RENPAR'16 / CFSE'4 / SympAAA'2005 / JC
Le Croisic, France, 5 au 8 avril 2005

Ordonnancement dynamique distribué

Bernard Goossens et David Defour
Laboratoire LP2A, équipe DALI,
Université de Perpignan,
52 avenue Paul Alduy,
66860 Perpignan Cedex,
email : goossens@univ-perp.fr, ddefour@univ-perp.fr

Abstract

Cet article présente un algorithme de traitement distribué des instructions adapté aux micro-architectures superscalaires de degré 16. La méthode consiste à répartir à la fois le banc de registre 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 registre de renommage, permet, tout en augmentant le nombre de registres, de conserver un temps d'accès inférieur au cycle.

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 registre d'un pentium 4 [5] (128 registres de 32 bits avec 18 ports d'accès) est accédé en deux cycles.

La surface occupée par le banc de registre, 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.

Pour passer de p opérations par cycle à $2p$, on doit étendre le banc de registre tant en nombre de mots (de 2^n à 2^{n+1}) qu'en nombre de ports (de $3p$ à $6p$). La surface du banc de registre s'en trouve multipliée par 8 car on double le nombre de cellules dont on quadruple la surface.

On déduit de cela que pour conserver une surface constante, si on double le nombre de ports, il faut en même temps diviser par quatre le nombre de registres. A défaut on peut toujours augmenter la puissance des signaux pour les accélérer, mais la consommation s'en ressent fortement.

Une solution proposée pour augmenter la taille du banc de registre sans dégrader le cycle est de le hiérarchiser [15][17][2][1]. Un premier niveau sert de cache, le banc de registre formant un second niveau. Néanmoins, il faut noter que le premier niveau hiérarchique ne peut bénéficier d'aucune réduction sur le nombre de ports. Par conséquent, si l'on souhaite doubler le nombre de ports d'un banc de registre pour doubler le degré superscalaire du processeur, la hiérarchisation doit diviser par quatre le nombre de registres du premier niveau. C'est à ce prix que la latence d'accès aux registres ne s'allonge pas. 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 au premier niveau pour conserver un temps d'accès de deux cycles. Pour 24 opérations, le premier niveau devrait se limiter à seulement 8 registres. Le taux d'échec en serait gravement affecté[1].

Un certain nombre de travaux ont été consacrés à la réduction du nombre de ports : en partitionnant le banc[6][18], en limitant le nombre de lectures et d'écritures[7][4] ou une combinaison des deux méthodes[14][11].
Le banc de registre 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 [10]. 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, [3] propose de classer les instructions en deux catégories selon qu'elles attendent au moins une source ou aucune. Pour 256 stations, [3] note qu'environ 20% des instructions sont dans la seconde catégorie et ne nécessitent aucun comparateur. Un aménagement supplémentaire permet de ne doter les stations de la première catégorie que d'un seul comparateur.
Une autre possibilité est de hiérarchiser les stations en séparant par exemple les instructions en fonction de leur latence [9][8]. Cela permet de ne laisser qu'une partie réduite des instructions dans les stations de premier niveau.
Une autre voie pour simplifier les stations [16][13][12] est de remplacer l'ensemble des comparateurs par des pointeurs. Chaque instruction désigne ses successeurs par un vecteur booléen : le bit i du vecteur de l'instruction j est positionné si l'instruction i a pour source la destination de j. Pour une fenêtre de 2^n instructions, l'ensemble des vecteurs, organisé en matrice, comprend 2^{2n} bits. Pour réduire la taille de la matrice, [12] propose de conserver pour chaque instruction un pointeur sur son premier successeur et d'ajouter un vecteur aux instructions ayant plus d'un successeur. Pour une fenêtre de 32 instructions, [12] montre que 4 vecteurs suffisent.
Parcequ'elles sont peu extensibles, ces diverses solutions 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. Il faut donc s'orienter vers d'autres solutions pour atteindre 16 instructions par cycle.

2. Le partitionnement du banc de registre

Pour lancer une opération sur l'unité fonctionnelle, il est nécessaire de lire ses deux sources. Un banc de registre 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 registre doit être dotée de quatre ports. Nous ajoutons encore $2r * b$ transistors pour $2b$ 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 registre, chaque partition ayant un nombre limité de ports. Partitionner pose deux problèmes : répartir les demandes de lecture et répartir les demandes d'écriture.
Pour une extraction et un renommage de 2^p instructions par cycle, on divise le banc en 2^p partitions, chacune avec deux ports de lecture. Pour répartir les lectures, les instructions ne lisent plus directement leurs sources. Ce sont les partitions qui délivrent aux 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 registre 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 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 2^p résultats par cycle.
Ainsi, l'ordonnancement dynamique distribué permet de découper le banc de registres en 2^p parties où chaque registre est muni de deux ports de lecture et d'un port d'écriture. Au total, le banc est en mesure d'envoyer 2^{p+1} sources (2 sources par instruction pour 2^p instructions) et de recevoir 2^p résultats par cycle. Chaque partition occupe la surface d'un banc de 2^{n-p} registres (pour 2^n registres au total) à quatre ports (un port de lecture supplémentaire sert à la validation : voir l'avant-dernière section). Par exemple, un processeur de degré superscalaire 16, plutôt que d'employer un banc de 512 registres avec une cinquantaine de ports, serait découpé en 16 partitions de 32 registres à quatre ports. Une telle partition occuperait le seizième de la surface du banc de registre du pentium 4.

3. Le traitement distribué des instructions

Le partitionnement du banc de registre s'accompagne d'un traitement distribué des instructions décrit par la figure 1. 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 dans la partition $c = 221 \bmod 2^p$. La micro-architecture garantit qu'on ne puisse écrire qu'un résultat et ne lire que deux sources par cycle dans chaque partition.
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 $a = 196$

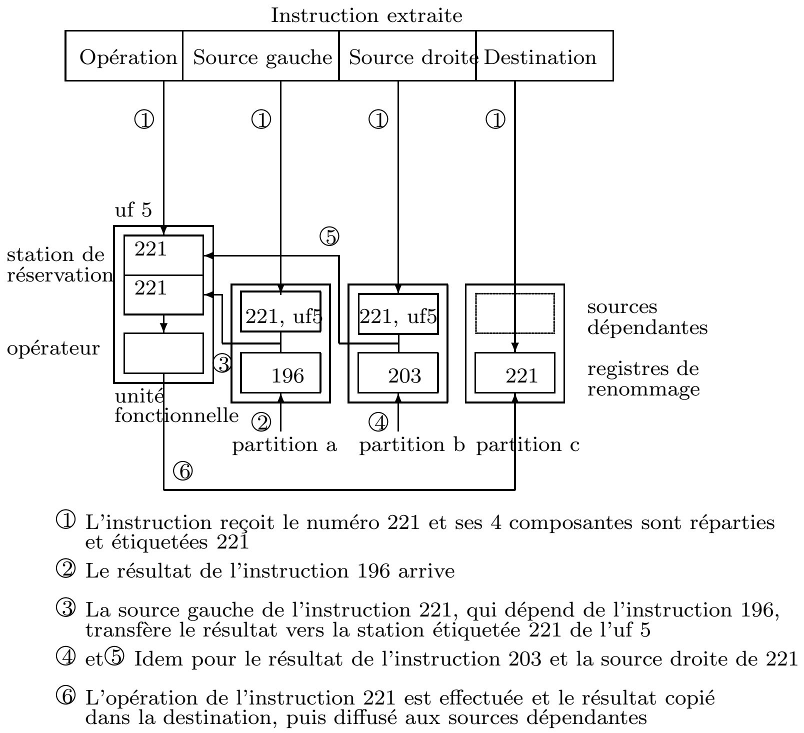

Figure 1: Algorithme de traitement distribué des instructions

mod 16). Les stations d'attente des sources dépendantes indiquent que ces dernières sont prêtes. Ultérieurement, la valeur stockée dans le registre 196 y est lue par la station d'attente et transmise à sa station de réservation dans l'unité fonctionnelle 5. Il en est de même pour le résultat de l'instruction 203 (partition $b = 203$ mod 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).

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 regroupées dans la même partition.

4. Vue générale de la micro-architecture

La figure 2 présente le schéma général de la micro-architecture distribuée que nous proposons. Les tailles de tables et de bus données en légende correspondent au cas particulier d'un banc de 512 registres de renommage.

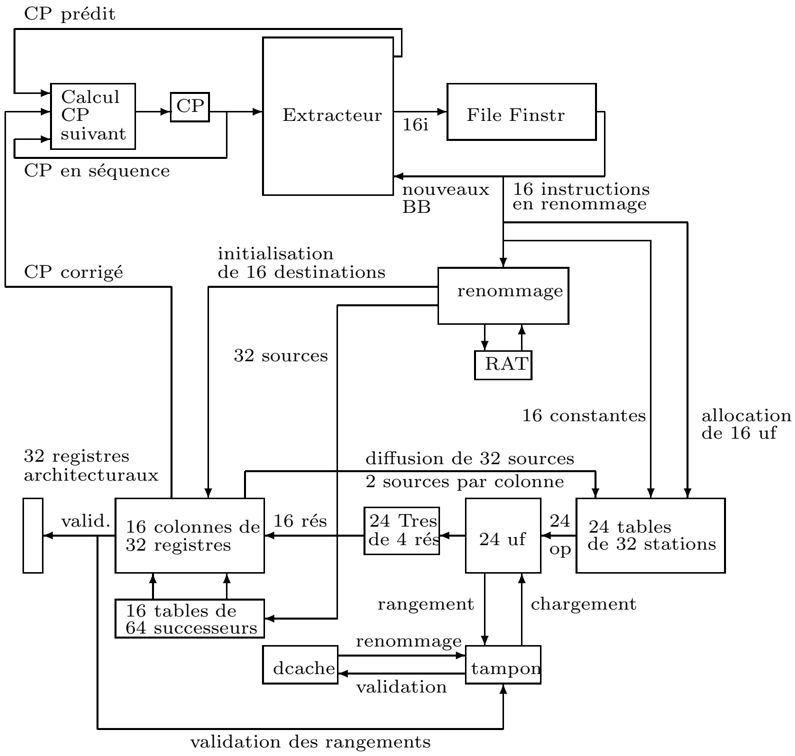

Figure 2: Micro-architecture distribuée

5. L'identification des instructions

Supposons que l'extracteur accumule les instructions extraites par bloc complet de 16 instructions à renommer. 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 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 3 illustre le stockage des instructions extraites et leur identification avant renommage.

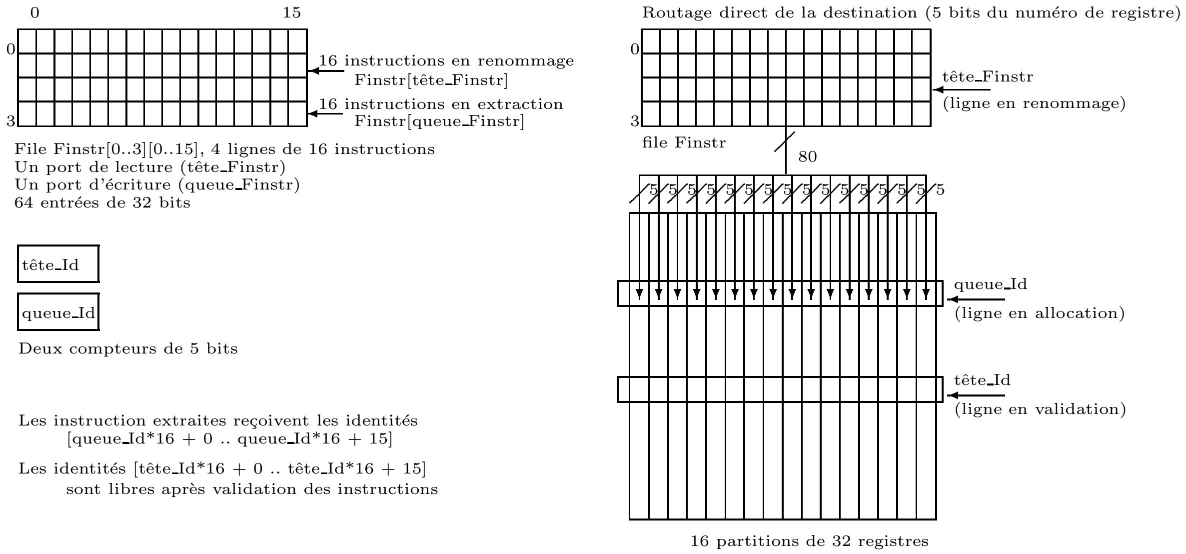

Figure 3: À gauche : stockage et identification des instructions extraites. À droite : initialisation des registres alloués par le renommage.

6. 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 [5]) associe chaque registre architectural au registre de son renommage le plus récent. À 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 registre 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. À 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 renommage, 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 2^n registres de renommage en une matrice 2^{n-4}x2^p dont chaque colonne forme une partition de 2^{n-4} registres (16 partitions de 32 registres pour $n = 9$ et $p = 4$). Aux 2^p instructions extraites au même cycle, on alloue une ligne de 2^p registres (un par partition). Simultanément, 2^p 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.
Chacun des 2^n registres comporte quatre ports d'accès. Il y a deux ports de lecture pour la diffusion aux sources dépendantes, un port de lecture pour la validation et un port d'écriture.
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 3 montre comment la ligne de registres allouée est initialisée.

7. La répartition des opérations

Pour chaque instruction, le matériel alloue dynamiquement une unité fonctionnelle. 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.
Nous supposons que le processeur, qui doit traiter 2^p instructions par cycle (16 par exemple), comprend $u = 2^p + q, q \geq 0$ unités fonctionnelles (24 par exemple) dont nous ne détaillons pas les types.
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 de l'opérateur. Au total, il faut au moins 2^n entrées, ce qui fait une moyenne de $2^n/(2^p + q)$ par table (512/24 par exemple).
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 ($130+n$ bits par entrée pour des sources de 64 bits).
Le routage des opérations vers les unités fonctionnelles s'effectue en deux temps. A chaque unité fonctionnelle, on transmet un vecteur booléen de 2^p bits (16 bits par exemple) des instructions extraites l'employant. Chaque unité alloue le nombre de stations nécessaires (nombre de bits à 1 dans le vecteur) étiquetées par les identités des instructions concernées (*queue_Id* qui désigne la ligne de registres alloués, forme la partie haute de l'identité. La position du bit dans le vecteur forme la partie basse). La partie gauche de la figure 4 présente le routage des opérations de la file d'extraction aux unités fonctionnelles.

[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.

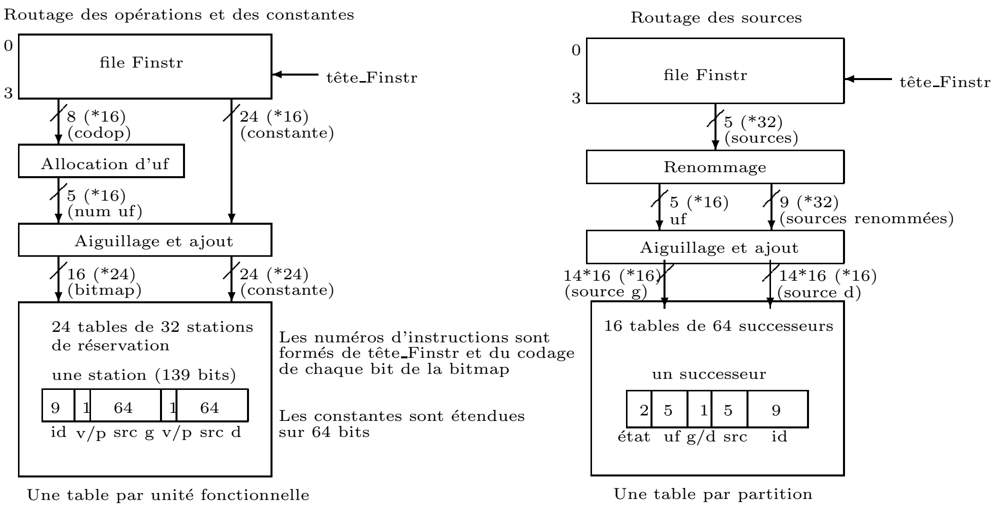

Figure 4: À gauche : routage des opérations. À droite : routage des sources.

8. La répartition des sources

Le renommage associe à chaque source l'identité de l'instruction produisant la destination dont elle dépend. Faute d'une telle instruction, le renommage fait pointer la source sur le registre architectural.

Les sources, une fois renommées, sont distribuées pour que chacune soit placée près de la destination dont elle dépend. Le mode de distribution s'apparente à celui employé pour les opérations. Un vecteur de 2^p bits indique à chaque colonne de registres quelles instructions de la ligne en renommage viennent y chercher leur source gauche (le vecteur vg transmis à la colonne c est tel que vg[i]=1 ssi la source gauche de l'instruction i se trouve en colonne c, ce qui est le cas si son renommage r est tel que $r \bmod 2^p = c$). Un vecteur identique sert pour les sources droites. Chaque vecteur est accompagné d'une table de 2^p entrées contenant les numéros de ligne des destinations dont les sources dépendent (16 entrées de 5 bits par exemple pour des colonnes de 32 registres).

Chaque source prend une place au sein d'une table couplée à la colonne de registres dont elle dépend. Il y a 2^p tables de chacune 2^{n-p+1} entrées (de $2n+4$ bits), soit un total de 2^{n+1} entrées pour conserver les sources de 2^n instructions. Il convient de signaler que les constantes ne sont pas envoyées dans ces tables mais sont directement transmises aux unités fonctionnelles (sources prêtes).

Une entrée de table se compose de cinq champs : son état (*libre*, *vide*, *plein*), l'identité de l'instruction dont la source fait partie (déduite du vecteur de bits et de *queue_Id*), le type de 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.

On transmet donc depuis le renommage jusqu'à chaque table deux bus de $2^p * (n + f)$ bits où f est la taille d'un numéro d'unité fonctionnelle (deux bus de 16x14 bits par exemple). Le bus g contient les sources gauches et le bus d contient les sources droites. Chaque bus est la concaténation de 2^p groupes de $n + f$ bits. Le groupe i du bus g à destination de la colonne

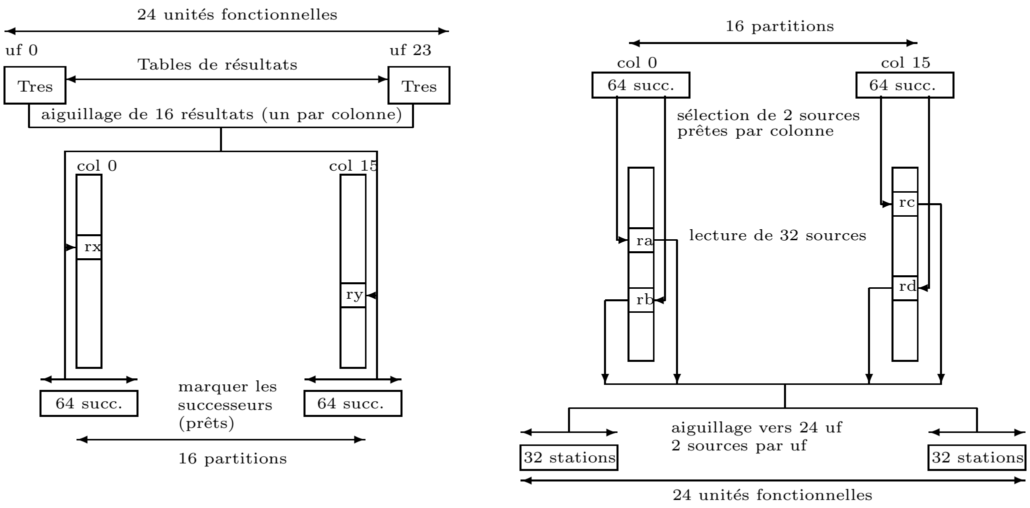

Figure 5: À gauche : routage des résultats. À droite : propagation des sources.

c (partition c) représente la source gauche sg de l'instruction i si le renommage de sg l'associe à une destination d située dans c ($d \bmod (2^p) = c$). Dans le cas contraire, le groupe est sans signification.
Chaque groupe se compose des f bits du numéro d'unité fonctionnelle attribué à l'opération, des $n - p$ bits de poids fort du renommage de la source et des p bits de poids faible de l'identité de la source. La partie droite de la figure 4 montre le routage des sources de la file des instructions vers les tables de successeurs.

9. La propagation des résultats aux sources en attente

Le résultat issu d'un opérateur est stocké dans une table à e entrées 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 2^p 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 $n-p$ bits).
Les u unités fonctionnelles détiennent à elles toutes jusqu'à $u * e$ résultats en attente de transmission (24x4, soit 96 par exemple). Il faut en choisir 2^p avec un résultat au plus par colonne de registre. 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 [10].
Une fois dans sa colonne $i \bmod 2^p$, un résultat est écrit dans son registre de destination r[$i/(2^p)$]. Le registre passe de l'état *vide* à l'état *plein* et simultanément, toutes les sources portant l'étiquette $i/(2^p)$ sont prêtes à être transmises à leur unité fonctionnelle. Cela nécessite 2^{n+1-p} comparateurs $n-p$ bits (64 comparateurs 5 bits par exemple par table de successeurs). La partie gauche de la figure 5 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 2^p.

10. La diffusion des sources aux unités fonctionnelles

On doit pouvoir transmettre 2^{p+1} sources par cycle (deux sources par instruction pour 2^p instructions). Pour cela, on dispose de deux ports de lecture par colonne de registre. Ainsi, chaque colonne peut délivrer au plus deux sources. A l'autre extrémité, chaque unité fonctionnelle reçoit deux sources, provenant de deux bus. Les 2^p colonnes de registres sont liées aux u unités fonctionnelles par 2^{p+1} bus. Chaque bus véhicule une valeur (64 bits) et l'identité de l'instruction dont elle est la source (n bits). Chaque unité confronte les deux identités qu'elle reçoit à celle de toutes ses stations (deux comparateurs de n bits par station). La partie droite de la figure 5 fait apparaître la propagation des valeurs des sources vers les stations de réservation.
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 registre 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.

11. 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 registre 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 $x = 32/(2^p)$ 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 2^px64 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*.

12. 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 registre de renommage.
Un travail conséquent est en cours pour valider cette idée qui 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.

Bibliographie

1. Butts (J.A.) et Sohi (G.S.). – Use-based register caching with decoupled indexing. *In : Proceedings of the 31st Annual International Symposium on Computer Architecture, 2004*, pp. 302–313. – 2004.
2. Cruz (J.L.), Gonzalez (A.), Valero (M.) et Topham (N.P.). – Multiple-banked register file architectures. *In : Proceedings of the 27th Annual International Symposium on Computer Architecture, 2000*, pp. 316–325. – 2000.
3. Ernst (D.) et Austin (T.). – Efficient dynamic scheduling through tag elimination. *In : Proceedings of the 29th Annual International Symposium on Computer Architecture, 2002*, pp. 37–46. – 2002.
4. Gonzalez (R.), Cristal (A.), Pericàs (M.), Veidenbaum (A.) et Valero (M.). – Scalable Distributed Register File. *In : Workshop on Complexity-effective Design held in conjunction with the 31st International Symposium on Computer Architecture, 2004*. – 2004.
5. Hinton (G.), Sager (D.), Upton (M.), Boggs (D.), Carmean (D.), Kyker (A.) et Roussel (P.). – The microarchitecture of the Pentium 4 processor. *In : Intel technology journal, Q1, 2001*. – 2001.
6. Kailas (K.), Franklin (M.) et Ebcioglu (K.). – A register file architecture and compilation scheme for clustered ILP processors. *In : Proceedings of the 8th International Euro-Par Conference on Parallel Processing, 2002*, pp. 500–511. – 2002.
7. Kim (N. S.) et Mudge (T.). – Reducing register ports using delayed write-back queues and operand pre-fetch. *In : Proceedings of the 17th Annual International Conference on Supercomputing, 2003*, pp. 172–182. – 2003.
8. Lebeck (A. R.), Koppanalil (J.), Li (T.), Patwardhan (J.) et Rotenberg (E.). – A large, fast instruction window for tolerating cache misses. *In : Proceedings of the 29th Annual International Symposium on Computer Architecture, 2002*, pp. 59–70. – 2002.
9. Michaud (P.) et Seznec (A.). – Data flow prescheduling for large instruction windows in out-of-order processors. *In : Proceedings of the 7th International Symposium on High-Performance Computer Architecture, 2001*, pp. 27–36. – 2001.
10. Palacharla (S.), Jouppi (N.P.) et Smith (J.E.). – Complexity-effective superscalar processors. *In : Proceedings of the 24th Annual International Symposium on Computer Architecture, 1997*, pp. 206–218. – 1997.
11. Park (I.), Powell (M. D.) et Vijaykumar (T. N.). – Reducing register ports for higher speed and lower energy. *In : Proceedings of the 35th International Symposium on Microarchitecture, 2002*, pp. 171–182. – 2002.
12. Ramirez (M.), Cristal (A.), Veidenbaum (A.), Villa (L.) et Valero (M.). – Direct instruction wakeup for out-of-order processors. *In : Proceedings of the International Workshop on Innovative Architecture (IWIA'04), 2004*, pp. 2–9. – 2004.
13. Sato (T.), Nakamura (Y.) et Arita (I.). – Revisiting direct tag search algorithm on super-

scalar processors. *In : Workshop on Complexity-effective Design held in conjunction with the 28th International Symposium on Computer Architecture, 2001.* – 2001.

14. Seznec (A.), Toullec (E.) et Rochecouste (O.). – Register write specialization register read specialization : a path to complexity-effective wide-issue superscalar processors. *In : Proceedings of the 35th annual international symposium on Microarchitecture, 2002*, pp. 383–394. – 2002.
15. Swenson (J.A.) et Patt (Y.N.). – Hierarchical registers for scientific computers. *In : Proceedings of the International Conference on Supercomputing, 1988*, pp. 346–353. – 1988.
16. Weiss (S.) et Smith (J.E.). – Instruction issue logic for pipelined supercomputers. *In : Proceedings of the 11th International Symposium on Computer Architecture, 1984*, pp. 110–118. – 1984.
17. Yung (R.) et Wilhelm (N.). – Caching processor general registers. *In : Proceedings of the International Conference on Computer Design, 1995*, pp. 307–312. – 1995.
18. Zalamea (J.), Llosa (J.), Ayguadé (E.) et Valero (M.). – Hierarchical clustered register file organization for VLIW processors. *In : Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, p. 77. – 2003.