Améliorer la performance séquentielle à l'ère des processeurs massivement multicœurs

Nathanaël Prémillieu

ANNÉE 2013

UNIVERSITÉ DE RENNES 1

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

Nathanaël PRÉMILLIEU

préparée à l'unité de recherche IRISA – UMR6074

Institut de Recherche en Informatique et Systèmes Aléatoires
Université de Rennes 1

Améliorer la performance séquentielle à l'ère des processeurs massivement multicœurs

Thèse soutenue à Rennes
le 3 décembre 2013

devant le jury composé de :

Nathalie DRACH-TEMAM
Professeur à l'Université Pierre et Marie Curie (Paris 6) / Rapporteuse

Frédéric PETROT
Professeur à l'ENSIMAG, Institut Polytechnique de Grenoble / Rapporteur

Alain KETTERLIN
Maitre de conférence à l'Université de Strasbourg / Examinateur

François BODIN
Professeur à l'Université de Rennes 1 / Examinateur

André SEZNEC
Directeur de recherche à l'INRIA / Directeur de thèse

Si le savoir peut créer des problèmes, ce n'est pas l'ignorance qui les résoudra.
L'univers de la science, Isaac Asimov

Table des matières

Introduction

Le monde moderne est dominé par les ordinateurs. Ils sont présents presque partout, sous une forme visible comme les micro-ordinateurs et les téléphones portables, ou sous une forme moins visible comme dans les voitures ou les avions. Ils sont essentiels au fonctionnement de beaucoup d'aspects de la vie moderne. Ils permettent d'apporter des fonctionnalités avancées aux divers appareils qui en sont équipés.

Mais comme l'Homme ne se contente jamais de ce qu'il a, toujours plus de fonctionnalités et de réactivité sont demandées à ces ordinateurs. Pour fournir ces nouvelles fonctionnalités tout en restant réactifs, les ordinateurs ont besoin de devenir de plus en plus puissants. On mesure la puissance d'un ordinateur à sa vitesse de traitement de données. Plus précisément, on parle de la puissance de son processeur, unité de calcul au cœur de l'ordinateur.

Un processeur est un composant électronique dont la fonction essentielle est le traitement de données. Pour cela, il lit un flot de directives, appelées instructions, qui pilotent son comportement. Ces instructions décrivent le traitement à effectuer sur les données stockées dans la mémoire de l'ordinateur. Un programme informatique est un ensemble cohérent d'instructions, chacune effectuant un traitement local. La suite de traitements locaux effectués par les instructions permet d'avoir le traitement global effectué par ce programme. Un système informatique est donc composé d'une couche matérielle, essentiellement le processeur, pilotée par une couche logicielle, essentiellement le programme.

Un processeur peut être vu comme une puissante calculatrice qui opère sur des données. Celles-ci sont stockées dans la mémoire du système, que l'on peut voir comme un grand tableau dont chaque case en stocke une. Ces données représentent principalement des nombres, entiers ou à virgule flottante, des chaînes de caractères et des valeurs booléennes. Elles sont en fait codées en binaire (codage en base 2) et c'est sur cette version codée que le processeur effectue son traitement. Il est capable de lire ces données en mémoire, de les manipuler pour effectuer des opérations arithmétiques, des opérations de transformations (inversement, décalage) et des tests dessus, puis de stocker les résultats en mémoire.

Le programme fait aussi partie des données stockées en mémoire que le processeur doit traiter. Ce traitement se compose de plusieurs étapes effectuées sur chaque instruction constituant le programme. Tout d'abord, le processeur doit lire l'instruction en mémoire. Ensuite, il décode la valeur binaire lue pour en déduire le traitement associé à l'instruction en question. Il effectue ensuite ce traitement, qui passe souvent par la lecture de données en mémoire puis écrit le résultat en mémoire. De manière globale, on dit que le processeur exécute les instructions, exécutant ainsi

le programme.

Un programme effectue un travail global, lui-même subdivisé en un ensemble de traitements locaux, réalisé par les instructions. La performance d'un processeur se mesure à la rapidité d'exécution du programme et donc à la rapidité de traitement de chaque instruction. Il existe deux grands axes d'amélioration de la vitesse du traitement des instructions. Le premier est l'augmentation de la vitesse de traitement d'une instruction en particulier. Il passe généralement par l'accroissement de la fréquence de traitement du processeur. Le second est la parallélisation du traitement des instructions. Cette parallélisation peut se faire à différents niveaux. Au niveau le plus haut, plusieurs parties du programme sont exécutées en parallèle par plusieurs unités d'exécution (soit plusieurs processeurs, soit plusieurs cœurs d'exécution sur un même processeur). Cette tendance se généralise dans les processeurs actuels et à l'avenir, ce sont des processeurs avec des centaines, voire des milliers de cœurs d'exécution qui verront le jour, d'où l'appellation de processeurs massivement multicœurs. Cependant, un programme étant essentiellement une séquence d'instructions, il est souvent difficile de pouvoir en exécuter des parties en parallèle tout en s'assurant que le travail du programme est correctement fait. Heureusement, tout en respectant cette sémantique séquentielle, c'est-à-dire le fait que les instructions doivent être exécutées dans l'ordre dans lequel le programme le définit et qui donne donc son sens au programme, il est possible de traiter plusieurs instructions en parallèle au niveau le plus bas.

Dans les processeurs modernes, les différentes étapes de traitement d'une instruction sont organisées sous la forme d'un pipeline de traitement : le traitement d'une seconde instruction commence avant que le traitement de la première soit terminé. Ainsi, pendant le décodage de la première instruction, le processeur peut commencer à lire la seconde instruction en mémoire. Cela permet de paralléliser les différentes étapes de traitement des instructions. On parle alors des différents étages de traitement du pipeline qui effectuent leur travail en parallèle les uns des autres.

Une organisation spécifique du pipeline permet de casser temporairement l'organisation séquentielle des instructions, tout en conservant la sémantique du programme, et d'exécuter les instructions dans le désordre. Le moteur d'exécution exploite donc ce qu'on appelle le parallélisme d'instructions ou *Instruction Level Parallelism* (ILP). En effet, il existe des instructions dépendantes les unes des autres et qui doivent donc être exécutées dans l'ordre défini par le programme. Il existe aussi des instructions indépendantes et donc qui peuvent être exécutées dans n'importe quel ordre les unes par rapport aux autres. C'est cette propriété qui est utilisée par les moteurs d'exécution dans le désordre.

Des instructions spécifiques, les branchements, permettent d'orienter l'exécution vers une partie du programme ou une autre. Certains branchements sont conditionnels : ils n'agissent que si une certaine condition est remplie. Dans le cas où un branchement agit et modifie donc le flot d'instructions, on dit qu'il est pris. On dit qu'il est non pris dans le cas contraire. Concrètement, l'exécution se fait de manière séquentielle, les instructions étant traitées les unes après les autres dans l'ordre donné par le programme. Une instruction de branchement vient casser ce caractère séquentiel en définissant quelle est l'instruction qui la suivra. Cependant, cette information n'est pas connue avant l'étape d'exécution de l'instruction de branchement. Le processeur doit donc attendre cette étape pour savoir quelle instruction il doit aller lire ensuite en mémoire. Cela

empêche le mécanisme du pipeline de fonctionner à plein régime car certains étages se retrouvent sans travail à accomplir. On dit dans ce cas que des bulles sont insérées dans le pipeline.

L'existence de ces bulles est une preuve que le processeur n'est pas exploité à son maximum et qu'un gain de performance est possible si l'on parvient à supprimer ces bulles. De nombreux efforts de recherche ont été accomplis pour résoudre ce problème. On peut dégager deux grands axes de solutions : la prédiction de branchements et les instructions prédiquées. Le premier a essentiellement été pensé pour ce problème mais les principes mis en œuvre sont largement exploitables pour d'autres problèmes, le deuxième est encore plus général.

Le mécanisme de prédiction de branchements indique, de manière incertaine, au processeur quelle instruction suivra l'instruction de branchement courante. Cela permet au processeur de ne pas attendre que le branchement soit exécuté pour commencer à traiter l'instruction suivante. On parle d'exécution spéculative puisqu'elle se base sur une information incertaine. Au moment où le branchement est exécuté, la prédiction est vérifiée. Si celle-ci est correcte, tout se passe bien et le processeur peut continuer son traitement sans problème. Si celle-ci est fausse, on dit que le processeur est sur un mauvais chemin d'exécution, il faut donc corriger l'exécution pour la remettre sur le bon chemin. Le processeur annule alors le travail qu'il est en train de faire sur les instructions après le branchement et recommence son traitement à partir de la bonne instruction. Une mauvaise prédiction induit donc une pénalité au niveau de la performance qui est plus grande que celle induite par l'insertion de bulles. Un mécanisme de prédiction de branchements n'est donc intéressant du point de vue des performances que s'il est suffisamment précis.

La deuxième manière d'éviter le problème des instructions de branchement est de tout simplement les supprimer. En effet, la plupart des branchements sont conditionnels, c'est-à-dire qu'ils ne sont pris que si un certain test est vrai, par exemple si une certaine valeur est supérieure à 0. Ce branchement est dans ce cas utilisé pour sauter par-dessus les instructions à exécuter dans le cas où le test est faux pour aller vers les instructions à exécuter dans le cas où le test est vrai. Il existe un type particulier d'instructions appelées instructions prédiquées. Un prédicat est associé à chacune de ces instructions et celles-ci ne sont exécutées que si le prédicat est évalué à vrai. Dans le cas contraire, elles sont juste traitées comme des instructions sans effet, des no-op (pour *no operation*). Un branchement conditionnel et les instructions qui en dépendent peuvent donc être remplacés par une instruction calculant un prédicat équivalent au test du branchement et des instructions prédiquées par ce prédicat. On appelle cette transformation une *if-conversion* [4].

Contributions

Dans ce document, nous revenons sur les deux grands axes pour traiter le problème de performance posé par les instructions de branchement.

Dans le cas de la prédiction de branchements, plusieurs décennies d'efforts ont permis d'accroître la précision des prédicteurs, jusqu'à atteindre un seuil qui semble désormais difficile à dépasser. En effet, les mécanismes de prédiction de branchements reposent sur une propriété empirique des programmes : le fait que le travail qu'ils effectuent est essentiellement la répétition d'un même sous-traitement. Ainsi, le comportement du programme est relativement cyclique, du

moins aux niveaux des instructions et de leur résultat. Cette constatation concerne surtout les instructions de branchement. Celles-ci ont donc souvent un comportement prévisible, fonction de leur comportement passé. La prédiction de branchements est donc essentiellement un mécanisme qui enregistre le comportement cyclique des branchements et qui produit une prédiction en fonction de la position dans le cycle en question. Cependant, les branchements n'ont pas toujours un comportement cyclique ou celui-ci n'est pas toujours facilement détectable, ce qui fixe les limites de l'efficacité de la prédiction de branchements.

Partant de ce constat, puisqu'améliorer la précision devient difficile, une voie possible pour accroître l'efficacité globale du mécanisme est de réduire le coût d'une imprécision. C'est ce que nous proposons d'accomplir grâce au mécanisme que nous présentons dans la première partie de ce document. Appelé SYRANT (pour *SYmmetric Resource Allocation on Not-taken and Taken paths*), ce mécanisme tente de réduire la pénalité due à une mauvaise prédiction en réutilisant une partie du travail effectué par le processeur sur le mauvais chemin. SYRANT tire parti des propriétés de reconvergence du flot de contrôle et d'indépendance de contrôle. En effet, un branchement conditionnel définit deux chemins d'exécution, le chemin pris et le chemin non pris. Il existe un point dans le flot séquentiel d'exécution où ces deux chemins reconvergent. Ce point est appelé point de reconvergence. Toutes les instructions après ce point sont communes aux deux chemins et donc indépendantes du branchement définissant ces deux chemins. À l'exécution, l'un des deux chemins est choisi. Dans le cas d'une mauvaise prédiction et donc d'un mauvais choix de chemin, il arrive que l'exécution dépasse le point de reconvergence et que des instructions indépendantes du branchement commencent à être traitées. C'est le travail effectué sur ces instructions que SYRANT tente de sauver pour éviter de le refaire sur le bon chemin. Pour simplifier l'implémentation d'un tel mécanisme, SYRANT impose au processeur d'utiliser les mêmes ressources sur les deux chemins. On s'assure donc que les instructions indépendantes occuperont les mêmes entrées dans les structures utilisées par le processeur et qu'ainsi leur travail sera stocké au même endroit, simplifiant sa réutilisation.

La deuxième partie de ce document s'organise autour de l'étude des instructions prédiquées. Nous sommes partis du constat que l'exécution des instructions prédiquées dans un moteur d'exécution dans le désordre pose différents problèmes. Plusieurs solutions ont été proposées pour résoudre ce problème. Elles sont satisfaisantes du point de vue fonctionnel mais pas du point de vue des performances. Nous proposons donc un mécanisme appelé SPREPI (pour *Selective Prediction and REplay for Predicated Instructions*) qui permet d'améliorer la performance d'exécution des instructions prédiquées dans un moteur d'exécution dans le désordre. Ce mécanisme se base sur la prédiction de la valeur du prédicat des instructions prédiquées ainsi que sur plusieurs concepts mis en œuvre dans SYRANT.

Organisation du document

Ce document de thèse s'organise en cinq chapitres : une description de l'organisation d'un processeur moderne, une présentation rapide de la prédiction de branchements et un état de l'art sur l'utilisation de la reconvergence de flot de contrôle et de l'indépendance de contrôle pour limiter les effets d'une mauvaise prédiction, la description détaillée du mécanisme SYRANT, un

état de l'art sur les instructions prédiquées et la description détaillée du mécanisme SPREPI.

Le premier chapitre permet de détailler l'image que nous avons d'un processeur moderne, en particulier celle du moteur d'exécution. Nous revenons sur le principe du pipeline, celui du pipeline superscalaire et enfin sur celui de l'exécution dans le désordre. Ce chapitre est essentiellement inclus dans ce document pour présenter l'architecture de base que les mécanismes SYRANT et SPREPI tentent d'améliorer.

Le deuxième chapitre expose de manière non exhaustive la prédiction de branchements pour insister sur l'état de l'art de l'utilisation de la reconvergence de flot de contrôle et l'indépendance de contrôle pour réduire les pertes dues à une mauvaise prédiction de branchement. Ce chapitre permet de situer notre proposition SYRANT dans le contexte des autres propositions et d'en comprendre les avantages par rapport à celles-ci.

Le troisième détaille la description de notre première contribution : le mécanisme SYRANT. Celui-ci présente d'abord la partie du mécanisme permettant une utilisation symétrique des ressources du processeur sur le mauvais et le bon chemin d'exécution. Ensuite, la partie concernant le sauvetage et la réutilisation du travail effectué par le processeur sur le mauvais chemin sur les instructions indépendantes du branchement mal prédit est décrite. Un étude, menée sur simulateur, des performances d'une implémentation du mécanisme est aussi présentée.

Le quatrième chapitre présente une partie de l'état de l'art sur les instructions prédiquées, leur utilisation pour supprimer les branchements et les propositions pour améliorer les performances de leur exécution. Ce chapitre permet de détailler ce que sont les instructions prédiquées, comment elles sont utilisées pour remplacer les branchements et quels sont les problèmes posés par leur exécution. Plusieurs solutions proposées sont aussi présentées, pour comprendre les différences et avantages de notre proposition par rapport aux autres.

Le cinquième et dernier chapitre est la description détaillée du mécanisme SPREPI. Celui-ci présente d'abord le mécanisme de prédiction de la valeur du prédicat des instructions prédiquées, mécanisme essentiellement dérivé de la prédiction de branchements. Nous montrons que la sélection de l'utilisation d'une telle prédiction est essentielle pour les performances. Ensuite, un mécanisme de rejeu sélectif, essentiellement dérivé des concepts mis en œuvre dans SYRANT, est introduit. Ce mécanisme permet d'éviter de réexécuter certaines instructions après une mauvaise prédiction de prédicat, réduisant ainsi son coût.

Chapitre 1

Architecture d'un processeur moderne

Depuis son invention jusqu'à aujourd'hui, l'architecture d'un processeur a considérablement évolué, principalement en direction d'un seul but : augmenter la vitesse de traitement des instructions. L'architecture est la description d'un jeu d'instructions et de la manière dont est organisé le processeur en vue d'effectuer le traitement de ces instructions. Dans notre cas, nous allons nous intéresser à la partie traitement de l'architecture, c'est-à-dire la micro-architecture. Ce document se concentre uniquement sur l'architecture des processeurs dit scalaires et ne traite pas des processeurs dits vectoriels.

1.1 Les instructions et le programme

Un processeur est avant tout une unité de traitements pilotable. Les directives qui coordonnent ces traitements sont appelées les instructions. On appelle jeu d'instructions l'ensemble des instructions permettant de piloter un processeur implémentant ce jeu d'instructions. On parle aussi de langage machine, puisque le jeu d'instructions est l'ensemble des «mots» qu'il faut utiliser pour «parler» au processeur. Une instruction est généralement une opération à effectuer sur des valeurs.

Un programme est un ensemble cohérent d'instructions, organisé de manière séquentielle. C'est une sorte de «phrase» écrite dans le langage machine. Le binaire d'un programme est la version codée de cet ensemble d'instructions, c'est-à-dire la suite du code binaire des instructions. On parle aussi du code du programme. C'est le binaire qui est stocké en mémoire et qui est traité par le processeur.

1.1.1 Représentation des instructions

L'opération est encodée par un *opcode* et les valeurs d'entrées sont contenues dans des mémoires locales appelées registres. Le résultat est écrit dans un autre registre. La figure 1.1 montre la représentation d'une instruction ainsi qu'un exemple. Celui-ci représente une instruction arithmétique entière effectuant une opération d'addition sur les valeurs contenues dans les registres $R1$ et $R2$ et stockant le résultat dans le registre $R3$.

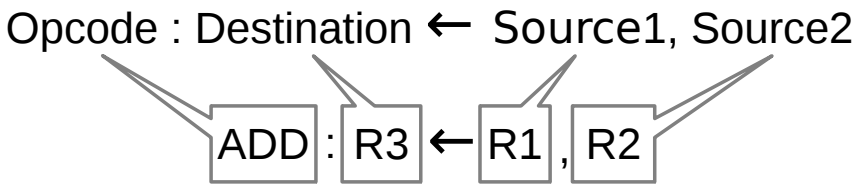


FIGURE *1.1* – *Représentation et exemple d'une instruction.*

1.1.2 Les registres

Le jeu d'instructions définit un certain nombre de registres, appelés registres logiques ou architecturaux, destinés à contenir les valeurs binaires des données traitées par le processeur et donc utilisées par les instructions. Leur nombre et leur utilité varie d'un jeu d'instructions à l'autre, mais on peut généralement distinguer les registres entiers destinés à contenir les valeurs binaires codant des valeurs entières et les registres flottants destinés à contenir les valeurs binaires codant les valeurs flottantes. À ces registres logiques sont associés des registres physiques, c'est-à-dire l'emplacement matériel où sera stockée la valeur dans le processeur. Ces registres physiques sont généralement situés dans un fichier de registres, auquel on peut accéder en lecture et en écriture. Cependant, le nombre d'accès simultanés est limité à quelques uns par cycle, en lecture comme en écriture.

Ces registres sont appelés registres généraux. Le processeur possède aussi un ensemble de registres spéciaux. Ceux-ci servent pour la plupart à définir et à contrôler l'état du processeur.

1.1.3 Registre source et registre destination

On appelle registres sources d'une instruction les registres contenant les valeurs à traiter par l'instruction. Dans la plupart des jeux d'instructions, les instructions ont au plus deux registres sources. Dans le cas de l'exemple de la figure 1.1, les registres sources sont $R1$ et $R2$. On appelle registre destination le registre dans lequel le résultat de l'opération réalisée par l'instruction est stocké. Il n'y a généralement qu'un seul registre destination par instruction. Dans le cas de l'exemple de la figure 1.1, le registre destination est $R3$.

1.1.4 État architectural

L'état architectural du processeur est l'ensemble des valeurs contenues dans les différents registres du processeur à un point donné de l'exécution. Une instruction est essentiellement une opération qui modifie l'état architectural du processeur. Elle peut donc agir sur les registres généraux comme sur les registres spéciaux. Cette opération est faite de manière atomique, c'est-à-dire que deux opérations ne peuvent pas se faire en même temps, elles sont exécutées obligatoirement l'une après l'autre du point de vue de la modification de l'état architectural du processeur.

L'état architectural du processeur est celui vu par le programmeur, celui qui écrit le programme. De son point de vue, les instructions sont donc exécutées les unes après les autres, dans l'ordre dans lequel elles ont été écrites.

1.2 Le pipeline simple

Un processeur moderne est principalement organisé autour de sa structure de traitement des instructions, le pipeline [22]. Celui-ci est séparé en différentes étapes, ou étages, que nous allons détailler.

1.2.1 Les différents étages du pipeline

La figure 1.2 illustre un pipeline simple à cinq étages. Plus un pipeline a un nombre d'étages important plus on dit que le pipeline est profond. Un processeur est le plus souvent synchrone et tous les étages sont synchronisés sur la même horloge. L'exécution de chaque étage se fait en un cycle d'horloge. La fréquence d'un processeur désigne communément la fréquence de cette horloge. Une instruction ne passe d'un étage à un autre que si le traitement de l'étage courant a pu se faire. Sinon l'instruction est bloquée à l'étage en question, attendant de pouvoir être traitée. Cela provoque une blocage global du traitement puisqu'un étage ne peut traiter qu'une instruction à la fois. On dit alors que le pipeline ou le processeur est bloqué.

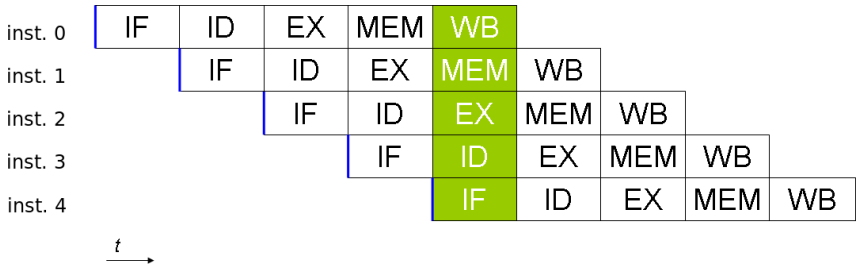


Figure *1.2* – *Exemple d'un pipeline classique à cinq étages.*

Le premier étage est celui de la récupération de l'instruction en mémoire, ou étage de l'*Instruction Fetch* (IF). À cet étage, le processeur lit en mémoire la valeur binaire de l'instruction. Le plus souvent, le processeur dispose d'un cache réservé aux instructions courantes. Un cache est une petite mémoire située dans le processeur, et donc rapide d'accès, qui contient une copie locale d'une partie de la mémoire principale de l'ordinateur, qui est lente d'accès.

Le deuxième étage est celui où la valeur binaire de l'instruction est décodée, ou étage de l'*Instruction Decode* (ID). Cela permet au processeur de savoir quel est le type de l'instruction et quel traitement lui est associé. Les principaux types d'instructions sont :

- les instructions arithmétiques entières : opérations arithmétiques sur les nombres entiers ;
- les instructions binaires : opérations de modification de la valeur binaire, comme les déplacements, les opérations booléennes ;
- les instructions mémoires : opérations de lectures et d'écritures en mémoire :
- les instructions de branchement : définition de l'instruction suivante ;
- les instructions flottantes : opérations arithmétiques sur les valeurs flottantes, qui sont un codage binaire permettant d'approximer des nombres réels.

Les registres sources et le registre destination sont également connus à partir de cet étage.

Le troisième étage est celui de l'exécution proprement dite, ou étage d'*EXecute* (EX). Le traitement spécifique à chaque instruction est fait par le processeur. Selon le type de l'instruction, celle-ci est dirigée vers l'unité fonctionnelle appropriée, c'est-à-dire l'élément du processeur qui implémente la fonctionnalité requise par l'instruction. Par exemple, dans le cas d'une instruction arithmétique entière, l'unité fonctionnelle associée est celle capable d'exécuter l'opération arithmétique en question. On distingue trois principales unités fonctionnelles : l'unité arithmétique entière capable d'exécuter les instructions arithmétiques et les instructions binaires, l'unité flottante capable d'exécuter les instructions flottantes et l'unité mémoire qui traite les instructions mémoires. C'est aussi à cet étage que les valeurs nécessaires à l'exécution de l'instruction sont lues dans les registres sources de l'instruction. Cependant, dans certaines organisations, un étage dédié à cette opération est ajouté avant l'étage d'exécution.

Le quatrième étage est dédié au traitement des instructions mémoires (MEM). Il n'est pas forcément nécessaire que cet étage soit distinct de l'étage d'exécution. Toutefois, la plupart du temps, l'étage d'exécution sert à calculer l'adresse mémoire qui sera accédée par l'instruction et l'étage mémoire sert à envoyer la requête de lecture ou d'écriture à la hiérarchie mémoire. Cette hiérarchie est constituée d'un ensemble de mémoires caches organisé en plusieurs niveaux de plus en plus spacieux mais de plus en plus distants du processeur et donc de plus en plus lents d'accès. Dans les processeurs actuels, on distingue quatre niveaux :

- le cache de premier niveau, situé sur le processeur, est très rapide d'accès, généralement en un ou deux cycles d'horloge, mais très petit (de l'ordre de quelques dizaines de kilo-octets) ; il ne sert généralement que pour les données, les instructions ayant un cache de premier niveau réservé ;
- le cache de deuxième niveau, lui aussi généralement situé sur le processeur, est un peu moins rapide d'accès, en une dizaine de cycles, mais plus gros (de l'ordre de la centaine de kilo-octets, voire un méga-octet) ; dans la plupart des cas, il sert également de cache de deuxième niveau pour les instructions ;
- le dernier niveau de cache, qui peut être sur le processeur ou en dehors, est le plus gros, de l'ordre de plusieurs méga-octets, lent d'accès, généralement en plusieurs dizaines de cycles ; lui aussi sert pour les instructions.
- la mémoire principale, qui est toujours, à l'heure actuelle, en dehors du processeur, est très lente d'accès, de l'ordre de la centaine de cycles, est très grande, de l'ordre du giga-octet, et extensible ; elle stocke tout ce qui est nécessaire au programme, instructions et données traitées par celles-ci.

On peut considérer qu'il existe un niveau supplémentaire, le niveau zéro, constitué des registres physiques utilisés par le processeur pour stocker les valeurs binaires qu'il manipule.

Le dernier étage est celui où le résultat des instructions est écrit dans le registre destination de l'instruction, ou étage de *Write-Back* (WB). Cet étage permet au processeur d'écrire le résultat de l'opération de l'instruction, exécutée par l'unité fonctionnelle adéquate, dans son registre destination. Il marque la fin du traitement de l'instruction par le pipeline.

Du point de vue du programmeur, une instruction est une opération atomique. Pour émuler ce comportement atomique malgré la superposition du traitement des instructions dans le pipeline, une instruction est considérée comme exécutée lorsqu'elle quitte le pipeline. C'est seulement à ce moment-là que la modification qu'elle effectue sur l'état architectural du processeur est validée. C'est donc à cet étage que se fait cette validation.

Ces différents étages de pipeline sont les principaux étages que l'on retrouve dans la plupart des processeurs. Toutefois, le nombre d'étages peut varier si le processeur a besoin de plusieurs cycles pour une étape de traitement particulier. Ainsi, il n'est pas rare d'avoir deux à trois étages pour le *Fetch* ou le *Decode*. De même, l'étage d'exécution peut être largement plus profond selon le type d'opérations à exécuter. Ainsi une opération complexe comme la division nécessite souvent plusieurs dizaines de cycles. Ces différences de longueur de traitement peuvent impliquer une organisation en plusieurs pipelines, chacun dédié à un type d'instruction particulier. Un étage supplémentaire est alors ajouté, l'étage de distribution ou *Dispatch* dans lequel les instructions sont redirigées vers le pipeline de traitement approprié.

1.2.2 Mécanisme de contournement

Une instruction dépendante de la valeur produite par une autre instruction doit non seulement attendre que cette valeur soit produite mais aussi que celle-ci soit écrite dans un registre pour pouvoir ensuite lire ce registre. Pour éviter l'attente de l'écriture puis de la lecture de la valeur, un mécanisme de contournement ou *bypass* est utilisé. Celui-ci permet de faire circuler la valeur produite en sortie d'une unité fonctionnelle directement en entrée de l'unité fonctionnelle qui en a besoin. De plus, ce mécanisme permet de réduire le taux d'utilisation du fichier de registres, qui est souvent l'un des goulots d'étranglement du pipeline.

1.2.3 Parallélisme apporté par le pipeline

L'organisation du traitement des instructions sous forme d'un pipeline permet de paralléliser le traitement de plusieurs instructions. Comme le montre la figure 1.2, les différents étages du pipeline peuvent être actifs en même temps. Ainsi, au premier cycle le premier étage traite l'instruction 0. Au deuxième cycle, pendant que le deuxième étage traite l'instruction 0, le premier étage peut traiter l'instruction 1. Au cinquième cycle, tous les étages sont actifs et traitent une instruction différente chacun. Cette organisation permet donc de commencer à traiter une instruction sans attendre que le traitement de la précédente soit fini. Le pipeline ne permet pas d'accélérer le traitement d'une instruction en particulier. Celui-ci nécessite toujours autant d'étapes et donc de cycles pour se faire. C'est le traitement d'une séquence d'instructions qui se trouve accéléré. Si l'on reprend l'exemple de la figure 1.2, il faudrait $5 \times 5 = 25$ cycles sans pipeline. Avec celui-ci, il faut seulement 9 cycles pour traiter ces cinq instructions.

1.2.4 Limitations

La principale limitation de l'organisation sous forme de pipeline est la dépendance qu'il existe entre certaines instructions. Il y a deux types de dépendances : les dépendances de données et

les dépendances de contrôle.

Les dépendances de données sont de trois types : les dépendances lecture après écriture (LAE), écriture après écriture (EAE), écriture après lecture (EAL). Celles-ci résultent de la réutilisation des mêmes registres par plusieurs instructions, mais seules les dépendances LAE représentent les vrais dépendances sémantiques du programme, c'est-à-dire les dépendances de type producteur-consommateur. Les dépendances EAE et EAL ne sont que des artefacts de la réutilisation des mêmes registres par plusieurs instructions. Il existe des techniques de compilation qui permettent de les éviter, par exemple la forme SSA (*Single Static Assignement*) [17], mais elles ne sont pas forcément applicables à cause du nombre limité de registres logiques dans les jeux d'instructions. Une technique matérielle appelée renommage des registres permet toutefois de supprimer ces fausses dépendances. Cette technique sera détaillée dans la partie 1.4.2 de ce document. L'existence de ces dépendances implique qu'une instruction dépendante d'une autre doit attendre que la première soit exécutée avant de pouvoir elle-même s'exécuter. Cela engendre l'apparition de bulles, puisque l'étage d'exécution peut se retrouver sans travail à faire.

Les dépendances de contrôle sont liées aux instructions de branchement. Celles-ci servent à détourner le flot séquentiel d'instructions vers un nouveau flot séquentiel. En pratique, elles permettent d'implémenter des structures de code telles que les boucles (*while*, *for*), les choix (*switch*) ou les conditionnelles (*if-then-else*). Cette dépendance implique que l'étage de *Fetch* attende l'exécution du branchement pour savoir quelle instruction il doit traiter ensuite. Cette attente provoque l'apparition de bulles et dénote donc une sous-utilisation du pipeline. Plusieurs techniques existent pour limiter ces effets, notamment la prédiction de branchements et les instructions prédiquées. Ces techniques seront détaillées dans les chapitres 2 et 4 respectivement.

1.3 Le pipeline superscalaire

Une limitation supplémentaire liée au pipeline simple est l'attente inutile que subit une instruction lorsqu'une instruction entrée avant elle dans le pipeline subit elle-même une attente. En effet, si une instruction doit attendre pour s'exécuter, tous les précédents étages du pipeline sont aussi bloqués. Et donc toutes les instructions suivant cette instruction en attente doivent elles aussi subir cette attente. L'organisation superscalaire du pipeline [56] tente de résoudre en partie ce problème en permettant à plusieurs instructions d'être traitées en parallèle à chaque étage. La figure 1.3 illustre schématiquement ce concept. Ainsi, dans cet exemple, chaque étage peut traiter jusqu'à deux instructions chaque cycle. On dit alors que la largeur du pipeline superscalaire est de deux.

Un ordonnanceur est nécessaire pour déterminer si les deux instructions qui vont être traitées à l'étage d'exécution sont effectivement indépendantes et peuvent s'exécuter en parallèle. Si ce n'est pas le cas, l'instruction dépendante est mise en attende. Mais cela ne provoque pas nécessairement le blocage du pipeline, cela réduit temporairement la largeur du pipeline au niveau de l'étage d'exécution.

Le gain d'une organisation superscalaire est donc double : plus d'instructions sont traitées chaque cycle et l'effet des dépendances de données est réduit. Le parallélisme d'instructions est donc mieux exploité.

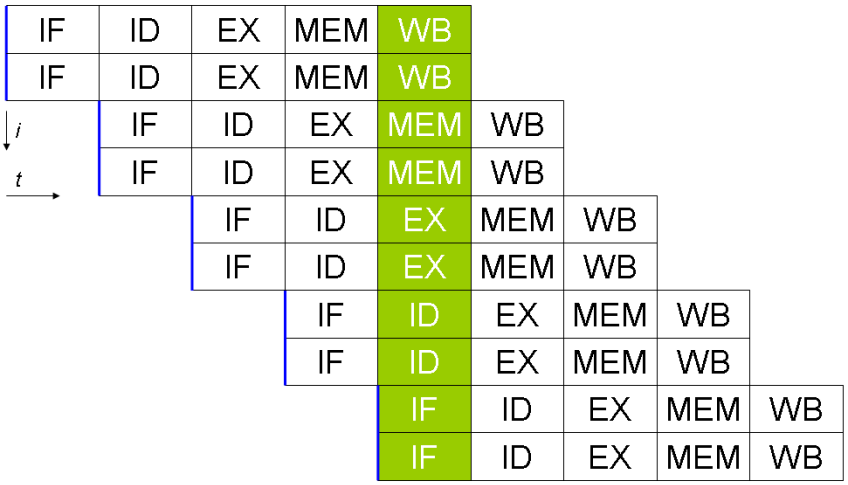


FIGURE *1.3* – *Exemple d'un pipeline superscalaire de largeur 2 à cinq étages.*

Les limitations du pipeline simple sont quand même présentes pour le pipeline superscalaire, même si l'influence indirecte des dépendances de données est réduite. Les dépendances de contrôle posent donc les mêmes problèmes de sous-utilisation du pipeline. Un potentiel plus important est même perdu puisque ce sont plusieurs instructions par cycle qui ne peuvent être traitées.

1.3.1 Limitations spécifiques

La principale limitation de l'organisation superscalaire est son coût. Elle requiert en effet une logique beaucoup plus importante qu'une organisation simple. La complexité des structures est souvent en $O(N^2)$ où N est la largeur du pipeline. Le mécanisme de *bypass* et l'ordonnanceur sont particulièrement coûteux. Ainsi, dans les architectures actuelles, et sûrement futures, on se limite à des largeurs inférieures ou égales à 4.

1.4 Principe de l'exécution dans le désordre

L'organisation superscalaire réduit l'influence exercée par les dépendances de certaines instructions sur les instructions qui en sont indépendantes. Cependant, l'ordre dans lequel les instructions sont entrées dans le pipeline détermine l'ordre dans lequel elles seront traitées. L'étape d'exécution d'une instruction est donc indirectement dépendante de l'exécution des instructions qui la précède. L'exécution dans le désordre cherche à casser cette dépendance pour que l'exécution d'une instruction se fasse dès que ses dépendances de données et de contrôle sont résolues. Cela requiert une nouvelle organisation du pipeline, qui est détaillée dans la suite.

1.4.1 Vue d'ensemble

Le mécanisme d'exécution dans le désordre s'appuie sur plusieurs structures pour permettre aux instructions d'être traitées dans le désordre tout en conservant la sémantique du programme,

c'est-à-dire de telle sorte que les dépendances de données et de contrôle entre instructions soient respectées.

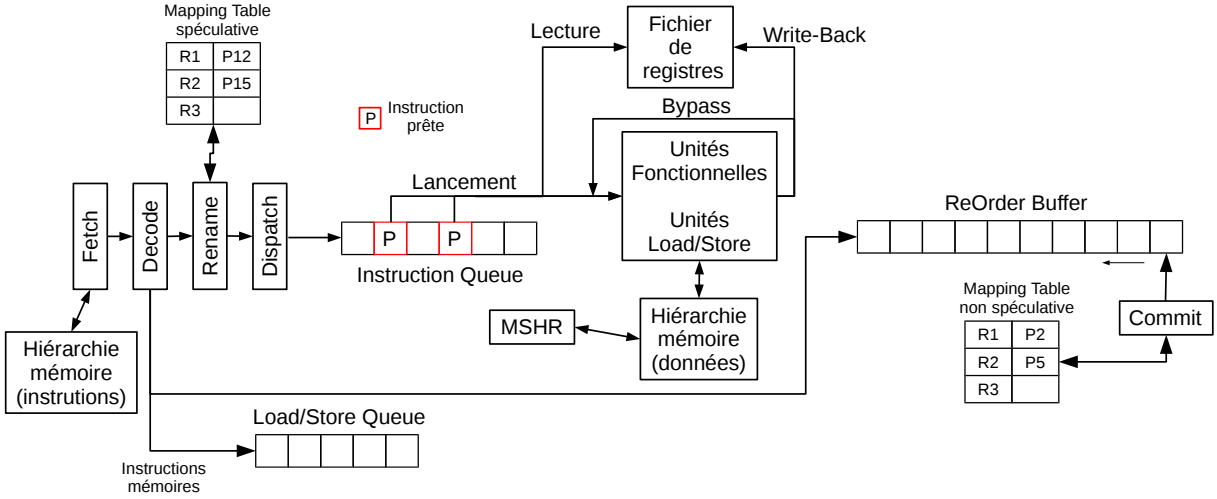


FIGURE *1.4* – *Une organisation possible d'un pipeline dans le désordre.*

La figure 1.4 présente une organisation possible pour un pipeline implémentant l'exécution dans le désordre. Dans les premiers étages de ce pipeline, le *Fetch* et le *Decode*, les instructions sont encore traitées dans l'ordre. En effet, comme le traitement à ces étages est non bloquant, il n'y a aucun gain à pouvoir le faire dans le désordre.

L'étage suivant est celui du renommage des registres (*Rename*). Il sert à calculer les vraies dépendances de données de l'instruction. À cet étage, les instructions sont aussi traitées dans l'ordre. L'organisation dans le désordre commence à l'étage suivant où les instructions sont stockées dans une structure, l'*Issue Queue* (IQ), dans l'attente de pouvoir être exécutées.

En parallèle de leur traitement dans ces premiers étages, les instructions sont enregistrées dans une structure permettant de conserver l'ordre dans lequel elles sont entrées dans le pipeline, pour pouvoir les faire sortir du pipeline et les valider dans ce même ordre. Cette structure est appelée tampon de réordonnancement ou *Re-Order Buffer* (ROB). Les instructions mémoires sont aussi stockées dans une structure dédiée, la file des *loads* (instruction de chargement mémoire) et des *stores* (instruction d'enregistrement en mémoire) ou *Load/Store Queue* (LSQ). Selon les architectures, celle-ci peut-être organisée comme une seule file ou comme une file dédiée aux *loads* plus une file dédiée aux *stores*.

L'ensemble des instructions présentes dans l'IQ et donc en attente d'exécution s'appelle la fenêtre d'instructions (ou *instruction window*). Une fois qu'une instruction est prête à être exécutée du point de vue de ses dépendances, celle-ci est placée dans une liste des instructions prêtes (ou *ready queue*). À chaque cycle, un ordonnanceur choisit les instructions dont l'exécution est lancée parmi celles présentent dans cette *ready queue*. Le nombre d'instructions choisies dépend principalement du nombre d'instructions prêtes à être lancées et de la disponibilité des unités fonctionnelles requises mais aussi de la capacité de l'ordonnanceur. Cette capacité est en général directement liée à la largeur du pipeline. Plus précisément, l'ordonnanceur doit être capable de

lancer au moins autant d'instructions que les étages précédents sont capables de traiter, pour éviter un goulot d'étranglement à ce niveau. Le choix des instructions peut aussi dépendre du type et de la latence d'exécution de celles-ci. La plupart des ordonnanceurs privilégient donc les instructions mémoire et de branchement, tout en ne négligeant pas les instructions les plus vieilles pour éviter un phénomène de famine. Construire un ordonnanceur efficace est un problème assez complexe et la solution de donner la priorité aux instructions les plus vieilles n'est pas forcément la meilleure [27].

Comme dans une organisation simple, une fois lancée, chaque instruction est dirigée vers l'unité fonctionnelle capable de l'exécuter. Le mécanisme de contournement est lui aussi utilisé.

Une fois que l'instruction a été exécutée et que son résultat a été écrit dans son registre destination, elle se retrouve en attente de validation finale. Celle-ci se fait dans le dernier étage du pipeline, l'étage de validation ou étage de *commit*. À cet étage, chaque instruction est validée dans l'ordre dans lequel elle est entrée dans le pipeline grâce au tampon de réordonnancement. L'état architectural du processeur, c'est-à-dire l'état visible du point de vue du programmeur, est modifié par l'instruction à ce moment-là. Les éventuelles exceptions liées à l'instruction sont prises en compte et traitées à cette étape. Une fois l'instruction validée, elle sort du pipeline.

1.4.2 Renommage des registres

Comme dit précédemment, les dépendances de données sont le reflet de la sémantique du programme. Cependant, seules les dépendances LAE sont des vraies dépendances. Les dépendances EAL et EAE sont des fausses dépendances, uniquement liées à la réutilisation de mêmes ressources. Afin d'éliminer ces dépendances, et donc exposer plus de parallélisme d'instructions, un mécanisme de renommage des registres [60] est utilisé. Dans celui-ci, on distingue les registres architecturaux, utilisés dans la nomenclature du jeu d'instructions, et les registres physiques, utilisés par le processeur pour effectivement stocker les données. Le principe, dérivé de celui de la forme SSA, est d'attribuer un nouvel emplacement mémoire, autrement dit un registre physique, chaque fois qu'une variable, autrement dit un registre architectural, est redéfinie. Le processeur doit donc disposer d'une plus grande quantité de registres physiques que de registres architecturaux.

Un nouveau registre physique est donné pour chaque registre destination de chaque instruction. Il est pris dans une file de registres libres, ou *free list*. La table de mise en correspondance des registres architecturaux et des registres physiques ou *mapping table*, est utilisée pour garder le lien entre un registre architectural et le registre physique dans lequel est stockée sa définition courante. La *mapping table* construite au *Rename* est appellée la *mapping table* spéculative car les liens enregistrés dans celle-ci ne sont pas encore validés et ne participent pas à l'état architectural, visible par le programmeur, du processeur.

Cette table est lue pour trouver le renommage des registres sources de l'instruction, afin que le processeur sache dans quels registres physiques les valeurs sources doivent être lues et aussi quelles sont les dépendances de données de l'instruction. Ce mécanisme de renommage doit se faire dans l'ordre pour calculer correctement ces dépendances et ainsi conserver la sémantique du programme.

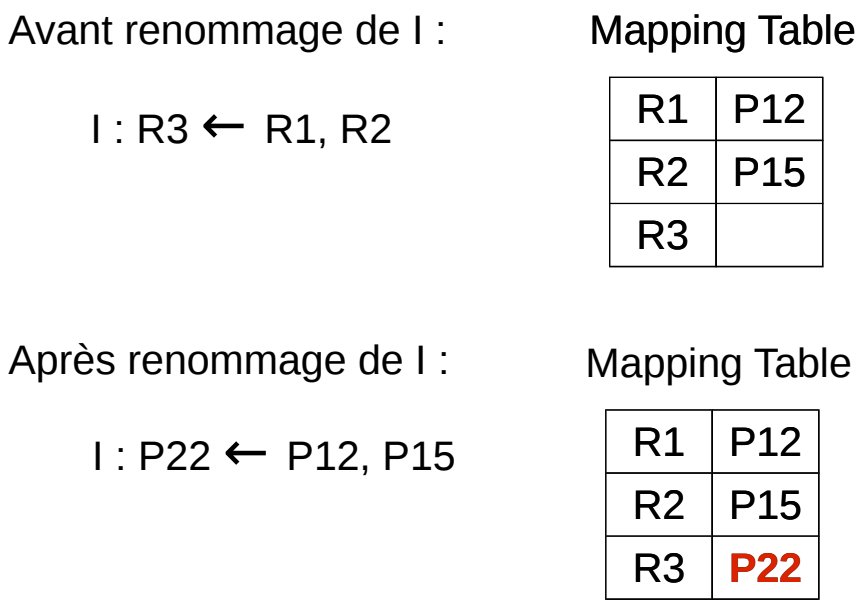


FIGURE ***1.5*** **–** *Exemple de renommage des registres.*

La figure 1.5 montre un exemple de renommage pour une instruction. Les registres architecturaux sont nommés avec la lettre R suivie de leur numéro et les registres physiques sont nommés avec la lettre P suivie de leur numéro. La forme de l'instruction avant renommage utilise des registres architecturaux et celle après renommage utilise des registres physiques. Dans l'exemple, les registres sources de l'instruction I sont $R1$ et $R2$ et son registre destination est $R3$. Pendant son renommage, un nouveau registre physique, $P22$, est attribué à $R3$. La *mapping table* est donc mise à jour avec ce nouveau lien. La table indique que les registres $R1$ et $R2$ sont associés respectivement aux registres $P12$ et $P15$. L'ensemble des ces informations permet de construire la forme renommée de l'instruction I. C'est sous cette forme que l'instruction sera ensuite traitée par le reste du pipeline.

1.4.3 Mécanisme de lancement des instructions

Une fois renommée, une instruction est placée dans une file d'attente, l'IQ. Cette structure est chargée de déterminer si une instruction est prête à être exécutée ou non. Pour cela, elle analyse les informations venant des étages supérieurs du pipeline qui indiquent quels registres sont prêts. Un registre est prêt une fois que la valeur qui doit y être écrite a effectivement été calculée et donc prête à être utilisée. Une fois que l'ensemble des registres sources d'une instruction sont prêts, l'instruction est considérée comme prête et placée dans une autre file : la *ready queue*. L'instruction peut alors être choisie par l'ordonnanceur pour que son exécution soit lancée. Ce choix dépend de la disponibilité de l'unité fonctionnelle nécessaire à l'exécution de l'instruction, de la priorité que donne l'ordonnanceur aux différents types d'instructions et de l'âge de l'instruction, directement relié au temps qu'elle a passé à attendre dans l'IQ.

Le lancement de l'exécution est donc principalement limité par les dépendances de don-

nées de l'instruction et n'est plus directement contraint par l'ordre dans lequel les instructions entrent dans le pipeline. C'est essentiellement l'IQ qui permet d'implémenter l'exécution dans le désordre. Cette structure est donc cruciale pour la performance d'un processeur à exécution dans le désordre. Une de ses caractéristiques primordiales est sa taille, puisque celle-ci conditionne la taille de la fenêtre d'instructions et donc le nombre d'instructions parmi lesquelles le processeur peut chercher à extraire du parallélisme. Cette taille est cependant souvent limitée du fait de la complexité de la logique requise par les fonctionnalités de l'IQ et de l'ordonnanceur.

1.4.4 Réseau de bypass

Le réseau de *bypass* permet comme dans le cas du processeur simple de contourner l'écriture d'une valeur produite par une unité fonctionnelle dans le registre approprié en amenant directement cette valeur à l'entrée de l'unité fonctionnelle qui en a besoin. Sans ce *bypass*, il faudrait attendre que cette valeur soit écrite dans le registre en question pour que celui-ci soit ensuite lu par l'instruction qui l'utilise. Avec ce mécanisme, on considère donc qu'une instruction est prête lorsque l'ensemble des valeurs dont elle a besoin sont prêtes à être acheminées par le réseau de *bypass* au cycle courant (en estimant que le réseau transmet la donnée entre la fin du cycle où elle est produite et le début du cycle où elle est nécessaire et qu'une instruction commence son exécution un cycle après avoir été lancée). La figure 1.6 illustre le gain apporté par le *bypass*.

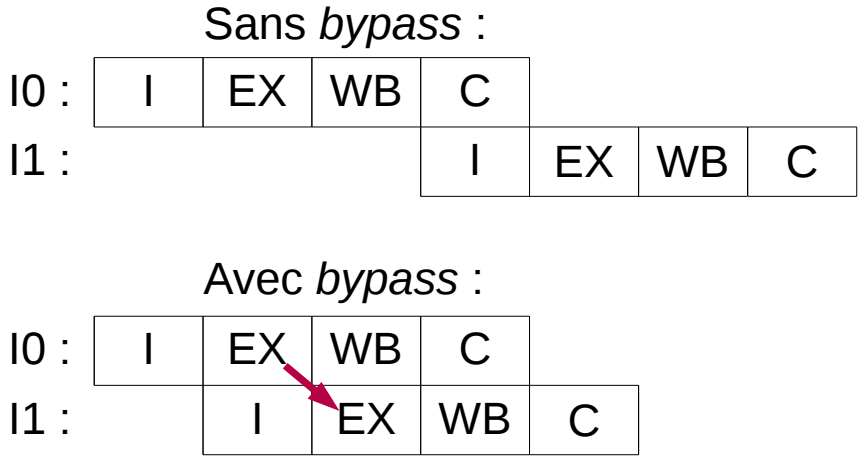


Figure *1.6* – *Ordonnancement avec et sans réseau de* bypass. *$I1$ est dépendante de $I0$ et a besoin de la valeur produite par celle-ci.*

Dans cet exemple, l'instruction $I1$ dépend de l'instruction $I0$. La valeur produite par $I0$ doit donc être fournie à $I1$ pour que celle-ci puisse s'exécuter. Ici, la lecture des registres sources se fait en parallèle du lancement de l'instruction (étage d'*Issue* (I)). Dans la version sans *bypass*, cette lecture doit attendre que l'instruction $I0$ atteigne l'étage de *Write-Back* pour que sa valeur soit effectivement écrite dans son registre destination et puisse ainsi être lue par $I1$. Cette attente provoque une bulle de deux cycles. Avec *bypass*, cette valeur est directement transmise de la sortie de l'unité fonctionnelle qui l'a produite à l'entrée de celle qui en a besoin, évitant ainsi la présence des deux bulles.

1.4.5 File des *loads* et des *stores* et parallélisme mémoire

La file des *loads* et des *stores* ou LSQ sert à stocker les informations relatives aux instructions mémoire, en particulier l'adresse mémoire qui sera accédée et la donnée qui sera écrite dans le cas des *stores*. L'exécution d'une instruction mémoire se fait en deux temps : d'abord le calcul de l'adresse mémoire et ensuite l'accès à la mémoire. Le calcul de l'adresse est traité comme n'importe quelle autre instruction et est soumis aux mêmes problématiques. Le cas de l'accès mémoire est un peu plus complexe à gérer. Pour respecter la sémantique du programme, l'ordre des accès mémoire doit être respecté. Cependant, les *loads* peuvent être exécutés dans n'importe quel ordre les uns par rapport aux autres. Par contre, les *stores* doivent être traités dans l'ordre et les *loads* doivent se faire dans l'ordre par rapport aux *stores*. Plus précisément, un *load* doit être lancé obligatoirement après un *store* si la lecture en mémoire se produit à la même adresse qu'un *store* précédent. Comme les *loads* et les *stores* sont insérés dans la LSQ dans l'ordre, cette structure permet naturellement de vérifier l'ordonnancement mémoire.

Pour conserver l'ordre entre les *stores*, l'accès mémoire correspondant est uniquement exécuté lorsque le *store* atteint l'étage de *commit* et donc dans l'ordre. Lorsque l'accès d'un *load* est prêt, la LSQ vérifie que les adresses de tous les *stores* plus anciens que le *load* sont connues pour s'assurer qu'aucun conflit n'existe avec un des ces *stores*. Si c'est le cas, le lancement du *load* est retardé jusqu'à ce que la valeur du *store* soit prête. À ce moment-là, la valeur du *store* est directement transmise au *load*, sans devoir faire appel à la hiérarchie mémoire. On parle du renvoi de l'écriture ou *store forwarding*. Dans le cas où l'adresse d'un *store* n'est pas encore connue, le processeur peut adopter deux comportements. Dans le cas où il est conservatif, il part du principe que ce *store* peut être en conflit avec le *load* et il retarde donc son exécution. Dans le cas où il est optimiste, il estime que le *store* n'est pas en conflit et que l'accès mémoire du *load* peut donc s'effectuer. L'accès est donc lancé spéculativement, et sa validité doit être vérifié lorsque l'adresse du *store* devient disponible. Ce mécanisme de vérification est en fait exécuté lorsque l'accès du *store* est lancé, au *commit*. À ce moment-là, le *store* vérifie l'ensemble des *loads* plus jeunes que lui dont l'accès a déjà été fait. Si l'un des *loads* est en conflit, le pipeline doit être vidé et l'exécution doit recommencer à partir de ce *load*.

Lors d'un accès à la hiérarchie mémoire, si c'est un succès (la donnée est présente dans le premier niveau de cache), la hiérarchie mémoire peut être accédée de nouveau au cycle suivant. Si l'accès est un défaut (la donnée doit être demandée dans le niveau de cache suivant), la hiérarchie mémoire ne peut être accédée tant que la donnée n'est pas revenue et le lancement de nouveaux accès mémoire est donc bloqué. Pour éviter cet état et ainsi permettre du parallélisme mémoire, des registres spéciaux contenant toutes les informations concernant l'accès sont utilisés. Ils sont appelés registres de statut du traitement du défaut ou *Miss Status Handling Registers* (MSHR). Le nombre de MSHR définit donc le nombre de défauts que la hiérarchie mémoire peut traiter en parallèle. Comme ces défauts sont traités en parallèle, et non plus de manière successive, les différentes latences ne se cumulent plus, réduisant drastiquement l'impact sur les performances. Les MSHR ne sont pas directement liés à l'organisation dans le désordre et peuvent être employés dans des organisations plus simples du processeur.

1.4.6 Tampon de réordonnancement et validation des instructions

Le tampon de réordonnancement, ou *Re-Ordering Buffer* (ROB), sert à conserver l'ordre dans lequel les instructions sont entrées dans le pipeline et donc l'ordre dans lequel elles doivent être validées. La validation d'une instruction consiste à valider son résultat. Elle se fait à l'étage de *commit*. Pour les *stores*, il s'agit d'envoyer la requête d'écriture à la hiérarchie mémoire. Pour les autres instructions, cela consiste à valider le lien entre le registre destination architectural de l'instruction et le registre physique associé. Ainsi, la valeur courante de ce registre architectural est modifiée du point de vue du programmeur. Une seconde *mapping table* est donc utilisée au *commit* pour conserver les liens validés entre les registres architecturaux et les registres physiques. Elle est appelée *mapping table* non spéculative, car elle représente l'état architectural du processeur au niveau des registres, visible par le programmeur. De son point de vue, l'instruction est considérée comme exécutée une fois cette l'étape de *commit* accomplie.

Les exceptions liées aux instructions, comme une division par zéro ou un appel système, sont traitées à cet étage. En effet, dans ce cas, l'exécution du programme est souvent stoppée, temporairement ou non, pour traiter l'événement. Le processeur est donc dans un état où l'instruction produisant l'exception est validée et considérée comme exécutée. Avant le traitement de l'exception, l'état de l'exécution du programme est sauvegardé, notamment la valeur courante de tous les registres architecturaux. Ces valeurs sont récupérées en lisant les registres physiques associés aux registres architecturaux dans la *mapping table* du *commit*. Une fois l'exception traitée, si besoin est, le processeur peut recommencer à traiter le programme à partir de l'état sauvegardé de l'exécution.

1.5 Problèmes liés aux branchements

L'organisation du traitement des instructions sous forme de pipeline permet de paralléliser ce traitement. L'organisation dans le désordre accroît l'exploitation du parallélisme présent entre les instructions. Ce parallélisme provient de l'indépendance en termes de données entre les instructions. Les dépendances dues au contrôle et donc aux instructions de branchement restent par conséquent un problème et réduisent l'efficacité du pipeline. En effet, lorsqu'une instruction normale entre dans le pipeline, l'instruction qui la suit dans le code du programme sera la suivante à y entrer. Pour les instructions de branchement, cette relation n'est plus vraie. Le processeur a besoin d'exécuter le branchement pour savoir quelle sera l'instruction suivante. On dit alors que le branchement est résolu. Cela introduit donc autant de bulles dans le pipeline qu'il y a d'étages entre le *fetch* et l'étage d'exécution. Les branchements étant des instructions en nombre conséquent dans la plupart des programmes (environ 20% des instructions sont des branchements), le pipeline est rarement utilisé au maximum.

Pour éviter cette situation, le processeur a plusieurs solutions. La plus utilisée est la prédiction de branchements. Une autre solution consiste tout simplement à transformer le code du programme pour éliminer les branchements et remplacer les instructions dépendantes d'un branchement en termes de contrôle par des instructions prédiquées. Une instruction prédiquée est une instruction accompagnée d'un prédicat dont la valeur détermine si l'instruction est exé-

cutée ou non. Plus précisément, la valeur du prédicat détermine si l'instruction impacte l'état architectural du processeur.

1.5.1 Prédiction de branchements

Pour savoir quelle instruction doit être traitée après une instruction de branchement, le processeur doit calculer deux informations : la direction du branchement et la cible du branchement. La direction dépend du type de branchement et du test associé à celui-ci. Pour les branchements inconditionnels, il n'y a pas de test et le branchement est toujours pris. Pour les branchements conditionnels, ils peuvent être soit pris soit non pris. Si un branchement conditionnel est non pris, alors l'instruction suivante à traiter est celle qui le suit dans le code du programme. Si un branchement est pris, il faut alors connaître sa cible, qui sera donc la prochaine instruction à être traitée.

La prédiction de branchements est donc composée de deux sous-domaines : la prédiction de la direction et la prédiction de la cible. Elles seront détaillées dans le chapitre 2.

Le fait que l'information prédite ne soit pas exacte est le principal désavantage de cette approche. Dans le cas d'une mauvaise prédiction, l'exécution doit être corrigée. La correction consiste principalement à sauvegarder l'état du processeur au moment où de la prédiction, pour ainsi pouvoir recharger cette sauvegarde et recommencer l'exécution avec des informations correctes, une fois celles-ci connues. Ce mécanisme de correction est largement plus coûteux que la simple insertion de bulles dans le pipeline lorsque la prédiction n'est pas utilisée.

La précision de la prédiction est donc un élément essentiel pour l'intérêt du mécanisme. De nombreux prédicteurs ont été proposés dans la littérature au fil des années. La tendance qui semble se confirmer [54, 50] est que leur précision a atteint un seuil *a priori* difficile à dépasser.

Augmenter la performance de ce mécanisme est aussi possible d'une manière orthogonale à l'amélioration de la précision de la prédiction : la diminution du coût d'une mauvaise prédiction. C'est cette direction que ce document explore dans les chapitres 2 et 3.

1.5.2 Instructions prédiquées

Une instruction prédiquée est une instruction à laquelle un prédicat est associé. La valeur de ce prédicat détermine si l'instruction est exécutée ou non, du point de vue du programmeur. Ce prédicat est une fonction booléenne, souvent simple, dont l'évaluation doit être faite dynamiquement à l'exécution. Ces instructions peuvent être utilisées pour remplacer les branchements et les instructions que ces branchements contrôlent. Cette transformation de code s'appelle une *if-conversion* [4]. Les instructions prédiquées peuvent être utilisées pour de nombreuses autres optimisations, notamment dans le pipeline logiciel [3]. Ces utilisations ne seront cependant pas abordées de manière détaillée dans ce document.

L'utilisation de ces instructions en replacement des branchements permet donc de s'affranchir du problème posé par ceux-ci. L'exécution pipelinée de ces instructions pose cependant d'autres problèmes détaillés dans le chapitre 4 ainsi que les solutions proposées par la littérature. Dans le chapitre 5, nous présentons notre propre solution pour résoudre ces problèmes.

Chapitre 2

Prédiction de branchements et réduction de la pénalité lors d'une mauvaise prédiction

Dans ce chapitre nous survolerons rapidement l'état de l'art de la prédiction de branchements pour nous intéresser plus particulièrement aux différentes propositions qui ont été faites dans la littérature pour réduire le coût de la correction de l'exécution qui est nécessaire lors d'une mauvaise prédiction de branchement.

2.1 Prédiction de branchements

La prédiction de branchements se base sur une observation simple : le comportement d'un programme est globalement cyclique. Les instructions le composant sont donc exécutées de nombreuses fois lors de l'exécution complète du programme. Et chaque nouvelle instance dynamique de la même instruction statique a un comportement qui est souvent le même. En particulier, les instructions de branchement ont souvent un comportement périodique. De plus, la cible de la plupart des branchements est toujours la même. Un prédicteur de branchements analyse donc le comportement dynamique des branchements et exploite le caractère périodique de ce comportement pour fournir sa prédiction. La précision d'un prédicteur mesure sa capacité à produire des prédictions correctes. C'est le rapport du nombre de bonnes prédictions sur le nombre total de prédictions.

2.1.1 Prédiction de la cible

Selon son type, un branchement peut avoir une ou plusieurs cibles.

Les branchements de type direct ont leur cible directement définie dans le code de l'instruction. Elle est donc connue dès que l'instruction est décodée. Néanmoins, elle est quand même prédite pour éviter l'insertion de bulles le temps que le branchement arrive à l'étage de décodage. Cependant, cette prédiction est très simple puisqu'une fois que le branchement a été rencontré

pour la première fois, sa cible peut être enregistrée. Comme elle ne change pas au cours de l'exécution, tant que cet enregistrement existera, la prédiction donnée sera toujours exacte. La structure qui est utilisée pour enregistrer la cible des branchements est le tampon des cibles des branchements ou *Branch Target Buffer* (BTB). Il est généralement organisé sous la forme d'une mémoire associative dont la taille et le niveau d'associativité influent directement sur sa précision. En effet, sa précision dépend uniquement de sa capacité à retenir les cibles qui y ont été enregistrées.

Le BTB est aussi utilisé pour l'autre type de branchements, les branchements indirects. Ceux-ci ont leur cible qui est stockée dans un registre. C'est donc l'exécution du programme qui calcule dynamiquement cette cible. Celle-ci peut donc changer à chaque instance dynamique du branchement. Comme ce n'est pas toujours le cas, le BTB parvient à être relativement efficace sur ce type de branchements. Cependant, il est possible de concevoir un prédicteur dont le rôle est de prédire la cible des branchements indirects. Plusieurs de ces prédicteurs ont été proposés dans la littérature [19, 49].

2.1.2 Cas spécifiques des appels de fonctions et des retours

La cible des instructions de retour de fonctions sont facilement prédictibles et sont donc traitées par un mécanisme spécifique : la pile des adresses de retours ou *Return Address Stack* (RAS). L'idée est qu'à chaque instruction d'appel de fonction correspondra une instruction de retour. Lorsqu'une instruction d'appel est rencontrée, l'adresse de l'instruction suivante est empilée. Lorsqu'une instruction de retour est ensuite rencontrée, l'adresse présente au sommet de la pile est dépilée et utilisée comme prédiction pour la cible de l'instruction de retour. Ce mécanisme est très précis tant que la profondeur d'appel ne dépasse pas la taille maximale de la pile.

2.1.3 Prédiction de la direction

Le domaine de prédiction de la direction a beaucoup plus été exploré par la recherche que celui de la prédiction de la cible. Cela s'explique par la difficulté qu'il y a à concevoir un prédicteur de direction précis et à la relative simplicité du problème posé par la prédiction de la direction, notamment pour les branchements directs. Il existe de nombreux types de prédicteurs de direction. Ce document en liste quelques uns, mais ne prétend pas être exhaustif.

Comme un prédicteur de branchements est essentiellement une structure qui enregistre le comportement dynamique des branchements, un historique de ce comportement est conservé par le prédicteur et ensuite utilisé pour produire la prédiction. Cet historique peut être conservé de différentes manières, plus ou moins exactes. L'historique exact consiste en une suite de bits, 0 pour la direction non pris et 1 pour la direction pris. On constate deux grandes familles de prédicteurs à historique exact. Ceux qui conservent un historique pour chaque branchement, les prédicteurs à historique local, et ceux qui conservent l'historique global, c'est-à-dire la suite des directions de la suite des branchements qui sont traités par le pipeline. Une manière moins exacte consiste à enregistrer le comportement du branchement dans un automate. Celui-ci est généralement implémenté à l'aide d'un compteur à saturation. La figure 2.1 représente l'automate

qui est utilisé. Lorsque que le branchement est pris, l'état de l'automate est déplacé vers la droite. Lorsque le branchement est non pris, l'état de l'automate est déplacé vers la gauche.

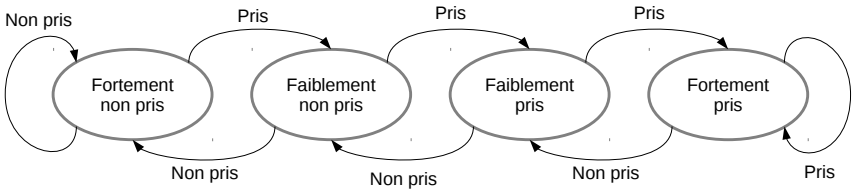


FIGURE ***2.1*** **–** *Diagramme d'état d'un compteur à saturation à 2 bits.*

Prédiction avec compteurs à saturation

Le prédicteur *bimodal* est un prédicteur qui utilise des compteurs à saturation pour enregistrer le comportement des branchements. L'état du compteur sert ensuite pour donner la prédiction. L'intérêt d'avoir plusieurs états donnant la même prédiction est d'éviter une mauvaise prédiction si le comportement du branchement dévie une seule fois de son comportement habituel. Par exemple, un branchement avec un comportement de boucle est habituellement tout le temps pris et il est non pris une seule fois pour ensuite recommencer à être tout le temps pris par la suite. Avec ce mécanisme, seule la fois où il est non pris est mal prédite. Ça ne serait pas le cas si l'automate avait seulement deux états (ce qui correspond à une prédiction qui répète le dernier état enregistré). Avec cette configuration, il serait mal prédit la fois où il est non pris et la fois suivante.

Comme chaque branchement peut avoir un comportement propre, il n'est pas efficace d'utiliser un seul compteur. Un ensemble de compteurs est donc rassemblé dans une table qui est indexé par l'adresse de l'instruction de branchement ou *Program Counter* (PC). Cette organisation permet de tendre vers une situation où un compteur n'est associé qu'à un seul branchement. Si ce n'est pas le cas, il y a alors des phénomènes de partage des entrées par plusieurs branchements, qui peuvent être constructifs (l'information enregistrée dans le compteur est renforcée), mais qui sont le plus souvent destructifs (l'information enregistrée dans le compteur est faussée, menant à mal prédire les branchements associés à cette entrée). Un grand nombre d'entrées est donc nécessaire pour éviter ces phénomènes.

Ce prédicteur est très efficace pour les branchements dont le comportement est toujours le même sur une grande période.

Prédiction avec historique local

La première famille des prédicteurs à historique exact, ceux à historique local, est constituée de prédicteurs qui sont organisés autour d'une table indexée par le PC des branchements et dont chaque entrée stocke l'historique du branchement qui lui est associée. Cet historique peut ensuite être utilisé pour indexer une deuxième table qui contiendra soit directement la prédiction, soit un compteur à saturation à 2 bits et plus dont l'état donnera la prédiction. La figure 2.4 résume cette dernière organisation.

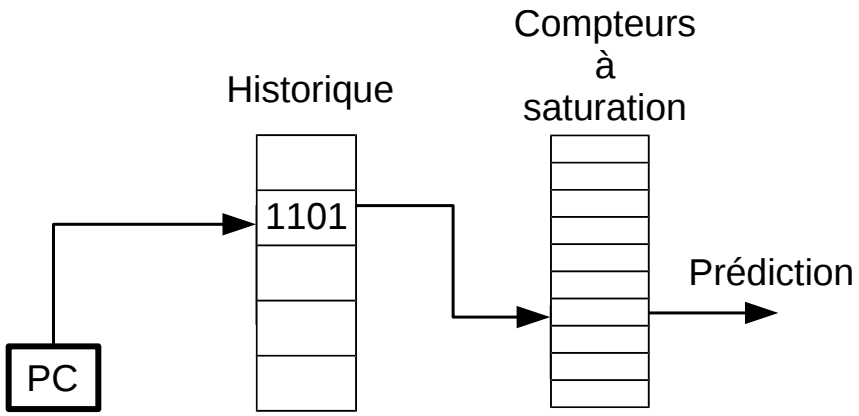


FIGURE *2.2 – Prédicteur local à deux niveaux : la première table, indexée par le PC du branchement, contient l'historique de celui-ci. Cet historique sert à indexer la deuxième table qui contient des compteurs à saturation donnant la prédiction finale.*

Ce type de prédicteurs est généralement plus précis que le prédicteur *bimodal* car il est capable d'enregistrer un comportement périodique pour chaque branchement. Par exemple, si un branchement est pris trois fois de suite pour être ensuite non pris et que cette séquence se répète, le prédicteur *bimodal* prédira tout le temps le branchement comme pris. Un prédicteur à historique local avec un historique suffisamment grand (au moins 4 bits) sera capable de prédire correctement cette séquence.

Prédiction avec historique global

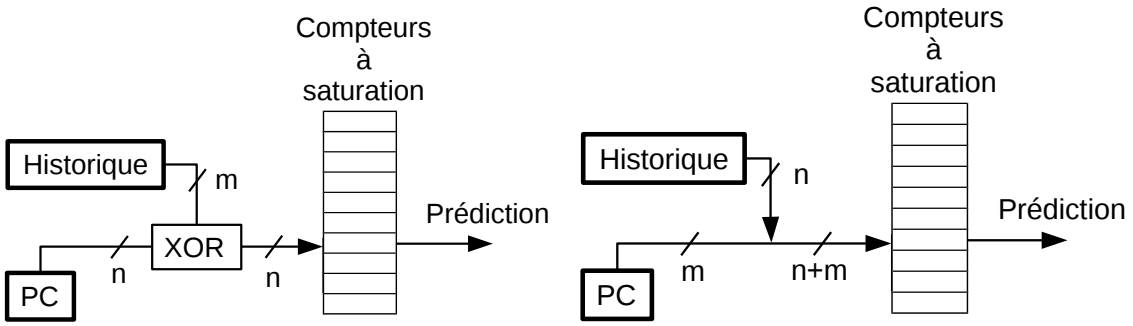
Cette second famille de prédicteurs à historique exact tente de tirer parti de la corrélation qu'il peut exister entre différents branchements. En effet, il arrive souvent que la direction de deux branchements consécutifs soit liés.

Il existe de nombreuses organisations différentes. Le principe commun est l'utilisation d'un vecteur contenant l'historique global des derniers branchements exécutés. Cet historique peut être utilisé pour indexer une table contenant des compteurs 2 bits. Le principal problème provient de l'augmentation exponentielle de cette table en fonction de la taille de l'historique.

Une seconde limitation est la perte d'information pour un même branchement par rapport à l'historique local. Pour pallier ce problème, une partie du PC du branchement est utilisée en plus de l'historique pour indexer la table. Si le PC et l'historique sont mélangés par une fonction XOR, on parle de prédicteur *gshare* (voir la figure 2.3a), si le PC et l'historique sont concaténés, on parle de prédicteur *gselect* (voir la figure 2.3b).

Prédiction hybride

Chaque type de prédicteurs est plus ou moins efficace à prédire différents types de comportement des branchements. L'idée des prédicteurs hybrides est de rassembler plusieurs types de prédicteurs pour avoir ainsi plusieurs sources de prédictions. Un choix est ensuite fait pour déterminer la prédiction finale.



(a) Prédicteur à historique global *gshare* : le PC et l'historique sont mélangés pour indexer une table contenant des compteurs à saturation qui donnent la prédiction finale.

(b) Prédicteur à historique global *gselect* : le PC et l'historique sont concaténés pour indexer une table contenant des compteurs à saturation qui donnent la prédiction finale.

FIGURE *2.3 – Exemple de prédicteurs à historique global : le predicteur* gshare *et le prédicteur* gselect.

La décision peut se faire par un vote à la majorité. Ce schéma nécessite un nombre impair de prédicteurs. La prédiction finale est donc celle qui a été donnée par la majorité des prédicteurs.

Un méta prédicteur peut aussi être utilisé. Celui-ci est chargé d'enregistrer quel prédicteur a été le plus précis dans le passé sur le branchement à prédire. La prédiction finale est ensuite donnée par le prédicteur ainsi choisi. Le méta prédicteur peut être une simple table de compteurs à saturation si le prédicteur hybride est constitué de deux sous-prédicteurs.

2.1.4 Fonctionnement du prédicteur TAGE

Le prédicteur de direction de branchements TAGE [54] peut être considéré comme l'un des prédicteurs les plus précis à l'heure actuelle. Sa structure et son fonctionnement sont présentés ici car ce prédicteur et un dérivé ont été utilisés dans les travaux présentés dans ce document. Le prédicteur TAGE a connu plusieurs améliorations [52, 53, 50]. Nous ne présenterons que l'organisation de base qui est restée sensiblement la même.

Le prédicteur TAGE (pour *TAgged GEometric history length*) est un prédicteur hybride composé d'un prédicteur de base, qui peut être un prédicteur *bimodal* par exemple, et d'une série de tables. Celles-ci sont indexées par un historique global de taille croissante et elles contiennent des entrées (partiellement) étiquetées. L'étiquette est calculée en fonction du PC du branchement et de l'historique global de la taille associée à la table. En plus des étiquettes, ces entrées contiennent un compteur à saturation dont l'état donne la prédiction et un compteur à saturation non signé mesurant l'utilité de l'entrée et qui sert d'indicateur pour la politique de remplacement des entrées.

Les différentes tables (le prédicteur de base et les tables étiquetées) sont appelés les composants du prédicteur. Un composant étiqueté fournit une prédiction si une correspondance entre l'étiquette calculée et une étiquette présente dans le composant est trouvée. Le composant producteur est celui qui a été utilisé pour donner la prédiction finale. La prédiction alternative est la prédiction qui aurait été donnée si la prédiction n'avait pas été trouvée dans le composant

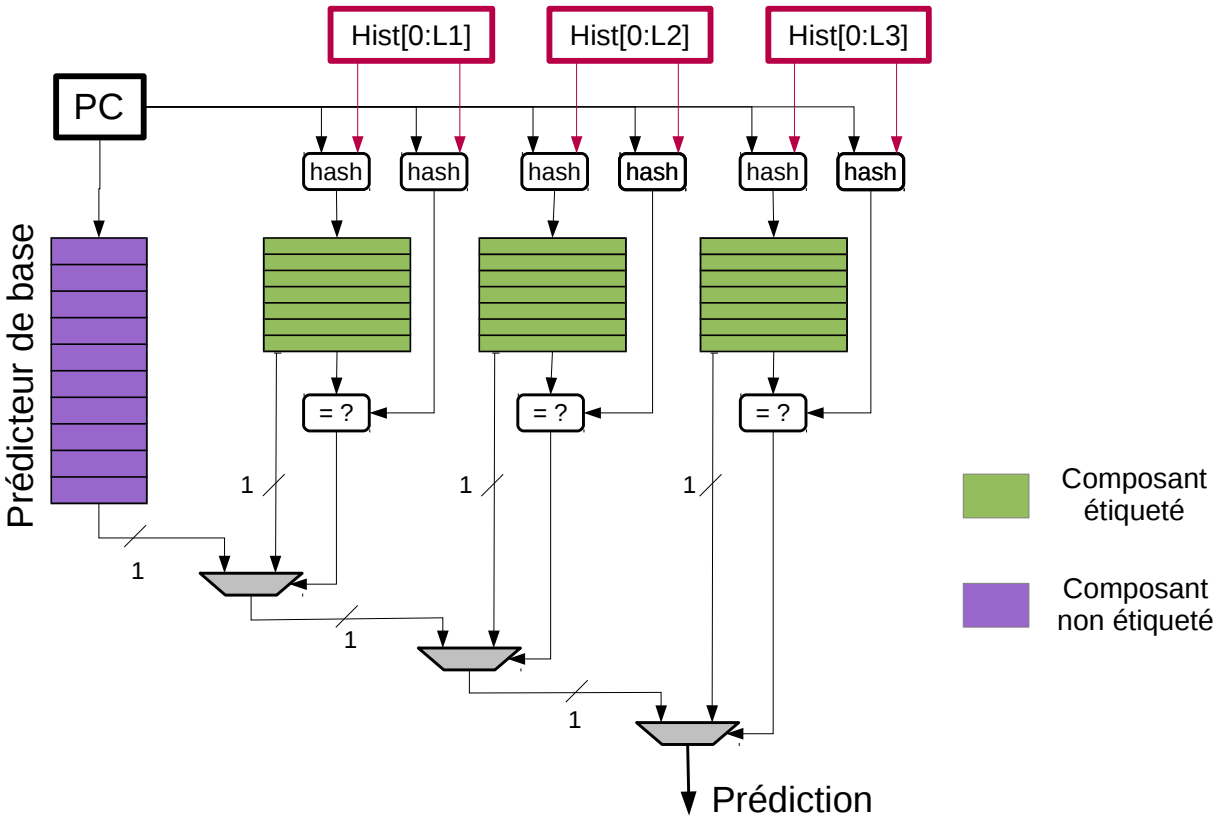


FIGURE *2.4 – Schéma du prédicteur TAGE : un prédicteur de base augmenté par plusieurs composants de prédicteurs étiquetés qui sont indexés par un historique de taille croissant.*

producteur. C'est donc soit une prédiction fournie par un autre composant étiqueté ou par le prédicteur de base.

Calcul de la prédiction

Lors du calcul de la prédiction, le prédicteur de base et les composants étiquetés sont accédés en parallèle. Le prédicteur de base fournit toujours une prédiction. Les composants étiquetés ne fournissent une prédiction que s'il y a correspondance des étiquettes.

La prédiction est donnée par le composant pour lequel il y a eu correspondance. S'il y a correspondance dans plusieurs composants, le composant utilisant le plus grand historique est sélectionné. La prédiction est fournie par la lecture de l'état du compteur à saturation de l'entrée de ce composant. S'il n'y a eu aucune correspondance, c'est la prédiction donnée par le prédicteur de base qui est utilisée.

Il a été observé que lorsque le compteur à saturation du composant étiqueté donnant la prédiction est dans un état faible (c'est-à-dire proche de passer dans un état où la prédiction est inversée), la précision de la prédiction est assez basse (souvent moins de 60%). Cela est particulièrement le cas sur les entrées nouvellement allouées dans les composants étiquetés. Dans cette situation, la prédiction alternative peut être utilisée. Comme cette propriété semble essentiellement être un propriété globale de l'exécution et non de chaque branchement, un simple compteur à saturation à 4 bits est utilisé pour enregistrer ce comportement. L'état de ce compteur est utilisé pour décider si la prédiction alternative est exploitée ou non dans ce cas.

Mise à jour du prédicteur

Une étape primordiale pour avoir une prédiction précise est de bien enregistrer le comportement des différents branchements. Cette étape est réalisée lors de la mise à jour du prédicteur.

Comme dans tout prédicteur avec des compteurs à saturation, il faut mettre à jour le compteur qui a donné la prédiction finale. Le compteur d'utilité de l'entrée qui a donné cette prédiction est incrémenté si la prédiction était correcte et, en même temps, que la prédiction alternative ne l'était pas, prouvant ainsi l'utilité de l'entrée en question.

L'ajout de nouvelles entrées dans les composants étiquetés se fait uniquement lors d'une mauvaise prédiction. Cet ajout se fait dans un composant étiqueté utilisant un historique plus grand que celui qui a fourni la mauvaise prédiction. Si le composant ayant donné la prédiction est déjà celui utilisant le plus grand historique, aucune nouvelle entrée n'est allouée. La nouvelle entrée est choisie parmi les entrées ayant des compteurs d'utilité à zéro. Elle est initialisée avec son compteur à saturation dans l'état faiblement correct et avec son compteur d'utilité à zéro.

Pour éviter un phénomène de famine, c'est-à-dire pour éviter l'absence d'entrées disponibles, les compteurs d'utilité doivent être remis à zéro périodiquement.

Les politiques d'allocation de nouvelles entrées et de gestion des compteurs d'utilité sont les éléments de la politique de mise à jour du prédicteur qui ont connus le plus d'améliorations dans les différentes version du prédicteur TAGE.

2.2 Reconvergence du flot de contrôle et indépendance de contrôle

Dans un processeur avec exécution dans le désordre qui utilise la prédiction de branchements pour résoudre les problèmes posés par ceux-ci, on parle d'exécution spéculative lorsque l'exécution des instructions est dirigée par des informations provenant d'une prédiction. Ces informations doivent être vérifiées une fois leur vrai source disponible et l'exécution éventuellement corrigée.

On parle d'instructions sur le mauvais chemin lorsque celles-ci sont traitées après un branchement mal prédit. Ces instructions ne devraient donc pas être présentes dans le pipeline et celui-ci est vidé de ces instructions lorsque la mauvaise prédiction est détectée et que l'exécution est corrigée. Les instructions sur le bon chemin sont celles qui sont dans la suite d'instructions qui doit normalement être traitée par le processeur.

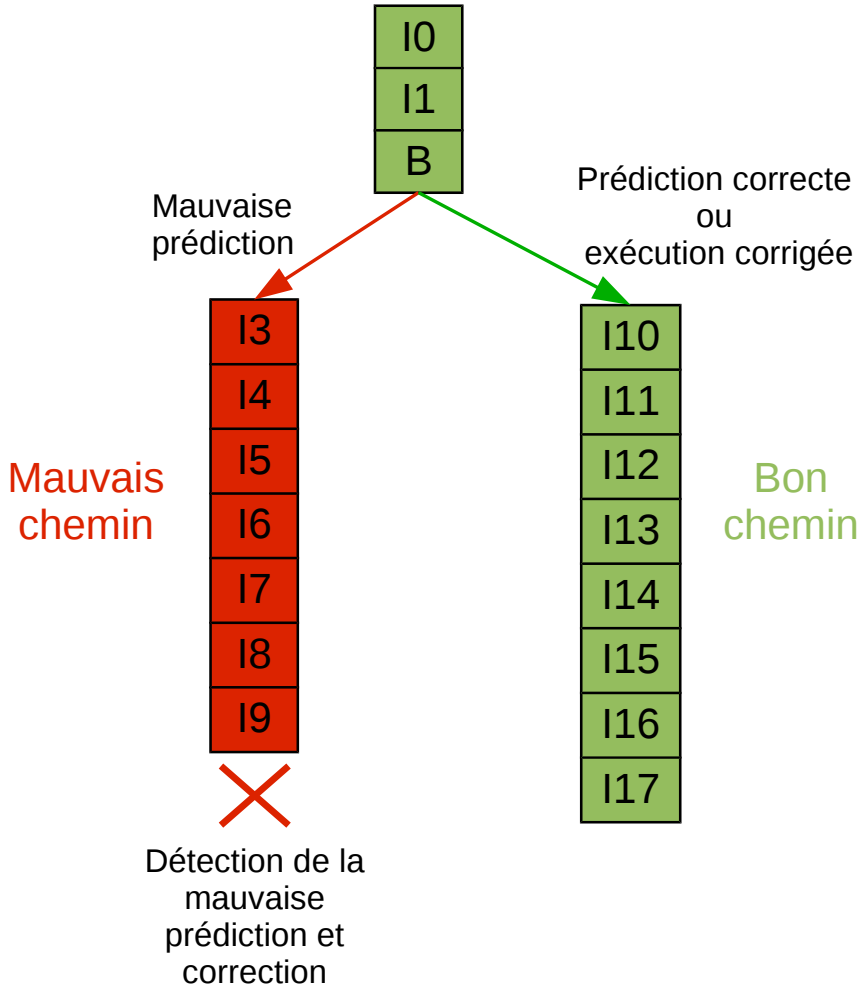


Figure 2.5 – *Mauvais et bon chemins d'exécution : suite à une mauvaise prédiction, le processeur commence à traiter des instructions qu'il ne devrait sémantiquement pas traiter, s'engageant sur un mauvais chemin d'exécution. La correction de l'exécution le remet sur le bon chemin.*

La figure 2.5 illustre ces deux notions. Lorsqu'une mauvaise prédiction est faite, le processeur

commence à traiter des instructions qu'il ne devrait pas traiter sous peine de ne pas respecter la sémantique du programme. Le processeur est donc en train de traiter un mauvais chemin. Comme ce traitement se fait en mode spéculatif, ces instructions ne seront pas validées tant que le branchement ne sera pas exécuté et la prédiction vérifiée. Lors de cette vérification, la mauvaise prédiction est détectée, ce qui déclenche le mécanisme de correction de l'exécution. Celui-ci vide le pipeline des instructions sur le mauvais chemin, et le traitement peut recommencer à partir de la bonne instruction, déterminée par l'exécution du branchement. Le processeur est alors de retour sur le bon chemin.

2.2.1 Reconvergence du flot de contrôle

Les branchements conditionnels définissent statiquement deux chemins d'exécution possibles. Dans la plupart des cas, ces deux chemins se rejoignent en un point que l'on appelle le point de reconvergence. Ce phénomène est appelé reconvergence du flot de contrôle. La figure 2.6 donne un exemple de ce phénomène lorsque le branchement est issu de l'instruction de test *if*. Dans ce cas, les deux chemins sont celui qui est exécuté si le test est vrai (chemin *then*) et celui qui est exécuté si le test est faux (chemin *else*). Le point de reconvergence est donc la première instruction qui suit le corps du *if-then-else* dans le code du programme.

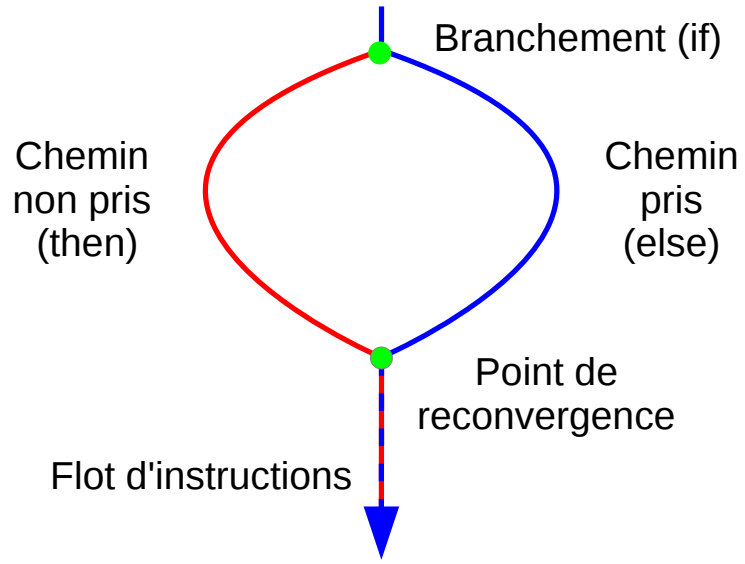


Figure *2.6* – *Exemple de reconvergence de flot de contrôle dans le cas d'une instruction de branchement* if.

À l'exécution, seul l'un des deux chemins est traité, puisque le branchement est soit pris soit non pris. Cependant, dans le cas d'une mauvaise prédiction de la direction, le chemin qui ne devrait pas être exécuté commence à être traité par le processeur. Le temps que la mauvaise prédiction soit détectée, une bonne partie de ce chemin peut avoir été traitée, allant même jusqu'à dépasser le point de reconvergence. Dans certains cas, le processeur peut donc commencer à traiter, sur le mauvais chemin, des instructions qu'il retrouvera sur le bon chemin, une fois

l'exécution corrigée.

2.2.2 Indépendance de contrôle

L'indépendance de contrôle est la propriété de l'ensemble des instructions qui sont indépendantes d'un branchement en particulier, c'est-à-dire les instructions qui sont exécutées quelque soit le résultat du branchement, que celui-ci soit pris ou non pris.

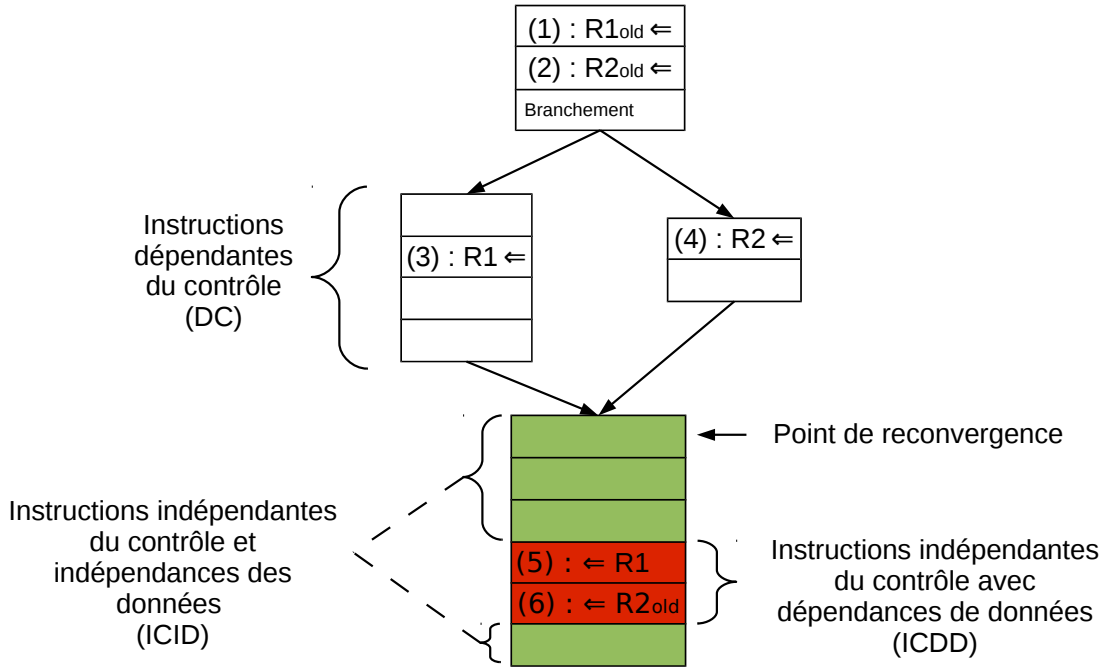


Figure *2.7 – Illustration des instructions dépendantes et indépendantes du contrôle. Parmi les instructions indépendantes du contrôle, on distingue celles qui sont dépendantes des données et celles qui sont aussi indépendantes des données.*

Les instructions qui sont dépendantes du branchement, qui ne sont donc exécutées que si le branchement dirige l'exécution vers ces instructions, sont dites dépendantes du contrôle (DC). Les instructions qui sont exécutées quelque soit le résultat du branchement sont dites indépendantes du contrôle (IC). Le point de reconvergence est la première des instructions IC. On considère deux catégories d'instructions IC : celles qui ont des dépendances de données avec les instructions DC et celles qui n'en ont pas. Les instructions de la première catégorie sont dites indépendantes du contrôle mais dépendantes des données (ICDD) et celles de la deuxième catégorie sont dites indépendantes du contrôle et indépendantes des données (ICID) [2].

Dans la figure 2.7, des exemples d'instructions DC, IC, ICDD et ICID sont donnés. En particulier, l'instruction (5) est ICDD car quand elle a été exécutée sur le mauvais chemin, elle a utilisé la valeur du registre $R1$ en tant que source, valeur qui a été produite par l'instruction (3) sur le mauvais chemin. L'exécution de l'instruction (5) a utilisé une mauvaise valeur, elle est donc fausse. La bonne valeur à utiliser est celle contenue dans le registre $R1_{old}$, produite par l'instruction (1). On parle dans ce cas de fausse dépendance de données car la dépendance se

fait avec une donnée produite sur le mauvais chemin, et donc avec une donnée qui ne devrait pas exister. De même, l'instruction (6) utilise la valeur contenue dans $R2_{old}$. Or cette valeur est redéfinie par l'instruction (4) sur le bon chemin. L'exécution de l'instruction (6) a elle aussi utilisé une mauvaise valeur et est donc également fausse. La bonne valeur à utiliser est celle produite par l'instruction (4). On parle dans ce cas de vrai dépendance de données car la dépendance se fait avec une donnée qui existe légitimement.

2.2.3 Exploitation de la reconvergence du flot de contrôle et de l'indépendance de contrôle

Il est possible de réduire le coup d'une mauvaise prédiction grâce à la reconvergence du flot de contrôle et à l'indépendance de contrôle. La reconvergence associée à l'exécution spéculative sur le mauvais chemin permet dans certains cas de commencer à traiter les instructions IC sur le mauvais chemin. Lors de la correction suite à la détection de la mauvaise prédiction, le pipeline est vidé de l'ensemble des instructions qui ont commencé à être traitées par le processeur après le branchement mal prédit. Cependant, comme les instructions IC seront aussi traitées sur le bon chemin, une partie du travail fait par le processeur sur le mauvais chemin peut être sauvé pour éviter de le refaire sur le bon chemin. En particulier, le résultat produit par les instructions ICID est le même que celui qui sera produit sur le bon chemin, celles-ci n'ont donc pas besoin d'être réexécutées et le résultat calculé sur le mauvais chemin peut directement être utilisé sur le bon chemin. En effet, comme ces instructions n'utilisent que des opérandes qui sont indépendantes du chemin pris par le branchement, leur résultat est lui aussi indépendant et reste le même quelque soit la direction que prend le branchement.

2.2.4 Difficultés liées à l'exploitation de l'indépendance de contrôle

Un mécanisme exploitant la reconvergence du flot de contrôle et l'indépendance de contrôle cherche donc avant tout à déterminer quelles sont les instructions IC et parmi celles-ci quelles sont les instructions ICID. La deuxième étape est de sauver le travail fait par le processeur sur ces instructions lors de l'exécution sur le mauvais chemin pour le réutiliser sur le bon chemin sans avoir à le refaire. Cela permet de mitiger la perte engendrée par le traitement des instructions, qui est généralement jugé inutile, sur le mauvais chemin en réutilisant une partie de ce travail sur le bon chemin.

La première difficulté est donc de pouvoir discerner quelles sont les instructions DC et quelles sont les instructions IC, c'est-à-dire détecter le point de reconvergence. Des techniques logicielles ont été proposées, s'appuyant sur le compilateur ou utilisant du profilage, pour détecter ce point de reconvergence [45]. Cependant celles-ci nécessitent d'étendre le jeu d'instructions et sont ainsi difficilement applicables sur les programmes binaires actuels.

Une fois que le point de reconvergence est identifié, la seconde difficulté majeure est de conserver le travail fait par le processeur sur les instructions IC, notamment le résultat de ces instructions. En effet, dans un processeur superscalaire à exécution dans le désordre, le résultat, mais aussi la chaîne de dépendances des instructions en cours de traitement sont stockés dans les différentes ressources du pipeline : les registres physiques, le ROB et la LSQ. Les entrées de

ces structures sont allouées dynamiquement par les premiers étages de traitement du pipeline dans l'ordre dans lequel les instructions entre dans celui-ci. Lors de la détection d'une mauvaise prédiction et de sa correction, ces entrées sont simplement désallouées et remises dans la liste des entrées libres de chaque structure. Ainsi, dans la plupart des cas, l'allocation dynamique des entrées dans ces structures est complètement différente sur le chemin pris et sur le chemin non pris. En particulier, il est peu probable que chaque instruction IC occupent la même entrée dans chaque structure sur les deux chemins. Afin d'exploiter l'indépendance de contrôle après un branchement mal prédit, il est donc nécessaire d'avoir un mécanisme matériel qui permette de sauvegarder le contenu des registres physiques ainsi que celui des entrées de la LSQ et du ROB sur le mauvais chemin. Il est aussi nécessaire d'avoir une méthode simple pour récupérer ces données lors de l'exécution sur le bon chemin.

Toutefois, sauvegarder les résultats n'est pas suffisant. Il faut aussi trier ces résultats, car tous ne sont pas utiles. En effet, seul les résultats produits par des instructions ICID peuvent être réutilisés. Il faut donc pouvoir différencier les instructions ICID des instructions ICDD. Une fois cette identification faite, le résultat des instructions ICID peut être directement réutilisé. Quant aux instructions ICDD, elles doivent être réexécutées pour obtenir un résultat valide.

2.3 État de l'art des travaux visant à réduire la pénalité due à une mauvaise prédiction

La pénalité due à une mauvaise prédiction se décompose en deux facteurs : le coût de la correction (vidage du pipeline, restauration de l'état du processeur) et l'utilisation de ressources pour du travail inutile. La plupart des travaux que nous présentons dans cette partie s'intéressent au deuxième facteur pour essayer de réduire le coût d'une mauvaise prédiction.

Exploiter l'indépendance de contrôle est une des possibilités. Ainsi, de nombreuses propositions de mécanismes matériels ont été faites pour tirer profit de son potentiel.

2.3.1 Étude de l'indépendance de contrôle

Ce potentiel a été étudié en détail [45]. De cette étude ressort le fait que le principal coût des mauvaises prédictions provient "des ressources qui sont gaspillées par les instructions DC incorrectes". Il en ressort aussi que l'exploitation de l'indépendance de contrôle peut apporter des gains qui peuvent aller jusqu'à la moitié des gains obtenus si aucune mauvaise prédiction n'était faite.

Ce travail présente aussi une implémentation matérielle d'un mécanisme exploitant l'indépendance de contrôle. La difficulté de la détection du point de reconvergence est résolue par l'utilisation d'une analyse logicielle qui ajoute l'information directement dans le codage des instructions. Pour ce qui est de la différenciation entre les instructions ICID et ICDD, l'algorithme est le suivant : les premières étapes de l'exécution de l'instruction sont rejouées. Si il y a une différence dans les opérandes sources de l'instruction par rapport au mauvais chemin, alors l'instruction est identifiée comme ICDD et elle est réexécutée. Les instructions qui ne sont pas réexécutées sont elles identifiées comme ICID. Pour les instructions mémoires, tout changement dans l'ordre

d'exécution est détecté. Cela permet de sélectionner les *loads* qui ont besoin d'être réexécutés (pour rappel, les *stores* sont seulement exécutés lors du commit et donc jamais spéculativement). L'ensemble du mécanisme de rejeu est dérivé de celui des processeurs à traces et est décrit en détail dans un autre travail [44].

En moyenne, les résultats de 20% des instructions IC peuvent être sauvés et réutilisés. Cela résulte en un gain de performance allant de 10% à 30% (16% en moyenne) comparé à une architecture qui n'exploite pas l'indépendance de contrôle.

2.3.2 *Selective Branch Recovery*

Une technique appelée *Selective Branch Recovery* (SBR) [20] a été proposée pour exploiter partiellement l'indépendance de contrôle. Plus précisément, en vue de réduire la complexité matérielle, ce mécanisme ne cible que l'ensemble des branchements prédit non pris et donc le chemin pris est vide.

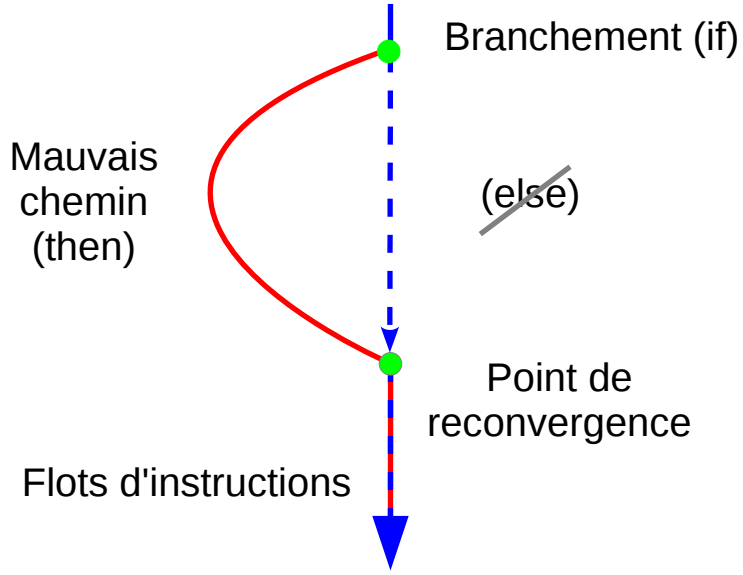


FIGURE *2.8 – Branchements avec reconvergence exacte : il n'y a pas d'instructions DC sur le bon chemin, le point de reconvergence est donc directement la cible du branchement.*

La figure 2.8 donne un exemple de ce type de branchements pour lequel on parle de reconvergence exacte. Le cas présenté est un branchement conditionnel *if* prédit non pris (chemin *then*) et qui ne possède pas de partie *else*. Les avantages de se concentrer sur de tels cas sont multiples. Le point de reconvergence est facilement identifiable puisque c'est la cible du branchement. De plus, comme il n'y a pas d'instruction DC sur le bon chemin, il n'y a pas de vrais dépendances de données, uniquement des fausses dépendances de données. L'identification des instructions ICID est donc facilitée.

L'étude présentant ce mécanisme précise que 32% des branchements mal prédits dans le jeu de tests considéré ont une reconvergence exacte. De plus, si une technique d'inversion de la prédiction est utilisée sur certains branchements, augmentant artificiellement le nombre de

reconvergence exacte, ce taux peut monter théoriquement à 61%. En pratique, un taux de 39% est observé, cette différence étant principalement due aux cas d'inversions incorrectes.

La détection du point de reconvergence se fait à l'aide d'un tampon appelé *Alternate Target Buffer* (ATB). Ce tampon stocke le PC de la cible alternative. Celle-ci est l'instruction qui suivrait le branchement si la direction de la prédiction était inversée. Si le branchement est prédit comme étant pris, cette cible alternative est l'instruction suivant le branchement dans le code du programme. Si le branchement est prédit comme étant non pris, cette cible alternative est la cible du branchement. Ensuite, pour chaque instruction qui entre dans le pipeline, on cherche si son PC est dans l'ATB. Si c'est le cas, le branchement associé peut donc être classé dans la catégorie des reconvergences exactes potentielles et son point de reconvergence est l'instruction stockée dans l'ATB.

Ce mécanisme pour exploiter l'indépendance de contrôle a été pensé pour demander le moins de modifications possibles de l'organisation superscalaire avec exécution dans le désordre de base. Pour corriger l'exécution, il suffit de réexécuter les instructions IC, puisqu'il n'y a pas d'instructions DC sur le bon chemin. Théoriquement, il faut donc changer le renommage de ces instructions pour que leurs opérandes sources soient les bonnes. Pour SBR, ce ne sont pas les opérandes sources qui changent, mais les valeurs de ces opérandes. Pour cela, les instructions DC du mauvais chemin sont transformées en instructions *move* qui vont copier les bonnes valeurs dans les bons registres.

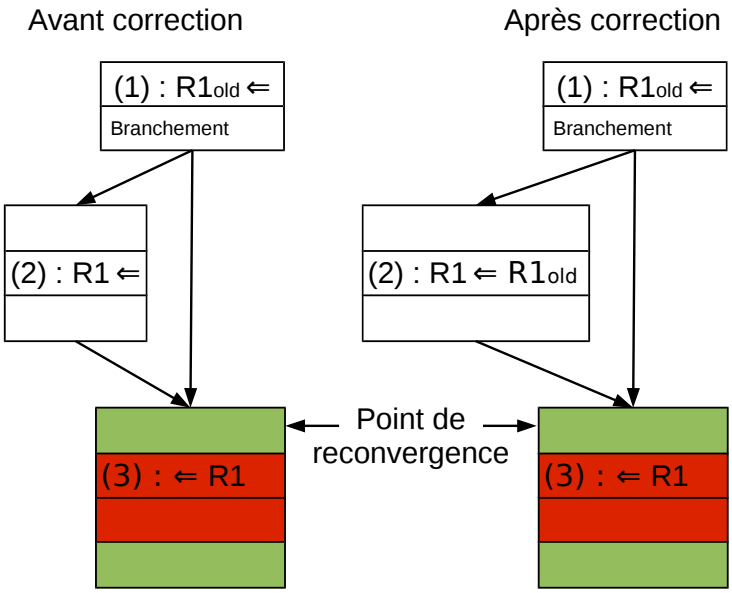


FIGURE ***2.9*** *– Mécanisme de correction de l'exécution pour SBR : les instructions DC sur le mauvais chemin sont transformés en instructions* move *pour déplacer les bonnes valeurs vers les bons registres.*

La figure 2.9 illustre ce mécanisme : sur le mauvais chemin, l'instruction (3) a utilisé le registre $R1$ comme source. Cependant, sur le bon chemin, c'est la valeur contenue dans le registre $R1_{old}$ que l'instruction (3) doit utiliser. Une solution est de renommer une nouvelle fois l'instruction

(3) pour changer le nom de son opérande source. La solution utilisée par le mécanisme SBR est de transformer l'instruction (2) en instruction *move* qui va transmettre la valeur contenue dans le registre $R1_{old}$ dans le registre $R1$. Ainsi, comme le registre $R1$ contient maintenant la valeur correcte nécessaire à l'instruction (3), celle-ci n'a plus besoin d'être renommée une nouvelle fois avant d'être réexécutée.

Le défaut de ce mécanisme est qu'il oblige à réexécuter plus d'instructions. En effet, en plus des instructions ICDD, il faut réexécuter toutes les instructions DC, transformées en *move*, qui étaient sur le mauvais chemin.

Les simulations conduites dans cette étude [20] montrent un gain de 8% en moyenne comparé à une configuration de base. Ce gain peut monter à 12% en considérant une implémentation idéalisée de la méthode pour forcer les cas de reconvergence exacte.

2.3.3 *Transparent Control Independence*

Al-Zawawi, Reddy, Rotenberg et Akkary ont proposé une technique qu'ils appellent *Transparent Control Independence* (TCI) [2]. Celle-ci est toutefois assez complexe à implémenter en matériel. L'idée centrale de ce mécanisme est de séparer les instructions ICID des instructions DC et ICDD pendant l'exécution des instructions DC sur le mauvais chemin. TCI construit ainsi un programme de récupération auto-suffisant à exécuter lorsque le branchement est mal prédit au lieu du mécanisme standard de correction. Pour construire cette suite d'instructions, une structure de tampon FIFO appelée *re-execution buffer* (RXB) est utilisée. Les instructions ICDD y sont stockées, avec une copie de leurs valeurs sources si celles-ci sont fournies par des instructions ICID. Lorsque le processeur détecte la mauvaise prédiction, le programme de récupération, constitué des instructions DC du bon chemin et des instructions contenues dans le RXB, est exécuté. Ainsi, la correction de la mauvaise prédiction se fait de manière transparente pour le processeur, puisqu'il a juste besoin de continuer à exécuter des instructions sans avoir à en supprimer d'autres.

Contrairement à SBR, TCI est capable de traiter tout type de reconvergences de flot de contrôle. Pour la détection du point de reconvergence, TCI utilise le prédicteur proposé par Collins et al. [16]. Plusieurs modifications ont été faites sur celui-ci pour pouvoir rassembler au cours de l'exécution les informations nécessaires au mécanisme. Par exemple, l'ensemble des registres influencés est rassemblé pour chaque branchement. Cet ensemble contient les registres qui seront utilisés en tant que registre destination par les instructions DC. Il sert principalement à détecter les instructions ICDD.

TCI est capable de traiter plusieurs branchements à la fois. Ainsi, le RXB contient toutes les instructions ICDD des différents branchements non encore résolus. Comme les instructions constituant le programme de récupération sont traitées comme n'importe quelles autres instructions, certaines peuvent être ICDD par rapport à d'autres branchements. Ainsi, le même travail de séparation que sur les instructions normales doit être fait. En conséquence, les instructions ICDD par rapport au branchement courant qui sont aussi ICDD par rapport à un branchement précédent non encore résolu doivent être maintenues dans le RXB.

Le principal désavantage de ce mécanisme qui est capable d'exploiter pleinement l'indépen-

dance de contrôle est sa grande complexité. En effet, il requiert beaucoup plus de ressources matérielles que les autres solutions proposées.

Les gains de performance reportés sont de 16% en moyenne pour une configuration capable de traiter 4 instructions par cycle et de 20% en moyenne pour une configuration capable de traiter 8 instructions par cycle. Cette différence provient du nombre plus important d'instructions traitées en parallèle dans le pipeline, augmentant la quantité effective de travail qui peut être sauvé par le mécanisme.

2.3.4 *Skipper*

Cher et Vijaykumar [12] ont proposé un approche relativement différente des autres pour exploiter l'indépendance de contrôle. Ils sont partis de la principale conclusion de l'étude présentée dans la partie 2.3.1 comme point de départ de leur travail. Ils ont tenté de construire une architecture permettant d'éviter le gaspillage des ressources dû à l'exécution des instructions DC sur le mauvais chemin. Ainsi, ils ont développé une technique appelée *Skipper* qui permet de directement traiter les instructions IC sans passer par le traitement des instructions DC, sur le bon chemin ou non. Ces instructions sont seulement traitées lorsque le branchement est résolu et que l'on connaît donc le chemin à prendre.

Cependant un tel mécanisme implique d'assez profondes modifications dans les algorithmes de traitement du processeur. En effet, sauter au delà des instructions signifie que les instructions entrent dans le pipeline dans le désordre. Pour éviter de tels changements, *Skipper* créé un vide dans la fenêtre d'instructions, suffisamment large pour y mettre les instructions DC une fois le branchement résolu. De plus, comme cela implique que des ressources d'exécution essentielles soit réservées, ce mécanisme est seulement utilisé pour les branchements difficile à prédire, ce qui permet de limiter la quantité de ressources utilisées.

La première étape de ce mécanisme est donc d'identifier les branchements qui sont difficiles à prédire. Pour cela, *Skipper* fait appel au prédicteur de confiance en la prédiction JRS, qui a été précédemment proposé par Jacobsen, Rotenberg et Smith [26], pour prendre la décision de sauter le branchement ou non. Le prédicteur de confiance JRS est principalement composé de compteurs à saturation qui sont incrémentés quand la prédiction est bonne et remis à zéro sur une mauvaise prédiction. Pour pouvoir sauter le branchement, *Skipper* utilise une heuristique pour déterminer le point de reconvergence. Cette information est stockée dans une table pour pouvoir la réutiliser la prochaine fois que le branchement doit être traité par le mécanisme. Cette heuristique tente d'identifier les branchements correspondant à des constructions de plus haut niveau : les *if-then-else*, les *if-then* et les boucles. Lorsque l'heuristique ne trouve rien, le branchement n'est pas sauté.

La taille du vide que doit laisser *Skipper* dans la fenêtre d'instructions est estimé d'après les exécutions précédentes du branchement. Une fois ce vide créé, *Skipper* fait entrer les instructions IC dans le pipeline et commence à traiter seulement les instructions ICID. Les registres destinations des instructions DC sont préalloués et marqués comme non prêt, ce qui empêche l'exécution des instructions ICDD d'être lancées. Ainsi, les instructions ICDD doivent attendre que leurs registres sources soient prêts et donc que les instructions DC soient exécutées.

Comme tout le travail fait par *Skipper* l'est de manière spéculative et se base sur les informations obtenues lors des exécutions dynamiques précédentes de chaque branchement, celles-ci peuvent être fausses. Ainsi, il est possible que *Skipper* doive corriger le mauvais travail qu'il a fait. Pour éviter le problème posé par des instructions DC qui seraient sans registre destination préalloué, *Skipper* utilise des nouveaux registres physiques et ajoute le même genre d'instructions *move* que SBR. Si une instruction qui se révèle être ICDD a été exécutée en tant qu'instruction ICID, le cas étant détecté lors du renommage, *Skipper* vide simplement le pipeline des instructions IC et redémarre l'exécution après les instructions DC.

L'étude rapporte que *Skipper* a des performances 10% supérieures à celle d'un processeur superscalaire à exécution dans le désordre de base. Les mesures indiquent qu'environ 5% des branchements mal prédit ne sont pas traités par *Skipper*, soit parce que le branchement n'a pas été identifié par le prédicteur JRS, soit parce que *Skipper* n'a pas réussi à les traiter correctement. Les performances pourraient donc être améliorées avec une meilleur implémentation de cette idée de sauter au delà des instructions DC.

2.3.5 *Ginger*

Ginger est un mécanisme proposé par Hilton et Roth [24] qui est construit sur la même idée que *Skipper*. *Ginger* insère lui aussi un vide dans les structures du processeur pour laisser de la place pour les instructions DC du bon chemin, si jamais le branchement a besoin d'être corrigé.

Plutôt que de réinsérer dans le pipeline et de renommer une nouvelle fois les instructions IC après avoir fait entrer les instructions DC du bon chemin dans le pipeline, *Ginger* effectue une opération de «recherche et remplacement» sur les étiquettes des registres en remplaçant les informations de correspondance calculées sur le mauvais chemin par celles calculées sur le bon chemin. Ainsi, le renommage de toutes les instructions IC en train d'être traitées par le processeur est changé, les différentes structures du pipeline sont alors mises à jour avec les bonnes informations concernant les dépendances de données.

Ginger nécessite de bloquer temporairement le pipeline pour effectuer l'opération de recherche et de remplacement. Lorsque l'exécution reprend, toutes les instructions avec un renommage qui a changé sont réexécutées puisqu'elles sont identifiées comme étant ICDD.

Les gains de performance mesurés sont de 5% sur le jeu de tests *SPECint2000* [58], 11% sur le jeu de tests *MediaBench* [33] et 12% sur le jeu de tests *CommBench* [65]. De plus, des gains dépassant les 15% sont observés sur 11 des 46 programmes de test.

2.3.6 *Register Integration*

Le principe d'intéregation des registres ou *register integration* (RI) a été proposé par Roth et Sohi [46]. Il a été développé pour pouvoir réutiliser le résultat produit par des instructions supprimées du pipeline suite à une mauvaise spéculation. En particulier, il est possible de l'utiliser pour récupérer les résultats des instructions IC lors d'une mauvaise prédiction de branchement. Cette récupération est inspirée d'une autre technique appelée *Dynamic Instruction Reuse* [57], plus généraliste.

Lors de la correction d'une mauvaise prédiction de branchement, la *mapping table* est corrigée, mais les résultats contenus dans les registres physiques ne sont pas supprimés. Pour s'assurer de cela, ces registres physiques ne sont pas remis dans la *free list* lors de la correction. Les informations nécessaires à propos des instructions sur le mauvais chemin sont conservées dans une table, l'*Integration Table* (IT). Celle-ci contient le PC de l'instruction, le numéro de chacun de ses registres physiques sources et le numéro de son registre physique destination, ainsi que des champs spécifiques aux instructions de branchement et aux *loads*.

Lors du redémarrage de l'exécution sur le bon chemin, au moment du renommage, le PC de chaque instruction est cherché dans l'IT. Si une correspondance est trouvée, le mécanisme vérifie si les numéro de registres physiques sources sont les mêmes. Si c'est le cas, cela signifie que le renommage de l'instruction n'a pas changé et que son résultat calculé sur le mauvais chemin peut directement être réutilisé. L'ancien registre physique destination contenant ce résultat est donc réattribué à l'instruction. Celle-ci n'a donc pas besoin d'être réexécutée. On dit alors que l'instruction a été intégrée.

Un traitement spécifique a besoin d'être fait pour les *loads* pour éviter les violations de dépendances de données avec des *stores* sur le bon chemin. Pour cela, l'adresse et le résultat du *load* calculés sur le mauvais chemin sont enregistrés dans l'IT. Si le *load* est intégré, ces informations sont rechargées dans la LSQ, ce qui assure que le mécanisme standard de vérification de violation des dépendances de données entre les *loads* et les *stores* puisse fonctionner de manière correcte.

L'ensemble du mécanisme a été étudié par simulation. Les gains de performance reportés sont de 8% en moyenne par rapport à une architecture superscalaire dans le désordre peu puissante (pipeline peu profond, fréquence d'horloge peu élevée) et de 11.5% par rapport à une architecture plus puissante (pipeline plus profond et fréquence d'horloge plus élevée). L'étude rapporte aussi que ce mécanisme permet, en plus de la réduction du nombre d'instructions exécutées, de réduire le nombre d'instructions traitées sur le mauvais chemin en accélérant la résolution des branchements qui sont intégrés.

2.3.7 *Recycling Waste*

Akkary, Srinivasan et Lai ont proposé une technique qui exploite partiellement la reconvergence du flot de contrôle et l'indépendance de contrôle [1]. Ils sont partis de l'observation que certains branchements résolus sur le mauvais chemin et situés après le point de reconvergence avaient la même résolution sur le bon chemin. Ils ont donc proposé un mécanisme capable de recycler le résultat des branchements sur le mauvais chemin pour aider à la prédiction des branchements sur le bon chemin.

Pour cela, deux structures sont utilisées : le *branch recycling predictor* (BRP) et le *branch recycling cache* (BRC). Le BRC sert à enregistrer le résultat des branchements sur le mauvais chemin de manière séquentielle. Le BRP est utilisé pour décider si les informations contenues dans le BRC vont être utilisées pour un branchement en particulier.

Ainsi, sur le bon chemin, après le redémarrage de l'exécution suite à la correction d'un branchement mal prédit, le BRP est accédé pour chaque branchement. Si le BRP prédit que

l'information contenue dans le BRC sera utile, celle-ci est cherchée dans le BRC. Si une correspondance est trouvée, le résultat du branchement sur le mauvais chemin est utilisé comme source de la prédiction et vient outrepasser celle donnée par le prédicteur de branchements. Pour éviter que l'entrée de la BRC correspondante serve pour plusieurs branchements, celle-ci est invalidée une fois utilisée et la recherche suivante ne se fera qu'à partir de l'entrée qui la suit.

Selon la profondeur du pipeline (et donc le coût d'une mauvaise prédiction de branchement), les gains de performance rapportés sont de 5% à 20%. Les profondeurs considérées sont 20, 40, 60 et 80 cycles (un étage pouvant prendre plusieurs cycles), le mécanisme étant plus performant sur les pipeline les plus profond puisque les gains de performance provient de la réduction du nombre de mauvaises prédictions.

Cette technique n'exploite donc ni complètement ni de manière directe l'indépendance de contrôle. Mais si des branchements ont la même résolution sur le bon chemin et sur le mauvais chemin, c'est essentiellement parce qu'ils sont IC par rapport au branchement mal prédit courant.

Chapitre 3

SYRANT : allocation symétrique des ressources sur les chemins pris et non pris

Ce chapitre présente les travaux réalisés dans le cadre de cette thèse concernant l'exploitation de la reconvergence de flot de contrôle et l'indépendance de contrôle. Le principe générique du mécanisme, consistant à forcer la même allocation de ressources sur les chemins pris et non pris, est d'abord présenté. Puis les différents éléments de ce mécanisme sont détaillées. Nous revenons ensuite sur une contribution annexe issue de notre technique de détection du point de reconvergence. Certaines limitations de notre mécanisme sont ensuite discutées. Les performances ont été évaluées à l'aide d'un simulateur et les résultats de cette étude sont présentés.

3.1 Principe de l'allocation forcée des ressources de SYRANT

Dans le chapitre précédent, nous avons pointé le fait que lors d'une mauvaise prédiction, pour une instruction indépendante du contrôle (IC) donnée, deux ensembles d'entrées différentes sont alloués dans le ROB, la LSQ et le fichier de registres. Pour exploiter l'indépendance de contrôle, les différentes informations concernant l'instruction (dépendances, valeurs des registres, etc.) doivent être préservées (copiées par exemple) suite à la détection d'une mauvaise prédiction pour ensuite être récupérées sur le bon chemin. Cela peut conduire à une organisation complexe et un nombre important de copies.

La proposition détaillée dans ce document, SYRANT, pour *SYmmetric Resource Allocation on Not-taken and Taken paths*, contourne cette difficulté en forçant l'allocation des mêmes entrées dans les principales structures du pipeline à exécution dans le désordre sur les chemins pris et non pris. Ainsi, sur les chemins pris et non pris, une instruction IC particulière aura le même numéro de registre physique destination (et donc le même registre physique), la même entrée dans le ROB et la même entrée dans la LSQ si c'est une instruction mémoire.

Pour forcer cette allocation symétrique, SYRANT insère des vides dans ces structures pour s'assurer que les deux chemins utilisent le même nombre de registres physiques, le même nombre

d'entrées dans le ROB et le même nombre d'entrées dans la LSQ. Comme l'allocation des ressources est séquentielle, ces vides sont simplement des ressources qui ne sont allouées à personne mais qui sont quand même retirées de la liste des ressources utilisables. Ainsi, au point de reconvergence, le pipeline a utilisé exactement le même nombre de ressources sur les deux chemins. Après le point de reconvergence, une instruction IC qui a déjà été renommée sur le mauvais chemin aura sur le bon chemin le même registre physique, la même entrée dans le ROB et la même entrée dans la LSQ si c'est une instruction mémoire. Ainsi, les informations (dépendances, valeurs des registres, etc.) associées à l'instruction IC sont directement disponibles sur le bon chemin, puisque déjà présentes dans les entrées redonnées à l'instruction.

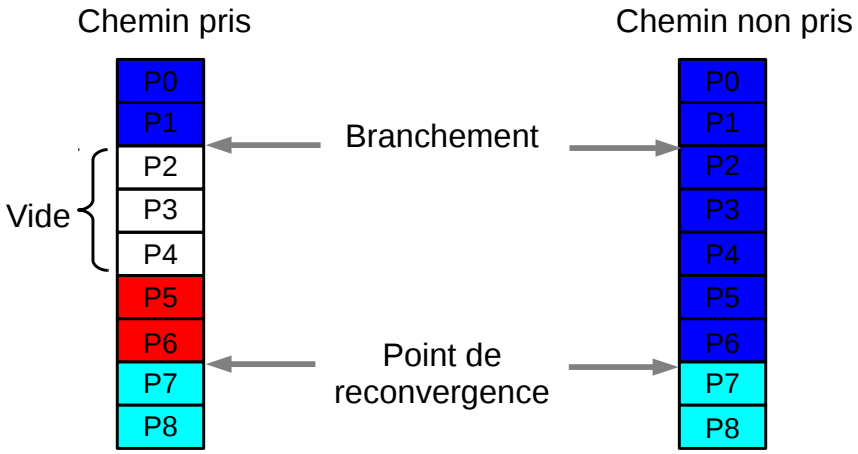


FIGURE 3.1 – *Illustration du mécanisme d'insertion des vides pour forcer l'allocation symétrique des ressources sur les chemins pris et non pris. L'exemple montre le principe pour l'allocation des registres physiques destination.*

La figure 3.1 illustre le mécanisme d'insertion de vides dans le cas des registres physiques. Dans cet exemple, le chemin pris consomme deux registres ($P5$ et $P6$) et le chemin non pris en consomme cinq ($P2$ à $P6$). En « perdant » trois registres ($P2$, $P3$ et $P4$) sur le chemin pris, on s'assure que les mêmes registres physiques seront utilisés sur les deux chemins après le point de reconvergence.

Forcer l'allocation des mêmes ressources exactement pour les instructions IC sur les deux chemins est possible seulement si le nombre de ressources consommées sur les deux chemins est connu. Dans la partie 3.2.1, un mécanisme simple de détection du point de reconvergence est présenté. Il est utilisé en particulier pour compter l'utilisation des ressources. Une fois que cette utilisation est connue pour les deux chemins, cette connaissance étant acquise lors des précédentes instances dynamiques du branchement, les informations ainsi récupérées peuvent être utilisées lors de la prochaine occurrence du branchement pour créer des vides de la taille appropriée.

Comme précédemment précisé, seuls les résultats des instructions indépendantes du contrôle et indépendantes des données (ICID) peuvent être préservés lors de l'exécution sur le bon chemin. Les dépendances de données, qu'elles découlent des registres ou de la mémoire, doivent donc être identifiées. Ce processus sera détaillé dans la partie 3.2.2.

3.2 Description détaillée du mécanisme SYRANT

Dans cette partie, les principaux mécanismes utilisés dans SYRANT sont détaillés : tout d'abord le mécanisme de détection de la reconvergence et ensuite celui permettant l'identification des instructions ICID.

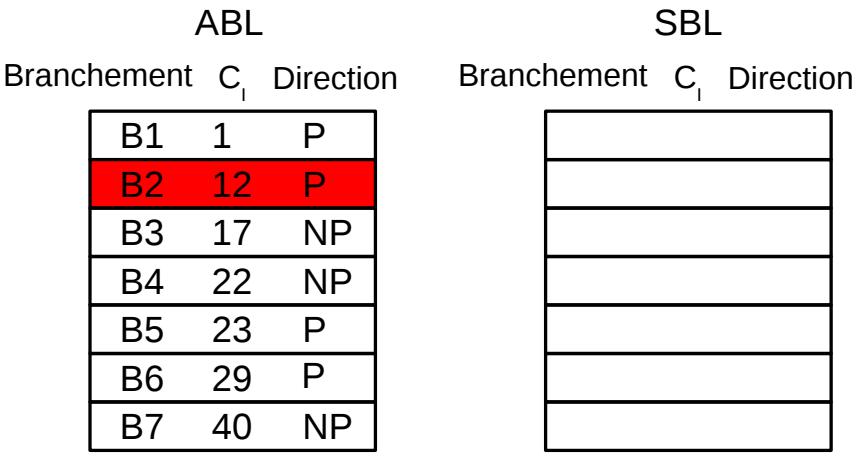
3.2.1 Détection du point de reconvergence : la ABL/SBL

Pour dénombrer les ressources consommées sur les chemins pris et non pris, le point de reconvergence doit être détecté. Néanmoins, en pratique, il n'est pas nécessaire de connaître le besoin en ressources effectif mais plutôt la différence des besoins entre les deux chemins, qui est directement la taille du vide nécessaire. Ainsi, plutôt que de détecter le point de reconvergence précis, ce qui nécessiterait de comparer chaque instruction des deux chemins, nous avons choisi de détecter le premier branchement après le point de reconvergence. Le point de reconvergence tel que nous le définissons n'est donc pas le point de reconvergence exact, mais un point qui se situe forcément après (non strictement) le point de reconvergence réel. Néanmoins, cette limitation ne pose pas de problème pour notre mécanisme et en pratique, le point de reconvergence que nous détectons n'est pas très éloigné du vrai point de reconvergence.

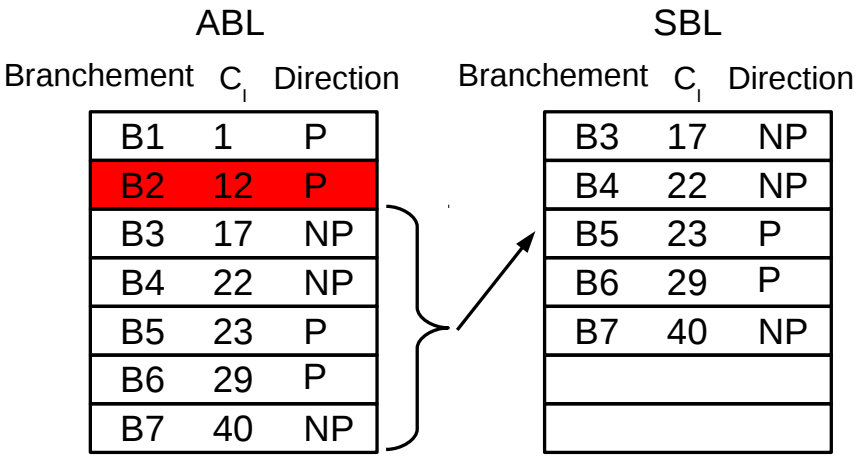
Détection de la reconvergence

Pour la détection du point de reconvergence, trois structures matérielles sont utilisées. La première est la liste des branchements actifs ou *Active Branch List* (ABL). Celle-ci est utilisée pour mémoriser les branchements sur le chemin en cours de traitement (figure 3.2a). Lors de la correction d'une mauvaise prédiction, tous les branchements du mauvais chemin qui sont dans l'ABL sont copiés dans la liste de sauvegarde des branchements ou *Shadow Branch List* (SBL) (figure 3.2b). La mémorisation des branchements dans l'ABL reprend sur le bon chemin après la résolution de la mauvaise prédiction de branchement. Tout nouveau branchement qui entre dans le pipeline est cherché dans la SBL. Le premier branchement pour lequel il y a correspondance est le point de reconvergence (figure 3.2c). Si le branchement mal prédit est une instruction de boucle, c'est-à-dire que plusieurs instances du même branchement sont présentes dans l'ABL ou la SBL, nous avons choisi de limiter la détection du point de reconvergence à la première boucle mémorisée. Cela signifie que, dans ce cas, la recherche dans la SBL se termine à la deuxième instance du branchement mal prédit. De même, la recherche du point de reconvergence s'arrête lorsqu'une seconde instance du branchement mal prédit est mémorisée sur le bon chemin.

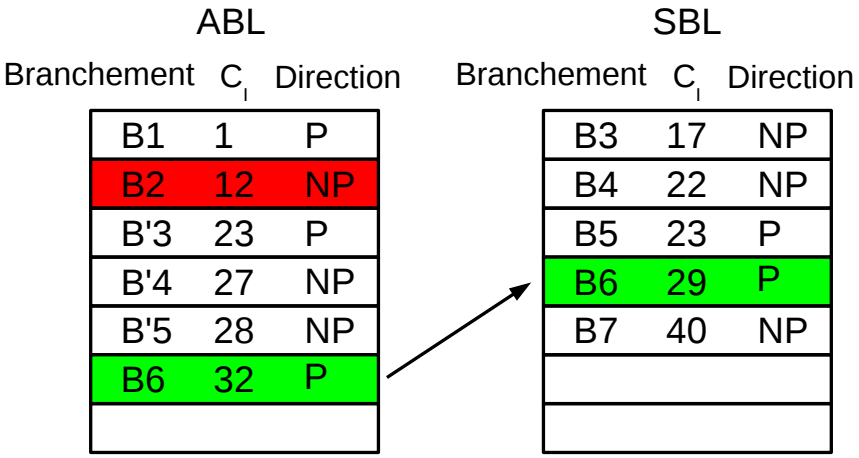
Les entrées de l'ABL et de la SBL sont identiques (figure 3.3a). Une entrée permet d'identifier un branchement et d'enregistrer la quantité de ressources utilisées sur le chemin courant. Elle est composée du PC du branchement, du nombre de registres physiques qui ont été utilisés jusqu'au branchement, du nombre d'instructions qui sont entrées dans le pipeline avant le branchement, du nombre d'entrées de la LSQ consommées et de la direction courante du branchement (pris ou non pris). Ainsi, lors de la détection du point de reconvergence, on peut déterminer la taille du vide à créer en termes de ressources en calculant simplement la différence entre les différents champs de l'entrée courante de l'ABL et l'entrée correspondante trouvée dans la SBL. Cependant,



(a) L'ABL et la SBL avant la détection d'une mauvaise prédiction de branchement $B2$.



(b) Première étape de la correction de $B2$.



(c) Détection du point de reconvergence de $B2$.

FIGURE *3.2* – *Processus d'observation durant la correction d'un branchement.*

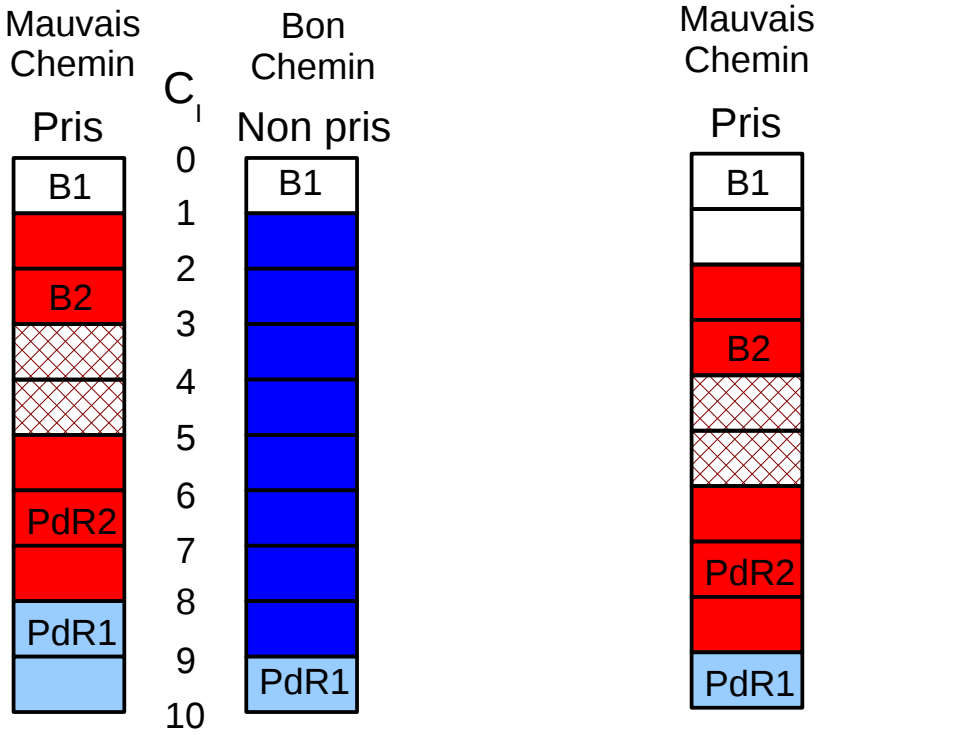
PC	#Reg$_{\text{util.avt.br.}}$	#Insts$_{\text{trait.avt.br.}}$	#LSQ$_{\text{util.avt.br.}}$	Direction Courante

(a) Une entrée de l'ABL/SBL : PC du branchement, nombre de registres physiques alloués avant ce branchement, nombre d'instructions traitées avant ce branchement et direction courante du branchement.

PC	TailleVide$_{R}$	TailleVide$_{ROB}$	TailleVide$_{LSQ}$

(b) Une entrée de la RANT : PC du branchement, valeur signée de la taille du vide pour les registres physiques (R), pour les entrées du ROB (ROB) et pour les entrées de la LSQ (LSQ).

FIGURE *3.3 – Une entrée de l'ABL/SBL et une entrée de la RANT*



(a) Allocation des entrées du ROB après la correction du branchement $B1$, lors de la première mauvaise prédiction de ce branchement.

(b) Allocation des entrées du ROB, lors des fois suivantes où $B1$ est rencontré et prédit pris.

FIGURE *3.4 – Processus d'observation du branchement à différents moments de l'exécution du programme.*

il faut noter que la taille du vide ainsi calculée comprend aussi le vide qui a pu être créé pour des branchements reconvergents situés entre le branchement mal prédit courant et son point de reconvergence (voir la figure 3.4).

Utilisation de la reconvergence

Lorsqu'une reconvergence est détectée, les tailles des vides associés à ce branchement reconvergent sont calculées. Cette information est ensuite sauvegardée dans la table de l'allocation des ressources sur les chemins pris et non pris ou *Resource Allocation on Not-taken and Taken paths* (RANT) *table*. Une entrée de cette table (figure 3.3b) est composée du PC du branchement et de la valeur signée de la taille des vides à créer pour les registres physiques, les entrées du ROB et les entrées de la LSQ.

Lorsque les instructions entrent dans le pipeline, une recherche est faite dans la table RANT. Si une correspondance est trouvée, le mécanisme d'insertion de vides peut être activé. Dans la partie 3.4, nous verrons que ce mécanisme d'insertion n'est pas toujours bénéfique mais qu'il peut être activé sous certaines conditions.

3.2.2 Identification des instructions indépendantes du contrôle et respect des dépendances de données

Seuls les résultats des instructions ICID doivent être sauvegardés. Le résultat d'une instruction IC peut être dépendant des données de deux manières différentes : les dépendances au niveau des registres, c'est-à-dire qu'un registre source est calculé différemment sur le bon chemin et donc sa valeur change, et les dépendances liées à la mémoire. Une instruction IC doit donc être à la fois indépendante des registres (IR) et indépendante de la mémoire (IM).

Identification des instructions indépendantes du contrôle

Le mécanisme de détection du point de reconvergence présenté précédemment associé au mécanisme d'insertion des vides devrait assurer que, après une mauvais prédiction, une instruction IC aura la même entrée dans le ROB que celle qu'elle avait sur le mauvais chemin. Ainsi, détecter une instruction IC se fait directement. L'instruction est comparée avec celle qui occupait l'entrée du ROB sur le mauvais chemin. Si l'instruction à mettre dans cette entrée est la même que celle l'occupant déjà (si les PC sont les mêmes), alors l'instruction est indépendante du contrôle.

Bien que le point de reconvergence réel puisse être avant celui qui est détecté, SYRANT est néanmoins capable d'identifier les instructions entre ces deux points comme étant IC. Comme ces instructions se trouvent sur les deux chemins (car ce sont des instructions IC), les ressources qu'elles consomment sont comptées sur les deux chemins. De cette manière, la différence calculée est exactement celle des ressources consommées par les instructions dépendantes du contrôle (DC) seulement, et non pas celle des ressources consommées entre le branchement et son point de reconvergence détecté. La taille du vide à créer est donc calculée de manière exacte malgré la détection d'un point de reconvergence sûr mais approximatif. Il en résulte que si un vide est inséré, toutes les instructions IC auront les mêmes ressources allouées sur les deux chemins, même celles situées entre le point de reconvergence réel et celui détecté. Ainsi, tant que la taille du vide inséré est correcte, toutes les instructions IC peuvent être identifiées.

Il est à noter que même si des vides sont insérés, il est possible que la taille des deux chemins ne soit pas la même. Dans ce cas, l'identification des instructions IC ne peut pas fonctionner.

Le traitement du pipeline continue alors normalement, perdant potentiellement des opportunités de gains de performance, mais sans provoquer de pertes de performance ni introduire d'état incorrect.

Identifier et propager les dépendances de registres

Le processus de renommage est chargé de préserver les résultats des instructions ICID qui ont déjà été exécutées sur le mauvais chemin et d'invalider les résultats des instructions DC et ICDD.

Après une mauvaise prédiction, le traitement des instructions reprend et celles qui entrent dans le pipeline sont comparées à celles qui occupaient leur entrée dans le fichier de registres, dans le ROB et dans la LSQ. Pour vérifier la validité des données déjà présentes dans ces structures, différentes règles sont appliquées. Le résultat des instructions autres que les *loads* peut être conservé si leurs opérandes restent valides sur le bon chemin.

Une des difficultés provient des différentes versions qu'une même opérande peut avoir successivement dans le même registre physique et pour les mêmes instances successives d'une instruction. Ce cas arrive en particulier quand une instruction IC a la même forme renommée sur le mauvais et le bon chemins mais qu'une de ses opérandes a eu sa valeur modifiée sur le bon chemin. Pour s'assurer de l'utilisation de la bonne version de l'opérande, nous proposons un processus d'étiquetage, décrit dans la suite.

Nous définissons une séquence de renommage comme une séquence d'instructions qui entrent dans le pipeline, qui ont été décodées et renommées sans interruption par la correction d'une mauvaise prédiction de branchement ou la correction d'une dépendance *load*/*store*. Une étiquette unique appelée *Rename Sequence tag* (RS-tag) est associée à chaque séquence de renommage. Pour simplifier, cette étiquette est utilisée pour déterminer à quel moment de l'exécution les informations associées aux instructions ont été calculées.

Le processus de renommage des registres agit de la manière suivante pour préserver le travail ICID : après une mauvaise prédiction, le RS-tag courant est modifié (incrémenté par exemple). Au renommage, la valeur du RS-tag courant est associée à chaque instruction dans le ROB et à son registre physique destination dans la *mapping table*. Pour les instructions autres que les *loads*, les règles suivantes sont appliquées :

1. **Si** le PC de la nouvelle instruction est différent du PC de l'ancienne instruction dans la même entrée du ROB, alors on enregistre le nouveau RS-tag à la fois dans l'entrée du ROB et dans la *mapping table*. On marque aussi le registre physique destination comme invalide et l'instruction comme non exécutée.
2. **Sinon** :
   - **Si** l'instruction n'a aucun registre opérande mais produit un résultat, alors on garde l'ancien RS-tag et on conserve la validité du résultat et le statut d'exécution de l'instruction (si elle est exécutée ou non).
   - **Si** l'instruction a des registres opérandes dont le renommage, en incluant le RS-tag, est identique au renommage du mauvais chemin, alors on garde l'ancien RS-tag et on

conserve la validité du résultat et le statut d'exécution de l'instruction.
Sinon on enregistre le nouveau RS-tag à la fois dans l'entrée du ROB et dans la *mapping table*. On marque aussi le registre physique destination comme invalide et l'instruction comme non exécutée.

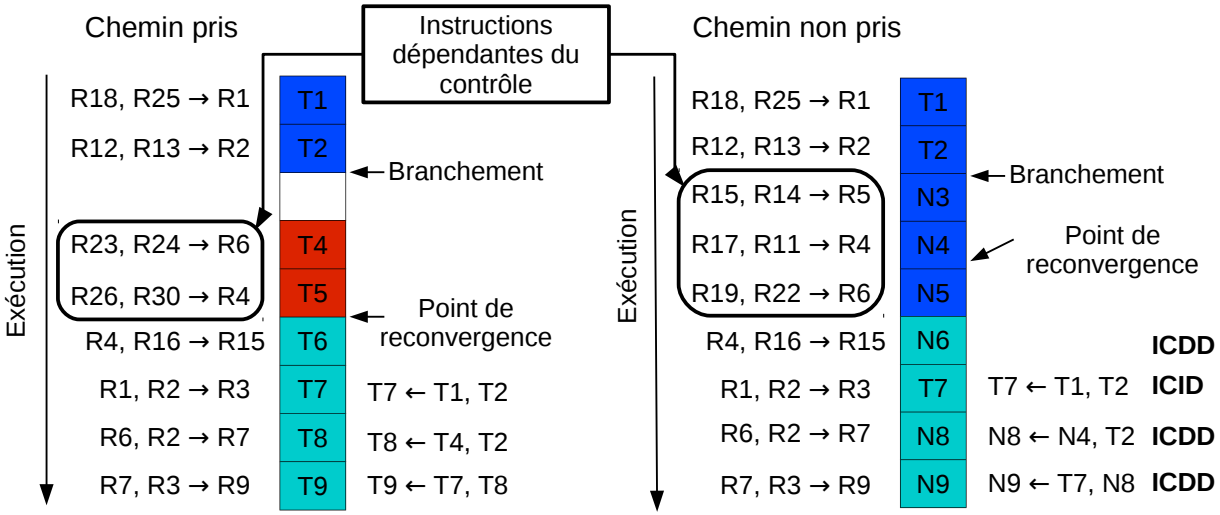


FIGURE *3.5* – *Illustration du mécanisme de renommage modifié pour permettre l'identification des instructions dépendantes des données. Ti désigne le registre physique numéro i d'étiquette T. Ni désigne le registre physique numéro i d'étiquette N.*

Ce processus, illustré sur la figure 3.5, suffit pour toutes les instructions excepté les *loads*. Pendant le premier traitement de l'instruction (sur le mauvais chemin), une étiquette T est associée au résultat de chaque instruction. Après une mauvaise prédiction, l'étiquette N est associée à toutes les instructions DC et ICDD.

Pour propager les dépendances mémoire, les instructions *load*/*store* ont besoin d'un traitement spécial impliquant la LSQ. Dans la LSQ, l'entrée associée à un *store* sera marquée comme invalide, c'est-à-dire sera considérée comme abritant des données invalides, si le *store* ne correspond pas à celui déjà présent dans l'entrée de la LSQ ou l'entrée du ROB qui lui ont été données. S'il y a correspondance, l'entrée sera quand même marquée comme invalide si l'opérande de l'instruction est détectée comme invalide. Dans les autres cas, la validité de l'exécution sur le mauvais chemin est préservée.

Pour préserver la validité du résultat d'une instruction *load* IC, son calcul d'adresse doit être valide, c'est-à-dire que ses registres opérandes doivent être valides. Cependant, la validité de la donnée chargée par le *load* dépend aussi de la validité effective de la donnée lue en mémoire. Un *load* peut charger sa donnée soit à partir de la mémoire soit à partir d'un *store* non encore validé en mémoire, c'est-à-dire à partir de données présentes dans une entrée de la LSQ. Ainsi, la donnée du *load* peut avoir été transmise au *load* sur le mauvais chemin par un *store* qui est devenu invalide sur le bon chemin. Pour pouvoir traiter ce cas, une fonctionnalité supplémentaire est implémentée dans la LSQ. Lorsqu'une donnée d'un *store* S non validé en mémoire est transmise à un *load* L, l'index de l'entrée de la LSQ associée à S est enregistrée dans l'entrée de la LSQ

de L. Lorsque L passe à l'étage de renommage sur le bon chemin, la validité du *store* S est vérifiée dans la LSQ. Si la donnée associée à S est invalide alors L est marqué comme invalide (son registre destination et son entrée de la LSQ sont invalidés).

Remarque importante concernant la LSQ dans SYRANT

Lors de l'exécution d'un *store* S, tous les *loads* plus jeunes que S qui ont été exécutés spéculativement, c'est-à-dire qui ont lu leur donnée en mémoire avant que S ne soit exécuté, doivent être vérifiés pour s'assurer qu'aucune dépendance mémoire n'a été violée. Comme SYRANT préserve des résultats de *loads* du mauvais chemin qui peuvent être postérieurs à S, ces résultats doivent eux aussi être invalidés dans le cas d'une violation de dépendance mémoire avec S.

Prédiction de dépendances mémoire

Les violations de dépendances mémoire sont coûteuses en termes de performance. C'est pourquoi des prédicteurs sont utilisés pour tenter de les éviter. Plusieurs prédicteurs ont été proposés dans la littérature. Le prédicteur *synonym* [38], le prédicteur *store sets* [14] et le prédicteur *store barrier* [23] tentent d'identifier les *loads* qui sont dépendants de *stores* pour les lancer après que les *stores* dont ils dépendent aient été exécutés.

Notre implémentation de SYRANT est compatible avec ces prédicteurs et nous utilisons le prédicteur *store sets* dans notre simulateur.

3.2.3 Continuer l'exécution du mauvais chemin après la correction d'une mauvaise prédiction

Dans un processeur superscalaire conventionnel, il y a peu d'intérêt à continuer l'exécution après une mauvaise prédiction de branchement. Cependant, lorsque l'on cherche à exploiter l'indépendance de contrôle, il devient intéressant de continuer l'exécution des instructions, en particulier de celles qui sont ICID. Ainsi, les instructions sur le mauvais chemin ne sont pas totalement effacées du pipeline lors de la détection d'une mauvaise prédiction. Nous appelons ces instructions des instructions fantômes. Celles-ci continuent à être exécutées par le pipeline comme des instructions valides mais avec une priorité moindre que les instructions normales. Une instruction fantôme est invalidée si une des ses ressources est réclamée par le pipeline pour une instruction valide. Si tout se passe bien, cette instruction est l'instance valide sur le bon chemin de l'instruction fantôme.

L'utilité d'un mécanisme similaire a été discutée dans [32].

3.2.4 Correspondance artificielle de la taille des chemins

Lorsqu'un branchement entre dans le pipeline, une recherche est lancée dans la table RANT. S'il y a une correspondance, les informations sur la taille des vides à créer sont récupérées. En utilisant ces informations, si nécessaire, des vides sont créés sur le chemin le moins consommateur de ressources.

Insertion des vides

Les vides sont insérés après le branchement, soit au moment où il entre dans le pipeline, soit après sa correction. Après cette étape, les instructions continuent à entrer dans le pipeline, à être décodées et renommées comme d'habitude. En pratique, insérer un vide dans le ROB, la *free list* des registres physiques et la LSQ revient simplement à déplacer un pointeur et à laisser quelques entrées libres.

Recyclage des ressources

Lorsqu'un vide est inséré après un branchement, les ressources sont réservées d'une manière différente que pour les instructions normales. Ces ressources doivent donc être recyclées afin d'éviter un phénomène de famine. Les entrées du ROB et de la LSQ sont assez simples à recycler. En effet, ces structures sont des tampons circulaires et les entrées sont libérées dans le même ordre qu'elles ont été allouées. Libérer des entrées revient simplement à incrémenter un pointeur. Pour les registres physiques, tous ceux associés au vide créé doivent être recyclés dans la *free list* lorsque le branchement est validé à la fin du pipeline.

3.2.5 Invalidation sélective d'instructions en utilisant SYRANT

Lorsqu'une dépendance mémoire lecture après écriture est violée, c'est-à-dire qu'un *load* est exécuté prématurément et charge une mauvaise valeur, la majorité de la chaîne d'instructions dépendantes de ce *load* peut déjà avoir été exécutée ou lancée avant que la violation ne soit détectée. Toutes ces instructions doivent être invalidées. L'invalidation sélective est un mécanisme complexe à implémenter dans un pipeline et la plupart des processeurs vident simplement le pipeline de toutes ces instructions, tout en comptant sur les mécanismes de prédiction des dépendances pour éviter ce vidage autant que possible. SYRANT offre une solution intermédiaire entre une implémentation *ad-hoc* préservant toutes les instructions exécutées encore valides et un vidage complet du pipeline.

3.2.6 Considérations sur la complexité matérielle du mécanisme

SYRANT induit quelques modifications du pipeline d'un processeur superscalaire, mais le flot d'informations d'un processeur superscalaire conventionnel est essentiellement respecté. Les principales structures du pipeline à exécution dans le désordre ne sont que marginalement modifiées (RS-tag ajouté au nom du registre stocké dans le ROB et l'index ajouté dans la LSQ pour retrouver le *store* lors d'une transmission de données entre ce *store* et un *load*). Le processus d'observation permettant de calculer les tailles des vides représente le coût majeur, avec l'introduction de l'ABL, de la SBL et de la table RANT. Mais ce processus peut être effectué en arrière plan et n'est pas sur le chemin critique. L'ajout de quelques comparateurs supplémentaires dans les premiers étages du pipeline nécessaires pour identifier les instructions ICID peut conduire à l'ajout d'un étage de pipeline supplémentaire.

3.3 Utilisation des branchements calculés sur le mauvais chemin pour améliorer la prédiction de branchements

La structure ABL/SBL proposée dans la partie 3.2.1 pour détecter le point de reconvergence après un branchement mal prédit peut aussi être utilisée pour enregistrer les directions des branchements du mauvais chemin. Elle devient de manière évidente une aide dans le contexte de SYRANT puisqu'elle permet d'exploiter directement le résultat des branchements ICID lors du traitement des instructions sur le bon chemin. Il est intéressant de remarquer que la structure ABL/SBL peut être utile en elle-même même si le reste des mécanismes de SYRANT n'est pas implémenté. Un mécanisme similaire a été présenté dans [1].

Lorsqu'un branchement B a été calculé lors de l'exécution sur le mauvais chemin, sa direction calculée a été enregistrée dans l'ABL et est donc copiée dans la SBL. Si le mécanisme de l'ABL/SBL détecte que le branchement B est postérieur au point de reconvergence, alors, lorsque B entre à nouveau dans le pipeline sur le bon chemin, sa direction qui a été calculée sur le mauvais chemin peut être utilisée comme source de la prédiction à la place de celle donnée par le mécanisme de prédiction de branchements usuel. Le mécanisme ABL/SBL n'est pas capable de déterminer à lui seul si les branchements enregistrés sont ICID ou ICDD. Cependant, nous avons observé que pour de nombreux programmes, la qualité de la prédiction donnée par l'ABL/SBL est meilleure que celle du prédicteur TAGE, considéré comme l'état de l'art, que nous utilisons dans nos simulations. De plus, nous avons découvert qu'un simple compteur à saturation à 4 bits suffisait à suivre au cours de l'exécution si cette propriété s'appliquait ou non, c'est-à-dire si cela est intéressant d'utiliser la prediction donnée par l'ABL/SBL ou non.

Dans le suite de ce document, nous ferons référence à une prédiction s'appuyant sur les informations enregistrées dans la SBL comme étant une prédiction SBL.

Il est important de remarquer que l'ajout du mécanisme de la prédiction SBL dans le pipeline reste localisé au niveau du prédicteur de branchements, puisqu'aucun autre composant du processeur superscalaire ne subit de modification de sa structure globale.

3.4 Limitation de la taille des vides

Les premières expérimentations qui ont été conduites ont montré que pour la plupart des programmes de test, l'application systématique de SYRANT conduit à gaspiller une grande quantité de ressources dans les vides, amenant généralement des pertes de performance. Dans notre environnement de simulation, tous les programmes de test sauf un souffraient de pertes de performance. Ces pertes sont principalement dues au gaspillage des entrées du ROB et de la LSQ, puisque cela implique une diminution du nombre d'instructions qui sont traitées dans le pipeline.

En conséquence, nous avons exploré plusieurs techniques pour limiter le nombre de vides insérés ainsi que leur taille, basées sur leur utilité attendue et sur l'espérance d'un impact modéré sur les performances en cas d'insertion sur le bon chemin (c'est-à-dire que le branchement n'est pas mal prédit et donc les vides sont inutiles). Les filtres les plus utiles concernant l'insertion de

vides sont décrit dans la suite.

Pour limiter les pertes de performance possibles sur le bon chemin, une première possibilité est de n'insérer les vides que si le branchement est mal prédit. Les vides ne sont donc insérés qu'après la correction d'un branchement mal prédit. Ainsi, des vides sont insérés seulement s'il y a une chance que du travail utile puisse être récupéré. Cependant, avec cette méthode, les vides ne sont insérés que si le mauvais chemin était le chemin le plus consommateur en ressources. Cette stratégie cible approximativement les mêmes branchements que pour la *Selective Branch Recovery* [20].

Bien que l'insertion de vides sur le chemin corrigé semble naturelle, on peut utiliser plusieurs indicateurs pour estimer l'utilité de l'insertion de vides sur le chemin prédit. La confiance en la prédiction de branchements est un indicateur naturel. Pour le prédicteur TAGE [54], nous utilisons l'estimateur de confiance défini dans [51], qui consiste principalement à regarder quel est le composant du prédicteur qui a fourni la prédiction et quelle est la valeur du compteur associé. Le prédicteur TAGE a aussi été modifié d'après [51] pour assurer une très bonne couverture des prédictions de faible confiance et un très faible taux de mauvaises prédictions pour les prédictions à confiance élevée.

La qualité des informations de reconvergence est aussi un très bon estimateur de l'utilité de l'insertion des vides. En effet, on ne voudra insérer de vides que si les informations sur la consommation en ressources des chemins pris et non pris sont suffisamment stables, c'est-à-dire si la reconvergence a été détectée plusieurs fois et que la taille des vides à insérer est la même à chaque fois. Cet estimateur peut être implémenté avec un compteur de stabilité dans chaque entrée de la table RANT servant à compter combien de fois le branchement associé a reconvergé. Si la taille des vides à insérer change entre deux reconvergences, le compteur est remis à zéro. Les vides ne sont insérés que si le compteur atteint un seuil prédéfini.

Limiter la taille de chaque vide inséré est aussi un moyen de réduire le gaspillage de ressources généré par ces vides. La probabilité que les instructions IC soient dépendantes des données augmente avec la taille des vides puisque cela signifie qu'un des deux chemins possédait un grand nombre d'instructions DC. Ainsi, les vides ne sont insérés que si leur taille est inférieure à une limite prédéfinie.

Ces différents filtres peuvent bien sûr être combinés pour mieux sélectionner les vides à insérer qui seront les plus à même d'être utiles.

3.5 Évaluation des performances

Une étude basée sur la simulation a été conduite dans le but d'évaluer le mécanisme SYRANT. Nous avons dérivé notre simulateur à exécution dans le désordre de l'environnement SimpleScalar [8]. Plusieurs modifications ont été apportées à l'environnement de base. Le modèle du pipeline a notamment été complètement refait à partir de zéro.

3.5.1 Caractéristiques du simulateur

Le simulateur est construit autour d'un modèle dirigé par l'exécution. L'exécution sur le mauvais chemin est pleinement modélisée et les branchements sont résolus dans le désordre. Les instructions *load* peuvent être exécutées dans le désordre par rapport aux instructions *store*, de manière spéculative. Une *free list* des registres physiques libres est maintenue. La validité des résultats des instructions simulées est vérifiée au moment de la validation de l'instruction en sortie du pipeline par un simulateur fonctionnel de confiance. Le modèle mémoire a été corrigé pour tenir compte de la contention du bus mémoire.

Sauf mention contraire, le simulateur modélise un processeur superscalaire très puissant avec un pipeline capable de traiter huit instructions par cycle. La taille du ROB est de 1024 entrées, celle de la LSQ est de 512 entrées. Il y a 2048 registres entiers et flottants. Nous avons choisi des structures très grandes afin de maximiser le nombre d'instructions en vol. La largeur des différents étages est définie de manière à faire entrer assez d'instructions dans le pipeline avant la détection d'une mauvaise prédiction pour atteindre le point de reconvergence d'un maximum de branchements mal prédits. En ce qui concerne SYRANT, l'ABL et la SBL comportent 256 entrées et la table RANT a 4096 entrées.

Le processeur modélisé possède un prédicteur de branchements à l'état de l'art, le prédicteur TAGE, dans sa version décrite dans [54]. Au maximum deux blocs de base peuvent entrer dans le pipeline pour un maximum de huit instructions par cycle. Le prédicteur *store sets* [14] est utilisé pour prédire les dépendances mémoires. La pénalité minimale en cas de mauvaise prédiction de branchement est de vingt cycles. Les autres caractéristiques du simulateur sont résumées dans la table 3.1.

Nous ferons référence à cette configuration comme étant la configuration de base (BASE).

3.5.2 Jeu de tests

Le jeu de tests utilisé est une partie du jeu de tests SPEC2006 [59]. Comme nous avons ciblé le jeu d'instructions *Alpha* [37], nous n'avons été capable de compiler que dix-huit des programmes de ce jeu de tests. Il y a onze programmes de test de type entier (faisant majoritairement appel à des instructions opérant sur des valeurs entières) et sept programmes de test de type flottant (faisant majoritairement appel à des instructions opérant sur des valeurs à virgule flottante). Les programmes de type entier sont : *473.astar*, *401.bzip2*, *403.gcc*, *445.gobmk*, *464.h264ref*, *456.hmmer*, *429.mcf*, *471.omnetpp*, *400.perlbench*, *462.libquantum* et *458.sjeng*. Les programmes de type flotant sont : *470.lbm*, *437.leslie3d*, *433.milc*, *444.namd* et *453.povray*. Pour réduire le temps de simulation, nous avons utilisé la méthodologie *Simpoint* [21] pour résumer chaque programme de test en un ensemble de tranches de cent millions d'instructions. Chaque tranche représente une certaine partie de l'exécution du programme de test à laquelle est associé un poids qui correspond à l'importance de cette partie par rapport à l'exécution totale. Pour chaque programme, les résultats montrés sont la moyenne pondérée des résultats obtenus sur chaque tranche de l'ensemble. La table 3.2 présente le nombre de *Simpoint* utilisés pour chaque programme de test.

Largeur entrée/décodage/lancement/validation	8/8/8/8
Entrées du ROB & Registres physiques	1024
Entrées de la LSQ	512
Cache de données L1	64Ko 4 voies associatif par ensemble
Cache d'instructions L1	64Ko 4 voies associatif par ensemble
Cache partagé L2	4Mo 8 voies associatif par ensemble
TLB pour les données	4096 entrées totalement associatif
Latence mémoire	100 cycles
UAL entière	6
Multiplication entière	2
UAL flottante	4
Multiplication flottante	4
Ports mémoire	4
Prédicteur de branchements	BTB : 4 voies, 1K entrées RAS : 16 entrées, détection de la corruption due au mauvais chemin TAGE : 256 Kbits, 1+12 composants, 15K entrées au total
Prédicteur de dépendances mémoire	*Store sets*
Pénalité minimale de mauvaise prédiction	20 cycles

Table *3.1* – *Caractéristiques principales de l'architecture simulée*

Programmes de test	Nombre de *Simpoint*	IPC (BASE)	Taux de mauvaises prédictions (MPKI)	
			(BASE)	(avec prédiction SBL)
Programmes de test de type flottant				
410.bwaves	20	5,01	0,01	0,01
435.gromacs	19	4,84	1,14	1,05
470.lbm	13	0,97	0,03	0,03
437.leslie3d	22	2,56	0,01	0,01
433.milc	19	1,39	0,001	0,001
444.namd	20	4,18	1,56	1,17
453.povray	19	3,95	0,35	0,35
Programmes de test de type entier				
473.astar	18	2,96	11,14	8,95
401.bzip2	17	4,26	2,91	2,63
403.gcc	19	3,65	0,78	0,77
445.gobmk	17	2,60	6,93	6,63
464.h264ref	18	4,87	0,60	0,54
456.hmmer	9	2,57	9,05	7,88
429.mcf	24	0,60	8,32	6,66
471.omnetpp	13	1,80	2,56	2,50
400.perlbench	8	2,89	0,33	0,32
462.libquantum	20	1,74	0,05	0,05
458.sjeng	26	2,67	4,01	4,00

Table *3.2* – *Caractéristiques des programmes du jeu de tests : l'IPC mesure le nombre d'instructions par cycle que le processeur est capable de traiter. Le nombre de mauvaises prédictions par kilo-instructions (MPKI) mesure la précision du prédicteur de branchements.*

3.5.3 Taux de mauvaises prédictions des programmes de test

La table 3.2 montre aussi le taux de mauvaises prédictions de branchements pour chacun des programmes de test utilisés. Ces taux sont mesurés en nombre de mauvaises prédictions par kilo-instructions (MPKI), c'est-à-dire par millier d'instructions. Comme l'illustre la table 3.2, certains programmes de test ont un taux de mauvaises prédictions très bas. Sur ces programmes, il est attendu qu'un mécanisme exploitant l'indépendance de contrôle n'augmentera pas les performances d'exécution. Il s'agit de *410.bwaves*, *470.lbm*, *437.leslie3d*, *433.milc*, *453.povray*, *403.gcc*, *464.h264ref*, *400.perlbench* et *462.libquantum*. Pour la classe de programmes ayant des taux de mauvaises prédictions très bas, l'activation et la désactivation dynamique de SYRANT devrait être considérée pour optimiser la consommation d'énergie. Un tel mécanisme ne sera pas décrit dans le présent document.

Par ailleurs, les autres programmes de test présentent un taux de mauvaises prédictions assez significatif, en particulier pour *473.astar*, *445.gobmk*, *456.hmmer* et *429.mcf*.

3.5.4 Caractérisation partielle et détection de la reconvergence

Le mécanisme ABL/SBL est capable de détecter une part significative des cas de reconvergence sur les programmes de test ayant un fort taux de mauvaises prédictions de branchements (figure 3.6).

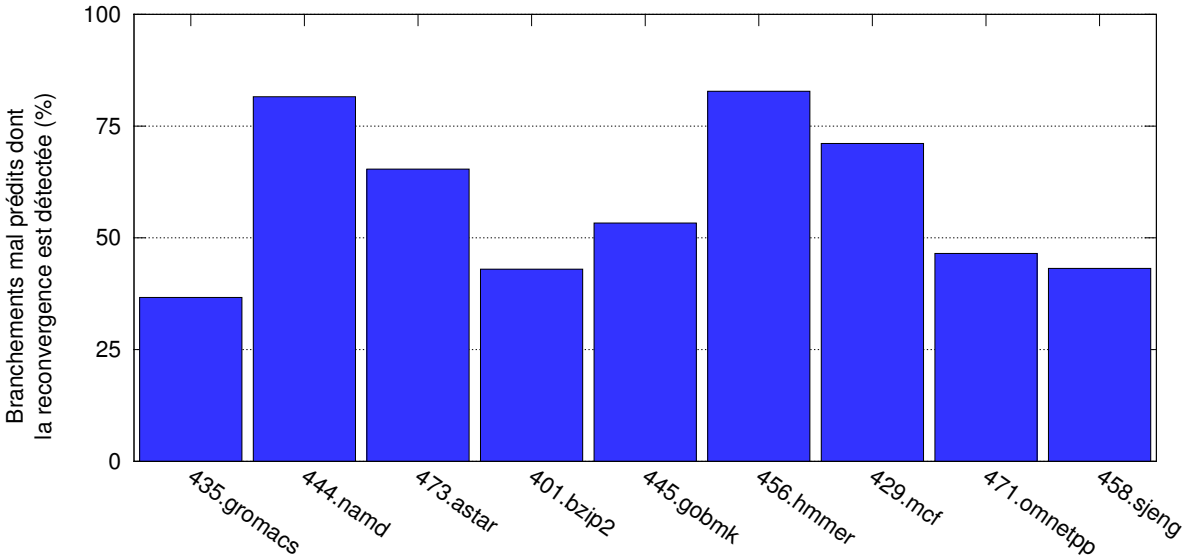


Figure 3.6 – *Part des branchements dynamiques mal prédits pour lesquels un point de reconvergence est détecté.*

La reconvergence n'est pas détectée dans plusieurs situations. Le principal cas arrive quand le branchement qui doit être détecté comme point de reconvergence n'entre pas dans le pipeline avant que la mauvaise prédiction ne soit résolue. De plus, comme nous ne cherchons pas à détecter plusieurs points de reconvergence en parallèle, seul le point de reconvergence du dernier

branchement mal prédit est recherché. Ainsi, si une mauvaise prédiction est détectée avant que le point de reconvergence pour le branchement mal prédit courant ne soit détecté, la recherche de celui-ci est abandonné au profit de la recherche de celui du nouveau branchement mal prédit.

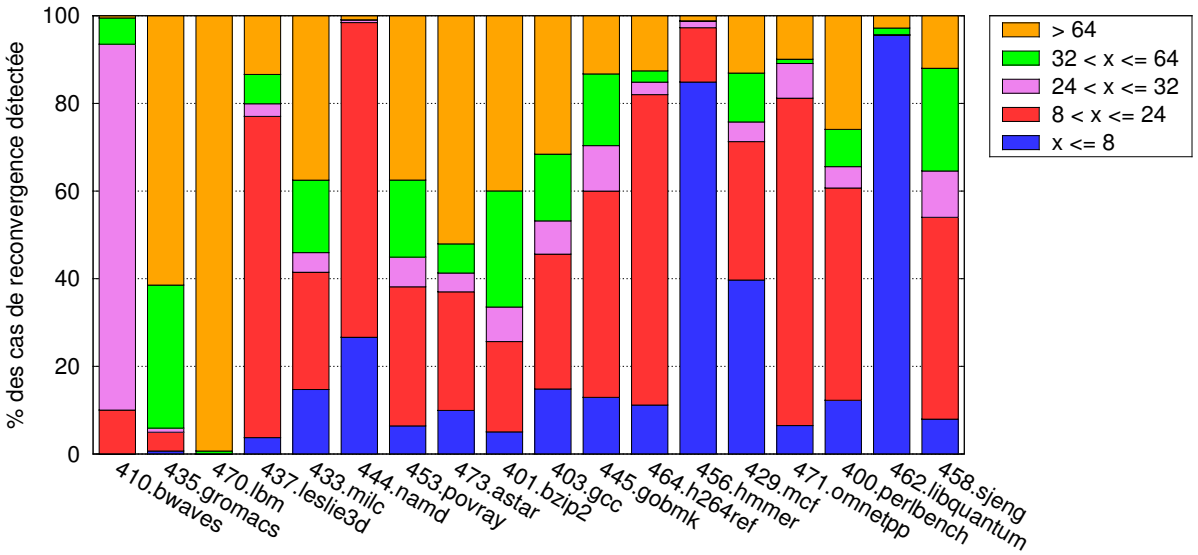


Figure *3.7* – *Décomposition des cas de reconvergence : la taille x est la taille du plus long des deux chemins reconvergents. On parle aussi de la taille du chemin reconvergent.*

De manière intuitive, on peut penser que plus le chemin reconvergent (c'est-à-dire le chemin constitué des instructions DC) est court, plus la probabilité d'avoir des instructions ICID est grande et plus la probabilité que ces instructions ICID soient déjà exécutées sur le mauvais chemin est élevée. La figure 3.7 illustre la distribution des tailles du chemin le plus long entre les deux chemins reconvergents. Pour la plupart des programmes de test, la grande majorité des cas de reconvergence détectée se font après moins de vingt-quatre instructions. Le chemin reconvergent est encore plus court pour *456.hmmer* et *462.libquantum* (moins de huit instructions). Presque tous les cas de reconvergence pour *410.bwaves* arrivent entre vingt-quatre et trente-deux instructions. Mais pour quelques autres programmes de test comme *435.gromacs*, *470.lbm* et *473.astar*, la plupart de leurs cas de reconvergence se font après plus de soixante-quatre instructions. Même si *435.gromacs* et *470.lbm* ne présentent pas un fort potentiel pour SYRANT parce qu'ils ne souffrent pas de beaucoup de mauvaises prédictions, *473.astar* est le programme de test avec le plus fort taux de mauvaises prédictions de notre jeu de tests. Donc, même si ce programme de test présente un fort potentiel pour SYRANT, il sera amoindri par la part importante des reconvergences qui arrivent après un nombre trop grand d'instructions.

3.5.5 Résultats de SYRANT et de la prédiction SBL

La figure 3.8 illustre nos expérimentations basées sur un pipeline très puissant capable de lancer huit instructions par cycle, avec 1024 entrées de ROB, 1024 registres physiques entiers,

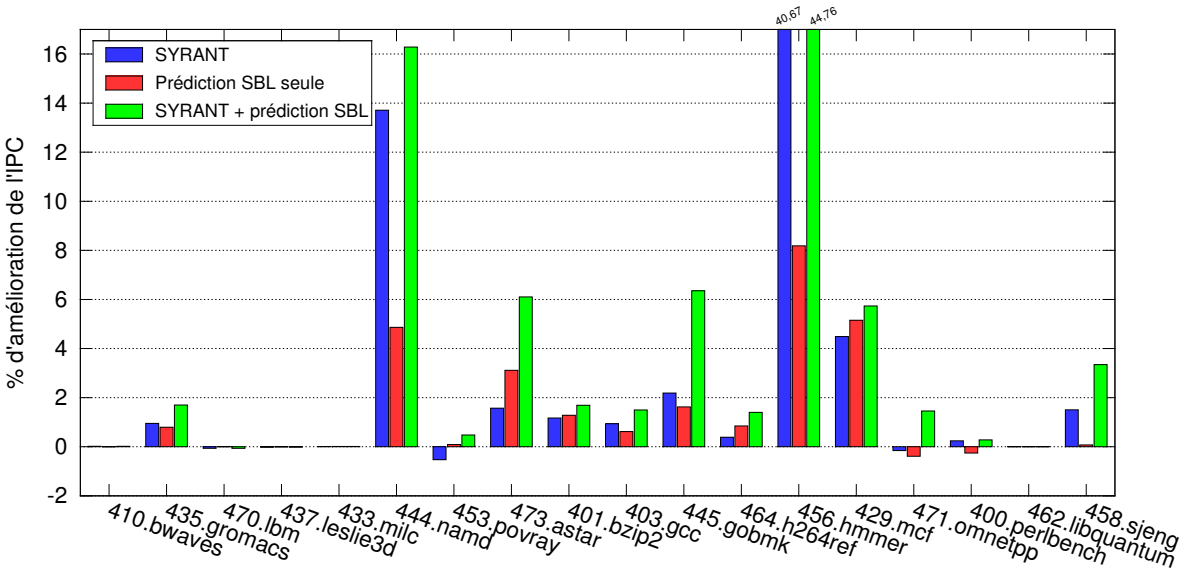


Figure *3.8* – *Amélioration de l'IPC pour SYRANT, la prédiction SBL et la combinaison des deux (SYRANT + prédiction SBL) par rapport à la configuration BASE.*

1024 registres physiques flottants et 512 entrées de LSQ. Les performances sont représentées par l'accélération de l'exécution par rapport à une configuration de base sans SYRANT.

Les résultats pour SYRANT ont été obtenus pour une combinaison de filtres d'insertion des vides qui produit les meilleurs résultats en moyenne. Lors de la correction, l'insertion de vides est seulement limitée par la taille des vides à insérer, c'est-à-dire que si la taille du vide est supérieure à une limite, trente-deux dans notre cas, le vide n'est pas inséré. Au moment où le branchement est enregistré dans le ROB, des vides sont insérés si l'une des conditions suivantes est vérifiée :

- le compteur de stabilité associé au branchement a atteint son seuil (deux dans notre cas) et la taille du vide à insérer est inférieure à quatre ;
- le compteur de stabilité associé au branchement a atteint son seuil (deux dans notre cas), la confiance en la prédiction du branchement n'est pas forte et la taille du vide à insérer est inférieure à seize.

Comme souligné dans la partie 3.3, le mécanisme matériel ABL/SBL peut être utilisé pour améliorer la précision de la prédiction de branchements en exploitant les branchements déjà exécutés après celui détecté comme point de reconvergence.

La table 3.2 montre l'amélioration obtenue sur la précision de la prédiction par l'utilisation de la prédiction SBL couplée au prédicteur TAGE. Cette amélioration est significative pour quelques programmes de test et conduit à des gains de performance (figure 3.8). Par exemple, *456.hmmer*, *429.mcf*, *473.astar* et *444.namd* voient leur taux de mauvaises prédictions se réduire de manière significative et profitent d'une amélioration visible de leurs performances. Au contraire, *458.sjeng* et *471.omnetpp* ne gagnent rien avec la prédiction SBL.

La figure 3.8 présente aussi les résultats de la combinaison du mécanisme SYRANT avec

la prédiction SBL (barre *SYRANT + SBL prédiction*). Pour quelques programmes de test (*444.namd*, *473.astar* et *456.hmmer*), les bénéfices de SYRANT et de la prédiction SBL semblent presque se cumuler, tandis que pour d'autres (*429.mcf* et *401.bzip2*) les gains de performance se cumulent moins. Toutefois, il semble que pour tous les programmes de test, la combinaison de SYRANT et de la prédiction SBL apporte toujours plus de gains de performance que l'un des deux mécanismes seul.

3.5.6 Commentaires sur les résultats

Comme SYRANT essaie de réduire les pertes liées aux mauvaises prédictions de branchements, ce mécanisme est seulement utile lorsque de telles mauvaises prédictions existent. Ainsi, cela explique les faibles gains de performance observés sur certains programmes de test comme *410.bwaves*, *433.milc* et *462.libquantum*.

Le même argument est valable pour les phases d'exécution du programme. Le taux de mauvaises prédictions d'un programme de test est la moyenne pondérée du taux de chacune de ses tranches *Simpoint*. De même que les phases du programme peuvent être différentes, leurs taux de mauvaises prédictions peuvent l'être aussi. En conséquence, les tranches *Simpoint* de certains programmes de test présentent de grandes différences en termes de taux de mauvaises prédictions. Comme les bénéfices de SYRANT sont fortement corrélés au taux de mauvaises prédictions, SYRANT n'est efficace que sur les tranches *Simpoint* avec un taux élevé de mauvaises prédictions. Ce comportement est mis en évidence sur la figure 3.9 pour le programme de test *473.astar*.

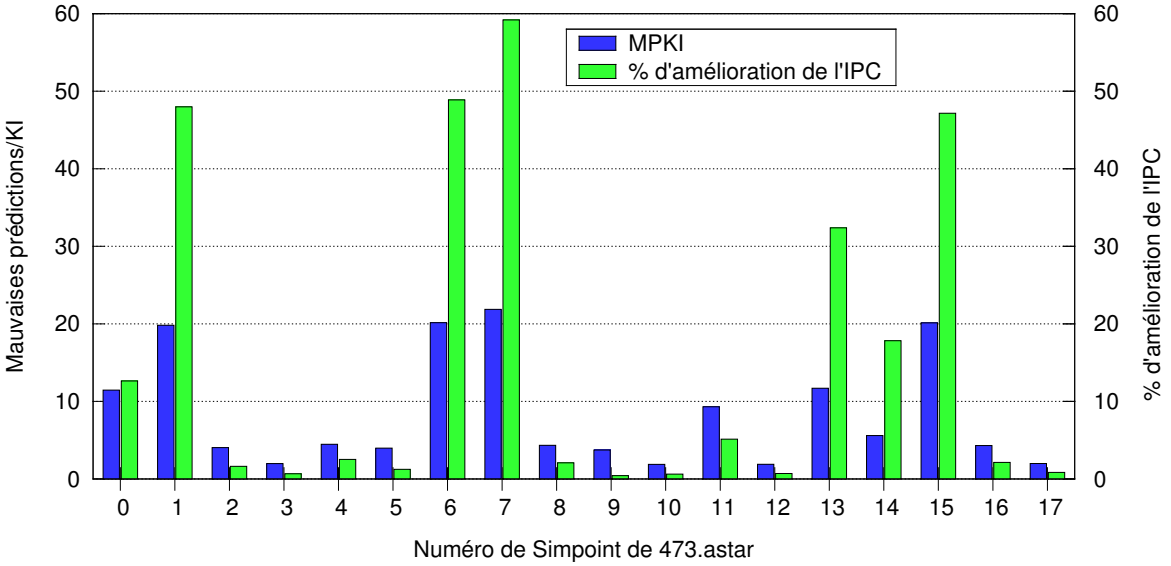


Figure *3.9* – *Taux de mauvaises prédictions et amélioration de l'IPC avec SYRANT par rapport à BASE, sur le programme de test* 473.astar, *par tranche* Simpoint.

Cela signifie que l'amélioration des performances induite par SYRANT est principalement effective pendant les phases d'exécution pour lesquelles les performances ont réellement besoin d'être améliorées. Ainsi, même si les performances de SYRANT ne sont pas élevées en moyenne

sur la plupart des programmes de test, SYRANT est capable de réduire efficacement les pertes de performance pendant les phases d'exécution avec un taux élevé de mauvaises prédictions de branchements.

SYRANT est plus efficace lorsque les deux chemins reconvergents sont suffisamment courts. Par exemple, SYRANT se comporte bien sur le programme de test *444.namd* bien que celui-ci ne possède pas un taux de mauvaises prédictions élevé. En effet, le taux de détection de reconvergence est très haut pour *444.namd*. Et en moyenne, la taille des chemins reconvergents est courte (moins de dix-huit instructions). SYRANT est donc capable d'exploiter la grande majorité des cas de reconvergence pour améliorer les performances.

L'effet opposé peut être observé sur le programme de test *473.astar* pour lequel de nombreux chemins reconvergents font plus de soixante-quatre instructions de long. Un grand nombre d'instructions dans le chemin reconvergent réduit les chances d'avoir des instructions ICID après le point de reconvergence, ainsi que les chances qu'elles soient exécutées avant que le branchement ne soit résolu. De plus, lorsque la taille des deux chemins n'est pas bien équilibrée, l'insertion des vides provoque un gaspillage significatif des ressources. Bien que la figure 3.9 ne le montre pas, nous avons vérifié qu'il y a une forte disparité dans la taille des chemins reconvergents entre les différentes tranches *Simpoint* du programme de test *473.astar*. Les améliorations de performance sont meilleures pour les tranches *Simpoint* pour lesquelles les chemins reconvergents sont plus courts.

3.5.7 Changement de la taille du ROB

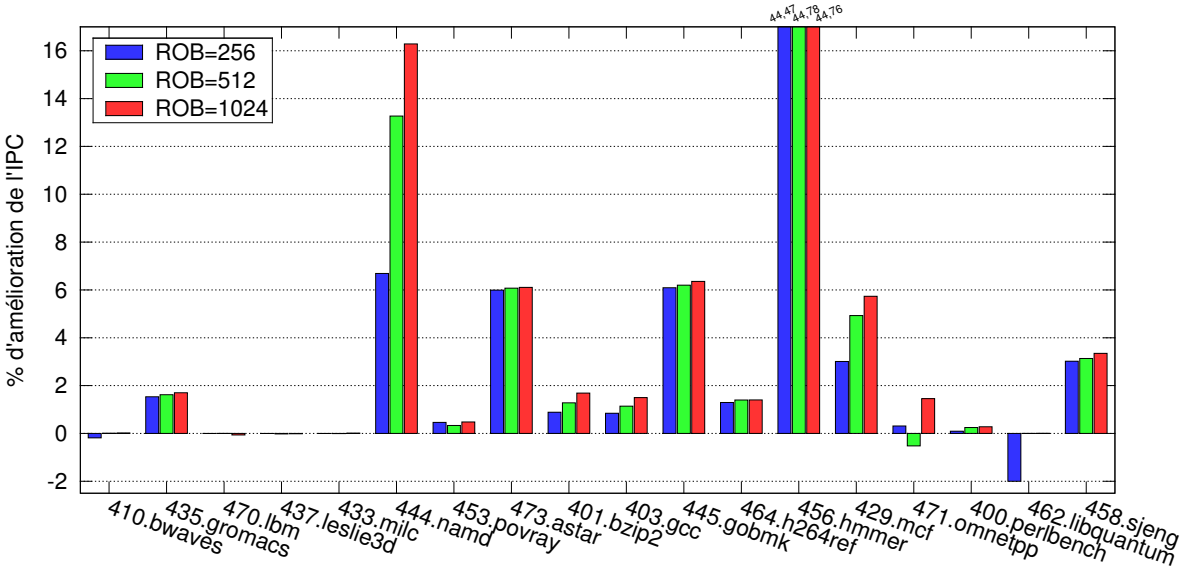


Figure 3.10 – *Variation des résultats en fonction de la taille du ROB.*

La figure 3.10 illustre les simulations effectuées avec des tailles de ROB de respectivement 256, 512 et 1024 entrées en utilisant le mécanisme *SYRANT + prédiction SBL*. À part pour le programme de test *471.omnetpp*, les performances provenant de l'utilisation de SYRANT

augmentent avec la taille du ROB. De plus, un ROB de 512 entrées semble suffisant pour observer des résultats significatifs en utilisant SYRANT. Même pour un ROB de 256 entrées, des gains en performances sont observés pour la plupart des programmes de test. Pour les autres programmes de test, des filtres plus puissants seraient requis.

3.5.8 Limitation du nombre d'instructions lancées par cycle

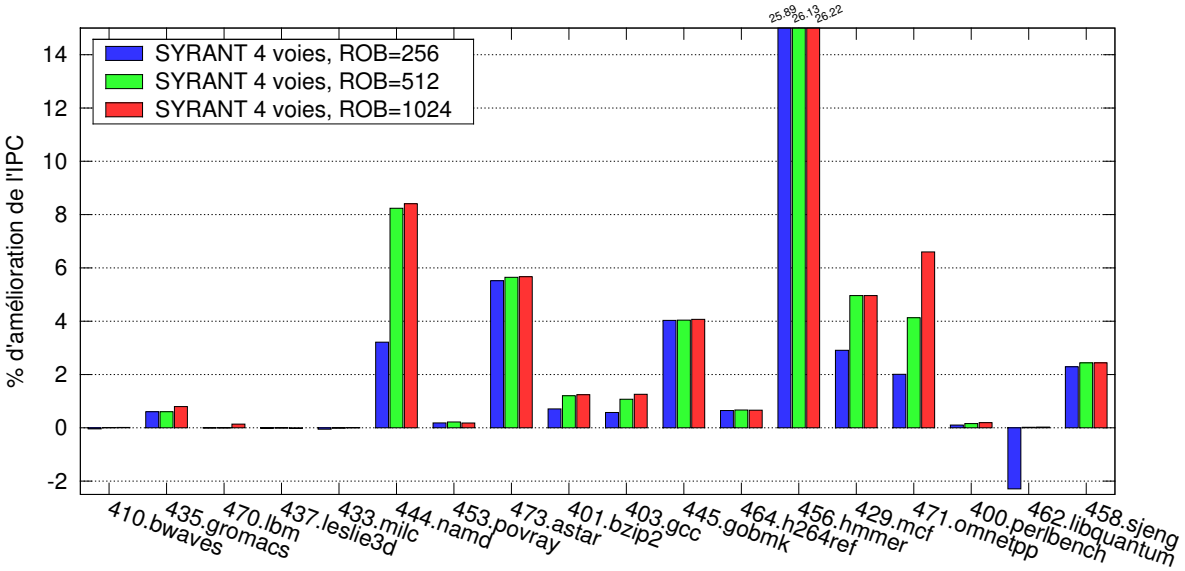


Figure *3.11* – *Amélioration de l'IPC avec SYRANT par rapport à la configuration BASE sur un processeur superscalaire à 4 voies, en fonction de la taille du ROB.*

Nous avons simulé le comportement de SYRANT sur un processeur superscalaire pouvant traiter quatre instructions par cycle et utilisant la moitié des ressources d'exécution de la configuration puissante. Sur la figure 3.11, nous n'avons illustré que les résultats pour la configuration *SYRANT + prédiction SBL* (appelée seulement SYRANT). Les mêmes programmes de test que pour la configuration puissante présentent des gains de performance. La figure 3.11 montre que les gains de performance suivent le même tendances que pour la configuration à huit voies lorsque la taille du ROB est réduite. Ainsi, sur une configuration comparable avec un processeur actuel (quatre voies avec un ROB de 256 entrées), SYRANT est capable d'obtenir des gains de performance pour la plupart des programmes de test avec un taux de mauvaises prédictions significatif.

3.6 Conclusion

Pour atteindre la performance ultime sur les codes séquentiels, l'exploitation de la reconvergence du flot de contrôle est intéressante puisqu'elle permet la réutilisation d'instructions déjà exécutées. Cependant, les propositions précédentes décrites dans la littérature se basent sur des mécanismes matériels complexes [20, 12, 24, 2] nécessitant d'importantes modifications dans le pipeline d'exécution d'un processeur superscalaire.

Cette complexité matérielle pourrait empêcher les architectes de processeurs d'implémenter un mécanisme exploitant la reconvergence du flot de contrôle. Nous avons introduit une nouvelle proposition appelée SYRANT, *SYmmetric Resource Allocation on Not-taken and Taken paths*. Son implémentation n'implique pas de modifications majeures sur le cœur d'exécution d'un processeur superscalaire. SYRANT est conçu pour allouer les mêmes ressources du cœur d'exécution dans le désordre aux mêmes instructions présentes après le point de reconvergence sur le chemin pris et le chemin non pris. Ainsi, il n'est plus nécessaire d'avoir des mouvements de données complexes pour exploiter l'indépendance de contrôle. Et de cette manière, il devient trivial de réassocier le résultat d'une instruction indépendante du contrôle I déjà exécutée sur le mauvais chemin à la version de cette même instruction I sur le bon chemin.

L'allocation symétrique des ressources sur les deux chemins s'opère à travers l'insertion de vides dans les structures du moteur d'exécution dans le désordre (*free list* des registres physiques, ROB, LSQ). Cela permet de s'assurer que les mêmes ressources sont utilisées sur les deux chemins. Nous avons présenté des mécanismes simples pour détecter la reconvergence et pour respecter les dépendances de données, tout en préservant les instructions indépendantes du contrôle et des données déjà exécutées.

Les simulations présentées dans ce document indiquent que, si l'insertion des vides est correctement filtrée, l'utilisation de SYRANT contribue notamment à l'amélioration des performances sur la plupart des programmes avec un taux élevé de mauvaises prédictions de branchements.

Durant le processus de création de SYRANT, nous avons dû inventer un nouveau mécanisme efficace pour détecter les points de reconvergence. Notre mécanisme ABL/SBL apparaît comme une importante contribution pour améliorer les performances des processeurs superscalaires tout en ayant un impact très limité sur la structure du processeur. L'ABL/SBL permet d'observer la reconvergence des branchements et de sauvegarder les résultats des branchements exécutés sur le mauvais chemin. Cette information est utilisée pour améliorer la prédiction de branchements après une mauvaise prédiction de branchement. L'ajout de l'ABL/SBL à un pipeline conventionnel n'est pas intrusif mais permettrait d'améliorer significativement la précision de la prédiction de branchements pour certains programmes de test avec des branchements difficiles à prédire.

Chapitre 4

Instructions prédiquées et *if-conversion*

Dans ce chapitre, nous nous intéresserons aux instructions prédiquées, à leur définition et leurs utilisations. Nous nous concentrerons plus particulièrement sur le mécanisme de *if-conversion* qui permet de transformer des dépendances de contrôle en dépendances de données et ainsi éliminer des branchements. Puis nous reviendrons sur les divers travaux liés aux instructions prédiquées, en particulier ceux traitant du problème de leur exécution sur des processeurs à exécution dans le désordre.

4.1 Instructions prédiquées

4.1.1 Définition

Une instruction prédiquée est une instruction possédant une opérande supplémentaire appelée *prédicat* qui conditionne l'exécution de l'instruction. Le prédicat est une fonction booléenne, souvent simple, dont l'évaluation doit être fait dynamiquement à l'exécution. Si le prédicat est évalué à vrai, l'instruction est exécutée comme n'importe quelle autre instruction et son résultat modifie l'état architectural du processeur. Si le prédicat est évalué à faux, l'instruction est convertie en no-op et elle n'influence pas l'état du processeur.

4.1.2 Utilisations

Les instructions prédiquées permettent l'exécution conditionnelle d'instructions sans avoir besoin de rediriger le flot d'instructions à l'aide de branchements. Elles permettent donc cette exécution conditionnelle sans avoir de dépendances de contrôle mais uniquement des dépendances de données. L'utilisation des instructions prédiquées expose donc naturellement plus de parallélisme d'instructions, ce qui tend à augmenter les performances sur les processeurs superscalaires. En effet, comme on l'a vu précédemment, les branchements sont coûteux en terme de performance dans les architectures pipelinées. En évitant de les utiliser à l'aide des instructions prédiquées, on diminue donc naturellement le problème posé par les branchements.

Le mécanisme de *if-conversion* [4] permet de transformer une structure de code conditionnelle basée sur des instructions de branchement en une structure basée sur des instructions prédiquées. Cette transformation consiste principalement à enlever les instructions de branchement, à remplacer les instructions contrôlées par ces branchements par leur équivalent prédiqué et à éventuellement ajouter des instructions pour calculer le ou les prédicats nécessaires. Un exemple de cette transformation est donné dans la figure 4.1. Cet exemple illustre la transformation d'une structure conditionnelle *if-then-else*. Les deux chemins définis par cette structure sont fusionnés en un seul, les instructions du chemin pris étant prédiquées avec le prédicat p et les instructions du chemin non pris étant prédiquées avec le prédicat inverse $\overline{p}$. Le principal défaut de cette technique est qu'elle oblige le processeur à traiter les deux chemins puisque ceux-ci sont mélangés en un seul. Il en résulte une augmentation significative du nombre total d'instructions traitées.

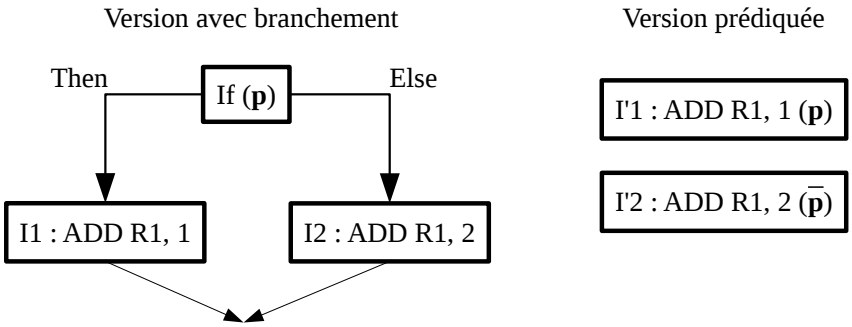


Figure *4.1* – *Exemple de* if-conversion *: une structure conditionnelle* if-then-else *est convertie en une structure utilisant des instructions prédiquées.*

Les instructions prédiquées peuvent être utilisées pour d'autres cas, comme par exemple le pipeline logiciel [43, 31, 47, 3]. Le pipeline logiciel consiste à transformer une structure de code de boucle pour que différentes itérations de la boucle puissent être exécutées de manière pipelinée, c'est-à-dire que l'exécution d'une itération commence avant que l'exécution de l'itération précédente ne soit terminée. De cette manière, on parallélise l'exécution du corps de la boucle, de la même manière que le pipeline du processeur parallélise l'exécution des instructions. Dans ce contexte, les instructions prédiquées sont utilisées pour éviter de devoir ajouter spécialement des instructions pour le prologue et l'épilogue de la boucle transformée pour le pipeline logiciel [63, 18]. L'architecture EPIC (*Explicitely Parallel Instruction Computing* [48]) Itanium [25] implémente un support matériel pour le pipeline logiciel.

Comme les instructions prédiquées permettent d'avoir des structures de codes conditionnelles qui utilisent moins d'instructions qu'une version avec des branchements, Cheung et al. [13] se sont intéressés à la possibilité de produire du code plus compact en utilisant des instructions prédiquées. Ils se sont en particulier penchés sur l'abstraction procédurale qui consiste à remplacer plusieurs séquences de codes identiques par un appel à une seule fonction représentative. À l'aide d'instructions prédiquées, ils parviennent à faire l'abstraction procédurale d'un plus grand ensemble de codes.

4.1.3 Prédication partielle et prédication totale

L'implémentation des instructions prédiquées dans un jeu d'instructions existant est très difficile. En effet, il faut suffisamment de place dans l'espace de codage pour rajouter l'ensemble des instructions prédiquées que l'on veut intégrer dans le jeu d'instructions. En conséquence, on constate que l'intégration des instructions prédiquées est partielle dans de nombreux jeux d'instructions, c'est-à-dire que seules quelques instructions ont une version prédiquée. Par exemple, dans le jeu d'instructions *Alpha* [28], la seule instruction prédiquée est le *conditional move* (CMOV) qui permet de copier, de manière conditionnelle, une valeur d'un registre à un autre. La plupart des jeux d'instructions existants qui n'ont pas été conçus à l'origine pour être prédiqués ont une prédication partielle voire inexistante.

Il existe cependant des jeux d'instructions qui ont été pensés dès le départ pour être prédiqués comme le jeu d'instruction IA-64 de l'architecture *Itanium* [25] ou encore le jeu d'instruction *ARM* [6]. Pour ces jeux d'instructions, on parle de prédication totale car la (quasi)-totalité des instructions peuvent être prédiquées. Les avantages de la prédication totale sont une plus grande souplesse d'utilisation des instructions prédiquées et une plus grande efficacité puisque le programmeur (ou plutôt le compilateur dans la plupart des cas) n'est pas limité dans son utilisation de la prédication. En l'occurrence, un code utilisant la prédication totale utilise moins d'instructions qu'un code sémantiquement équivalent utilisant la prédication partielle.

Mahlke et al. [35] ont étudié les bénéfices liés à la prédication partielle et à la prédication totale. Leur étude conclue que la prédication partielle suffit à implémenter la transformation de *if-conversion*. Par contre, ils observent que l'utilisation de la prédication totale est bien plus intéressante et qu'elle permet une exécution plus efficace et parallèle.

4.1.4 Implémentation matérielle

Le traitement des instructions prédiquées requiert un support particulier de la part du processeur. Cela pose différents problèmes que nous allons détailler.

Codage des instructions prédiquées

Le prédicat est implémenté comme une opérande source supplémentaire, ce qui complique la plupart des architectures qui ne sont pas prévues pour avoir cette opérande en plus. Dans la plupart des jeux d'instructions existants, il n'est pas possible de rajouter une opérande supplémentaire dans le codage des instructions. Cela explique que beaucoup d'architectures ne proposent que le *conditional move* en tant qu'instruction prédiquée, puisque l'instruction *move* normale possède une opérande de moins que les autres instructions et il est donc aisé d'utiliser cette opérande pour le prédicat.

Vérification de la valeur du prédicat

La vérification de la valeur du prédicat peut se faire à différents étages. Plus elle est faite tôt dans le pipeline, plus cela peut entraîner des blocages du pipeline, puisque les données nécessaires

au calcul du prédicat ne sont pas forcément prêtes au bon moment. Plus elle est faite tard et plus des ressources sont gaspillées par les instructions avec un prédicat faux.

Registres de prédicats

Selon les architectures, des registres spéciaux peuvent être utilisés pour enregistrer la valeur du prédicat. Contrairement à un branchement conditionnel pour lequel la valeur servant au test ne sert que pour le branchement, ces registres spéciaux servent pour toutes les instructions prédiquées utilisant le même prédicat. Leur durée de vie, c'est-à-dire la quantité de cycles pendant laquelle une même valeur et donc un même registre est utilisé, est donc plus longue que dans le cas du branchement, ce qui peut poser un problème de ressources et donc engendrer un blocage du pipeline.

Le problème des définitions multiples

La prise en charge des instructions prédiquées dans un processeur à exécution dans le désordre pose un problème supplémentaire. Le mécanisme de renommage des registres impose qu'à tout moment de l'exécution, un seul registre physique contienne la valeur courante d'un registre architectural. Ce lien est conservé dans la *mapping table* et permet de renommer les opérandes sources des instructions qui entrent dans le pipeline. Pour une instruction prédiquée, tant que la valeur de son prédicat n'est pas connu, on ne sait pas si elle écrira dans son registre destination ou non. Un registre architectural qui est le registre destination d'une instruction prédiquée peut donc être associé au registre physique attribué à l'instruction prédiquée en question si le prédicat de celle-ci est vrai ou au registre physique précédent si le prédicat est faux. Ce problème est illustré sur la figure 4.2. Comme il n'est pas toujours possible de connaître la valeur du prédicat au moment du renommage et que bloquer le pipeline pour attendre que cette valeur soit disponible serait trop coûteux, il faut une solution pour résoudre ce problème. Nous reviendrons dans la partie 4.2.3 de ce chapitre sur les diverses solutions proposées dans la littérature.

4.1.5 Spécificités de la prédication dans le jeu d'instructions ARM

Tous les travaux récents sur les instructions prédiquées utilisent le jeu d'instructions IA-64 de l'architecture *Itanium* [25]. Pour ce jeu d'instructions, la valeur du prédicat des instructions prédiquées est contenue dans des registres spéciaux, les registres à prédicats.

Dans ce document, nous avons pris comme base de travail le jeu d'instructions totalement prédiqué ARM. Ce choix se base principalement sur le fait que le jeu d'instructions ARM est le jeu totalement prédiqué le plus répandu à l'heure actuelle.

Dans le modèle de prédication utilisé pour le jeu d'instructions ARM, le prédicat des instructions est calculé à partir des valeurs binaires de quatre drapeaux : le drapeau Negatif (N), le drapeau Zero (Z), le drapeau de retenue (C), et le drapeau de débordement (V). Une instruction qui s'exécute peut éventuellement provoquer la modification des valeurs binaires des drapeaux en fonction du résultat de son exécution. Les drapeaux ne sont donc modifiés que par certaines instructions, en particulier les instructions de comparaisons et certaines instructions arithmétiques

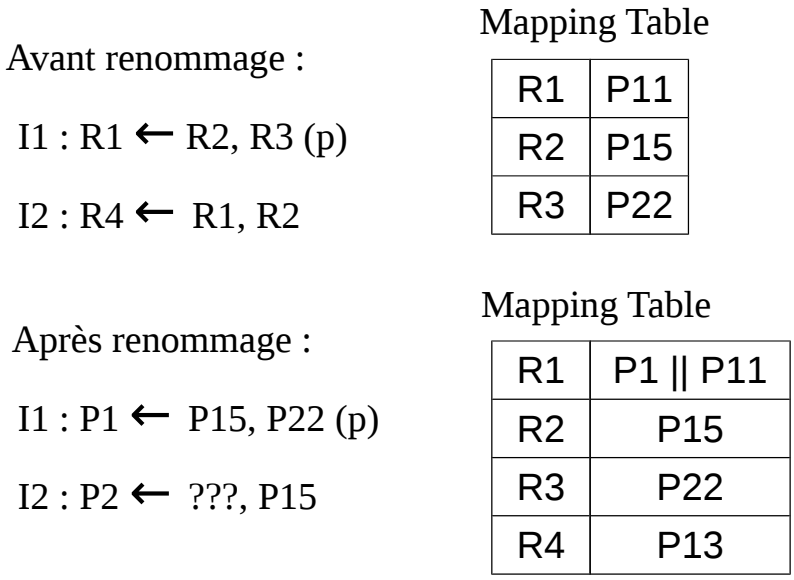


Figure *4.2 – Illustration du problème des définitions multiples. L'instruction $I1$ définie de manière conditionnelle la valeur du registre architectural $R1$ dans le registre physique $P1$. Cependant, comme le prédicat de $I1$ n'est pas connu au moment du renommage, $P1$ ne peut être associé de manière sûre à $R1$, puisque si le prédicat de $I1$ est faux, c'est dans $P11$ qu'est la vrai valeur de $R1$. L'instruction $I2$, qui lit dans $R1$, devra donc lire la valeur de son opérande soit dans $P1$ soit dans $P11$. Elle ne peut donc pas être renommée correctement.*

particulières [6]. Le drapeau N est mis à 1 si le résultat est négatif et à 0 s'il est positif ou nul. Le drapeau Z est mis à 1 si le résultat est nul et à 0 sinon. Le drapeau C est mis à 1 si le résultat est accompagnée d'une retenue (comme pour une addition qui donne un résultat trop grand) et à 0 sinon. Le drapeau V est mis à 1 si le calcul a provoqué un débordement, c'est-à-dire que la valeur calculée est trop grande pour être enregistrée dans le registre destination.

La table 4.1 résume les différents prédicats possibles dans le jeu d'instructions ARM. Ainsi, la valeur de ces prédicats n'est pas directement lue dans un registre. Il est nécessaire d'évaluer la formule logique associée au prédicat pour en connaître la valeur.

4.2 Travaux traitant des instructions prédiquées

Il existe de nombreux travaux traitant des instructions prédiquées. Certains cherchent à montrer l'intérêt d'utiliser les instructions prédiquées et la *if-conversion* pour réduire le nombre de branchements, d'autres s'intéressent à la relation entre les instructions prédiquées et la prédiction de branchements. Nous reviendrons enfin sur les travaux proposés pour résoudre le problème des définitions multiples.

Nom du prédicat	Formule logique
EQ	Z == 1
NE	Z == 0
CS	C == 1
CC	C == 0
MI	N == 1
PL	N == 0
VS	V == 1
VC	V == 0
HI	(C == 1) && (Z == 0)
LS	(C == 0) || (Z == 1)
GE	N == V
LT	N != V
GT	(Z == 0) && (N == V)
LE	(Z == 1) || (N != V)
AL	(Toujours vrai)

Table *4.1* – *Les différents prédicats possibles pour une instruction du jeu d'instructions ARM.*

4.2.1 Bénéfices de la *if-conversion*

Pnevmatikatos et Sohi [39] ont conduit une étude détaillée sur l'intérêt des instructions prédiquées et de la transformation de *if-conversion*. Le bénéfice direct de la transformation de *if-conversion* est l'élimination de certains branchements. Ils ont montré que cela offre plusieurs avantages. Tout d'abord, la taille des blocs de base est augmentée, jusqu'à 52% en moyenne. Cette augmentation offre au compilateur de plus grandes opportunités pour appliquer des optimisations. Cela permet aussi d'agrandir la taille de la fenêtre d'instructions dynamique, c'est-à-dire le nombre d'instructions qui entrent dans le pipeline entre deux vidages dus à une mauvaise prédiction. Pour les programmes de test de leur étude, cette fenêtre d'instructions peut aller jusqu'à 258 instructions, là où elle n'est que de 156 instructions sans utilisation de la prédication. Le désavantage est une augmentation conséquente du nombre d'instructions inutiles qui sont traitées par le pipeline. Ils ont constaté que le processeur doit traiter 33% plus d'instructions en moyenne. Ces résultats sont obtenus avec la prédication totale. Si seule une prédication partielle est utilisée, pour laquelle seuls les blocs de base ne contenant pas d'instructions mémoires sont prédiqués, le nombre d'instructions traitées n'augmente que de 8%. Par contre, la taille de la fenêtre dynamique ne va que jusqu'à 184 instructions en moyenne. Ils ont aussi constaté que la *if-conversion* ne transformait pas forcément des branchements qui étaient difficiles à prédire à la base.

Pour pallier ce problème, Mahlke et al. [34] ont travaillé sur une technique de sélection des branchements à convertir basée sur l'étude de leur comportement et sur leur propension à être mal prédit. En utilisant une structure de code qu'ils appellent un *hyperblock* [36], leur compilateur est capable de produire du code où les branchements avec un comportement difficilement prédictible sont éliminés pour être remplacés par des instructions prédiquées. Les branchements dont le

comportement est facilement prédictible ne sont pas touchés. Leurs tests montrent que leur technique permet de réduire le nombre de branchements dynamiques de 27% en moyenne, ce qui réduit le taux de mauvaises prédictions de 20%. Ils ont aussi constaté une augmentation du parallélisme d'instructions puisque le nombre d'instructions entre chaque branchement passe de 3,7 à 6,1 instructions en moyenne.

Chang et al. ont également étudié un mécanisme de sélection des branchements à convertir basé sur leur étude comportementale [11]. Ils parviennent à des conclusions similaires à l'étude de Mahlke et al. en ce qui concerne la réduction du nombre de mauvaises prédictions. Ils montrent aussi que l'étude comportementale des branchements est un procédé efficace de sélection puisque les branchements ainsi sélectionnés sont responsables, en moyenne, pour 73% du total des mauvaises prédictions.

August et al. ont proposé une structure de travail pour la compilation de programmes ciblant les architectures avec des instructions prédiquées [7]. Pour cela, ils se basent aussi sur la structure d'*hyperblock* [36] en appliquant la transformation de *if-conversion* de manière très poussée au plus tôt dans la compilation. Cela permet au compilateur de profiter pleinement de la représentation prédiquée pour appliquer des optimisations favorisant le parallélisme d'instructions et des transformations du flot de contrôle. Plus tard dans le processus de compilation, au moment où le compilateur a suffisamment d'informations pour juger de la pertinence de la *if-conversion*, certaines *if-conversion* sont retransformés en branchements à l'aide de l'inversion de *if-conversion* [64]. Cela permet d'annuler certaines *if-conversion* qui ont des effets négatifs sur les performances. Leurs résultats montrent que leur structure de travail est efficace et permet de réduire les pertes que l'on observe parfois pour certains *hyperblocks* qui sont créés de manière trop poussée.

4.2.2 Interaction des instructions prédiquées et de la prédiction de branchements

Comme la prédiction de branchements se base sur la corrélation qui existe entre différents branchements pour établir sa prédiction, l'élimination des certains branchements par la *if-conversion* peut réduire cette corrélation et influencer de manière négative la qualité de la prédiction de branchements. Certaines études se sont intéressées à cet effet et aux moyens de l'atténuer.

Simon et al. [55] ont proposé un mécanisme dans lequel les informations liées aux prédicats sont incorporées au prédicteur de branchements. Cela leur permet d'améliorer la précision du prédicteur sur les branchements contenus dans une région du code qui est prédiquée en associant ce prédicat au branchement. La première amélioration, qu'ils appellent *Squash False Path*, permet de prédire avec une précision de 100% les branchements dont le prédicat associé a été calculé avant que le branchement n'entre dans le pipeline. La deuxième amélioration, qu'ils appellent *Predicate Global Update Branch Predictor*, ajoute la valeur des prédicats dans l'historique global des branchements. Cela permet de retrouver une partie de la corrélation perdue par la *if-conversion*. Cependant, comme les valeurs des prédicats ne sont pas enregistrées dans l'historique au même moment que les directions des branchements, cela pose un problème de cohérence

de l'historique. Pour résoudre cette difficulté, ils ont proposé un mécanisme appelé *Deterministic Predicate Update Table* qui garantit que les informations sont enregistrées dans l'historique dans l'ordre du programme et donc à chaque fois dans le même ordre. Leurs résultats montrent que le mécanisme simple de *Squash False Path* est suffisant pour observer des améliorations de performance puisque les mauvaises prédictions sont réduites de 0.5% à 4.3% selon les programmes de test.

Kim et al. [30] ont travaillé sur une technique de compilation liée à la *if-conversion* qui transforme les branchements sans les éliminer. À la place, ils sont convertis en un type de branchements spécial qu'ils appellent *wish-branch*. Le code obtenu est donc le code prédiqué avec les branchements toujours présents mais transformés en *wish-branch*. Le code prédiqué et le code non prédiqué est donc présent dans le programme. Au moment de l'exécution, le processeur choisit s'il prédit et exécute les branchements et le code non prédiqué ou s'il exécute le code prédiqué sans tenir compte des branchements. Cette décision est prise en fonction de l'estimation de la confiance en la prédiction donnée par le prédicteur de branchements. Le but des *wish-branch* est de ne pas perdre la corrélation qui existe entre les différents branchements tout en permettant d'utiliser la prédication pour les branchements trop difficile à prédire. Leurs résultats montrent que sur leur jeu de tests, leur technique permet de réduire le temps d'exécution de 14% comparé à un code avec des branchements normaux et de 13% par rapport à un code prédiqué offrant la meilleure performance.

Dans leur travail suivant [29], ils sont allés encore plus loin dans ce concept en prédiquant dynamiquement certains chemins reconvergents préalablement identifiés par le compilateur. Cela leur permet, comme avec les *wish-branch* de ne convertir que les branchements difficiles à prédire. Mais contrairement à leur précédent travail, cette conversion n'est pas limitée par le compilateur et un plus grand nombre de branchements peut être traité par ce mécanisme. Concrètement, lorsqu'un branchement qui peut être traité par cette technique (cette identification se fait par le compilateur) entre dans le pipeline, les deux chemins qu'il définit sont traités par le pipeline en même temps, un des chemins étant prédiqué avec le prédicat lié au branchement, l'autre avec son inverse. Ainsi, seul le bon chemin est finalement validé, sans pour autant payer le coup de la mauvaise prédiction si le branchement est mal prédit. Leur étude précise que leur mécanisme améliore les performances de 19,3% en moyenne par rapport à un processeur puissant avec un prédicteur de branchements de grande taille.

Quiñones et al. ont proposé un prédicteur qui ne prédit pas directement les branchements mais les résultats des instructions de test qui calculent les conditions des branchements [42]. Leur prédicteur n'est donc pas un prédicteur de branchements mais un prédicteur de prédicats. Cela permet de retrouver la corrélation perdue par la *if-conversion* qui est toujours présente dans la valeur des prédicats. Si l'instruction de test qui calcule la condition d'un branchement est exécuté suffisamment en avance et que sa valeur est ainsi calculée avant que le branchement n'entre dans le pipeline, cette valeur est directement utilisée pour la prédiction du branchement, ce qui permet d'avoir une prédiction sûre à 100% dans ce cas. Ces cas sont appelés des branchements résolus en avance.

4.2.3 Travaux portant sur le problème des définitions multiples

Les problèmes posés par l'implémentation matérielle des instructions prédiquées ont empêché l'adoption massive de celles-ci sur les architectures à exécution dans le désordre, en particulier le problème des définitions multiples. Plusieurs travaux se sont intéressés à ce problème et diverses solutions ont été proposées.

Pnevmatikatos et Sohi [39] ont été les premiers à mettre en lumière le problème des définitions multiples. Ils ont proposé une solution simple qui consiste à transformer les instructions prédiquées en *false predicate conditional move* (FPCMOV). Dans cette forme, une instruction prédiquée a une opérande implicite supplémentaire qui est le registre physique précédemment associé au registre architectural destination de l'instruction. Si le prédicat de l'instruction est vrai, alors la nouvelle valeur est écrite dans le nouveau registre physique associé à l'instruction. Si le prédicat est faux, l'ancienne valeur présente dans l'ancien registre physique est copiée dans le nouveau registre physique. De cette manière, du point de vue du renommage, il n'y a plus d'ambiguïté, la définition courante d'un registre architectural est donnée par la dernière instruction qui écrit dedans, qu'elle soit prédiquée ou non. C'est ensuite le mécanisme des FPCMOV qui se charge de transmettre la bonne valeur dans le registre physique en question. L'opération effectuée par ces FPCMOV est donc :

$$P_{nouveau} = \textbf{(prédicat) ?}\ Opération(Op1, Op2)\ \textbf{:}\ P_{ancien}$$

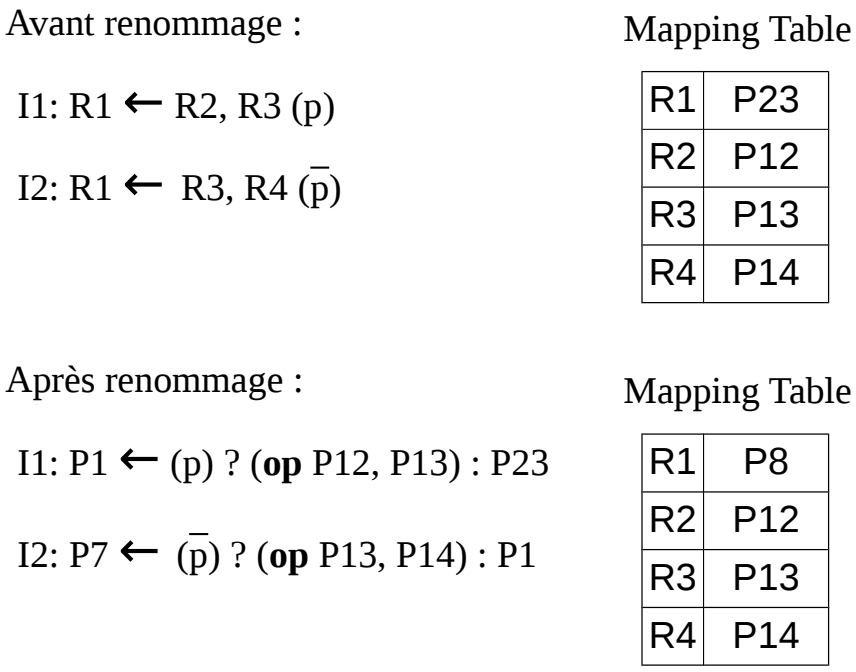


Figure *4.3 – Exemple de renommage en utilisant la solution des FPCMOV.*

La figure 4.3 illustre comment les instructions prédiquées sont transformées en FPCMOV au moment du renommage. Le principal désavantage de cette solution, aussi illustré sur cette

figure, est qu'elle force les instructions prédiquées ainsi converties à être exécutées de manière séquentielle. En effet, comme l'une des opérandes d'une instruction FPCMOV est le résultat de l'instruction qui définie précédemment le même registre architectural qu'elle, elle est forcée d'attendre que cette instruction précédente soit exécutée alors qu'il n'y a pas de dépendances sémantiques entre les deux instructions.

Une version plus simple de cette solution est utilisé dans les processeurs *Alpha* pour prendre en charge l'instruction *CMOV* [28]. Dans cette version, une instruction prédiquée est transformée en deux micro-instructions : la première effectue l'opération de l'instruction prédiquée originale, la seconde sélectionne si c'est la valeur calculée qui doit être écrite dans le registre destination ou alors la valeur précédente, selon la valeur du prédicat. L'opération effectuée par cette micro-instruction est donc :

$$P_{final} = \textbf{(prédicat) ?}\ P_{nouveau}\ \textbf{:}\ P_{ancien}$$

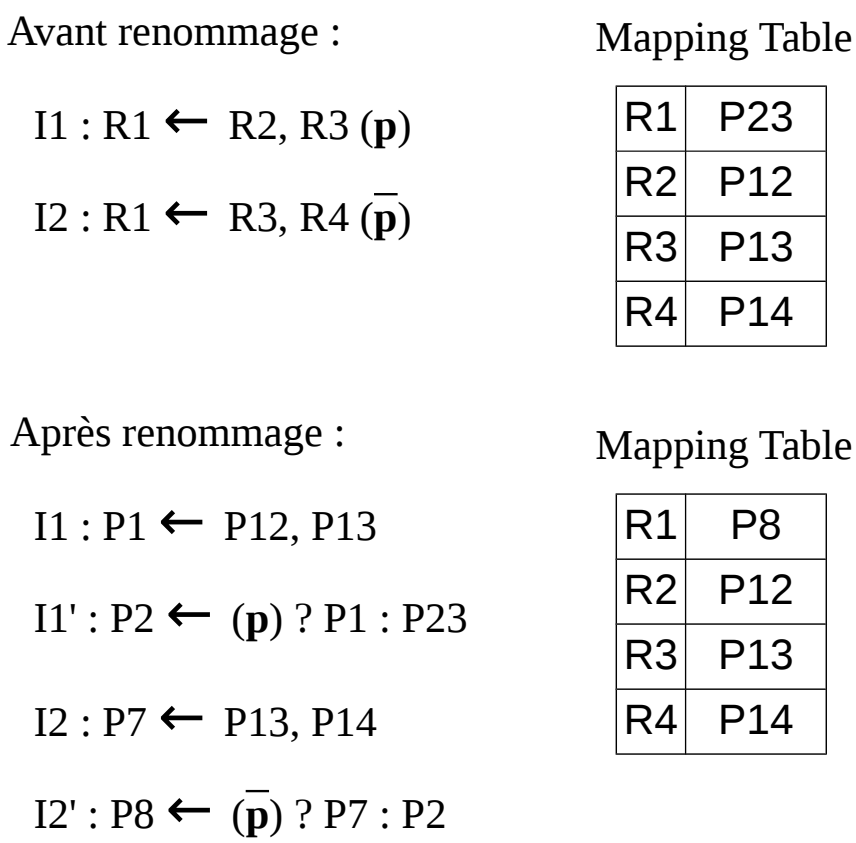


Figure *4.4 – Exemple de renommage en utilisant la solution des FPCMOV avec une implémentation plus simple au niveau matériel.*

Cette version est illustrée par la figure 4.4. L'avantage de cette solution est sa plus grande simplicité au niveau matériel, puisqu'elle n'oblige pas l'instruction à avoir une opérande implicite supplémentaire. Par contre, elle génère deux instructions pour chaque instruction prédiquée et chaque instruction prédiquée demandera donc un cycle supplémentaire pour être traitée. Et la

chaîne de dépendances artificielles existent encore. En effet, même si les instructions $I1$ et $I2$ sont exécutées au même cycle T, l'instruction supplémentaire $I'1$ ne peut s'exécuter qu'au cycle $T+1$ puisqu'elle doit au minimum attendre le résultat de $I1$ et l'instruction supplémentaire $I'2$ ne peut s'exécuter qu'au cycle $T+2$ puisqu'elle doit au minimum attendre le résultat de $I'1$. Dans la première version des FPCMOV, si l'instruction $I1$ est exécutée au cycle T, l'instruction $I2$ peut directement s'exécuter au cycle $T+1$.

Wang et al. [62] ont proposé de résoudre le problème des définitions multiples par une solution qui étend le principe de la deuxième version des FPCMOV. Lorsque nécessaire, une instruction spéciale appelée *select-μop* est insérée dans le flot d'instructions du pipeline. Cette instruction permet de sélectionner quelle est la bonne valeur pour un registre architectural défini par des instructions prédiquées. Elle joue donc le même rôle que la deuxième micro-instruction de la deuxième version des FPCMOV mais le nombre de valeurs parmi lesquelles elle fait une sélection peut être plus grand. L'idée du *select-μop* est dérivée de la fonction ϕ utilisée par les compilateurs dans la forme *static single assignment* (SSA) [17]. Pour former cette instruction spéciale, ils proposent d'ajouter des informations dans la *mapping table*. Pour chaque registre architectural, l'entrée de la table contient un lien vers chaque registre physique qui contient une définition prédiquée du registre en question ainsi que les prédicats associés. Lorsqu'une instruction est renommée, si pour une de ses opérandes sources l'entrée du registre architectural associé contient plusieurs définitions prédiquées, une instruction *select-μop* est générée avant l'instruction à renommer. Les différents registres physiques contenant les multiples définitions, accompagnés des prédicats associés, servent d'opérandes sources pour l'instruction *select-μop*. Un nouveau registre physique est associé à l'instruction *select-μop*, de telle sorte que celui-ci soit la seule version valide pour le registre architectural. L'instruction à renommer peut donc utiliser ce nouveau registre en tant qu'une de ses opérandes sources, levant l'ambiguïté des définitions multiples. Lorsque l'instruction *select-μop* est exécutée, la valeur d'une de ses opérandes est sélectionnée en fonction des différents prédicats et copiée dans le registre destination. Le principal rôle de l'instruction *select-μop* est donc de différer le problème posé par les définitions multiples de l'étage de renommage à celui de l'exécution.

Chuang et Calder ont proposé un mécanisme basé sur la prédiction de prédicats [15]. Pour éviter le problème des définitions multiples, la valeur du prédicat de chaque instruction prédiquée est prédite. Ainsi, au moment de renommer une instruction prédiquée, si son prédicat est prédit vrai alors elle est traitée comme une instruction normale et la *mapping table* est donc mise à jour avec son registre physique destination. Si le prédicat est prédit faux, l'instruction est transformée en no-op et la *mapping table* n'est pas mise à jour. Ainsi, il n'y a pas de problème de définitions multiples. Si un prédicat est mal prédit, il faut corriger l'exécution en vidant le pipeline. Pour éviter un coût en performance qui serait équivalent à une mauvaise prédiction de branchement et qui réduirait ainsi fortement l'intérêt d'avoir transformé les branchements en instructions prédiquées, Chuang et Calder ont proposé un mécanisme de rejeu sélectif des instructions. Ce mécanisme permet de ne pas vider le pipeline et de seulement réexécuter les instructions qui sont dépendantes du prédicat mal prédit et des instructions utilisant ce prédicat. Les autres instructions ne sont pas affectées par la correction et continuent à être exécutées normalement. Des étiquettes sont ajoutées dans l'*Issue Queue* pour pouvoir identifier les différents graphes

de dépendances. Lors de la correction d'une mauvaise prédiction de prédicat, les instructions prédiquées qui ont besoin d'être réexécutées sont converties en FPCMOV, ce qui induit une séquentialisation de la réexécution. Leurs résultats montrent que leur technique permet des gains de performance de 6,9% en moyenne par rapport à une version du programme sans prédication.

Quiñones et al. [41] ont aussi proposé un mécanisme basé sur la prédiction de prédicats. L'amélioration notable par rapport au travail de Chuang et Calder [15] est le mécanisme de sélection de l'utilisation de la prédiction. En effet, la prédiction n'est utilisée que quand la confiance que l'on a en elle est suffisamment forte. Dans le cas où la prédiction n'est pas utilisée, l'instruction prédiquée est simplement convertie en FPCMOV. L'évaluation de la confiance en la prédiction se fait à l'aide du prédicteur JRS [26] (voir la partie 2.3.4). La prédiction n'est utilisée que si le compteur atteint un certain seuil. Ce seuil est fixé de manière dynamique par un algorithme qui évalue la quantité de prédictions utilisées sur la dernière tranche d'exécution par rapport à la quantité de mauvaises prédictions. Le seuil est augmenté lorsque le nombre de mauvaises prédictions augmente et il diminue lorsque trop peu de prédictions sont utilisées. Contrairement à [15], aucun mécanisme n'est utilisé pour limiter le coût d'une mauvaise prédiction puisque la sélection de l'utilisation de la prédiction est là pour éviter un trop grand nombre d'utilisation de mauvaises prédictions. Les résultats montrent qu'en moyenne seul 16% des instructions prédiquées ne sont pas prédites. Les auteurs de l'étude observent un gain de performance de 17% en moyenne par rapport à une architecture où toutes les instructions sont converties en FPCMOV.

Chapitre 5

SPREPI : prédication et rejeu sélectif pour les instructions prédiquées

Ce chapitre présente les travaux de cette thèse concernant le traitement des instructions prédiquées dans un processeur superscalaire à exécution dans le désordre. Le mécanisme proposé utilise un prédicteur de prédicats dérivé du prédicteur de branchements TAGE [54] pour résoudre le problème des définitions multiples. Comme pour [41], un mécanisme de sélection de l'utilisation de la prédiction est utilisé. À l'instar de [15], un mécanisme de rejeu sélectif est mis en œuvre pour réduire le coup d'une mauvaise prédiction. Ce mécanisme reprend globalement les idées développées au chapitre 3.

5.1 Prédiction sélective de prédicats

La prédiction de prédicats est utilisée pour résoudre le problème des définitions multiples. Seules les instructions prédiquées qui sont prédites avec un prédicat vrai sont renommées. Celles qui sont prédites avec un prédicat faux sont transformées en no-op à l'étage de renommage. Ainsi, il y a une seule définition valide pour chaque registre architectural. La prédiction de prédicats peut se faire en utilisant les mêmes mécanismes que ceux utilisés pour la prédiction de branchements. Pour notre étude, nous avons dérivé notre prédicteur du prédicteur de branchements à historique global TAGE [54], considéré comme l'état de l'art.

La qualité et la précision de la prédiction donnée par un prédicteur dépendent fortement de la qualité du vecteur d'informations qu'il utilise. Dans le cadre d'un prédicteur à historique global, le vecteur d'informations est l'historique global. Nous avons travaillé avec deux types d'historique : celui des branchements et celui des branchements et prédicats, c'est-à-dire l'historique composé des directions des branchements et de la valeur des prédicats rencontrés.

5.1.1 Groupe prédiqué

La transformation de *if-conversion* tend à générer plusieurs instructions prédiquées qui utilisent le même prédicat ou son complément. Cela correspond au chemin pris et au chemin non

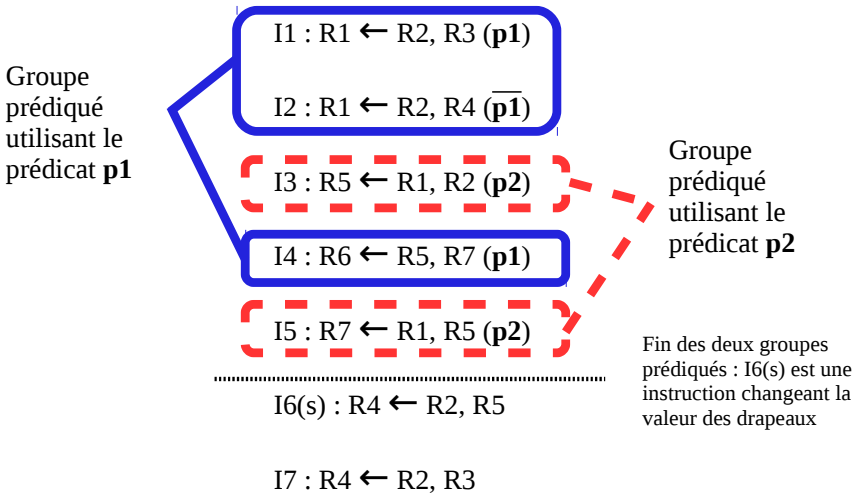


FIGURE 5.1 – *Exemple de deux groupes prédiqués. Chacune des instructions les composant utilise le même prédicat ou son complément. Ces deux groupes se terminent lorsqu'une instruction changeant la valeur des drapeaux est rencontrée.*

pris originel. Cela nous a conduit à introduire le concept de groupe prédiqué d'instructions : c'est le groupe d'instructions qui utilisent la même valeur de prédicat ou son complément. Un groupe prédiqué est associé à l'utilisation de la même occurrence de la valeur d'un prédicat. Les instructions qui composent un groupe prédiqué ne sont pas forcément contiguës dans le code puisque la transformation de *if-conversion* tend à organiser les instructions provenant du chemin pris, celles provenant du chemin non pris et celles provenant des deux chemins dans le même bloc de base. Comme notre étude cible le jeu d'instructions ARM, un groupe prédiqué commence à la première utilisation de la valeur d'un prédicat et finit lorsqu'une instruction modifiant la valeur des drapeaux est rencontrée.

La figure 5.1 est un exemple de deux groupes prédiqués. Une seule prédiction par groupe prédiqué est produite dans le début du pipeline, avant l'étage de renommage. Dans la suite, un groupe prédiqué sera représenté par sa première instruction.

Le prédicat a besoin d'être prédit pour la première instruction du groupe prédiqué et un mécanisme est nécessaire pour propager la valeur de cette prédiction à l'ensemble du groupe prédiqué au moment du renommage. Dans la suite de ce document, quand nous ferons référence à l'historique global branchements et prédicats, nous partirons du principe que la valeur d'un prédicat n'est ajoutée que lorsqu'on le rencontre pour la première fois dans le flot d'instructions, même si le même prédicat apparaît plusieurs fois.

5.1.2 Prédicteur sélectif de prédicats

Historique des branchements versus historique des branchements et prédicats

La précision d'un prédicteur de branchements ou de prédicats dépend de l'algorithme de prédiction, de la taille du prédicteur et aussi de la qualité du vecteur d'informations que le prédicteur exploite. Il a été montré que l'historique global des branchements est un vecteur

d'informations de très bonne qualité pour prédire les branchements. Le prédicteur TAGE [54] est généralement considéré comme l'état de l'art des prédicteurs à historique global. En conséquence, nous utilisons un prédicteur dérivé de TAGE pour prédire les prédicats.

La valeur des prédicats est corélée aux directions des branchements qui les précèdent, mais aussi aux valeurs des prédicats antérieurs. Ainsi, il est naturel de vouloir utiliser l'historique global des branchements et prédicats pour prédire les prédicats. Dans le reste de ce chapitre, nous appellerons *BrPred* le prédicteur qui utilise l'historique global des branchements et prédicats.

Une approche alternative est d'utiliser l'historique global conventionnel des branchements pour prédire les prédicats. Dans la suite, nous appellerons *BrO* le prédicteur qui utilise l'historique global des branchements.

Dans nos résultats (voir la figure 5.4 dans la partie 5.3), lorsque la prédiction est utilisée systématiquement nous observons que le prédicteur à historique global des branchements et prédicats obtient des performances légèrement supérieures à celles obtenues par le prédicteur à historique global des branchements. Cependant, nous observons aussi que pour un certain nombre de programmes de test le taux de mauvaises prédictions de prédicats est très élevé. Cela provoque des pertes de performance par rapport à notre architecture de base où les instructions prédiquées sont converties en *false predicate conditional move* (FPCMOV). Cela semble indiquer qu'en cas de prédicat difficile à prédire, il est plus intéressant de traiter l'instruction prédiquée associée en la transformant en FPCMOV.

Sélection de la prédiction

Pour différencier les prédicats difficiles à prédire de ceux qui sont faciles à prédire, le moyen le plus naturel est d'utiliser l'estimation de la confiance en la prédiction. Un tel estimateur de confiance en la prédiction a été proposé spécifiquement pour le prédicteur TAGE [51]. Son principal avantage est qu'il est très peu coûteux à implémenter. Il semble être relativement efficace de n'utiliser la prédiction que lorsque la confiance en celle-ci est élevée et de convertir en FPCMOV les instructions non prédites, comme le montrent nos expérimentations qui utilisent le prédicteur *BrO* (voir la figure 5.5 dans la partie 5.3). Dans la suite, nous appellerons *BrO+HighConf* le prédicteur *BrO* dont la prédiction n'est utilisée que si sa confiance est élevée.

Malheureusement, le filtre de la confiance élevée ne peut pas être utilisé avec le prédicteur *BrPred* à cause de la corruption de l'historique des branchements et prédicats qui en dériverait. En effet, la prédiction est efficace et précise si le prédicteur est accédé avec le même vecteur d'informations (historique + PC) au moment de la lecture, c'est-à-dire au moment de la prédiction, et au moment de la mise à jour du prédicteur, c'est-à-dire au moment où l'instruction est validée à la sortie du pipeline. Si la prédiction d'un prédicat n'est pas utilisée et que l'instruction associée au prédicat est transformée en FPCMOV, l'historique spéculatif des branchements et prédicats peut être corrompu, puisqu'en cas de mauvaise prédiction, comme elle n'a pas été utilisée, il n'est pas nécessaire de déclencher le mécanisme de correction de l'exécution et ainsi de corriger l'historique. La figure 5.2 illustre ce problème. Cette corruption de l'historique entraîne potentiellement des mauvaises prédictions pour les instructions suivantes. Un filtre où la prédiction ne serait utilisée que si sa confiance est élevée provoquerait donc une telle corruption de l'historique

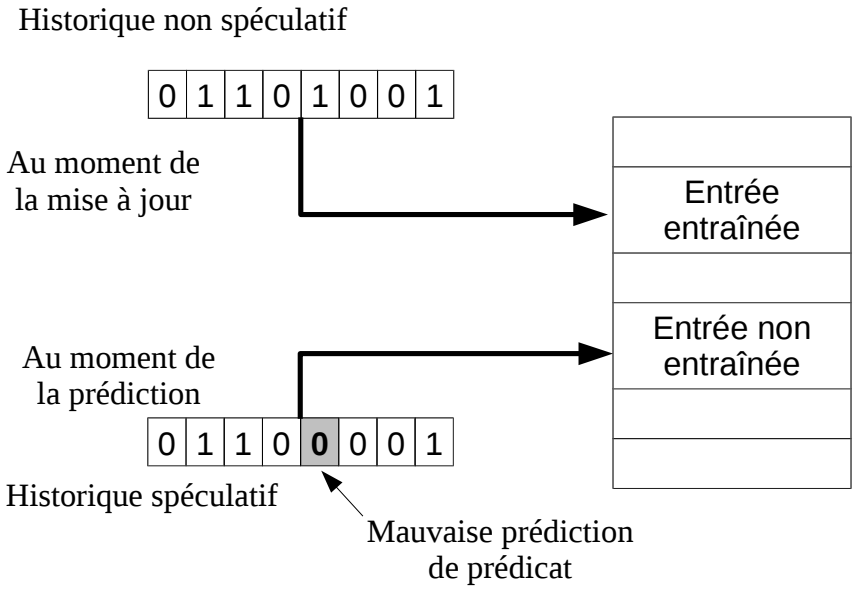


FIGURE 5.2 – *Un historique des branchements et prédicats corrompu entraîne la lecture de la mauvaise entrée du prédicteur, causant alors potentiellement une mauvaise prédiction.*

des branchements et prédicats.

En conséquence, pour *BrPred* nous utilisons un mécanisme plus global pour décider si la prédiction de prédicats doit être utilisée ou non. Ainsi, deux modes sont définis : le mode *on* dans lequel la prédiction est utilisée et le mode *off* dans lequel la prédiction n'est pas utilisée. Passer du mode *off* au mode *on* demande un vidage complet du pipeline pour pouvoir repartir d'un historique global des branchements et prédicats spéculatif correct. Cela impose donc de ne pas changer de mode trop souvent. Nous avons construit une heuristique de changement de mode assez efficace. Elle est décrite dans la suite.

Le prédicteur *BrPred* est continuellement mis à jour au moment de la validation des instructions en sortie du pipeline. Sa précision est observée et la sélection du mode *on* ou *off* se fait périodiquement. L'intervalle d'observation que nous avons choisi est de dix mille instructions prédiquées traitées par le pipeline. Le mode *on* est activé si le nombre de mauvaises prédictions pendant l'intervalle observé est inférieur à cinq cents. Dans la suite, nous appellerons *BrPred-OnOff* le prédicteur *BrPred* pour lequel les modes *on* et *off* sont utilisés.

5.1.3 Utilisation de la prédiction

Comme mentionné auparavant, la prédiction de prédicats est systématiquement utilisée pour neutraliser les instructions prédiquées faux pour le prédicteur *BrPred* et pour le prédicteur *BrPred-OnOff* lorsque celui-ci est en mode *on*. Cela reste valable pour le prédicteur *BrO* et pour le prédicteur *BrO-HighConf* lorsque la confiance en la prédiction est élevée. De plus, lorsque la valeur du prédicat est déjà connue avant que l'instruction soit renommée, c'est cette valeur qui est directement utilisée et non la prédiction. Pour ces cas, on obtient donc une précision de 100% comme pour les cas de branchements résolus en avance.

Dans notre modèle de simulation, les instructions prédiquées faux sont quand même stockées dans l'*Issue Queue* (IQ). Elles y restent tant que la prédiction de prédicats n'a pas été vérifiée. Dans le cas d'une mauvaise prédiction l'exécution doit être corrigée.

5.2 Rejeu sélectif pour les mauvaises prédictions de prédicats

Lorsqu'une mauvaise prédiction de prédicat est détectée, il peut arriver qu'un certain nombre d'instructions suivant l'instruction mal prédite soient déjà exécutées. Un mécanisme de rejeu sélectif peut être utilisé pour corriger l'exécution au lieu de juste vider le pipeline et de réexécuter la séquence complète d'instructions. Le mécanisme de rejeu sélectif ne réexécute que les instructions qui font partie de la chaîne de dépendances de l'instruction prédiquée mal prédite. Sur les processeurs haute performance actuels (d'architecture x86), le rejeu sélectif est implémenté pour être utilisé lors d'évènements comme une mauvaise prédiction de succès ou de défaut dans le cache de premier niveau ou encore un conflit dans les banques de celui-ci.

Cependant, dans le cas d'une mauvaise prédiction de prédicat, une difficulté majeure apparaît : le renommage des registres n'est plus valable. En effet, comme la prédiction de prédicats est utilisé pour résoudre le problème des définitions multiples, si la prédiction est fausse, le problème a mal été résolu et il faut dont refaire le renommage avec la bonne valeur du prédicat. Pour résoudre ce problème sans devoir vider complètement le pipeline pour chaque mauvaise prédiction de prédicat, nous proposons d'adapter le mécanisme de SYRANT présenté dans le chapitre 3, même si celui-ci a été initialement conçu pour exploiter l'indépendance de contrôle sur un jeu d'instructions non prédiquées.

5.2.1 Utilisation symétrique des ressources

Pour permettre l'exploitation de l'indépendance de contrôle, SYRANT force la même allocation des ressources du moteur d'exécution dans le désordre sur le chemin pris et sur le chemin non pris d'un branchement conditionnel. Ces ressources sont les registres physiques, les entrées du ROB et de la LSQ. SYRANT alloue le même nombre d'entrées et de registres sur les deux chemins. Forcer cette répartition égale se révèle assez complexe dans le cas des branchements conditionnels.

Cependant, dans ce chapitre, nous nous limitons à exploiter l'indépendance de contrôle des instructions prédiquées. Dans ce cadre, l'allocation égale des ressources pour les deux chemins (prédiqué faux et prédiqué vrai) est immédiate : registre, entrée du ROB et entrée de la LSQ sont allouées même si l'instruction est prédiquée faux. Ainsi, le renommage de l'instruction ne dépend pas de la prédiction de son prédicat (du point de vue de la consommation de ressources) et une entrée du ROB lui est toujours allouée, de même pour l'entrée de la LSQ si c'est une instruction mémoire. La chaîne de dépendance (prédite) dépend par contre toujours de la prédiction du prédicat. Et donc le renommage des registres doit être recalculé.

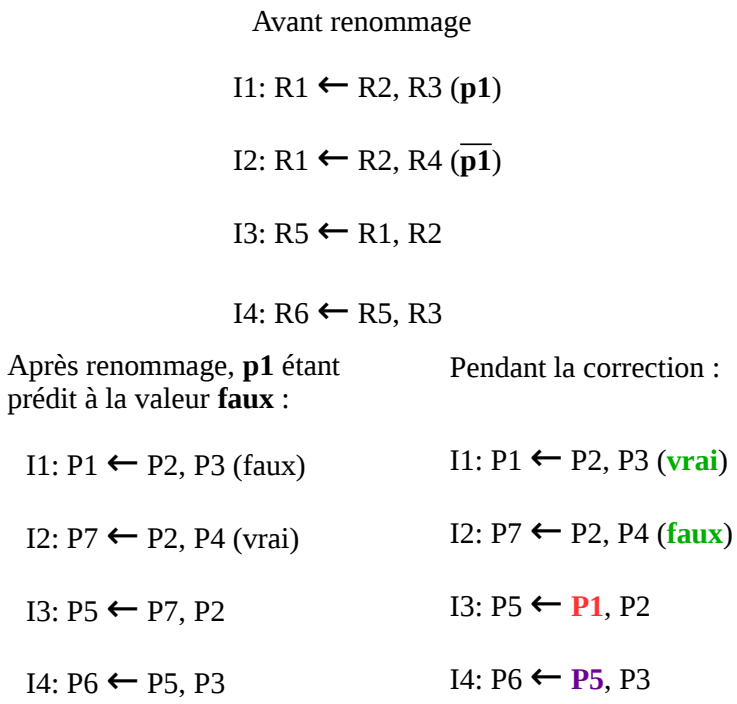


FIGURE *5.3 – Les instructions qui n'ont pas la même forme de renommage avant et après la correction sont à réexécuter. Les instructions dépendantes de celles-ci doivent aussi être réexécutées.*

5.2.2 Initialisation d'un rejeu lors d'une mauvaise prédiction de prédicat

Dans le cas de la correction d'une mauvaise prédiction de prédicat, contrairement au cas d'une mauvaise prédiction de branchement, la séquence d'instructions correcte est exactement la même que celle qui a utilisé la prédiction, seul le renommage change. En conséquence, notre modèle suppose l'existence d'un tampon dans lequel sont stockées toutes les instructions qui sont entrées dans le pipeline. La taille de ce tampon est égale au nombre maximum d'instructions en cours de traitement dans le pipeline. Cela évite de devoir corriger l'exécution en faisant entrer à nouveau ces instructions dans le pipeline. Le traitement de ces instructions est juste relancé à partir de l'étage de renommage.

Le renommage est relancé à partir du tampon à la vitesse maximale, avec la bonne valeur du prédicat. Dans le même temps, le prédicteur de prédicats est réinitialisé avec un historique correct (dans le cas du prédicteur *BrPred*).

5.2.3 Identification des résultats valides

Il arrive souvent que des instructions suivant l'instruction prédiquée mal prédite soient déjà exécutées. Pour conserver leur résultat, notre mécanisme préserve l'allocation du registre physique destination, de l'entrée du ROB et de la LSQ. Cependant, il faut aussi s'assurer de la validité du résultat ainsi sauvegardé comme illustré dans la figure 5.3.

Sur cet exemple, l'instruction $I3$ doit être réexécutée puisqu'un de ses registres opérandes

a changé et que son résultat précédemment calculé est donc maintenant invalide. Bien qu'utilisant les mêmes registres opérandes, l'instruction $I4$ doit aussi être réexécutée puisqu'un des ses registres opérandes (produit par $I3$) est maintenant invalide.

Pour résoudre ce problème de validité, nous avons utilisé exactement la même solution à base d'étiquettes, les *rename-sequence tag* (RS-tag), que pour SYRANT. Ce mécanisme préserve les résultats des instructions indépendantes des données qui suivent l'instruction mal prédite.

5.2.4 SPREPI

Notre mécanisme SPREPI est donc composé d'un prédicteur de prédicats (*BrO* ou *BrPred*), d'un mécanisme de sélection de l'utilisation de la prédiction associée (le filtre de confiance élevé pour *BrO* et le filtre *OnOff* pour *BrPred*) et du mécanisme de rejeu sélectif.

5.3 Étude expérimentale

L'étude expérimentale pour valider le mécanisme proposé a été conduite sur un simulateur basé sur l'environnement de simulation Gem5 [10].

5.3.1 Paramètres du simulateur

Sauf mention contraire, le simulateur modélise un processeur superscalaire puissant à quatre voies avec un ROB de 128 entrées, une *Load Queue* (LQ) de 64 entrées, une *Store Queue* (SQ) de 64 entrées et 256 registres physiques entiers et flottants. Le processeur possède aussi un prédicteur de branchements à l'état de l'art, le prédicteur TAGE [54]. Le prédicteur *store sets* [14] est utilisé pour prédire les dépendances mémoires. Un mécanisme est utilisé pour améliorer la gestion de la *Return Address Stack* (RAS) [61]. Lorsque la prédiction de prédicats n'est pas appliquée à une instruction prédiquée, celle-ci est transformée en la première version de FPCMOV (celle sans ajout de micro-instruction de sélection) pour permettre son traitement. Les autres caractéristiques du simulateur sont rassemblées dans la table 5.1. Nous nous référerons à cette configuration, où aucun prédicat n'est prédit, en tant que configuration de base (BASE). La table 5.1 montre aussi une configuration à huit voies qui sera utilisée pour estimer les résultats de notre mécanisme sur une configuration plus puissante.

5.3.2 Jeu de tests

Le jeu de tests qui a été utilisé est constitué d'un sous-ensemble du jeu de tests SPEC2006 [59]. L'ensemble des programmes sélectionnés est listé dans la table 5.2. Pour réduire le temps de simulation, nous avons utilisé la méthodologie *Simpoint* [21] pour résumer chaque programme de test en un ensemble de tranches de cent millions d'instructions. Chaque tranche représente une certaine partie de l'exécution du programme de test à laquelle est associé un poids qui correspond à l'importance de cette partie par rapport à l'exécution totale. Pour chaque programme, les résultats montrés sont la moyenne pondérée des résultats obtenus sur chaque tranche de l'en-

Exécution dans le désordre à 1GHz	4 voies	8 voies
Mémoire principale	100 cycles 12.8Go/s, à travers un bus de 128 octets	
Caches (associatifs par ensemble)	Données L1 : 64Ko, 4 voies, ligne de 64 octets, 1 cycle Instructions L1 : 64Ko, 4 voies, ligne de 64 octets, 1 cycle Partagé L2 : 4Mo, 8 voies, ligne de 64 octets, 8 cycles *Prefetcher* de type *Stride* pour le L2	
TLBs	Parfait, pages de 4Ko	
ROB	128 entrées	256 entrées
IQ	128 entrées	256 entrées
LSQ	128 entrées (64L/64S)	256 entrées (196L/64S)
Largeur (entrée/decodage/lancement/validation)	4	8
Pipeline	12 étages	
UF(latence) IntAlu(1) IntMultDiv(3/12*) FpAlu(5) FpMultDiv(4/9*) Ld/Str(2)	 3 1 2 2 2	 6 2 4 4 4
Prédicteur de branchements	BTB : 4 voies, 1K entrées RAS : 16 entrées, détection de la corruption due au mauvais chemin TAGE : 256 Kbits, 1+12 composants, 15K entrées au total	
Prédicteur de dépendances mémoire	*Store sets*	
Pénalité minimum de mauvaise prédiction	15 cycles	15 cycles

TABLE *5.1* – *Résumé des configurations du simulateur. *non pipelinée.*

semble. La table 5.2 montre la moyenne pondérée du nombre d'instructions par cycle (IPC) pour chaque programme de test, pour les configurations BASE à quatre et à huit voies.

Comme nous ciblons le jeu d'instructions ARM, certains programmes de test ou certains de leurs jeux de données d'entrée ne fonctionnent pas avec ce jeu d'instructions. Il y a trois raisons pour lesquelles certains programmes de test sont exclus : le binaire produit par notre compilateur ciblant l'architecture ARM ne fonctionne pas sur une architecture ARM native, le binaire n'est pas exécutable sur *qemu-arm* [9], qui a été utilisé pour calculer les *basic bloc vectors* (BBV) nécessaires pour produire les *Simpoint*, ou le simulateur n'est pas capable de les traiter. Finalement, nous sommes parvenus à utiliser douze programmes de test entiers (la suite complète des programmes de test entiers) et sept programmes de test flottants. Certains programmes de test sont utilisés avec plusieurs jeux de données d'entrée. En tout nous avons trente-huit charges de travail différentes.

Les binaires ont été obtenus avec le compilateur *gcc* avec le niveau d'optimisation *O3*. Cela active la transformation de *if-convertion* mais pas celle du pipeline logiciel puisque nous ciblons principalement les instructions prédiquées produites par la *if-conversion*. La décision de *gcc* de convertir un branchement à l'aide de la *if-conversion* dépend du nombre d'instructions contrôlées par ce branchement. Pour l'architecture *ARM*, cette limite est codée en dur et s'élève à quatre instructions.

Programmes de test	Données d'entrée	IPC (BASE)		% instructions prédiquées	
		4 voies	8 voies	avec branchements	sans branchement
400.perlbench	checkspam	1,46	1,82	17,17	4,57
	diffmail	1,31	1,57	16	4,68
401.bzip2	chicken	2,21	3,01	15,17	3,49
	combined	1,73	2,16	15,77	5,20
	liberty	2,33	3,38	17,98	5,72
	program	1,85	2,23	14,82	3,92
	source	1,66	2,02	17,75	5,45
	text	2,11	3,15	17,28	4,29
403.gcc	166	1,7	2,37	35,91	24,23
	200	1,58	2,1	31,07	18,69
	c-typeck	1,65	2,41	44,44	34,27
	cp-decl	1,72	2,46	37,1	25,9
	expr	1,84	2,71	38,16	26,12
	scilab	1,47	1,92	29,96	17,6
416.gamess	cytosine	2,62	4,6	7,01	2,85
	h2ocu2+	2,76	5,09	6,39	2,93
429.mcf	ref	0,7	0,81	24,5	6,10
435.gromacs	ref	2,81	4,74	4,62	1,02
436.cactusADM	ref	2,26	4,3	0,09	0,02
444.namd	ref	2,49	3,95	8,54	5,18
445.gobmk	13x13	1,75	2,27	18,92	7,39
	nngs	1,73	2,24	18,02	6,79
	trevorc	1,71	2,22	18,39	7,13
	trevord	1,87	2,52	16,86	6,11
453.povray	ref	1,48	2	8,09	1,9
456.hmmer	nph3	2,62	5,21	17,18	14,55
	retro	2,48	4,61	17,64	14,78
458.sjeng	ref	1,82	2,38	18,76	7,05
459.GemsFDTD	ref	2,17	3,89	1,06	0,001
462.libquantum	ref	1,82	2,31	23,56	13,42
464.h264ref	baseline	2,04	3,59	6,45	2,68
	main	1,7	2,99	6,75	2,42
	sss	1,68	3,05	5,99	2,13
470.lbm	ref	1,88	2,36	0,6	0,02
471.omnetpp	ref	0,8	0,92	17,1	4,33
473.astar	BigLakes	1,29	1,57	15,53	3,88
	rivers	1,42	1,69	15,7	3,29
483.xalancbmk	ref	1,81	2,49	21,09	3,4

Table *5.2* – *Les programmes de test, leur jeu de données d'entrée, leur IPC sur la configuration BASE en quatre et huit voies et le ratio d'instructions prédiquées par rapport au nombre total d'instructions.*

5.3.3 Ratio d'instructions prédiquées dans les programmes de test

La table 5.2 liste aussi le ratio d'instructions prédiquées dans les programmes de test sélectionnés. La première colonne présente le pourcentage total d'instructions prédiquées. Cela inclut les branchements conditionnels qui ne sont pas prédits par le prédicteur de prédicats. La seconde colonne exclut les branchements conditionnels.

Pour tous les programmes de test, les branchements conditionnels représentent une part importante des instructions prédiquées. Cependant, certains programmes de test tels que *401.bzip2*, *403.gcc*, *445.gobmk* et *456.hmmer* contiennent une portion significative d'instructions prédiquées effectives, c'est-à-dire d'instructions prédiquées qui ne sont pas des branchements conditionnels. Certains autres programmes de test comme *436.cactusADM*, *459.GemsFDTD* et *470.lbm* ne possèdent quasiment pas d'instructions prédiquées. Une optimisation simple pour réduire la consommation d'énergie serait d'observer au cours de l'exécution le ratio d'instructions prédiquées effectives et d'arrêter la prédiction de prédicats quand ce ratio est en dessous d'un seuil prédéfini.

5.3.4 Résultats de simulations

Les résultats de simulations que nous reportons (qui sont des accélérations de l'exécution par rapport à l'exécution sur la configuration de base) sont obtenus sur une configuration simulant un processeur superscalaire à quatre voies sauf pour la partie 5.3.4 qui montre que les tendances obtenues sont amplifiées sur un processeur superscalaire à huit voies.

Historique des branchements versus historique des branchements et prédicats

Les premières expérimentations où la prédiction de prédicats est systématiquement utilisée (figure 5.4) montrent que l'utilisation de l'historique de branchements et prédicats apporte en général un gain de performance légèrement plus important que l'utilisation de l'historique de branchements seul. Dans certains cas, la différence est assez significative, par exemple pour *462.libquatum* et pour *401.bzip.liberty*.

Il est à noter que pour certains programmes de test, le ratio d'instructions prédiquées effectives est assez bas mais les gains de performance sont significatifs. C'est le cas par exemple pour *429.mcf* et pour *471.omnetpp.ref*. Cependant, dans de nombreux cas (par exemple *444.namd*, *445.gobmk*, *464.h264ref*, ...), l'utilisation systématique de la prédiction de prédicats induit des pertes de performance par rapport à la configuration de base. Cette perte de performance peut être très importante, par exemple de 37% pour *456.hmmer*. En conséquence, l'utilisation systématique de la prédiction de prédicats ne devrait pas être considérée pour implémentation dans un vrai processeur.

Filtrage de l'utilisation de la prédiction de prédicats

Nous avons proposé deux prédictions de prédicats sélectives, le but principal étant d'éviter les pertes de performance dramatiques décrites précédemment.

Lorsque l'historique global des branchements seul est utilisé, le prédicteur *BrO-HighConf* sélectionne l'utilisation de la prédiction de prédicats à un grain très fin en se basant sur la confiance en la prédiction. Lorsque l'historique global des branchements et prédicats est utilisé, le prédicteur *BrPred-OnOff* filtre l'utilisation de la prédiction de prédicats à un grain plus gros en activant ou non cette utilisation sur une période donnée.

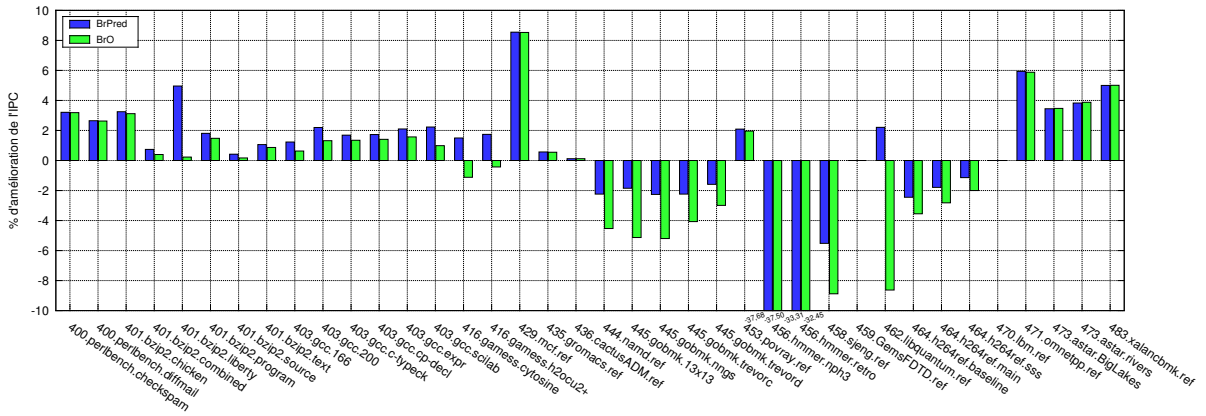


Figure 5.4 – *Accélération de l'exécution en utilisant* BrPred *(historique des branchements et prédicats) et* BrO *(historique des branchements seul)*

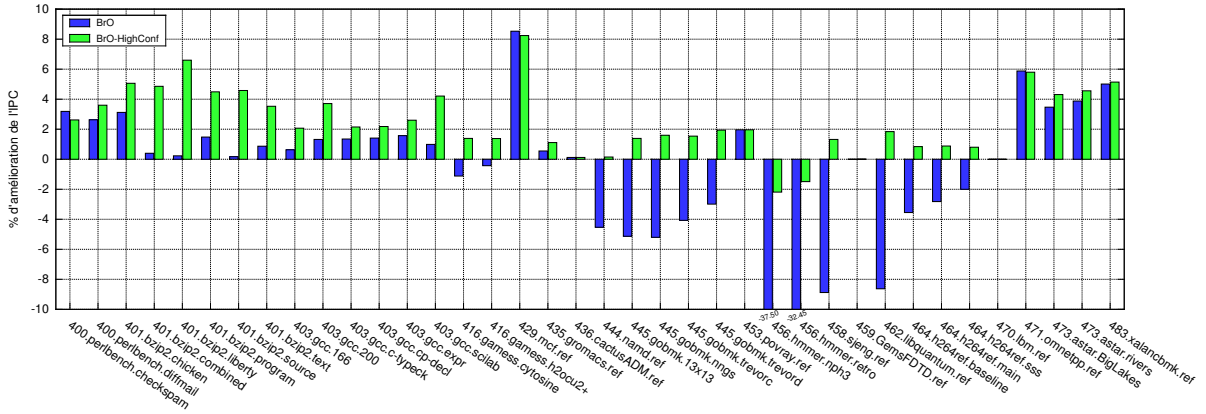


Figure 5.5 – *Accélération en utilisant* BrO-HighConf *(prédiction sélective et historique des branchements seuls)*

Les figures 5.5 et 5.6 illustrent les expérimentations associées. Comme prévu, ces filtres servent essentiellement à supprimer les pertes de performance induites par des taux de mauvaises prédictions élevés. De plus, ils maintiennent et parfois améliorent les gains de performance dus à la prédiction de prédicats lorsqu'ils existent.

Lorsque le prédicteur *BrO-HighConf* est utilisé, les pertes de performance par rapport à la configuration de base deviennent marginales (au maximum 2,2% pour *456.hmmer*). De plus, en éliminant la plupart des mauvaises prédictions tout en conservant la majorité des prédictions correctes, le filtre de confiance élevée permet des gains de performance significatifs sur la plupart des programmes de test (par exemple pour *401.bzip2* et pour *403.gcc*).

Lorsque le prédicteur *BrPred-OnOff* est utilisé, le filtre *OnOff* supprime toutes les pertes de performance excepté pour *444.namd* (perte de 1,9%). Ce filtre est aussi très efficace pour éviter les mauvaises prédictions de prédicats et permettre des gains de performance.

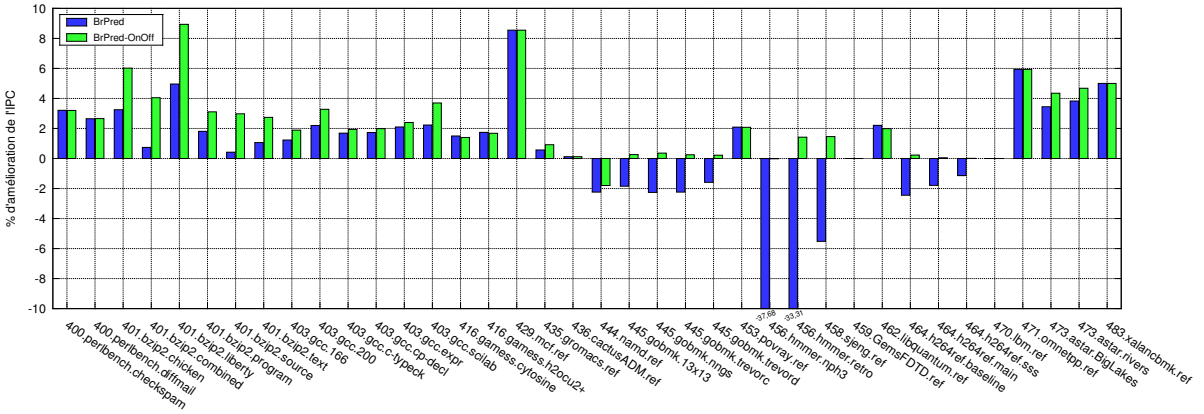


Figure 5.6 – *Accélération en utilisant* BrPred-OnOff *(prédiction sélective et historique des branchements et prédicats)*

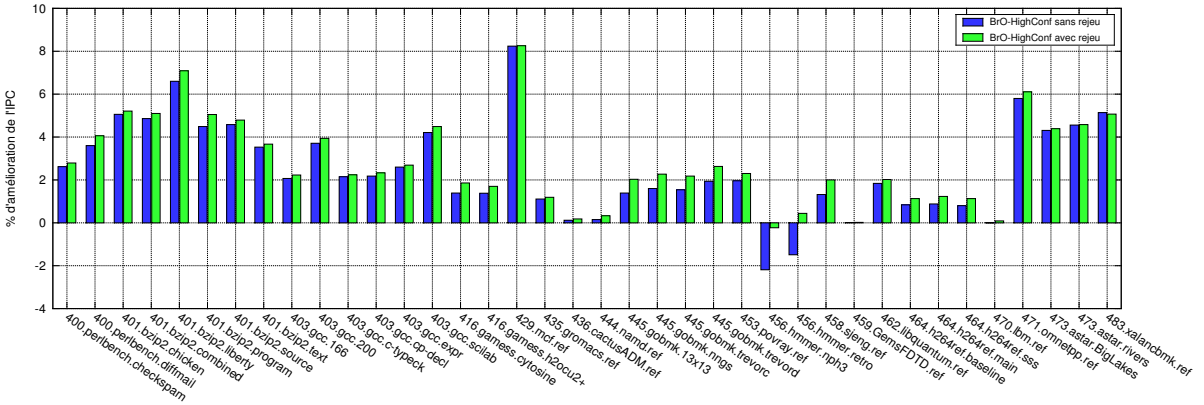


Figure 5.7 – *SPREPI : rejeu sélectif et* BrO-HighConf

Bénéfices du rejeu sélectif

Le but de notre mécanisme de rejeu sélectif est de réduire les pertes de performance dues aux mauvaises prédictions de prédicats. Les résultats sont illustrés sur les figures 5.7 et 5.8.

Pour les deux prédicteurs, des gains de performance marginaux sont obtenus par l'utilisation du mécanisme de rejeu (généralement moins de 1%). En pratique, le filtre de confiance élevée et le filtre *OnOff* sont assez efficaces pour éviter l'utilisation de la prédiction de prédicats pour les prédicats difficiles à prédire. Le nombre d'utilisations d'une mauvaise prédiction est donc faible et le mécanisme de rejeu n'est donc pas utilisé souvent. En conséquence, le bénéfice général qui peut être obtenu en réduisant la coût d'une mauvaise prédiction est faible. Cependant les tests que nous avons effectués en utilisant systématiquement la prédiction de prédicats et en activant le mécanisme de rejeu sélectif montrent que celui-ci est efficace pour absorber la pénalité due aux mauvaises prédictions. La perte de performance observée sur *456.hmmer* chute ainsi de 37% à 9%.

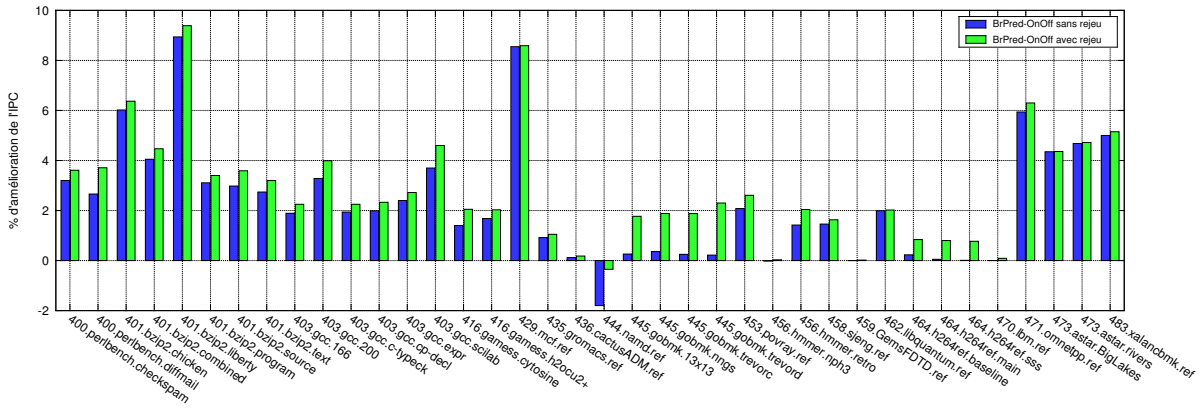


Figure *5.8* – *SPREPI : rejeu sélectif et* BrPred-OnOff

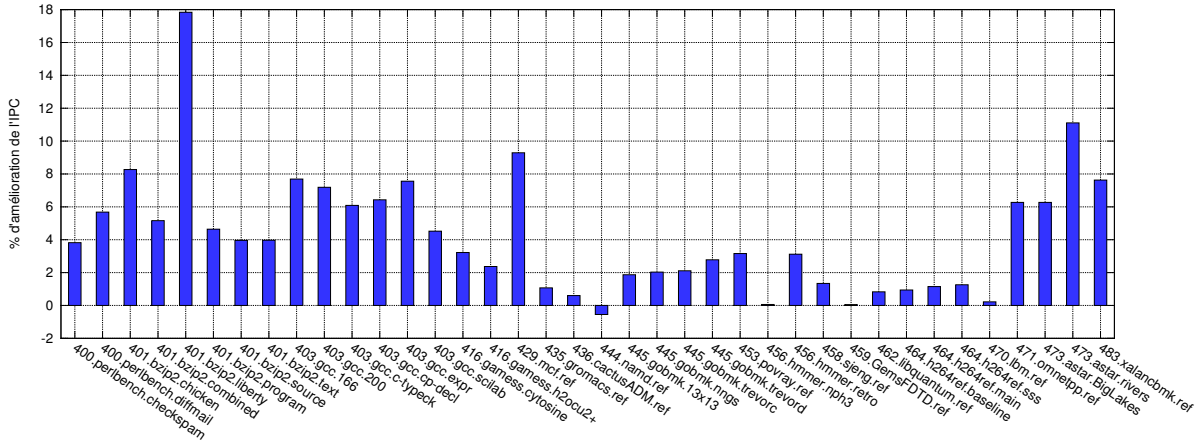


Figure *5.9* – *Bénéfices de SPREPI (version* BrPred-OnOff *avec rejeu sélectif) sur un processeur superscalaire à huit voies.*

Bénéfices de SPREPI sur un processeur superscalaire à grande largeur de traitement

L'accélération de l'exécution permise par SPREPI sur un processeur superscalaire à quatre voies est limitée à quelques pourcent (jusqu'à 9% sur deux de nos charges de travail). Cependant, cet avantage augmente lorsque l'on considère une implémentation plus puissante avec un processeur dans le désordre à grande largeur de traitement. La figure 5.9 illustre les résultats obtenus sur un processeur à huit voies pour le prédicteur *BrPred-OnOff* avec le mécanisme de rejeu sélectif activé. L'accélération par rapport à un processeur à huit voies de base s'élève jusqu'à 17% et l'accélération relative est systématiquement plus élevée pour un processeur à huit voies que pour un processeur à quatre voies pour presque tous les programmes de test. Cette tendance s'explique par la capacité plus importante d'un processeur à huit voies à exploiter le parallélisme d'instructions. En effet, celui-ci étant augmenté par l'utilisation de la prédiction de prédicats qui évite la séquentialisation de l'exécution des instructions prédiquées, un processeur à huit voies est plus à même d'exploiter ce surplus de parallélisme qu'un processeur à quatre voies.

Première et deuxième version des FPCMOV

Les performances de l'exécution dans le désordre d'instructions prédiquées dépendent de comment le problème des définitions multiples est résolu. Jusqu'à présent nous avons utilisé la première version des FPCMOV comme traitement de base des instructions prédiquées. Des expérimentations ont aussi été faites avec la deuxième version des FPCMOV, moins performante puisque transformant une instruction prédiquée en deux micro-instructions dont l'implémentation et l'exécution sont plus faciles. Cependant, comme cela ajoute des instructions à traiter pour le processeur, cette version a un coût plus élevé.

La figure 5.10 illustre la performance relative d'une configuration où l'exécution des instructions prédiquées passe par leur transformation en FPCMOV version 2 (étiqueté BASE FPCMOV2) par rapport à la configuration BASE dans laquelle les instructions prédiquées sont traitées en les transformant en FPCMOV version 1 (BASE FPCMOV1). À cela s'ajoutent les performances relatives de SPREPI (*BrPred-OnOff* + rejeu sélectif) utilisant FPCMOV version 2 (étiqueté SPREPI FPCMOV2) et celle de SPREPI utilisant FPCMOV version 1 (étiqueté SPREPI FPCMOV1) par rapport à cette même configuration BASE FPCMOV1.

On peut d'abord observer que la différence de performance entre BASE FPCMOV1 et BASE FPCMOV2 est relativement limitée. Quelques pourcents (inférieur à 4%) de pertes de performance sont parfois présents mais pour la plupart de nos programmes de test, la perte de performance est marginale. Pour *470.lbm*, il y a même un gain de performance. Cependant, comme *470.lbm* ne possède que 0,02% d'instructions prédiquées effectives, il est fortement probable que cela soit associé à un quelconque artefact d'ordonnancement des instructions.

Puisque la plupart des instructions prédiquées sont prédites la différence entre SPREPI FPCMOV1 et SPREPI FPCMOV2 est d'autant plus faible. Elle est de moins d'1% dans nos expérimentations, sauf pour *458.sjeng*. La différence de performance entre BASE FPCMOV1 et BASE FPCMOV2 ne devrait pas justifier la complexité matérielle additionnelle et l'énergie supplémentaire consommée par la configuration BASE FPCMOV1 (ports de fichiers de registres additionnels, réseau de bypass plus large, logique de lancement des instructions plus complexe). Lorsque l'on considère SPREPI avec ces deux implémentations possibles, celle utilisant FPCMOV version 1 ne vaut évidemment pas le coût matériel additionnel et l'énergie supplémentaire consommée.

5.4 Conclusion et perspectives

Les processeurs basés sur le jeu d'instructions ARM sont devenus omniprésents dans les appareils modernes, en particulier les smartphones et les tablettes tactiles. La demande de toujours plus de performance pousse les concepteurs de processeurs ARM à utiliser les mêmes techniques utilisées pendant les deux dernières décennies pour les processeurs haute performance à destination des PC et des serveurs, incluant les processeurs superscalaires à grande largeur de traitement des instructions. Le jeu d'instructions 32 bits ARM possède des instructions prédiquées. Fournir une solution efficace pour exécuter dans le désordre ces instructions prédiquées de manière performante est difficile en raison du problème des définitions multiples.

Dans ce document, nous avons montré que l'état de l'art de la prédiction de branchements

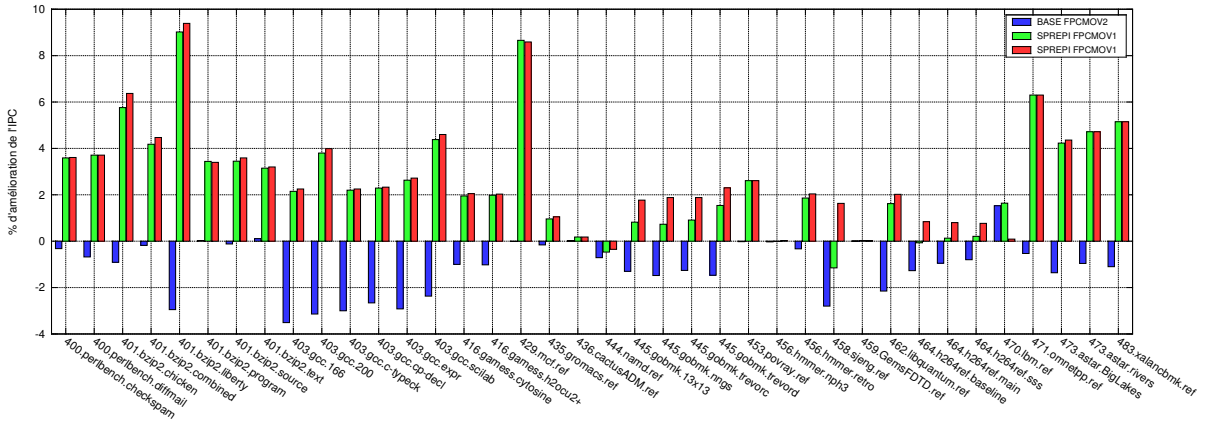


Figure *5.10* – *Exécution des instructions prédiquées avec FPCMOV version 1 et avec FPCMOV version 2.*

peut être adapté pour faire de la prédiction de prédicats. La prédiction de prédicats résout le problème des définitions multiples, ce qui permet un certain gain en performance si une prédiction avec une précision raisonnable est obtenue. Utiliser l'historique des branchements et prédicats à la place de l'historique des branchements seul améliore encore ce gain. Cependant, l'utilisation systématique de la prédiction de prédicats n'est pas toujours très efficace et certains programmes de test ou phases d'exécution de programmes de test avec des taux de mauvaises prédictions très élevés ont des performances pires que sans la prédiction de prédicats. Nous avons montré qu'un filtrage efficace de l'utilisation de la prédiction de prédicats peut être implémenté pour le prédicteur utilisant l'historique des branchements seul et pour celui utilisant l'historique des branchements et prédicats.

La seconde contribution de ce chapitre est l'adaptation du mécanisme de rejeu sélectif de SYRANT [40] pour l'utiliser lors de la correction des mauvaises prédictions de prédicats. SYRANT a été initialement créé pour exploiter l'indépendance de contrôle dans les processeurs à exécution dans le désordre génériques. L'adapter pour qu'il traite seulement les instructions prédiquées est assez naturel et permet de simplifier grandement l'organisation du mécanisme. Le rejeu sélectif des instructions prédiquées réduit les pénalités de performance associées aux mauvaises prédictions de prédicats.

La combinaison de ces deux mécanismes, *Selective Prediction and Replay for Predicated Instructions* (SPREPI), permet des accélérations de l'exécution de codes générés par le compilateur standard *gcc*. Ce gain de performance est de quelques pourcents pour un processeur superscalaire à quatre voies, mais il augmente avec la largeur de traitement du pipeline (jusqu'à 17% pour certains programmes de test sur un processeur à huit voies).

En transformant l'exécution de la plupart des instructions prédiquées en instructions non prédiquées (au risque d'une mauvaise prédiction), SPREPI réduit le nombre de séquentialisations produites par la transformation des instructions prédiquées en FPCMOV. SPREPI rend presque négligeable les bénéfices de performance obtenus en utilisant une solution matérielle de base plus optimisée, et permet donc une implémentation plus intéressante au niveau coût et énergétique.

Les solutions standards pour implémenter l'exécution dans le désordre des jeux d'instructions prédiqués ont tendance à pousser les concepteurs de compilateurs à ne pas utiliser d'instructions prédiquées puisque celles-ci peuvent induire une certaine séquentialisation de l'exécution. À l'opposé, notre proposition SPREPI pourrait permettre au compilateur d'utiliser la transformation de *if-conversion* de manière plus poussée sans prendre le risque de produire des programmes dont les performances sont moins bonnes.

Conclusion

Les travaux présentés dans cette thèse sont construits sur l'observation, prédite par la loi d'Amdahl [5], que malgré le nombre toujours plus important de cœurs d'exécution présents dans un même processeur, augmentant aussi les ressources disponibles pour l'exécution parallèle d'un programme, de bonnes performances en exécution séquentielle sont toujours essentielles. Or la performance en exécution séquentielle passe par un traitement efficace des branchements.

Deux voies ont particulièrement été explorées pour le traitement des branchements : la prédiction de branchements et l'élimination des branchements en les remplaçant par des instructions prédiquées. Dans cette thèse, nous présentons deux propositions architecturales s'orientant chacune dans une de ces deux directions.

La prédiction de branchements a été longuement étudiée et la précision des prédicteurs semble avoir atteint un niveau difficilement améliorable. Il devient donc nécessaire d'explorer une direction alternative à l'amélioration de la précision de la prédiction. Une des pistes explorée au cours des dernières années est l'exploitation de la reconvergence du flot de contrôle et de l'indépendance de contôle. Un branchement conditionnel définit deux chemins d'exécution possibles : le chemin pris et le chemin non pris. La reconvergence du flot de contrôle est la propriété selon laquelle ces deux chemins reconvergent en un seul au bout d'un certain nombre d'instructions. On peut donc distinguer les instructions qui sont sur un chemin et pas l'autre, et qui sont dépendantes du branchement, des instructions qui sont sur les deux chemins, après le point de reconvergence, et qui sont donc indépendantes du contrôle réalisé par le branchement. L'exploitation de l'indépendance de contrôle consiste à réutiliser sur le bon chemin les résultats des instructions indépendantes du branchement qui ont été exécutées sur le mauvais chemin. Plusieurs mécanismes exploitant l'indépendance de contrôle ont été proposés dans la littérature. La plupart sont cependant soit trop complexes soit ne ciblent pas assez de cas. En conséquence, notre proposition SYRANT, pour *SYmmetric Resource Allocation on Not-taken and Taken paths*, utilise les principes de reconvergence du flot de contrôle et d'indépendance de contrôle pour réduire le coup d'une mauvaise prédiction de branchement en réutilisant le travail déjà accompli par le processeur sur le mauvais chemin sur des instructions communes au bon et au mauvais chemins. Notre proposition se veut peu intrusive dans le pipeline, n'induisant ainsi pas une grande complexité matérielle. Les principales composantes de SYRANT sont un mécanisme d'insertion de vides dans les structures du pipeline (ROB, LSQ, *free list* des registres physiques) permettant de forcer l'allocation des mêmes ressources sur les chemins pris et non pris pour les instructions communes à ces deux chemins, un mécanisme de détection du point de reconvergence permettant

entre autres de calculer la taille des vides à insérer et un mécanisme d'étiquetage des résultats permettant d'invalider les résultats indépendants du contrôle mais pas des données et donc non réutilisables sur le bon chemin. Une contribution annexe du mécanisme de détection du point de reconvergence est la possibilité d'exploiter le résultat des branchements calculés sur le mauvais chemin pour améliorer la qualité de la prédiction de branchements sur le bon chemin.

L'élimination des branchements par la transformation de *if-conversion* permet d'obtenir des instructions prédiquées à la place des branchements et des instructions qu'ils contrôlaient. Cependant, l'exécution des instructions prédiquées sur un processeur superscalaire à exécution dans le désordre pose plusieurs problèmes, dont le principal est celui des définitions multiples. Il se pose lorsqu'une instruction prédiquée est renommée, puisque le mécanisme de renommage ne peut pas s'assurer que cette instruction écrira bien dans son registre destination R tant que la valeur du prédicat n'est pas connue. Cela provoque donc une incertitude sur le registre physique associé au registre architectural R : si le prédicat est vrai, ce sera le registre physique P qui a été été attribué à l'instruction prédiquée, si le prédicat est faux, cela restera l'ancien registre physique, celui attribué à l'instruction définissant précédemment R. Le renommage d'une instruction utilisant R en tant que registre source n'est donc pas possible tant que cette incertitude n'est pas levée. Une manière simple de résoudre ce problème est de transformer cette instruction prédiquée en instruction *false predicate conditional move* (FPCMOV). Sous cette forme, si son prédicat est vrai, l'instruction s'exécute normalement, écrivant son résultat dans son registre physique destination. Si son prédicat est faux, l'ancienne valeur de R, contenue dans le registre physique destination de l'instruction définissant précédemment R, est copiée dans son registre physique destination. Ainsi, le registre physique destination de l'instruction prédiquée contient toujours la bonne valeur sémantique. C'est donc toujours ce registre physique qu'il faut associer au registre architectural, quelle que soit la valeur du prédicat de l'instruction prédiquée. Cette solution introduit cependant de nouvelles dépendances entre les instructions. En effet, sous cette forme, l'instruction prédiquée devient dépendante de l'instruction définissant précédemment son registre destination, ce qui revient à perdre le bénéfice du renommage des registres pour ces instructions et induit une séquentialisation de l'exécution des instructions concernées. Notre proposition SPREPI résout ces problèmes par différents mécanismes complémentaires. La prédiction de prédicats permet de résoudre le problème des définitions multiples tout en évitant la séquentialisation de l'exécution des instructions prédiquées. La sélection de l'utilisation de la prédiction permet de ne prédire que les prédicats qui ne sont pas difficiles à prédire, évitant ainsi les mauvaises prédictions. Enfin, pour limiter le coût des quelques mauvaises prédictions restantes, un mécanisme de rejeu sélectif est utilisé pour ne réexécuter que les instructions mal prédites et celles qui en sont dépendantes au lieu de réexécuter toutes les instructions après celles mal prédites.

Nos deux propositions architecturales participent donc à l'amélioration de la performance de traitement des branchements et ainsi à l'amélioration des performances séquentielles à l'ère des processeurs massivement multicœurs.

Perspectives

Cette thèse a contribué à l'augmentation des performances séquentielles des processeurs à exécution dans le désordre en proposant des mécanismes permettant l'amélioration de la prise en charge de l'exécution des instructions de branchement. Il reste cependant d'importants efforts de recherche à fournir dans le domaine et de nombreuses pistes à explorer.

Il est intéressant de constater que nos propositions, bien que s'orientant chacune vers des domaines relativement différents pourraient être rassemblées en un seul mécanisme où les deux chemins d'un branchement difficile à prédire seraient traités en même temps, transformés dynamiquement en instructions prédiquées et exécutées en tant que tel par le processeur. Un tel fonctionnement est décrit dans [29], mais la détection des branchements qui peuvent être traités ainsi se fait par le compilateur. En utilisant la détection de la reconvergence de SYRANT, il nous semble possible de faire cette détection dynamiquement. Les instructions prédiquées ainsi créées viendraient remplacer les vides qui sont insérés par SYRANT, tout en obtenant le même résultat : allouer les mêmes ressources aux instructions indépendantes du contrôle.

On obtient ainsi un mécanisme qui permet d'une part de traiter les branchements difficiles à prédire, contrairement à la *if-conversion* qui transforme parfois des branchements faciles à prédire, et d'autre part d'exploiter l'indépendance de contrôle, opération facilitée par les avantages de la prédication, pour éviter de trop grosses pertes en cas de mauvaise prédiction.

Publications personnelles

[PS12] Nathanaël Prémillieu et André Seznec, « Syrant : Symmetric resource allocation on not-taken and taken paths », *ACM Trans. Archit. Code Optim.*, vol. 8, n° 4, p. 43 :1–43 :20, jan. 2012.

[PS13] Nathanaël Prémillieu et André Seznec, « SPREPI : Selective Prediction and REplay for Predicated Instructions », Rapport de recherche RR-8351, INRIA, août 2013.

Bibliographie

[1] Haitham Akkary, Srikanth T. Srinivasan, and Konrad Lai. Recycling waste : exploiting wrong-path execution to improve branch prediction. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 12–21, 2003.

[2] Ahmed S. Al-Zawawi, Vimal K. Reddy, Eric Rotenberg, and Haitham Akkary. Transparent control independence (tci). In *ISCA*, pages 448–459, 2007.

[3] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Comput. Surv.*, 27(3) :367–432, September 1995.

[4] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189, 1983.

[5] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring) : Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[6] ARM. Arm architecture reference manual. arm v7-a and arm v7-r edition.

[7] David I. August, Wen-mei W. Hwu, and Scott A. Mahlke. A framework for balancing control flow and predication. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 92–103, Washington, DC, USA, 1997. IEEE Computer Society.

[8] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar : An infrastructure for computer system modeling. *Computer*, 35(2) :59–67, 2002.

[9] Fabrice Bellard. QEMU. *http://wiki.qemu.org/Main_Page*.

[10] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2) :1–7, August 2011.

[11] Po-Yung Chang, Eric Hao, Yale N. Patt, and Pohua P. Chang. Using predicated execution to improve the performance of a dynamically scheduled machine with speculative execution. *International Journal of Parallel Programming*, 24(3) :209–234, 1996.

[12] Chen-Yong Cher and T. N. Vijaykumar. Skipper : a microarchitecture for exploiting control-flow independence. In *MICRO*, pages 4–15, 2001.

[13] Warren Cheung, William S. Evans, and Jeremy Moses. Predicated instructions for code compaction. In *SCOPES*, pages 17–32, 2003.

[14] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *ISCA '98 : Proceedings of the 25th annual international symposium on Computer architecture*, pages 142–153, 1998.

[15] Weihaw Chuang and Brad Calder. Predicate prediction for efficient out-of-order execution. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 183–192, 2003.

[16] Jamison D. Collins, Dean M. Tullsen, and Hong Wang. Control flow optimization via dynamic reconvergence prediction. In *MICRO*, pages 129–140, 2004.

[17] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4) :451–490, 1991.

[18] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt. Overlapped loop support in the cydra 5. *SIGARCH Comput. Archit. News*, 17(2) :26–38, April 1989.

[19] Karel Driesen and Urs Hölzle. Accurate indirect branch prediction. In *Proceedings of the 25th annual international symposium on Computer architecture*, pages 167–178, 1998.

[20] Amit Gandhi, Haitham Akkary, and Srikanth T. Srinivasan. Reducing branch misprediction penalty via selective branch recovery. In *HPCA*, pages 254–264, 2004.

[21] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0 : Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, vol. 7, September 2005.

[22] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition : A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 2006.

[23] James Henry Hesson, Jay LeBlanc, and Stephen J. Ciavaglia. Apparatus to dynamically control the out-of-order execution of load/store instructions in a processor capable of dispatching, issuing and executing multiple instructions in a single processor cycle. US Patent 5,615,350, March 1997.

[24] Andrew D. Hilton and Amir Roth. Ginger : control independence using tag rewriting. In *ISCA*, pages 436–447, 2007.

[25] Intel Corp. Intel itanium architecture software developer's manual. volume 3 : Instruction set reference. 2002.

[26] Erik Jacobsen, Eric Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *MICRO 29 : Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 142–152, Washington, DC, USA, 1996. IEEE Computer Society.

[27] Stéphan Jourdan, Pascal Sainrat, and Daniel Litaize. Exploring configurations of functional units in an out-of-order superscalar processor. In *ISCA*, pages 117–125, 1995.

[28] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 19(2) :24–36, 1999.

[29] Hyesoon Kim, Jose A. Joao, Onur Mutlu, and Yale N. Patt. Diverge-merge processor (dmp) : Dynamic predicated execution of complex control-flow graphs based on frequently executed paths. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 53–64, 2006.

[30] Hyesoon Kim, Onur Mutlu, Jared Stark, and Yale N. Patt. Wish branches : Combining conditional branching and predication for adaptive predicated execution. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 43–54, 2005.

[31] M. Lam. Software pipelining : an effective scheduling technique for vliw machines. *SIGPLAN Not.*, 23(7) :318–328, June 1988.

[32] Chang Joo Lee, Hyesoon Kim, Onur Mutlu, and Yale N. Patt. Performance-aware speculation control using wrong path usefulness prediction. In *HPCA*, pages 39–49, 2008.

[33] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench : a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 330–335, 1997.

[34] Scott A. Mahlke, Richard E. Hank, Roger A. Bringmann, John C. Gyllenhaal, David M. Gallagher, and Wen-mei W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 217–227, 1994.

[35] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen-Mei W. Hwu. A comparison of full and partial predicated execution support for ilp processors. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 138–150.

[36] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th annual international symposium on Microarchitecture*, pages 45–54, 1992.

[37] E. J. McLellan and D. A. Webb. The Alpha 21264 microprocessor architecture. In *ICCD '98 : Proceedings of the International Conference on Computer Design*, page 90, 1998.

[38] Andreas Moshovos and Gurindar S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *MICRO 30 : Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 235–245, Washington, DC, USA, 1997. IEEE Computer Society.

[39] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and branch prediction in dynamic ilp processors. In *Proceedings of the 21st annual international symposium on Computer architecture*, ISCA '94, pages 120–129, 1994.

[40] Nathanael Premillieu and Andre Seznec. Syrant : Symmetric resource allocation on not-taken and taken paths. *ACM Trans. Archit. Code Optim.*, 8(4) :43 :1–43 :20, January 2012.

[41] Eduardo Quiñones, Joan-Manuel Parcerisa, and Antonio Gonzalez. Selective predicate prediction for out-of-order processors. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 46–54, 2006.

[42] Eduardo Quiñones, Joan-Manuel Parcerisa, and Antonio González. Improving branch prediction and predicated execution in out-of-order processors. In *HPCA*, pages 75–84, 2007.

[43] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th annual workshop on Microprogramming*, MICRO 14, pages 183–198, 1981.

[44] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and James E. Smith. Trace processors. In *MICRO*, pages 138–148, 1997.

[45] Eric Rotenberg, Quinn Jacobson, and James E. Smith. A study of control independence in superscalar processors. In *HPCA*, pages 115–124, 1999.

[46] Amir Roth and Gurindar S. Sohi. Register integration : a simple and efficient implementation of squash reuse. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 223–234, 2000.

[47] John Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown : optimal vs. heuristic methods in a production compiler. *SIGPLAN Not.*, 31(5) :1–11, May 1996.

[48] M.S. Schlansker and B.R. Rau. Epic : Explicitly parallel instruction computing. *IEEE Computer*, 33(2) :37–45, 2000.

[49] André Seznec. A 64-Kbytes ITTAGE indirect branch predictor. In *JWAC-2 : Championship Branch Prediction*, June 2011.

[50] André Seznec. A new case for the tage branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 117–127, 2011.

[51] Andre Seznec. Storage free confidence estimation for the tage branch predictor. In *Proceedings of the 17th International Symposium on High-Performance Computer Architecture*, pages 443–454, 2011.

[52] André Seznec. The L-TAGE branch predictor. *Journal of Instruction Level Parallelism*, May 2007.

[53] André Seznec. A 64 kbytes ISL-TAGE branch predictor. *JWAC-2 : Championship Branch Prediction*, June 2011.

[54] André Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *Journal of Instruction Level Parallelism*, February 2006.

[55] Beth Simon, Brad Calder, and Jeanne Ferrante. Incorporating predicate information into branch predictors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 53–64, 2003.

[56] James E. Smith and Gurindar S. Sohi. The microarchitecture of superscalar processors. In *Proceedings of the IEEE*, volume 82, pages 1609–1624, 1995.

[57] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *ISCA*, pages 194–205, 1997.

[58] SPEC. SPEC CPU2000. *http://www.spec.org/cpu2000/*, 2000.

[59] SPEC. SPEC CPU2006. *http://www.spec.org/cpu2006/*, 2006.

[60] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, 11(1) :25–33, January 1967.

[61] Hans Vandierendonck and André Seznec. Speculative return address stack management revisited. *ACM Trans. Archit. Code Optim.*, 5(3) :15 :1–15 :20, December 2008.

[62] Perry H. Wang, Hong Wang, Ralph-Michael Kling, Kalpana Ramakrishnan, and John Paul Shen. Register renaming and scheduling for dynamic execution of predicated code. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 15–25, 2001.

[63] Nancy J. Warter, Daniel M. Lavery, and Wen mei W. Hwu. The benefit of predicated execution for software pipelining. In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, pages 497–506, 1993.

[64] Nancy J. Warter, Scott A. Mahlke, Wen-Mei W. Hwu, and B. Ramakrishna Rau. Reverse if-conversion. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 290–299, 1993.

[65] T. Wolf and M. Franklin. Commbench-a telecommunications benchmark for network processors. In *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '00, pages 154–162, 2000.

Résumé

L'omniprésence des ordinateurs et la demande de toujours plus de puissance poussent les architectes processeur à chercher des moyens d'augmenter les performances de ces processeurs. La tendance actuelle est de répliquer sur une même puce plusieurs cœurs d'exécution pour paralléliser l'exécution. Si elle se poursuit, les processeurs deviendront massivement multicœurs avec plusieurs centaines voire un millier de cœurs disponibles. Cependant, la loi d'Amdahl nous rappelle que l'augmentation de la performance séquentielle sera toujours nécessaire pour améliorer les performances globales.

Une voie essentielle pour accroître la performance séquentielle est de perfectionner le traitement des branchements, ceux-ci limitant le parallélisme d'instructions. La prédiction de branchements est la solution la plus étudiée, dont l'intérêt dépend essentiellement de la précision du prédicteur. Au cours des dernières années, cette précision a été continuellement améliorée et a atteint un seuil qu'il semble difficile de dépasser. Une autre solution est d'éliminer les branchements et de les remplacer par une construction reposant sur des instructions prédiquées. L'exécution des instructions prédiquées pose cependant plusieurs problèmes dans les processeurs à exécution dans le désordre, en particulier celui des définitions multiples.

Les travaux présentés dans cette thèse explorent ces deux aspects du traitement des branchements. La première partie s'intéresse à la prédiction de branchements. Une solution pour améliorer celle-ci sans augmenter la précision est de réduire le coût d'une mauvaise prédiction. Cela est possible en exploitant la reconvergence du flot de contrôle et l'indépendance de contrôle pour récupérer une partie du travail fait par le processeur sur le mauvais chemin sur les instructions communes aux deux chemins pour éviter de le refaire sur le bon chemin. La deuxième partie s'intéresse aux instructions prédiquées. Nous proposons une solution au problème des définitions multiples qui passe par la prédiction sélective de la valeur des prédicats. Un mécanisme de rejeu sélectif est utilisé pour réduire le coût d'une mauvaise prédiction de prédicat.

Abstract

Computers are everywhere and the need for always more computation power has pushed the processor architects to find new ways to increase performance. The today's tendency is to replicate execution core on the same die to parallelize the execution. If it goes on, processors will become manycores featuring hundred to a thousand cores. However, Amdahl's law reminds us that increasing the sequential performance will always be vital to increase global performance.

A perfect way to increase sequential performance is to improve how branches are executed because they limit instruction level parallelism. Branch prediction is the most studied solution, its interest greatly depending on its accuracy. In the last years, this accuracy has been continuously improved up to reach a hardly exceeding limit. An other solution is to suppress the branches by replacing them with a construct based on predicated instructions. However, the execution of predicated instructions on out-of-order processors comes up with several problems like the multiple definition problem.

This study investigates these two aspects of the branch treatment. The first part is about branch prediction. A way to improve it without increasing the accuracy is to reduce the cost of a branch misprediction. This is possible by exploiting control flow reconvergence and control independence. The work done on the wrong path on instructions common to the two paths is saved to be reused on the correct path. The second part is about predicated instructions. We propose a solution to the multiple definition problem by selectively predicting the predicate values. A selective replay mechanism is used to reduce the cost of a predicate misprediction.