HABILITATION A DIRIGER DES RECHERCHES

présentée devant

l'Université Paul Sabatier de Toulouse

par

Pascal Sainrat

Parallélisme niveau tâche et parallélisme niveau instruction

Soutenue le 26 novembre 1997 devant le jury

Alain Greiner	Professeur à l'université de Paris VI	Rapporteur
Bernard Tourancheau	Professeur à l'université de Lyon 2	Rapporteur
André Seznec	Directeur de recherche à l'IRISA	Rapporteur

Daniel Etiemble	Professeur à l'université de Paris-Sud
Bernard Lécussan	Professeur à l'ENSAE
Daniel Litaize	Professeur à l'université Paul Sabatier de Toulouse

Remerciements

La recherche est-elle un travail de solitaire ? Si c'est l'image qu'on en donne le plus souvent dans la littérature ou dans les films[1], ce n'est pas comme cela que je la conçois. Pour moi, la recherche se pratique par l'échange, par la discussion, par la réflexion commune. Et, même si, par moments, une réflexion solitaire amène une idée, c'est souvent par son exposé et la confrontation à d'autres points de vue que cette idée progresse.

En conséquence, il me faudrait remercier beaucoup de gens ici. Je n'aime pas beaucoup l'exhaustivité qui entraîne souvent la lassitude à moins d'adopter un style flamboyant, ce qui n'est pas ma spécialité. Je m'en tirerai donc en remerciant tous les gens avec qui j'ai eu le loisir de travailler[2] ou d'échanger des idées et dont je ne fais pas suivre les noms de peur d'en oublier.

Enfin, je me dois de remercier les organismes qui nous ont financés : le MENESR et le CNRS pour leur soutien récurrent, le GDR-PRC "Architecture de Machines Nouvelles", le réseau doctoral des jeunes chercheurs en architecture des systèmes informatiques, la région Midi-Pyrénées, l'ANVAR et le programme inter-PRC "Nouvelles technologies pour l'architecture".

[1] Voir, par exemple, Dr Jekyll et Mr Hyde

[2] Certains pourront trouver cette formule un peu curieuse et pourtant tout travail que l'on aime peut devenir un loisir, non ?

Table des matières

1. PARALLELISME NIVEAU TACHE 11

1.1. MODELE DE PROGRAMMATION.....11

1.2. ARCHITECTURE DES MACHINES A MEMOIRE PARTAGEE.....11

1.2.1. Généralités 11

1.2.2. L'architecture UMA.....14

1.2.3. L'architecture NUMA 15

1.2.4. L'architecture COMA 16

1.3. L'ARCHITECTURE M3S 17

1.3.1. Architecture générale - Le réseau d'interconnexion 17

1.3.2. Maintien de la cohérence 20

1.3.3. Evaluation de performances.....22

1.3.4. Réalisations matérielles 25

1.4. CONCLUSION.....30

2. PARALLELISME AU NIVEAU INSTRUCTION..... 33

2.1. INTRODUCTION.....33

2.2. (TRES) COURTE INTRODUCTION A L'ARCHITECTURE DES PROCESSEURS SUPERSCALAIRES A EXECUTION NON ORDONNEE.....35

2.3. CARACTERISTIQUES DES PROGRAMMES D'EVALUATION.....36

2.3.1. Introduction.....36

2.3.2. Répartition par types d'instructions 37

2.3.3. Parallélisme niveau instruction disponible dans les programmes 38

2.3.4. Limitation de l'exploration pour extraire le parallélisme.....39

2.4. LA FENETRE D'INSTRUCTIONS.....40

2.5. LE RETRAIT 41

2.6. UNITES D'EXECUTION.....42

2.7. TOPOLOGIE DU TAMPON 44

2.8. ACQUISITION DES INSTRUCTIONS.....46

2.8.1. Prédiction de branchement pipeline.....47

2.8.2. Prédiction de plusieurs blocs.....47

2.9. LES ACCES AUX DONNEES EN MEMOIRE 49
2.9.1. Introduction 49
2.9.2. Modèle utilisé 51
2.9.3. Mécanismes de renommage 51
2.9.4. Prédiction de dépendances aveugle 53
2.9.4.1.1.1.1.1.1 Prédiction d'adresse 53
2.9.5. Mécanismes de récupération 54
2.9.6. Prédiction d'adresse et dépassement spéculatif 55
2.9.7. Exécution 56

3. CONCLUSION 59

4. BIBLIOGRAPHIE 63

Introduction

Une large part de la recherche en architecture parallèles a concerné la réduction du coût de la communication tel qu'il est vu par le processeur : réduction du surcoût et de la latence des communications, accroissement de la bande passante et mécanismes permettant d'éviter la contention et de masquer la communication par du calcul ou d'autres communications. Une partie de ces thèmes se retrouve tout au long de ce document avec, bien entendu, la meilleure exploitation possible du parallélisme qui englobe cet aspect communication.

Le parallélisme peut être exploité en utilisant la structure parallèle des données (calculateurs vectoriels, SIMD, SPMD).

En réalité, ce que l'on exploite est le fait que les données sur lesquelles travaillent chaque processeur sont indépendantes des données traitées par les autres processeurs. La structure parallèle des données permet de détecter aisément cette absence de dépendances. Dès qu'il y a une dépendance, les éléments de calcul doivent communiquer entre eux et c'est cette communication qui est généralement le maillon le plus faible dans l'architecture d'une machine. On tente de limiter ces communications, lorsque c'est possible, en partageant le traitement en tâches les plus indépendantes possibles et donc, en exploitant un parallélisme de haut niveau. L'exécution de ces tâches peut alors se faire sur une machine MIMD. Naturellement, comme le problème à traiter est unique, les tâches ont toujours besoin de s'échanger des informations.

L'indépendance des données peut aussi être exploitée de manière beaucoup plus fine si on sait la détecter dynamiquement. Ainsi le pipeline que l'on trouve dans les processeurs applique des traitements différents à chaque étage et fonctionne bien tant que les données traitées à chaque étage sont indépendantes. De même, dans les processeurs superscalaires, plusieurs unités de calcul peuvent fonctionner en parallèle du moment que les données traitées sont indépendantes. Dans un processeur, le choix des données à traiter est donné par l'ordre séquentiel du programme et ce choix est fait dans une fenêtre d'instructions de taille restreinte. Dans une machine parallèle, l'indépendance des données est plus explicite, qu'elle soit donnée par la structure de ces données ou par des instructions spécifiques de partage des données (verrous).

Ce document traite de ces deux niveaux de parallélisme : tâche à travers l'architecture de la machine M3S et instruction à travers diverses études sur les processeurs superscalaires à exécution non ordonnée.

Afin d'exploiter le parallélisme au niveau tâche, un problème est décrit par une collection de tâches qui sont exécutées sur une collection de processeurs. Ces tâches ont besoin de se transmettre des informations, ce qui peut être fait par l'émission explicite de messages ou en utilisant des variables partagées.

La programmation par passage de messages nécessite de la part du développeur de prendre en compte les communications lors de la programmation pour assurer un bon fonctionnement de l'algorithme et, même en présence de bibliothèques de communication (PVM, MPI), demande un effort certain, sauf pour quelques algorithmes où cette approche est assez naturelle. Les bibliothèques de communication ont permis d'améliorer la portabilité de ce type d'applications au prix d'une perte de performance. Le programmeur aura intérêt à tenter de limiter les communications au maximum en raison de leur mauvaise influence sur le temps d'exécution d'un programme et, pour cela, à souvent prendre en compte la topologie du réseau d'interconnexion de la machine. Par contre, la mise au point de ces programmes est relativement aisée.

A contrario, la programmation par variables partagées permet de s'abstraire de l'architecture sous-jacente au prix d'une architecture plus complexe. Le portage de ces applications est simple mais la mise au point peut s'avérer plus complexe du fait de cette abstraction.

L'échange de messages est le seul mode de programmation possible sur une machine à mémoire distribuée (sauf en présence d'une mémoire virtuellement partagée qui est une solution de partage de la mémoire par logiciel qui présentent des latences de communication importantes). L'espace d'adressage n'étant pas unique, la réalisation d'une telle machine est moins coûteuse puisque chaque nœud peut être un élément standard, une station de travail par exemple. La communication étant explicite, elle est effectuée via des cartes d'entrées/sorties qui sont le seul élément spécifique de la machine. L'inconvénient majeur est que les dispositifs d'entrées/sorties ont généralement de grands temps de latence, ne serait-ce que parce que les mécanismes de protection sont logiciels.

Une architecture supportant les variables partagées fournit un espace d'adressage unique et doit donc assurer une cohérence des données partagées. Assurer cette cohérence est assez complexe pour des machines extensibles et nécessite un nombre de messages important et donc une grande bande passante. Les algorithmes de cohérence ont été l'objet de nombreuses recherches visant à diminuer les communications nécessaires pour assurer celle-ci [20] [61]. Pour des machines de petite taille, l'architecture la plus standard est fondée sur un bus partagé qui sert de point de synchronisation et des cartes processeur intégrant un espion de bus.

Toutefois, les bus standard existants ont une bande passante fixe et trop faible pour permettre la réalisation d'une machine à mémoire partagée de grande taille. D'autres solutions doivent donc être envisagées et, d'un point de vue matériel, on distingue deux types de machines suivant que la mémoire est physiquement centralisée ou distribuée. La première partie de ce document reviendra sur les différentes approches matérielles pour réaliser une machine à mémoire partagée.

Le parallélisme au niveau instruction consiste à exécuter plusieurs instructions en parallèle en profitant du fait que ces instructions manipulent des données n'ayant pas d'interaction entre elles au moment où on va les traiter. Le modèle de programmation peut alors être séquentiel et le parallélisme extrait automatiquement par la machine.

Cette extraction matérielle du parallélisme prend deux formes. D'une part, les instructions sont traitées en pipeline. Chaque étage du pipeline effectue une partie du traitement d'une instruction. Cela nécessite un minimum d'indépendance entre les données manipulées par les instructions à chaque étage du pipeline. Ainsi, une instruction ne peut lire ses opérandes que si ceux-ci ne sont pas produits par une ou plusieurs instructions qui n'ont pas encore calculé leur résultat. Garantir l'indépendance de ces données pour une meilleure utilisation possible du pipeline et donc offrir plus de parallélisme peut être réalisé en partie par le compilateur en réordonnant les instructions dans le programme. D'autre part, avec les processeurs superscalaires, plusieurs calculs peuvent être effectués simultanément grâce à plusieurs unités de calcul. Là aussi, les données manipulées doivent être indépendantes ce qui peut être garanti par le compilateur (approche VLIW) ou réalisé dynamiquement lors de l'exécution en réordonnant les instructions pour exécuter les instructions qui peuvent l'être, c'est-à-dire celles dont les opérandes ne dépendent pas ou plus d'autres instructions. On a alors une exécution non ordonnée.

Ces deux niveaux de parallélisme peuvent, bien entendu, être exploités conjointement. Je me suis intéressé au parallélisme de tâches (ou à gros grain) à travers l'architecture du multiprocesseur à mémoire partagée M3S (entre 1991 et 1994) puis à l'exploitation du parallélisme niveau instruction dans les processeurs (entre 1994 et 1997) qui a des retombées (en termes de bande passante par exemple) sur l'architecture d'un multiprocesseur. C'est une des raisons pour lesquelles je me suis intéressé aux architectures superscalaires à exécution non ordonnée afin de comprendre les implications de ce type d'architecture de processeur. Ce sujet est abordé dans la deuxième partie de ce document.

1. Parallélisme niveau tâche

Les multiprocesseurs à mémoire partagée servent à améliorer le débit pour des charges de travail multiprogrammées ainsi qu'à accélérer l'exécution de programmes parallèles. Nous nous intéressons plus particulièrement à ce deuxième aspect bien, qu'en termes de marché, le premier soit prépondérant. Toutefois, l'accélération de l'exécution des programmes parallèles constituent une niche où le besoin existe et ce qui peut être mis en œuvre pour cette accélération peut, bien souvent, être également utile dans le cadre de charges de ttavail multiprogrammées.

1.1. Modèle de programmation

Le modèle de programmation est à peu près le même que pour une exécution en temps partagé sur un monoprocesseur mais un parallélisme réel remplace l'entrelaçage de l'exécution dans le temps. Les processus peuvent être configurés pour partager une partie de l'espace d'adressage c'est-à-dire qu'ils pointent sur les mêmes emplacements en mémoire. La coopération entre les processus est assurée du fait qu'une écriture effectuée par un processus sera vue par les autres processus s'ils lisent le même emplacement en mémoire. La communication est donc assurée par les opérations usuelles de lecture et d'écriture en mémoire et la synchronisation par des opérations atomiques spéciales. Ce modèle logiciel de variables partagées facilite le développement rapide de logiciel de traitement parallèle [29] et assure une portabilité tout aussi rapide. Bien entendu, certaines applications se prêtent très bien à une approche de programmation par messages mais elles peuvent aussi être exécutées sur une architecture à mémoire partagée, les messages étant alors implémentés par le biais de variables partagées. L'architecture à mémoire partagée, objet de ce chapitre, semble donc une solution particulièrement intéressante en ce qui concerne son modèle de programmation.

1.2. Architecture des machines à mémoire partagée

1.2.1. Généralités

Les points critiques en matière de multiprocesseur sont la bande passante du réseau d'interconnexion, qui doit pourvoir aux besoins des processeurs, et la latence des accès à la

mémoire, qui doit être la plus faible possible. Ces deux points ne sont pas indépendants puisqu'un déficit en bande passante entraîne un accroissement des temps de latence du fait des conflits d'accès au réseau. Il parait donc raisonnable de résoudre d'abord le problème de la bande passante avant de s'attaquer à celui de la latence. Un autre aspect à caractère plus commercial est l'extensibilité. Pour couvrir les frais d'étude d'une machine à mémoire partagée, il semble nécessaire que l'architecture puisse être vendue dans des configurations variées. Il n'est question ici que de machines prévues pour de larges configurations, en effet, les multiprocesseurs à bus commun limités à quelques processeurs représentent un marché important et ont des coûts de développement plus faibles.

De manière générale, le problème de la bande passante offerte aux processeurs est résolu en mettant en œuvre un réseau adéquat. Toutefois, un compromis doit être réalisé entre le coût du réseau et sa bande passante. Le critère d'extensibilité est également prépondérant.

Le réseau le plus courant est le bus partagé. Il a pour avantage d'être simple mais son extensibilité est très limitée. Le bus a une bande passante fixe et, de ce fait, l'ajout de processeurs est limité à un nombre donné. Par contre, dans la limite du nombre maximum de processeurs, l'ajout d'un processeur est très simple. Il ne nécessite aucune modification du réseau. Quelque soit le nombre de processeurs présent dans une configuration de machine, celle-ci doit être vendue avec le bus supportant le nombre de processeurs maximum. Le coût du bus devient alors prépondérant pour les petites configurations. De plus, une machine a généralement une durée de vie bien supérieure à celle des processeurs. De ce fait, le réseau doit être prévu pour supporter les générations suivantes du processeur envisagé au départ. Ainsi, avec l'évolution des processeurs, le nombre de processeurs que l'on pourra mettre sur un bus donné va diminuer avec le temps. Le bus partagé n'est plus guère d'actualité compte tenu de l'évolution de la bande passante demandée par les processeurs. Aujourd'hui, quelques processeurs suffisent à saturer un bus partagé. La solution du bus partagé a été communément adoptée dans les machines commercialisées à petit nombre de processeurs (jusqu'à 36 dans la Challenge de SGI) car il a l'avantage de présenter un point de synchronisation et d'observation des échanges qui permet de garantir simplement la cohérence des données dans les caches. Il faut noter que le réseau de la Challenge, bien qu'étant spatialement dimensionné pour 36 processeurs au départ, n'est en réalité capable d'alimenter guère plus de huit processeurs aujourd'hui. Or, si le bus est dimensionné pour supporter une trentaine de processeurs, l'achat d'une petite configuration à 4 processeurs est onéreux car le coût du bus représente une part importante de celui de la machine. Aujourd'hui, un bus commun n'est guère capable de servir plus de quatre processeurs et même, on commence à trouver des machines à faible nombre de processeurs dont le réseau est un crossbar [62].

Pour un nombre de processeurs important, la solution crossbar n'est guère envisageable du fait que son coût évolue en n^2, n étant le nombre de modules à connecter et que le nombre de broches des composants du crossbar constitue une limite technologique. Des solutions

intermédiaires sont possibles avec les réseaux multi-étages qui offrent une bande passante moindre mais ont un coût plus raisonnable. On trouvera dans [1] ou dans [17] une étude comparative des réseaux d'interconnexion. Ces divers réseaux constituent un compromis entre l'extensibilité, leur coût et la latence des accès à la mémoire qui est souvent plus grande avec ces divers réseaux qu'avec un crossbar ou un bus du fait que plusieurs étages de logique doivent être traversés.

Le problème du temps de latence est partiellement résolu par l'utilisation de mémoires caches qui sont systématiquement utilisées aujourd'hui. Les mémoires caches sont des mémoires rapides placées auprès de chaque processeur. Comme ce sont des mémoires rapides, elles sont chères et donc de petite taille (de 256 KO à 1 MO aujourd'hui). Elles contiennent les données qui sont les plus utilisées par le processeur et ces données migrent dynamiquement de la mémoire vers un cache et inversement suivant leur utilisation. Les données migrent sous la forme de blocs plus ou moins grands (16 à 32 octets aujourd'hui), ce qui permet de profiter de la localité spatiale des accès aux données (si un processeur référence une donnée à l'adresse x, il y a de fortes chances pour qu'il accède aux données x+1, x+2, ... dans un avenir plus ou moins proche). La présence de ces mémoires caches a permis de diminuer le temps de latence moyen des accès à la mémoire et le débit nécessaire par processeur. Dans un multiprocesseur, chaque processeur possède sa ou ses mémoires caches, ce qui entraine un problème de cohérence des données. En effet, si un processeur A modifie la donnée présente dans son cache, il faut que cette donnée soit également modifiée dans les caches des autres processeurs ou, du moins, que les autres processeurs soient informés que la copie qu'ils possèdent n'est plus valable.

Dans le cas d'un bus partagé, cette cohérence est assurée simplement par les caches, ceux-ci pouvant espionner tous les accès aux bus et consulter leur répertoire pour vérifier s'ils possèdent la donnée qui va être modifiée. Il existe différentes versions d'algorithme de cohérence basés sur les espions de bus suivant, entre autres, le fonctionnement du cache lui-même.

Dans le cas de réseaux plus complexes, une telle solution n'est guère envisageable car il n'y a pas de point centralisé où les processeurs pourraient réaliser un espionnage. Dans ce cas, une solution est de réaliser un réseau dédié aux messages de cohérence, réseau qui permet de réaliser des diffusions afin d'informer tous les caches lors de l'écriture d'une donnée. Une autre solution consiste à adopter un algorithme de maintien de la cohérence des données à répertoire. De nmbreux algorithmes ont été proposés dans la littérature [60]. Le principe est de mémoriser dans un répertoire l'état de partage de tous les blocs et, éventuellement, qui possède une copie d'un bloc. Ainsi, lorsqu'un processeur veut modifier un bloc, ce répertoire doit être consulté pour prévenir les possesseurs du bloc. Tout ceci est réalisé de manière matérielle et automatique. Les algorithmes proposés dans la littérture visent à assurer un minimum de messages de cohérence sur le réseau et/ou à minimiser la taille du répertoire.

Trois grandes familles d'architectures matérielles mettant en œuvre une architecture à mémoire partagée existent aujourd'hui. On les distingue usuellement par leurs sigles américains : NUMA (Non Uniform Memory Architecture), COMA (Cache Only Memory Architecture), UMA (Uniform Memory Architecture).

1.2.2. L'architecture UMA

L'architecture UMA, la plus ancienne, tente de présenter un accès le plus uniforme possible à toute la mémoire de manière à ce que le programmeur d'une application parallèle ait le moins possible à se préoccuper de l'architecture sous-jacente.

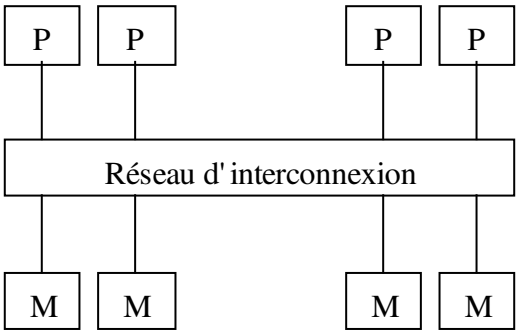

Figure 1.Architecture de type UMA

La façon la plus simple de la réaliser est d'avoir une mémoire unique (centralisée) à laquelle tous les processeurs sont reliés à travers un bus partagé. L'évolution de la performance des processeurs a été beaucoup plus rapide que celle des mémoires. Pour améliorer la bande passante des mémoires, une solution classique consiste à découper la mémoire en modules entrelacés. Cela suppose que les accès sont uniformément répartis sur les modules de mémoire, ce qui est favorisé par un entrelaçage suffisamment fin (en général de la taille maximale des données échangées). Diverses solutions ont été mises en œuvre pour que le réseau ait également une bande passante qui satisfasse les besoins. Ainsi, le transfert de l'adresse et de la donnée peuvent, par exemple, être effectués en pipeline ou, plus généralement, si la mémoire et les caches le permettent, de manière complètement indépendante. Cette approche paraît assez naturelle dans la mesure où l'information qui transite sur les bus n'est pas de même nature : une adresse comportant moins de 64 bits et un bloc de données en comprenant plusieurs centaines. Adopter un réseau unique pour ces deux transferts conduit donc soit à sous-utiliser le réseau si le chemin de données est aussi large qu'un bloc soit au risque de le saturer s'il est de la largeur d'une adresse. Il semble donc nécessaire de différencier le réseau destiné aux adresses (aux requêtes plus exactement) et celui destiné aux données. On peut même éventuellement envisager la présence d'un troisième réseau, destiné à certaines informations concernant la cohérence des données ou de natures diverses, qui aurait aussi des caractéristiques différentes du fait de besoins différents.

1.2.3. L'architecture NUMA

L'intérêt majeur de l'architecture à mémoire partagée, appelée NUMA du fait que la mémoire est physiquement distribuée auprès des processeurs (un processeur a donc un accès privilégié au module mémoire qui lui est associé), réside dans son extensibilité.

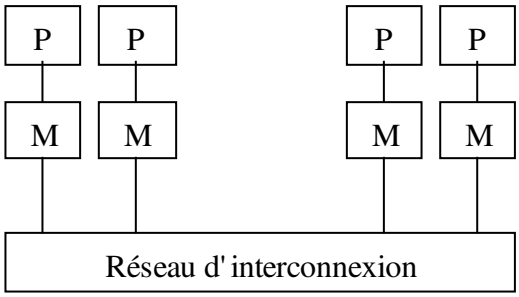

Figure 2.Architecture de type NUMA

Pour peu que le réseau d'une telle machine soit extensible (une grille 2D par exemple ou toute structure géométrique régulière), l'ensemble des caractéristiques de la machine évoluent avec le nombre de processeurs et le rapport entre le besoin et l'offre en bande passante est gardé. Ceci suppose bien entendu que les applications aient des besoins qui évoluent proportionnellement au nombre de processeurs, autrement dit que l'application soit, elle-aussi, extensible. Lié à la notion d'extensibilité de l'architecture, le coût d'une machine NUMA est à peu près proportionnel au nombre de processeurs. La base de l'architecture NUMA est une carte standard telle qu'on la trouve dans une station de travail, ce qui rend le développement de l'architecture aisé et le coût peu élevé. Les meilleurs exemples de machines NUMA sont la machine universitaire DASH de Stanford [36] et son pendant industriel, l'Origin 2000 de SGI [54].

Ce type de machine à mémoire physiquement distribuée mais à espace d'adressage partagé tente de résoudre à la fois les problèmes de bande passante et de latence et repose sur l'idée que l'on peut découper une application en tâches manipulant majoritairement des données privées. Ces données doivent donc être placées auprès du processeur qui exécute la tâche qui les manipule. Les instructions sont elles dupliquées si le même programme est utilisé par plusieurs processeurs. La mémoire doit donc avoir une capacité plus grande que dans le cas d'une architecture UMA où ce code est unique. Le besoin en bande passante sur le réseau est limité aux seules variables partagées, il est donc plus faible que pour une architecture UMA et il doit donc être moins coûteux de construire un réseau extensible satisfaisant ces besoins. Les accès aux données locales ayant des temps de latence équivalents à ceux d'un monoprocesseur, le temps de latence moyen des accès dans une telle machine n'est pas dégradé si les données sont en grande majorité placées auprès des tâches qui les utilisent. Par contre, le temps de latence des accès à des mémoires éloignées peut être très long. Il est donc impératif que les données soient locales et, par conséquent, que les données partagées soient peu utilisées, ce qui peut être

difficile à réaliser pour certaines applications. Ainsi, bien qu'on ait une architecture à mémoire partagée, le programmeur doit adopter certaines règles pour que son application puisse s'exécuter efficacement sur une telle machine, on perd ainsi un peu de la transparence des architectures UMA sans que la mise au point s'en trouve simplifiée. Pour éviter des accès lointains inutiles, des pages de mémoire peuvent être dupliquées dans des noeuds (le code par exemple) si elles sont en lecture seule et une page en lecture/écriture peut migrer d'un noeud à un autre, en fonction de son taux d'utilisation par tel ou tel nœud. Toutefois, ces migrations se faisant sur la base d'une page, il peut y avoir des problèmes de faux partage conséquents. De plus, la migration d'une page, même si celle-ci est de petite taille (quelques Kilo-octets), demande une bande passante non négligeable, ce qui doit limiter la migration aux cas où celle-ci est vraiment indispensable.

Les réseaux généralement choisis pour les machines NUMA étant généralement des grilles pour assurer l'extensibilité, la cohérence des données doit être maintenue par des protocoles à répertoires associés à la mémoire. C'est une solution assez complexe et relativement coûteuse qui nécessite des messages supplémentaires sur le réseau pour assurer cette cohérence et accroît les temps de latence pour certains accès à des mémoires se trouvant dans un autre nœud. Cray, dans sa machine NUMA T3E, a résolu le problème en empêchant la présence dans les mémoires caches de données provenant d'une mémoire d'un autre nœud. En matière de coût et de simplicité, c'est certainement efficace. Par contre, la performance doit s'en ressentir.

L'architecture NUMA est donc une très bonne candidate pour une machine multi-programmée mais peut révéler des faiblesses pour l'exécution d'applications parallèles si celles-ci présentent de forts taux de partages de données et donc utilisent intensivement le réseau.

1.2.4. L'architecture COMA

L'architecture COMA a essayé de pallier à ce problème de placement des données en amenant dynamiquement et matériellement les blocs de données au plus près des processeurs qui les utilisent.

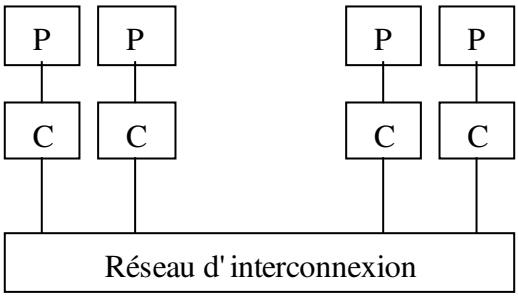

Figure 3.Architecture de type COMA

La différence majeure avec l'architecture NUMA vient de la taille des données qui migrent qui est beaucoup plus petite (typiquement la taille d'un bloc de cache), ce qui réduit les problèmes de faux partage sans toutefois les éliminer. L'idée est intéressante puisqu'elle réduit dynamiquement le temps de latence en rapprochant les données des processeurs qui les utilisent. C'est donc une extension du principe de la mémoire cache mais généralisée, toute la mémoire étant considérée comme une mémoire cache. Toutefois, la difficulté de réalisation, alliée au fait que certaines données comme les verrous des barrières de synchronisation ne peuvent pas avoir d'emplacements privilégiés puisqu'elle sont utilisées de la même façon par tous les processeurs, ont fait retomber l'intérêt pour ces architectures. KSR1 [32] est un exemple industriel d'une telle machine et DDM [26] un exemple universitaire.

1.3. L'architecture M3S[3]

L'architecture M3S veut garantir que le temps d'accès à une donnée est du même ordre de grandeur quel que soit l'endroit où elle se trouve, ce qui est le mieux à même de garantir l'absence de prise en compte de l'architecture lors de la programmation. Elle s'inscrit donc pleinement dans le principe d'architecture de type UMA. Le réseau d'interconnexion doit être suffisamment dimensionné et extensible pour permettre des accès rapides à une mémoire plus ou moins éloignée du processeur et cette mémoire doit aussi offrir la bande passante nécessaire pour satisfaire la demande des processeurs. On considérera ici une mémoire "centralisée", c'est-à-dire non distribuée auprès des processeurs mais découpée en modules entrelacés afin de fournir une bande passante suffisante.

1.3.1. Architecture générale - Le réseau d'interconnexion

Comment avoir un réseau qui présente une bande passante suffisante pour supporter les demandes des processeurs ? Le réseau le plus performant est évidemment le réseau totalement interconnecté dans lequel tout élément peut communiquer avec n'importe quel autre élément pour peu que celui-ci ne soit pas déjà en communication avec un troisième élément. Usuellement, ce type de réseau n'est pas viable pour deux raisons : son coût et le nombre de broches d'interconnexion. L'architecture M3S résout au moins un de ces deux problèmes par l'utilisation de liaisons série. La réduction du nombre de points d'interconnexion apparaît de suite puisqu'une liaison série ne comporte que quelques broches alors qu'un lien parallèle nécessite un nombre de broches égal à sa largeur. Pour le coût, c'est moins évident même si un

[3] M3S pour Multiprocesseur à Mémoire Multiport Série. Cette architecture et le composant mémoire multiport série ont fait l'objet d'un brevet international [XXXI]

circuit crossbar de liaisons série est plus simple à réaliser du fait du nombre de broches nettement plus faibles.

Une liaison série est une liaison point à point et présente donc des caractéristiques électriques (adaptation) nettement meilleures qui permettent donc une fréquence plus grande qu'une liaison multi-point. Les liaisons série peuvent être asynchrones ; l'horloge est alors récupérée à partir du signal de données, on évite ainsi les problèmes de déphasage entre l'horloge et les données. Comme il n'y a qu'un signal de données, il n'y a pas de problèmes de jitter entre les signaux comme c'est le cas pour une liaison parallèle. Tout ceci a été l'objet d'études dans notre équipe dès 1989 et les résultats sont rapportés dans [XXVII] [XLII] [XVIII] [X]. Si on accorde aujourd'hui certaines qualités aux liaisons série, leur utilisation dans ce cadre était nouveau à cette époque. Il semble donc nécessaire de l'étayer par quelques chiffres.

Il y a quelques années, Dominique Carrière (qui a soutenu sa thèse en 1996) et moi-même avons réalisé une deuxième forme de liaison série[4], avec des composants bipolaires en AsGa disponibles sur le marché et fonctionnant à 0,8 Gbit/s. A cette époque, les bus étant à 20 MHz, leur bande passante était de l'ordre de 0,7 Gbit/s (pour une largeur de 32 bits). Une liaison série offrait la même bande passante. Aujourd'hui, les mêmes composants d'interface série fonctionnent à 1,5 Gbit/s et Sun a conçu une interface de liaison série en CMOS fonctionnant à 1,3 Gbit/s [43]. Si l'on regarde les machines multiprocesseurs existantes, l'Origin2000 de SGI [54] assure une bande passante crête de 0,8 Goctets/s par processeur. Il faut donc 4 liaisons série par processeur pour obtenir une bande passante équivalente. Toutefois, l'Origin2000 est une NUMA et la bande passante moyenne disponible pour les accès distants est quatre fois plus faible et correspond donc à un seul lien série. L'Enterprise 10000 de Sun Microsystems [62] a une bande passante de 12,8 Goctets/s pour un maximum de 64 processeurs, ce qui donne 200 Moctets/s par processeur. Une bande passante équivalente est assurée par un ensemble de 64 liens série. Une liaison série bidirectionnelle nécessite entre 5 et 10 broches tandis que les bus parallèles ont au minimum 70 broches (64 de données plus les lignes de parité et de contrôle). Les liaisons série sont donc encore une alternative viable pour le réseau d'interconnexion d'un multiprocesseur puisqu'un ensemble de liens série présente une bande passante supérieure aux solutions actuelles et est plus économique en termes de nombre de broches.

Pour satisfaire la demande en bande passante des processeurs, la mémoire doit être divisée en modules entrelacés fonctionnant de manière indépendante. Le rapport entre le nombre de processeurs et le nombre de modules mémoire est fonction des caractéristiques de ces deux derniers. Toujours est-il que la meilleure utilisation des liaisons série d'un point de vue performance est que chaque processeur possède une (ou plusieurs) liaison série vers chaque

[4] La première est abondamment explicitée dans ma thèse [XLII], je ne reviendrais pas sur ce travail dans ce document.

module mémoire. Ainsi, si l'on considère une machine à 64 processeurs et 8 modules mémoire, chaque processeur dispose de huit liens série à 1,5 Gbit/s soit une bande passante totale par processeur de 1,5 Giga-octet et une bande passante globale de plus de 700 Goctets/s. La comparaison avec les chiffres de l'Enterprise 10000 est éloquente.

Cet ensemble de liaisons série réalise une interconnexion totale comme le montre la figure suivante. Il n'y a donc aucune contention intrinsèque au réseau. Par contre, chaque module mémoire reçoit autant de liens qu'il y a de processeurs dans la machine, il y a donc contention au niveau des modules mémoire. De cette contention découle la présence de registres au niveau de chaque lien et un arbitrage pour l'accès de ces registres à la mémoire. Dans le cas de caches non bloquants, ces registres doivent être en fait des FIFOs dont la taille est égale au nombre de requêtes autorisées au niveau des caches donc de faible profondeur. Ces FIFOs qui constituent les interfaces des liens série fonctionnent de manière complètement indépendante. Seul l'accès à la mémoire est sérialisé. Dans cette architecture, la base des données échangées est le bloc de cache soit 16 octets à l'époque. Aujourd'hui ce serait plutôt 32. Pour qu'un module mémoire puisse satisfaire suffisamment de requêtes en provenance des processeurs, celle-ci présente un front de la largeur d'un bloc. Ainsi, en un seul cycle, tout le bloc est transféré de la mémoire vers l'interface de la liaison série concernée. Une telle largeur de bus n'est pas un problème dans la mesure où ce bus reste interne au module.

A chaque cycle, un cache (qui doit être non bloquant) peut envoyer une requête vers l'un des modules mémoire. A un instant donné, plusieurs requêtes peuvent être en cours vers le même ou vers différents modules mémoire. On peut, bien entendu, imaginer différentes alternatives à ce modèle. Par exemple, plusieurs processeurs peuvent partager un cache L2 et donc des liens série, en fonction de la bande passante demandée par les processeurs et de celle offerte par les liens série. Mais, aujourd'hui, la solution la plus vraisemblable est que chaque processeur ait son propre cache L2. Par contre, la mémoire peut être divisée en un plus grand nombre de bancs. On peut alors imaginer que plusieurs processeurs ayant chacun leur cache L2 se partagent un jeu de liens série. Un réseau doit alors être implémenté au niveau de chaque module processeur mais le nombre de liens du réseau global est alors plus faible.

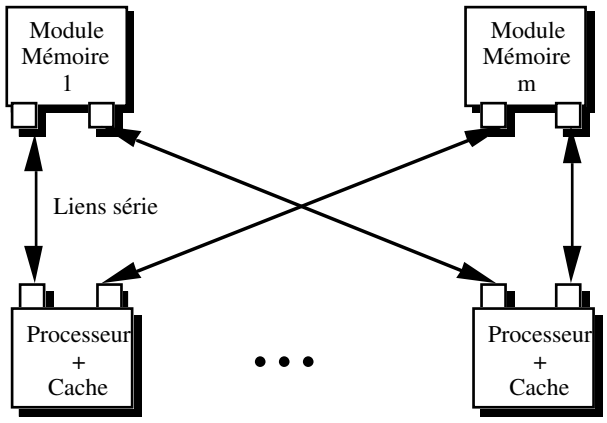

Figure 4.Base de l'architecture M3S

Cette architecture de base est la source de nombreuses autres solutions.

1.3.2. Maintien de la cohérence

Dans l'architecture de base ci-dessus, qui ne comporte qu'un réseau de liens série à très haut débit, la cohérence ne peut être assurée que par un protocole à répertoire. Il y a essentiellement deux types de répertoire suivant que l'on mémorise, pour chaque bloc, tous les processeurs pouvant posséder un bloc ou seulement quelques-uns. Dans les deux cas, des diffusions partielles doivent être effectuées pour mettre à jour l'état de plusieurs copies d'un bloc et dans le second, une diffusion totale est nécessaire si le bloc est possédé par plus de processeurs qu'il n'y a de pointeurs dans le répertoire. Le maintien de la cohérence peut donc nécessiter de nombreux messages sur le réseau et ces messages sont envoyés par l'unité de gestion du répertoire qui est située au niveau de la mémoire. A noter que, comme la mémoire est entrelacée, le répertoire est également distribué sur les modules mémoire, il est donc extensible.

1.3.2.1. Bus d'adresse partagé

Pour une machine comportant peu de processeurs, la présence d'un bus commun a l'énorme avantage d'être un point de synchronisation qui permet une gestion classique et peu coûteuse de la cohérence des données dans les caches par un protocole à espion de bus. Le problème est que le bus devient très rapidement un goulet d'étranglement lorsque le nombre de processeurs augmente. Une machine uniquement interconnectée par des liens série a besoin d'un algorithme de cohérence à répertoire, solution relativement coûteuse. Nous avons donc suggéré la présence de deux réseaux. Un bus commun ne sert qu'à transmettre les adresses et le type des accès à la mémoire et tous les processeurs espionnent ce bus. Ainsi, la cohérence est assurée par un protocole à espion de bus, solution parfaitement extensible puisque chaque processeur ajouté contient son propre espion. Les liaisons série, elles, n'assurent que le transport des données de manière privée. Un avantage qui découle d'une telle architecture est une plus grande simplicité des interfaces des liaisons série côté mémoire puisqu'elles n'ont pas à détecter si ce qui est reçu est une adresse ou une donnée. Par ailleurs, cette solution correspond bien au principe de transactions éclatées, la différence étant que le canal de transmission n'est pas le même pour les données et les adresses. Bien entendu, une telle architecture est moins extensible qu'une architecture totalement interconnectée par des liens série puisque basée sur un bus commun. Toutefois, un bus qui ne passe que les adresses peut être surdimensionné sans que son coût soit prohibitif puisque la bande passante nécessaire par processeur est plus faible. La taille maximale d'une telle machine est fonction du rapport entre le débit nécessaire pour le passage des adresses et la bande passante offerte par le bus partagé. Bien évidemment, un anneau comme SCI est également envisageable à ce niveau.

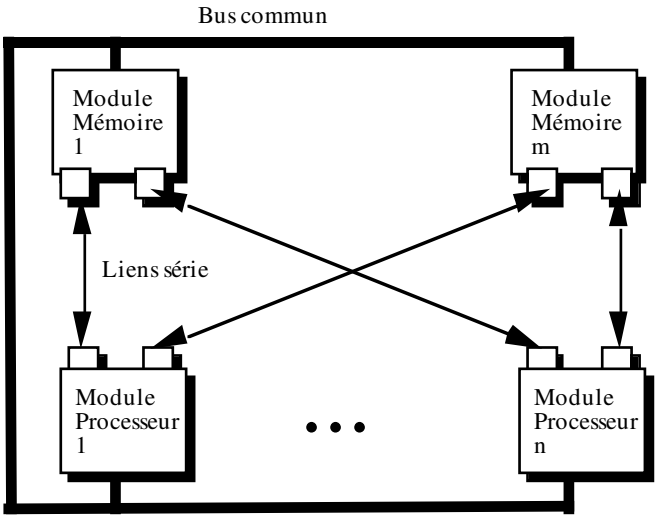

Figure 5. Architecture à bus partagé pour les adresses

1.3.2.2. Réseau spécifique

La nécessité de réaliser des diffusions nous a conduit à considérer l'ajout d'un réseau spécifique pour la cohérence. Ce réseau est en fait un ensemble liens, chaque lien reliant un module mémoire à tous les processeurs comme le montre la figure ci-dessous. Chaque lien est donc un bus multipoint monodirectionnel. Cette architecture est plus extensible que le bus partagé puisque la bande passante du réseau de cohérence est proportionnelle au nombre de modules mémoire qui, lui-même, doit être proportionnel au nombre de processeurs. Avec cette solution, si les caches sont bloquants, les liaisons série bidirectionnels à l'alternat et non full-duplex comme dans l'architecture de base. Même avec des caches bloquants, il est possible de garder des liens fonctionnant à l'alternat pour économiser le nombre de câbles mais c'est au prix d'une bande passante moindre car il faut réserver une tranche de temps sur chaque lien pour effectuer l'arbitrage de l'accès au lien.

D'un point de vue bande passante, cette architecture est extensible puisque tout ajout d'un élément entraîne l'ajout de liens dans le réseau. La base de l'extensibilité est le module mémoire, c'est-à-dire que si un module mémoire est capable de fournir la bande passante pour *n* processeurs, l'architecture est incrémentale sur la base de *n* processeurs. D'un point de vue coût, le démonstrateur que nous avons réalisé est modulaire, c'est-à-dire que l'on peut ajouter des ports au fur et à mesure des besoins par l'ajout de cartes filles. Je ne suis toutefois pas certain qu'une telle solution soit viable d'un point de vue industriel. En général, l'industriel préfère éviter réaliser une seule carte configurée pour la solution maximale car les cartes filles impliquent des connecteurs dont la fiabilité est médiocre si l'on veut garder un coût raisonnable. Le coût de petites configurations peut donc se retrouver grevé par le prix de base des modules.

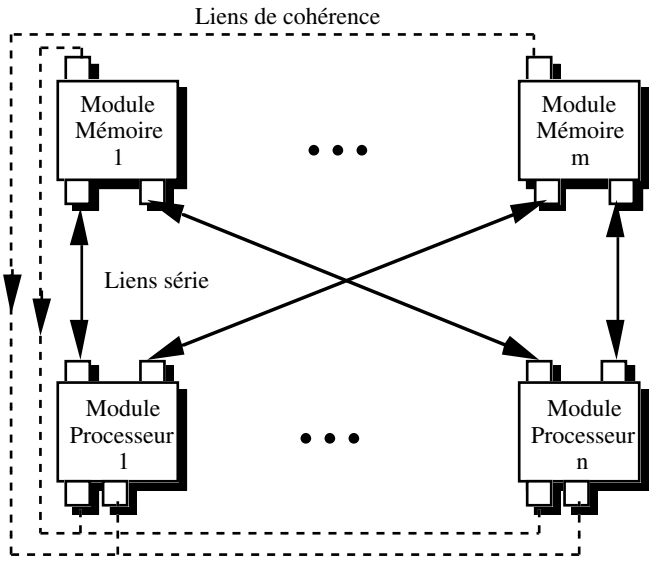

Figure 6.Architecture à réseau de cohérence

M3S est donc une famille de machines dont la caractéristique commune est l'utilisation d'une interconnexion totale à base de liaisons série à très haut débit. Les liens série permettent une meilleure extensibilité du fait du faible nombre de broches nécessaires pour chaque lien. L'interconnexion totale permet un parallélisme maximum au niveau du réseau et profite pleinement des possibilités offertes par les caches non bloquants. Les démonstrateurs qui ont été réalisés pour cette architecture et qui fonctionnent sont la liaison série synchrone puis asynchrone, la liaison de cohérence et le chemin de données d'un module mémoire ce qui constitue toute la partie haute fréquence de la machine. En parallèle, une évaluation de performance a été menée à l'aide de réseaux de files d'attente.

1.3.3. Evaluation de performances

Les premières évaluations de M3S ont été faites en utilisant des techniques de modélisation par réseaux de files d'attente. Le problème de ce type d'évaluation est que les résultats ne sont guère précis (ce point sera évoqué plus loin) et surtout qu'ils manquent de crédibilité. Par ailleurs, si ces résultats apportent des éléments sur les performances de la machine, ils ne prouvent en rien son bon fonctionnement. Seules la simulation et la réalisation d'une maquette permettent de montrer la validité d'une machine et de ses performances. La réalisation a bien entendu un impact plus important et peut laisser espérer un passage vers l'industrie. Mais elle

nécessite au préalable une phase de simulation qui permet d'affiner les paramètres de l'architecture et de déceler certains problèmes, ce qui allège d'autant la mise au point de la maquette.

1.3.3.1. Simulation par réseau de files d'attente

La simulation de M3S par le modèle du débit maximum puis par des réseaux de files d'attente a été menée par Christine Rochange lors de sa thèse [50]. Ce travail est rapporté ici car j'y ai participé à travers de nombreux échanges et discussions qui se sont traduits par l'écriture en commun de quelques articles.

Ces simulations visaient à analyser les performances de M3S en fonction des critères architecturaux choisis.

Dans un premier temps, nous avons utilisé un modèle très simple proposé par Dubois dans [12] et appliqué au cas de l'exécution d'un algorithme SOR de résolution d'équations différentielles partielles. Ce modèle a permis de montrer que, pour un nombre de modules mémoire donné, il existait un nombre optimal de processeurs et qu'au delà de ce nombre, la performance chutait [VI]. Bien entendu, ce modèle ne donne que des limites supérieures de débit et est extrêmement simpliste. Il a constitué une première expérience en la matière.

Nous avons par la suite décrit M3S par un réseau de files d'attente que nous avons résolu par Analyse de la Valeur Moyenne. Cette méthode de résolution présente l'avantage d'être très rapide puisqu'elle est analytique mais elle ne peut être appliquée qu'au prix d'un grand nombre d'hypothèses simplificatrices dans la conception du modèle. Ainsi, certains comportements de la machine ne peuvent pas être représentés de manière satisfaisante. Néanmoins, cette méthode a permis d'analyser les performances de divers algorithmes de cohérence à répertoire suivant le type d'application et le taux de partage de données.

Enfin, nous avons choisi la simulation à événements discrets pour résoudre les réseaux de file d'attente. Ce type de résolution est nettement plus lent que la résolution par analyse de la valeur moyenne (plusieurs heures contre quelques secondes) mais permet de décrire la machine de manière plus proche de la réalité. Nous avons défini 3 types de charges de travail suivant la classification des objets manipulés présentée dans [67]. De manière très synthétique, ces 3 charges sont définies par leur influence sur le maintien de la cohérence des données qui est due au type des données principalement utilisées qui ont des comportements particuliers en termes de taux de présence, de taux de lectures, de nombre d'invalidations générées, … Pour chacune de ces charges, nous avons considéré entre 10 et 40 % de données partagées, ce qui représente la majeure partie des applications. Nous avons ainsi observé que, dans certains cas, ce n'était ni la mémoire ni le réseau qui limitait la performance mais les caches qui étaient submergés par le nombre de messages de cohérence. Les liaisons série étaient alors sous-utilisées ce qui suggérait

de les utiliser aussi pour les messages de cohérence en adoptant des liaisons full-duplex, voire de les partager entre plusieurs processeurs.

Nous souhaitions également comparer notre architecture à une architecture NUMA qui, nous l'avons vu, a la réputation d'être extensible. Nous avons choisi la machine DASH de Stanford qui constitue une référence dans le domaine et dont les caractéristiques ont été largement publiées. Nous avons modélisé les deux architectures (DASH et M3S) par réseaux de files d'attente, avec des paramètres technologiques équivalents pour les deux machines (excepté les liaisons série qui n'existent que dans M3S). La similitude de nos résultats avec ceux publiés par Stanford [37] aussi bien par Analyse de la Valeur Moyenne que par simulation à événements discrets ont permis de valider cette approche. Suivant les types d'application, les deux machines présentent à peu près les mêmes comportements à savoir qu'elles souffrent plus ou moins d'un taux élevé de partage des données. Dans tous les cas, DASH présente de meilleures performances si les données sont bien réparties auprès des processeurs (donc si le programmeur a pris en compte l'architecture sous-jacente) mais, pour certaines applications, les performances de M3S sont très proches de celles de DASH même avec des données bien réparties. Par contre, si les données sont réparties aléatoirement (donc pas forcément auprès des processeurs qui les utilisent), les performances de DASH sont nettement dégradées et sont inférieures à celles de M3S.

Il est évident que représenter une architecture par un réseau de files d'attente et lui appliquer une charge de travail aléatoire ne peut pas donner de résultats aussi précis que développer un modèle algorithmique complet d'une machine et lui faire exécuter du code. Les techniques basées sur les réseaux de files d'attente présentent néanmoins l'intérêt d'être plus simples à mettre en œuvre et nous avons montré que l'on pouvait obtenir des résultats assez fiables. Elles ont cependant un gros défaut : elles sont peu crédibles, les architectes de machines préférant, de loin, la simulation par programme.

1.3.3.2. Simulation en VHDL

Le manque de crédibilité des méthodes analytiques n'est pas complètement injustifié. Ces méthodes donnent des résultats corrects au niveau de l'ordre de grandeur mais elles ne modélisent entièrement le fonctionnement de la machine décrite. C'est une des raisons qui nous a amené à considérer une simulation complète de la machine. De fait, lors de l'écriture du simulateur, nous nous sommes posé un certain nombre de questions que nous ne nous étions pas posées auparavant, en particulier au niveau de la consistance mémoire, ce qui a amené Dominique Carrière à examiner de manière plus formelle ce point avec Mamoun Filali et Jean-Paul Bodeveix.

Ont travaillé avec moi sur ce projet : Eric Peladan (stagiaire CNAM), Matthieu Poujol (DEA), Yvon Réquier (DEA), Eric Sicard (stagiaire IEE).

1.3.4. Réalisations matérielles

1.3.4.1. La liaison série asynchrone

L'étude de cette liaison est une suite directe du travail effectué durant ma thèse [XLII] où j'avais montré qu'il était possible de réaliser un réseau de liaisons série synchrones à 800 Mbit/s avec une horloge centralisée à 400 MHz. Toutefois, j'avais souligné que cette approche n'était pas viable dans un contexte industriel pour diverses raisons technologiques dont, en particulier la nécessité d'ajuster la longueur des câbles de distribution de l'horloge. De plus, cette expérience n'était pas reproductible, les composants ayant disparu du marché. Nous avons donc décidé de faire une seconde liaison mais, cette fois, asynchrone. Le caractère asynchrone évite les problèmes susmentionnés et permet d'utiliser une horloge locale de faible fréquence, un VCO étant intégré au circuit d'interface série pour constituer une horloge haute fréquence. Le principe consiste à recouvrer l'horloge à partir des données grâce à un encodage de celles-ci, des messages de synchronisation étant transmis en l'absence de données. Fort heureusement, des composants intégrant une interface série à 800 Mbit/s existaient et existent toujours [28] car une réalisation discrète du recouvrement d'horloge est très délicate. Sa réalisation n'a pas présenté de difficultés majeures de réalisation si ce n'est celles qui sont liées au très haut débit que j'avais déjà explorées lors de la réalisation de la première carte série et qui sont résolues par des lignes adaptées 50 ohms, un découplage et une alimentation très soignés (composants CMS), la séparation des parties haute fréquence et basse fréquence sur la carte et un refroidissement suffisant. Elle a été abondamment testée et a révélé un taux d'erreur extrêmement faible (de l'ordre de 10^{-14}).

Cette réalisation nous a permis de comparer les approches synchrones et asynchrones. L'approche asynchrone a l'avantage d'une réalisation plus simple, la complexité étant incluse dans le circuit d'interface. Par contre, elle ne permet pas de profiter pleinement de la bande passante du medium (un câble coaxial dans notre cas) car le recouvrement d'horloge à partir des données nécessite un encodage de celles-ci représentant environ 20% de débit supplémentaire dédié à cette tâche. Une meilleure solution passe peut-être par une horloge centralisée telle que je l'ai réalisée mais en ajoutant un recouvrement de phase rapide. A priori, le recouvrement de phase est plus simple et ne devrait pas nécessiter d'encodage mais seulement la présence d'un en-tête avant transmission de données. Le seul problème qu'il semble nécessaire de résoudre est celui de la métastabilité qui est très dépendant de la technologie utilisée.

Le principe de fonctionnement de cette carte série correspond à un multiplexage temporel de 16 signaux à 50 MHz vers une liaison série à 800 Mbit/s et inversement. Côté basse fréquence,

les échanges sont assurés par un protocole de type *poignée de mains*. L'horloge est générée de manière interne dans le composant d'interface.

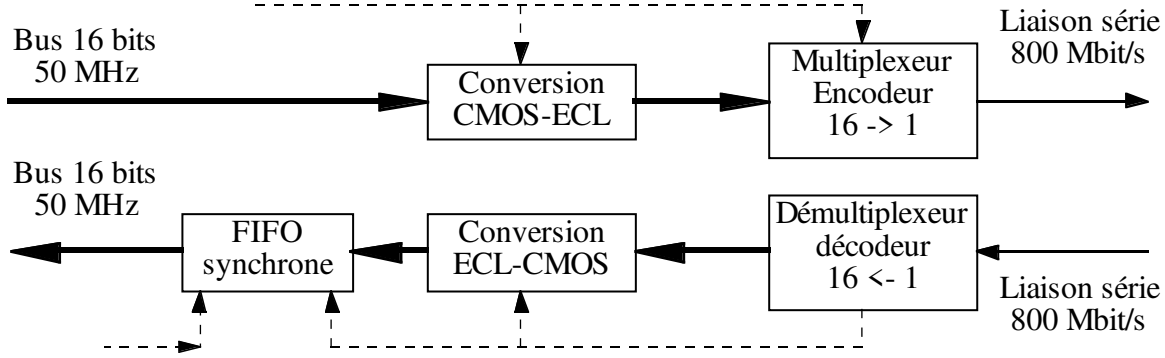

Figure 7. Schéma-bloc de l'interface de la liaison série haut débit

Cette liaison série a été réalisée dans le cadre de la thèse de Dominique Carrière sous ma direction et est décrite dans [XXXIII].

Il faut souligner les progrès énormes réalisés par les industriels pour fournir des solutions pouvant être mises en œuvre assez facilement, ces progrès étant liés essentiellement aux besoins des réseaux avec la norme Fiber Channel. Bien entendu, cette liaison série peut être implémentée en fibre optique moyennant l'adjonction des convertisseurs adéquats. Depuis la réalisation de cette carte sont apparues sur le marché des macro-cellules permettant d'intégrer une ou plusieurs interfaces séries CMOS 800 Mbit/s dans un composant [35][41]. Ainsi, en ce qui concerne l'architecture M3S, les cartes d'interface pourraient être remplacées par un seul composant offrant plusieurs ports série. Cette intégration apporterait une sérieuse baisse du coût de l'architecture la rendant encore plus viable économiquement.

1.3.4.2. La liaison de cohérence

Dans la version de M3S sans bus commun c'est-à-dire celle que nous avons envisagée pour la réalisation d'un prototype, la gestion de la cohérence se fait par répertoire. Chaque module mémoire dispose d'un répertoire qui gère la cohérence de la mémoire associée. A cet effet, il est nécessaire que la mémoire envoie des messages aux processeurs, par exemple pour confirmer à un processeur qu'il est bien le seul à posséder un bloc et invalider ce même bloc dans les caches des autres processeurs. Pour éviter des diffusions sur les liens série et donc une multiplication des messages, nous avons choisi d'avoir un lien spécifique unidirectionnel allant d'un module mémoire à tous les modules processeurs. Ainsi, la diffusion est effectuée économiquement puisqu'un seul message assure la diffusion. Pour respecter le critère d'extensibilité, chaque module mémoire possède son propre lien.

L'existence de circuits Taxi prévu pour FDDI nous a amené à réaliser aussi cette liaison sous forme série. Toutefois, elle est moins rapide (175 Mbit/s) que les liaisons décrites

précédemment par souci d'économie et parce que seuls les messages de cohérence y transitent. Nous avions estimé que la bande passante nécessaire serait de l'ordre du dixième de celle nécessaire pour les données à l'aide du modèle du débit maximum de Dubois [12].

Contrairement aux liens série précédents, cette liaison possède plusieurs destinations.C'est ce qui nous a amené à la réaliser car les problèmes d'ordre technologique ne sont pas les mêmes. On retrouve ici les problèmes de réflexion et de charge capacitive que l'on trouve sur les bus et qui sont un facteur de limitation de la fréquence de fonctionnement et de l'extensibilité. Pour résoudre ce problème, nous avons décidé de régénérer le signal sur chaque processeur. Cette solution induit un temps de latence un peu plus grand (minime toutefois par rapport au temps de consultation du répertoire par exemple) puisque deux couches logiques doivent être traversées sur chaque carte mais apporte une fiabilité indéniable. La présence d'une FIFO sur les cartes de réception assure l'absence de pertes de message du fait du temps de prise en compte de ces messages par le processeur même si ces messages ont la priorité au niveau de l'accès au répertoire du cache sur les autres requêtes.

Cette liaison et ses interfaces ont été réalisées et testées par François Baverez (stagiaire ENSICA) sous ma direction et sont décrites dans [XLIII].

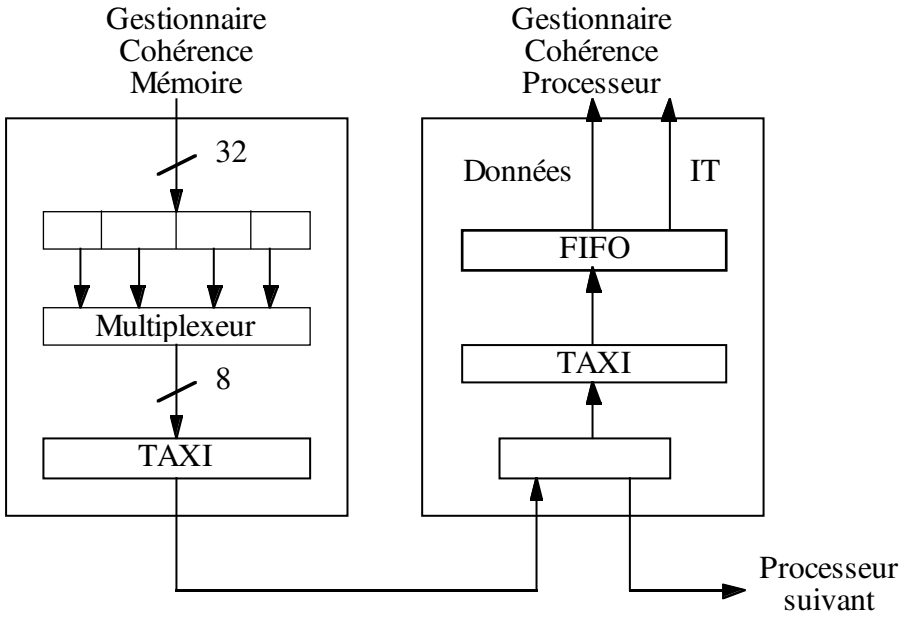

Figure 8.Schéma de principe de la liaison de cohérence

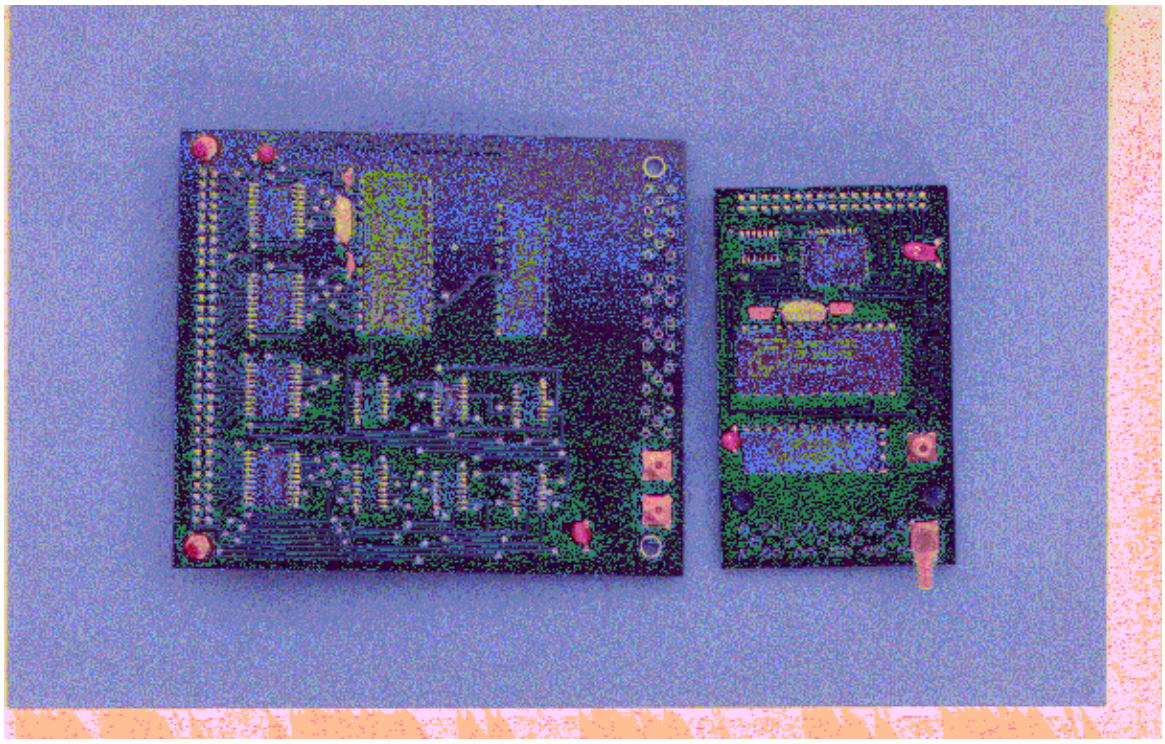

Figure 9. Les cartes d'émission et de réception de la liaison de cohérence

1.3.4.3. Le module mémoire multiport série

La définition de l'architecture du module mémoire multiport série date de l'époque où a été rédigé le brevet déjà mentionné [XXXI]. Nous voulions montrer à travers la réalisation de ce module la faisabilité de l'architecture, évaluer sa complexité et prouver sa viabilité. Nous ne l'avons réalisé que partiellement par manque de temps et surtout de main d'œuvre. Par contre, ce module mémoire a été entièrement simulé (au plus bas niveau) avec un logiciel de CAO. La partie qui a été réalisée (les zones grisées sur la figure ci-dessous) ne concerne que le flot de données provenant et repartant par le lien série haut débit, la partie contrôle ayant été réduite au strict nécessaire pour garantir le test. Il faut dire que le choix de réaliser la machine la plus extensible possible n'était sans doute pas le bon. Un prototype à bus et liaisons série aurait certainement été plus simple à réaliser et aurait aussi bien permis de valider l'approche.

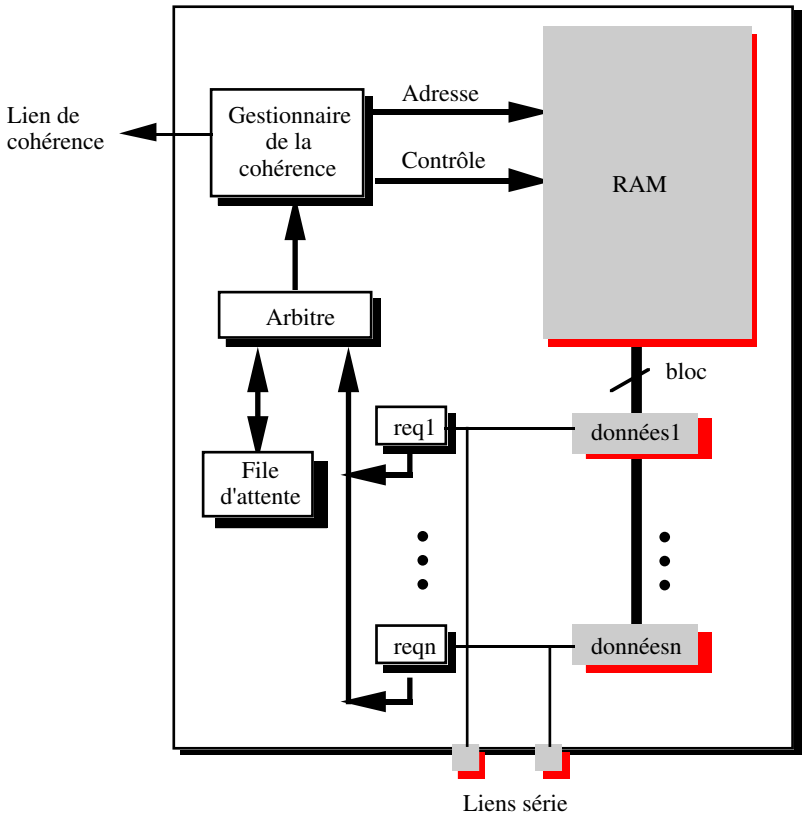

Figure 10.Schéma bloc du module mémoire

A chaque cycle mémoire, l'arbitre sélectionne une requête qu'il soumet au gestionnaire de cohérence. Celui-ci effectue diverses actions en fonction de l'état du bloc, noté dans le répertoire. Si la copie à jour du bloc est présente en mémoire, le bloc est envoyé au processeur demandeur via le registre de données correspondant. Si la copie du bloc en mémoire n'est pas à jour (c'est un processeur qui possède la copie à jour), la requête est mise en attente dans la file d'attente jusqu'à ce que la copie à jour soit renvoyée par le processeur qui la possède suite à un ordre envoyé par le gestionnaire de cohérence sur la liaison de cohérence pour ce processeur et ce bloc. Lorsque la purge sera effective et que le gestionnaire de cohérence aura noté l'exécution de cette requête, il sélectionnera au cycle suivant la requête en attente dans la file d'attente.

La partie réalisée comprend la DRAM et des registres de données fonctionnant à 50 MHz. Comme l'interface de la liaison série est constituée d'un bus 16 bits parallèle, la partie donnée du module mémoire est divisée en 16 plans identiques, chaque plan fournissant un octet à chaque cycle. La taille d'un bloc est donc de 16 octets (taille correspondant à celle des blocs du cache bien sûr) et il est transmis en 8 cycles d'horloge (à 50 MHz, les données étant transmises sur les deux fronts de l'horloge).

La réalisation de ce module a été faite, sous ma direction, par Dominique Carrière au cours de sa thèse. Une description détaillée du module mémoire peut être trouvée dans [9], [VIII], [VII].

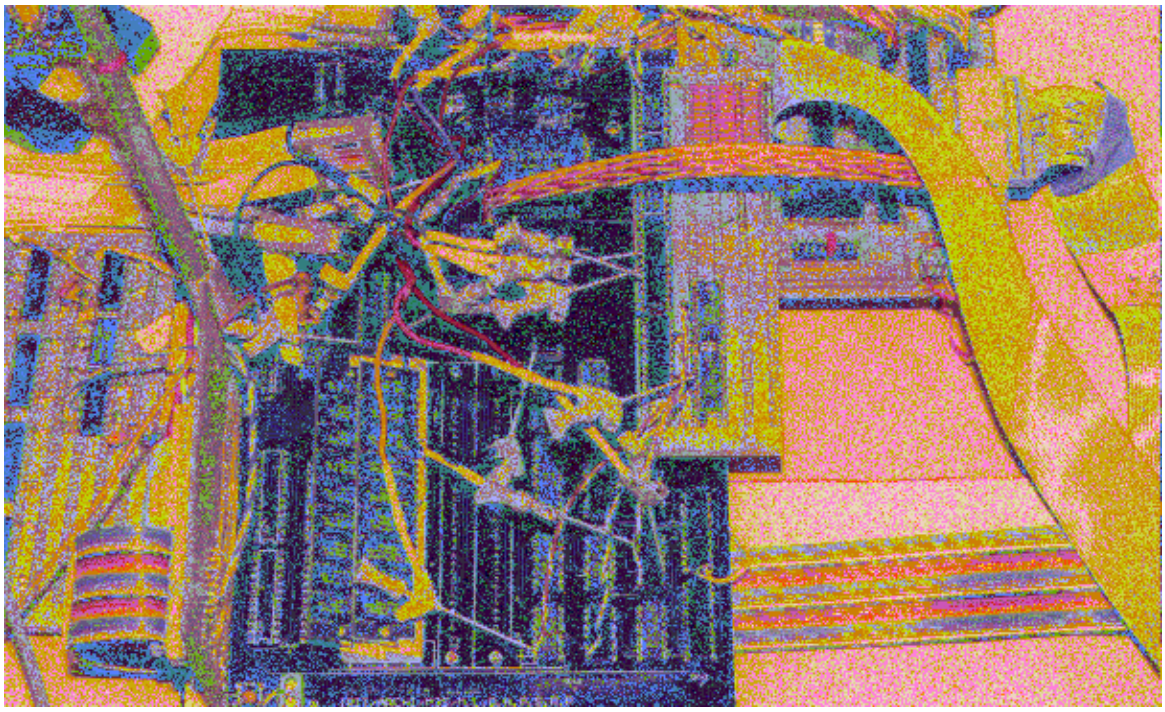

Figure 11.Une partie du module mémoire (avec ses cartes de test)

Bien entendu, cette maquette a été conçue dans un contexte universitaire et seule une réalisation intégrée d'une mémoire multiport série peut être intéressante d'un point de vue industriel. Cette intégration pourrait être faite aujourd'hui avec les macro-cellules de Bull [41]. Seul le chemin de données serait intégré, le reste de la logique pouvant être inclus au sein d'un contrôleur mémoire. Ainsi, il nécessiterait deux ASICs, un pour le gestionnaire de cohérence et tout ce qui y est associé et un autre pour huit interfaces de liaisons série et les registres attenants.

1.4. Conclusion

A travers ces évaluations et ces expérimentations, nous avons montré que l'architecture M3S est viable et qu'elle peut présenter de meilleures performances qu'une architecture NUMA en présence d'un fort taux de données partagées du fait qu'elle privilégie le débit global et assure un temps de latence identique quelle que soit la donnée adressée. Un reproche qui peut être fait à cette architecture est justement qu'elle ne résoud pas spécifiquement le problème de la latence. En fait, l'architecture M3S tente de limiter au mieux les conflits dans le réseau en les reportant au niveau des modules mémoire.

Par ailleurs, cette architecture nécessite un gros investissement dans la réalisation de composants mémoire sur mesure, en particulier pour l'intégration des ports série bien que la mise sur le marché de macro-cellules ait réduit ce coût. Il n'en reste pas moins que, par rapport à une architecture NUMA, une telle architecture nécessite un effort de développement matériel important que les constructeurs ne sont sans doute pas prêts à faire actuellement (encore qu'IBM a dû investir énormément pour réaliser le routeur de la SP2 [SP2]). A propos de ce routeur, il

est intéressant de noter qu'une des idées majeures de M3S est la réduction du nombre de liens, en augmentant leur fréquence et donc en adoptant des liaisons point à point, tout en gardant des bus larges de manière locale pour garder des fréquences et donc un coût faibles. Ceci vise à simplifier le réseau d'interconnexion. Or, c'est bien ce qui est fait dans le routeur de la SP2 qui, de manière externe, présente des liaisons de 8 bits de large tandis qu'une conversion est faite dans le circuit pour fonctionner sur 32 bits. Il en est de même pour SCI par exemple.

L'architecture des multiprocesseurs de demain va énormément dépendre de la façon dont seront réalisées les prochaines générations de processeurs. Il faut distinguer trois types de machines. Les multiprocesseurs comportant moins d'une dizaine de processeurs vont sans doute devenir monnaie courante et ils seront très probablement construits à l'aide d'un bus partagé ou d'un crossbar. Les multiprocesseurs comportant quelques dizaines de processeurs devraient couvrir le marché des serveurs (intranet ou non). Il semble que la solution NUMA soit en passe d'être largement adoptée pour ce type d'architecture. Enfin, dans le domaine du calcul scientifique, il est difficile de prévoir l'avenir compte tenu que les coûts de développement des architectures et des applications ne sont pas vraiment un critère prépondérant. Toutefois, ces hypothèses ne sont valables que pour le court terme. A plus long terme, bien des questions se posent. Comment utilisera t-on les milliards de transistors qui seront disponibles sur une puce ? Est-ce qu'on continuera à concevoir des processeurs de plus en plus complexes ? Intégrera t-on la mémoire dans le même boîtier que le processeur ou aura t-on plusieurs processeurs se partageant un gros cache L2 (ou L3) ? Toutes ces questions nous ont amené à étudier d'un peu plus près l'architecture des processeurs, objet de la deuxième partie de ce document.

2. Parallélisme au niveau instruction

2.1. Introduction

Les processeurs constituent un domaine de recherche à part entière qui a énormément progressé ces dernières années dans le sens d'un plus grand parallélisme au niveau instruction. Cette progression repose sur la formidable évolution technologique (techniques de lithographie, procédé de réalisation, etc) qui permet d'intégrer toujours plus de transistors sur une puce et, en même temps d'obtenir des fréquences de fonctionnement toujours plus élevées (voir la figure ci-dessous). Cette augmentation du nombre de transistors a entrainé l'exploitation du parallélisme au niveau instruction depuis les années 80 et l'avènement des architectures superscalaires à exécution non ordonnée depuis les années 90. Cette évolution reflète la volonté d'exploiter au mieux et le plus possible le parallélisme au niveau instruction en fonction du nombre de transistors disponibles. Cette recherche s'inscrit pleinement dans cette évolution.

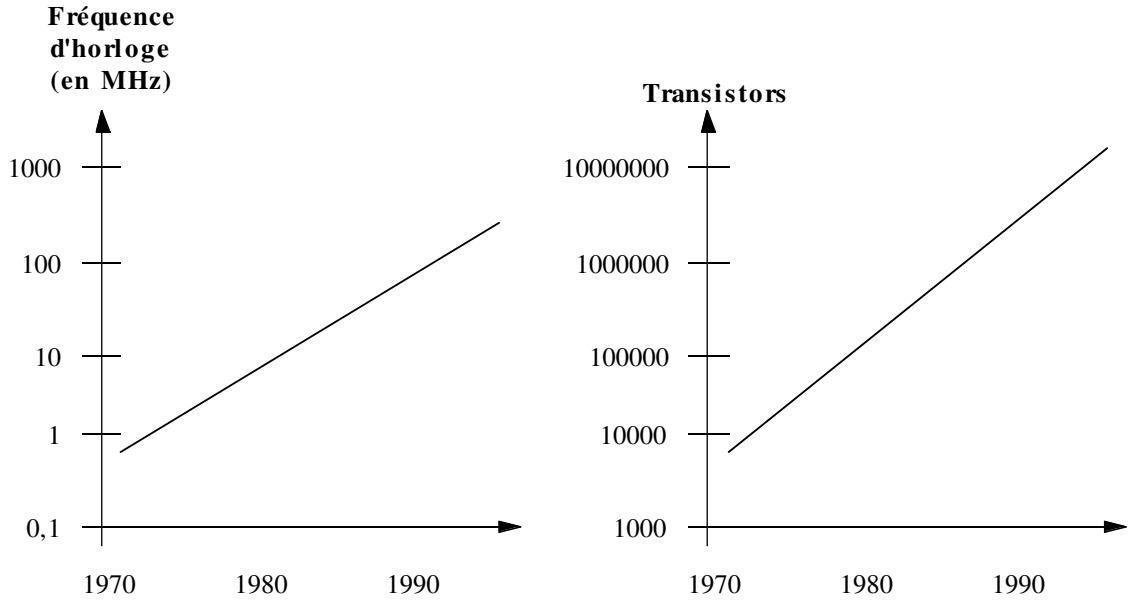

Figure 12. Evolution de l'intégration et de la fréquence d'horloge des microprocesseurs

Cette étude est, au départ, une étude quantitative visant à définir différents paramètres de l'architecture des processeurs superscalaires à exécution non ordonnée. Mais plutôt que de se fixer un budget en transistors, nous souhaitions partir des applications pour arriver à une architecture la mieux adaptée pour exploiter le parallélisme présent dans les programmes. Nous nous sommes toutefois toujours inquiété d'obtenir le meilleur compromis entre la performance obtenue et le coût de la logique à mettre en œuvre.

Après une étude des programmes d'évaluation utilisés, on distinguera l'étude des configurations des unités fonctionnelles du processeur (ce qui sert réellement à obtenir le résultat finalement). Ensuite, nous nous sommes intéressé aux structures de mémorisation de ces instructions en amont des unités fonctionnelles. Enfin, encore en amont, il est nécessaire de charger les instructions depuis la mémoire et nous avons proposé une solution pour le chargement simultané de plusieurs blocs d'instructions. Pour terminer, nous regardons actuellement le chargement des données car, à ce niveau, se pose un problème de latence des accès à la mémoire.

Fort de notre expérience en matière de simulation, nous avons décidé d'emblée de nous limiter à des simulations dirigées par des traces d'exécution de programmes ce qui est tout à fait envisageable pour l'étude de monoprocesseurs. Ce n'est pas le cas pour l'étude d'un multiprocesseur "massivement" parallèle où de telles traces n'existent pas et ne peuvent pas être générées simplement ou encore présentent des distorsions non acceptables. Pour l'étude d'un processeur, il suffit de récupérer la trace de l'exécution d'un programme que l'on a préalablement annoté. Nous avons choisi le jeu d'instructions MIPS car le programme Pixie permettant de récupérer les traces existe dans le système IRIX ; seule ombre au tableau, les appels système eux-mêmes ne sont pas tracés mais c'est généralement le cas dans les études publiées.

Nous souhaitions obtenir des traces complètes de l'exécution des programmes SPEC92 pour coller le plus possible à la réalité. Pour cela, nous avons du nous contenter des jeux d'entrée minimaux afin de limiter la taille des traces et le temps de simulation.

Ces études ont été essentiellement menées par Stéphan Jourdan au cours de sa thèse et y ont contribué dans une moindre mesure David Guignel, Dominique Carrière, Karl Alexandre et Pham Tuong Hai pour la partie concernant le chargement des données. Le travail sur le chargement des instructions a été effectué en collaboration avec André Seznec et Pierre Michaud de l'IRISA dans le cadre d'un contrat d'étude du PRC "Architecture de Machines Nouvelles". Par ailleurs, un nombre très important de simulations ont été réalisées pour cette étude et n'auraient pu être menées sans la présence à l'IRIT de trois serveurs financés par le réseau doctoral des jeunes chercheurs en architecture de machines.

2.2. (Très) courte introduction à l'architecture des processeurs superscalaires à exécution non ordonnée

Dans un processeur superscalaire à exécution non ordonnée, une instruction est traitée en 4 phases, chacune de ces phases pouvant être l'objet de plusieurs étages pipelinés [27] [57].

1. Chargement et décodage
2. Lancement (ou émission)
3. Amorçage et Exécution
4. Retrait

Ces phases sont séparées les unes des autres par des structures de mémorisation des instructions qui permettent de les découpler et de les mémoriser en attendant la résolution d'aléas qui leur permettront de passer à la phase suivante. Ces structures de mémorisation s'appellent le tampon d'émission entre les phases 1 et 2, le tampon d'amorçage entre les phases 2 et 3 et le tampon de réordonnancement entre les phases 3 et 4.

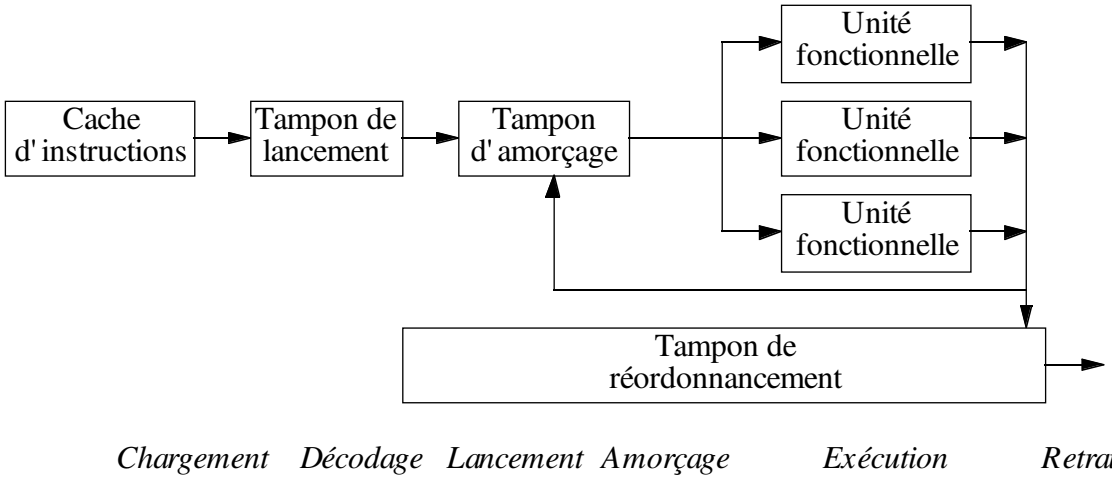

Figure 13.Architecture d'un processeur superscalaire à exécution non ordonnée

Le tampon de réordonnancement n'a pas ce seul rôle de tampon puisqu'il mémorise l'ordre des instructions c'est-à-dire que les instructions y sont introduites dès que l'ordre du programme n'est plus respecté (par exemple lors du lancement). Il y a donc une double mémorisation : les instructions présentes dans le tampon d'amorçage sont dupliquées dans le tampon de réordonnancement qui contient cependant moins d'informations et garde l'ordre des instructions afin d'effectuer le retrait de manière ordonnée pour assurer la validité de l'exécution du programme. En pratique, ces deux tampons (amorçage et réordonnancement) peuvent être

réunis dans la même structure, ce qui évite la redondance au prix d'un temps d'accès plus grand du fait que le tampon de réordonnancement a un grand nombre d'entrées. Usuellement, les instructions sont lancées dans l'ordre mais elles pourraient l'être dans le désordre si les instructions étaient insérées dans le tampon de réordonnancement dès la fin du décodage. On a alors aussi redondance entre les tampons de réordonnancement et d'émission.

Chacune de ces phases peut avoir des limitations qui peuvent provoquer des aléas interdisant aux instructions de passer à la phase suivante. Ces aléas provoqueront un éventuel blocage des phases précédentes si les structures de mémorisation interphases sont pleines. Par exemple, les instructions doivent attendre la disponibilité de leurs opérandes dans le tampon d'amorçage. Si celui-ci est plein d'instructions en attente d'opérandes, l'étage de décodage est bloqué, ce qui peut entraîner un blocage de l'étage de chargement. Nous nous sommes intéressés à ces limitations et aux structures de mémorisation. Mais avant cela, il était nécessaire d'étudier les caractéristiques des programmes d'évaluation que nous allions utiliser afin de déterminer les limites intrinsèques de ces programmes (paragraphe 2.3). Ensuite, nous verrons les sources de limitations depuis la fin du pipeline vers le début de celui-ci. En effet, s'il existe une limitation à la fin du pipeline, il ne servira à rien d'améliorer le début de celui-ci. Nous verrons donc successivement le retrait (paragraphe 2.5 page 41), les unités fontionnelles (paragraphe 2.6 page 42), le tampon d'amorçage (paragraphe 2.7 page 44) pour finir par le chargement des instructions (paragraphe 2.8 page 46). Enfin, le cas particulier des accès à la mémoire sera vu séparément.

2.3. Caractéristiques des programmes d'évaluation

2.3.1. Introduction

De nombreuses études ont été faites sur le parallélisme d'instructions existant dans les programmes [4] [8] [13] [33] [49] [65] [66]. Toutefois, chaque étude ne couvre pas tous les programmes que nous souhaitions utiliser ou bien toutes les données n'étaient pas disponibles dans les articles. Nous avons donc étudié les programmes d'évaluation issus de la suite SPEC92, qui constitue un standard en matière d'évaluation de processeurs [18] [59]. Afin de les utiliser au mieux, il était nécessaire de connaître pleinement leurs caractéristiques compte tenu du jeu d'instructions et du compilateur dont nous disposions. Nous avons donc déterminé la composition de ces programmes en termes de pourcentages de types d'instructions (calcul entier, branchements, calcul flottant, etc.), le parallélisme disponible, la longueur du chemin critique, ... On peut trouver les résultats de cette étude dans [XXII] [XXXVIII].

Une étude similaire sur les SPEC95 a été effectuée (*DEA de Karl Alexandre, juin 1997*) mais cette fois avec des jeux d'entrée normaux et en vue d'un éventuel échantillonnage de traces pour limiter les temps de simulation. Les traces générées par les SPEC95 étant énormes, nous ne mémorisons plus les traces générées par Pixie [56] comme nous le faisions pour les SPEC92

mais nous les régénérons pour chaque simulation sur une machine MIPS tandis que un ou plusieurs simulateurs les consomment en s'exécutant sur d'autres stations suivant un modèle client/serveur. Cette approche résoud le problème de stockage au prix d'un taux important d'utilisation du réseau. Nous avons montré sur une petite application que cette approche semble viable. Il reste à le confirmer en grandeur nature, c'est-à-dire sur tous les mécanismes présents dans un processeur (prédiction de branchement, TLB, etc.).

2.3.2. Répartition par types d'instructions

L'analyse des SPEC92 nous a permis de confirmer la présence, dans les programmes entiers, de 20% de branchements dont la majeure partie sont conditionnels. Nous avons trouvé environ 25% d'instructions de chargement dans ces mêmes programmes. La proportion de branchements est beaucoup plus faible dans les programmes flottants (8%) tandis qu'on y trouve 30% de chargements. Le nombre de branchements présents dans le programme confirme donc la nécessité absolue de la prédiction de branchement et montre, surtout pour les programmes entiers, que l'exécution spéculative au delà de quelques dizaines d'instructions n'est pas envisageable car la validité des instructions chargées est trop peu probable même avec un bon mécanisme de prédiction. Le nombre important de chargements suggère que la prédiction d'adresse pourrait apporter un gain de performance non négligeable même si l'on n'est pas certain que ces instructions soient sur le chemin critique, ce que l'on ne peut pas vérifier sans modéliser l'architecture du processeur.

Les mesures effectuées plus récemment sur les SPEC95 ont également confirmé ces chiffres. Ces mesures sont rapportées dans le tableau ci-dessous.

	BC	BCC	ARI	LOG	SHF	MUL	DIV	LD	ST	FOP	FMU	FDIV	FCO
go	2,97	11,98	39,36	3,32	10,17	0,06	0,00	22,94	9,19	0,00	0,00	0,00	0,00
m88ksim	6,52	14,72	22,32	14,10	10,38	0,01	0,27	19,49	12,19	0,00	0,00	0,00	0,00
gcc	4,91	16,49	25,43	8,32	4,26	0,10	0,05	26,28	14,16	0,00	0,00	0,00	0,01
compress	5,93	10,13	28,17	8,58	6,72	0,03	0,00	21,69	18,20	0,04	0,00	0,07	0,44
li	9,88	13,66	18,56	14,10	1,27	0,00	0,00	26,64	15,89	0,00	0,00	0,00	0,00
ijpeg	1,54	6,42	44,05	4,94	14,14	1,73	0,05	19,03	8,10	0,00	0,00	0,00	0,00
vortex	6,36	16,53	28,93	9,09	0,58	0,00	0,00	20,58	17,93	0,00	0,00	0,00	0,00
Cint	5,44	12,85	29,55	8,92	6,79	0,28	0,05	22,38	13,67	0,01	0,01	0,00	0,06
tomcatv	0,46	3,77	13,61	0,98	0,51	0,07	0,00	39,71	16,61	14,34	8,82	0,37	0,75
swim	0,49	2,49	18,64	0,48	0,01	0,00	0,00	24,57	8,07	26,30	15,35	0,74	2,86
applu	0,00	3,39	18,75	1,06	2,36	0,28	0,00	36,22	14,07	9,97	13,29	0,48	0,13
turb3d	0,77	3,44	24,83	2,46	5,32	2,30	0,00	24,67	19,18	8,78	7,85	0,03	0,39
apsi	0,54	2,94	21,09	1,34	3,48	1,97	0,03	31,54	16,42	9,77	7,61	1,64	1,64
fpppp	0,18	0,79	2,24	0,32	0,32	0,03	0,00	51,57	18,53	11,92	13,43	0,13	0,53
wave5	0,77	4,57	17,98	2,77	3,80	0,28	0,01	29,49	17,14	8,73	8,80	0,85	4,82
Cfp	0,46	3,06	16,73	1,34	2,26	0,7	0,06	33,97	15,72	12,83	10,74	1,59	1,59

BC : Branchement inconditionnels
BCC : Branchement conditionnels
ARI : Instructions arithmétiques
LOG : Instructions logiques
SHF : Décalages
MUL : Multiplications entières
DIV : Divisions entières
LD : Lecture
ST : Ecritures
FOP : Opérations flottantes
FMU : Multiplications flottantes
FDIV : Divisions flottantes
FCO : Changements de types

Tableau 1 : Distribution des instructions par type dans les SPEC95

2.3.3. Parallélisme niveau instruction disponible dans les programmes

Pour mesurer le parallélisme disponible, nous avons considéré un processeur pouvant exécuter un nombre infini d'instructions. On repère alors dans toute la trace du programme, les instructions dont les opérandes sont disponibles et qui peuvent donc être exécutées. C'est donc une machine purement *data-flow*. Nous avons dû faire des hypothèses sur les appels au système puisqu'ils ne sont pas tracés. Nous avons considéré deux cas extrêmes, soit les appels au système ne modifient aucune variable du programme (il n'y a donc aucune dépendance de données), soit ils constituent une barrière et les instructions qui sont situées après un appel système ne peuvent être exécutées avant celui-ci (on considère qu'elles sont toutes dépendantes du code exécuté lors de l'appel). Ces deux hypothèses nous donnent donc une fourchette dans laquelle se trouve le parallélisme réel. Enfin, un facteur limitatif du parallélisme est l'absence de renommage qui ajoute des dépendances artificielles dans le graphe de dépendances, nous avons donc considéré les cas où un renommage des registres et/ou de la mémoire est effectué. Un renommage de la mémoire peut être réalisé grâce à un prédicteur de dépendances, encore faut-il trouver un mécanisme qui ait un bon taux de prédiction.

Déterminer le parallélisme présent dans un programme revient à l'examiner entièrement et, à chaque examen, à déterminer toutes les instructions qui peuvent l'être, c'est-à-dire dont les opérandes sont connus en supposant un processeur dont les ressources sont infinies. Les programmes entiers affichent des parallélismes très variables (de 20 à 350) et en moyenne entre 70 et 80 instructions, les appels systèmes influant assez peu sur la quantité de parallélisme disponible. Par contre, l'impact du renommage (qui supprime les anti-dépendances et les dépendances de sortie) est énorme. On constate une baisse moyenne de 93% du parallélisme en absence de renommage et cette baisse est en grande partie liée au renommage des registres dans la plupart des cas (excepté pour le programme *sc* dont la perte de parallélisme due à l'absence de renommage mémoire est de 75% tandis qu'elle n'est que de 33% en moyenne). Le renommage de registres est systématiquement mis en œuvre dans les processeurs. Bien que le gain soit moins important, un renommage de la mémoire pourrait trouver sa place dans les prochains processeurs grâce à un prédicteur de dépendances [42].

Les programmes flottants affichent des parallélismes beaucoup plus importants (de 30 à 1300), avec une moyenne de 150-190. L'influence du renommage est plus grande pour ces programmes. En absence de renommage, le parallélisme chute énormément jusqu'à des valeurs proches voire inférieures à celles des programmes entiers. Deux cas particuliers sont à signaler : tomcatv a un parallélisme de 10 avec ou sans renommage, ce qui s'explique par le fait qu'il travaille sur de grands tableaux et que les opérations ne sont pas dépendantes et ora qui travaille également sur de grand tableaux mais qui a des instructions dépendantes les unes des autres gagne beaucoup de parallélisme avec un renommage mémoire. On voit donc que les programmes flottants sont loin d'être uniformes et sont assez différents des programmes entiers, ce qui peut s'expliquer par leur structure qui contient souvent de nombreuses boucles imbriquées. Ainsi, avec un renommage, le parallélisme de ces programmes est beaucoup plus grand.

2.3.4. Limitation de l'exploration pour extraire le parallélisme

Les chiffres ci-dessus mesurent le parallélisme disponible dans les programmes. Un processeur à exécution non ordonnée peut être vu comme la machine *data-flow* décrite ci-dessus mais fonctionnant sur un nombre d'instructions restreint (égal à la taille de sa fenêtre) sans ajouter aucune autre restriction de ressources. Nous avons donc examiné le parallélisme qui peut être obtenu en considérant une fenêtre d'une taille finie. Ce parallélisme est donné dans la Figure 14.

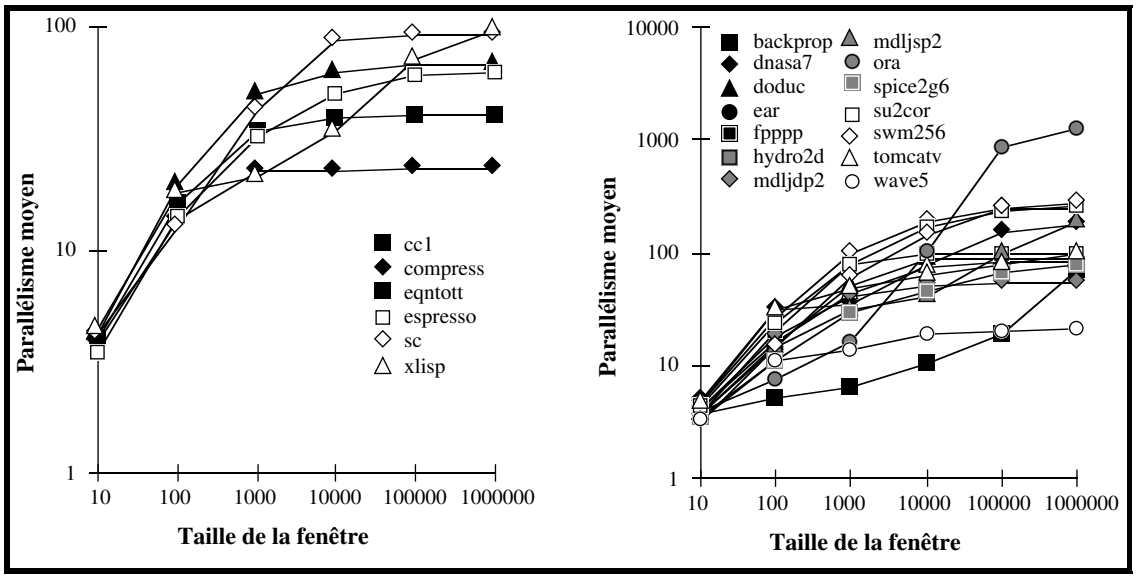

Figure 14.Localité du parallélisme

On a considéré ici que les appels système modifiaient toutes les variables et les renommages sont actifs. On remarque que les programmes entiers donnent la majeure partie de leur parallélisme dans une fenêtre de 1000 instructions. La fenêtre doit être encore plus grande pour les programmes flottants en termes de pourcentage par rapport au parallélisme disponible dans tout le programme mais, en valeur absolue, pour une même taille de fenêtre, ils présentent plus de parallélisme que les programmes entiers. On peut noter sur ces courbes les grandes variations entre les programmes. Backprop, par exemple, n'exhibe son parallélisme que pour une très grande fenêtre alors que wave5 fournit la majeure partie de son parallélisme avec une fenêtre de 100 instructions.

2.4. La fenêtre d'instructions

Nous avons vu que les programmes présentaient la majeure partie de leur parallélisme dans une fenêtre de 1000 instructions.

Afin de déterminer la taille optimale de la fenêtre d'instructions d'un processeur, il convient de considérer le degré d'émission de celui-ci qui détermine le parallélisme maximum que l'on pourra obtenir. Le degré d'émission est sans doute l'un des premiers critères que l'on se fixe lors de la conception d'un processeur même si ce critère peut être revu ensuite compte tenu des informations obtenues lors de la conception, notamment le temps de cycle que l'on pourra obtenir. Nous n'avons considéré aucune limitation autre que le degré d'émission. Le retrait des instructions est effectué de manière non ordonnée de manière à avoir une occupation optimale de la fenêtre. Nous avons considéré une prédiction de branchement parfaite, ce qui introduit plus de parallélisme que ce qui sera réellement obtenu. Ainsi, les tailles de fenêtre que nous obtenons sont des maxima et l'influence de la prédiction de branchement se fera d'autant plus sentir que la taille maximum de la fenêtre est grande. Par exemple, pour un degré 4, la taille que nous avons

déterminé est de 32 et c'est celle qui est généralement utilisée par les fabricants. Pour un degré 8, les fabricants adopteront sans doute des fenêtres de tailles inférieures à 96. Alors que les courbes précédentes montrent le parallélisme présent dans les programmes, ici des limitations structurelles à l'exploitation du parallélisme sont introduites.

Nous avons ainsi déterminé les tailles de fenêtres « optimales » pour des degrés d'émission de 4, 6 et 8. Le nombre d'instructions exécutées par cycle (IPC) donné dans le tableau ci-dessous est celui obtenu avec les latences du PowerPC 604 [58].

Degré d'émission	Taille de la fenêtre	IPC (entiers)	IPC (flottants)
4	32	3,85	2,43
6	64	5,57	3,28
8	96	6,93	3,97

Tableau 2. Tailles de fenêtres optimales

Il faut noter que le gain apporté par des degrés supérieurs à 8 apporte une augmentation faible de l'IPC par rapport à la complexité pour les mettre en œuvre pour les programmes entiers (CINT). Ce n'est pas le cas pour les programmes flottants (CFP) mais envisager des tailles de fenêtres supérieures à la centaine semble peu raisonnable car les accès aux structures de mémorisation nécessitent des arbitrages qui risquent d'influencer négativement le temps de cycle du processeur pour de grandes structures. Or, exploiter le parallélisme est important mais la performance du processeur dépend tout autant du temps de cycle. De plus, la prédiction de branchement n'est malheureusement pas parfaite et les dernières instructions chargées ont peu de chances d'être exécutées compte tenu du nombre de branchements prédits et non vérifiés dans une si grande fenêtre. Par ailleurs, les CFP sont fortement limités par les latences des opérations flottantes qui sont plus grandes que celles des opérations entières. Il semble donc plus judicieux de s'attacher à réduire ces latences bien que ce soit un problème très complexe si l'on ne veut pas que cela se fasse au détriment du temps de cycle. On peut tout de même penser qu'un processeur plus particulièrement dédié au calcul flottant pourrait avoir une fenêtre plus grande que les autres processeurs, d'autant plus que les branchements de ces programmes sont souvent hautement prédictibles.

2.5. Le retrait

Le retrait est habituellement effectué dans l'ordre. Un résultat trivial est que son degré doit être égal au degré d'émission pour ne pas perdre de performance, ce qui confirme les choix effectués par les fabricants de processeurs. Naturellement, le nombre de ports en écriture sur le banc de registres doit être égal au nombre d'instructions que l'on peut retirer pour ne pas provoquer d'aléas. On peut toutefois noter que les écritures n'ont pas de registres destination.

Nous avons évalué la possibilité de retirer avec un ordre minimal garantissant toutefois la possibilité de gestion précise des exceptions afin d'optimiser l'utilisation du tampon de réordonnancement. En effet, lorsqu'une instruction a terminé son exécution et n'est plus spéculative et qu'elle se trouve derrière des instructions qui ne peuvent plus provoquer d'exceptions, il est possible de retirer cette instruction même si les premières n'ont pas fini leur exécution. Par exemple, si l'on considère un processeur où les échecs dans la TLB sont traités par exception tandis que les erreurs sur le bus sont traitées de manière imprécise, une instruction qui suit un chargement peut être retirée dès que le chargement a effectué son accès à la TLB. On peut donc mémoriser dans le tampon de réordonnancement si les instructions peuvent ou non produire des exceptions et introduire un certain désordre dans le retrait. Nos mesures ont montré que cela apportait peu de gain par rapport à la complexité ajoutée [34]. Ce résultat s'explique par le fait que les instructions ne sont retirées que quelques cycles plus tôt et que les écritures en mémoire constituent une limitation car elles sont effectuées dans l'ordre, le gain est donc faible. On pourrait envisager d'écrire les écritures dans le désordre avec un verrou sur la ligne de cache où est effectuée l'écriture mais cela compliquerait encore le mécanisme de retrait dans le désordre qui, déja, n'est pas simple.

2.6. Unités d'exécution

En amont du retrait, les unités d'exécution limitent les performances si elles ne sont pas en nombre suffisant et, dans une moindre mesure, en raison des opérations non pipelinées. Chaque unité d'exécution traite un ou plusieurs types d'instructions et chaque processeur a adopté des configurations d'unités différentes. Le HP PA-8000 [22] [31] est un des rares processeurs qui comprend deux unités flottantes, chacune pouvant traiter tous les types d'instructions flottantes. Ainsi, il est un des processeurs les plus puissants en matière de calcul flottant. A l'opposé, le PowerPC 620 [38], par exemple, n'en comporte qu'une. En bref, de nombreux critères entrent en compte et, tout au long de la conception d'un processeur, des choix sont à faire et peuvent être remis en cause.

Toutefois, en fonction des caractéristiques des programmes d'évaluation, nous nous sommes attachés à définir une configuration qui représente un bon compromis entre le coût et les performances obtenues sans intégrer tous les critères auxquels un concepteur a à faire face. Les mesures précédentes supposaient un nombre d'unités infini. Nous avons donc limité, pour chaque classe d'instructions, le nombre d'instructions pouvant être amorcées et relevé la perte de performance associée. Ces mesures nous ont permis de définir des configurations apportant une perte de performance d'environ 10% sur les CINT et 5% sur les CFP. Avec les CINT, nous avons déterminé les unités arithmétiques et logiques, le nombre d'unités de branchement et d'accès à la mémoire. Avec les CFP, nous avons complété cette configuration avec des unités de calcul flottant. Naturellement, les unités entières sont sous-utilisées par les programmes flottants. Les accès à la mémoire ont été définis sur les CINT car ceux-ci nécessitent plus

d'accès à la mémoire que les flottants. La figure ci-dessous indique les configurations retenues en fonction du degré du processeur : il ne s'agit que de limites en deça desquelles une sévère perte de performance est enregistrée. A noter que la hiérarchie mémoire est supposée parfaite, or le nombre d'unités d'accès à la mémoire est fortement dépendant de la hiérarchie mémoire utilisée.

Mais la performance n'est pas le seul critère à considérer lors de la réalisation d'un processeur, il faut également tenir compte de la complexité. Pour les unités d'accès à la mémoire, un critère prépondérant est, bien entendu, le nombre de ports vers le cache, le nombre d'unités et le nombre de ports devant être égaux. On peut aussi vouloir privilégier ou non le calcul flottant (exemple ci-dessus). En ce cas, c'est plutôt le critère de marché qui est prépondérant. On peut aussi noter que notre configuration est fortement asymétrique en ce qui concerne les unités entières : elle en comprend trois types, ceci parce que les opérations de multiplication et de division entières sont très peu nombreuses et les décalages assez peu. Cette asymétrie est liée à une volonté d'économie de transistors mais elle peut poser des problèmes au niveau de la politique d'amorçage ou d'émission des instructions suivant la topologie du tampon d'amorçage. Par exemple, pour un degré 4, nos deux unités flottantes ne sont pas identiques. L'une ne traite pas les multiplications et l'autre ne traite pas les opérations arithmétiques usuelles. Si la différence de coût matériel entre cette configuration et une configuration à deux unités universelles n'est pas énorme, on aura tout intérêt à choisir la deuxième solution car cela simplifie l'envoi des instructions aux unités d'exécution : dans le premier cas, pour envoyer une instruction vers une unité d'exécution, il faudra regarder son type et pas dans le second. L'envoi des instructions peut avoir un impact non négligeable sur le temps de cycle ou sur la longueur du pipeline et, dans une moindre mesure, sur le nombre de transistors pour le mettre en œuvre. Le PA-8000 a systématiquement des ressources par paires pour permettre un arbitrage des instructions à exécuter plus rapide. C'est un exemple de retombée de l'implémentation sur l'architecture qui montre les limites d'une étude en milieu universitaire où l'implémentation est difficile à prendre en compte.

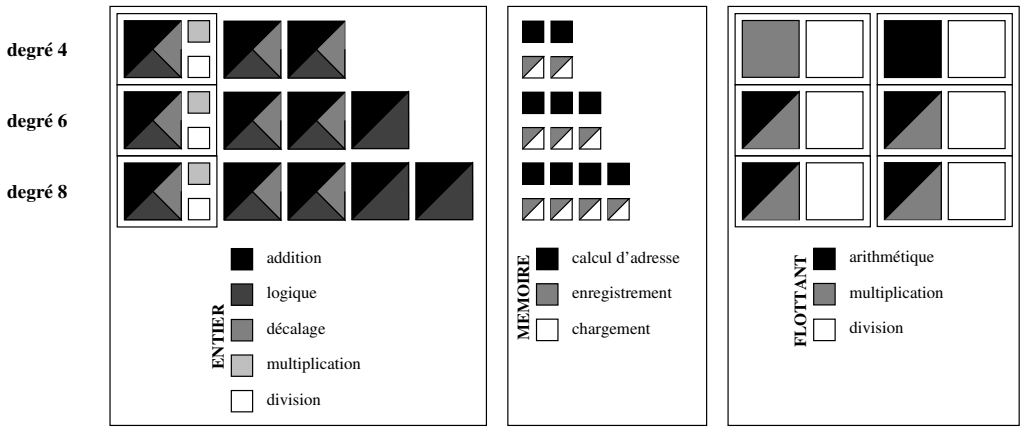

Figure 15.Configuration des unités fonctionnelles

2.7. Topologie du tampon

La structure qui conserve les instructions en amont des unités d'exécution est le tampon d'amorçage. Plusieurs politiques sont possibles pour répartir les instructions vers les unités fonctionnelles. La répartition peut être faite avant le rangement des instructions dans le tampon, (le tampon est alors réparti sur les unités d'exécution) ou après (dans le cas d'un tampon unique) ou bien encore en deux temps (si le tampon n'est pas unique ni réparti mais se situe quelque part entre les deux, ce que nous avons appelé semi-unifié). Là aussi, les constructeurs ont adopté des solutions différentes. Par exemple, le PowerPC620 a un tampon réparti, l'Intel Pentium Pro [45] a un tampon unifié. Pour les topologies semi-unifiées, on retrouve la même diversité avec une unification plus ou moins grande (de 2 partitions pour le HP-PA8000 à 5 pour le Sparc64 [24]) et une répartition dans les partitions tout aussi variée. Bien entendu, le choix de la topologie du tampon d'amorçage dépend en partie de la façon dont est découpé le pipeline, de la configuration des unités fonctionnelles, …

Dans le cas d'une topologie répartie, l'amorçage est très simple : il suffit de choisir une instruction parmi celles qui sont exécutables c'est-à-dire parmi celles dont les opérandes sont prêts. Comme chaque partition a peu d'entrées, ce choix est rapide. Par conséquent, avec une telle topologie, un étage de pipeline dédié à l'amorçage peut être économisé. Par contre, l'émission est plus complexe puisqu'elle doit se charger de la répartition. Il y a une certaine logique à effectuer cette répartition juste après le décodage puisque celui-ci indique le type des opérations et que cette répartition ne dépend que du type mais ce n'est pas impératif car l'information nécessaire pour la répartition suit l'instruction dans le pipeline. La topologie répartie induit forcément une perte de performance du fait que la sélection des instructions pour l'amorçage ne peut se faire que sur une partie du tampon. En conséquence, pour deux unités identiques, il est possible qu'une unité n'ait pas d'instructions prêtes à être exécutées alors qu'une autre en a plusieurs.

A contrario, une topologie unifiée nécessite une politique d'amorçage complexe qui doit choisir des instructions pour les différentes unités parmi un grand nombre d'entrées. Le fait que les entrées soient universelles (c'est-à-dire qu'elles peuvent contenir n'importe quel type d'opération) amène le meilleur partage possible de celles-ci au prix d'une sélection lors de l'amorçage plus complexe, l'économie en nombre d'entrées est ainsi contrebalancée par la complexité d'implémentation de la politique d'amorçage qui peut nécessiter un étage de pipeline pour être effectuée. Bien sûr, on peut ajouter un étage dans le pipeline pour l'amorçage ce qui résout le problème de l'arbitrage, mais l'ajout d'un étage présente d'autres inconvénients en particulier dans tous les cas où le pipeline doit être purgé.

Il faut noter ici la solution astucieuse du PA-8000 pour limiter le temps d'arbitrage sur un tampon à grand nombre d'entrées. Elle consiste à faire deux arbitrages en parallèle, l'un pour les entrées paires, l'autre pour les entrées impaires et pour des instructions de même type. Cette solution est particulièrement intéressante pour un tampon unifié car il est probable que les instructions amorçables sont uniformément réparties dans le tampon mais cela suppose que les ressources en unités fonctionnelles existent par paires.

Enfin, la topologie semi-unifiée essaye de tirer avantage des deux systèmes. Nous avons considéré pour les unités d'exécution les configurations définies dans la section précédente et évalué les différents choix possibles. Le tableau ci-dessous résume les résultats obtenus. Si l'on fait la somme du nombre optimal d'entrées pour chaque solution, on s'aperçoit que la solution la plus économique est la solution unifiée alors que la solution distribuée est la plus coûteuse. De plus, la perte de performance - non rapportée ici - est supérieure dans le cas d'une topologie distribuée (environ 5% contre moins de 1% pour la topologie unifiée [34] par rapport à un tampon unifié infini).

Degré	Unifié	Distribué (nombre d'entrées par UF)					
		décalage	entier	branchement	adresses	mémoire	flottant
4	28	6	4	4	4	8	12
6	48	8	8	8	6	10	16
8	72	10	10	8	6	12	22

Tableau 3. Tailles des tampons d'amorçage unifiés et distribués

Degré	Semi-unifié									
	mém.	E/B/F	mém.	ent/br	flottant	adres.	mém.	ent.	br.	flottant
4	12	24	12	20	24	8	12	16	8	24
6	24	36	24	36	32	16	24	32	16	32
8	36	52	36	52	44	24	36	48	24	44

Tableau 4. Tailles des tampons d'amorçage semi-unifiés

Un aspect dont il n'a pas été fait état jusqu'à maintenant est la politique d'amorçage des instructions : parmi plusieurs instructions prêtes, laquelle faut-il amorcer au plus tôt ? Nous avons évalué diverses alternatives visant à respecter l'ordre du programme, favoriser la résolution des branchements, etc. Contrairement à d'autres études, nous avons trouvé des différences de performance assez significatives. Il est possible que le choix de cette politique soit fortement dépendant du reste de l'architecture...

2.8. Acquisition des instructions

L'acquisition des instructions fait appel à deux structures : le cache bien sûr qui contient les instructions et les mécanismes de prédiction qui indiquent l'adresse des prochaines instructions à charger. Un tampon de lancement est chargé de réguler le flot d'instructions entre l'acquisition et l'amorçage (voir Figure 13 page 35).

Jusqu'à présent, nous avons considéré un mécanisme de chargement des instructions parfait. Or, la fréquence des instructions de branchement est de une toutes les 5 instructions pour les programmes entiers (les programmes flottants ont des blocs de base plus longs et ne sont pas concernés par cette étude). Dès lors que l'on a un degré supérieur à 4, plusieurs blocs de base doivent être chargés à chaque cycle puisqu'il y a de très fortes chances pour qu'un branchement soit présent dans les instructions chargées voire deux pour un degré 8.

Le mécanisme de prédiction de branchement et de chargement de blocs proposé ici a un double intérêt.

Dans les processeurs actuels, de degré 4, il n'est pas nécessaire de charger plusieurs blocs de base par cycle mais notre mécanisme permet de pipeliner la prédiction de branchement sans ajouter de pénalités ce qui est primordial pour les processeurs privilégiant le temps de cycle comme le DEC21164 [21]. Cela peut éviter l'insertion d'une bulle dans le pipeline pour chaque branchement pris [21][23][69] et empêcher la prédiction de branchement d'être un frein à la diminution du temps de cycle.

L'autre intérêt concerne les futurs processeurs qui auront certainement des degrés supérieurs à 4, 8 assurément et sans doute plus. Ce mécanisme permet d'accéder à plusieurs blocs de base simultanément, ce qui est indispensable pour soutenir la demande de tels processeurs.

2.8.1. Prédiction de branchement pipeline

La pénalité en cas de mauvaise prédiction est d'autant plus importante que le degré du processeur est grand et que le pipeline est profond, ces deux critères ayant tendance à croître d'une génération de processeurs à l'autre. En conséquence, il faut améliorer la qualité de la prédiction et cela passe, en absence de nouveaux algorithmes, par l'augmentation de la taille des structures de mémorisation pour la prédiction (Branch Target Buffer, Branch History Table). Or, augmenter la taille de ces structures accroît leur temps d'accès et risque ainsi de devenir un frein à la réduction du temps de cycle. La solution est donc (comme c'est de plus en plus le cas pour l'accès au cache) de pipeliner les accès à ces structures mais ceci entraîne un cycle de pénalité systématique en cas de branchement pris comme dans le Pentium Pro [7]. Le mécanisme que nous proposons prédit non pas le bloc qui suit un branchement mais le bloc qui suit le bloc situé après le branchement. Ainsi, la prédiction s'effectue avec un cran d'avance sur le chargement des instructions, ce qui supprime les suspensions du chargement dues à l'exécution pipeline sur deux cycles de la prédiction de branchement.

Le mécanisme d'indexation du tampon de cibles de branchements (BTB) est un peu plus complexe dans le sens où il nécessite plus d'informations mais le nombre d'entrées et la taille des entrées diffèrent peu d'un BTB classique. Toutefois, il est nécessaire que le degré d'associativité du BTB soit plus grand (4 au lieu de 2) car différents branchements peuvent être indexés avec la même adresse. Enfin, ce mécanisme nécessite une deuxième pile de retour pour charger deux blocs simultanément après un retour de sous-programme.

2.8.2. Prédiction de plusieurs blocs

Le mécanisme proposé ci-dessus peut également permettre de charger plusieurs blocs de base à la fois depuis le cache.

2.8.2.1. Intérêt

La Figure 16 montre l'apport de l'acquisition multiple pour un processeur ayant les caractéristiques décrites précédemment avec un tampon d'amorçage unifié en ne considérant que les programmes entiers car les programmes flottants ont des blocs de base beaucoup plus longs donc une acquisition simple introduit très peu de pertes.

Les performances données sont le rapport entre une configuration d'acquisition donnée et une acquisition parfaite. L'axe des x indique la taille de l'acquisition (1 ou 2 blocs de base) suivi de la taille du tampon d'émission.

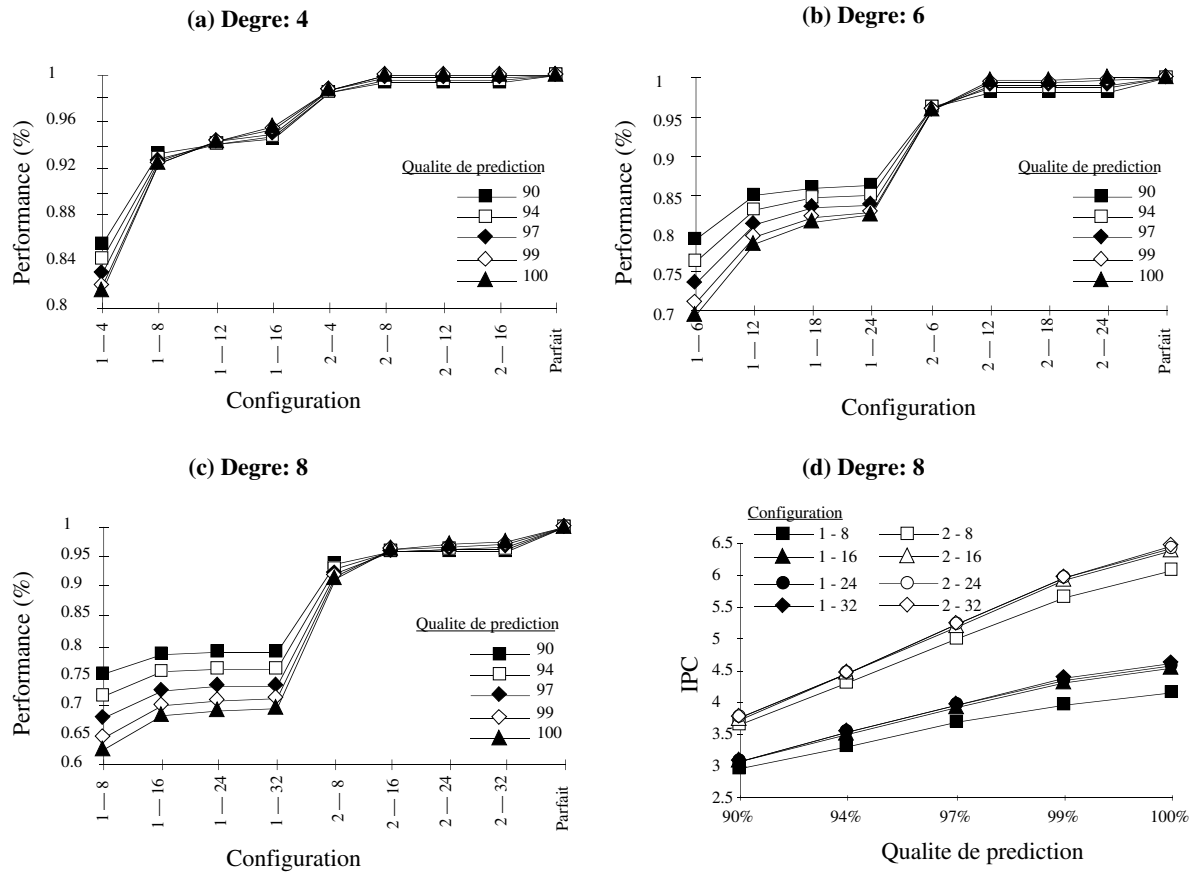

Figure 16. Performance de processeurs à chargement simple et multiple.

2.8.2.2. Travaux précédents

Quelques études [11][48][70] ont été proposées en ce sens mais elles nécessitent un surdimensionnement du BTB ou bien sont dédiées à un seul type de prédicteur ou encore reposent sur un partitionnement effectué par le compilateur.

2.8.2.3. Fonctionnement et résultats

Nous nous sommes concentrés sur le chargement de deux blocs mais le mécanisme semble généralisable à un nombre quelconque de blocs si l'on ne tient pas compte de la complexité de mise en œuvre. L'idée de base de notre mécanisme est que, pendant que l'on charge deux blocs d'instructions, leurs adresses peuvent être utilisées pour prédire les deux blocs à charger au cycle suivant. Comme deux prédictions sont effectuées à chaque cycle en utilisant les adresses des deux blocs en cours de chargement, les structures de prédiction doivent avoir deux ports. Cependant, le coût matériel peut être limité grâce à une organisation entrelacée. Ce mécanisme est donc moins coûteux que ceux évoqués ci-dessus ([11][48][70]) puisque la taille des structures est à peu près la même que pour un seul bloc. Il présente une qualité de prédiction à peu près identique à une prédiction d'un seul branchement (pour une table de 64K entrées, moins de 0,3%). L'associativité du BTB doit être multipliée par deux afin que, à nombre

d'entrées égal, le taux d'échec soit à peu près le même (- de 0,5% de différence pour la plupart des applications).

Les résultats que nous avons trouvés sont particulièrement intéressants puisque les limitations introduites par une prédiction d'un seul branchement sur un processeur de degré 8 sont quasiment effacées. On trouve ainsi une amélioration de l'IPC par rapport à la prédiction d'un seul bloc variant de 120 à 150% suivant la qualité de la prédiction de branchement.

Une autre approche à ce problème, le cache de traces, mémorise, d'une exécution sur l'autre, les blocs de base[5]. Son principal inconvénient est qu'il doit être maintenu cohérent avec la mémoire pour les codes qui s'auto-modifient, qu'il doit être de grande taille et il semble qu'un cache d'instructions soit tout de même nécessaire car le nombre d'échecs dans le cache de traces est important. Son principal avantage par rapport à notre mécanisme est qu'il ne repose plus sur un cache d'instructions dont la longueur des blocs est fixe. Le chargement de blocs de base depuis le cache pose un problème supplémentaire qui est que les blocs de base ne sont pas alignés sur les blocs du cache. Ainsi, il est possible qu'un bloc de base commence à la fin d'un bloc du cache. Comme on charge les instructions issues d'un bloc du cache, on ne charge que peu d'instructions alors que le bloc de base peut être beaucoup plus long mais chevauche plusieurs blocs du cache. Le cache de traces, lui construit ses traces à partir des blocs de base. L'alignement est donc automatique.

Il me semble que le problème reste ouvert et qu'un gain est encore possible à ce niveau aussi bien pour le cache de traces que le chargement multiple.

Cette étude a été réalisée dans le cadre d'un projet soutenu par le PRC ANM réunissant le projet CAPS de l'IRISA (P. Michaud et A. Seznec) et notre équipe (S. Jourdan et moi-même). Ce mécanisme est décrit en détail dans [34][I].

2.9. Les accès aux données en mémoire

2.9.1. Introduction

On a vu précédemment des configurations de processeurs où les ressources limitent peu l'exploitation du parallélisme de niveau instruction. En ce qui concerne la mémoire, nous avons défini un nombre d'unités de calcul des adresses et un nombre de ports sur la mémoire cache en

[5] Ce n'est pas son seul avantage puisqu'il permet d'éviter lors d'une seconde exécution d'un bloc son passage à travers les étages de chargement et de décodage.

fonction du degré. Toutefois, différentes techniques peuvent être implémentées à ce niveau qui n'ont pas été prise en compte dans le modèle précédent (avec cache d'instructions parfait).

L'étude rapportée ici vise à évaluer ces mécanismes et à montrer l'apport de chacun de ces mécanismes en termes de parallélisme d'instructions.

Elle a été menée conjointement par Pham Tuong Hai, Christine Rochange et moi-même durant la même période que l'étude qui précède et de manière indépendante.

Les accès à la mémoire ont deux particularités. D'une part, la latence des accès est variable suivant le niveau de la hiérarchie mémoire où l'on va trouver la donnée et, en moyenne, il est relativement long. D'autre part, il est possible de mettre en œuvre un mécanisme de renommage de la mémoire mais il est différent du renommage des registres et nécessite une structure spécifique qu'on appelera tampon des lectures/écritures. Ce tampon est similaire à un tampon de réordonanncement mais il mémorise l'ordre des instructions d'accès à la mémoire et des champs d'information différents sont nécessaires. Enfin, le nombre d'accès au cache de données de premier niveau est beaucoup plus limité que pour l'accès aux registres. Ceci provient du fait que le nombre d'emplacements mémoire est beaucoup plus grand que le nombre de registres, il est donc coûteux de mettre en œuvre plusieurs ports d'accès. En général, cette mise en œuvre passe par un entrelaçage des bancs du cache, ce qui introduit des restrictions sur les accès possibles simultanément. Après avoir rappelé le modèle utilisé, l'apport du renommage et des mécanismes de dépassement et de transmission associés sont évalués dans le paragraphe suivant.

Par ailleurs, même avec ce mécanisme de renommage, l'exécution des lectures peut être retardée du fait que l'adresse des écritures n'est pas calculée alors que l'adresse de lectures plus récentes l'est. Pour éviter ce problème, il faudrait prédire l'absence de dépendances entre les lectures et les écritures dont l'adresse n'a pas été calculée. Il a été montré que, pour les processeurs actuels, une prédiction aveugle supposant systématiquement l'absence de dépendances entre des lectures et les écritures dont l'adresse est inconnue était viable. Nous avons voulu vérifier ce résultat pour des processeurs ayant une plus grande fenêtre d'anticipation.

Cette exécution spéculative des lectures induit la nécessité de récupérer un état correct en cas d'erreur. Dans un premier temps, nous nous sommes limités à ramener la donnée dans le processeur sans la transmettre aux instructions en attente. Ceci permet une récupération aisée puisqu'en cas d'erreur, seule l'instruction de lecture fautive doit être ré-exécutée. Ensuite, nous verrons l'apport en performance que peut apporter la transmission du résultat aux instructions qui l'attendent avec un mécanisme de récupération identique à celui de la prédiction de branchement, c'est-à-dire annulation de toutes les instructions qui suivent une lecture erronée et ré-acquisition, décodage, exécution de ces instructions.

Puisqu'il y a exécution spéculative des instructions et mécanisme de récupération, une approche qui a été précédemment étudiée peut être mise en œuvre : la prédiction d'adresse.

L'apport de la prédiction d'adresse est évalué, pour une prédiction d'adresse parfaite et pour un mécanisme simple. Ce mécanisme de prédiction d'adresse introduit un nombre d'exécutions erronées plus grand, des mécanismes de récupération plus performants sont donc évalués et un mécanisme original est proposé.

2.9.2. Modèle utilisé

Le modèle utilisé ici est celui auquel on a abouti précédemment (paragraphe 2.6), c'est à dire qu'il possède quatre ports sur le cache de données de premier niveau, celui-ci étant entrelacé et quatre unités dédiées aux calculs d'adresse.

Par ailleurs, en ce qui concerne la hiérarchie mémoire, nous avons considéré un cache de données de 32KO à correspondance directe, non bloquant et ayant un temps d'accès de un cycle. Le cache de second niveau est également non bloquant, il a une capacité de 256 octets et est associatif par ensemble de 4 voies et ne possèdent qu'un port. Son temps d'accès est de 4 cycles. La mémoire a un temps d'accès de 20 cycles. Ces valeurs correspondent aux caches de données des processeurs actuels, ce qui peut paraître faible puisque l'on considère un processeur dont les caractéristiques sont futuristes. Cela s'explique par le fait que les programmes test utilisés ont des jeux de données relativement petits, ce qui interdit l'utilisation de caches plus grands pour avoir un taux d'échecs proche de la réalité.

2.9.3. Mécanismes de renommage

2.9.3.1. Principe du renommage

En ce qui concerne le renommage des registres, celui-ci est effectué après le décodage des instructions c'est à dire dès que l'on connait les registres logiques utilisés par l'instruction. Généralement, une table de correspondance (mise à jour après le décodage des instructions et lors du retrait) permet d'effectuer ce renommage.

Une telle approche ne peut être envisagée en ce qui concerne les instructions d'accès à la mémoire puisque l'adresse doit être calculée pour connaître l'emplacement concerné. Ainsi, les instructions à la mémoire sont composées de deux opérations, l'une destinée à connaître l'adresse de l'emplacement en mémoire, l'autre destinée à effectuer l'accès. Connaître l'adresse de l'emplacement permet d'effectuer le renommage tandis que l'accès est l'exécution proprement dite de l'instruction qui permettra à celle-ci d'être retirée. Il y a bien sûr une relation de précédence entre ces deux opérations.

Le renommage permet de supprimer les dépendances de sortie et les anti-dépendances et de détecter les dépendances vraies entre instructions garantissant ainsi une exécution correcte. L'utilisation d'emplacements supplémentaires permet également d'avoir un état spéculatif du processeur défini par ces emplacements et un emplacement non spéculatif par les registres

logiques qui ne sont mis à jour que lors du retrait des instructions et donc lorsque leur bonne exécution est garantie.

2.9.3.2. Renommage de la mémoire

Les instructions d'accès à la mémoire profite du renomage de registres au même titre que les autres instructions puisqu'elles ont pour opérandes des registres. Par contre, elles sont quelque peu différentes des autres instructions dans le sens où les écritures n'ont pas de registre résultat. Le résultat est l'écriture en mémoire. Autre différence, les écritures doivent impérativement être effectuées dans l'ordre en mémoire pour garantir le modèle de consistance mémoire alors que les lectures, comme les autres instructions, peuvent être effectuées dans le désordre (en respectant les dépendances). Ainsi, les écritures en mémoire sont effectuées lors du retrait des instructions d'écriture. Il y aurait possibilité d'utiliser une autre méthode en se servant du cache comme emplacement de renommage. Dans ce cas, il faudrait sauvegarder la valeur de l'emplacement du cache dans un tampon d'historique, effectuer l'écriture en mémoire en bloquant l'accès à ce bloc tant que l'écriture est spéculative. C'est une solution que nous n'avons pas envisagée car elle nous semble complexe et il nous semble préférable de disposer d'une file des écritures en attente quitte à l'inclure dans le mécanisme de maintien de la cohérence des données dans les caches.

Le renommage mémoire a les mêmes fonctions que le renommage de registres. Mais l'utilisation d'emplacements supplémentaires (la file des lectures/écritures) permet également un gain car l'accès à ces emplacements est beaucoup plus court que l'accès à la hiérarchie mémoire. Ainsi, en cas de dépendance vraie, la transmission de la valeur d'une écriture vers une lecture est beaucoup plus rapide. Un autre gain est apporté par la détection des dépendances qui permet le dépassement des écritures par les lectures du moment que les adresses sont différentes. En effet, le seul effet des écritures est l'écriture en mémoire. Les seules instructions qui dépendent d'une écriture sont des lectures à la même adresse et elles peuvent profiter d'une transmission si l'écriture est dans la file des lectures/écritures. Par contre, de nombreuses instructions dépendent des lectures : la donnée est lue en mémoire pour être utilisée. Par conséquent, on a tout intérêt à privilégier l'accès au cache par les lectures plutôt que par les écritures. Donc, la détection des dépendances doit être effectuée pour permettre ce dépassement des écritures par les lectures.

2.9.3.3. Résultats

En ce qui concerne le dépassement, il faut noter qu'un dépassement peut avoir lieu dès que l'adresse de l'emplacement des écritures plus anciennes est calculée. Cette adresse peut être calculée dès que les opérandes pour le calcul de l'adresse sont connus. Peu importe la donnée qui va être écrite. Dans ce cas, on a une accélération moyenne de 1,75 alors que, si on attend que la donnée soit prête pour effectuer un dépassement (ce qui ne sert à rien), le gain n'est que de 1,2. Ceci montre bien l'intérêt de découpler le calcul de l'adresse de l'accès et donc

d'effectuer le calcul de l'adresse au plus tôt. Ces accélérations varient grandement (de 1,25 à 3,4) en fonction des programmes selon qu'ils présentent beaucoup d'écritures ou non et en fonction du taux de synonymes entre lectures et écritures présentes dans le processeur. Un programme (Alvinn) présente un résultat inattendu. Le fait de réordonner les instructions d'accès à la mémoire a considérablement augmenté la localité dans le cache de second niveau, le nombre de défauts étant réduits de 30 %. Nous n'avons pas considéré ici les possibilités de combiner les requêtes de lecture (ou d'écritures) entre elles pour diminuer le nombre d'accès qui peut également permettre de diminuer un peu le nombre de défauts dans le cache de premier niveau.

Ajouter au dépassement la transmission du résultat des écritures vers les lectures apporte encore une accélération de 1,25. La combinaison de ces deux mécanismes est donc particulièrement intéressante, ce qui explique leur mis en œuvre dans tous les processeurs récents. Bien entendu, chaque fois, qu'il y a transmission, un accès au cache est évité. Ainsi, le nombre total d'accès au cache est en moyenne réduit d'un peu moins de 10 %. Par contre, le dépassement et la transmission permettent d'augmenter le parallélisme. De ce fait, le nombre de cycles pour exécuter le programme est diminué et le nombre d'accès au cache par cycle augmente. On mesure donc bien toute l'importance de ce mécanisme.

2.9.4. Prédiction de dépendances aveugle

sans exécution spéculative des instructions dépendantes

avec exécution spéculative des instructions dépendantes

2.9.5. Prédiction d'adresse

Le prédicteur d'adresse ressemble aux prédicteurs proposés dans la littérature, c'est-à-dire qu'il est basé sur une table qui mémorise le comportement des chargements précédents. Il s'agit d'un mécanisme de prédiction dirigé par le pas des adresses des données (la différence entre les adresses de deux accès consécutifs par la même instruction de chargement). Toutefois, pour un processeur de degré important (8 dans notre cas), plusieurs instances d'un chargement peuvent être présentes dans le processeur (en particulier dans le cas d'un chargement à l'intérieur d'une boucle dont plusieurs itérations sont présentes dans le processeur). Nous avons donc, comme dans [10], un champ supplémentaire qui mémorise ce nombre et qui, multiplié par le pas, nous permet de calculer l'adresse prédite. Ce champ est très peu coûteux puisqu'on a rarement plus de 4 occurrences d'un chargement dans le processeur.

Nous avons choisi d'effectuer la prédiction d'adresse lors de l'émission. Ceci permet d'éviter une pollution de la table de prédiction puisqu'au moment de l'émission, on connaît les instructions qui sont des chargements. Pour une application directe, cette table peut comporter 256 à 512 entrées (ce qui représente 3 ou 6 KO), des valeurs plus grandes étant inutiles. Des

mécanismes d'applications plus complexes ont un impact réduit sur le nombre d'entrées et sur la qualité de la prédiction. Une prédiction lors du chargement nécessite une table beaucoup plus grande ou alors un mécanisme de filtrage pour éviter la pollution de celle-ci.

Cette table est accédée par l'adresse de l'instruction (connue puisqu'on prédit lors de l'émission) mais plusieurs prédictions doivent être effectuées en parallèle puisque huit instructions sont chargées à chaque cycle. Sur ces huit instructions, il y a rarement plus de 4 chargements donc 4 ports d'accès à la table suffisent. Une prédiction lors du chargement nécessite également plusieurs ports, la différence étant que l'accès à la table se fait avec l'adresse du bloc mais on peut également limiter le nombre de prédictions.

En ce qui concerne l'efficacité du mécanisme, le nombre de chargements prédits et exécutés avec l'adresse prédite est de l'ordre de 30 à 40% et ce pourcentage est d'autant plus grand que la fenêtre d'instructions du processeur est petite. Il est limité par le pourcentage d'instructions qui peuvent être prédites et, surtout, par le fait que l'adresse de nombreuses instructions prédites est finalement calculée avant leur exécution. Suivant le rapport entre le nombre d'unités de calcul de l'adresse et le nombre de ports d'accès au cache, ce pourcentage varie. La qualité de la prédiction du mécanisme est de 96%. On peut certainement améliorer le taux de prédictions (qui est de l'ordre de 60%) en gardant une qualité de prédiction équivalente mais ce n'était pas le but de cette étude.

2.9.6. Mécanismes de récupération

==========

Quelques études ont été menées en ce domaine [5] [6] [16] [19] [52] [68]. Dans un processeur à exécution ordonnée (cas considéré dans la plupart des études), on gagne alors un

ou plusieurs[6] cycles suivant le moment où l'on effectue la prédiction pour peu que l'instruction de chargement soit sur un chemin critique.

2.9.7. Prédiction d'adresse et dépassement spéculatif

Les études énoncées ci-dessus soit considèrent un processeur à exécution ordonnée, soit sont assez peu précises quant aux mécanismes mis en œuvre au niveau des accès mémoire. Nous avons donc décidé d'étudier conjointement ces divers mécanismes pour exécuter les chargements le plus rapidement possible.

Les résultats que nous avons obtenus pour cette étude sont issus d'un simulateur exécutant le programme et non des traces comme pour les précédentes expérimentations.

Le mécanisme de dépassement spéculatif en lui-même, qui consiste à supposer que l'adresse d'un chargement n'est pas identique aux adresses des rangements présents dans la file et dont l'adresse n'est pas connue, est très simple et très efficace en absence de prédiction d'adresse ; la plupart des programmes ont des taux de mauvais dépassement spéculatifs nuls ou très proches de 0. Seuls tomcatv et doduc ont plus de 1% de mauvais dépassement spéculatifs. Par contre, la répercussion sur la performance globale du processeur est assez faible.

Le fait d'autoriser les dépassements spéculatifs augmente légèrement le taux d'exécution de chargements prédits (et surtout des chargements bien prédits) de quelques pour-cent. Par contre, la présence de la prédiction de l'adresse des chargements augmente le nombre de mauvais dépassements spéculatifs. Comme on prédit les chargements, on peut prédire les rangements et, ainsi diminuer fortement le taux de mauvais dépassements jusqu'à retrouver des valeurs très faibles : 0,7% en moyenne contre 13% sans prédiction de l'adresse des rangements. La taille de la table de prédiction doit alors être multipliée par 1,5. Une autre solution, que nous n'avons pas étudiée, serait de prédire les dépendances entre les chargements et les rangements [42].

Les différents champs de la table doivent être mis à jour : l'automate qui autorise la prédiction, l'adresse prédite et le champ indiquant le nombre occurrences d'un chargement présent dans le processeur. Ce dernier champ est toujours incrémenté lors de l'émission et décrémenté lors du retrait. Pour les deux autres champs, nous avons considéré dans ce qui précède une mise à jour lors du retrait. Mettre à jour lors de l'exécution, c'est-à-dire dans le désordre, permet de gagner quelques pour-cent sur l'exécution de chargements bien prédits que le dépassement existe ou non. En fait, la prédiction des adresses des chargements est meilleure

[6] Cela dépend de la profondeur du pipeline en amont de l'unité d'exécution

car les ruptures de pas sont détectées plus tôt et parce que la politique d'amorçage des instructions consiste à exécuter, parmi les instructions prêtes, les plus anciennes, ce qui favorise une exécution dans l'ordre. L'ordre des instructions est donc assez bien respecté. Ce respect de l'ordre allié au fait que la moitié des chargements prédits sont finalement effectués avec l'adresse calculée entraîne peu de mauvaises mises à jour. Toutefois, il faut préciser que nous avons considéré une prédiction de branchement parfaite. Une prédiction de branchement réelle pourrait remettre en cause ces résultats, de même que la taille de la table qui devrait peut-être être un peu plus grande.

2.9.8. Exécution

On peut envisager une exécution spéculative des instructions dépendant d'un chargement dont l'adresse a été prédite à condition, bien entendu, soit que le taux de bonnes prédictions soit très élevé, soit que la récupération en cas d'erreur se fasse très rapidement. Dans le cas de la prédiction de branchements, la récupération est relativement simple car toutes les instructions situées après un branchement doivent être annulées. Pour la prédiction d'adresse, on peut faire de même mais il semble plus astucieux d'annuler sélectivement les instructions qui dépendent du chargement mal prédit. Il faut, pour cela, mémoriser le graphe de dépendances ou une partie de celui-ci. Le mécanisme de récupération n'est pas décrit dans les articles cités dans l'introduction mais elle est supposée être effectuée en un cycle.

Des études sont en cours sur l'exécution spéculative des instructions dépendant de chargements. Les premiers résultats montrent que l'exécution des instructions dépendant de chargements dont l'adresse est prédite est particulièrement intéressante pour de petites fenêtres d'anticipation et moins sur de grandes fenêtres. Ceci peut s'expliquer par le fait qu'avec une grande fenêtre, les instructions dépendant d'un chargement sont moins souvent sur un chemin critique. Cette étude nécessite encore d'autres investigations.

En ce qui concerne la récupération en cas d'erreur, on peut s'attendre, de manière intuitive, à ce qu'un mécanisme qui annule toutes les instructions qui suivent un chargement mal prédit pour les recharger ensuite se révèle catastrophique. Une autre solution est de garder les instructions dans les stations de réservation tant qu'elles sont spéculatives. Ainsi, on peut annuler uniquement l'exécution des instructions, il n'est pas utile de les recharger puisqu'elles sont toujours présentes dans les stations. Mais cela implique un nombre de stations plus grand puisqu'il faut mémoriser des instructions plus longtemps. Cette solution n'est donc guère envisageable que dans le cas où les stations et le tampon de réordonnancement constituent une structure unique. Une récupération de toutes les instructions dépendant d'un chargement prédit en un seul cycle semble difficilement réalisable du fait qu'il faudrait une matrice de dépendances soit n^2 bits, n étant la taille du tampon de réordonnancement. Différentes options existent. Par exemple, chaque bit de dépendance peut être partagé entre plusieurs instructions consécutives si l'on fait en sorte que le compilateur génère rarement plusieurs chargements consécutifs. Une

autre idée qui semble encore plus intéressante est de ne mémoriser, pour chaque instruction, que les instructions dont elle dépend directement et d'annuler les instructions dépendantes petit à petit en parcourant un étage du graphe à chaque cycle. Avec une politique d'amorçage des instructions privilégiant l'instruction la plus ancienne parmi les instructions prêtes, la propagation d'une donnée erronée devrait vite s'arrêter. Cette solution a l'avantage de ne nécessiter, pour chaque instruction, que de deux pointeurs vers ses parents. Or, ceux-ci existent déjà pour la propagation des résultats. Les premières évaluations montrent qu'en termes d'IPC, ces deux solutions ont les mêmes performances. Seule un petite baisse de performance est relevée pour nasa7.

Cette étude sur la prédiction d'adresse n'est pas terminée. Elle va être poursuivie et probablement complétée par des travaux sur la prédiction de valeur.

3. Conclusion

L'écriture de ce manuscrit permet de faire une petite pause dans la course effrénée que nous menons tous les jours à la fois pour suivre les développements dans ce secteur encore en effervescence et pour essayer d'y apporter une modeste (le moins possible, quand même) contribution. Ce retour sur les années passées m'a permis de mettre en évidence un point que je considère aujourd'hui comme extrêmement important : LA METHODE. Quand je parle de méthode, je ne veux pas parler d'un carcan qui ressemblerait à une administration mais bien d'un certain nombre de règles qu'il est nécessaire de respecter si l'on veut être efficace. Je n'ai nullement l'intention d'écrire un discours sur la méthode, d'ailleurs cela a déjà été fait. Je souhaite simplement souligner que ces années m'ont fait découvrir ce besoin. Je pense qu'un minimum d'enseignement en ce domaine pourrait être très profitable.

En ce qui concerne la recherche en architecture des processeurs superscalaires, il est clair qu'aujourd'hui une grande part de la logique est dédiée au contrôle des instructions en cours d'exécution dans le processeur et non au calcul lui-même. Ceci peut sembler un paradoxe compte tenu des idées à l'origine de l'approche RISC. Mais la logique est dédiée ici à l'exploitation du parallélisme et n'est pas la cause d'une limitation du temps de cycle ou, du moins, si c'est le cas, cette limitation est compensée par la quantité de parallélisme exploité.

Cette approche ne semble pas poser de problèmes particuliers d'intégration avec l'augmentation prévisible du nombre de transistors sur la puce mais pourrait bien en poser en termes de temps de cycle. En effet, les structures de mémorisation sont de plus en plus grosses et les arbitrages d'autant plus longs. On peut pipeliner à peu près jusqu'au point où la longueur du pipeline devient un facteur pénalisant en raison des purges nécessitées par des erreurs de prédiction, des exceptions, ... Une autre limite à long terme concerne les temps de propagation dans les lignes qui traversent la puce, problème que l'on connait déja au niveau des circuits imprimés alors qu'il y a quelques années, ce problème n'apparaissait que pour des liaisons longues distances. cet aspect relève de l'implémentation mais pourrait avoir un impact sur l'architecture si l'implémentation devient impossible.

Par ailleurs, en proportion, la taille d'une unité fonctionnelle ne change pas ; l'accroissement du nombre de transistors sur la puce pourrait donc permettre de disposer de beaucoup plus d'unités fonctionnelles, quitte à ne pas les utiliser toutes s'il n'y a pas suffisamment

d'instructions indépendantes dans le flot d'instructions. Ceci permet, entre autres, de simplifier le contrôle des instructions, en particulier lors de l'émission.

Le problème qui semble majeur est de tolérer la latence de la mémoire puisque cette latence ne cesse de s'accroître. La prédiction d'adresse dont j'ai parlé est un début de réponse mais ne suffira sans doute pas. Comme la place disponible sur la puce augmente, il paraît assez logique d'intégrer plus de mémoire cache sur la puce et, pourquoi pas, une partie de la mémoire DRAM comme dans l'approche IRAM [46][47][51]. Mais plus généralement, le problème de la latence de la mémoire est liée au fait que des instructions dépendent de la valeur lue en mémoire et même que les instructions dépendent les unes des autres, ce qui est inhérent à notre modèle de programmation. En ce sens, l'idée de prédiction de valeur [40] parait très séduisante. Elle devrait permettre une exécution parallèle spéculative d'instructions sans attendre la résolution des dépendances à un prix qui ne devrait pas être excessif. Encore faut-il trouver les algorithmes de prédiction adéquats, de nombreuses propositions vont certainement être faites dans les années à venir.

Une autre approche qui semble tout aussi intéressante est l'approche *multithread* qui consiste, par exemple, à changer de flot lors d'un échec dans le cache L2 ou mieux, en *multihread simultané*, à changer de flot dès qu'il y a le moindre blocage. Cela nécessite un peu de place sur la puce mais ne rajoute pas beaucoup de contrôle si l'on considère une rotation sur les flots relativement simple. Cette approche semble d'autant plus intéressante qu'elle correspond bien à l'idée que les processeurs doivent supporter des applications multimédia et donc doivent traiter plusieurs actions en parallèle. Par ailleurs, les systèmes opératoires intègrent déjà la notion de *multithreading*, les efforts logiciels devraient donc être faibles (encore faut-il que les applications utilisent réellement cette approche, c'est-à-dire que l'environnement de programmation soit suffisamment attractif pour séduire le programmeur ou bien que le compilateur puisse générer automatiquement des threads avec toutes les bonnes caractéristiques). Enfin, le multithreading peut au moins exécuter plusieurs tâches unix, reste à savoir comment se comporte la TLB dans ce cas. De même, il semble que cette approche pose quelques problèmes au niveau de la prédiction de branchement [14] bien que [30] ait montré le contraire pour des hypothèses différentes, il en est de même pour le cache de deuxième niveau. Ces deux études s'accordent toutefois pour montrer un goulot d'étranglement au niveau du cache de données de premier niveau [14][30].

Cette approche doit être opposée à l'approche qui consisterait à mettre en œuvre plusieurs processeurs, fussent-ils plus simple, sur la même puce. L'approche multiflot simultané est plus séduisante puisqu'elle est destinée à optimiser l'utilisation des ressources en sélectionnant dynamiquement les instructions de plusieurs flots. C'est toutefois au prix d'une logique compliquée pour cette sélection. L'approche multiprocesseurs a pour avantage sa simplicité et

une relative indépendance des différents processeurs. C'est en fait une réplication de logique. Ainsi, dans cette approche, le problème des temps de propagation dans les lignes évoqué plus haut aurait moins d'importance que dans un approche multiflot simultané où toute la logique de la puce effectue la même fonction. Autre avantage du multiprocesseur : sa simplicité de réalisation qui peut permettre des temps de développement plus courts, ce qui est un argument commercial et industriel non négligeable. Par contre, l'approche multiflot a pour avantage de fournir un processeur superscalaire surdimensionné : s'il n'y a qu'un seul flot, celui-ci devrait s'exécuter au mieux. Dans le cas du multiprocesseur, s'il n'est pas possible de dégager plusieurs flots dans une application, celle-ci aura de très mauvaises performances.

Le compilateur va donc avoir une importance considérable au moins pour détecter automatiquement les flots mais pas seulement. Un certain nombre d'informations pourraient être transmises au processeur pour être utilisées lors du chargement (avec un prédécodage) ou même après le décodage. Ces informations, qui peuvent être des informations issues du langage de haut niveau pour peu que le compilateur soit à même de les véhiculer tout au long de la chaîne de compilation, pourraient servir, par exemple, à affiner la prédiction de branchement, d'adresse ou de valeur mais aussi à indiquer les dépendances entre instructions, ce qui pourrait simplifier considérablement la logique d'émission des instructions. L'interaction entre ce que peut indiquer le compilateur et ce que peut faire le matériel me semble trop faible aujourd'hui et pourrait certainement être largement améliorée, ce qui nécessitera une modification du jeu d'instructions et peut-être un passage à un jeu d'instructions 64 bits. C'est peut-être là qu'est le plus grand problème. Pour justifier l'accroissement de bande passante que demande un jeu d'instructions 64 bits, il faut que les informations données par le compilateur apportent réellement beaucoup. Une thèse est en cours sur ce sujet à l'IRIT sous la direction de Louis Féraud et je suis ce travail car la recherche en compilation doit, à mon avis, être très liée à celle concernant l'architecture des processeurs.

En ce qui concerne les multiprocesseurs à mémoire partagée, il me semble que les trafics que représentent les adresses, les données et les informations de cohérence doivent être différenciés et véhiculés par des réseaux distincts. C'est d'ailleurs ce qui est fait dans les serveurs récents comme Escala de Bull où les adresses sont véhiculées sur un bus partagé tandis que les données transitent via un crossbar.

Il est clair que si on arrive à augmenter suffisamment la bande passante entre le processeur et la mémoire par un crossbar complet grâce à des liaisons série pour de petites configurations et peut-être à l'optique pour de plus grosses configurations, le problème du temps de latence n'en est pas pour autant résolu. J'en ai déjà fait état dans le cadre de l'architecture des processeurs. Le préchargement de données peut être profitable si la bande passante est suffisante à tous les niveaux (cache L2, réseau, mémoire), ce qui peut être coûteux. Un préchargement dirigé par le

processeur, et non par un mécanisme indépendant, plus précis et plus efficace, pourrait être une solution [25]. Une fois de plus, la conception du multiprocesseur ramène à celle du processeur avec une contrainte supplémentaire de cohérence des données dans les caches qui rend encore plus cruciale la qualité du préchargement.

Enfin, l'approche IRAM que j'ai évoquée semble impliquer une architecture NUMA ou COMA puisque la mémoire présente sur la puce est, de ce fait, à accès privilégié. Le circuit réalisé par Mitsubishi [53] qui comporte sur la même puce un cœur de microprocesseur 32 bits et une DRAM de 16 Mbits peut constituer l'élément de base d'une telle architecture.

Distribué/partagé: convergence. Culler pp. 53-56

Plus généralement, de nombreux palliatifs (ayant chacun leurs défauts) ont été proposés du fait que l'on ne sait pas réaliser une architecture à mémoire partagée à accès uniforme avec un grand nombre de processeurs. Le projet M3S a le mérite de s'attaquer à un problème difficile et vise une solution globale et non un ersatz permettant de s'abstraire d'un problème existant. L'approche n'est sans doute pas pragamatique mais la recherche doit elle être pragmatique ? C'est un prix nobel qui a dit que sans une recherche fondamentale le téléphone n'existerait pas mais on aurait des pigeons très efficaces. L'architecture UMA est certainement la meilleure solution, il faut chercher le moyen de la réaliser. En attendant, il va de soi que d'autres propositions plus viables sont intéressantes car elles auront des retombées industrielles.

J'ai évoqué ci-dessus quelques pistes qui semblent intéressantes mais les pronostics en matière d'évolution de l'informatique sont particulièrement dangereux et les plus grands noms s'y sont fourvoyés, je m'abstiendrai donc d'y ajouter les miens.

4. Bibliographie

1. G. S. Almasi, A. Gottlieb
Highly Parallel Computing
2nd edition, Benjamin/Cummings Publishing Company, 1994. ISBN 0-8053-0443-6

2. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Poterfield, B. Smith
The TERA Computer System
International conference on Supercomputing, Amsterdam, septembre 1990.

3. G. Astfalk, T. Brewer
An Overview of the Convex SPP2000 Hardware
Hewlett-Packard, 1996.

4. T. M. Austin, G. S. Sohi
Dynamic Dependency Analysis of Ordinary Programs
19th International Symposium on Computer Architecture, mai 1992, Australie, pp. 342-351. ISBN 0-89791-509-7.

5. T.M. Austin, D. N. Pnevmatikatos, G. S. Sohi
Streamlining Data Cache Access with Fast Address Calculation
22nd International Symposium on Computer Architecture, juin 1995, Santa Margherita Ligure, Italy, pp. 369-380. ISBN 0-89791-698-0.

6. T. M. Austin, G. S. Sohi
Zero-Cycle Loads: Microarchitecture Support for Reducing Load Latency
28th Annual International Symposium on Microarchitecture, Ann Arbor, décembre 1995, pp. 82-92. ISBN 0-8186-7349-4.

7. D. Bhandarkar, J. Ding
Performance Characterization of the Pentium Pro Processor
3rd International Symposium on High-Performance Computer Architecture, février 1997, pp. 288-297. ISBN 0-8186-7764-3.

8. M. Butler, T.-Y. Yeh, Y. N. Patt et al.
Single Instruction Stream Parallelism Is Greater Than Two

18th International Symposium on Computer Architecture, mai 1991, Toronto, pp. 276-286. ISBN 0-89791-394-9.

9. D. Carrière
Architecture, Consistance et Etude formelle du module mémoire d'une architecture à mémoire physiquement partagée
Thèse de l'université Paul Sabatier, juillet 1996.

10. T.-F. Chen
Data Prefetching for High-Performance Processors
PhD thesis, Department of Computer Science and Engineering, University of Washington, 1993.

11. T. M. Conte, K. N. Menezes, P. M. Mills, B. A. Patel
Optimization of Instruction Fetch Mechanisms for High Issue rates
22nd International Symposium on Computer Architecture, juin 1995, Santa Margherita Ligure, pp. 333-344. ISBN 0-89791-698-0.

12. M. Dubois
Throughput Analysis of Cache-Based Multiprocessors with Multipe Busses
IEEE Transactions on Computers, janvier 1988, vol. 37, num. 1. ISSN 0018-9340.

13. P. K. Dubey, G. B. Adams III, M. J. Flynn
Instruction Window Size Trade-Offs and Characterization of Program Parallelism
IEEE Transactions on Computers, avril 1994, vol. 43, num. 4, pp. 431-442. ISSN 0018-9340.

14. S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, D. M. Tullsen
Simultaneous Multithreading: A Platform for Next-Generation Processors
IEEE Micro, septembre 1997. ISSN 0272-1732.

15. R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, M. S. Squillante, . Liu
Evaluation of Multithreaded Uniprocessors for Commercial Application Environments
23rd International Symposium on Computer Architecture, mai 1996, Philadelphie, pp. 203-312. ISBN0-89791-786-3

16. R. J. Eickemeyer, S. Vassiliadis
A load-instruction unit for pipelined processors
IBM Journal of Research and Development, volume 37, num. 4, juillet 1993.

17. T. Y. Feng
A Survey of Interconnection Networks
IEEE Computer, vol. 14, décembre 1981, pp. 12-27. ISSN 0018-9162.

18. R. Giladi, N. Ahituv
SPEC as a Performance Evaluation Measure

IEEE Computer, août 1995, vol. 28, num. 8, pp. 33-42. ISSN 0018-9162.

19. M. Golden, T. N. Mudge
Hardware Support for Hiding Cache Latency
Technical Report of the university of Michigan, n° CSE TR152/93, 1993.

20. A. R. Karlin, M. S. Manasse, L. Rudolph, D. D. Sleator
Competitive Snoopy Caching
27th Annual Symposium on Foundations of Computer Science, 1986, pp. 244-254.

21. L. Gwennap
Digital Leads the Pack with 21164
Microprocessor Report, vol. 8, num. 11, septembre 1994. ISSN 0899-9341

22. L. Gwennap
PA-8000 combines Complexity and Speed
Microprocessor Report, vol. 8, num. 15, novembre 1994. ISSN 0899-9341.

23. L. Gwennap
Intel's P6 Uses Decoupled Superscalar Design
Microprocessor Report, vol. 9, num. 2, février 1995. ISSN 0899-9341.

24. L. Gwennap
Hal Reveals Multichip SPARC Processor
Microprocessor Report, vol. 9, num. 3, mars 1995. ISSN 0899-9341.

25. J. Gonzales, A. Gonzales
Speculative Execution via Address Prediction and Data Prefetching
International Conference on Supercomputing, Vienne, juillet 1997.

26. E. Hagersten, A. Landin, S. Haridi
DDM - A Cache-Only Memory Architecture
IEEE Computer, vol. 25, num. 9, septembre 1992, pp. 44-54. ISSN 0018-9162.

27. J. L. Hennessy, D. A. Patterson
Architecture des ordinateurs, une approche quantitative
International Thomson Publishing, 2e édition, 1996. ISBN 2-84180-022-9.

28. Hewlett-Packard
Gigabit Rate Transmit Receive Chip Set, Technical Data of HDMP1002 & HDMP-1004
Optoelectronics designer's Catalog, pp. 5:151-5:180, 1993.

29. M. D. Hill, S. Eggers, J. Larus et al.
Design Decisions in Spur

IEEE Computer, vol. 19, num. 10, novembre 1986, pp. 8-22. ISSN 0018-9162.

30. S. Hilly
Etude du parallélisme monolithique: cas du multiflot simultané
Thèse de l'université de Rennes 1, juin 1997.

31. D. Hunt
Advanced Performance Features of the 64-bit PA-8000
COMPCON'95, San Francisco, mars 1995, pp. 123-129. ISSN 1063-6390.

32. Kendall Square Research
KSR-1 Technical Summary
Massachusetts, 1992.

33. N. P. Jouppi, D. W. Wall
Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines
3rd International Conference on Architectural Support for Programming Languages and Operating Systems, avril 1989, pp. 272-282. ISBN 0-89791-300-0.

34. S. Jourdan
Mécanismes d'exécution non ordonnée et d'acquisition dans les processeurs superscalaires
Thèse de l'université Paul Sabatier, juillet 1996.

35. K. Lee, S. Kim, G. Ahn, D.-K. Jeong
A CMOS Serial Link for Fully Duplexed Data Communication
IEEE Journal of Solid-State Circuits, vol. 30, num. 4, avril 1995.

36. D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. L. Hennessy
The Stanford DASH Multiprocessor
17th International Symposium on Computer Architecture, juin 1990, Seattle, pp. 148-159.

37. D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. L. Hennessy, M. Horowitz, M. S. Lam
The Stanford DASH Multiprocessor
IEEE Computer, vol. 25, num. 3, mars 1992. ISSN 0018-9162.

38. D. Levitan, T. Thomas, P. Tu
The PowerPC60 Microprocessor: A High Performance Superscalar RISC Microprocessor
COMPCON'95, San Francisco, mars 1995, pp. 285-291. ISSN 1063-6390

39. M. H. Lipasti, C. B. Wilkerson, J. P. Shen
Value Locality and Load Value Prediction
7th International Conference on Architectural Support for Programming Languages and Operating Systems, octobre 1996, pp. 138-149. ISBN 0-89791-767-7.

40. M. H. Lipasti, J. P. Shen
Exceeding the Dataflow Limit via value Prediction
29th Annual International Symposium on Microarchitecture, pp. 226-237, Paris, décembre 1996. ISBN 0-8186-7641-8.

41. R. Marbot, J.-C. Lebihan, A. Cofler, R. Nezamzadeh-Moosavi
BULLIT Data Sheet
BULL Serial Link Technology, rev 2.0, mai 1995.

42. A. Moshovos, S. E. Breach, T. N. Vijaykumar, G. S. Sohi
Dynamic Speculation and Synchronisation of Data Dependences
24th International Symposium on Computer Architecture,Denver, juin 1997, pp. 181-193. ISBN 0-89791-901-7.

43. A. G. Nowatzyk, M. C. Browne, E. J. Kelly, M. Parkin
S-Connect: from Networks of Workstations to Supercomputer Performance
22nd International Symposium on Computer Architecture, Santa Margherita Ligure, juin 1995, pp. 71-82. ISBN 0-89791-698-0.

44. A. R. Omondi
Computer Arithmetic Systems - Algorithms, Architecture and Implementations
Prentice Hall International, 1994.

45. D. B. Papworth
Tuning the Pentium Pro Microarchitecture
IEEE Micro, avril 1996. ISSN 0272-1732.

46. D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, K. Yelick
A Case for Intelligent RAM: IRAM
IEEE Micro, avril 1997. ISSN 0272-1732.

47. D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, K. Yelick
Intelligent RAM/ Chips that remember and compute
International Solid-State Circuits Conference , San Francisco, CA, 1997.

48. D. N. Pnevtikatos, M. Franklin, G. S. Sohi
Control Flow Prediction for Dynamic ILP Processors
26th Annual International Symposium on Microarchitecture, Austin, décembre 1993, pp. 153-163. ISBN 0-8186-5280-2.

49. L. Rauchwerger, P. K. Dubey, R. Nair

Measuring Limits of Parallelism and Characterizing its Vulnerability to Resource Constraints
26th Annual International Symposium on Microarchitecture, Austin, décembre 1993, pp. 105-117. ISBN 0-8186-5280-2.

50. C. Rochange
Evaluation des performances d'architectures multiprocesseurs à mémoire logiquement partagée
Thèse de doctorat de l'université Paul Sabatier, Toulouse, décembre 1993.

51. A. Saulsbury, Fong Pong, A. Nowatzyk
Missing the Memory Wall: The Case for Processor/Memory Integration
23rd International Symposium on Computer Architecture, pp. 90-101, Philadelphie, juin 1996, ISBN 0-89791-786-3.

52. Y. Sazeides, S. Vassiliadis, J. E. Smith
The Performance Potential of Data Dependence Speculation and Collapsing
29th Annual International Symposium on Microarchitecture, pp. 238-249, Paris, décembre 1996. ISBN 0-8186-7641-8.

53. Shimizu et al.
A Multimedia 32b RISC Microprocessor with 16 Mb DRAM
International Solid-State-Circuits Conference, février 1996, pp. 216-217.

54. Silicon Graphics Inc., Joe Heinrich
Origin 2000 Technical Report, SN0 Theory of Operations
Document number 007-3439-001.

55. B. J. Smith
Architecture and Applications of the HEP Multiprocessor Computer System
IEEE Tutorial "Supercomputers: Design and Applications", edited by Kai Hwang. ISBN 0-8186-0581-2.

56. M. D. Smith
Tracing with Pixie
Rapport technique CSL-TR-91-497, Center for Integrated Systems, Stanford University, novembre 1991.

57. J. E. Smith, G. S. Sohi
The microarchitecture of Superscalar Processors
Proceedings of the IEEE, décembre 1995, Special Issue on Microprocessors, vol. 83, num. 12. ISSN 0018-9219.

58. S. P. Song, M. Denman, J. Chang
The PowerPC 604 RISC Microprocessor
IEEE Micro, octobre 1994, vol. 14, num. 5, pp. 8-17.
ISSN 0272-1732.

59. SPEC
SPEC92 Technical Manual, v 1.1, 1992.

60. P. Stenström
A Survey of Cache Coherence Schemes for Multiprocessors
IEEE Computer, juin 1990, vol. 23, num. 6, pp. 12-24. ISSN 0018-9162.

61. P. Stenström, M. Brorsson, L. Sandberg
An Adaptative Cache Coherence Protocol Optimized for Migratory Sharing
20th Annual International Symposium on Computer Architecture, San Diego, mai 1993, pp. 109-118. ISBN 0-8186-3810-9.

62. Sun Microsystems Inc.
The Ultra Enterprise 10000 Server
Technical White Paper.

63. D. M. Tullsen, S. J. Eggers, H. M. Levy
Simultaneous Multithreading: Maximizing On-Chip Parallelism
22nd International Symposium on Computer Architecture, juin 1995, Santa Margherita Ligure, pp. 392-403. ISBN 0-89791-698-0

64. A. K. Uht, V. Sindagi
Disjoint Eager Execution: An Optimal Form of Speculative Execution
28th Annual International Symposium on Microarchitecture, Ann Arbor, décembre 1995, pp. 313-325. ISBN 0-8186-7349-4.

65. D. W. Wall
Limits of Instruction-Level Parallelism
4th International Conference on Architectural Support for Programming Languages and Operating Systems, avril 1991, pp. 176-188. ISBN 0-89791-380-9.

66. D. W. Wall
Limits of Instruction-Level Parallelism

Rapport de recherche 93/6, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, novembre 1993.

67. W.-D. Weber, A. Gupta
Analysis of Cache Invalidation Patterns in Multiprocessors
3rd International Conference on Architectural Support for Programming Languages and Operating Systems, avril 1989, pp. 243-256. ISBN 0-89791-300-0.

68. L. Widigen, E. Sowadsky, K. McGrath
Eliminating Operand Read Latency
ACM Computer Architecture News, vol. 24, n° 5, pp. 18-22, décembre 1996. ISSN 0163-5964.

69. K. C. Yeager,
The Mips R10000 Superscalar Microprocessor
IEEE Micro, vol. 16, n° 2, avril 1996, ISSN 0272-1732.

70. T-Y. Yeh, D. T. Marr, Y. N. Patt
Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache
7th International Conference on Supercomputing, juillet 1993, Tokyo, pp. 67-76. ISBN 0-89791-600-X.

Annexe : Publications

Conférences internationales avec actes

I. A. Seznec, S. Jourdan, P. Sainrat, P. Michaud
Multiple-Block Ahead Branch Predictors
7th International Conference on Architectural Support for programming Languages and Operating Systems ; Cambridge, USA, octobre 1996.
ACM SIGARCH Computer Architecture News, volume 24, numéro spécial, pp. 116-127.

II. S. Jourdan, P. Sainrat, D. Litaize
An Investigation of the Performance of Various Instruction-Issue Buffer Topologies
28th Annual IEEE/ACM Symposium on Microarchitecture (MICRO-28) ; Ann Arbor, USA, décembre 1995, pp. 279-284.

III. S. Jourdan, P. Sainrat
Is Symmetry Really Worthwhile ?
Euromicro95, 21st Euromicro Conference : Design of Hardware/Software Systems ; Como, Italie, septembre 1995, pp. 196-203.

IV. S. Jourdan, P. Sainrat, D. Litaize
Exploring Configurations of Functional Units in an Out-Of-Order Superscalar Microprocessor
22nd International Symposium on Computer Architecture ; Santa Margherita Ligure, Italie, juin 1995.
ACM SIGARCH Computer Architecture News, volume 23, numéro 2, pp. 117-125.

V. D. Litaize, D. Carrière, A. Mzoughi, C. Rochange, P. Sainrat
Using Ultra High-Speed Serial Links in Shared-Memory Multiprocessors
Open Bus Systems '94 ; novembre 1994, Paris, pp. 103-110.

VI. C. Rochange, P. Sainrat, D. Litaize
Performance of M3S for the SOR algorithm
PARLE'93 ; Munich, Allemagne, juin 1993, pp. 676-679.

VII. D. Litaize, A. Mzoughi, C. Rochange, P. Sainrat
The design of the M3S project : a Multiported Shared Memory Multiprocessor
Supercomputing'92 ; Minneapolis, USA, novembre 1992, pp. 326-335. ISBN 0-8186-2630-5

VIII. D. Carrière, D. Litaize, C. Rochange, P. Sainrat
Design of the memory plane of the M3S project
2nd International Workshop on Field-programmable Logic and Applications ; Vienne, Autriche, septembre 1992.

IX. D. Litaize, A. Mzoughi, C. Rochange, P. Sainrat
Towards a shared-memory massively parallel multiprocessor
19th Annual International Symposium on Computer Architecture ; Gold Coast, Australie, mai 1992.
ACM SIGARCH Computer Architecture News, volume 20, numéro 2, pp. 70-79.

X. P. Sainrat, D. Litaize
On some technical problems inherent to the use of high-speed digital circuits
International Workshop on Parallel Computing ; Trani, Italy, septembre 1991, pp. 194-197.

XI. F. Elkhlifi, O. Hammami, M. Lalam, D. Litaize, A. Mzoughi, P. Sainrat
Multiprocessors system with caches, serial multiport memory and pseudo crossbar of serial links
International Association Of Science and Technology for Development - IASTED ; Grindelwald, Switzerland, février 1989 pp. 167-170.

XII. D. Litaize, F. Elkhlifi, M. Lalam, A. Mzoughi, P. Sainrat, J-C Salinier
Multiprocessors with serial multiport memories.
Conference on Parallel ARchitectures and Languages Europe - PARLE89 ; Eindhoven, The Netherlands, juin 1989, pp. 34-51.

Conférences internationales sans actes

XIII. D. Carrière, D. Litaize, A. Mzoughi, C. Rochange, P. Sainrat
Update on the M3S project
3rd Workshop on Scalable Shared-Memory Multiprocessors ; San Diego, USA, mai 1993.

XIV. D. Litaize, O. Hammami, A. Mzoughi, C. Rochange, P. Sainrat
Project M3S : How serial links are well suited for scalability
Second Workshop on Scalable Shared Memory Multiprocessors ; Toronto, Canada , mai 1991.

Chapitres de livre

XV. D. Litaize, A. Mzoughi, C. Rochange, P. Sainrat
La liaison série à ultra haut débit : une (la?) solution pour les liaisons inter-modules en environnement multiprocesseur
Technologies Matérielles Futures pour l'ordinateur, chapitre 16, éditions Frontières, Paris, pp. 167-177.

Revues

XVI. C. Rochange, P. T. Hai, P. Sainrat, D. Litaize
Préchargement de données dans les caches
Technique et Sciences Informatiques, vol.16, n°4, avril 1997.

XVII. C. Rochange, P. Sainrat, D. Litaize
M3S : A Scalable UMA Multiprocessor
IEEE TCCA Newsletter - Spring 1994.

XVIII. P. Sainrat
800 Mbit/s grâce à l'arséniure de gallium
Electronique, le mensuel des ingénieurs de conception, numéro 21, octobre 1992.

XIX. D. Litaize, O. Hammami, M. Lalam, A. Mzoughi, P. Sainrat
Multiprocessors with a serial multiport memory and a pseudo-crossbar of serial links used as a processor memory switch.
ACM SIGARCH Computer Architecture News - Vol.17, N°6, Dec.89, pp. 8-21.

Conférences nationales avec actes

XX. P. T. Hai, C. Rochange, P. Sainrat, D. Litaize,
Prédiction de l'adresse des lectures pour tolérer la latence des accès à la mémoire
Neuvièmes rencontres du Parallélisme (RenPar9), Lausanne, 20-23 mai 1997.

XXI. S. Jourdan, P. Sainrat, D. Litaize
Tampons d'amorçage des microprocesseurs superscalaires à exécution non ordonnée : une analyse
Huitièmes rencontres du parallélisme (Renpar'8) ; Bordeaux, mai 1996, pp. 121-124.

XXII. S. Jourdan, P. Sainrat
Parallélisme potentiel de grain fin des programmes d'évaluation SPEC92
Quatrième symposium sur les Architectures Nouvelles de Machines ; Rennes, décembre 1995, pp. 97-106.

XXIII. D. Litaize, C. Guittenit, A. Mzoughi, P. Sainrat
Applications des liaisons série en environnement multiprocesseur à mémoire commune
Journées PRC/GDR Architecture de Machines Nouvelles ; Orsay, mars 1994, pp. 31-36.

XXIV. D. Litaize, A. Mzoughi, P.Sainrat, R. Pulou, D. Carrière, C. Rochange
Projet M3S : multiprocesseur à mémoire multiport série
Journées “Modèles d’exécution et architectures parallèles” ; PRC-ANM, Rennes, décembre 1993, pp. 47-52.

XXV. D. Litaize, D. Carrière, A. Mzoughi, C. Rochange, P.Sainrat
Le module mémoire et la liaison série du multiprocesseur M3S
Forum des recherches en informatique ; juin 1993, pp. 64-65.

XXVI. D. Litaize, O. Hammami, A. Mzoughi, C. Rochange, P. Sainrat, R. Pulou
Multiprocesseur massivement parallèle à mémoire multiport série
Troisième symposium sur les Architectures Nouvelles de Machines ; Palaiseau, France, juin 1991.

XXVII. D. Litaize, O. Hammami, P. Sainrat, R. Pulou
Les liaisons série du multiprocesseur M3S : justification, problèmes techniques, solutions
Deuxième symposium sur les Architectures Nouvelles de Machines ; Toulouse, septembre 1990, pp. 243-266.

XXVIII. D. Litaize, F. Elkhlifi, O. Hammami, M. Lalam, A. Mzoughi, P. Sainrat
Présentation de M3S : Multiprocesseur à mémoire multiport série .
Journées " Algorithmes parallèles et architectures nouvelles" ; Toulouse, mars 1989, pp 1-2.

Conférences nationales sans actes

XXIX. D. Carrière, S. Jourdan, D. Litaize, C. Rochange, P. Sainrat
Le parallélisme de données et d’instructions dans les multiprocesseurs à mémoire commune
Journées sur le parallélisme de l’IRIT ; avril 94.

XXX. D. Litaize, A. Mzoughi, C. Rochange, P. Sainrat
La liaison série à ultra haut débit : une (la?) solution pour les liaisons inter-modules en environnement multiprocesseur
Colloque "Technologies Matérielles Futures pour l'ordinateur" ; Paris, mai 1992.

Brevet

XXXI. D. Litaize, F. Elkhlifi, M. Lalam, A. Mzoughi, P. Sainrat, J-C Salinier
Système multiprocesseur, procédé d'échange d'informations mis en oeuvre et composant mémoire multiport série susceptible d'équiper le dit système.
Dépot international PCT du brevet (Europe-Japon-USA) 28-12-88.

Rapports de fin de contrat

XXXII. G. Berger-Sabbatel, P. Sainrat, A. Seznec, Y. Jégou, P. Garda, E. Belhaire, B. Zerrouk, A. Derieux, D. Trystram, P. Michallon
Impact des liaisons séries haut débit sur l'architecture des machines parallèles
Rapport final programme de recherches inter-PRC "Nouvelles technologies pour l'architecture", 1995.

XXXIII. D. Litaize, D.Carrière, A. Mzoughi, C. Rochange, P. Sainrat
Evaluations, applications des liaisons série à haut débit
Rapport final contrat num. 9300422 pour la région Midi-Pyrénées, juin 1994.

XXXIV. D. Litaize, D.Carrière, A. Mzoughi, C. Rochange, P. Sainrat, R. Pulou
Rapport final du projet M3S pour la région Midi-Pyrénées, juin 1993.

XXXV. D. Litaize, F. Elkhlifi, O. Hammami, M. Lalam, A. Mzoughi, P. Sainrat
Rapport final, contrat de plan état-région, juin 1990.

XXXVI. D. Litaize, F. Elkhlifi, O. Hammami, M. Lalam, A. Mzoughi, P. Sainrat
Rapport d'avancement des travaux, contrat de plan état-région, juin 1989.

Rapports internes et thèse

XXXVII. A. Seznec, S. Jourdan, P. Sainrat, P. Michaud
Multiple-Block Ahead Branch Predictors
Rapport interne IRIT/96-09-R et IRISA 990, 1996.

XXXVIII. S. Jourdan, P. Sainrat
Génération de trace avec pixie sous IRIX 5.3, et analyse de l'impact de dépendances de données à partir des SPEC92
Rapport interne IRIT n°95-10-R, 1995.

XXXIX. S. Jourdan, P. Sainrat
Configuration exécutive des Processeurs Superscalaires à Initiation Non Ordonnée
Rapport interne IRIT no 94-47-R, 1994.

XL. C. Rochange, P. Sainrat
Approcher VHDL et utiliser VDS
Rapport interne, IRIT/92-27-R, 1992.

XLI. R. Pulou, P.Sainrat
Carte horloge 400 MHz pour le multiprocesseur M3S
Rapport interne IRIT n° 91-90-R, 1991.

XLII. P. Sainrat
Réseau d'interconnexion du multiprocesseur M3S : étude et mise en oeuvre
Thèse de doctorat de l'université Paul Sabatier, Toulouse, janvier 1991.

XLIII. F. Baverez, D. Litaize, R. Pulou, P.Sainrat
La liaison de cohérence de M3S et ses interfaces
Rapport interne IRIT n° 91-38-R, 1991.

XLIV. D. Litaize, F. Elkhlifi, O. Hammami, M. Lalam, A. Mzoughi, P. Sainrat
Cache-based multiprocessors using serial multiport memories.
Rapport interne LSI, 1989.