RENPAR'14 / ASF / SYMPA
Hamamet, Tunisie, 10 – 13 avril 2002

Etude préparatoire à la réutilisation de chaînes

Philippe Reynes, Thierry Haquin, Christine Rochange, Pascal Sainrat

Université Paul Sabatier,
IRIT, 118 route de Narbonne,
31000 Toulouse - France
{reynes, haquin, rochange, sainrat}@irit.fr

Résumé
La réutilisation est une méthode destinée à augmenter le nombre d'instructions exécutées par cycle par un processeur en évitant de refaire des calculs qui ont été déjà effectués par le passé. Elle est d'autant plus performante qu'elle traite des groupes d'instructions. Deux types de groupes ont été étudiés, les blocs de base et les traces qui, dans les deux cas, sont des suites d'instructions exécutées en séquence. La réutilisation de traces se révèle plus performante que la réutilisation de blocs. Comme une trace est plus longue qu'un bloc, il est probable qu'elle contient plus d'instructions dépendantes et donc nécessairement exécutées en séquence et non en parallèle sous réserve d'un nombre suffisant d'unités fonctionnelles. Cet article s'intéresse à la réutilisation de chaînes d'instructions dépendantes. Nos premiers résultats montrent qu'il est nécessaire de faire un compromis entre la taille des chaînes et le taux d'instructions réutilisables. Les caractéristiques des chaînes sont proches de celles des traces mais elles ont l'avantage d'être composées uniquement d'instructions dépendantes. Elles sont donc promises à un meilleur gain en terme de parallélisme d'instructions que les traces.

Mots-clés : processeur superscalaire, réutilisation, instructions dépendantes

1. Introduction

L'augmentation du parallélisme d'instructions est un des buts premiers des recherches en architecture menées depuis dix ans. Parmi les techniques envisagées, la réutilisation tente de supprimer les calculs ayant déja été effectués par le passé en profitant du calcul redondant [3]. Sodani [5] a montré qu'au cours de l'exécution d'un programme réalisant du calcul entier [1], 80% des instructions sont exécutées plusieurs fois avec les mêmes opérandes. Ceci est dû essentiellement à la nature algorithmique du programme.

La réutilisation d'instructions est une méthode qui évite de re-exécuter ces instructions et donc d'utiliser une unité fonctionnelle. Une instruction exécutée est conservée dans une table avec la valeur de ses opérandes et le résultat calculé. Plus tard lorsqu'elle est de nouveau chargée, on vérifie si elle a les mêmes opérandes en entrées que la (les) fois précédente(s). Si cela est vrai, l'instruction est dite "réutilisable" et le résultat est disponible immédiatement et fourni par la table. Le temps de vérifier si une instruction a les mêmes opérandes que par le passé, et d'envoyer le résultat est de un cycle, tout comme le temps d'exécution d'une instruction[2]. La réutilisation ne pouvant être effectuée que tard dans le pipeline, le gain de performance ne sera guère dû qu'aux instructions à longue latence.

La réutilisation de groupes d'instructions semble plus prometteuse. De la même façon que pour les instructions, les entrées et sorties des groupes sont enregistrées dans une table. Les entrées du groupe sont les entrées des instructions qui ne sont pas produites par une des instructions du groupe. Les sorties sont les résultats produits par les instructions du groupe. Plus tard, lorsqu'un groupe est retrouvé, ses opérandes sont vérifiés. Si la vérification est positive, la table fournit alors les résultats de toutes les instructions du groupe et le groupe est dit "réutilisable". Le gain n'est toujours pas assuré car il se peut que toutes ou une partie des instructions d'un groupe soient indépendantes et donc puissent être

[1] Ce sont les seuls programmes auxquels on s'intéressera par la suite.
[2] pour les instructions entières et lorsqu'il n'y a pas de défaut de cache pour les opérations mémoire.

exécutées en parallèle. Par contre, si certaines de ces instructions sont dépendantes, il sera possible de les réutiliser dans le même cycle alors que leur exécution demanderait plusieurs cycles. Dans ce cas, le gain en cycle est égal à la profondeur de l'arbre des dépendances de données construit à partir des instructions du groupe, c'est à dire le temps qui aurait été nécessaire pour les exécuter sans limitation de ressources. Plus l'arbre est profond, plus le gain est important.

Deux types de groupes ont été étudiés, les blocs de base et les traces.

Huang [2] a étudié la répétition de blocs de base et a montré que 16 % à 41 % des blocs sont exécutés plusieurs fois avec les mêmes opérandes pour chacune de leurs instructions. Il montre que malgré un taux d'instructions réutilisées plus faible que pour la réutilisation d'instructions, le fait de réutiliser plusieurs instructions à la fois permet d'obtenir un meilleur gain de performance.

Molina a étudié la réutilisation de traces dans [1]. Une trace est une suite d'instructions exécutées en séquence qui peut contenir un nombre quelconque de branchements et d'instructions, ce qui peut correspondre à plusieurs blocs de base consécutifs. La réutilisation de traces fournit un gain de performance supérieur à la réutilisation de blocs de base.

La construction des blocs et des traces n'est guidé que par la contiguité des instructions qui les composent. Il est très peu probable qu'un bloc de base ou une trace ne possèdent que des instructions dépendantes. Dans le but d'assurer un gain meilleur grâce à la réutilisation, nous proposons de construires des groupes composés uniquement d'instructions dépendantes, groupes que nous appelerons chaîne d'instructions dépendantes ("chaîne" par la suite). Ces chaînes devront être les plus longues possibles pour avoir une profondeur d'arbre de dépendances de données maximum et donc un meilleur gain de performance. A taille égale, une chaîne doit apporter un meilleur gain de performance qu'une trace.

La deuxième partie de cet article décrit une manière de construire des chaînes et la partie suivante décrit les caractéristiques de ces chaînes.

2. Construction des chaînes

Une chaîne est un ensemble d'instructions exécutées en séquence ou non, toutes dépendantes les unes des autres. Un opérande qui n'est pas produit par une instruction de la chaîne est appelé *entrée de la chaîne*. Toute valeur produite par une instruction composant la chaîne est appelée *sortie de la chaîne* car elle est susceptible d'être consommée par des instructions non présentes dans la chaîne.

Dans cet article, on envisage deux méthodes de construction dynamique de ces chaînes. Avec la première, on parcourt les instructions dans l'ordre d'exécution du programme (construction "en avant"), et on ajoute une instruction à une chaîne si l'instruction consomme un résultat produit par la chaîne. La seconde méthode consiste à remonter les instructions dans le sens inverse d'exécution du programme (construction "en arrière"). On ajoute une instruction à une chaîne si elle produit un résultat consommé par cette chaîne. Cette dernière méthode nécessite de mémoriser un ensemble d'instructions exécutées en séquence pour en extraire ensuite des chaînes en partant des dernières instructions exécutées.

Pour que les chaînes ainsi construites puissent être réutilisées, une contrainte à la construction est nécessaire : toutes les instructions productrices des entrées de la chaîne doivent être situées (dans l'ordre d'exécution) avant la première instruction de la chaîne. Dans le cas contraire, les premières instructions de la chaîne seraient probablement déjà exécutées avant que la vérification des entrées de la chaîne ne soit terminée.

Par ailleurs, pour qu'une chaîne soit totalement réutilisée, il est nécessaire que les instructions composant cette chaîne puissent être simultanément présentes dans la fenêtre d'instructions. Il est donc inutile de construire une chaîne qui s'étale exagérément dans le temps. Dans la suite de cet article, on limitera la distance entre les extrémités d'une chaîne à six blocs de base[3].

La figure 1 présente un exemple de code et donne les chaînes produites en fonction de la méthode utilisée : avant ou arrière. Cette figure comporte trois parties : à gauche, le code, au centre, les dépendances de données entre les instructions du code, et à droite les chaînes créées par chaque méthode.

On peut voir que la méthode dite en "avant" ne peut pas gérer de façon triviale les instructions ayant deux opérandes. Il est nécessaire d'effectuer plusieurs tests pour savoir comment continuer la construc-

[3] les résultats de simulation montrent qu'il est inutile d'aller au delà.

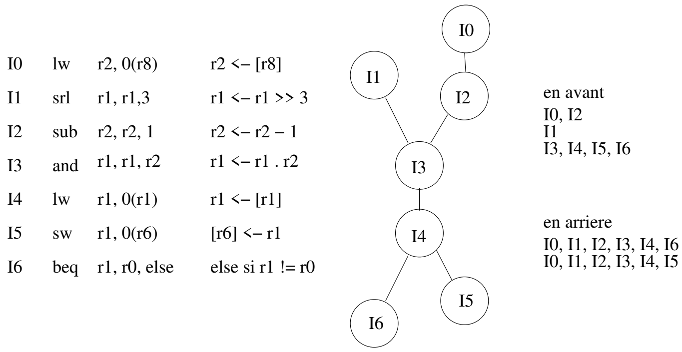


Figure 1: Exemple de code et de construction de chaînes

tion dans le but d'obtenir des chaînes les plus longues possibles. Dans l'exemple donné (figure 1), on ne peut pas insérer l'instructions I3 à la chaîne (I0, I2) à cause de la dépendance entre I1 et I3. Cette entrée serait produite après la première instruction de la chaîne et donc enfreindrait la contrainte que nous avons énoncée précédemment. Pour une raison similaire[4], on ne peut pas insérer l'instruction I3 dans la chaîne composée de la seule instruction I1. La construction de ces deux chaînes doit donc être arrêtée et une nouvelle chaîne doit commencer à partir de I3.
La méthode dite en "arrière" n'entre jamais en conflit avec notre contrainte. Il n'est donc pas nécessaire de réaliser des tests au cours de la construction. Cette solution nous apparaissant plus simple, c'est celle que nous adoptons pour la suite de cette étude. Par ailleurs, nous étudierons deux options possibles imposant des contraintes supplémentaires.
Notre but étant de réutiliser les chaînes construites, il semble peu intéressant d'insérer une instruction non réutilisable dans une chaîne, ce qui condamnerait la réutilisation de la chaîne entière. On introduit donc un filtre qui consiste à ne pas insérer les instructions non réutilisables dans les chaînes. Cette option sera appellée FAIR (Filtre A Instructions Réutilisables). La construction (en arrière) d'une chaîne ne peut commencer que sur une instruction réutilisable et se terminera dès que l'on rencontrera une instruction non réutilisable dont dépend la chaîne. Ceci devrait réduire le nombre de chaînes construites et leur taille, mais devrait permettre de réutiliser un plus grand nombre d'instructions.
Par ailleurs, lorsque l'on voudra réutiliser des chaînes, il sera nécessaire de vérifier les entrées de ces chaînes. Ces entrées peuvent être des registres ou des emplacements mémoire. La vérification des entrées mémoires augmente le nombre d'accès à la hiérarchie mémoire qui est une ressource critique. La deuxième option, appelée FAEM (Filtre A Entrées Mémoires), consiste à construire des chaînes sans entrées mémoires, c'est à dire sans instruction de lecture en mémoire.
Enfin, dans le but d'éviter de faire de la réutilisation d'instructions, nous ne considérerons que les chaînes contenant au moins deux instructions. La réutilisation d'une instruction seule ne permet pas d'apporter un gain significatif et ce n'est pas l'objet de notre étude.

3. Résultat

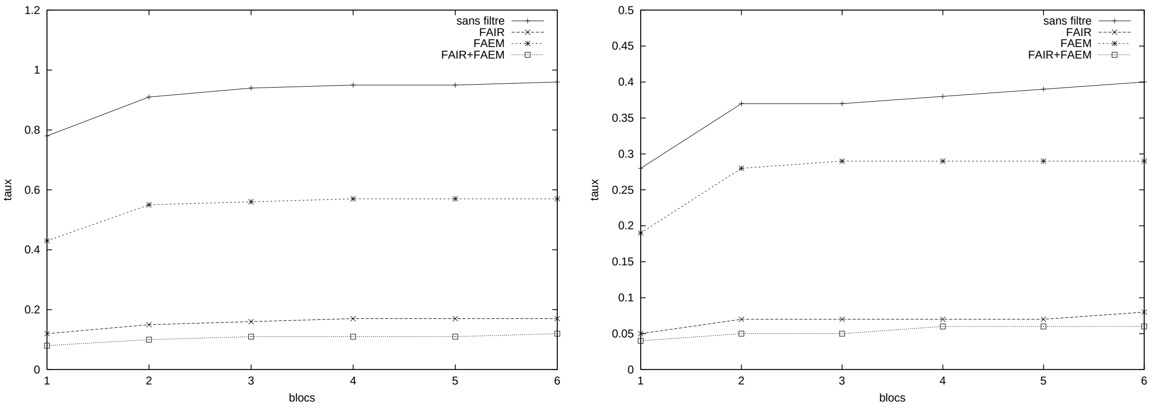
3.1. Simulation

Les simulations ont été réalisées à partir de sim-fast de la suite SimpleScalar qui exécute les instructions séquentiellement. Par conséquent, nous donnons les caractéristiques des chaînes (taille, entrées, ...) et les taux de réutilisabilité des chaînes et des instructions, mais pas d'information sur le gain en performance sur un processeur réel. Les simulations sont effectuées sur cinq cents millions d'instructions aprés avoir passé les deux premiers milliards sur les programmes entiers des spec2000, avec en entrée les jeux de référence fournis.
Une fois construites, les chaînes (valeurs des entrées, des sorties, instructions, comportement des bran-

[4] la dépendance entre I2 et I3 ne respecterait pas notre contrainte.

chements) sont mémorisées dans une table. Cette dernière est consultée pour chaque nouvelle instruction exécutée. Une chaîne est dite réutilisable si ses valeurs d'entrée sont les mêmes que lors de la dernière exécution.
On mémorise dans la table la dernière exécution des chaînes, la table est indexée par l'adresse de la première instruction de la chaîne. Nous pouvons remarquer dans le code donné en exemple dans la partie précédente, qu'une instruction peut commencer plusieurs chaînes. Pour cette étude qui ne veut pas prendre en compte l'implémentation matérielle, on peut mémoriser plusieurs chaînes débutant sur une même instruction mais pas une même chaîne avec des valeurs d'opérande différentes.
Toutes les tables ont 8k entrées, et leur mise à jour est immédiate.

3.2. Caractéristiques



(a) Taux de couverture

(b) Taux de chaînes construites

Figure 2: Taux en fonction du nombre d'instructions exécutées

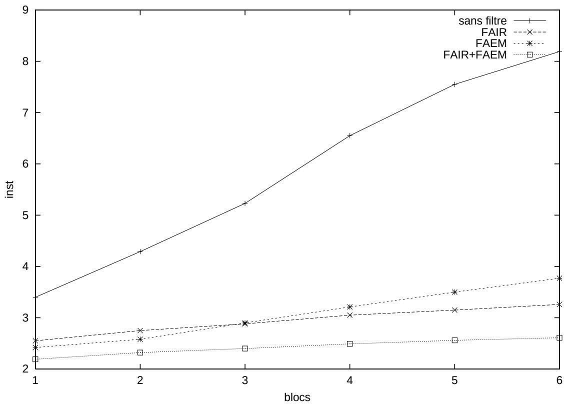
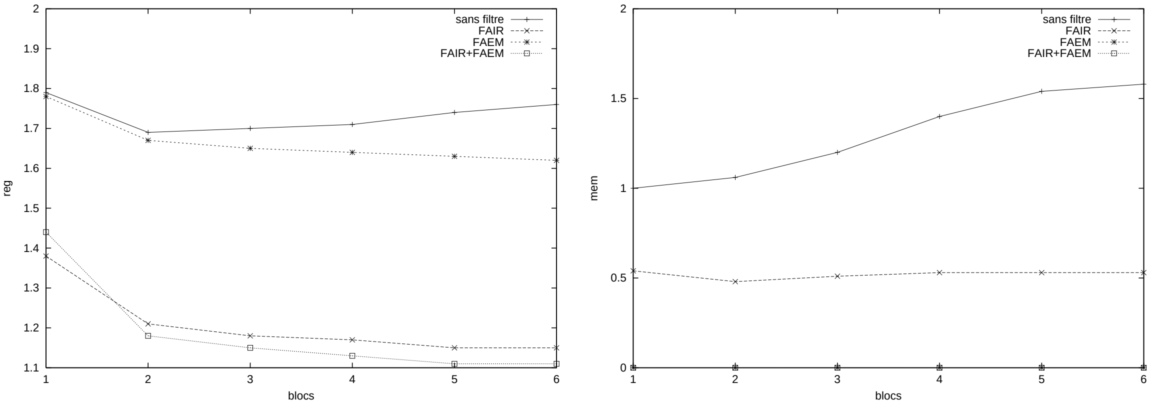


Figure 3: Taille des chaînes (en instructions)

Examinons différentes caractéristiques des chaînes et commençons par le taux de couverture. C'est le nombre d'instructions insérées dans au moins une chaîne, par rapport au nombre d'instructions



(a) registre

(b) mémoire

Figure 4: Nombre d'entrées registre et mémoire

exécutées. La figure 2(a) représente le taux de couverture pour différents types de chaînes en fonction du nombre maximum de blocs sur lequel s'étend la chaîne.

Pour tous les types de chaînes, le taux de couverture semble approcher son maximum à partir de 2 blocs. Même sans filtre, les chaînes ne couvrent pas la totalité des instructions. En effet, comme nous ne considérons que les chaînes d'au minimum deux instructions, les instructions n'ayant pas de dépendances (instructions de types `jmp` immédiat et `load` immédiat) et les instructions isolées dont le résultat est utilisé bien après avoir été produit ne sont pas insérées dans une chaîne.

Le taux de chaînes construites (figure 2(b)) est le nombre de chaînes créées par rapport au nombre d'instructions exécutées. On ne compte que les chaînes les plus longues et pas toutes les sous-chaînes possibles. L'application du filtre FAIR divise par 3 et plus le nombre de chaînes créées. Au delà de deux blocs, de même que pour le taux de couverture, le taux de chaînes créées est quasiment constant.

La taille des différents types de chaînes en fonction du nombre de blocs maximum par chaîne est indiquée figure 3. On constate que les chaînes avec options (FAIR ou FAEM) sont plus courtes, ce qui est logique puisque dans ce type de chaîne on ajoute des conditions d'arrêt de construction de chaînes.

La figure 3 montre que la longueur des chaînes augmente avec le nombre de blocs traversés. L'application des filtres limite grandement la taille des chaînes, empêchant celles-ci de grandir lorsque le nombre de blocs augmente contrairement aux chaînes sans filtre.

Pour les chaînes sans filtre, la taille des chaînes augmente avec le nombre de blocs traversés alors que les taux de chaînes construites et de couverture sont constants, ceci est dû au fait que des instructions sont insérées dans plusieurs chaînes.

Le nombre d'entrées registre (figure 4(a)) diminue légèrement avec le nombre de blocs pour les chaînes avec filtre. Il ne faut pas oublier que l'on construit les chaînes en arrière. Plus on ajoute d'instructions à une chaîne plus on a de chance que cette instruction et la chaîne aient une entrée registre commune, éliminant ainsi une entrée de la chaîne. La diminution du nombre d'entrées registre entre les chaînes réparties sur un bloc et sur deux blocs est un peu plus forte. Si l'on regarde la courbe 2(b), on voit que le nombre de chaînes construites augmente. Il est probable que ces nouvelles chaînes aient un nombre d'entrées registre plus faible, accentuant ainsi la baisse du nombre moyen d'entrées registre des chaînes. Le nombre d'entrées registre des chaînes sans filtre augmente, la taille de ces chaînes augmentant fortement, il semble logique que cela ait une influence sur le nombre d'entrées registre.

Pour les entrées mémoire (figure 4(b)), la situation est quelque peu différente, en effet il y a peu de chances que l'on rencontre une instruction de type store qui produise une valeur lue par une instruction de type load de la chaîne, on a donc très peu de chances de décroître le nombre d'entrées mémoire de la chaîne. Le nombre d'instructions de type load insérés dans les chaînes augmentant avec le nombre

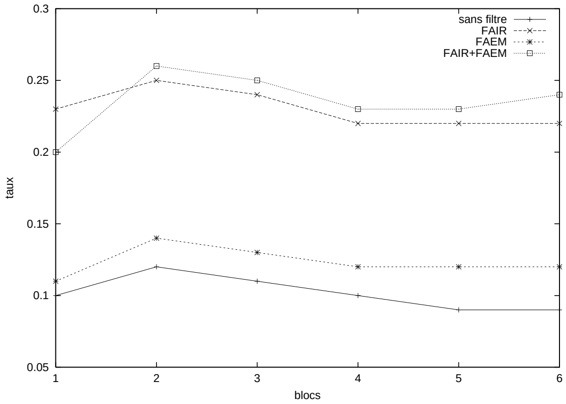
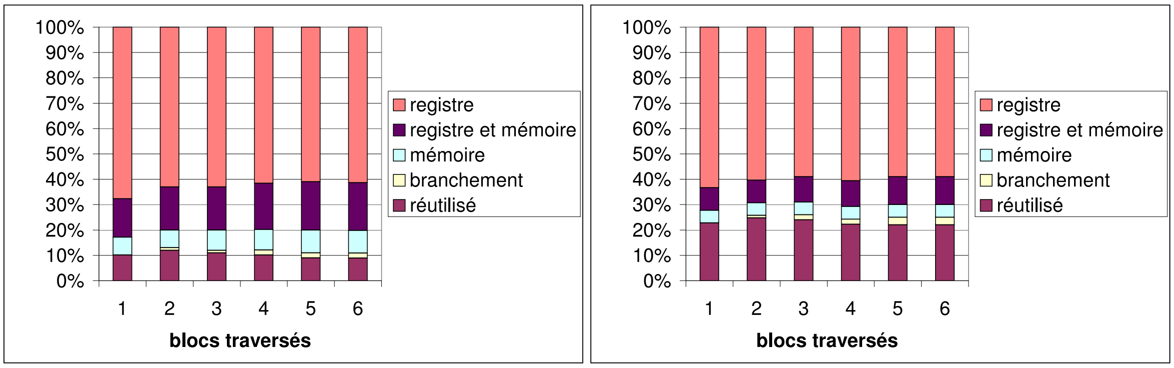


Figure 5: Taux de chaînes réutilisables



(a) sans filtre

(b) FAIR

Figure 6: Causes de non réutilisabilité

de blocs, le nombre d'entrées mémoire augmente pour les chaînes sans filtre. Pour les chaînes FAIR, on voit sur la figure 3 que la taille de ces chaînes n'augmente pas donc on n'insère pas plus d'instructions de type load. Le nombre d'entrée mémoire reste stable. La baisse du nombre d'entrées mémoire entre les chaînes réparties sur un bloc et sur deux blocs est probablement dû à l'augmentation du nombre de chaînes créées (figure 2(b)).

Il est intéressant de noter qu'en moyenne le nombre de registres en entrée et le nombre d'entrées mémoire ne dépassent pas deux. Ce n'est guère surprenant sur les chaînes avec filtre étant donné leur faible longueur. C'est tout aussi vrai sur des chaînes sans filtre dont la longueur moyenne atteint huit instructions. Le nombre de valeurs à vérifier pour déterminer si une chaîne est réutilisable ou non est donc faible par rapport au nombre d'instructions.

3.3. Réutilisabilité

Le taux de réutilisabilité de chaînes représente le taux de chaînes réutilisables (c'est à dire les chaînes dont les entrées n'ont pas varié entre deux exécutions consécutives) par rapport au nombre de chaînes retrouvées dans la table. Sans surprise, la figure 5 montre que le taux le plus élevé est obtenu pour les chaînes ne possédant que des instructions réutilisables (FAIR), taux qui est proche du double de celui des chaînes sans filtre.

Par contre, on peut être surpris de voir un taux inférieur à 30 % alors que les instructions dans ces

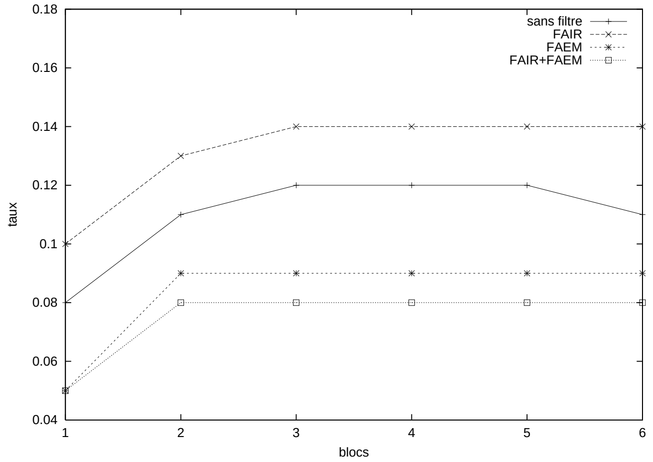


Figure 7: Taux d'instructions réutilisables présentes dans les chaînes

	Taux de couverture	Taille moyenne	registre en entrée	mémoire entrée	Taux d'instructions réutilisables
instruction	100 %	1	-	-	27 %
trace	20 %	3,67	1,89	0,84	17 %
chaîne (sans filtre)	94 %	5,23	1,70	1,20	12 %
chaîne (FAIR)	17 %	2,88	1,18	0,51	14 %

Figure 8: Synthèse des différents types de réutilisation

chaînes étaient censées être réutilisables. Les instructions ont un taux de réutilisabilité de 27 % mais une instruction semblant réutilisable lors de la construction peut ne pas l'être lors de l'exécution suivante.

On peut voir sur la figure 6 que la principale cause de non réutilisabilité (dans plus de 60 % des cas) provient des registres d'entrée dont les valeurs changent. Les chaînes possèdent plus d'entrées registre que d'entrées mémoire, il est donc logique que ce soient les registres qui empêchent majoritairement la réutilisabilité. On constate également que les branchements ne sont pas une grande cause de non réutilisabilité ce qui n'est pas surprenant compte tenu de leur caractère biaisé.

Le taux d'instructions réutilisables indiqué dans la figure 7 est le nombre d'instructions présentes dans des chaînes réutilisables par rapport au nombre d'instructions exécutées.

Sur cette courbe on peut voir l'effet des différents filtres. FAEM diminue le taux d'instructions réutilisables. On a vu que le taux de couverture était plus bas avec FAEM car les instructions de type load ne sont pas pris en compte, il semble donc logique que cette baisse se répercute sur le taux d'instructions réutilisables. FAIR augmente le taux d'instructions réutilisables global de 2 % par rapport aux chaînes sans filtre. Il permet de choisir les instructions ayant une forte probabilité d'être réutilisables dans le futur. L'inconvénient est une taille de chaînes plus courtes, ce qui diminue le potentiel de gain de performance des chaînes. On remarque aussi que l'ajout de FAIR à FAEM ne donne pas de résultat significatif. Le taux d'instructions réutilisables ne change pas, l'augmentation du taux de chaînes réutilisables est compensé par la baisse du taux de couverture.

3.4. Comparaison avec la réutilisation de traces

La figure 8 présente une comparaison avec les réutilisations d'instructions et de traces telles qu'elles ont été présentées dans la littérature. La taille est donnée en nombre d'instructions, le taux de couverture et le taux d'instructions réutilisables en pourcentage du nombre d'instructions exécutées.

La réutilisation d'instructions est implémentée comme indiqué dans [4]. Les traces sont construites avec des instructions réutilisables et contiguës [1], ce qui limite sévèrement leur longueur par rapport aux

traces que l'on trouve dans un cache de traces. Pour rester cohérents avec nos chaînes, nous ne considérons que les traces de plus d'une instruction. Les chaînes sont limitées à 3 blocs traversés.
La taille des traces est proche de celle des chaînes sans filtre mais le nombre d'entrées registre et mémoire est plus important pour les traces que pour les chaînes. Cela augmente la pression sur la file des registres lors de la vérification. Le taux d'instructions réutilisables est supérieur pour les traces (17 % contre 12 %).
Il est possible d'augmenter le taux d'instructions réutilisables des chaînes en utilisant le filtre FAIR. Mais dans ce cas, la taille des chaînes est plus faible. Si on compare les chaînes FAIR et les traces qui ont pour point commun de ne considérer que les instructions réutilisables, on s'aperçoit que les traces ont le taux de couverture et le taux d'instructions réutilisables les plus élevés. On peut expliquer la différence sur le taux de couverture car certaines instructions (comme les instructions de type `jmp` ou des instructions isolées) peuvent être insérées dans une trace, mais pas dans une chaîne. Cela se traduit par un taux de couverture supérieur et une partie des instructions qui sont ajoutées comme les instructions de type `jmp` sont totalement réutilisables. Il est donc logique que les traces aient un taux d'instructions réutilisables supérieur à FAIR. Mais le gain apporté par la réutilisation de ces instructions indépendantes ne devrait apporter qu'un faible gain de performance.
D'un autre côté, il ne faut pas oublier que les chaînes ont l'avantage d'être composées uniquement d'instructions dépendantes, le gain de performance est donc potentiellement supérieur.

4. Conclusion

Nous venons d'étudier la construction et le potentiel de réutilisabilité des chaînes d'instructions dépendantes. Nous pouvons retenir que la construction de longues chaînes et l'utilisation de filtres sont incompatibles. Il est possible d'augmenter le taux d'instructions réutilisables en ne considérant que les instructions réutilisables, mais au détriment de la longueur des chaînes. Il sera nécessaire de faire un choix entre longueur et réutilisabilité.
Les chaînes montrent des caractéristiques (taux de couverture, taille, taux d'instructions réutilisables) proches de celles des traces sans parvenir à les égaler. Les traces réutilisent plus d'instructions mais elles ne sont pas toutes dépendantes. L'avantage des traces sur les chaînes est une utilisation moindre des unités fonctionnelles. Les chaînes réutilisent plus d'instructions dépendantes, augmentant ainsi le parallélisme d'instructions puisque, sans réutilisation, ces instructions devraient être exécutées séquentiellement. Le gain dépendra de l'architecture de base de la machine. Si le nombre d'unités fonctionnelles est faible, les traces fourniront probablement la plus grande amélioration en libérant les unités fonctionelles. Dans le cas contraire, on peut espérer un gain supérieur en réutilisant des chaînes.
Nous comptons poursuivre cette recherche dans deux directions. La première est d'étudier d'autres méthodes de construction de chaînes, en particulier un filtre plus précis et l'implémentation de la réutilisation de chaînes dans des processeurs superscalaires, avec l'objectif d'augmenter le parallélisme d'instructions. La seconde est de continuer à chercher d'autres façons d'exploiter les calculs redondants mis en évidence par Richardson [3].

Bibliographie

1. Antonio Gonzalez, Jordi Tubella, and Carlos Molina. Trace-level reuse. In *Proceedings of the 28th International Conference on Parallel Processing (28th ICPP'99)*, Aizu-Wakamatsu, Fukushima, Japan, September 1999. University of Aizu. Universitat Politecnica de Catalunya, Spain.
2. Jian Huang and David J. Lilja. Exploiting basic block value locality with block reuse. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 106–114, Orlando, Florida, January 9–13, 1999. IEEE Computer Society TCCA.
3. Stephen E. Richardson. Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation. Technical Report TR-92-1, September 1992.
4. Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In 24^{th} *Annual International Symposium on Computer Architecture*, pages 194–205, 1997.
5. Avinash Sodani and Gurindar S. Sohi. An empirical analysis of instruction repetition. In *Architectural Support for Programming Languages and Operating Systems*, pages 35–45, 1998.