus courantes en ont de 32 à 192, mais on trouve des processeurs en ayant jusqu'à 2048. Cependant, le plus souvent,

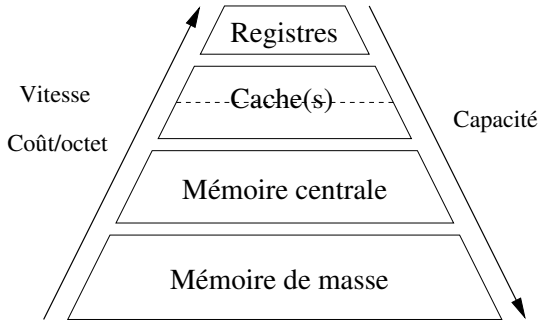


FIG. 4.1 – Hiérarchie mémoire.

tous ne sont pas simultanément accessibles à l'utilisateur.

Pour éviter que les changements de contexte liés aux appels de fonctions ne génèrent des accès mémoire coûteux, certains processeurs disposent d'un mécanisme de « fenêtres de registres », introduit dans le processeur RISC I de l'Université de Berkeley en 1972, et repris par les processeurs SPARC. Ceux-ci disposent de 32 registres visibles pour exécuter les programmes. Huit sont des registres globaux, communs à tous les contextes, et les 24 autres sont des registres fenêtrés associés à chaque procédure, comme illustré en figure 4.2.

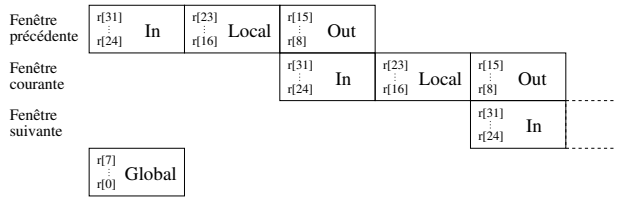


FIG. 4.2 – Fenêtres de registres du processeur SPARC.

Chaque fenêtre est divisée en trois sections :
- les *In*s : paramètres passés par la procédure appelante ;
- les *Local*s : accessibles seulement à la procédure courante ;
- les *Out*s : paramètres passés aux procédures appelées, pouvant également servir de variables locales.

Les 136 registres du processeur SPARC sont donc organisés en huit fenêtres glissantes de 24 registres chacune, auxquelles il faut ajouter huit registres globaux (comprenant les pointeurs d'instructions et de pile), comme illustré en figure 4.3. Un indicateur de fenêtre courante et des bits d'invalidité permettent de déterminer quelles fenêtres sont utilisables ou nécessitent une sauvegarde de contexte.
En théorie, on peut parcourir sept niveaux de récursion sans effectuer d'accès mémoire, ce qui permet un gain considérable en efficacité pour les programmes fortement récursifs. En pratique, l'intégralité de la fenêtre de registres doit être sauvegardée entre chaque changement de contexte de processus, ce qui est très pénalisant dans un environnement multi-processus en temps partagé.

Cette idée a néanmoins été reprise et étendue dans l'architecture Itanium, qui possède 128 registres visibles, organisés en 32 registres globaux, numérotés de 0 à 31, visibles de tous les contextes, et 96 registres empilés, numérotés de 32 à 127. Chaque fonction déclare lors de la création de son contexte le nombre de registres *In*

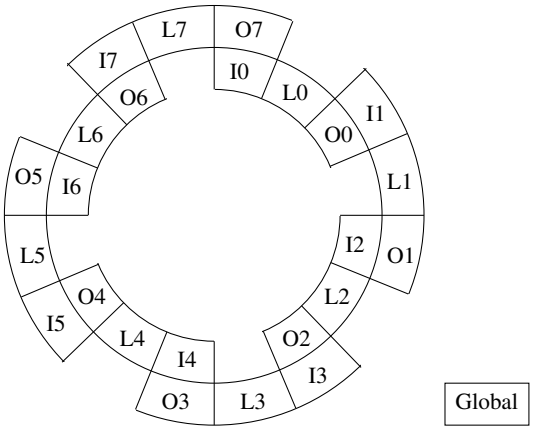


FIG. 4.3 – Recouvrement des fenêtres de registres glissantes du processeur SPARC.

qu'elle reconnait, ainsi que le nombre total de registres disponibles dont elle a besoin (*In*, *Local* et *Out*), nécessairement inférieur ou égal à 96. Un mécanisme matériel spécifique, appelé *Register Stack Engine*, est chargé de sauvegarder (« *spill* ») dans la pile mémoire, appelée zone de *backing store*, le contenu des registres physiques des contextes appelants les plus anciens et devant servir comme registres locaux du contexte courant, et à les restaurer (« *fill* ») lorsqu'on retournera à ces contextes, donnant ainsi l'illusion d'une pile de registres de taille infinie. Une étude menée par des ingénieurs d'Intel a montré que le mécanisme de pile de registres pouvait conduire à un gain de performance de l'ordre de 10% par rapport à un processeur n'en disposant pas [12].

4.3 Mémoire cache

La mémoire cache est une mémoire rapide faisant tampon entre le processeur et la mémoire centrale. Selon leur localisation, on distingue :

- les caches internes, situés sur le processeur, d'une vitesse presque équivalente à celle des registres, et de taille comprise entre 1 et 32 ko ;
- les caches externes, extérieurs au processeur, plus lents mais de capacité plus importante, pouvant aller jusqu'à 1 Mo.

Ces deux types de cache peuvent coexister au sein de la même architecture ; le cache interne est alors appelé cache de premier niveau, et le cache externe, cache de second niveau. Dans certaines architectures, on trouve même un troisième niveau de cache.

4.3.1 Mécanismes d'accès

Quand le processeur souhaite lire une donnée à partir de la mémoire, il génère l'adresse correspondante, et émet une requête sur le bus, qui est interceptée par le cache. Si la donnée est présente dans le cache (on parle alors de « *cache hit* »), elle est directement envoyée au processeur. Sinon, en cas de défaut de cache (« *cache miss* »), la requête est transmise à la mémoire centrale. Lorsque la donnée est fournie par la mémoire, une copie est conservée dans le cache (en cas d'accès futur), qui doit libérer la place nécessaire à son stockage.

Afin de tirer parti du phénomène de localité, les transferts entre le cache et la mémoire s'effectuent par blocs (ou « lignes », « *lines* »). Pour des raisons d'efficacité (liées au temps de transfert ainsi qu'à la limitation de la place dans le cache), la

taille de ces blocs est cependant limitée à quelques octets (entre 16 et 64).

Les lignes de cache sont chargées sur demande. Lorsqu'une donnée à lire n'est pas présente dans le cache, celui-ci demande à la mémoire de lui transmettre la ligne à laquelle appartient la donnée demandée, et libère l'espace nécessaire à son stockage.

Plusieurs optimisations permettent d'accélérer les lectures à partir de la mémoire. Habituellement, en l'absence de cache, les mots sont lus individuellement à partir de la mémoire : l'adresse du mot à lire est placée sur le bus, et la mémoire renvoie alors son contenu, au bout d'un certain temps. Pour optimiser le chargement des lignes de cache à partir de la mémoire (opération dite de « *cache line fill* »), on peut effectuer une lecture en mode « rafale » (« *burst* »). Dans ce cas, on place sur le bus l'adresse du premier mot de la ligne à charger, les mots de la ligne étant alors envoyés par la mémoire les uns après les autres. Par ce moyen, on diminue grandement le temps de chargement d'une ligne de cache, puisqu'on ne spécifie qu'une seule fois l'adresse de lecture, au lieu de le faire une fois par mot de la ligne. De plus, lorsque le cache charge une ligne, il retourne le mot demandé au processeur dès qu'il le reçoit, sans attendre la fin du chargement complet de la ligne.

Quand le processeur souhaite écrire une donnée en mémoire, trois techniques peuvent être utilisées par le cache pour réaliser cette opération :

- « *write through* » : toute opération d'écriture demandée par le processeur provoque l'écriture effective de la donnée en mémoire, même en cas de « *cache hit* ». Si la donnée était déjà présente dans le cache, celui-ci est également mis à jour.
  Faute d'optimisations, cette technique reviendrait à supprimer le cache lors des opérations d'écriture, et ralentirait celles-ci de façon catastrophique. Pour éviter cela, les caches de ce type disposent de tampons d'écriture (« *fast write buffers* »), qui permettent le traitement asynchrone des opérations d'écriture, en évitant au processeur d'attendre leur réalisation effective. Le problème ne resurgit que lorsque les tampons sont pleins.
  Le grand avantage de la technique « *write through* » est qu'elle rend les lignes de cache immédiatement disponibles pour leur réallocation ;
- « *write back* » : toute opération d'écriture demandée par le processeur provoque la mise à jour du cache, mais la ligne modifiée n'est recopiée en mémoire que lorsqu'elle doit faire place à de nouvelles données dont le processeur a besoin.
  La technique « *write back* » ralentit le remplacement de lignes de cache, du fait des écritures à réaliser avant le chargement des nouvelles lignes, mais ceci est en général compensé par la suppression des (multiples) opérations d'écriture en mémoire qui ont ainsi été évitées ;
- « *write allocate* » : dans le cas où la donnée à écrire n'est pas déjà présente dans le cache, cette technique consiste à allouer la ligne de cache correspondante, en la chargeant à partir de la mémoire une fois que l'écriture a été prise en compte par celle-ci. Comme la donnée a effectivement été écrite, le processeur peut immédiatement poursuivre son traitement, pendant que le chargement de la ligne s'effectue de façon asynchrone.
  La technique « *write allocate* » n'est vraiment utile que pour les caches de type « *write back* ». De fait, lors du chargement de la nouvelle ligne de cache, on aura souvent d'abord à écrire une ligne de cache modifiée avant de charger la nouvelle ligne à sa place.
  À cause de cette complexité, la plupart des caches n'implémentent pas de stratégie « *write allocate* » ; les opérations d'écriture provoquant des défauts de cache sont simplement répercutées vers la mémoire centrale, et ignorées par le cache.

Aucune des deux stratégies, « *write back* » ou « *write through* », ne l'emporte clairement sur l'autre : leurs performances relatives dépendent de la structure des accès mémoire réalisés par les programmes.

4.3.2 Structure

Tout cache nécessite, en plus de la zone mémoire réservée aux données (servant au stockage des lignes), des informations de contrôle servant d'index de recherche (ou de répertoire, « *directory* ») dans le cache. On distingue quatre types principaux d'organisation des données dans les caches :

– la correspondance directe (« *direct mapping* ») : chaque donnée de la mémoire a une place précalculée unique dans le cache. Ainsi, avec un cache de capacité 2^{s+l}, les données situées aux adresses a, $a+2^{s+l}$, $a+2\cdot 2^{s+l}$, $a+3\cdot 2^{s+l}$ seront-elles stockées à la même place de la même ligne du cache, comme illustré en figure 4.4.

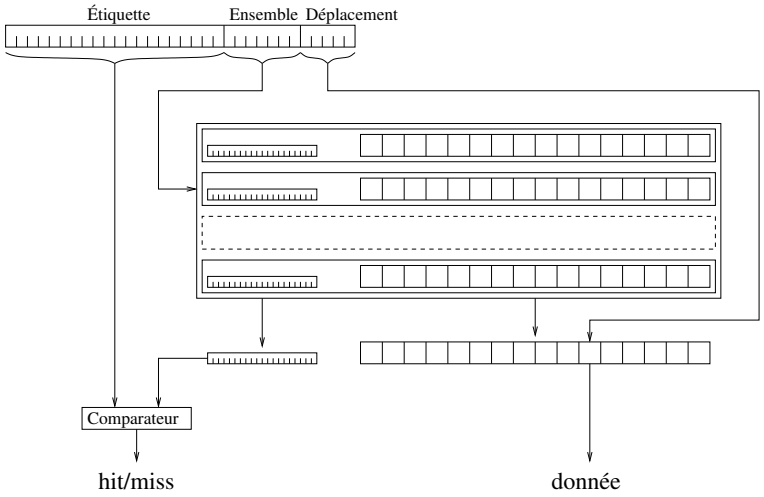


FIG. 4.4 – Structure d'un cache à correspondance directe.

Cette organisation est extrêmement simple et rapide, car elle ne requiert qu'un unique comparateur pour tester si l'étiquette de la ligne de cache correspond bien à la partie haute de l'adresse fournie ; si c'est le cas, on a un « *cache hit* », sinon un « *cache miss* ». L'inconvénient majeur de ces caches est que leurs performances dépendent fortement de l'alignement des structures de données qu'ils cachent. Dans le cas d'une copie élément par élément entre deux tableaux dont les adresses de début diffèrent d'un multiple de 2^{s+l}, par exemple, on aura deux défauts de cache par élément ;

– la k-associativité par ensemble (« *k-way set associativity* ») : le cache est subdivisé en ensembles (« *sets* ») contenant plusieurs lignes (de deux à seize), et disposant d'informations de contrôle propres servant à la gestion individuelle de ces lignes. Chaque donnée de la mémoire a un unique ensemble destination, mais dans cet ensemble sa position est totalement libre. Cette structure est illustrée en figure 4.5, pour un cache 2-associatif. Le choix de la ligne à remplacer lorsqu'une nouvelle ligne doit être chargée dans un ensemble s'effectue au moyen d'une politique LRU (« *Least Recently Used* ») ;

– l'associativité totale (« *full associativity* ») : le cache est constitué d'un unique ensemble, et donc une donnée peut être placée dans n'importe quelle ligne. Ce type de cache offre les meilleurs taux de réussite (« *hit ratio* »), mais est le plus complexe à implémenter ;

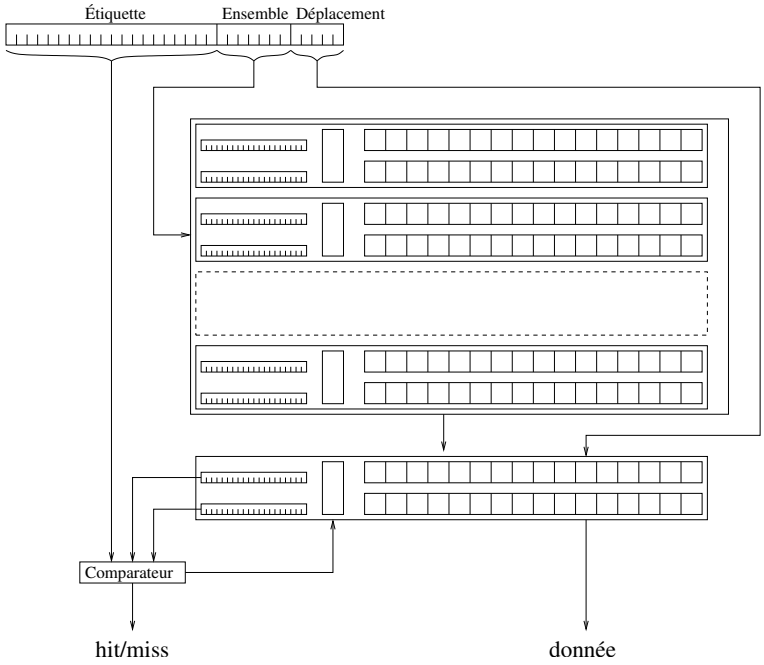


FIG. 4.5 – Structure d'un cache 2-associatif.

– la correspondance par secteurs (« *sector mapping* ») : le cache est subdivisé en secteurs (« *sectors* ») contenant plusieurs lignes (de 2 à 32) qui disposent chacune d'un bit de validité. La recherche d'une donnée dans le cache s'effectue par comparaison associative totale entre son étiquette de secteur (« *sector frame* ») et toutes les étiquettes de secteur du cache, puis par indexation directe à l'intérieur du secteur choisi. Cette structure est illustrée en figure 4.6. Cette architecture revient à augmenter la granularité du cache, tout en conservant une taille de données en lecture/écriture égale à celle d'une ligne. Elle est moins coûteuse que l'associativité totale, en ce que les comparaisons s'effectuent sur un nombre plus restreint d'étiquettes. Elle convient bien aux machines dédiées au calcul numérique, car les boucles réalisées dans les algorithmes de calcul scientifique n'accèdent en général qu'à quelques zones de grande taille à la fois.

4.3.3 Adressage

Les adresses mémoire soumises aux caches peuvent être soit les adresses logiques fournies par le processeur, soit les adresses physiques résultant de la traduction des adresses logiques par la MMU (« *Memory Management Unit* »).

Lorsque le cache utilise l'adressage physique, l'accès au cache s'effectue après que l'adresse logique émise par le processeur a été traduite en adresse physique par la MMU, ce qui ralentit les accès mémoire.

Lorsque le cache utilise l'adressage logique, les accès au cache s'effectuent parallèlement à la traduction de l'adresse logique en adresse physique, ce qui permet d'accélérer le traitement des accès mémoire. En revanche, cela génère un problème de cohérence pour les systèmes multi-processus, puisqu'alors la même adresse logique, utilisée par des processus différents, doit correspondre à des données différentes. Pour remédier à cela, une solution simple consiste à invalider l'ensemble des lignes du cache lors des changements de contextes entre processus. Cette solution est cependant extrêmement coûteuse, surtout dans le cas des caches « *write back* », qui

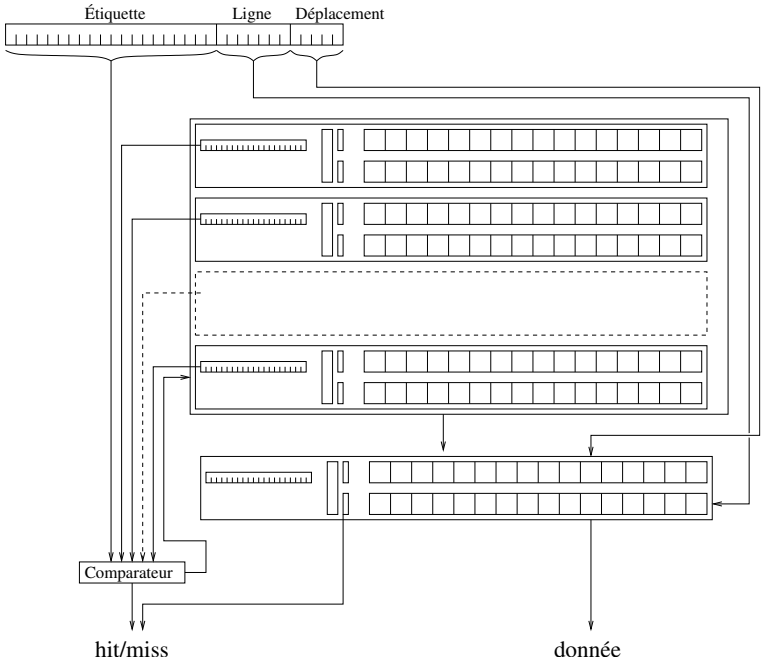


FIG. 4.6 – Structure d'un cache sectoriel.

nécessitent l'écriture en mémoire de toutes leurs lignes modifiées. Une autre solution consiste à associer aux lignes du cache, en plus de leur étiquette, un identificateur de processus. Cette solution est viable, mais induit un surcoût mémoire qui peut être important.

4.3.4 Cohérence

L'existence de caches internes aux processeurs rend la réalisation de machines multi-processeurs à mémoire partagée plus délicate, du fait des incohérences pouvant exister entre les valeurs d'une même référence mémoire contenues dans les caches de processeurs différents, comme illustré en figure 4.7.

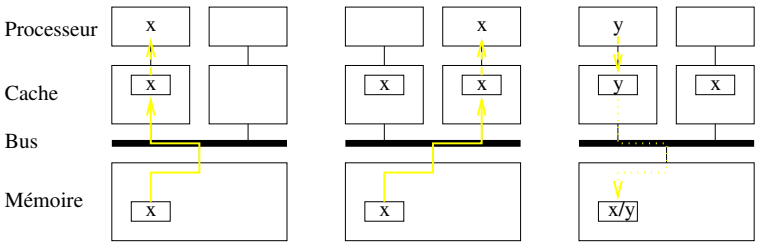


FIG. 4.7 – Mise en évidence d'incohérences potentielles entre caches locaux et mémoire commune sur une machine bi-processeur. Lorsque le premier processeur demande la valeur d'une case mémoire, cette valeur est conservée dans son cache local. Il en est de même lorsque le deuxième processeur effectue la même requête. Si le premier processeur modifie la valeur, la modification peut être ou non répercutée en mémoire selon le type de cache (« *write through* » ou « *write back* »), mais lors d'accès ultérieurs, le deuxième processeur verra toujours l'ancienne valeur contenue dans son cache local.

L'incohérence entre caches est uniquement causée par les écritures. Elle se produit lorsqu'une opération d'écriture, qui modifie la valeur de la référence mémoire contenue dans le cache du processeur effectuant l'écriture (et également la mémoire elle-même, dans le cas d'un cache « *write-through* »), n'est pas répercutée sur les autres caches possédant l'ancienne valeur.

Deux politiques sont envisageables pour maintenir la cohérence entre caches :
- l'invalidation sur écriture (« *write-invalidate* ») : la mise à jour effectuée sur un cache provoque l'invalidation (réinitialisation du bit de validité) de toutes les copies de la ligne possédées par les autres caches ; les lignes invalidées sont dites « périmées » (« *dirty* »).
  Ainsi, lorsqu'un autre processeur demandera à lire cette référence mémoire, la valeur transmise sera lue à partir de la mémoire centrale, et non à partir d'une ligne de cache périmée.
  De fait, cette politique n'est envisageable que pour les caches de type « *write-through* », qui assurent en permanence la cohérence entre l'état de la mémoire et des caches.
- la mise à jour sur écriture (« *write-update* ») : la mise à jour effectuée sur un cache est également effectuée sur tous les autres caches possédant la référence mémoire.

La prise en compte par tous les caches des opérations d'écriture effectuées par l'un d'entre eux nécessite dans tous les cas une circuiterie supplémentaire.

Lorsque les caches sont de type « *write through* », un protocole d'espionnage du bus (« *snooping protocol* ») permet de tracer toutes les opérations d'écriture réalisées par les autres caches, et éventuellement de lire à la volée la nouvelle valeur des références mémoires afin de répercuter localement la mise à jour. Cependant, dans un environnement multi-processeurs, l'utilisation systématique du bus par les caches « *write-through* » fait de celui-ci un goulet d'étranglement.

Pour remédier à cela, tout en assurant la cohérence des caches dans un environnement multi-processeurs, a été développé un protocole de gestion de caches dit « à écriture unique » (« *write-once* ») [3], dont le représentant le plus connu est le protocole MESI (« *Modified, Exclusive, Shared, Invalid* »). Selon ce protocole, chaque ligne de cache peut prendre quatre états distincts :
- « *invalid* » : la ligne de cache ne contient pas de données valides ;
- « *shared* » : la ligne de cache contient des données à jour, qui n'ont pas été modifiées depuis leur chargement dans le cache. D'autres processeurs peuvent également posséder des copies de ces données dans leurs propres caches ;
- « *exclusive* » : les données de la ligne de cache n'ont été modifiées localement qu'une seule fois depuis leur chargement dans le cache (elles étaient alors en mode « *shared* »), et la modification a été répercutée en mémoire centrale, selon le principe « *write-through* ». Aucun autre cache ne possède de copie valide de la ligne (d'où le nom) ;
- « *modified* » : les données de la ligne de cache ont été modifiées localement plusieurs fois depuis leur chargement dans le cache, mais les modifications successives n'ont pas été répercutées en mémoire centrale, selon le principe « *write-back* ». On ne peut arriver à cet état qu'à partir de l'état « *exclusive* ». Ici encore, aucun autre cache ne possède de copie valide.

Lorsqu'une nouvelle ligne est chargée dans le cache à partir de la mémoire, son état est positionné à « *shared* ». D'autres caches peuvent également charger les mêmes données, qui seront également étiquetées localement « *shared* ».

Lorsqu'une ligne « *shared* » est modifiée localement pour la première fois, son état passe à « *exclusive* », et la modification est répercutée à travers le bus vers

la mémoire centrale, selon le principe « *write-through* ». Ainsi, par l'espionnage du bus, tous les caches possédant une copie « *shared* » de la ligne l'invalideront.

Lorsqu'une ligne « *exclusive* » est modifiée localement pour la première fois, son état passe à « *modified* », et la modification n'est pas répercutée vers la mémoire centrale, de même que les suivantes. D'après ce qui précède, aucun autre cache ne possède de copie valide de la ligne, puisque le passage précédent de la ligne en mode « *exclusive* » les a toutes invalidées. Cependant, la mémoire centrale n'est plus à jour, et tout processeur redemandant cette ligne chargera des données périmées. Pour éviter cela, le cache possédant une copie en mode « *modified* » d'une ligne demandée par un autre cache doit intercepter la requête, et placer lui-même la nouvelle valeur de la ligne sur le bus, en prenant le pas sur la mémoire centrale ; cette notion de préemptivité du bus est implémentée dans tous les bus récents (*Multibus* et *Futurebus*). Dans le même temps, le cache propriétaire écrira la ligne en question en mémoire centrale, et remettra l'état de sa ligne à « *shared* », puisqu'une autre copie existe sur un autre cache.

L'idée des caches « *write-once* » est donc de remplacer la mise à jour systématique de la mémoire, génératrice d'engorgements, par une mise à jour à la demande, les caches propriétaires des lignes modifiées se chargeant alors de les fournir aux caches demandeurs. Un grand avantage de ce protocole est qu'il permet d'associer librement sur le même bus mémoire des unités de traitement disposant de caches (comme les processeurs) et d'autres n'en ayant pas (comme les périphériques d'entrées/sorties) ; c'est en fait la préemptibilité du bus qui permet de les interfacer sans circuiterie supplémentaire.

4.3.5 Hiérarchies de caches

Le besoin de performance accrue en terme de débit mémoire le processeur a conduit les concepteurs de caches à réaliser des caches à triple niveau. Ainsi, sur le processeur Itanium2, on trouve :

- un cache de premier niveau constitué d'un cache d'instructions de 16 Ko et d'un cache de données de 16 Ko également, capable de servir quatre requêtes en lecture par cycle, ou deux requêtes en lecture et deux en écriture, structuré en lignes de 64 octets ;
- un cache de deuxième niveau unifié de 256 Ko, 8-*way associative write-back*, structuré en 16 bancs de lignes de 128 octets, et capable de servir quatre requêtes par cycle, avec une latence d'au moins 6 cycles. Le cache est non-bloquant, capable de gérer simultanément, grâce à une file spécifique appelée L2OzQ, jusqu'à 32 requêtes provenant du cache de premier niveau, de réordonner les requêtes en fonction des conflits de bancs et des identités d'adresses sur les *Load* et les *Store*, et de gérer jusqu'à 16 requêtes simultanées vers le cache de troisième niveau ;
- un cache de troisième niveau unifié de 1.5 Mo, 12-*way associative*, avec une latence d'au moins 12 cycles.

La complexité des mécanismes mis en œuvre dans ces hiérarchies de cache, qui peuvent interférer entre eux et induire des pertes de performance considérables (de plus de la moitié) selon les positions relatives en mémoire des flots de données manipulés, nécessitent une analyse approfondie afin de réaliser les noyaux de calcul les plus efficaces possible [5].

Comme en moyenne, sur les processeurs Itanium de première génération, il a été mesuré que le traitement des *cache miss* de données représentait plus de la moitié du temps d'exécution des programmes, l'architecture Itanium2 permet de spécifier, au niveau des instructions *Load*, la localisation probable de la donnée dans la hiérarchie de cache (pour adapter la latence de chargement) ainsi que le

niveau de cache dans lequel il sera préférable de conserver la donnée une fois qu'elle aura été accédée (qui permet aux caches de niveaux inférieurs de marquer la donnée comme pouvant être remplacée de préférence à des données plus utiles). Ces indices de gestion des caches (« *cache hints* ») sont positionnés par le compilateur après analyse statique du code à la compilation, ou bien par le programmeur en langage machine souhaitant réaliser des noyaux de calul efficaces.

4.4 Mémoire centrale

Sur la plupart des machines, la vitesse de la mémoire centrale n'est pas suffisante à elle seule pour alimenter le processeur selon ses besoins ; plusieurs cycles doivent s'écouler entre le moment où une donnée est demandée et celui où elle est disponible sur le bus.

Afin d'améliorer le débit de la mémoire, on organise celle ci en bancs entrelacés (« *interleaved banks* ») indépendants, dont chacun gère une partie de l'espace d'adressage. Typiquement, si l'on dispose de N bancs mémoire numérotés de 0 à $N-1$, le banc i est affecté aux adresses de la forme $bN+i$, ce qui permet un accès concurrent à des adresses consécutives de la mémoire.

Pour que la mémoire puisse fournir le débit demandé par le processeur, il faut que le nombre de bancs de mémoire soit au moins égal au nombre de cycles de latence de celle-ci. Sur le CRAY 1 (1976), qui avait un temps de cycle τ de $12,5$ ns et une latence mémoire de 50 ns, soit 4 cycles, la mémoire était divisée en 16 bancs indépendants, permettant ainsi un débit entre le processeur et la mémoire de 4 mots par cycle, comme illustré en figure 4.8.

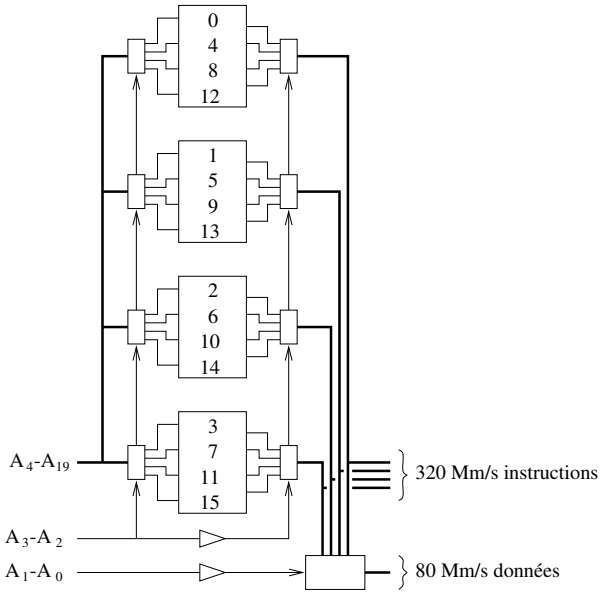


FIG. 4.8 – Schéma d'accès aux bancs mémoire du CRAY 1. Ce schéma est à rapprocher de celui de l'architecture fonctionnelle générale du CRAY 1, présentée en figure 3.30, page 38.

Il y a conflit d'accès si deux opérations sont demandées au même banc dans un intervalle inférieur au temps de latence de la mémoire. Par exemple, si une mémoire de 16 bancs a une latence de 4 cycles, un conflit se produira si au moins deux accès

Incrément	Cray X-MP/48	NEC SX-2	Fujitsu VP-200
Bancs	64	512	128
1	86	255	365
2	67	244	127
3	72	244	228
4	52	211	67
5	75	244	227
6	71	244	127
7	76	244	228
8	63	160	67
9	68	244	225
16	73	103	67

TAB. 4.1 – Performance, en Mflop/s, du calcul terme à terme d'un produit scalaire entre deux vecteurs dont les termes utiles sont espacés d'un incrément donné.

sur quatre consécutifs diffèrent d'un multiple de 16, comme c'est le cas pour un incrément multiple de 8 (un conflit tous les deux accès), voire de 16 (un conflit par accès). Le tableau 4.1 donne la performance en Mflop/s du calcul terme à terme d'un produit scalaire entre deux vecteurs dont les termes utiles sont espacés d'un incrément donné, en fonction de la valeur de cet incrément.

La performance de la mémoire dépend donc assez fortement de l'alignement des données, qu'il est souhaitable de modifier en conséquence. Ainsi, dans le cas d'un programme de diffusion sur une grille périodique de taille 128×128, tel que celui présenté en figure 4.9, on allouera les tableaux `TAB1` et `TAB2` comme des grilles `(130, 128)` plutôt que `(128, 128)`, pour optimiser le schéma d'accès à la mémoire, chaque cellule et ses quatre voisines étant alors situées sur des bancs mémoire tous différents.

```
      DO 20 I = 1, 128
        DO 10 J = 1, 128
          IA = (I + 127) MOD 128
          IP = (I + 1)   MOD 128
          JA = (J + 127) MOD 128
          JP = (J + 1)   MOD 128
          TAB2 (I, J) = (TAB1(IA, J) + TAB1(I, JA) +
     *                   TAB1(I, JP) + TAB1(IP,J)) / 4
10      CONTINUE
20    CONTINUE
```

FIG. 4.9 – Boucle principale d'un programme résolvant l'équation de la chaleur sur un réseau torique carré.

Dans le cas d'architectures disposant d'instructions mémoire/mémoire, et comme certaines instructions agissent sur trois opérandes, il faut disposer d'un débit mémoire effectif entre processeur et mémoire de trois mots par cycle, et garantir ce débit pour chaque processeur dans le cas d'architectures multi-processeurs. Pour cela, certaines architectures disposent de chemins d'accès multiples. Ainsi, sur le CRAY Y-MP, qui a 8 processeurs de latence τ égale à 6 ns, et nécessite donc un débit de 4 Gm/s, la mémoire est divisée en 128 bancs, accessibles par trois chemins séparés (deux en lecture et un en écriture) pour chaque processeur, ce qui permet d'atteindre les 4 Gm/s si aucun conflit n'intervient.

4.5 Disques

4.5.1 Gestion des accès

Lorsqu'on manipule de très gros volumes de données, celles-ci ne peuvent tenir entièrement en mémoire centrale. Lorsque cela est techniquement réalisable (quand chaque processeur possède son propre disque local, ou que l'on ne risque pas d'écrouler le réseau d'interconnexion), il est alors possible d'utiliser des disques comme espace de stockage temporaire. Cette fonctionnalité peut être gérée à deux niveaux :

- au niveau du système (matériel et noyau) : ce sont les mécanismes classiques de mémoire virtuelle et de « va-et-vient » (« *swapping* »). Ces mécanismes automatiques évitent de modifier l'algorithme, mais il est souvent possible d'exhiber des cas pathologiques d'écroulement résultant d'interférences entre l'algorithme de calcul et l'algorithme de gestion du va-et-vient ;
- au niveau de l'application elle-même (logiciel) : le chargement et la sauvegarde explicites des ensembles de données sont spécifiés par le programmeur, en fonction de l'algorithme, qui est alors dit « *out-of-core* ». Cette approche est la plus efficace, mais elle est coûteuse en temps de développement et demande une connaissance approfondie des paramètres du système (temps moyen des accès disques, taille des tampons système, etc.).

4.5.2 Organisation des données

L'organisation des données et leurs schémas d'accès peuvent avoir des conséquences extrêmement importantes sur les temps d'exécution des programmes. À titre d'exemple, considérons un algorithme calculant le produit matriciel $C = AB + C$ sur des matrices carrées de taille 1024×1024 rangées par colonne (style FORTRAN), sur une architecture disposant d'une mémoire centrale de 16 pages de 65536 valeurs chacune (chaque page pouvant ainsi stocker 64 colonnes de matrice).

En écrivant l'algorithme de produit matriciel de façon classique, comme décrit en figure 4.10, chaque lecture d'une ligne de A provoque 16 défauts de page, d'où plus de 16 millions de défauts de page au total.

```
      DO 30 I = 1, 1024
        DO 20 J = 1, 1024
          DO 10 K = 1, 1024
            C(I, J) = C(I, J) + A(I, K) * B(K, J)
10        CONTINUE
20      CONTINUE
30    CONTINUE
```

FIG. 4.10 – Écriture classique de l'algorithme de calcul d'un produit de matrices.

Avec une approche par blocs colonne, où l'on partitionne la matrice A en blocs de 64 colonnes et B en blocs de 64×64 valeurs, comme présenté en figure 4.11, chaque parcours de J génère 16 défauts de page, et la boucle KK en génère elle-même 16, d'où $16 \times (16 + 1) = 272$ défauts de page au total.

Avec une approche purement par blocs, où l'on partitionne les matrices en blocs de 256×256 valeurs, on descend jusqu'à 96 défauts de page au total !

Le principal problème des disques provient de la nature mécanique des accès : les vitesses de rotation et les contraintes thermiques sont en effet telles qu'un

```
      DO 40 KK = 1, 1024, 64
        DO 30 J = 1, 1024
          DO 20 I = 1, 1024
            DO 10 K = KK, KK + 63
              C(I, J) = C(I, J) + A(I, K) * B(K, J)
10          CONTINUE
20        CONTINUE
30      CONTINUE
40    CONTINUE
```

FIG. 4.11 – Écriture classique de l'algorithme de calcul d'un produit de matrices.

Caractéristique	IBM 3380	Conners CP3100	Ratio
Capacité (Mo)	7500	100	75,0
Prix par Mo	\$18-\$10	\$10-\$7	1,00-2,50
MTTF annoncé (h)	30000	30000	1,00
MTTF effectif	100000	?	?
Nombre de têtes	4	1	4,00
Débit (Mo/s)	3	1	3,00
Puissance (W)	6600	10	660

TAB. 4.2 – Caractéristiques des disques IBM 3380 modèle AK4 et Conners CP3100.

réalignement de la tête de lecture est nécessaire entre chaque accès, même si les blocs à lire sont situés sur la même piste, ce qui limite actuellement les débits aux environs de 20 Mo/s. Le « *disk striping* », qui répartit les fichiers sur plusieurs disques, permet le transfert des données en parallèle sur plusieurs unités, au moyen de protocoles évolués tels que IPI3 (« *Intelligent Parallel Interface, version 3* ») ou HiPPI (« *High Performance Parallel Interface* »), qui permettent d'atteindre des débits de 100 Mo/s.

Dans le futur, les techniques holographiques, encore confidentielles, permettront des temps d'accès de l'ordre de 1 à 10 μs et des taux de transfert de 100 Mo à 1 Go/s. En revanche, le stockage ne peut être que de courte durée, ce qui limitera l'utilisation de ces techniques aux caches des unités centrales et aux unités de stockage temporaires.

4.5.3 Baies de disques (RAID)

Les baies de disques (« *disk arrays* ») sont une solution efficace. Elles sont constituées d'un grand nombre de disques peu chers, accédés en parallèle, et disposant de protocoles évolués dits RAID (« *Redundant Arrays of Inexpensive Disks* ») [10] permettant la correction d'erreurs et la reprise à chaud.

L'émergence de la technologie RAID tient au fait que, si la capacité unitaire des disques hautes performances (« *Single Large Expensive Disk* », ou SLED) a crû en rapport avec l'augmentation des puissances de calcul et des tailles des mémoires centrales, le temps de positionnement des bras n'a diminué que d'un facteur deux de 1971 à 1981.

La table 4.2 compare quelques paramètres significatifs d'un disque SLED IBM 3380 modèle AK4 et d'un disque de PC Conners CP3100, disponibles en 1987 lors de la publication de l'article définissant le RAID.

Ces caractéristiques ont permis d'imaginer la définition de systèmes de stockage constitués d'un grand nombre de disques peu chers et de petite capacité, gérés soit de manière entrelacée pour absorber les gros volumes produits par les supercalcu-

Caractéristique	RAID 1
MTTF annoncé (années)	> 500
Nombre total de disques	$2\,D$
Surcoût (%)	100
Capacité utile (%)	50
Grosses lectures (1/s)	$2\,D/S$
Grosses écritures (1/s)	D/S
Grosses L-M-É (1/s)	$2\,D/3\,S$
Petites lectures (1/s)	$2\,D$
Petites écritures (1/s)	D
Petites L-M-É (1/s)	$2\,D/3$

TAB. 4.3 – Caractéristiques d'un système RAID 1.

lateurs, soit de manière indépendante pour traiter les nombreux petits transferts générés par les applications transactionnelles.

Le problème majeur des systèmes RAID est la tolérance aux pannes. En effet, le MTTF (« *Mean Time To Failure* ») d'un système composé de plusieurs disques est inversement proportionnel au nombre de ces disques. Ainsi, un système RAID de 100 disques Conners CP3100 disposerait d'un MTTF annoncé de 300 heures, soit moins de deux semaines !

Pour remédier à cela, il faut mettre en place des mécanismes autorisant le système à fonctionner malgré la panne d'au moins un disque, et permettant la réparation « à chaud ». Du point de vue du stockage, ceci nécessite l'utilisation de disques supplémentaires pour dupliquer l'information, afin de palier la panne d'un disque et de reconstruire l'information manquante lors de son remplacement par un disque neuf vierge. On organise donc les D disques de données en groupes de G disques, à chacun desquels sont adjoints C disques de contrôle. Si ont définit MTTR (« *Mean Time to Repair* ») comme le temps moyen de maintenance, on obtient la formule :

$$\mathrm{MTTF}_{\mathrm{RAID}} = \frac{\mathrm{MTTF}^2_{\mathrm{Disk}}}{D\left(1+\frac{C}{G}\right)(G+C-1)\mathrm{MTTR}} \ .$$

Plusieurs niveaux d'organisation RAID ont été définis, qui offrent chacun des niveaux de sécurité et de performance différents. Pour évaluer cette dernière, on distingera les « grosses » entrées/sorties générées par les supercalculateurs, qui mobilisent au moins un secteur de chaque disque d'un groupe, des « petites » entrées/sorties générées par les systèmes transactionnels, qui sont basées sur des cycles indépendants de lecture-modification-écriture.

Dans tous les cas, on supposera que la taille des blocs de données manipulées par l'utilisateur est au moins égale à la taille d'un secteur disque.

Notons que l'utilisation de systèmes RAID pour des entrées/sorties mobilisant plusieurs disques génère un surcoût S par rapport au temps d'un accès unique sur un disque unique, car il faut attendre la terminaison d'un ensemble de disques non synchronisés. Dans tous les cas énumérés ci-dessous, on prendra $\mathrm{MTTR} = 1$ heure, et $D = 100$ disques, afin de rendre la capacité du système RAID équivalente à celle du SLED de même génération décrit plus haut.

– RAID 1 : disques miroirs (« *mirroring* »). Chaque disque est pourvu d'une copie conforme, donc $G = 1$ et $C = 1$. C'est l'option la plus coûteuse, puisque chaque donnée est dupliquée. Elle est d'ailleurs économiquement bien trop coûteuse, puisque le MTTF annoncé est très largement supérieur à la durée de vie du produit, comme indiqué dans le tableau 4.3.

– RAID 2 : codage d'erreur par code de Hamming. Ce codage ne nécessite que $O(\log_2(G))$ disques de contrôle pour déterminer le disque fautif et regénérer l'information manquante. Ainsi, avec $G = 25$, on a $C = 5$. Afin de paralléliser les accès, chaque bloc de données est réparti sur tous les disques de données du groupe chargé de son stockage, comme illustré en figure 4.12. Les caractéristiques du RAID 2 sont données dans le tableau 4.4.

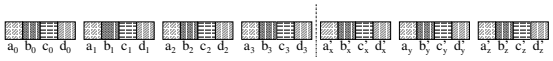


FIG. 4.12 – Organisation des données dans un système RAID 2.

Caractéristique	RAID 2
MTTF annoncé (années)	12
Nombre total de disques	$1,2\,D$
Surcoût (%)	20
Capacité utile (%)	83
Grosses lectures (1/s)	D/S
Grosses écritures (1/s)	D/S
Grosses L-M-É (1/s)	$D/2\,S$
Petites lectures (1/s)	$D/S\,G$
Petites écritures (1/s)	$D/2\,S\,G$
Petites L-M-É (1/s)	$D/2\,S\,G$

TAB. 4.4 – Organisation des données et caractéristiques d'un système RAID 2.

– RAID 3 : codage d'erreur par parité. Le codage de Hamming utilisé dans le RAID 2 est lui-même trop coûteux, puisqu'il permet de déterminer le disque fautif, alors que dans la presque totalité des cas ceci pourra être déterminé simplement au niveau du contrôleur. On n'a donc besoin que d'un codage par parité pour regénérer l'information manquante, qui ne coûte qu'un disque supplémentaire par groupe, comme illustré en figure 4.13. Avec $C = 1$ et $G = 25$, la réduction du nombre de diques de contrôle permet même d'augmenter le MTTF par rapport au RAID 2, comme indiqué dans le tableau 4.5.

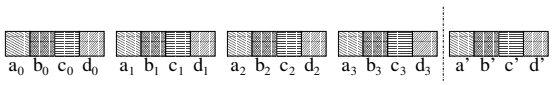


FIG. 4.13 – Organisation des données dans un système RAID 3.

– RAID 4 : lectures et écritures indépendantes. Répartir un transfert sur plusieurs disques a comme avantage de réduire le temps de transfert de grosses entrées/sorties car l'ensemble de la bande passante peut être exploitée. Cependant, les petites entrées/sorties nécessitent d'utiliser tous les disques du groupe concerné, et donc les RAID 2 et 3 ne peuvent effectuer qu'une entrée/sortie par groupe à la fois. De plus, si les disques ne sont pas synchronisés, le temps de réalisation de l'entrée/sortie est celui du disque ayant terminé le dernier, d'où l'existence du facteur S pour les petites entrées/sorties. Le RAID 4, en conservant chaque bloc de données sur un unique disque, permet d'effectuer plusieurs entrées/sorties simultanées dans chaque groupe. Avec cette nouvelle organisation, illustrée en figure 4.14, le calcul de parité s'effectue sur les mêmes portions de blocs différents.

Caractéristique	RAID 3
MTTF annoncé (années)	40
Nombre total de disques	$1,04\,D$
Surcoût (%)	4
Capacité utile (%)	96
Grosses lectures (1/s)	D/S
Grosses écritures (1/s)	D/S
Grosses L-M-É (1/s)	$D/2\,S$
Petites lectures (1/s)	$D/S\,G$
Petites écritures (1/s)	$D/2\,S\,G$
Petites L-M-É (1/s)	$D/2\,S\,G$

TAB. 4.5 – Caractéristiques d'un système RAID 3.

On pourrait penser qu'une petite écriture implique tous les disques d'un groupe, du fait de la nécessité de recalculer le contrôle d'erreur. Cependant, comme celui-ci se fait par parité, on peut calculer localement sa variation par ou exclusif entre les anciennes données et les nouvelles. Les caractéristiques du RAID 4 sont données dans le tableau 4.6.

FIG. 4.14 – Organisation des données dans un système RAID 4.

Caractéristique	RAID 4
MTTF annoncé (années)	40
Nombre total de disques	$1,04\,D$
Surcoût (%)	4
Capacité utile (%)	96
Grosses lectures (1/s)	D/S
Grosses écritures (1/s)	D/S
Grosses L-M-É (1/s)	$D/2\,S$
Petites lectures (1/s)	D
Petites écritures (1/s)	$D/2\,G$
Petites L-M-É (1/s)	$D/2\,G$

TAB. 4.6 – Caractéristiques d'un système RAID 4.

– RAID 5 : équivalent au RAID 4, mais avec entrelaçage des disques de contrôle. La faiblesse du RAID 4 réside dans les disques de contrôle de parité, qui sont des goulots d'étranglement puisqu'ils sont sollicités à chaque écriture dans un groupe. Pour remédier à cela, le RAID 5 répartit les secteurs de contrôle sur tous les disques, de façon cyclique. Les caractéristiques du RAID 5 sont données dans le tableau 4.7.

4.6 Systèmes de fichiers parallèles

Une autre approche consiste à distribuer les disques sur les nœuds de la machine parallèle, afin que chacun d'entre eux dispose d'une zone de va-et-vient (« *swap* »)

Caractéristique	RAID 5
MTTF annoncé (années)	40
Nombre total de disques	$1,04\,D$
Surcoût (%)	4
Capacité utile (%)	96
Grosses lectures (1/s)	D/S
Grosses écritures (1/s)	D/S
Grosses L-M-É (1/s)	$D/2\,S$
Petites lectures (1/s)	$\left(1+\frac{C}{G}\right)\,D$
Petites écritures (1/s)	$\left(1+\frac{C}{G}\right)\,D/4$
Petites L-M-É (1/s)	$\left(1+\frac{C}{G}\right)\,D/4$

TAB. 4.7 – Caractéristiques d'un système RAID 5.

et d'un espace temporaire de stockage (pour l'exécution de programmes « *out-of-core* ») propres. Ces disques peuvent être fédérés au moyen de protocoles parallèles tels que PIOFS (« *Parallel I/O File System* ») pour constituer un système de fichiers distribué.

Chapitre 5

Systèmes d'exploitation

5.1 Généralités

Les systèmes d'exploitation supportant le parallélisme appartiennent à deux familles distinctes :

- celle des systèmes d'exploitation distribués (ou répartis), qui permettent d'utiliser et de partager des ressources et services répartis sur le réseau, en assurant à l'utilisateur la transparence de celui-ci, ainsi qu'une fiabilité maximale ;
- celle des systèmes d'exploitation des machines multiprocesseurs et parallèles dédiées au calcul intensif, pour lesquels l'obtention de performances élevées est primordiale.

À mesure que les représentants de ces deux familles gagnent en maturité, on assiste à un rapprochement entre ces deux tendances, par influence mutuelle, mais le rapprochement est loin d'être achevé, si tant est qu'il puisse l'être.

Les architectures supportant ces systèmes d'exploitation peuvent être regroupées en trois classes :

- les machines multiprocesseurs à mémoire partagée, de type UMA ou NUMA, voire COMA ;
- les machines parallèles à mémoire distribuée, de type NORMA, dont les éléments sont liés par un réseau d'interconnexion rapide (de 1 à 10 Gbit/s) disposant éventuellement de fonctionnalités spécifiques : diffusion, synchronisation ;
- les systèmes distribués, constitués d'un ensemble de machines autonomes liées par un réseau local (avec un débit de 10 Mbit/s à 1 Gbit/s).

5.2 Structure

Un nombre très important de (prototypes de) systèmes d'exploitation a été proposé durant les dix dernières années, qui diffèrent par leurs buts, leur structure, et leurs fonctionnalités. Du point de vue de leur structure, on peut les regrouper en plusieurs catégories :

- les systèmes monolithiques. Ils sont constitués d'un noyau complexe et de grande taille, isolé de l'espace utilisateur par des moyens matériels simples (segmentation et mode privilégié), mais ne possédant pas de barrières entre ses différents modules. Toute communication entre processus implémentant des fonctionnalités système de haut niveau (démons) s'effectue au moyen de zones de mémoire partagée gérée par le système, ou par envoi explicite de requêtes (« *sockets* »). C'est le cas d'UNIX et de VMS, par exemple.

Le manque de barrières solides au sein du noyau, ainsi que la grande taille et la complexité de celui-ci, rendent de tels systèmes difficiles à maintenir et à valider, et à fortiori à adapter à un environnement distribué. De fait, les implémentations parallèles de tels systèmes se sont limitées aux architectures de type UMA : Solaris pour Sun, IRIX pour SGI, AIX pour IBM, par exemple ;

– les systèmes micro-noyaux. Ils sont constitués d'un micro-noyau minimal s'exécutant sur chaque processeur et ne supportant qu'un nombre restreint de services (gestion de processus, de la mémoire, des communications inter-processus, et support des gestionnaires de périphériques), le reste des fonctionnalités du système étant assuré par des serveurs éventuellement situés sur des nœuds spécialisés (entrée/sorties). L'exécution de la plupart des primitives système s'effectue donc par l'intermédiaire d'appels de procédures distantes (« *Remote Procedure Call* », ou RPC), ce qui fait du réseau d'interconnexion un facteur critique de performance. Parmi les systèmes micro-noyaux, on trouve MACH (et OSF/1), Amoeba, Chorus, Choices, Clouds, etc.
  La modularité de l'architecture micro-noyau facilite l'adaptation, l'extension, et la maintenance du système, de même que son implémentation en environnement distribué. Cependant, les chercheurs du projet de micro-noyau PEACE sont arrivés à la conclusion qu'un micro-noyau supportant le multiprocessus, même reconfigurable, pénalisait le fonctionnement d'une application mono-processus, et donc qu'il était préférable de disposer d'une famille de micro-noyaux distincts plutôt que d'un micro-noyau évolutif. Des méthodes de conception orientées objet permettent alors, par la définition de classes interchangeables, la définition aisée d'une famille de systèmes d'exploitation ; c'est le cas de Choices, Apertos, Clouds, etc.
– les systèmes orientés objet. à la différence des systèmes uniquement basés objet, les systèmes orientés objet permettent à l'utilisateur d'utiliser les mécanismes objet de définition par héritage, de renommage dynamique lors de l'appel des méthodes de l'objet, et de polymorphisme.
  Le support du modèle objet repose sur quatre concepts clés : le nommage, la protection, la synchronisation, et la reprise sur erreur. Chorus et Mach ne sont pas intrinsèquement des systèmes orientés objet, mais permettent d'implémenter des environnements de programmation orientés objet, tels que COOL pour Chorus et Avalon/C++ pour Mach.

5.3 Fonctionnalités

Un système d'exploitation de machine parallèle doit disposer des mêmes fonctionnalités que celles présentes dans un système monoprocesseur. Cependant, leur complexité est largement supérieure, du fait des contraintes fortes de performance à respecter, et des nouvelles fonctionnalités à prendre en compte. Parmi les problèmes spécifiques aux machines parallèles, on peut citer la gestion et la protection de grands espaces d'adressage, la prévention des interblocages, la gestion efficace d'entités asynchrones telles que les processus et les tâches légères, leur synchronisation, l'équilibrage de charge et la distribution des données, etc.

5.3.1 Gestion et ordonnancement de processus

Dans les systèmes d'exploitation traditionnels, à un processus correspond un domaine de protection et un espace d'adressage virtuel servant à l'exécution d'un unique flot d'instructions. De tels processus sont appelés « lourds », car la création et la destruction de tels processus sont coûteuses. Le parallélisme exprimé par ce moyen est à gros grain, et correspond rarement au niveau de granularité des

problèmes irréguliers.

La plupart des systèmes d'exploitation actuels découplent l'espace d'adressage et les flots d'instructions, permettant à plusieurs d'entre eux de partager le même espace. Ces tâches, dites tâches moyennement lourdes (« *middleweight threads* »), sont directement gérées par le noyau (on les appelle aussi « *kernel threads* »), et disposent de toutes les fonctionnalités système offertes aux processus lourds. De fait, la gestion de ces tâches se fait au moyen d'appels système lourds (POSIX Pthreads, par exemple), qui ne permettent pas de mettre en œuvre un parallélisme à grain fin.

Pour exprimer le parallélisme à grain fin sont apparues les tâches légères (« *light-weight threads* »), qui s'appuient sur des systèmes de poids lourd ou moyen, el laissent à la charge de l'utilisateur les fonctions d'ordonnancement. L'utilisateur peut ainsi définir la politique de gestion des tâches convenant le mieux à son application. C'est le cas des LWP et threads de SunOS et Solaris, des Cthreads de MACH, etc. Cette architecture à deux niveaux ne permet cependant pas aux tâches légères de réagir aux événements liés au noyau (préemption, interruptions I/O, ordonnancement des processus moyens, ...), ce qui empêche d'adapter le séquencement des tâches légères au fonctionnement du système. Plusieurs solutions ont été proposées, comme le report des événements système à l'ordonnanceur des tâches légères, ou la possibilité pour les tâches utilisateur d'influencer l'ordonnancement des tâches moyennes sur les processeurs (tel que le « *concurrency level* » des threads Solaris).

L'ordonnancement (« *scheduling* ») des processus influe grandement sur les performances des machines parallèles. Il s'agit de minimiser le temps de réponse moyen du système, en répartissant la charge (« *load balancing* ») de façon efficace (mais ce problème est NP-dur)

- l'ordonnancement statique est calculé lors du lancement du programme parallèle, et n'est jamais remis en cause. Il génère un faible surcoût, mais suppose que le comportement de l'application est stable dans le temps ;
- l'ordonnancement dynamique permet une évolutivité dans le temps convenant aux applications très irrégulières, mais alourdit beaucoup le code, du fait des mécanismes d'évaluation de la charge et de migration des données devant être implémentés, qui doivent parfois opérer de façon asynchrone, par threads ;
- le co-ordonnancement (« *coscheduling* », ou « *gang scheduling* ») a pour but de favoriser l'exécution simultanée de processus appartenant au même programme parallèle, ce qui est très utile dans le cas de processus coopératifs à grain fin et communiquant fréquemment. Cette technique est complexe, et soulève de nombreux problèmes, tels la préemption simultanée, l'attente des processus retardataires, etc.

5.3.2 Gestion de la mémoire

La gestion de la mémoire par les machines de type UMA est semblable à celle des machines uniprocesseur multiprogrammées. Un gestionnaire de mémoire convertit les adresses de l'espace virtuel des processus en adresses physiques, gère les défauts de page, et assure éventuellement les opérations de synchronisation si une page est accédée simultanément par plusieurs tâches.

Les machines de type NUMA et NORMA nécessitent des mécanismes plus évolués, qui s'appuient cependant souvent sur les gestionnaires de mémoire locaux, afin d'offrir un service de mémoire virtuellement partagée (« *Distributed Shared Memory* ») ; c'est le cas du CRAY T3D et de la SGI Origin, par exemple.

Les premières machines NUMA ne géraient que des caches locaux, et s'appuyaient sur une couche logicielle pour gérer la cohérence de la mémoire entre processeurs, en permettant toutefois à l'utilisateur d'influer sur la répartition des

données en mémoire pour augmenter la localité des accès ; c'etait le cas de l'Uniform System de la BBN Butterfly.

Les architectures NUMA récentes comme le T3D offrent maintenant une mémoire virtuellement partagée globalement cohérente, mais proposent toujours des instructions matérielles de pré-chargement et de post-écriture afin d'optimiser leur performance.

Les recherches actuelles sur la gestion de la mémoire dans les architectures distribuées portent sur la possibilité de décharger le programmeur du placement explicite du code et des données, en s'appuyant sur les similitudes existant avec la gestion des caches sur les machines UMA. Plusieurs politiques de gestion ont été étudiées et implantées dans des systèmes tels que Mach OSF/1, Psyche, Platinum, etc :

- migration : les données sont migrées vers la mémoire locale du processeur qui les référence, afin de tirer parti le plus possible de la localité des références pour amortir le coût de la migration ;
- duplication en lecture : afin de permettre à plusieurs processus de lire localement les mêmes données, on duplique celles-ci sur tous les lecteurs qui en font la demande. Cependant, les opérations d'écriture deviennent plus coûteuses, car il faut alors invalider ou mettre à jour les copies des données sur tous les processeurs qui en possèdent. Des mécanismes matériels peuvent être utilisés pour optimiser les écritures, comme c'était le cas avec les mécanismes de diffusion et d'invalidation des machines KSR. La relaxation des contraintes de cohérence forte permet également de gagner en vitesse, si les caractéristiques de l'application le permettent (accès à des données périmées).

5.3.3 Synchronisation

Lorsque des processus coopérants s'exécutent simultanément, des primitives de synchronisation sont nécessaires pour contrôler leur concurrence, en particulier afin d'assurer l'exclusion mutuelle et l'ordonnancement global d'événements. Ceci se fait principalement au moyen de verrous (« *locks* »).

Un verrou est un objet qui n'appartient qu'à un seul processus à la fois. Pour entrer en section critique, un processus doit d'abord acquérir le verrou qui lui est associée. Dans le cas contraire, il doit s'endormir (« *blocking lock* »), ou boucler en attente active (« *spin lock* »). Cette dernière solution, qui peut sembler onéreuse, se révèle plus efficace lorsque la section critique est petite ou que la machine est sous-utilisée, car on évite ainsi de coûteux changements de contexte, et on réduit la latence entre le moment où le verrou est libéré et celui où on en acquerra la propriété.

5.3.4 Systèmes de fichiers parallèles et distribués

Le parallélisme, en permettant de traiter des problèmes de grande taille, pose le problème du stockage et de l'accès aux données manipulées. Pour le résoudre, plusieurs solutions ont été proposées :

- la délégation des fonctions d'entrées/sorties à des processeurs spécialisés (ou des nœuds de la machine, pour les architectures NORMA). Dans ce cas, le système de fichiers est physiquement centralisé, et les requêtes issues des nœuds de calcul sont traitées par appel distant de procédure (RPC). Cette approche est simple à mettre en œuvre, mais tant le réseau que les nœuds disques peuvent constituer des goulots d'étranglement du système ;
- la distribution du stockage sur les nœuds de la machine, au moyen de disques locaux. Ceci suppose de pouvoir maintenir une vision cohérente des systèmes de fichiers, et de savoir sur quels disques se trouvent les différentes portions

des fichiers. En effet, afin de répartir la charge d'accès sur tous les processeurs, les fichiers sont découpés en blocs (« *disk stripping* ») qui sont distribués sur l'ensemble des disques des processeurs.
Des implémentations de ces mécanismes commencent à être proposées par les constructeurs (PIOFS sur la SP-2 d'IBM, par exemple), et une interface de programmation a même été normalisée dans le cadre de la norme MPI-2 (même si l'on s'éloigne quelque peu de la communication par échange de messages).

Bibliographie

[1] Y. Choi, A. Knies, L. Gerke, and T. Ngai. The impact of If-conversion and branch prediction on program execution on the Intel Itanium$^{\text{tm}}$ processor. In *Proceedings of the 34$^{\text{th}}$ Annual IEEE/ACM International Symposium on Microarchitecture*, pages 30–40, December 2001. `http://www.capsl.udel.edu/COMPILER/MICRO34/pdf/choi_y.pdf`.

[2] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computers*, 21(9) :948–960, 1972.

[3] J. R. Goodman. Cache memory optimization to reduce processor/memory traffic. Technical Report 580, University of Wisconsin–Madison, 1985.

[4] R. W. Hockney and C. R. Jesshope. *Parallel Computers – Architecture, programming and algorithms*. Adam Hilger, Bristol, 1983.

[5] W. Jalby and C. Lemuet. Exploring and optimizing Itanium2$^{\text{tm}}$ cache(s) performance for scientific computing. In *Proceedings of EPIC2*, pages 4–19, November 2002. `http://systems.cs.colorado.edu/EPIC2/`.

[6] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, 1994.

[7] J. Lee and A. Smith. Branch prediction strategies and branch target buffer design. *IEEE Trans. Computers*, 21(7) :6–22, 1984.

[8] P. Michaud. *Chargement des instructions sur les processeurs superscalaires*. Thèse de Doctorat, IRISA, Université Rennes I, November 1998.

[9] OpenMP Fortran Application Program Interface. `http://www.openmp.org/`.

[10] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). Research Report 87/391, CS Division, University of California at Berkeley, 1987. Disponible à partir de l'URL `ftp://sunsite.berkeley.edu/pub/techreps/CSD-87-391.html`.

[11] S. Raina. Virtual shared memory : A survey of techniques and systems. Research Report CSTR-92-36, Department of Computer Science, University of Bristol, December 1992. Disponible à partir de l'URL `http://www.cs.bris.ac.uk/Tools/Reports/Abstracts/1992-raina.html`.

[12] R. Rakvic, E. Grochowski, B. Black, M. Annavaram, T. Diep, and P. Shen. Performance advantage of the register stack in Intel Itanium$^{\text{tm}}$ processors. In *Proceedings of EPIC2*, pages 30–40, November 2002. `http://systems.cs.colorado.edu/EPIC2/`.

[13] `http://www.top500.org/`. Site recensant les systèmes installés les plus puissants au monde.

[14] D. W. Wall. Limits of instruction-level parallelism. Research Report 93/6, DEC Western Research Laboratory, November 1993. Disponible à partir de l'URL `http://www.research.digital.com/wrl/techreports/`.