Ordonnancement distribué d'instructions

Ordonnancement distribué d'instructions

Bernard Goossens — David Defour

Laboratoire LP2A, équipe DALI,
Université de Perpignan,
52 avenue Paul Alduy,
66860 Perpignan Cedex,
goossens@univ-perp.fr, ddefour@univ-perp.fr

RÉSUMÉ. *Cet article présente un algorithme de traitement distribué des instructions adapté aux micro-architectures superscalaires de degré élevé. La méthode consiste à répartir à la fois le banc de registres de renommage et les stations de réservation afin de limiter le nombre de ports d'accès aux registres et le nombre de comparateurs des stations. L'association entre les résultats produits et les sources dépendantes n'est plus globale mais ciblée grace à un mécanisme d'identification des instructions et de leurs composantes. La méthode, en limitant à quatre ports l'accès au banc de registres de renommage, permet, tout en augmentant le nombre de registres, de conserver un temps d'accès inférieur au cycle.*

ABSTRACT. *This article presents an algorithm to perform a distributed computation of the instructions, suited to high degree superscalar microarchitectures. The method relies on a partitionning of both the register file and the reservation stations in order to decrease the number of register file access ports and the number of stations comparators. Matching the results with the depending sources is no more global but point to point thanks to an identification of the instructions and their components. The method, by limiting the access resources to each renaming register to four ports allows, despite an increase of the number of registers, to keep the access time beyond the cycle time.*

MOTS-CLÉS : *micro-architecture, processeur superscalaire, traitement distribué.*

KEYWORDS: *micrarchitecture, superscalar processor, distributed computation.*

1. Introduction

Un processeur doté de p opérateurs doit à chaque cycle lire $2p$ sources et écrire p résultats. Un cœur d'exécution centralisé emmagasinant un ensemble de 2^n instructions comprend un banc de 2^n registres de renommage avec pour chacun, $2p$ ports de lecture et p ports d'écriture. A titre d'exemple, le banc de registres d'un pentium 4 (Hinton *et al.,* 2001) (128 registres de 32 bits avec 18 ports d'accès) est accédé en deux cycles.

La surface occupée par le banc de registres, de laquelle dépend son temps d'accès, est fonction du nombre de registres, de la largeur de mot et surtout du carré du nombre de ports. En effet, l'ensemble des ports se présente sous la forme d'une matrice dont les lignes sont les commandes d'accès, les colonnes forment les données et la diagonale contient les transistors. Ajouter un port étend la matrice d'une ligne et d'une colonne.

Par conséquent, si l'on souhaite doubler le nombre de ports d'un banc de registres pour doubler le degré superscalaire du processeur, tout en conservant un temps d'accès constant, il faut diviser par quatre le nombre de registres du banc. Par exemple, un processeur pentium pour exécuter 12 opérations par cycle, soit deux fois plus qu'un pentium 4, devrait n'avoir plus que 32 registres de renommage pour conserver un temps d'accès de deux cycles. Pour 24 opérations, le banc de registres devrait se limiter à seulement 8 registres. En tentant de favoriser l'ILP d'un côté, on finit par le pénaliser plus fortement de l'autre.

Une solution proposée pour augmenter la taille du banc de registres sans dégrader le cycle est de le hiérarchiser (Swenson *et al.,* 1988) (Yung *et al.,* 1995) (Cruz *et al.,* 2000) (Butts *et al.,* 2004). Un premier niveau sert de cache, le banc de registres formant un second niveau. Néanmoins, il faut noter que le premier niveau hiérarchique ne peut bénéficier d'aucune réduction sur le nombre de ports.

Un certain nombre de travaux ont été consacrés à la réduction du nombre de ports : en partitionnant le banc (Kailas *et al.,* 2002) (Zalamea *et al.,* 2003), en limitant le nombre de lectures et d'écritures (Kim *et al.,* 2003) (Gonzalez *et al.,* 2004) ou une combinaison des deux méthodes (Seznec *et al.,* 2002) (Park *et al.,* 2002). Dans tous les cas, le nombre de ports est fonction du nombre d'opérations lancées par cycle. Plus le degré superscalaire augmente, plus le nombre de ports nécessaires est élevé.

Le banc de registres n'est pas la seule structure critique. Les stations de réservation, qui indiquent pour chaque instruction si ses sources sont disponibles, contiennent des comparateurs chargés de repérer les destinations des résultats calculés par les unités fonctionnelles (Palacharla *et al.,* 1997). Le nombre de tels comparateurs varie en fonction de deux fois le nombre de résultats produits par cycle multiplié par le nombre de stations. Là encore, l'augmentation est quadratique par rapport au nombre d'opérations.

Pour diminuer le nombre de comparateurs, on peut remarquer que peu d'instructions attendent deux sources. (Ernst *et al.,* 2002) propose de ne laisser qu'un seul

comparateur par instruction. On divise ainsi par deux le nombre total de comparateurs.

Une autre possibilité est de hiérarchiser les stations en séparant par exemple les instructions en fonction de leur latence (Michaud *et al.,* 2001) (Lebeck *et al.,* 2002). Cela permet de ne laisser qu'une partie réduite des instructions dans les stations de premier niveau (le nombre de comparateurs n'est plus fonction que du degré superscalaire et ne dépend plus du nombre total de stations).

Une autre voie pour simplifier le mécanisme de réveil des instructions (Weiss *et al.,* 1984) (Sato *et al.,* 2001) (Ramirez *et al.,* 2004) est de remplacer les comparateurs de stations par des vecteurs de successeurs. Chaque instruction désigne ses successeurs par un vecteur booléen : si l'instruction j est source de la destination de l'instruction i (j est successeur de i), le vecteur conservé en station i a son bit j levé. Pour une fenêtre de n instructions, l'ensemble des vecteurs comprend n^2 bits. (Ramirez *et al.,* 2004) propose une méthode pour réduire le nombre de vecteurs. A chaque terminaison d'instruction, on propage l'arrivée de son résultat à l'ensemble des successeurs grâce au vecteur.

On voit que si le problème du réveil a déjà fait l'objet de solutions raisonnablement extensibles (il semble que le nombre de vecteurs de successeurs nécessaires croisse linéairement avec le degré superscalaire), il n'en va pas de même pour le nombre de ports. Dans ce cas, les diverses solutions proposées jusqu'ici ne sont envisageables que pour des processeurs de degré superscalaire légèrement supérieur à ceux d'aujourd'hui, c'est-à-dire pour passer de 4 à 6 ou 8 instructions par cycle. Au-delà, d'autres solutions sont indispensables pour palier à la limitation du nombre de ports, du nombres de comparateurs et des stations de réservation.

Nous présentons dans la section 2 l'ordonnancement dynamique distribué des instructions. Nous prenons tout au long de cet article l'exemple d'une micro-architecture de degré superscalaire 16. Le partitionnement du banc de registres qui est au cœur de notre proposition est traité dans la section 3. Nous décrivons dans la section 4 le traitement des instructions induit par ce partitionnement. La répartition des destinations, des sources et des opérations des instructions extraites est détaillée en section 5. La section 6 décrit l'envoi des résultats aux sources en attentes. Le lien entre les partitions proposées par la micro-architecture et les unités fonctionnelles est précisé dans la section 7. Enfin la section 8 montre les simplifications obtenues avec cette micro-architecture sur la validation des instructions.

2. Présentation de l'architecture avec ordonnancement dynamique distribué

La figure 1 présente le schéma général de la micro-architecture distribuée que nous proposons. Les tailles données en légende correspondent au cas particulier d'un degré superscalaire 16 avec une fenêtre de 512 instructions. Cette architecture est composée de 24 unités fonctionnelles dont nous ne détaillons pas les types.

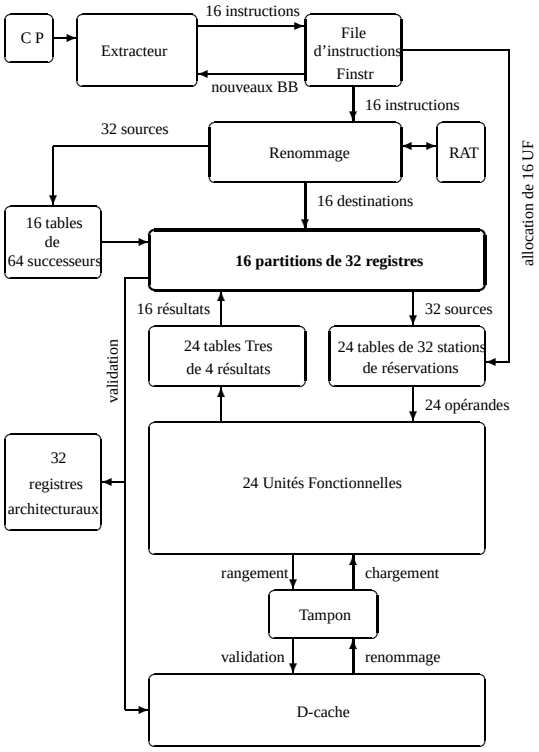

Figure 1. *Micro-architecture distribuée*

En partie haute de la figure se trouve la boucle d'extraction, dont on suppose qu'elle repose sur un cache de traces et un prédicteur multi-sauts. L'extracteur délivre 16 instructions par cycle, ajoutées en queue d'une file des instructions en attente de renommage. La file *Finstr* ne comporte qu'un seul port d'écriture (une ligne de 16 instructions fournie par l'extracteur) et un seul port de lecture (une ligne de 16 instructions renommées solidairement).

Ces 16 instructions sont ensuite envoyées vers l'unité de renommage. Elle prend en charge l'allocation des registres de destination et le renommage des sources. Si l'une des sources est une constante, alors cette dernière est directement envoyée vers la station allouée. Dans le cas contraire, les stations d'attente (au sein des tables de successeurs) et les stations de réservation sont étiquetées avec les identités des instructions renommées.

L'architecture proposée est composée de 32 registres architecturaux visibles de l'utilisateur et de 512 registres de renommage invisibles à l'utilisateur. Ces 512 registres sont distribués équitablement sur 16 colonnes (également dénommées *partitions* dans la suite). Chacune des 16 partitions est composée de 32 registres de renommage.

A chaque cycle, les 24 unités fonctionnelles délivrent leurs résultats en parallèle, qui sont emmagasinés dans des tampons avant aiguillage vers les registres de destination. A chaque cycle, au maximum 16 résultats sont écrits dans 16 destinations (une écriture au plus par partition) et les successeurs sont marqués. A chaque cycle, les successeurs marqués lisent leur valeur en registre (deux lectures au plus par partition) et l'envoient à leur station de réservation (deux sources au plus par unité fonctionnelle). C'est ainsi que s'effectue la diffusion des sources aux unités fonctionnelles. A chaque cycle, une station prête dans chaque unité fonctionnelle lance son calcul.

3. Le partitionnement du banc de registres

Pour lancer une opération sur l'unité fonctionnelle, il est nécessaire de lire ses deux sources. Un banc de registres est bien adapté à cet usage. Les deux ports dont il bénéficie par cellule lui permettent de délivrer les deux sources en parallèle, ce que ne pourrait faire une mémoire avec son unique port de lecture. Notons tout de même que le prix est élevé. Pour r registres de b bits, on ajoute $r * b$ transistors par port dont seuls b servent à chaque accès. Il arrive même parfois que les deux sources soient issues du même registre : un seul port aurait pu suffire.

Pour lancer deux opérations en parallèle, il faut pouvoir lire quatre sources. Chaque cellule du banc de registres doit être dotée de quatre ports. Au total, nous avons $4r * b$ transistors pour $4b$ seulement qui vont servir. Il arrive assez fréquemment que parmi les quatre sources, deux soient issues du même registre.

On comprend qu'à doubler ainsi le nombre de sources à lire, il vient un moment où le prix à payer pour un accès parallèle devient exhorbitant car le nombre de ports se rapproche du nombre total de registres et dans ces conditions, autant accéder systématiquement à tous les registres, ce qui ne nécessiterait plus qu'un seul port.

Cette possibilité n'est pas vraiment une solution car elle ne fait que déplacer le problème de l'accès à l'aiguillage, chaque registre lu devant être acheminé vers la ou les stations où il est attendu.

Un juste milieu est un choix plus judicieux : on partitionne le banc de registres, chaque partition ayant un nombre limité de ports. Partitionner pose deux problèmes : répartir les demandes de lecture et répartir les demandes d'écriture.

Dans l'architecture superscalaire de degré 16 proposée, nous avons divisé le banc de registres en 16 partitions, chacune avec deux ports de lecture pour permettre d'alimenter les deux sources de chaque instruction. Pour répartir les lectures, les instructions ne lisent plus directement leurs sources. Ce sont les partitions qui délivrent aux

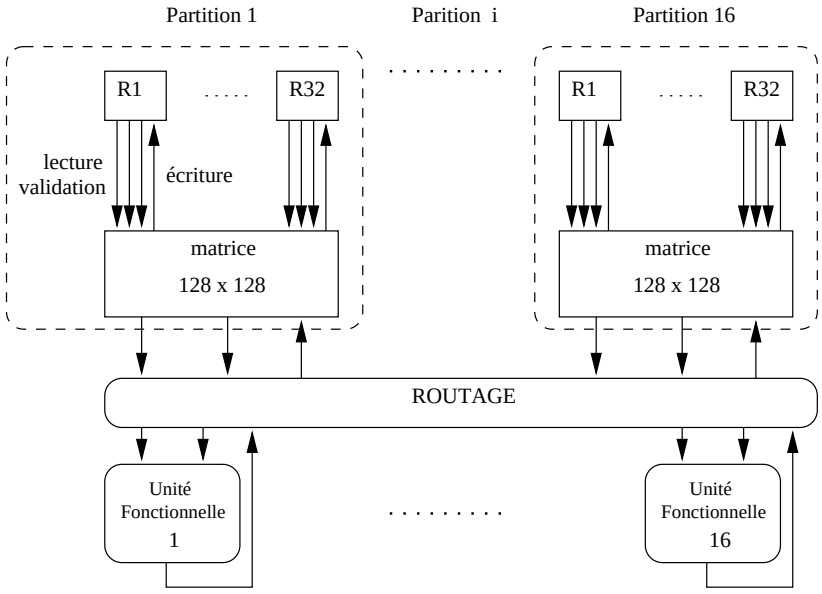

Figure 2. *Partitionnement du banc de registres*

stations autant de sources que le permettent leurs ports. En quelque sorte, on passe d'une logique de diffusion globale des demandes d'accès vers tous les registres à une logique de passage de message ciblée d'un registre vers la station qui en attend une copie.

Partitionner le banc de registres ne permet pas de réduire le nombre de ports d'écriture sauf si l'on contraint les écritures. C'est ce que l'on fait en attribuant aux instructions renommées au cours d'un même cycle des destinations dans des partitions deux à deux disjointes. On distribue ainsi les destinations vers toutes les partitions. En regroupant les écritures par ligne de renommage, il suffit d'un port par partition pour écrire jusqu'à 16 résultats par cycle.

L'ordonnancement dynamique distribué permet de répartir un total de 512 registres de renommage dans 16 partitions, correspondant au nombre maximum d'instructions extraites et renommées par cycle. Chaque partition est composée de 32 registres, 2 ports de lecture servant à alimenter 2 sources d'une instruction, 1 port de lecture servant à la validation dont nous détaillerons le fonctionnement en section 8 et 1 port d'écriture servant à réceptionner un résultat en provenance d'une unité fonctionnelle. Au total, le banc, tout en n'offrant que quatre ports d'accès par registre, est en mesure d'envoyer 32 sources et de recevoir 16 résultats par cycle.

3.1. *Estimation du gain en surface*

Comme indiqué en introduction le coût principal d'un banc de registres est lié au nombre de ports. Comparons le coût en surface et par conséquent en temps d'accès d'une micro-architecture superscalaire de degré 16 sans ordonnancement dynamique distribué à celui d'une micro-architecture avec ordonnancement dynamique distribué.

Soit un processeur superscalaire de degré 16 avec 512 registres. Chaque registre doit disposer de 2 ports de lecture et 1 port d'écriture pour chacune des unités fonctionnelles qui peut y accéder. Ces ports sont organisés sous la forme d'une matrice où les lignes sont les commandes d'accès et les colonnes sont les données. Finalement, le banc de registres centralisé est composé de 512 registres de 32 bits, soit 2^{14} cellules de mémoire. Chaque cellule, qui a 48 ports d'accès, est une matrice 48×48. Le banc est par conséquent une matrice carrée de surface 512 registres $\times 32$ bits $\times 48^2 = 9 \times 2^{22}$.

Considérons maintenant une micro-architecture avec ordonnancement dynamique distribué de même degré superscalaire 16 avec toujours un banc de 512 registres. Ces registres sont divisés en 16 partitions de 32 registres de 32 bits. Chaque registre est accédé à l'aide de 4 ports. Les 16 partitions formant le banc forment une matrice carrée de surface 16 partitions $\times 32$ registres $\times 32$ bits $\times 4^2 = 2^{18}$. La surface du banc de registres distribué représente donc approximativement $1/144$ de celle du banc de registre centralisé. Le rapport des côtés des matrices est quant à lui de $1/12$. Par rapport au banc de registres du pentium 4 actuel, le rapport des surfaces est de $1/4$ et le rapport des côtés est de $1/2$ (128 registres $\times 32$ bits $\times 18^2 > 2^{20}$).

4. Extraction et distribution des instructions

L'extraction et la distribution des instructions dans une micro-architecture superscalaire de degré 16 nécessite quelques aménagements pour pouvoir opérer un ordonnancement dynamique distribué des instructions.

4.1. *L'identification des instructions*

La première étape est l'extraction des instructions. Supposons que l'extracteur accumule les instructions extraites par bloc complet de 16 instructions à renommer (pour un degré superscalaire 16). L'allocation et la libération des identifiants peuvent être organisées pour fonctionner en file. Pour l'allocation, à chaque cycle on renomme un bloc de 16 instructions ce qui nécessite de pouvoir attribuer autant d'identités et de registres de destination. Pour la validation, on peut retarder l'opération pour que les 16 instructions renommées solidairement soient validées également toutes ensembles. Ainsi, à chaque cycle l'entier *queue_Id* fixe le numéro d'ordre de la ligne en cours de renommage. Il sert de queue de file. Un second entier, *tête_Id*, désigne le numéro d'ordre de la ligne en cours de validation. Il sert de tête de file. La file quant à elle

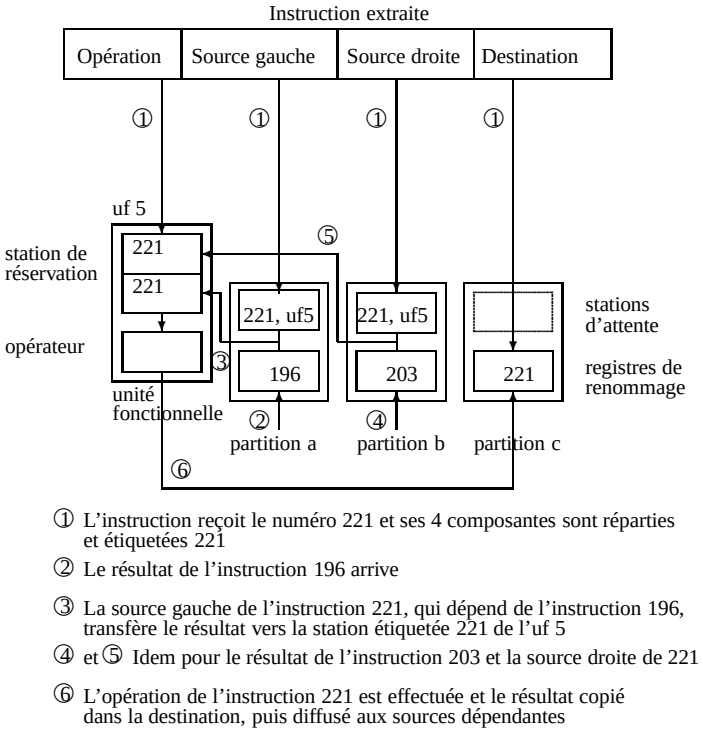

Figure 3. *Algorithme de traitement distribué des instructions*

n'existe pas vraiment car une fois les instructions identifiées et distribuées, il n'est pas nécessaire de les conserver (la fenêtre centralisée n'existe pas).

En fin de cycle, les compteurs *queue_Id* et *tête_Id* sont ajustés. Les extractions sont suspendues quand la file virtuelle des identités en service est pleine.

Les instructions extraites sont conservées jusqu'à leur renommage dans une seconde file, *Finstr*, réelle cette fois. On écrit en queue (*queue_Finstr*) la ligne des instructions extraites. Le pointeur de tête (*tête_Finstr*) désigne la ligne en renommage. La partie gauche de la figure 4 illustre le stockage des instructions extraites et leur identification avant renommage.

4.2. *Le traitement distribué des instructions*

Le partitionnement du banc de registres s'accompagne d'un traitement distribué des instructions décrit par la figure 3. Les instructions extraites reçoivent un identifiant (221 sur la figure) qui permet d'étiqueter leur opération, leurs deux sources et leur destination. Pour 2^n instructions conservées, l'identifiant est un mot de n bits.

L'opération est aiguillée vers une unité fonctionnelle où une station de réservation est allouée. Elle est étiquetée. Deux emplacements sont réservés pour les valeurs des sources. Lorsqu'une source est une constante, celle-ci est directement écrite dans son emplacement.

Les deux sources, une fois renommées, sont liées aux instructions dont elles dépendent par leur identité et à l'unité fonctionnelle allouée pour le calcul par son numéro. Dans l'exemple, la source gauche est fournie par le résultat de l'instruction d'identité 196 et la source droite l'est par l'instruction d'identité 203. Toutes deux sont associées à l'unité fonctionnelle numéro 5.

Chaque source est représentée par une *station d'attente* de la terminaison de l'instruction dont elle dépend. Chaque station d'attente regroupe l'identité de la source et le numéro de son unité fonctionnelle. La station est située dans la partition de la destination dont la source dépend. On regroupe ainsi autour de cette destination les pointeurs sur ses successeurs.

Enfin, la destination se voit allouer un registre de renommage pour le résultat. Nous avons construit les 16 partitions pour limiter le nombre d'écritures à au plus une par partition. Pour garantir cette unicité d'écriture par partition, la destination de l'instruction de rang i dans l'extraction est placée dans la partition i. Par exemple, la destination de l'instruction d'identité 221, de rang $r = 221 \bmod 16$ est placée dans la partition $c = 221 \bmod 16$.

Quand le résultat de l'instruction 196 est établi par l'unité fonctionnelle qui le calcule, sa valeur étiquetée 196 est transmise à la partition contenant la destination 196 (partition $a = 196 \bmod 16$). Les stations d'attente des sources dépendantes marquent ces dernières comme étant désormais prêtes. Ultérieurement, la valeur stockée dans le registre 196 de la partition a y est lue par la station d'attente (attente de la source gauche de l'instruction 221) et transmise à la station de réservation d'identité 221 dans l'unité fonctionnelle 5. Il en est de même pour le résultat de l'instruction 203 (partition $b = 203 \bmod 16$), propagé à la source droite de la même station de réservation. Une fois les deux sources reçues, le calcul de l'instruction 221 commence dans l'opérateur de l'unité fonctionnelle 5. Le résultat est ensuite écrit dans le registre 221 (partition c).

Le principe de propagation des sources est détaillé dans la section 6. L'éclatement de l'instruction en ses composantes permet de répartir les sources et les destinations en faisant en sorte que d'une part chaque registre de renommage ne nécessite qu'un seul port d'écriture et trois ports de lecture (dont un pour la validation) et que d'autre part

une destination et les pointeurs sur les sources qui en dépendent soient physiquement regroupés dans la même partition.

5. La répartition des composantes de l'instruction

Les limites imposées sur le nombre de ports disponibles sur chaque partition (3 ports de lecture, 1 port d'écriture) minimise la surface nécessaire au banc de registres. En revanche ces limites sur le nombre de ports et le partitionnement des registres de renommage choisi implique une politique de répartition des sources et destinations adaptée. L'association entre ces partitions et les unités fonctionnelles nécessite également de regarder de près la façon de répartir les opérations.

5.1. *La répartition des destinations*

La façon dont on organise l'association entre les registres architecturaux et les registres de renommage est déterminante pour la simplicité de l'allocation et la libération de ces derniers. Les deux dernières micro-architectures du Pentium illustrent parfaitement l'alternative.

Dans la micro-architecture P6 (celle du Pentium III), une table (appelée RAT (Hinton *et al.*, 2001)) associe chaque registre architectural au registre de son renommage le plus récent. A la validation en ordre d'une instruction, le registre de renommage RR est copié dans le registre architectural RA qu'il renomme. Si le pointeur stocké en RAT[RA] est RR, l'entrée RAT[RA] est vidée, ce qui indique que le registre architectural RA n'est plus renommé (il est son propre représentant le plus récent). Dans le cas contraire, RAT[RA] n'est pas modifiée : elle contient un renommage postérieur qui n'est pas remis en cause et qui demeure le renommage le plus récent de RA. Dans tous les cas, le registre RR est libéré.

Ce procédé a l'avantage d'allouer et de libérer les registres de renommage dans le même ordre. Ainsi, le banc de registres de renommage peut, pour ces opérations, être organisé en file. Son inconvénient est que, les registres architecturaux étant distincts des registres de renommage, la validation nécessite une copie de registre.

Dans la micro-architecture Netburst (celle du Pentium 4), la table RAT est dédoublée. Une table RATS spéculative conserve les renommages les plus récents. Une seconde table RATD définitive contient les associations après validation. A la validation en ordre d'une instruction i de destination RA renommée RR, on écrit RR dans RATD[RA]. Le registre de renommage RR n'est pas libéré puisqu'il reste pointé par RATD[RA]. En revanche, le registre RR' pointé par RATD[RA] avant la validation de i est libéré.

L'avantage de ce procédé est qu'on ne recopie pas le registre de renommage à la validation. Les registres architecturaux sont mappés au sein des registres de renom-

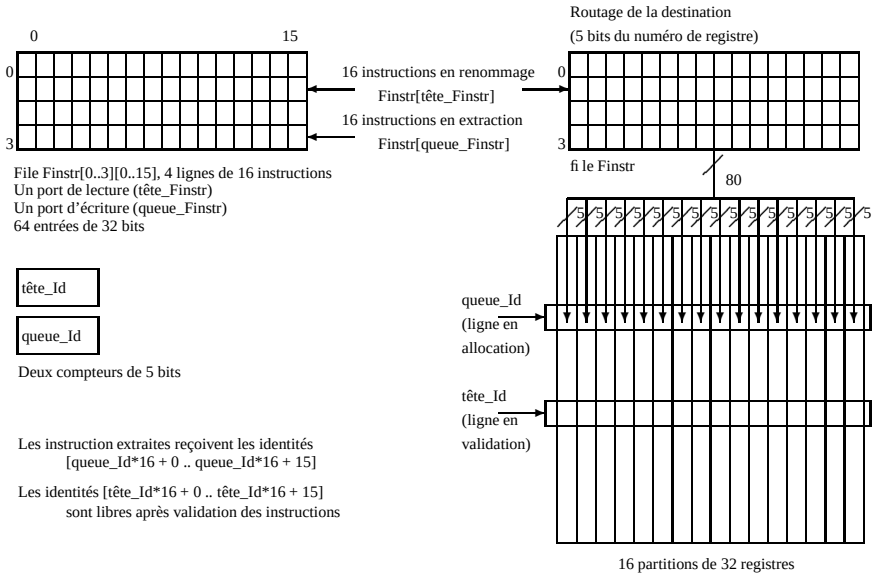

Figure 4. *à gauche : stockage et identifi cation des instructions extraites. à droite : initialisation des registres alloués par le renommage.*

mage, au gré des validations. L'inconvénient est que les registres de renommage sont libérés dans un ordre différent de celui de leur allocation.

Ce dernier argument, dans un ordonnancement dynamique distribué, l'emporte sur celui de la recopie. En effet, la répartition des écritures à la validation permet de les effectuer avec un seul port d'écriture par registre architectural. Pour cette raison, l'ordonnancement dynamique distribué est basé sur le modèle de validation de la micro-architecture P6. Cela permet aussi de disposer de davantage de registres de renommage.

Pour l'implémentation proposée, on partitionne les 512 registres de renommage en une matrice 16x32 dont chaque colonne forme une partition de 32 registres. Aux 16 instructions extraites au même cycle, on alloue une ligne de 16 registres (un par partition). Simultanément, 16 instructions validées libèrent une autre ligne[1].

Les compteurs *tête_Id* et *queue_Id* délimitent les registres alloués : *tête_Id* désigne la ligne libérée et *queue_Id* désigne la ligne allouée.

1. La libération ne peut se faire que solidairement. Cela augmente la latence de la validation, donc le nombre de registres nécessaires, ce que l'organisation distribuée du banc permet.

Chaque registre se compose de trois champs : l'état (2 bits pour 3 états possibles : *libre*, *vide*, *plein*), le numéro de registre architectural de destination (5 bits pour une architecture à 32 registres) et la valeur du registre (64 bits). La partie droite de la figure 4 montre comment la ligne de registres allouée est initialisée.

5.2. *La répartition des sources*

Le renommage associe chaque source à l'instruction produisant la destination dont elle dépend. Cette association se fait par l'identité de l'instruction en question. Faute d'une telle instruction, le renommage fait pointer la source sur le registre architectural qui porte une identité propre, distincte de celles des instructions.

Les sources, une fois renommées, sont distribuées pour que chacune soit placée près de la destination dont elle dépend. L'unité de renommage est reliée à chaque partition du banc de registres par deux bus de 15 bits. Chaque bus permet de transmettre une source. Chaque bus regroupe l'identité de l'instruction à laquelle la source appartient, sur 4 bits complétés par le préfixe *tête_Finstr*, le numéro de la destination dont la source dépend, sur 5 bits, le numéro de l'unité fonctionnelle allouée pour le calcul, sur 5 bits et la position, gauche ou droite, de la source, sur 1 bit.

Encore une fois, si plus de deux sources sont associées à une même partition par le renommage, la transmission peut être sérialisée.

Chaque source prend une place au sein d'une station d'attente. Ces stations sont regroupées en une table couplée à la colonne de registres dont elle dépend. Il y a 16 tables de chacune 64 entrées, soit un total de 1024 entrées pour conserver les sources de 512 instructions.

Une entrée de table se compose de cinq champs : son état (*libre*, *vide*, *plein*), l'identité de l'instruction dont la source fait partie, la position de la source (gauche ou droite), l'identité (partie haute) de l'instruction calculant le résultat dont la source dépend et le numéro de l'unité fonctionnelle.

La partie droite de la figure 5 montre le routage des sources de la file des instructions vers les tables de successeurs.

5.3. *La répartition des opérations*

Pour chaque instruction, le matériel alloue dynamiquement une unité fonctionnelle parmi les 24 disponibles. Cette allocation doit autant que possible répartir le travail équitablement sur chaque unité d'un même type d'opérateur. Nous ne précisons pas quelle méthode d'allocation équilibrée est employée.

Au sein de chaque unité fonctionnelle, on conserve les opérations en attente dans une table dont le nombre d'entrées dépend de la fréquence moyenne d'emploi

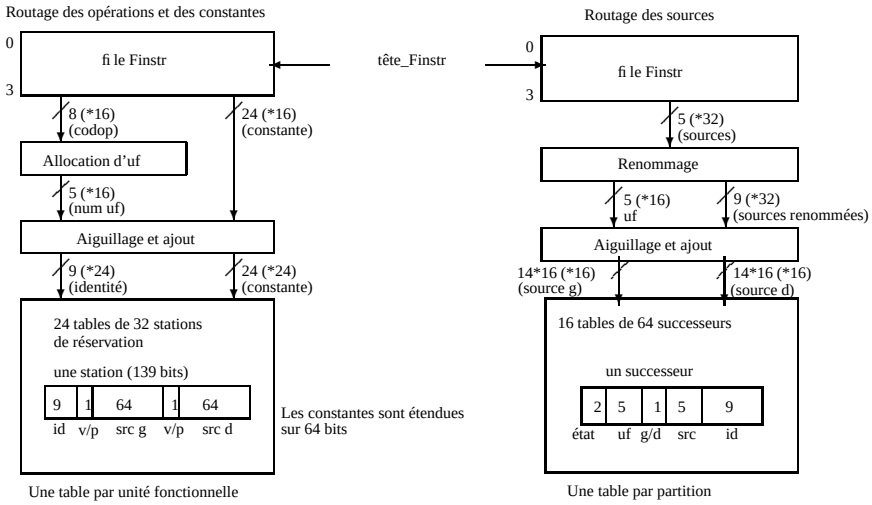

Figure 5. *à gauche : routage des opérations. à droite : routage des sources.*

de l'opérateur. Au total, il faut au moins 512 entrées, ce qui fait une moyenne de $512/24 \approx 21$ entrées par table.

Chaque entrée de table prévoit un champ pour la valeur de la source gauche et un autre pour celle de la source droite, ainsi que des indicateurs *vide/plein* et l'identité de l'instruction ($130 + 9$ bits par entrée pour des sources de 64 bits).

Les instructions en renommage sont conservées dans la file *Finstr* entre les lignes *tête_Finstr* et *queue_Finstr*. Une fois les unités fonctionnelles attribuées à chaque instruction de la ligne *tête_Finstr*, il faut transmettre à chaque unité fonctionnelle la liste des identités des instructions qu'elle doit traiter. On doit aussi acheminer les constantes (une au plus par instruction). On peut supposer qu'on dispose d'un bus de 28 bits par unité fonctionnelle (reliant l'unité de renommage à l'unité fonctionnelle) pour véhiculer une constante (24 bits issus de l'instruction qui sont étendus à 64 bits au sein de l'unité fonctionnelle) et l'identité de l'instruction (4 bits complétés du préfixe *tête_Finstr*).

Si la même unité fonctionnelle est allouée à plusieurs instructions de la même ligne, la transmission de ces instructions via le bus unique menant à l'unité fonctionnelle peut être sérialisée.

La partie gauche de la figure 5 présente le routage des opérations de la file d'extraction aux unités fonctionnelles.

6. La propagation des résultats aux sources en attente

Le résultat issu d'un opérateur est stocké dans une table *Tres* jusqu'à ce qu'il puisse être écrit dans son registre de destination. Il s'accompagne de l'identité i de l'instruction commanditaire. La partie basse de ce numéro identifie le banc contenant la destination.

On relie les unités fonctionnelles aux colonnes de registres par 16 bus, soit un bus par colonne (puisque chaque colonne n'a qu'un port d'écriture). Chaque bus se compose de la valeur transmise (64 bits) et de l'identité de la cible (numéro du registre de destination dans la colonne, sur 5 bits).

Les 24 unités fonctionnelles détiennent à elles toutes jusqu'à $24 \times e$ résultats en attente de transmission (e est le nombre d'entrées de chaque table *Tres* ; pour $e = 4$, on a jusqu'à 96 résultats en attente). Il faut en choisir 16 avec un résultat au plus par colonne de registres. Sans vouloir trop préciser comment cet aiguillage est effectué, remarquons qu'il s'agit d'un problème identique à celui de la sélection des opérations à démarrer parmi celles qui sont prêtes, dont on trouve une proposition de solution dans (Palacharla *et al.*, 1997).

Une fois dans sa colonne i mod 16, un résultat est écrit dans son registre de destination r[$i/16$)]. Le registre passe de l'état *vide* à l'état *plein* et simultanément, toutes les sources portant l'étiquette $i/16$ sont prêtes à être transmises à leur unité fonctionnelle. Cela nécessite 64 comparateurs 5 bits par table de successeurs. La figure 6 illustre la transmission des résultats aux partitions.

C'est le partitionnement des sources par destination (le regroupement d'une destination et de ses successeurs) qui permet de réduire le nombre de comparateurs d'un facteur 16.

7. La diffusion des sources aux unités fonctionnelles

On doit pouvoir transmettre 32 sources par cycle (deux sources par instruction pour 16 instructions). Pour cela, on dispose de deux ports de lecture par colonne de registres. Ainsi, chaque colonne peut délivrer au plus deux sources. A l'autre extrémité, chaque unité fonctionnelle reçoit deux sources, provenant de deux bus. Les 16 colonnes de registres sont liées aux 24 unités fonctionnelles par 32 bus. Chaque bus véhicule une valeur (64 bits) et l'identité de l'instruction dont elle est la source (9 bits). Chaque unité confronte les deux identités qu'elle reçoit à celle de toutes ses stations (deux comparateurs de 9 bits par station). La figure 7 fait apparaître la propagation des valeurs des sources vers les stations de réservation.

L'aiguillage des 32 sources de 64 bits chacune en provenance des 16 partitions vers les 24 unités fonctionnelles est réalisé à l'aide d'une matrice de fonction de transfert. Cette matrice dont les lignes sont les 32 sources de 64 bits chacune route les résultats vers les 24 unités fonctionnelles et permet ainsi toutes les combinaisons possibles entre unités fonctionnelles et sources en provenance des partitions. La taille de la

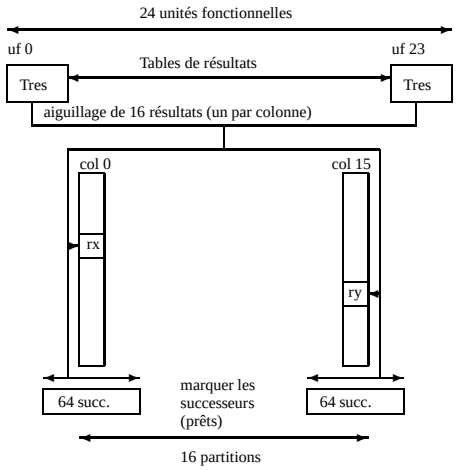

Figure 6. *Routage des résultats.*

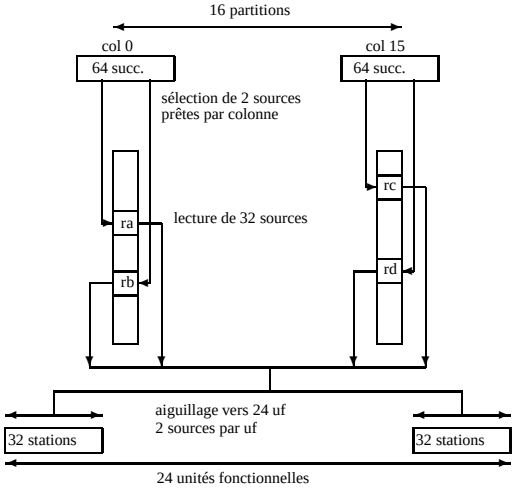

Figure 7. *Propagation des sources.*

table $Tres$ nous garantit l'absence de conflits (une unité fonctionnelle qui a besoin de plusieurs registres en provenance d'une même partition) entre les sources et les unités fonctionnelles.

Diverses options sont envisageables pour la méthode de choix des sources, allant d'une simple répartition équitable par unité fonctionnelle à des heuristiques favorisant les sources les plus bloquantes. Un travail de simulation est nécessaire pour faire apparaître quelle méthode permet la meilleure diffusion.

Il faut noter que cette phase de transmission des sources n'existe pas dans un cœur centralisé. L'efficacité de l'ordonnancement distribué comparativement à celle de l'ordonnancement centralisé repose pour l'essentiel sur la différence de latence entre l'envoi des sources et la lecture d'un banc de registres hiérarchisé. Les remarques de l'introduction laissent penser que si l'efficacité respective des deux méthodes n'est pas claire pour un degré superscalaire modeste, la limitation du nombre de ports ne devrait plus laisser de choix pour un degré superscalaire élevé.

8. La validation des instructions

La validation groupée des instructions simplifie le travail de sauvegarde des résultats. En effet, on ne sauvegarde que les dernières versions des registres : si le registre Ri est modifié par trois instructions d'une ligne, mettons les instructions 2, 7 et 12, on ne procède qu'à l'écriture du résultat de l'instruction 12 dans le registre Ri. Ainsi, nous n'avons qu'une écriture au plus par registre par cycle.

A chaque cycle, on tente de valider la ligne *tête_Id*. Elle est validée si toutes ses instructions sont terminées. Chaque colonne de registres émet l'état de son registre *tête_Id*. Si tous sont pleins, la ligne est validable (elle l'est aussi si toutes les instructions précédant un saut mal prédit sont terminées).

Les registres architecturaux sont répartis à raison de $x = 32/16$ registres par colonne (pour un ensemble de 32 registres architecturaux).

Notons que le registre architectural r, qui se trouve dans la colonne r mod 16, peut être modifié par l'une quelconque des instructions en validation, dont le registre de renommage est éventuellement localisé dans une autre colonne.

Pour reporter la ligne validée *tête_Id* dans les registres architecturaux, on emploie x bus de chacun 16×64 bits (par exemple, 2 bus). Chaque bus atteint un registre architectural par colonne. Il reçoit une copie de la ligne *tête_Id* des valeurs validées. Chaque registre choisit le mot du bus qui le concerne. Un masque d'écriture est établi à partir des dépendances EAE de la ligne *tête_Id*.

9. Conclusion

Nous venons de décrire une micro-architecture distribuée qui offre l'avantage de répartir les requêtes de lecture et d'écriture visant le banc de registres de renommage. La micro-architecture avec ordonnancement dynamique distribué permet d'envisager un banc de registres suffisant pour atteindre un degré superscalaire de 16 et résoudre les limitations des bancs de registres actuels liées à l'augmentation du nombres de ports et du nombre de comparateurs dans les stations de réservation. En effet, le banc de registres proposé n'occupe que $1/4$ de la surface du banc de registres du pentium 4 pour quatre fois plus de registres de renommage.

Un travail conséquent est en cours pour valider cette idée et fixer les détails absents de cet article. Ce travail passe par une modélisation du cheminement des instructions pour faire apparaître les éventuels goulots d'étranglement ou au contraire pour réduire si possible la complexité de certaines structures en fonction du nombre d'instructions conservées, du degré superscalaire ou du nombre d'unités fonctionnelles.

10. Bibliographie

Butts J., Sohi G., « Use-based register caching with decoupled indexing », *Proceedings of the 31st Annual International Symposium on Computer Architecture, 2004*, p. 302-313, 2004.

Cruz J., Gonzalez A., Valero M., Topham N., « Multiple-banked register file architectures », *Proceedings of the 27th Annual International Symposium on Computer Architecture, 2000*, p. 316-325, 2000.

Ernst D., Austin T., « Efficient dynamic scheduling through tag elimination », *Proceedings of the 29th Annual International Symposium on Computer Architecture, 2002*, p. 37-46, 2002.

Gonzalez R., Cristal A., Pericàs M., Veidenbaum A., Valero M., « Scalable Distributed Register File », *Workshop on Complexity-effective Design held in conjunction with the 31st International Symposium on Computer Architecture, 2004*, 2004.

Hinton G., Sager D., Upton M., Boggs D., Carmean D., Kyker A., Roussel P., « The microarchitecture of the Pentium 4 processor », *Intel technology journal, Q1, 2001*, 2001.

Kailas K., Franklin M., Ebcioglu K., « A register file architecture and compilation scheme for clustered ILP processors », *Proceedings of the 8th International Euro-Par Conference on Parallel Processing, 2002*, p. 500-511, 2002.

Kim N. S., Mudge T., « Reducing register ports using delayed write-back queues and operand pre-fetch », *Proceedings of the 17th Annual International Conference on Supercomputing, 2003*, p. 172-182, 2003.

Lebeck A. R., Koppanalil J., Li T., Patwardhan J., Rotenberg E., « A large, fast instruction window for tolerating cache misses », *Proceedings of the 29th Annual International Symposium on Computer Architecture, 2002*, p. 59-70, 2002.

Michaud P., Seznec A., « Data flow prescheduling for large instruction windows in out-of-order processors », *Proceedings of the 7th International Symposium on High-Performance Computer Architecture, 2001*, p. 27-36, 2001.

Palacharla S., Jouppi N., Smith J., « Complexity-effective superscalar processors », *Proceedings of the 24th Annual International Symposium on Computer Architecture, 1997*, p. 206-218, 1997.

Park I., Powell M. D., Vijaykumar T. N., « Reducing register ports for higher speed and lower energy », *Proceedings of the 35th International Symposium on Microarchitecture, 2002*, p. 171-182, 2002.

Ramirez M., Cristal A., Veidenbaum A., Villa L., Valero M., « Direct instruction wakeup for out-of-order processors », *Proceedings of the International Workshop on Innovative Architecture (IWIA'04), 2004*, p. 2-9, 2004.

Sato T., Nakamura Y., Arita I., « Revisiting direct tag search algorithm on superscalar processors », *Workshop on Complexity-effective Design held in conjunction with the 28th International Symposium on Computer Architecture, 2001*, 2001.

Seznec A., Toullec E., Rochecouste O., « Register write specialization register read specialization : a path to complexity-effective wide-issue superscalar processors », *Proceedings of the 35th annual international symposium on Microarchitecture, 2002*, p. 383-394, 2002.

Swenson J., Patt Y., « Hierarchical registers for scientifi c computers », *Proceedings of the International Conference on Supercomputing, 1988*, p. 346-353, 1988.

Weiss S., Smith J., « Instruction issue logic for pipelined supercomputers », *Proceedings of the 11th International Symposium on Computer Architecture, 1984*, p. 110-118, 1984.

Yung R., Wilhelm N., « Caching processor general registers », *Proceedings of the International Conference on Computer Design, 1995*, p. 307-312, 1995.

Zalamea J., Llosa J., Ayguadé E., Valero M., « Hierarchical clustered register fi le organization for VLIW processors », *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, p. 77, 2003.

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com

ANNEXE POUR LE SERVICE FABRICATION

A FOURNIR PAR LES AUTEURS AVEC UN EXEMPLAIRE PAPIER
DE LEUR ARTICLE ET LE COPYRIGHT SIGNE PAR COURRIER
LE FICHIER PDF CORRESPONDANT SERA ENVOYE PAR E-MAIL

1. ARTICLE POUR LA REVUE :
   L'objet. Volume 8 – n° 2/2005

2. AUTEURS :
   Bernard Goossens — David Defour

3. TITRE DE L'ARTICLE :
   Ordonnancement distribué d'instructions

4. TITRE ABRÉGÉ POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :
   Ordonnancement distribué d'instructions

5. DATE DE CETTE VERSION :
   14 novembre 2005

6. COORDONNÉES DES AUTEURS :

   – adresse postale :

   Laboratoire LP2A, équipe DALI,
   Université de Perpignan,
   52 avenue Paul Alduy,
   66860 Perpignan Cedex,

   goossens@univ-perp.fr, ddefour@univ-perp.fr

   – téléphone : 04 68 66 21 35
   – télécopie : 04 68 66 22 87
   – e-mail : goossens@univ-perp.fr

7. LOGICIEL UTILISÉ POUR LA PRÉPARATION DE CET ARTICLE :
   LaTeX, avec le fichier de style `article-hermes.cls`,
   version 1.2 du 03/03/2005.

8. FORMULAIRE DE COPYRIGHT :
   Retourner le formulaire de copyright signé par les auteurs, téléchargé sur :
   `http://www.revuesonline.com`

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com