Accélération matérielle pour la traduction dynamique de programmes binaires

Simon Rokicki

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE **U**NIVERSITE **B**RETAGNE **L**OIRE

Ecole Doctorale N°601
Mathématique et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par

« Simon ROKICKI »

« Accélération matérielle pour la traduction dynamique de programmes binaires »

Thèse présentée et soutenue à RENNES , le 17 Décembre 2018
Unité de recherche : Irisa
Thèse N° :

Rapporteurs avant soutenance :

Fréderic Petrot - Professeur - Polytech Grenoble - Université Grenoble Alpes, Grenoble
Henri-Pierre Charles - Directeur de Recherche - CEA, Grenoble

Composition du jury :

Président :	Olivier Sentieys - Professeur - Université de Rennes 1
Examinateurs :	Karine Heydemann - Maître de Conférence - UPMC
	Fréderic Pétrot - Professeur - Ensimag Grenoble INP, Grenoble
	Henri-Pierre Charles - Directeur de Recherche - CEA, Grenoble
	Sylvain Collange - Chargé de Recherche - Inria Rennes
	Tanguy Risset - Professeur - INSA Lyon
Dir. de thèse :	Steven Derrien - Professeur - Université de Rennes 1
Co-dir. de thèse :	Erven Rohou - Directeur de Recherche - Inria Rennes

TABLE OF CONTENTS

Introduction

Des téléphones aux centres de calcul, les systèmes informatiques jouent aujourd'hui un rôle majeur dans la plupart des activités humaines. On les retrouve dans des contextes variés, dont la diversité reflète celle de leurs applications. En première approche, on peut distinguer trois types de système informatique : les systèmes généralistes qui correspondent aux ordinateurs utilisés dans la vie de tous les jours, les systèmes embarqués comme les smartphones, ou les supercalculateurs. Malgré leurs différences, le fonctionnement de tous ces systèmes informatiques est basé sur l'utilisation d'un processeur, qui réalise les calculs, et de la mémoire, qui permet de stocker les données. La puissance d'un tel système correspond à la vitesse à laquelle le processeur peut calculer.

Entre les années 1960 et 2010, le développement des processeurs est dicté par des progrès technologiques rapides. A cette époque, ces progrès suivent les lois de Moore et de Dennard et la puissance des ordinateurs double tous les deux ans. Cette avancée technologique est maintenant en très net ralentissement : les spécialistes estiment qu'au rythme actuel, il faut attendre 20 ans pour observer un doublement de la performance des processeurs. Ce ralentissement oblige les fabricants à repenser l'architecture des processeurs et à se tourner vers la spécialisation. Développer des processeurs spécialisés permet d'obtenir de meilleures performances sur un champ d'applications précis.

Le nombre de systèmes informatiques embarqués (par exemple les smartphones) a également explosé ces 15 dernières années. L'émergence de l'internet des objets laisse à penser que cette augmentation va croitre encore plus rapidement dans les années à venir. Les systèmes embarqués se caractérisent par de fortes contraintes en termes de consommation énergétique et de performance. Lorsqu'un fabricant de circuit développe un processeur destiné à un système généraliste (machine de bureau, serveur, etc.), il cherche à maximiser la performance obtenue dans une enveloppe thermique donnée. Au contraire, pour un processeur embarqué, les fabricants cherchent à minimiser l'énergie consommée pour atteindre le niveau de performance nécessaire.

Ces systèmes embarqués se caractérisent également par le très large spectre d'applications qu'ils doivent exécuter. Par exemple, sur un smartphone, certaines applications requièrent un très haut niveau de performances, tandis que d'autres doivent pouvoir être exécutées en consommant le moins d'énergie possible. De fait, l'architecture matérielle doit offrir au système d'exploitation, ou au développeur, la possibilité de changer dynamiquement le compromis entre la performance et la consommation énergétique. Depuis les années 2000, cette adaptation dynamique a été réalisée grâce au changement dynamique de la tension d'alimentation et de la fréquence de fonctionnement du processeur (DVFS pour *Dynamic Voltage and Frequency Scaling*). Toutefois, la DVFS est de moins en moins intéressante sur les nouvelles générations de transistors à cause de la fin de la loi de Dennard. En effet, la hausse de performance se fait au prix d'une plus grande hausse de la consommation énergétique ; la baisse de performance a un impact plus faible sur la consommation énergétique.

Pour offrir plus de compromis entre la performance et la consommation énergétique, les fabricants de processeurs ont mis au point des systèmes multi-cœurs hétérogènes. Le principe de ces systèmes est de proposer différents types de processeurs intégrés sur un même circuit. Le compromis entre la performance et la consommation énergétique peut donc être géré en choisissant le processeur qui exécute toute (ou une partie de) l'application. L'architecture big.LITTLE de ARM est un exemple de plateforme

multi-cœurs hétérogène. Celle-ci embarque sur un même circuit des processeurs faible-consommation (*LITTLE*) et des processeurs haute performance (*big*). Les processeurs *big* et les processeurs *LITTLE* sont capables d'exécuter les mêmes programmes car ils supportent le même jeu d'instructions ARM. Cette compatibilité binaire est un aspect fondamental des systèmes big.LITTLE. En effet, celle-ci permet au système d'exploitation de migrer de façon quasi-transparente une application d'un type de processeur à un autre.

Pour des sytèmes multi-cœurs hétérogènes, le besoin de compatibilité binaire limite le choix des processeurs pouvant être utilisés. Pourtant, ces systèmes gagneraient à embarquer d'autres types de processeurs (par exemple les processeurs *Very Long Instruction Word*, ou VLIW). Ceux-ci permettent, pour certains types d'applications, d'atteindre un haut niveau de performance, tout en maintenant une consommation énergétique faible. Sur certaines applications, un processeur VLIW peut donc être plus rapide et consommer moins qu'un processeur *big* utilisé dans les big.LITTLE actuels. Les processeurs VLIW reposent cependant sur une architecture de jeu d'instructions explicitement parallèle, qui ne peut être rendue compatible avec un jeu d'instructions standard (ARM, x86, etc.), tel qu'utilisé dans les systèmes de type big.LITTLE.

La traduction dynamique de binaires (DBT pour *Dynamic Binary Translation*) permet de s'affranchir de cette limitation. Le principe de la DBT est d'exécuter une application compilée pour un processeur basé sur un jeu d'instructions A sur un processeur basé sur un jeu d'instructions B. Lors de l'exécution, l'outil de DBT traduit les instructions de l'application en un équivalent compatible avec le processeur B. Cette traduction augmente toutefois le temps d'exécution, puisqu'il faut traduire chaque instruction avant de pouvoir l'exécuter.

L'intérêt de la DBT dans un système hétérogène est de permettre l'intégration de processeurs VLIW tout en offrant l'illusion d'une compatibilité binaire. Quand une application est migrée vers le processeur VLIW, celle-ci est traduite par l'outil de DBT. Le surcoût lié à la traduction impacte cependant les performances et la consommation énergétique du processeur VLIW, ce qui réduit son efficacité énergétique.

Il existe plusieurs outils de DBT permettant d'exécuter des applications sur un processeur VLIW. Par exemple, le *Code Morphing Software* de Transmeta [25] ou l'architecture Denver de NVidia [12]. Dans les deux cas, l'objectif est de contourner la limitation du jeu d'instructions et d'exploiter l'efficacité énergétique des processeurs VLIW. Il n'existe cependant que très peu d'informations sur le fonctionnement de ces deux outils commerciaux. A notre connaissance, il n'existe pas d'infrastructure de DBT pour VLIW ouverte pouvant être utilisée dans le cadre de travaux de recherche sur ce sujet.

Contributions

Les travaux de cette thèse portent sur l'utilisation d'accélarateurs matériels pour améliorer l'efficacité des approches à base de DBT. Nous avons proposé et réalisé un flot complet de DBT, basé sur un partitionement logiciel/matériel des différentes étapes de la traduction. Ainsi, des accélérateurs matériels ont été développés pour réduire le coût des étapes critiques du flot. Pour chacune des transformations à accélérer, nous avons étudié différents algorithmes et différentes représentations des données afin de mettre au point un accélérateur performant et peu coûteux. Ces composants matériels permettent de réduire la durée d'exécution et le coût énergétique des étapes accélérées par deux ordres de grandeurs.

Ces travaux sont la contribution majeur de cette thèse et sont présentés en détail dans le chapitre 3 de ce document.

Nos travaux ont également abordé la gestion des processeurs VLIW dynamiquement reconfigurable. Ceux-ci sont capables d'activer ou de désactiver des unités d'exécution pour s'adapter à l'application en cours d'exécution. Dans ce type de machine, une application compilée pour une configuration donnée ne peut pas être exécutée sur une autre. Afin de lever cette limitation, nous avons modifié notre flot de traduction afin que celui-ci puisse gérer les 24 configurations possibles de notre processeur VLIW de référence. Pour chaque sous-programme d'une application, l'outil d'optimisation explore les différentes configurations du VLIW pour déterminer les plus intéressantes.

Dans le but d'augmenter les performances du VLIW, nous avons également développé un système de spéculation de dépendances mémoire. Le principe de l'approche est d'ignorer certaines dépendances entre des lectures et des écritures mémoire afin de pouvoir extraire plus de parallélisme d'instructions. Au cours de l'exécution, un mécanisme matériel vérifie que ces accès mémoire accèdent effectivement à des adresses différentes. Dans ce cas, le processeur doit annuler et ré-exécuter la lecture mémoire ainsi que toutes les instructions ayant utilisé la valeur lue. Le flot d'optimisation dynamique permet de profiler les accès mémoire et de choisir précisemment les endroits où la spéculation est bénéfique. L'approche logicielle/matérielle que nous avons utilisée permet de profiter de la spéculation de dépendance mémoire tout en réduisant grandement leur coût matériel.

Ces deux flots d'optimisation continue sont présentés dans le chapitre 4 de ce document.

Le flot de traduction et d'optimisation mis au point est présenté sur la figure 1. Il est organisé en trois étapes :

— Dans le niveau d'optimisation 0, les instructions RISC-V sont traduites en instructions VLIW. L'outil de DBT ne cherche pas à exploiter le parallélisme d'instructions disponible.
— Dans le niveau d'optimisation 1, les binaires sont optimisés à l'échelle des blocs de base. L'outil de DBT construit une représentation intermédiaire de plus haut-niveau que les binaires. Cette représentation est utilisée pour réaliser un ordonnancement des instructions sur les unités d'exécution du VLIW.
— Dans le niveau d'optimisation 2, l'outil de DBT réalise des optimisations à l'échelle de la procédure, puis ordonnance les instructions pour générer les binaires VLIW.

Ce flot de traduction et d'optimisation repose sur trois accélérateurs matériels permettant de réduire le coût de la DBT de plusieurs ordres de magnitude. Ces accélérateurs permettent de réaliser la première traduction, de construire la représentation intermédiaire et d'ordonnancer les instructions.

Le niveau d'optimisation 2 permet également de gérer deux aspects d'optimisation dynamique : la gestion des VLIW dynamiquement reconfigurables et la gestion de la spéculation de dépendances mémoire.

Plan du document

Ce document est organisé en quatre chapitres. Dans le premier chapitre, nous présentons en détail l'importance de l'adaptation dynamique dans le systèmes embarqués et les principales techniques pour y arriver. Nous étudions également les principaux types de processeurs utilisés dans ces systèmes.

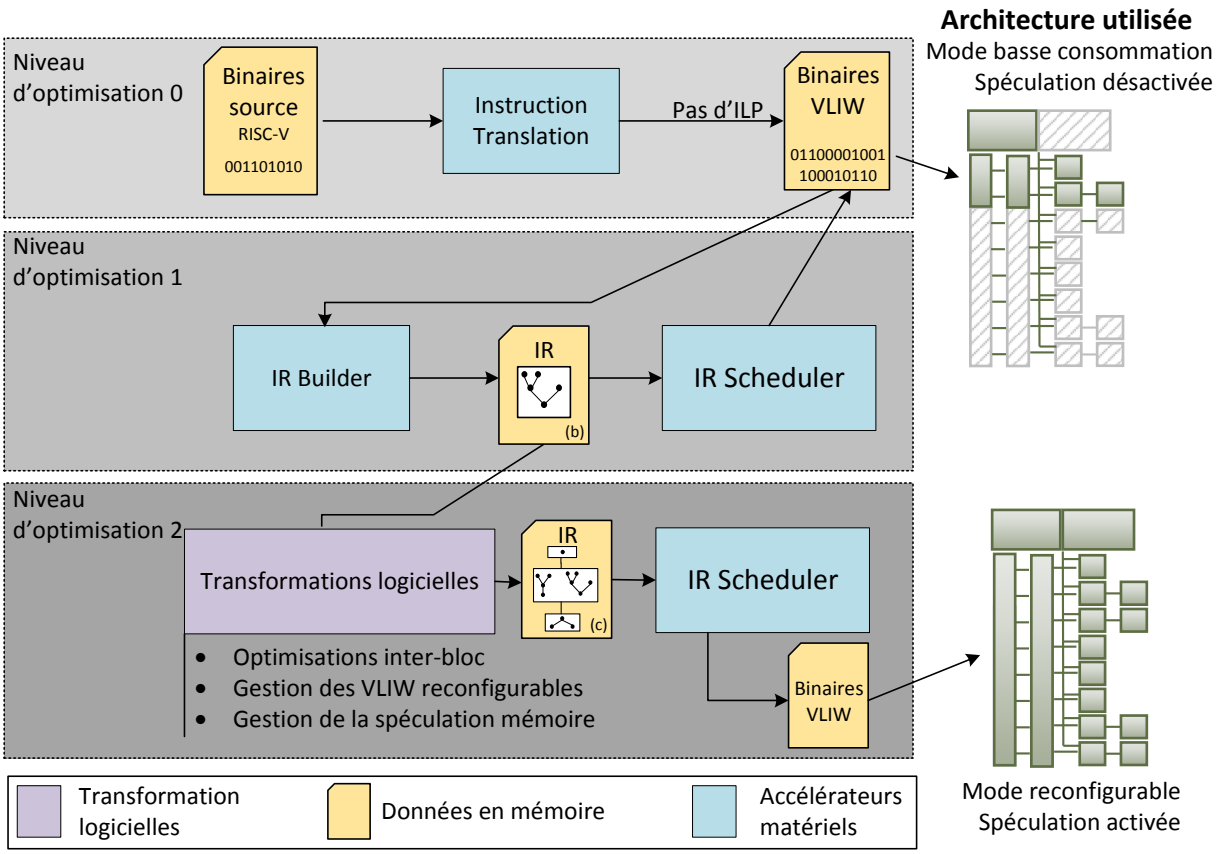


FIGURE 1 – Présentation du flot de traduction et d'optimisation de Hybrid-DBT. Ce flot s'appuie sur trois accélérateurs matériels pour réaliser les phases critiques de la traduction : la première traduction, la construction d'une représentation intermédiaire et l'ordonnancement des instructions. Le niveau d'optimisation 2 fait également intervenir des transformations logicielles, dont la gestion de processeur VLIW dynamiquement reconfigurable et la gestion de la spéculation de dépendances mémoire.

Dans le chapitre 2, nous présentons le principe de la compilation dynamique et plus particulièrement de la traduction dynamique de binaires. Le chapitre 3 est consacré à l'étude du fonctionnement de Hybrid-DBT, l'outil de traduction dynamique de binaires développé pendant cette thèse. Nous étudions en détail le flot de traduction et d'optimisation ainsi que les accélérateurs mis au point. Dans le chapitre 4, nous présentons les deux approches d'optimisation dynamique que nous avons mises au point : la gestion des VLIW dynamiquement reconfigurables et la gestion de la spéculation de dépendances mémoire.

CHAPITRE 1

ARCHITECTURE DES PROCESSEURS

Trois facteurs impactent le développement des architectures de processeurs : la technologie, l'architecture du jeu d'instructions et la micro-architecture.

La **technologie** des transistors a beaucoup évolué depuis les années 1960. Cette évolution suit deux lois empiriques : la loi de Moore et la loi de Dennard. La loi de Moore prédit une évolution exponentielle du nombre de transistors pouvant être intégrés dans un circuit. Ces transistors étant de plus en plus petits, le temps de propagation du signal diminue et la fréquence de fonctionnement des circuits peut être augmentée. La loi de Dennard prédit, quant à elle, que la densité de puissance des transistors n'augmente pas d'une génération à l'autre.

Jusqu'aux années 2000, ces évolutions technologiques ont été déterminantes dans le développement des processeurs. En effet, celles-ci ont permi d'obtenir régulièrement de nouvelles générations de processeurs plus rapides, et moins énergivores.

A partir des années 2000, les fabricants de circuits intégrés se sont heurtés à une barrière technologique, surnommée *power wall*. La puissance thermique dissipée à la surface d'une puce est devenue trop importante, entrainant un risque pour le bon fonctionnement de la puce. Cette barrière technologique correspond à la fin de la loi de Dennard : les avancées ne permettent plus de réduire la tension d'alimentation des transistors et d'augmenter, en même temps, la fréquence de fonctionnement des nouvelles générations de processeurs. Cette difficulté a freiné l'augmentation des performances *single-thread* des processeurs. En réponse à ces nouvelles contraintes et pour exploiter le nombre croissant de transistors disponibles, les fabricants de circuits se sont tournés vers des architectures multi-cœurs.

La fin de la loi de Dennard a également mené à un deuxième phénomène, qui a pris le nom de *utilization wall* ou encore *dark silicon*. L'intégralité des transistors présents sur une puce ne peut plus être utilisée à un niveau de performance maximal : ou bien une partie des transistors fonctionne à fréquence maximale, et le reste du circuit est désactivé, ou l'intégralité du système doit fonctionner à une fréquence réduite.

A cause de cet *utilization wall*, les fabricants de processeurs ont commencé à proposer des architectures multi-cœurs hétérogènes. Ces systèmes intègrent plusieurs types de processeurs, chacun offrant un compromis différent entre la performance et l'efficacité énergétique.

L'**architecture du jeu d'instructions** est l'interface entre le processeur et le compilateur et définit le modèle de programmation d'un processeur. Un changement de jeu d'instructions provoque l'impossibilité d'exécuter les applications compilées pour l'ancien jeu d'instructions. De ce fait, chaque marché de processeurs (par exemple l'informatique bureautique ou l'informatique embarquée) s'est organisé autour d'un jeu d'instructions dominant (le x86 ou le Arm). Par exemple, le x86 de Intel est utilisé dans la quasi-totalité des machines de bureau et serveurs, le jeu d'instructions ARM domine quant à lui le

marché de l'embarqué. Ces deux jeux d'instructions sont propriétaires : par exemple, si un fabricant souhaite commercialiser un processeur supportant le jeu d'instructions x86, il doit avoir l'autorisation de Intel [1]. Puisque ces jeux d'instructions sont propriétaires, nos travaux se sont basés sur le RISC-V, un jeu d'instructions libre de droits.

Enfin, la *micro-architecture* détermine comment le processeur exécute les instructions. Pour augmenter la performance, il est nécessaire d'exploiter le parallélisme d'instructions. Pour ce faire, les fabricants de processeurs ont développé des mécanismes complexes permettant, par exemple, d'analyser dynamiquement les instructions d'une application et de changer leur ordre d'exécution pour faire apparaitre plus de parallélisme d'instructions.

L'objectif de ce chapitre est d'appréhender et de motiver les travaux réalisés dans le cadre de cette thèse. Dans la section 1.1, nous commençons par discuter des spécificités des systèmes embarqués et de leur besoin d'adaptabilité. Dans la section 1.2, nous étudions les différentes micro-architectures utilisées dans les systèmes multi-cœurs hétérogènes en comparant leur performance et leur efficacité énergétique. Enfin, nous présentons les processeurs VLIW, leurs avantages et leurs inconvénients.

1.1 Gestion dynamique du compromis performance/énergie

Les systèmes embarqués se caractérisent par de fortes contraintes en performance, en coût de production et en consommation énergétique. Lorsqu'un fabricant développe un système de calcul généraliste, son objectif est de maximiser les performances tout en respectant une enveloppe thermique imposée. Dans le cas d'un système embarqué, l'objectif est de minimiser la consommation énergétique, tout en atteignant un niveau de performance imposé (par exemple la capacité de traiter un flux vidéo HD en temps réel). Les plateformes matérielles embarquées actuelles doivent également exécuter un large spectre d'applications. Par exemple, sur un téléphone portable, certaines applications requièrent de très hautes performances tandis que d'autres doivent être exécutées en consommant le moins d'énergie possible. En raison de ce large spectre, les fabricants de processeurs ont proposé des techniques permettant de contrôler dynamiquement le compromis entre la performance du processeur et sa consommation énergétique.

Afin de mieux appréhender ces compromis, nous commençons par rappeler les différentes sources de la dissipation de puissance dans un circuit VLSI. Cette dissipation est la combinaison de la puissance statique P_{stat} et de la puissance dynamique P_{dyna}. La puissance statique est liée aux courants de fuite dans les transistors. La puissance dynamique est liée à la puissance dissipée lors des changements d'état des transistors. Ces deux grandeurs physiques peuvent être approximées par les formules suivantes :

$$P_{dyna} = \alpha \times C \times V_{dd}^2 \times f \tag{1.1}$$

$$P_{stat} = V_{dd} \times I_{fuite} \tag{1.2}$$

où α correspond à la fréquence moyenne de basculement des transistors, C correspond à la capacitance

1. Actuellement, seul Transmeta et AMD ont obtenu cette autorisation, et Intel a racheté Transmeta depuis. Pour obtenir une license ARM, le fabricant doit payer plusieurs millions de dollars.

du circuit, I_{fuite} est le courant de fuite, f est la fréquence de fonctionnement de l'architecture et V_{dd} sa tension d'alimentation. Les paramètres C et I_{fuite} dépendent de la technologie utilisée, de la taille du circuit et du taux de transition des transistors (c'est-à-dire combien de fois ils changent d'état en un temps donné).

Dans la suite de cette section, nous présentons différentes approches permettant d'adapter dynamiquement le compromis entre la performance et la consommation énergétique d'un processeur. Nous commençons par présenter la technique de DVFS (modification dynamique de la fréquence et de la tension d'alimentation). Nous introduisons ensuite les approches à base de systèmes multi-cœurs hétérogènes.

1.1.1 L'utilisation de la DVFS et ses limites

La première technique étudiée est la modification dynamique de la fréquence et de la tension d'alimentation du processeur (*Dynamic Voltage and Frequency Scaling*, DVFS). Le principe est de modifier la fréquence de fonctionnement conjointement avec la tension d'alimentation du processeur afin de contrôler le compromis performance/consommation énergétique. Une augmentation de la tension d'alimentation d'un circuit accroit le temps de propagation du signal, ce qui permet au circuit de fonctionner avec une fréquence plus importante, au prix d'une augmentation de la puissance dissipée.

Les paramètres V_{dd} et f_{max} sont ainsi liés par la formule suivante :

$$T_{cc} = \frac{1}{f_{max}} = K \times C_{out} \times \frac{V_{dd}}{(V_{dd} - V_{th})^2} \tag{1.3}$$

dans laquelle K et C_{out} sont des facteurs dépendant de l'architecture et de la technologie et où V_{th} correspond à la tension de seuil du transistor (la tension à laquelle il change d'état). T_{cc} est le temps de propagation sur le chemin critique du circuit et f_{max} est la fréquence de fonctionnement maximale du circuit.

A titre d'exemple, Flatresse et al. [35] ont déterminé des couples de valeurs possibles pour V_{dd} et f_{max}, dans le cas d'un processeur ARM, pour une technologie 28 nm à basse tension de seuil (c'est à dire la technologie basse consommation). Nous avons utilisé ces valeurs pour modéliser l'impact de la DVFS sur la consommation d'énergie dynamique. Le résulat de cette modélisation est représenté sur la figure 1.1

Les résultats présentés sur la figure 1.1 peuvent être interprétés de deux manières différentes :

— le système d'exploitation décide d'augmenter les performances. Il peut augmenter la tension d'alimentation et la fréquence de fonctionnement du processeur. La performance du système augmente linéairement, au prix d'une augmentation quadratique de la consommation énergétique.

— le système d'exploitation décide de réduire l'énergie consommée. Il peut baisser la tension d'alimentation et la fréquence du processeur. L'énergie consommée diminue de façon quadratique, au prix d'une réduction linéaire des performances.

Cependant, les compromis possibles dépendent du ratio entre la puissance dynamique et la puissance statique d'une architecture, ainsi que de la possibilité de réduire la tension d'alimentation du processeur. Les évolutions technologiques tendent à augmenter la part de la puissance statique par

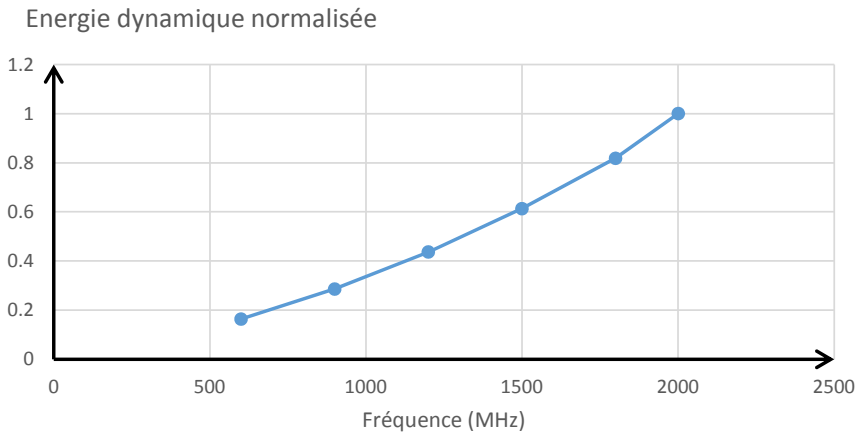


FIGURE 1.1 – Evolution de la consommation énergétique selon les différents niveaux de DVFS. Les couples de fréquence et de tension sont ceux définis dans [35] . Les valeurs sont normalisées selon l'énergie consommée à 2GHz.

rapport à la puissance dynamique. De plus, la tension d'alimentation devient de plus en plus proche de la tension de seuil ce qui réduit les possibilités de réduction de la tension par DVFS. Les gains dans les technologies les plus récentes sont donc moins intéressants [59].

1.1.2 Les systèmes multi-cœurs hétérogènes

Une plateforme multi-cœurs hétérogène regroupe plusieurs types de processeurs sur une même puce. Chacun de ces processeurs offre un niveau de performance et de consommation énergétique différent. Les processeurs varient de par leur capacité à extraire du parallélisme d'instructions, impactant ainsi leur performance. Toutefois, un processeur plus performant se traduit également par un coût (en transistors) plus élevé, ce qui à son tour impacte la consommation énergétique à travers les paramètres $\alpha, C, I_{fuite}, K, C_{out}$ vus précédemment.

Les systèmes hétérogènes les plus courants sont formés d'un processeur associé à un accélérateur graphique intégré. Dans ce type de machine, les deux architectures sont basées sur des modèles de programmation différents : l'un est séquentiel et l'autre est basé sur l'utilisation intensive de *threads*. De fait, la répartition des tâches entre le processeur et l'accélérateur graphique est en général réalisée statiquement, lors du développement ou lors de la compilation de l'application.

Pour changer le compromis entre la performance et la consommation énergétique, l'application exécutée doit être migrée depuis un type d'architecture (par exemple le processeur) vers un autre (par exemple l'accélérateur graphique). Puisqu'il n'y a pas de compatibilité binaire entre ces deux architectures, plusieurs versions de l'application doivent donc être embarquées.

Pour résoudre ce problème, Kumar et al. proposent, en 2003, l'utilisation de systèmes hétérogènes à jeu d'instructions unique [56, 55]. L'idée est d'utiliser un système multi-cœurs hétérogène dans lequel chaque cœur peut exécuter les mêmes instructions. Le choix de la micro-architecture utilisée est la seule différence entre les différents processeurs, celui-ci impactant la performance et la consommation énergétique. L'adaptation dynamique est alors possible puisqu'une application peut être migrée d'un

cœur à l'autre de manière transparente.

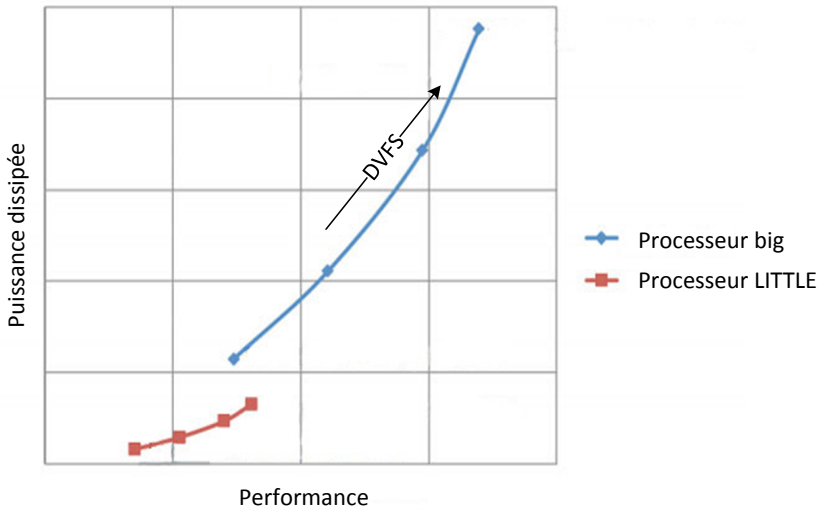


FIGURE 1.2 – Exemples de compromis atteignables grâce à l'utilisation combinée d'un système hétérogène et de la DVFS. Cette image est tirée d'un document ARM et n'est pas le résultat d'une étude expérimentale. Pour cette raison, aucune valeur ni aucune unité n'est donnée sur les axes.

L'architecture big.LITTLE de ARM est l'exemple le plus connu de système multi-cœurs hétérogène à jeu d'instructions unique [41]. Ce système intègre deux types de cœur sur un même circuit :

— les processeurs *big* sont des processeurs à exécution dans le désordre capables d'atteindre un niveau de performance élevé ;
— les processeurs *LITTLE* sont des processeurs superscalaires à exécution dans l'ordre qui permettent de calculer en dépensant peu d'énergie.

Pour mieux comprendre les différences entre ces deux types de processeurs, ces deux types de micro-architectures sont étudiées en détail dans la section 1.2 de ce document.

La figure 1.2 illustre les compromis offerts par l'architecture big.LITTLE. Chaque courbe représente les compromis atteignables avec un type de cœur, en utilisant la DVFS. L'hétérogénéité du système permet d'atteindre des points de fonctionnement que la DVFS seule ne pourrait pas offrir. Il existe de nombreuses implémentations de systèmes big.LITTLE combinant des types de processeurs différents. Certaines implémentations proposent même trois types de cœurs différents [61]. Un système multi-cœurs hétérogène peut également être constitué de cœurs basés sur la même micro-architecture. Par exemple Rangan et al. [74] ont proposé un système composé de plusieurs processeurs à exécution dans l'ordre, dont l'architecture a été optimisée pour une fréquence et une tension différentes. Le niveau de performance et la consommation énergétique sont donc spécifiques à chaque cœur.

Dans les approches que nous venons d'étudier, le système peut changer le compromis entre la performance et la consommation énergétique en migrant les applications d'un cœur à un autre. Or cette migration entraine un surcoût important : au-delà des coûts dus au changement de contexte, le processus continue son exécution sur un nouveau cœur, avec un autre cache et un autre prédicteur

de branchement. Ces deux mécanismes ont besoin de temps pour arriver à un état de fonctionnement efficace. Pour éviter ce surcoût important, certains travaux proposent d'utiliser des architectures de processeurs polymorphiques, c'est-à-dire capables de se modifier dynamiquement [73, 4, 7, 9, 16]. En d'autres termes, dans ces systèmes, les applications n'ont pas besoin d'être migrées d'un cœur à un autre, car c'est le cœur qui se reconfigure.

Dans les années 2000, plusieurs travaux ont exploré cette idée de processeurs polymorphiques. Cette technique permet par exemple de reconfigurer dynamiquement des processeurs à exécution dans le désordre [73, 4, 7, 9]. Brandon et al. l'ont également utilisée en proposant un VLIW capable de modifier dynamiquement le nombre de voies d'exécution [16]. Nous parlerons plus en détail de ces architectures dans le chapitre 4 de ce document qui présente nos propres travaux sur la question.

1.2 Présentation de différentes micro-architectures de processeur

Les systèmes multi-cœurs hétérogènes offrent, sur une même puce, différentes micro-architectures ayant différents niveaux de performance et de consommation énergétique. Migrer une tâche d'un type de cœur à un autre permet donc d'adapter dynamiquement l'efficacité énergétique du système. Pour mieux comprendre ces différents compromis, nous devons étudier de près le fonctionnement des différentes micro-architectures utilisées. Nous commençons par présenter le principe des processeurs superscalaires, puis décrivons le fonctionnement des processeurs à exécution dans le désordre.

Nous voyons ensuite que, si on s'autorise à changer le jeu d'instructions, il est possible d'utiliser des processeurs plus spécialisés, qui offrent une plus grande efficacité énergétique sur certains types de traitements, comme c'est le cas pour les processeurs VLIW.

1.2.1 Les micro-architectures dynamiquement ordonnancées

Pour mieux comprendre les compromis offerts par les architectures big.LITTLE, nous allons présenter le fonctionnement des micro-architectures utilisées dans cette plateforme. Les cœurs d'un système big.LITTLE reposent sur des micro-architectures superscalaires dynamiquement ordonnancées. Le principe de ces micro-architectures est d'exécuter un flot d'instructions séquentiel et de répartir dynamiquement les instructions sur les différentes unités d'exécution pour qu'elles soient exécutées en parallèle. Il existe plusieurs approches pour réorganiser ces instructions et extraire le parallélisme. Dans cette sous-section, nous présentons le fonctionnement des processeurs superscalaires à exécution dans l'ordre, puis nous introduisons les mécanismes d'exécution dans le désordre puis les mécanismes d'exécution spéculative.

Le parallélisme d'instructions et les architectures superscalaires

Pour augmenter la performance des processeurs et exploiter le nombre croissant de transistors disponibles sur une puce, les fabricants cherchent à exploiter le parallélisme d'instructions (ILP : *Instruction-Level Parallelism*). Le principe des processeurs superscalaires consiste à exécuter plusieurs instructions en même temps. Nous avons vu au début de ce chapitre que les jeux d'instructions ne peuvent pas être

modifiés facilement. Or, ceux qui sont utilisés majoritairement aujourd'hui (x86 et ARM) sont basés sur une exécution purement séquentielle. C'est donc la micro-architecture qui est chargée d'extraire et d'exploiter ce parallélisme à partir d'un flux (séquentiel) d'instructions à exécuter. Pour cette raison, ces architectures sont dites dynamiquement ordonnancées.

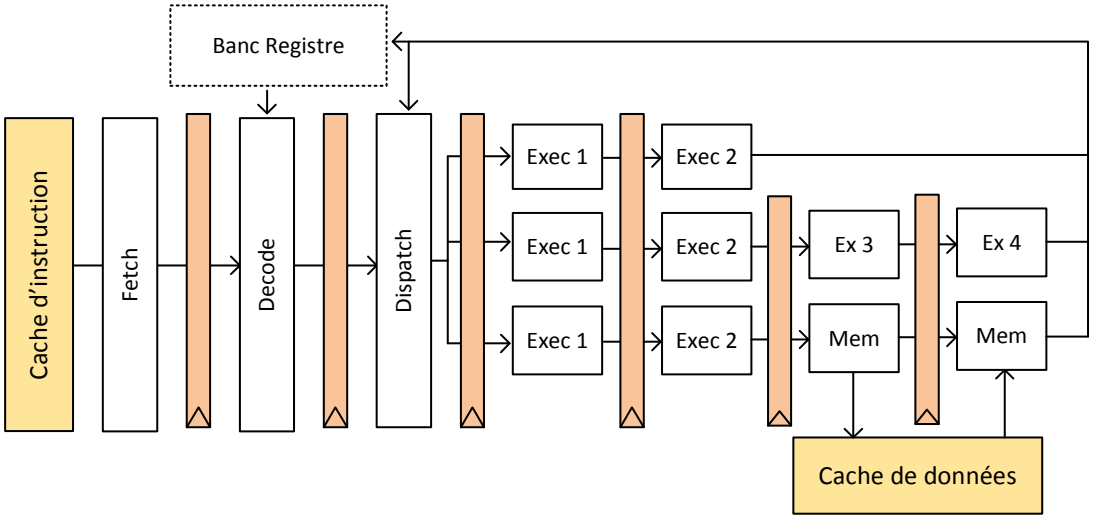


FIGURE 1.3 – Organisation simplifiée d'un processeur superscalaire à exécution dans l'ordre. Le pipeline est organisé en quatre étages, dont celui de *dispatch* en charge d'assigner les instructions à une unité d'exécution. Les unités d'exécution sont organisées en deux ou quatre étages de pipeline.

La micro-architecture superscalaire à exécution dans l'ordre est l'approche la plus simple pour exploiter du parallélisme d'instructions dans un flot séquentiel d'instructions. La figure 1.3 en propose une représentation simplifiée. Le processeur est organisé en quatre étages de pipeline : l'étage *Fetch* charge, à chaque cycle, une ou plusieurs instructions ; l'étage *Decode* décode l'instruction et lit la valeur des opérandes dans le banc de registres ; l'étage *dispatch*, qui est consacré à l'ordonnancement dynamique, détecte d'éventuelles dépendances (en comparant les registres utilisés par l'instruction entrante avec les registres modifiés par les instructions en cours d'exécution) et affecte l'instruction à une unité d'exécution ; celles-ci sont en charge de réaliser le calcul, ce qui peut nécessiter plus d'un cycle pour s'exécuter.

La figure 1.4 représente une trace d'exécution dans un processeur superscalaire à exécution dans l'ordre. Au cycle $t + 4$ de cette exécution, le processeur exécute quatre instructions en parallèle. Il est important de noter que les opérandes des instructions 7 et 8 sont disponibles dès le cycle $t + 5$ mais que l'exécution de ces instructions ne démarre qu'au cycle $t + 9$ à cause des contraintes d'ordonnancement dans l'ordre. Ainsi, le mécanisme de *Dispatch* permet d'exploiter le parallélisme d'instruction tout en s'assurant que les opérandes de l'instruction à exécuter sont prêts. Ce mécanisme ne peut cependant pas remettre en cause l'ordre des instructions : toute l'exécution peut ainsi être bloquée parce que l'exécution d'une instruction a besoin d'un résultat en cours de calcul, même si les instructions suivantes sont indépendantes. Dans la suite, nous étudions comment les mécanismes d'exécution dans le désordre permettent d'éviter ces blocages.

	t	t+1	t+2	t+3	t+4	t+5	t+6	t+7	t+8	t+9	t+10	t+11	t+12
1- addi r3 r5 5	F	DC	Di	E1	E2	WB							
2- addi r4 r6 16	F	DC	Di	E1	E2	WB							
3- mul r7 r9 r9		F	DC	Di	E1	E2	E3	E4	WB				
4- mul r8 r10 r10		F	DC	Di	E1	E2	E3	E4	WB				
5- ldw r12 15(r7)			F	DC	Di	Di	Di	Di	E1	E2	M1	M2	WB
6- ldw r13 0(r8)			F	DC	Di	Di	Di	Di	E1	E2	M1	M2	WB
7- andi r7 r3 r12				F	DC	Di	Di	Di	Di	E1	E2	WB	
8- andi r8 r4 0xfff				F	DC	Di	Di	Di	Di	E1	E2	WB	

FIGURE 1.4 – Exemple de trace d'exécution sur un processeur superscalaire à exécution dans l'ordre : le processeur utilisé peut charger deux instructions par cycle, exécute les opérations arithmétiques simples en deux cycles et les opérations arithmétiques complexes en 4 cycles. Sur la trace d'exécution, l'étage *Fetch* charge deux instructions à chaque cycle. Les instructions 3 et 4 commencent leur exécution dès le cycle $t + 4$ car ils ne dépendent pas des instructions 1 et 2. Cependant, l'exécution des instructions 5 et 6 utilise les résultats des instructions 3 et 4. Par conséquent, les instructions 5 et 6 sont bloquées dans l'étage de *Dispatch* jusqu'au cycle $t + 8$. Les opérandes des instructions 7 et 8 sont prêts dès le cycle $t + 5$ mais à cause de l'ordonnancement dans l'ordre, leur exécution ne commence qu'au cycle $t + 9$. Ils terminent toutefois leur exécution avant les instructions 5 et 6.

Exécution dans le désordre : l'approche de Tomasulo

Pour éviter les blocages liés à l'exécution des instructions dans l'ordre, les fabricants de processeurs ont mis en place des mécanismes d'exécution dans le désordre. L'approche de Tomasulo est l'un de ces mécanismes [86]. Elle a été utilisée pour la première fois dans l'unité flotante du IBM 360/91, en 1967. Tomasulo ajoute deux nouvelles idées : le renommage des registres et l'utilisation de files d'attente.

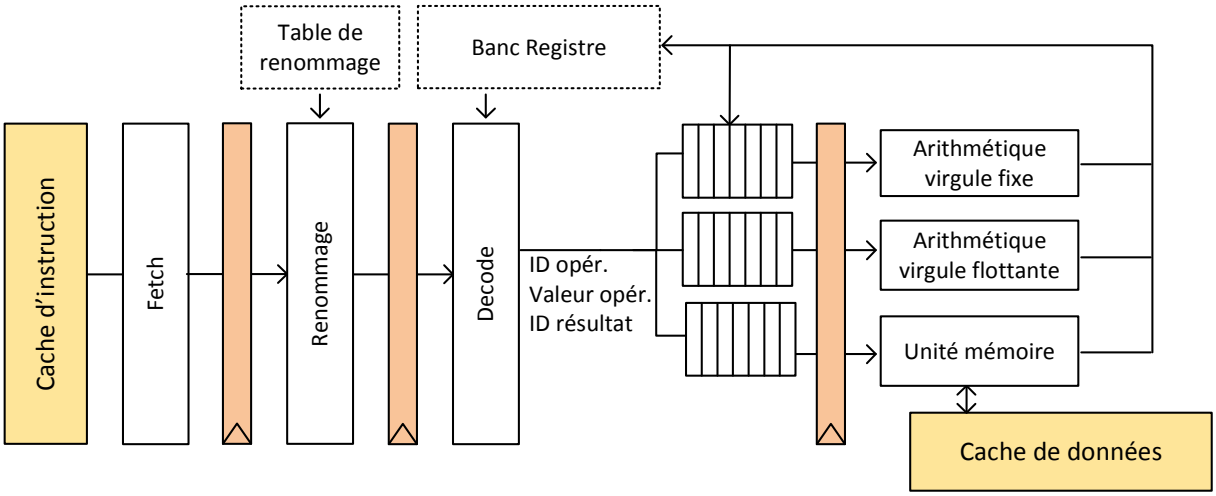


FIGURE 1.5 – Organisation simplifiée d'un processeur superscalaire à exécution dans le désordre basé sur l'algorithme de Tomasulo. Un étage de pipeline est consacré au renommage des registres et des files d'attente ont été ajoutées pour gérer l'exécution dans le désordre.

Le fonctionnement d'une telle micro-architecture est le suivant :

— les instructions sont lues dans l'ordre séquentiel lors de l'étage *Fetch*, puis un mécanisme est utilisé pour renommer les registres. Chaque valeur créée reçoit un identifiant unique qui est également utilisé lorsque la valeur sert d'opérande dans une autre instruction. Ce mécanisme de renommage permet de s'affranchir des dépendances de noms, c'est-à-dire des dépendances entre deux instructions écrivant dans le même registre (WAW pour *Write After Write*) ou des dépendances entre une instruction qui lit un registre et une qui écrit dans ce même registre (WAR pour *Write after Read*).
— les instructions décodées sont ensuite placées dans des files d'attente, regroupées avec les autres instructions s'exécutant sur le même type d'unité. En plus de l'instruction, ces files d'attente stockent également les identifiants et les valeurs des opérandes, ainsi que l'identifiant du résultat de l'instruction. Si la valeur d'un opérande est connue au moment du décodage de l'instruction, elle est stockée dans la file d'attente en même temps que l'instruction.
— lorsque tous les opérandes d'une instruction sont prêts, l'exécution de celle-ci commence. Le résultat, accompagné de son identifiant, est ensuite envoyé vers le banc de registres. De plus, l'identifiant du résultat est comparé à l'identifiant de chaque opérande de chaque instruction présente dans les files d'attente. En cas de correspondance, la valeur est stockée dans la file d'attente. Ainsi, de nouvelles instructions peuvent devenir prêtes à être exécutées. Il est important de noter que plusieurs résultats sont générés à chaque cycle, ce qui duplique la logique nécessaire à ce mécanisme.

Ce mécanisme d'exécution dans le désordre concerne uniquement les instructions arithmétiques, qui peuvent être réorganisées sans risque. Cependant, les accès mémoire et les instructions de branchement sont exécutées différemment, pour garantir une exécution correcte.

Les accès à la mémoire sont gérés en deux étapes. Tout d'abord, l'adresse de l'accès est calculée, ensuite l'instruction est ajoutée dans la file d'attente des accès mémoire. Quand l'unité mémoire est disponible, l'instruction est exécutée. Dans l'approche de Tomasulo, les accès à la mémoire doivent être exécutés dans le même ordre que dans le programme original. Cette restriction permet de s'assurer que les dépendances mémoire sont respectées. De même, aucune instruction ne peut être exécutée avant que toutes les instructions de branchement qui la précèdent ne soient résolues.

La figure 1.6 représente une trace d'exécution sur un processeur superscalaire à exécution dans le désordre. Les instructions exécutées sont les mêmes que sur la figure 1.4. Tout comme pour l'exécution dans l'ordre, les instructions 5 et 6 ne peuvent commencer avant le cycle $t + 8$ car les opérandes ne sont pas disponibles. Cependant, le mécanisme d'exécution dans le désordre autorise à commencer l'exécution des instructions 7 et 8 dès le cycle $t + 6$, donc avant que l'exécution des instructions 5 et 6 n'ait commencé. Ainsi, aux cycles $t + 6$ et $t + 7$, le processeur exécute quatre instructions en parallèle.

Ce mécanisme d'exécution dans le désordre permet d'identifier un plus grand nombre d'instructions indépendantes qu'un simple mécanisme d'exécution dans l'ordre. Cependant, la gestion des instructions de branchement et des accès mémoire limite fortement le parallélisme d'instructions pouvant être exploité. Le concept d'exécution spéculative présenté dans la suite permet de lever cette limite.

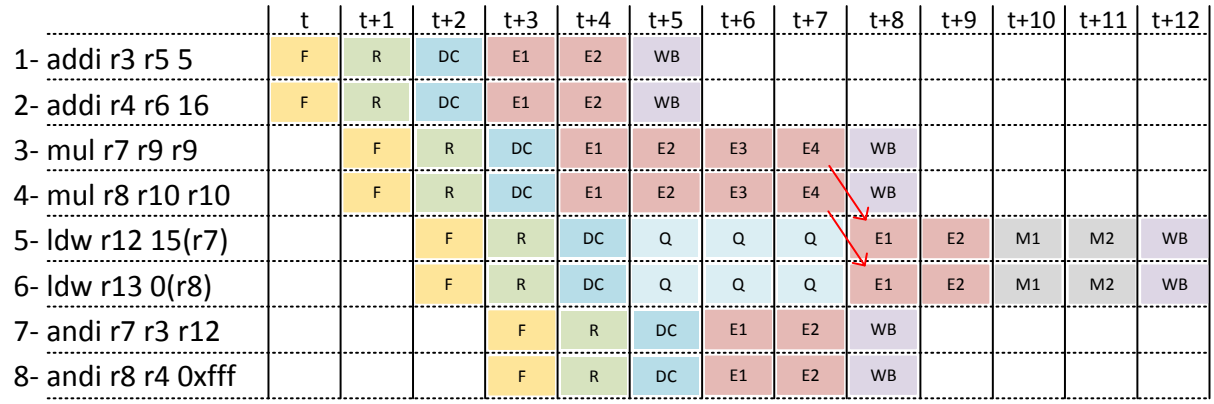


FIGURE 1.6 – Exemple de trace d'exécution sur un processeur superscalaire à exécution dans le désordre. Les paramètres d'exécution et les instructions exécutées sont les mêmes que pour la figure 1.4. Comme pour le processeur à exécution dans l'ordre, l'étage de *Fetch* charge deux instructions par cycle. Ces instructions passent ensuite par les étages de renommage et de décodage. Si tous ses opérandes sont disponibles, l'exécution de l'instruction commence. Sinon, celle-ci est stockée dans une file d'attente (étage Q sur la figure). Les instructions 5 et 6 passent ainsi trois cycles dans les files d'attente. Les opérandes des instructions 7 et 8 sont disponibles au cycle $t + 5$, ces instructions n'ont donc pas besoin d'attendre dans une file.

Exécution spéculative

Pour augmenter le parallélisme d'instructions exploitable, les micro-architectures ont été modifiées pour permettre l'exécution spéculative. Les instructions situées après un branchement conditionnel peuvent être exécutées sans que le résultat du branchement ne soit connu. Les accès mémoire peuvent être exécutés dans le désordre, afin de pouvoir commencer un chargement mémoire aussi tôt que possible. Pour préserver la sémantique du programme d'origine, des mécanismes matériels ont été ajoutés pour vérifier que la spéculation est correcte ou pour annuler des instructions lorsque celles-ci n'auraient pas dû être exécutées.

Le premier ajout est un mécanisme de prédiction de branchement perfectionné capable de prédire le bloc d'instructions qui sera exécuté après un branchement. Les instructions de ce bloc sont exécutées spéculativement. Ce mécanisme est non seulement capable de prédire les branchements conditionnels (c'est-à-dire de prédire si un branchement est pris ou non) mais également la destination de branchements indirects [81].

Deux autres mécanismes sont ajoutés pour permettre d'annuler l'exécution d'une instruction ou pour détecter la violation d'une dépendance mémoire :

— Le ***Re-Order Buffer*** (ROB) permet de stocker les instructions dont l'exécution est terminée (c'est-à-dire les instructions dont le résultat a été calculé par l'unité d'exécution) jusqu'à ce qu'elles soient validées (c'est-à-dire jusqu'à ce que le résultat soit écrit dans le banc registre). Les instructions sont validées dans l'ordre du programme séquentiel afin de garantir que la sémantique du programme d'origine est respectée. Le résultat de l'instruction peut cependant être utilisé pour d'autres opérations avant cette validation. Si jamais l'instruction n'est pas validée, toutes les

instructions ayant utilisé son résultat sont également annulées.

— La ***Load-Store Queue*** (LSQ) permet de vérifier la spéculation de dépendances mémoire. En effet, le mécanisme d'exécution dans le désordre permet d'effectuer les différentes opérations mémoire dans le désordre et la LSQ compare les adresses afin de déterminer si des dépendances mémoires ont été violées. Toutes les opérations de lecture et d'écriture de la mémoire, ainsi que l'adresse de l'accès sont conservées dans une mémoire dédiée. Les écritures mémoire sont effectives que lors de la phase de validation de l'instruction (donc dans l'ordre). Si, lors d'une lecture, la LSQ contient une écriture à la même adresse située après la lecture dans l'ordre séquentiel, la valeur souhaitée est toujours présente en mémoire (puisque l'écriture n'a pas pu être validée avant la lecture en cours). Si, lors d'une écriture, on trouve une lecture à la même adresse située après l'écriture dans l'ordre séquentiel, la lecture en question est annulée ainsi que toutes les instructions ayant utilisé cette valeur. Le fonctionnement précis d'une *Load-Store Queue* est décrit dans la section 4.2.1.

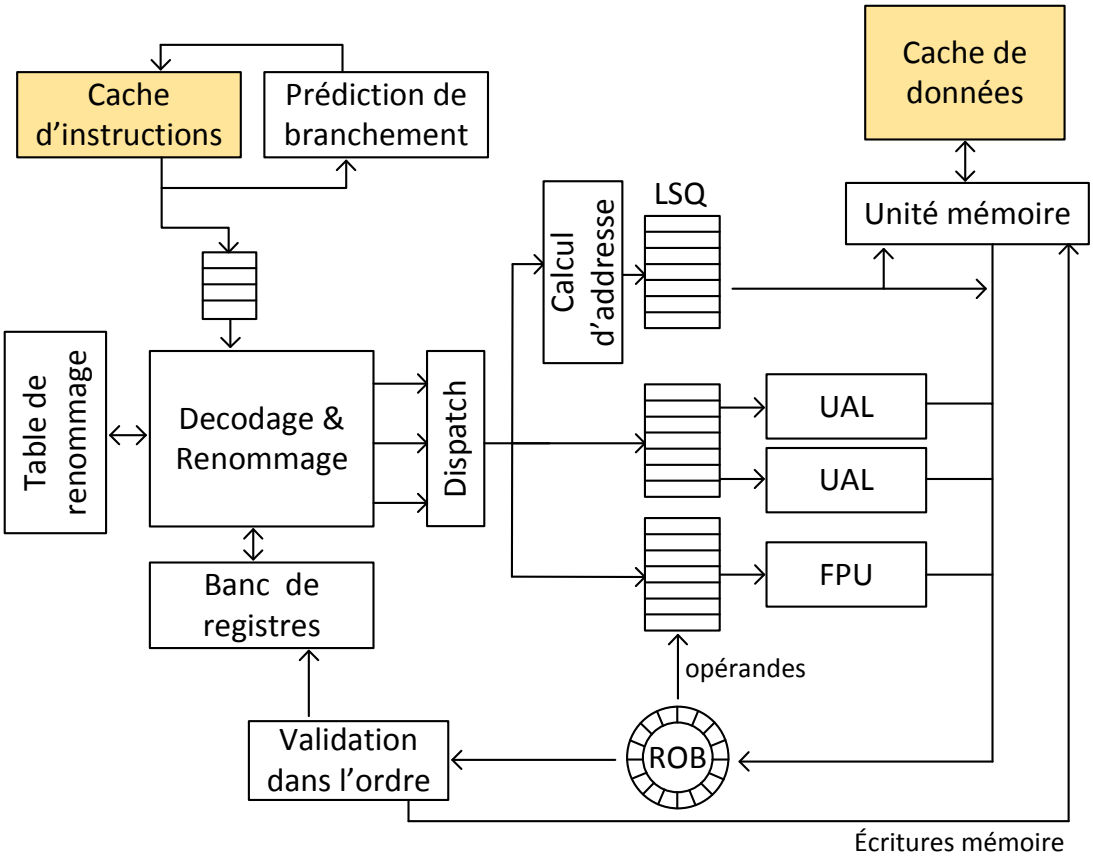


FIGURE 1.7 – Organisation simplifiée d'un processeur à exécution dans le désordre avec les mécanismes permettant l'exécution spéculative. En raison de la complexité du schéma, la représentation des étages de pipeline a été omise.

Maintenant que le principe de base des différents composants a été exposé, nous allons présenter l'organisation globale d'un processeur à exécution dans le désordre avec exécution spéculative.

La figure 1.7 est une représentation simplifiée d'une telle architecture. Les composants clés évoqués précédemment sont représentés ici :

- Le prédicteur de branchements et le cache d'instructions chargent les instructions à exécuter dans l'ordre séquentiel, en spéculant sur le chemin pris lors de l'exécution ;
- L'unité de décodage et de renommage décode les instructions et accède à la table de renommage et au banc de registres pour obtenir la valeur des opérandes ainsi qu'un identifiant pour le résultat. Ces instructions sont ensuite réparties entre les différentes files d'attente. Trois files sont utilisées ici : une pour les accès mémoires, une pour les instructions arithmétiques simples (qui sont exécutées sur les Unités Arithmétiques et Logiques (UAL)) et une pour les instructions en arithmétique flottante (qui sont exécutées par la *Floating Point Unit* (FPU)).
- Les instructions exécutées sont ensuite stockées dans le *Re-Order Buffer* en attendant d'être validées. Dans l'intervalle, leur résultat pourra être utilisé comme opérande pour les instructions à exécuter.
- Les accès mémoire sont gérés conjointement par la *Load-Store Queue* et par le *Re-Order Buffer* afin de détecter et de corriger les aléas lors des accès mémoire.

Le bon fonctionnement d'un processeur à exécution dans le désordre repose sur un équilibre délicat entre le nombre d'instructions entrantes, le nombre d'instructions pouvant être validées en même temps, la taille des files d'attente, de la LSQ et du ROB ainsi que du nombre d'unités d'exécution et de la taille du banc de registres. Tous ces paramètres impactent la performance de la micro-architecture mais également sa surface et sa consommation énergétique. Dans la suite, nous présentons quelques exemples d'implémentation de processeur à exécution dans le désordre. L'objectif est de fournir un ordre de grandeur de la taille des *ROB*, du nombre de registres et du nombre d'unités d'exécution dans différents processeurs.

Présentation de quelques implémentations

Pour appréhender les ordres de grandeur des différents paramètres utilisés dans les processeurs superscalaires à exécution dans le désordre (ou bien processeurs OoO pour *Out-of-Order*) modernes, nous allons présenter plusieurs exemples de tels processeurs. Ces processeurs ne sont pas destinés au même domaine (ordinateur de bureau ou système embarqué) et font des choix de valeurs de paramètres différents.

Les implémentations présentées sont :

- *Intel Coffee Lake micro-architecture :* Une des dernières générations de micro-architecture Intel, elle a été annoncée fin 2017 et constitue un bon exemple d'une configuration pour un processeur haute performance.
- *Arm Cortex A15 :* Processeur embarqué commercialisé en 2015. Celui-ci est notamment utilisé comme processeur *big* dans les premiers systèmes big.LITTLE de ARM.
- *Berkeley Out-of-Order Machine (BOOM) :* Processeur à exécution dans le désordre développé à Berkeley qui a la particularité d'être open-source et a été développé dans le cadre d'un projet académique [21]. Ce processeur est basé sur le jeu d'instructions BOOM. Pour ces raisons, le BOOM est souvent utilisé comme point de comparaison dans le reste de ce document. Notons

également que le BOOM est configurable, la version présentée ici est la configuration par défaut du processeur.

	Intel Coffee Lake	Arm Cortex A15	BOOM
Année	2017	2015	2017
Jeu d'instructions	x86_64	ARMv7	RISC-V
Unités d'exécution	8	8	4
Taille du décodage	5	3	2
Taille des files d'attente	97	40+	46
Taille du ROB	224	128	48
Taille de la LSQ	72+56	16	16
Nombre de registres (entiers + flottants)	180+168	?	70+64

TABLE 1.1 – Comparaison de différentes implémentations de processeur à exécution dans le désordre.

Le tableau 1.1 résume les caractéristiques des différents processeurs étudiés. Les processeurs Intel Coffee Lake et Arm Cortex A15 sont deux processeurs commerciaux. Les chiffres montrent que la micro-architecture Intel est beaucoup plus agressive dans les mécanismes d'exécution dans le désordre : les tailles des files d'attente, du *Re-Order Buffer* et de la *Load-Store Queue* sont nettement plus importantes que pour le processeur ARM, même si le nombre d'unités d'exécution est similaire. Cela vient entre autre du marché ciblé : les processeurs Intel sont utilisés dans les ordinateurs de bureau avec pour seul objectif d'offrir un niveau maximal de performance, tandis que les processeurs Arm sont utilisés dans des systèmes embarqués où les contraintes énergétiques sont beaucoup plus fortes.

Dans cette partie, nous avons montré que l'utilisation de micro-architectures complexes permettait d'améliorer les performances, au prix d'une complexité matérielle (et donc d'un surcoût en énergie important).

Dans la suite de cette section, nous présentons le fonctionnement des architectures statiquement ordonnancées, qui s'avèrent plus efficaces énergétiquement que les OoO sur certaines applications.

1.2.2 Les micro-architectures statiquement ordonnancées

Les architectures statiquement ordonnancées permettent d'atteindre des compromis performance/énergie différents de ceux offerts par des processeurs dynamiquement ordonnancés. Dans ces architectures, le parallélisme d'instructions est explicitement encodé dans le jeu d'instructions. Le compilateur est donc en charge d'analyser le code de l'application et d'appliquer des transformations permettant d'extraire du parallélisme d'instructions. Il réalise ensuite un ordonnancement des instructions sur les différentes unités d'exécution et génère le binaire de l'application. Lors de l'exécution, la micro-architecture exécute les instructions en exploitant le parallélisme d'instructions exposé par le compilateur. Parce qu'elles n'utilisent pas de mécanismes spéculatifs et/ou d'exécution dans le désordre, les architectures statiquement ordonnancées sont très efficaces énergétiquement. Toutefois, certaines applications exhibent un flot de contrôle difficile à prédire et ne sont pas adaptées à de telles architectures.

L'architecture statiquement ordonnancée la plus courante est le processeur *Very Long Instruction*

Word (VLIW). A chaque cycle d'exécution, le processeur lit et exécute un agrégat de plusieurs instructions indépendantes (qu'on appellera dans la suite un *bundle* d'instructions). Ces *bundles* sont construits statiquement par le compilateur lors de la phase d'ordonnancement des instructions. Le modèle de programmation des VLIW suppose que ces instructions peuvent être exécutées en parallèle [34].

Dans la suite de cette section, nous présentons en détail l'organisation d'un processeur VLIW et les différents choix possibles dans son architecture. Nous présentons ensuite leurs limitations et discutons des moyens de les contourner.

Architecture des VLIW

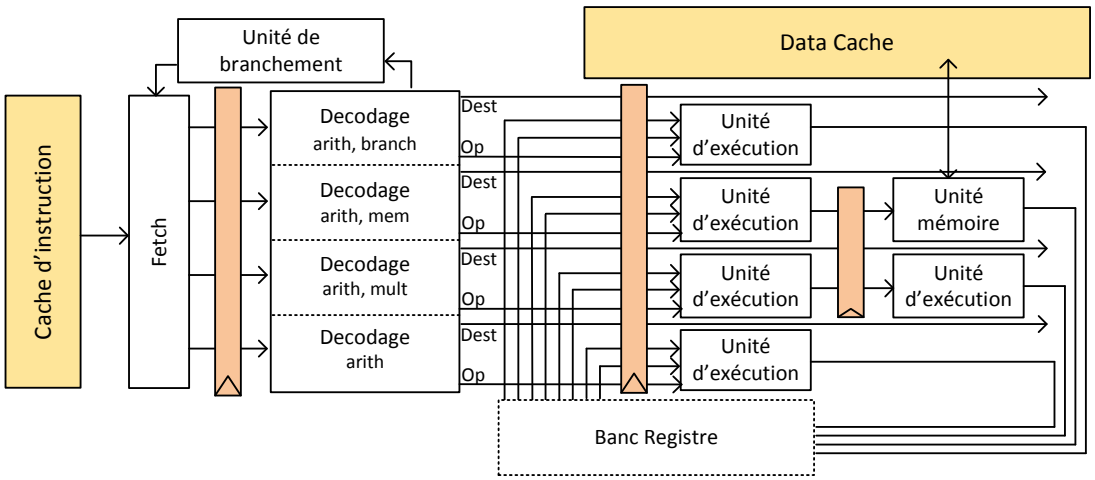


FIGURE 1.8 – Organisation simplifiée du processeur VLIW développé pendant cette thèse. Il peut exécuter jusqu'à quatre instructions par cycle et son pipeline est organisé en quatre ou cinq étages, selon la voie utilisée.

La figure 1.8 est une représentation simplifiée du VLIW utilisé dans nos différents travaux, librement inspiré du processeur ST200/Lx [32]. Ce processeur VLIW dispose de quatre voies d'exécution, la figure 1.8 représente donc quatre unités de décodage et quatre unités d'exécution. Le banc de registres permet de lire deux valeurs et d'en écrire une pour chaque unité d'exécution. L'exécution d'un *bundle* d'instructions comprend les étapes suivantes :

— Le *Fetch* lit des *bundles* d'instructions dans le cache d'instructions et les envoie vers les unités de décodage. Celles-ci vont lire le code de l'instruction, la décoder et charger les différents opérandes. Un ou deux accès au banc de registres sont ensuite réalisés pour chaque instruction afin de récupérer la valeur des opérandes.

— L'étape suivante concerne les unités d'exécution. La plupart des VLIW utilisent des voies d'exécution spécialisées : l'instruction n d'un *bundle* ne pourra appartenir qu'à un sous-ensemble du jeu d'instructions global. Cela permet de simplifier les unités de décodage et les unités d'exécution. Dans l'exemple de la figure 1.8, la liste des instructions pouvant être exécutées sur une voie est donnée dans l'unité de décodage associée (*arith, mem, branch* et *mult*). Toutes les voies peuvent exécuter les opérations arithmétiques simples cependant seule la première peut exé-

cuter des branchements, seule la deuxième a accès à la mémoire et seule la troisième peut exécuter les instructions arithmétiques complexes (multiplication et division).
— Une fois l'exécution terminée, son résultat est écrit dans le banc de registres.

La profondeur du pipeline (c'est-à-dire le nombre de cycles nécessaires à l'exécution d'une instruction) dépend de la spécialisation de la voie d'exécution. Dans notre exemple, les voies capables d'exécuter des opérations arithmétiques complexes ont besoin de deux étages d'exécution tandis que les autres n'en ont qu'un. Exécuter des opérations arithmétiques simples sur ces voies d'exécution demande quand même un cycle supplémentaire.

Le processeur VLIW que nous utilisons s'inspire du processeur ST200 de STMicroelectronics. Notre VLIW présente cependant une différence importante dans certains choix d'implémentation. L'architecture du ST200 détecte et gère les aléas en gelant le pipeline quand nécessaire. Ce mécanisme est similaire à celui utilisé dans les architectures pipelinées simples, cependant son coût est augmenté pour les VLIW. Le VLIW que nous utilisons laisse au compilateur le soin de gérer les aléas de pipeline : si une instruction a été ordonnancée à un endroit donné c'est que ses opérandes sont prêts.

Plusieurs paramètres peuvent être modifiés dans l'architecture d'un VLIW, impactant sa consommation énergétique et ses performances : le nombre de voies du VLIW, le nombre de registres dans le banc de registres et la spécialisation de chaque voie d'exécution. Les deux premiers paramètres ont un impact important sur la taille du VLIW (et sur sa consommation). En effet, plusieurs facteurs entrent en jeu : plus on exécute d'instructions en parallèle, plus le nombre de résultats à conserver dans le banc de registres est grand. La taille du banc de registres est donc liée au nombre de voies. De plus, chaque voie du VLIW peut réaliser un accès en écriture et plusieurs accès en lecture dans le banc de registres. Pour notre VLIW, une instruction ne peut utiliser plus de deux registres. Chaque voie réalise donc deux lectures. La taille du banc de registres dépend du nombre de bits, du nombre de registres, du nombre de ports de lecture et du nombre de ports d'écriture. Augmenter l'un des paramètres aura un impact linéaire sur la taille du banc de registres. En considérant qu'augmenter le nombre de voies d'un VLIW augmente linéairement le nombre de registres qu'il faut utiliser, et en prenant en compte l'augmentation du nombre de ports de lecture/écriture, augmenter le nombre de voies d'un VLIW a un impact quadratique sur la taille du banc de registres.

Le dernier paramètre (la spécialisation des différentes voies d'exécution) impacte principalement les performances de l'architecture. En effet, une limitation du nombre d'instructions pouvant être exécutées sur une voie donnée réduit la taille des unités de décodage et des unités d'exécution, ainsi que la profondeur du pipeline. Par exemple, le VLIW représenté sur la figure 1.8 utilise deux étages de pipeline pour réaliser une multiplication. Puisque l'architecture n'offre pas de mécanismes de *forwarding*, toute instruction exécutée sur cette voie demandera un cycle supplémentaire.

Dans cette première partie, nous avons présenté le fonctionnement des processeurs VLIW. Le fait d'encoder le parallélisme d'instructions permet de réduire la complexité de l'architecture et donc de réduire sa consommation énergétique. Dans la suite de cette section, nous allons comparer les processeurs VLIW et les processeurs OoO en terme de performance et de consommation énergétique.

1.2.3 Comparaison de ces deux paradigmes

L'objectif de cette sous-section est de comparer les deux paradigmes d'exécution que nous venons de présenter : les architectures statiquement ordonnancées et les architectures dynamiquement ordonnancées à exécution dans le désordre. Nous présentons également plusieurs approches qui ont tenté de réduire l'écart entre ces deux paradigmes.

Comparaison qualitative

Grâce à leurs mécanismes d'ordonnancement dynamique et de spéculation, les processeurs OoO permettent en général d'obtenir de meilleures performances que les processeurs VLIW. Les travaux de McFarlin et al. tentent de mesurer la différence de performance entre les architectures statiquement ordonnancées et les architectures dynamiquement ordonnancées et essaient d'étudier d'où provient cette différence [62]. Ils identifient deux sources potentielles pour la différence de performance :

— La capacité à générer un ordonnancement dynamique, qui permet à ces architectures de réagir extrêmement bien à des événements imprévisibles tels que les *cache miss*. Au lieu de stopper l'exécution en attendant la donnée, l'architecture dynamiquement ordonnancée sera capable d'exécuter d'autres instructions qui ne dépendent pas de l'accès mémoire bloqué.

— La capacité des architectures OoO à spéculer grâce à de nombreux mécanismes matériel : la prédiction de branchements permet de spéculer sur les prochains blocs exécutés, la *load-store queue* permet de spéculer que des accès mémoire sont indépendants et le *re-order buffer* permet de d'exécuter spéculativement des instructions en offrant des mécanismes d'annulation. La plupart des architectures statiquement ordonnancées n'ont pas accès à ce genre de mécanismes.

La première étude réalisée par McFarlin et al. permet de comparer la capacité de spéculation des processeurs statiquement et dynamiquement ordonnancés. Pour cela, les auteurs ont mesuré la capacité d'un processeur OoO à explorer les blocs de base spéculativement. Ils ont compté le nombre moyen de blocs de base présents, à un instant donné, dans le *re-order buffer*. Pour dériver une métrique similaire pour les processeurs statiquement ordonnancés, ils ont dénombré le nombre de blocs de base présents dans une région de code ordonnancée par le compilateur, c'est-à-dire la capacité du compilateur à fusionner des blocs avec différentes techniques. Cette étude montre que les principales techniques de fusion de blocs permettent de créer des régions allant de 3,5 à 5 blocs. Certaines techniques d'analyse dynamique permettent de dépasser les 6 blocs [70]. Les résultats pour le processeur OoO montrent que 70% de ses ordonnancements utilisent les instructions provenant de 4 blocs de base, et 82% utilisent les instructions venant de 6 blocs de base. L'étude montre également que certains ordonnancements du OoO utilisent jusqu'à 15 blocs de base en même temps. Les mécanismes du OoO permettent donc d'extraire du parallélisme d'instructions sur une fenêtre beaucoup plus grande que les compilateurs statiques, ce qui laisse à penser que la capacité à spéculer joue un rôle important dans la différence de performance entre architectures statiquement et dynamiquement ordonnancées.

Pour confirmer cette idée, les auteurs ont développé un outil permettant de simuler une exécution sur un processeur dynamiquement ordonnancé, d'extraire l'ordonnancement le plus fréquemment utilisé pour chaque région de l'application et de l'exécuter sur une architecture statiquement ordonnancée. Ils ont mesuré les différences de performances entre l'architecture statiquement ordonnancée, l'architec-

ture dynamiquement ordonnancée et leur simulateur. Leurs résultats suggèrent que l'architecture statiquement ordonnancée arrive en moyenne à 47% des performances de l'architecture dynamiquement ordonnancée. Leur outil de simulation permet en revanche d'atteindre 89% des performances du OoO. Les auteurs concluent ainsi que la différence de performance vient à 79% de la capacité à spéculer correctement. Le reste peut être attribué à la capacité à réagir dynamiquement à certains événements.

Afin de combler l'écart entre ces deux paradigmes d'exécution, de nombreux travaux ont proposé des techniques permettant de mieux gérer les événements dynamiques dans des architectures statiquement ordonnancées. Par exemple, pour gérer les *cache miss*, des mécanismes permettant de continuer d'exécuter les instructions sans attendre la valeur de l'accès mémoire ont été développés [45, 64, 29]. L'idée de ces mécanismes est simple, lorsqu'un accès mémoire bloque parce que la donnée n'est pas présente dans le cache, l'état courant du banc de registre est sauvegardé et la valeur du registre de destination est contaminée. L'exécution continue au lieu de se bloquer. Les instructions ayant des opérandes contaminés contaminent leur registre de destination et sont stockées dans une mémoire tampon. Les instructions n'ayant aucun opérande contaminé sont exécutées normalement. Lorsque la donnée devient disponible, les instructions mises de côté sont ré-exécutées en utilisant les valeurs des registres sauvegardées au moment de l'accès mémoire. L'exécution revient ainsi à un état cohérent.

Approches intermédiaires

Comme nous venons de le voir, les mécanismes de spéculation dans les processeurs dynamiquement ordonnancés permettent un gain de performance important mais entrainent une augmentation considérable de la puissance dissipée. Dans cette partie, nous étudions plusieurs architectures qui combinent les avantages des processeurs OoO et des processeurs VLIW. Certaines des approches discutées sont basées sur des travaux universitaires et leur évaluation repose souvent sur l'utilisation de simulateurs, d'autres approches sont en revanche implémentées et commercialisées.

Lorsqu'un processeur OoO exécute une boucle, il recalcule un ordonnancement pour chaque nouvelle itération de cette boucle. Il est tentant d'essayer de réutiliser (au moins partiellement) cet ordonnancement pour les itérations suivantes. Toutefois, il est important de souligner que les mécanismes d'ordonnancement dynamique d'un OoO ne sont pas pensés pour générer des ordonnancements réutilisables. Par exemple, le renommage de registres réalisé pour une itération d'une boucle n'a aucune raison d'être compatible avec la prochaine itération (les valeurs d'entrée ne sont pas forcément dans les mêmes registres physiques). Des mécanismes matériels non triviaux doivent donc être mis en oeuvre pour permettre de réutiliser ces ordonnancements et il n'existe pas à ce jour d'implémentation matérielle permettant de quantifier les surcoûts en performance ou en consommation causés par ces modifications profondes. Cette approche a été dérivée sous différentes formes : Padmanabha et al. proposent une version modifiée d'un processeur superscalaire à exécution dans l'ordre, dans lequel ils ont ajouté des mécanismes de spéculation simplifiés [67, 66]. Ces machines peuvent ainsi supporter un renommage contraint et sont équipées d'une *load-store queue*. Ils ont également modifié une architecture à exécution dans le désordre pour que les ordonnancements générés soient compatibles avec les mécanismes de spéculation simplifiés. Lors de l'exécution, si un ordonnancement est souvent réutilisé, il s'exécutera sur l'architecture statiquement ordonnancée afin de réduire sa consommation. Leurs estimations montrent

que l'utilisation de ces processeurs à exécution dans l'ordre permet de réduire l'énergie consommée de 30%, au prix d'une baisse de performance de 5%. L'approche de Villavieja et al. repose sur une idée similaire mais fournit moins de détails sur l'implémentation des différents cœurs utilisés [87].

Brandalero et al. proposent de suivre la même approche en couplant un processeur OoO et un CGRA simplifié [14]. Les ordonnancements souvent utilisés par le OoO sont traduits en une configuration du CGRA qui sera utilisée à la prochaine exécution. Les auteurs contournent les difficultés liées au renommage de registres et les instructions sortant du CGRA repassent dans le *re-order buffer* afin de s'assurer que les spéculations faites lors de la première exécution sont toujours valables. Ce mécanisme risque donc d'induire une forte pression sur le ROB, réduisant ainsi les performances. L'étude expérimentale réalisée montre que leur système est 30% plus performant que le OoO seul tout en consommant 30% moins d'énergie. Le modèle de CGRA sur lequel ils basent leur expérience suppose cependant qu'ils puissent effectuer trois opérations arithmétiques ou logiques chaînées dans un même cycle lorsqu'ils utilisent le CGRA tout en fonctionnant à la même fréquence que le processeur. Cette hypothèse de travail est peu raisonnable et biaise fortement les résultats en faveur de leur approche.

Les processeurs Itanium sont également une approche intermédiaire, dont l'objectif est de se positionner entre les processeurs OoO trop énergivores et les processeurs VLIW pas suffisamment performants. Ils ont été commercialisés par Intel entre 2001 et 2017 [82]. La micro-architecture peut être vue comme un superscalaire dynamiquement ordonnancé à exécution dans l'ordre : les instructions sont chargées séquentiellement et placées dans des files d'attente de la même façon que pour un OoO mais ici elles sont exécutées dans leur ordre d'arrivée uniquement. Cette restriction simplifie fortement l'architecture. Le jeu d'instructions utilisé (EPIC pour *Explicitely Parallel Instruction Computer*) est un jeu d'instructions explicitement parallèle, similaire à celui d'un VLIW. En effet, les instructions sont organisées en groupements indépendants qui peuvent être de taille variable. Des bits sont ajoutés dans l'encodage pour délimiter ces groupes. Ainsi le parallélisme est explicité dans le jeu d'instructions mais laisse néanmoins une certaine liberté au processeur pour les réorganiser. Ce jeu d'instructions a été construit pour améliorer les performances des architectures statiquement ordonnancées : il embarque de nombreux mécanismes permettant la spéculation logicielle (toutes les instructions sont prédiquées, des bancs de registres spéciaux sont utilisés pour ces prédicats et chaque registre a un bit de contamination permettant de voir si il résulte d'un accès incorrect à la mémoire).

1.3 Bilan

Dans ce chapitre, nous avons présenté en détail une partie du contexte dans lequel nos travaux ont été réalisés. Nous avons présenté en détail les différents moyens utilisés pour adapter dynamiquement le compromis entre la performance et l'efficacité énergétique d'un système. En particulier, les architectures multi-cœurs hétérogènes à jeu d'instructions unique utilisent, sur un même circuit, différentes micro-architectures, offrant chacune un compromis différent. Lorsqu'une application est exécutée sur un tel système, elle peut être migrée d'un type de cœur à un autre pour s'adapter à ses besoins. Nous avons également présenté le principe des architectures statiquement ordonnancées et conclu que de tels processeurs peuvent offrir un compromis différent de ceux utilisés dans les big.LITTLE actuels, si toutefois on peut s'affranchir du problème du jeu d'instructions. Dans le prochain chapitre, nous

présentons le principe de la traduction dynamique de binaires. Ce mécanisme permet d'exécuter une application compilée pour un jeu d'instructions A sur un processeur prévu pour un jeu d'instructions B. En utilisant cette approche, il est possible d'utiliser des processeurs VLIW dans un système hétérogène en donnant l'illusion d'un jeu d'instructions unique.

CHAPITRE 2

LA TRADUCTION DYNAMIQUE DE BINAIRES

Dans le chapitre précédent, nous avons introduit la notion de systèmes embarqués et mentionné leur besoin d'adaptation dynamique. Pour répondre à ces besoins, les frabricants de processeurs ont été amenés à proposer des systèmes multi-cœurs hétérogènes (entre autre). Un tel système embarque différents types de micro-architectures, chacune offrant un compromis différent entre la performance et la consommation énergétique. Pour que l'adaptation dynamique soit possible, ces différentes micro-architectures doivent être capables d'exécuter les mêmes binaires et donc de supporter le même jeu d'instructions.

Les processeurs pouvant être intégrés au sein d'un tel système sont contraints d'offrir de la compatibilité binaire. Les processeurs VLIW permettent d'obtenir, pour certaines applications, des performances plus élevées et une consommation d'énergie moindre que les processeurs OoO. Toutefois, ces architectures ne peuvent pas être intégrées dans un système multi-cœurs hétérogène à jeu d'instructions unique car elles sont basées sur des jeux d'instructions explicitement parallèles.

La traduction dynamique de binaire (DBT pour *Dynamic Binary Translation*) permet l'intégration des architectures statiquement ordonnancées dans les systèmes multi-cœurs hétérogènes. Le principe de la DBT est de traduire à l'exécution un binaire compilé pour un jeu d'instructions A vers un binaire pour un jeu d'instructions B. Dans le cadre d'un système multi-cœurs hétérogène, la DBT permet d'intégrer différents types d'architectures en conservant l'illusion d'un système à jeu d'instructions unique.

Traduire dynamiquement un programme binaire pour un jeu d'instructions séquentiel vers un binaire ciblant un jeu d'instructions explicitement parallèle est un problème difficile. En effet, si on souhaite pouvoir profiter du parallélisme d'instructions, l'outil de DBT doit transformer le graphe de flot de contrôle pour exposer plus de parallélisme puis ordonnancer les instructions. Malgrè ces difficultés, il existe déjà des systèmes basés sur cette approche. Par exemple, le processeur Denver de NVidia permet d'exécuter des binaires ARM sur une architecture de type VLIW [12].

Dans ce chapitre, nous présentons en détail le principe de la compilation dynamique ainsi que les défis posés par le temps de compilation. Nous étudions plus en détail les approches existantes, en nous focalisant sur celles qui ciblent des architectures statiquement ordonnancées.

2.1 La compilation dynamique

Dans cette première section, nous présentons les différentes formes de compilation dynamique existantes. Nous discutons également de la forte contrainte liée au temps de compilation et de plusieurs approches permettant de le réduire.

2.1.1 Des différents usages de la compilation dynamique

L'exemple le plus fréquent d'utilisation de techniques de compilation dynamique sont les machines virtuelles Java Hotspot [54], le compilateur .NET pour le C# [40], Dalvik dans certaines versions du système Android ou encore le Javascript dans les navigateurs tels que Chrome (V8) ou Firefox (Spidermonkey) [24]. Dans ces outils, l'utilisation de la compilation dynamique permet d'obtenir de meilleures performances qu'une approche basée sur des techniques d'interprétation. Ce type de compilation est appelée compilation JIT (*Just-in-Time*).

L'aspect dynamique de la compilation JIT est un avantage par rapport à la compilation statique : pour des langages dynamiquement typés, un outil de compilation JIT permet d'augmenter les performances à l'aide de techniques de spécialisation de type [36]. De plus, certains langages (par exemple le Javascript) permettent d'exécuter des programmes construits dynamiquement (en utilisant la fonction `eval()`), ce qui rend la compilation statique de programmes Javascript impossible.

La compilation dynamique est également utilisée pour l'optimisation dynamique de code. Par exemple, Kistler et al. ont étudié le coût et les bénéfices de l'optimisation continue [51]. Dans cette étude, ils ont analysé le coût du profilage et de certaines transformations (propagation de variables, elimination de code mort, construction dynamique de traces, optimisation de l'organisation mémoire pour les langages objet). Cette étude montre que, grâce aux informations de profilage, les optimisations réalisées dynamiquement s'avèrent souvent plus efficaces que les optimisations statiques.

Apollo est un autre exemple d'outil d'optimisation dynamique [50, 18]. Celui-ci permet d'appliquer des transformations polyédriques sur des boucles qui ne sont normalement pas représentables dans le modèle (par exemple des boucles dont les accès ne sont pas une fonction linéaire des indices de boucle). Appolo analyse l'exécution de la boucle et construit une représentation polyédrique approximée à partir de plusieurs itérations. Des analyses et des transformations polyédriques (analyse des dépendances inter-itération, tuilage, parallélisation et vectorisation automatique) sont ensuite appliquées à ce modèle et du code optimisé est re-généré. L'outil injecte également des instructions de vérification permettant d'assurer que la boucle suit toujours le modèle polyédrique approximé.

Un autre outil basé sur la technique de l'optimisation dynamique est l'outil deGoal [23]. Celui-ci permet de générer dynamiquement un noyau binaire spécialisé en fonction des données effectivement utilisées lors de l'exécution. L'outil deGoal est également original de part sa manière de fonctionner : le développeur décrit son noyau d'exécution dans un langage spécifique qui est ensuite analysé statiquement pour créer une *compilette*. Une *compilette* est un générateur de code spécialisé pour le noyau décrit qui est capable de propager les valeurs des variables utilisées et d'effectuer des optimisations classiques (déroulage, élimination de code mort) sur le noyau ciblé. Grâce à ce mécanisme, le coût de la génération de code est grandement réduit.

Le fonctionnement de deGoal s'apparente à la *split-compilation* qui consiste à séparer la compilation d'une application en deux phases : l'une statique et l'autre dynamique. Les phases coûteuses et indépendantes de l'architecture ciblée sont réalisées statiquement tandis que les transformations spécifiques au processeur ciblé et la génération de code sont effectuées dynamiquement. Si des optimisations spécifiques à l'architecture ciblée s'avèrent coûteuses, la phase statique peut précalculer certaines informations permettant de les simplifier. Par exemple, Nuzman et al. ont étudié la spécialisation dynamique de code vectorisé [65]. Selon les standards utilisés dans les différents processeurs (par exemple SSE3, SSE4 ou AVX), la taille des vecteurs et les opérations disponibles sont différentes. Une analyse dynamique des binaires permettant de générer ces vecteurs entraine une augmentation importante du temps d'exécution. Si le code est analysé statiquement, l'information calculée est perdue avant la génération dynamique du code machine. Nuzman et al. ont cherché à déterminer quelles étaient les informations les plus importantes obtenues lors de ces analyses statiques, et ont proposé de les encoder dans la représentation intermédiaire utilisée pour la *split-compilation*. Ces informations sont ensuite utilisées pour générer dynamiquement les instructions vectorisées qui exploitent le mieux l'architecture utilisée.

2.1.2 La contrainte du temps de compilation

Un des défis de la compilation dynamique est que le temps passé à optimiser le code doit être compensé par le temps d'exécution économisé grâce à ces optimisations. Par exemple, passer plusieurs secondes à optimiser des instructions qui ne seront exécutées qu'une seule fois n'est pas bénéfique. A l'opposé, si une boucle représente 90% du temps d'exécution d'un programme, passer plus de temps à l'optimiser peut s'avérer pertinent.

Pour raisonner sur ce type de situations, il est nécessaire d'introduire plusieurs notions :

— le *cold-code* est une région de code qui n'a encore jamais été exécutée et sur laquelle aucune information de profilage n'est disponible ;

— le *hot code* ou *hotspot* représente une part importante du temps d'exécution total. Cette partie du programme a déjà été exécutée plusieurs fois et des informations de profilage sont donc disponibles.

Pour réduire l'impact de la compilation dynamique sur le temps d'exécution, les outils sont organisés en plusieurs niveaux d'optimisation. Par exemple, lors de sa première exécution, le *cold-code* est interprété afin d'augmenter la réactivité du système. Quand un même code est ré-interprété plus d'une dizaine de fois, l'outil de compilation dynamique en génère une version non optimisée. En fonction du nombre d'exécutions de cette version du code, des optimisations supplémentaires peuvent-être mises en oeuvre, ainsi, les optimisations les plus coûteuses ne sont appliquées qu'aux parties les plus critiques.

L'utilisation d'accélérateurs matériels permet également de réduire le coût de la compilation dynamique. Par exemple, Carbon et al. utilisent cette approche pour accélérer le module de compilation JIT de LLVM [20]. Ils ont ainsi développé un accélérateur matériel pour manipuler les arbres bicolores. Cette structure de données est utilisée pour implémenter les *set* et les *map* dans la librairie standard C++. Les auteurs ont également modifié les fonctions d'allocation dynamique de mémoire pour qu'elles uti-

lisent cette structure de données. Leur approche permet une réduction de 20% du temps de compilation dynamique de LLVM.

Hu et al. ont également étudié les avantages de co-développer l'architecture et l'outil de compilation dynamique afin de réduire le surcoût de la compilation[48]. Leur étude montre que la réactivité du système est principalement impactée par la gestion du *cold-code*. Ils proposent donc deux solutions pour réduire ce surcoût. La première idée consiste à utiliser des décodeurs matériels similaires à ceux utilisés pour traduire les instructions x86 en micro-code. La deuxième consiste à ajouter des instructions spécialisées pour réaliser cette traduction. Leur étude expérimentale montre que chacune de ces deux approches permet de réduire le coût de la gestion du *cold-code*. L'approche basée sur le décodeur matériel est celle qui permet d'atteindre le plus rapidement le niveau de performance maximal, cependant, lorsque ce niveau est atteint, le décodeur matériel doit rester activé. A l'inverse, l'utilisation d'instructions spécialisées est moins efficace lors des premières exécutions mais, une fois le code optimisé, les composants matériels ne sont plus utilisés et, par conséquent, ne consomment plus d'énergie.

Enfin, les différents outils de compilation dynamique exposés précédemment se distinguent également par leur format d'entrée. Par exemple, la JVM utilise un *bytecode* qui résulte d'une première compilation statique. Ce *bytecode* a été développé dans le but de minimiser l'empreinte mémoire et de faciliter l'interprétation. A contrario, les outils de compilation dynamique de Javascript traitent directement le code source. Cette différence de format peut impacter la difficulté de la traduction mais également la quantité d'informations disponibles pour l'outil. Un autre aspect de la compilation dynamique est la traduction dynamique de binaire qui utilise un fichier binaire comme format d'entrée.

2.2 Introduction à la traduction dynamique de binaires

La traduction dynamique de binaire (DBT *Dynamic Binary Translation*) consiste à exécuter une application compilée pour un jeu d'instructions A (dit jeu d'instructions *guest*) sur une architecture utilisant un jeu d'instructions B (dit jeu d'instructions *host*). Les instructions du programme invité doivent alors être analysées et traduites en des instructions de la machine hôte. La traduction dynamique de binaire est principalement utilisée dans un contexte d'émulation d'une architecture (avec par exemple Qemu [11]). Les techniques de DBT sont également utilisées pour assurer la portabilité entre différentes générations de jeu d'instructions. Par exemple, le système DAISY a été utilisé par IBM pour assurer la compatibilité binaire entre différentes générations de processeurs VLIW [30, 31, 83]. La DBT permet également d'assurer la portabilité d'un système à l'autre. Par exemple, la société Apple a développé Rosetta pour faciliter la migration de ses machines basées sur des PowerPC vers des architectures x86, de même Microsoft utilise depuis peu la DBT pour exécuter des applications x86 sur Arm. La DBT peut également être utilisée pour effectuer de l'instrumentation de code. Cette approche est utilisée par les outils Valgrind et Dynamo-Rio [17] pour analyser l'utilisation de la mémoire, pour débugguer ou pour réaliser un profilage précis d'une application.

Enfin, la traduction dynamique de binaire est également utilisée pour exécuter un programme sur une architecture utilisant un jeu d'instructions pour lequel il n'a pas été compilé. Par exemple, la DBT permet de traduire une application compilée pour un jeu d'instructions séquentiel vers un jeu d'instructions VLIW. L'intérêt est de réduire la consommation énergétique tout en conservant un niveau de

performance comparable. C'est l'approche qui a été adoptée par l'outil *Code Morphing Software* (CMS) de la société Transmeta [25]. Cet outil est capable d'exécuter des applications compilées pour du x86 sur un processeur VLIW, moins énergivore qu'un processeur superscalaire classique. Plus récemment, NVidia a adopté une approche similaire pour le processeur Denver [12]. Ici encore, l'objectif est d'exécuter des programmes ARM sur une architecture basée sur un jeu d'instructions VLIW. Ici aussi, une phase de traduction permet d'assurer la compatibilité binaire. Il est important de noter que ces deux approches sont des produits commerciaux à propos desquels il existe très peu d'informations disponibles. Ces approches sont discutées plus en détail dans la section 2.3.

Si les processeurs Denver et Efficeon utilisent la DBT pour exécuter des binaires séquentiels (x86, Arm) sur des architectures VLIW, les travaux de Michel et al. [63] abordent le problème inverse à savoir la simulation rapide d'architectures VLIW via des techniques de traduction dynamique de binaires. En effet, l'interprétation ou la traduction naïve des binaires ne fonctionne pas directement à cause de certains mécanismes spécifiques aux VLIW. Par exemple, deux instructions au sein d'un même *bundle* doivent être exécutées en parallèle et peuvent comporter des dépendances de nom. De même, des registres peuvent garder leurs valeurs pendant un ou deux cycles après le début d'une instruction et certains outils d'allocation de registres peuvent tirer parti de ce mécanisme. Enfin, certaines instructions peuvent être placées après les instructions de branchement quand il y a un *delay slot* (c'est-à-dire quand l'architecture va toujours exécuter la ou les instructions suivant immédiatement le branchement). Les auteurs détaillent ces différentes difficultés et proposent des solutions pour les gérer efficacement.

La DBT est soumise aux mêmes contraintes que la compilation dynamique : chaque cycle passé à optimiser (ou traduire) du code doit être compensé par l'accélération apportée par l'optimisation. Les surcoûts causés par la traduction dynamique de binaire ont été étudiés par Borin et al. [13]. Ceux-ci ont tenté de caractériser ce surcoût afin d'en identifier les différentes causes. L'outil sur lequel ils basent leur étude est StarDBT [89]. StarDBT est un outil de DBT permettant de traduire des binaires x86 en des binaires x86. Cet outil a été développé pour caractériser les flots de DBT et faciliter la recherche sur ces techniques. Les auteurs ont identifié cinq facteurs de ralentissement : l'augmentation de la mémoire utilisée qui impacte le taux de *cache miss*, l'exécution de *cold-code*, la gestion des branchements indirects dont la destination ne peut être connue lors de la traduction, et enfin le coût du profilage. Les deux derniers points représentent la grande majorité du surcoût. De même, Hiser et al. ont étudié spécifiquement le problème des branchements indirects dans les outils de DBT [46] et proposent plusieurs options pour réduire ce surcoût. Par exemple, l'utilisation d'une table qui garde en mémoire les dernières destinations réduit le coût de ces branchements.

Pour réduire le coût lié à la traduction et à l'optimisation de binaires, il est possible d'utiliser différents niveaux d'optimisation. Ainsi, les transformations les plus coûteuses ne sont appliquées qu'aux points chauds de l'application. C'est la stratégie suivie par DAISY [30] et par Transmeta CMS [25]. Il est également intéressant de constater que QEMU n'utilise pas ces niveaux d'optimisation : parce que son objectif est d'être plus rapide que la simulation, un niveau d'optimisation unique suffit amplement. Notons tout de même les travaux de Hong et al. qui proposent une extension de QEMU avec un second niveau d'optimisation [47]. Pour cela, ils traduisent les binaires vers la représentation intermédiaire de LLVM, ce qui leur permet d'accéder aux transformations de ce compilateur. Ces optimisations ne sont déclenchées que sur les points chauds.

Les travaux de Wang et al. [88] étudient les intéractions entre un outil de compilation dynamique et un outil de traduction dynamique. Par exemple, l'exécution d'une application Android sur une architecture Efficeon de Transmeta. L'existence de deux niveaux de compilation dynamique entraine une perte des informations contenues dans le *bytecode* Android. Les auteurs proposent donc des techniques pour passer l'information directement au *Code Morphing Software* du processeur.

Les travaux de Wu et al. [91] présentent la plateforme TwinPeaks. Celle-ci fait intéragir un processeur à exécution dans le désordre avec peu d'unités d'exécution et un processeur à exécution dans l'ordre offrant beaucoup d'unités d'exécution. La traduction dynamique de binaires est utilisée pour augmenter les performances sur le processeur à exécution dans l'ordre et ce dernier n'est utilisé que si le système estime que l'application a les caractéristiques adéquates.

2.3 La traduction dynamique de binaires pour processeur VLIW

Dans cette section, nous présentons les principaux travaux portant sur la traduction dynamique de binaires pour les architectures VLIW. Notre étude sera élargie à d'autres approches basées sur la compilation dynamique. Afin de pouvoir appréhender la difficulté de la compilation dynamique pour VLIW, nous commençons par présenter les enjeux de la compilation pour des architectures statiquement ordonnancées.

2.3.1 Le défis de la compilation pour processeur VLIW

Compiler pour des architectures VLIW présente plusieurs difficultés. Comme nous l'avons vu dans la section 1.2.2, la philosophie des processeurs VLIW est de déléguer une grande partie du travail au compilateur. En effet, le VLIW ne fait qu'exécuter les *bundles* d'instructions. La compilation pour VLIW a été très étudiée mais, pour des raisons de place, seul les principaux résultats sont discutés dans cette section. La complexification de la chaîne de compilation se fait à deux niveaux : la phase d'optimisation (ou *middle-end*) et la phase de génération de code (ou *back-end*).

Générer du code binaire pour VLIW consiste principalement à ordonnancer les instructions. Ce problème consiste à placer chaque instruction sur une unité d'exécution, à un cycle donné [2, 26]. A ce problème s'ajoute l'allocation de registres, qui consiste à attribuer à chaque variable du programme un registre physique de l'architecture. L'allocation de registres peut également décider de sauvegarder des variables en mémoire pour libérer des registres quand c'est nécessaire (on parle de *spill*) [22, 72, 19]. Le problème de l'allocation de registres n'est pas spécifique aux architectures VLIW et est particulièrement important pour les processeurs x86 qui utilisent très peu de registres. L'allocation présente cependant une difficulté supplémentaire pour les architectures VLIW car elle est inter-dépendante du problème de l'ordonnancement des instructions [39, 71]. Si les instructions sont ordonnancées avant l'allocation de registres, on peut aboutir à des situations où le nombre de registres disponibles ne permet pas de stocker les variables vivantes, et où l'insertion d'instructions de *spill* dégrade l'ordonnancement. Au contraire, si l'allocation de registres est réalisée avant l'ordonnancement des instructions, des dépendances de nom sont créées et contraignent l'ordonnancement. Dans leur formulation classique, les problèmes d'ordonnancement des instructions et d'allocation de registres sont deux problèmes NP-complet

[22].

Notons que les problèmes d'allocation de registres et d'ordonnancement des instructions sont beaucoup moins critiques lorsque le code est destiné à une architecture à exécution dans le désordre. En effet, ces processeurs renomment les registres pour supprimer les dépendances de nom et modifient dynamiquement l'ordonnancement des instructions.

La seconde différence réside dans l'étape d'optimisation du code. Pour que plus de parallélisme d'instructions soit exposé lors de l'ordonnancement des instructions, il est nécessaire de modifier le graphe de flot de contrôle de l'application et de créer des régions contenant plus d'instructions. L'étude de McFarlin et al. (déjà discutée dans le chapitre précédent) s'est interessée au dénombrage, à un moment donné, du nombre de blocs de base différents représentés dans le *Re-Order Buffer* d'un processeur OoO. Cette étude, effectuée sur les applications de SPEC2000, a montré que dans 30% des cycles d'exécution d'un processeur OoO, le ROB contient des instructions provenant d'au moins 5 blocs de base différents. Dans 18% des cycles, les instructions proviennent d'au moins 7 blocs. A certains cycles, jusqu'à 15 blocs sont représentés dans le ROB. Cette étude montre l'efficacité des mécanismes de prédiction de branchement. Ceux-ci permettent au processeur OoO de commencer en avance l'exécution des blocs futurs. Cette capacité du processeur à travailler sur une fenêtre très large lui permet de trouver suffisamment de parallélisme d'instructions à exploiter.

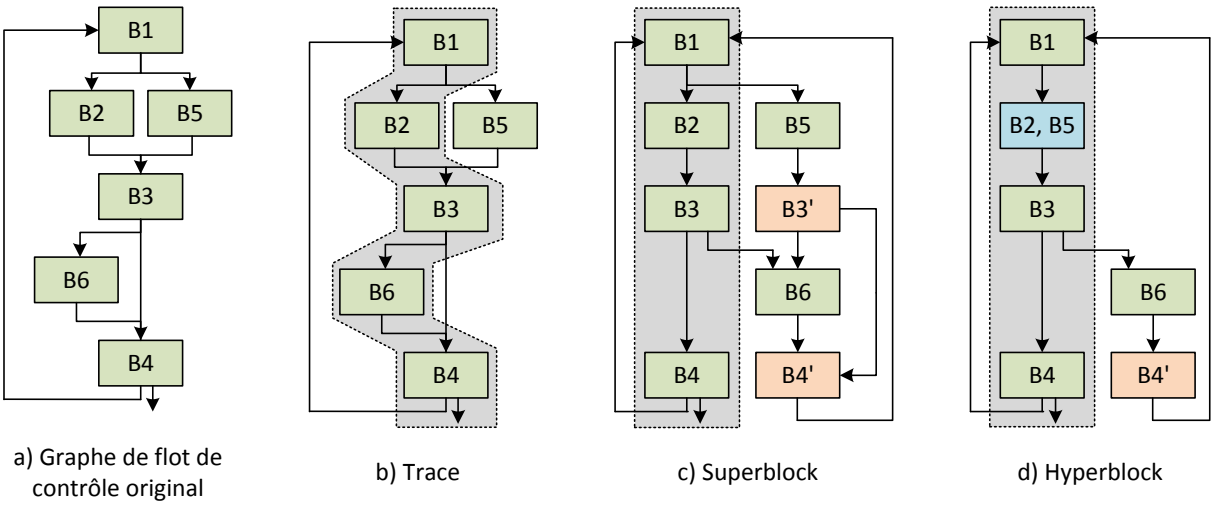


FIGURE 2.1 – Présentation des formes de régions les plus couramment utilisées lors de la compilation pour processeur VLIW. La partie a) donne un exemple de graphe de flot de contrôle et les parties b) c) et d) donnent trois exemples de région possible. Cet exemple est tiré de [34].

Les processeurs VLIW n'ont pas cette capacité et c'est au compilateur de jouer ce rôle. La technique consiste à augmenter la taille des régions à ordonnancer pour offrir plus de parallélisme d'instrutions. La figure 2.1 illustre les différents types de régions pouvant être construites. La partie $a)$ de la figure représente un exemple de graphe de flot de contrôle simple, constitué uniquement de blocs de base (blocs n'ayant qu'un seul point d'entrée et un seul point de sortie). Ce graphe de flot de contrôle peut être modifié pour créer les régions suivantes :

— Une trace est un chemin dans le graphe de flot de contrôle (généralement le chemin le plus

courant) et peut comporter plusieurs points d'entrée et plusieurs points de sortie [33]. La partie $b)$ de la figure 2.1 fournit un exemple d'une telle trace, avec des points de sortie au niveau de `B1` et `B3` et des points d'entrée au niveau de `B1`, `B3` et `B4`.

— Un *superblock* est une trace dans laquelle il n'y a qu'un seul point d'entrée [49]. Lorsqu'une jointure est nécessaire (c'est-à-dire quand deux arcs mènent à un même bloc), le bloc en question est dupliqué. La partie $c)$ de la figure 2.1 est un exemple de construction de *superblock* : les blocs `B3` et `B4` ont été dupliqués pour ne pas ajouter de points d'entrée dans le *superblock*.

— Un *hyperblock* est une région avec une entrée unique, plusieurs sorties et utilisant des instructions prédiquées. Par exemple, les deux branches d'un `If` peuvent être fusionnées, avec des instructions prédiquées pour contrôler la valeur des variables en sortie. La partie $d)$ de la figure 2.1 est un exemple d'*hyperblock* dans lequel les blocs `B2` et `B5` ont été fusionnés, ce qui évite de dupliquer les blocs `B3` et `B4` [60].

Il est à noter que la construction de blocs peut être encore plus complexe et se baser sur des analyses dynamiques et des mécanismes matériels pour créer de très larges groupes d'instructions [92].

L'exécution de boucles représente une grande part du temps d'exécution d'une application. Les compilateurs VLIW utilisent la technique de déroulage de boucle, qui est la continuation directe du principe de *superblock*. Cette technique permet d'augmenter le parallélisme d'instructions disponible et de réduire le nombre d'opérations à effectuer en factorisant certaines instructions. Le déroulage d'une boucle peut être réalisé plusieurs fois, on parle alors de facteur de déroulage. Un facteur de déroulage plus grand permet d'augmenter l'ILP disponible mais également la taille du binaire, impactant ainsi les performances du cache d'instructions : un binaire plus grand augmente le nombre de *cache miss* et peut réduire les peformances. Le meilleur compromis entre l'ILP et la taille du binaire peut être trouvé en essayant différents facteurs de déroulage et en modélisant l'efficacité du cache. Quand il y a plusieurs boucles dans une application, l'approche locale n'est pas suffisante : les différentes boucles ne sont pas exécutées le même nombre de fois et présentent des compromis différents entre l'ILP et la taille du binaire. Les travaux de Heydemann et al. [44] abordent ce problème en deux étapes. Dans un premier temps, l'outil teste différents facteurs de déroulage pour chaque boucle de l'application et mesure l'impact sur les performances et sur la taille du binaire. Ces mesures permettent de pondérer les contributions de chaque boucle. Heydemann et al. utilisent ensuite ces résultats pour construire un modèle de programmation en nombre entier représentant le problème global. La fonction objectif est de maximiser les performances tout en respectant une taille de binaire donnée.

Une autre solution consiste à utiliser des techniques de pipeline logiciel (on parle également de *modulo scheduling* ou de *software pipeline*) [75, 58]. Le principe est de commencer l'itération n d'une boucle avant que l'itération $n-1$ ne soit terminée. L'intervalle d'initiation (ou II) définit la période de démarrage d'une nouvelle itération de la boucle. Dans la suite de ce paragraphe, nous détaillons un algorithme permettant de construire un pipeline logiciel. La première étape consiste à choisir une valeur pour le II. Pour faire ce choix, des bornes inférieures peuvent être calculées en comptant le nombre d'opérations et le nombre d'unités d'exécution ou encore en mesurant la longueur de la plus longue chaîne de dépendance entre deux itérations. Une fois le II choisi, on réalise un ordonnancement modulo. Cela consiste à ordonnancer les instructions du bloc mais en travaillant toujours modulo II : si une

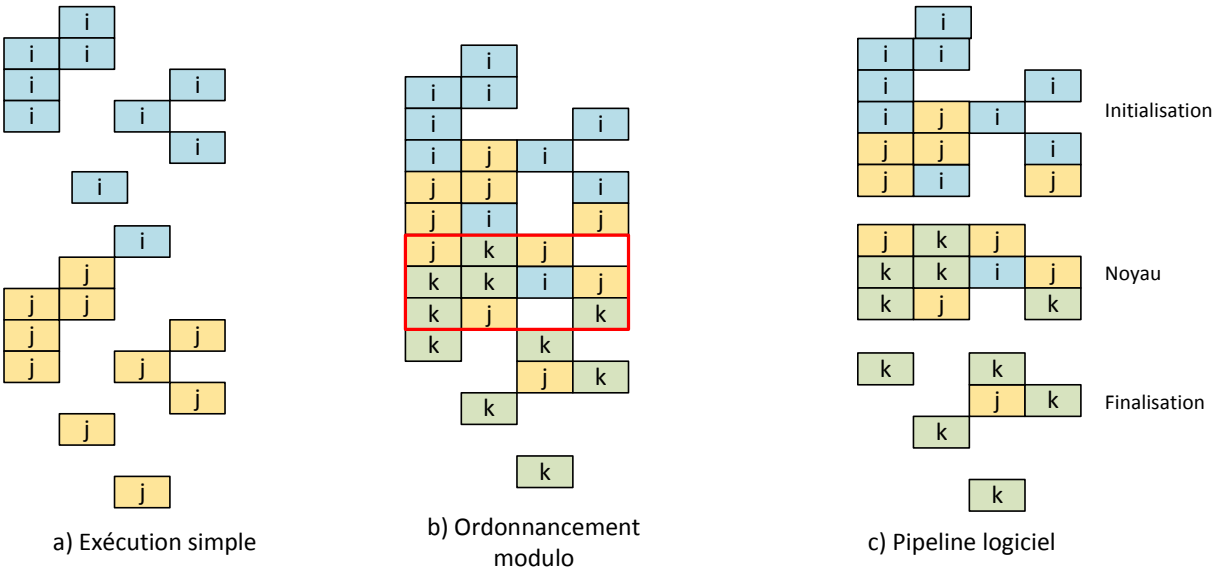


FIGURE 2.2 – Présentation du principe d'exécution pipelinée des boucles. La partie a) de la figure montre une boucle exécutée simplement, la partie b) illustre le principe d'ordonnancement modulo et de la construction du noyau et la partie c) le pipeline logiciel qui en résulte.

instruction est placée au cycle 4, le cycle 4+II est marqué comme occupé. Les différentes instructions du bloc sont ainsi placées sur la fenêtre. S'il n'y a plus d'emplacement disponible pour placer une instruction, l'algorithme choisit un II plus grand et recommence. Si toutes les instructions peuvent être placées, alors on a réussi à construire un noyau, c'est-à-dire un bloc de II cycles contenant toutes les instructions du bloc. L'ordonnancement qui précède le noyau correspond à l'initialisation et est exécuté une fois au début de la boucle. L'ordonnancement après le noyau correspond à la terminaison, et est exécuté une fois à la sortie de la boucle. Le noyau est exécuté pour chaque itération.

La figure 2.2 est une illustration du principe de pipeline logiciel : la partie $a)$ est une boucle exécutée simplement, nécessitant 8 cycles pour une itération donnée. S'il faut exécuter 100 itérations, il faut 800 cycles avec cette version du binaire. La partie $b)$ illustre le principe de l'ordonnancement modulo avec un II de 3. On constate que, pour chaque instruction de l'itération i, la même instruction pour l'itération j est placée 3 cycles plus bas et une troisième instance pour l'itération k est placée 6 cycles plus bas. On remarque également le premier noyau qui apparait sur l'ordonnancement. Enfin, la partie $c)$ montre le pipeline logiciel qui en résulte. Pour exécuter les 100 itérations, il faudra $6 + 98 * 3 + 5 = 305$ cycles.

Le pipeline logiciel est une technique très efficace pour exploiter le parallélisme d'instructions sans augmenter la taille du code. Cependant, sa mise en oeuvre est complexe car les dépendances inter-itérations (dont les dépendances mémoire) doivent être traitées soigneusement.

La compilation pour processeur VLIW nécessite d'extraire et d'expliciter le parallélisme d'instructions. Ces deux tâches sont complexes pour un compilateur statique travaillant sur un programme C. Dans le cas de la DBT, ces transformations sont réalisées dynamiquement et l'outil n'a que peu d'informations sur l'application en cours d'exécution, puisqu'il travaille sur un binaire. Malgré ces défis, plusieurs outils de traduction dynamique de binaire visant des architectures VLIW ont été développés et industrialisés.

Ces outils sont présentés en détail dans la prochaine sous-section.

2.3.2 Etudes des outils de DBT pour VLIW

Nous avons vu dans la partie précédente que la compilation pour une machine VLIW peut être complexe. Dans le cadre de la DBT, les analyses et les transformations doivent être réalisées dynamiquement, avec pour seules informations une représentation binaire de l'application. Malgré ces difficultés, plusieurs approches de DBT ciblant des architectures VLIW ont été développées. Dans cette section, nous étudions les outils suivants :

— l'outil DAISY de IBM qui permet d'abstraire l'architecture du VLIW utilisée et d'offrir une compatibilité binaire entre différentes générations de processeur [30, 31, 83],
— le *Code Morphing Software* (CMS) de Transmeta qui permet d'exécuter des applications x86 sur une architecture VLIW pour réduire la consommation énergétique [25],
— le *IA32 Execution Layer* (IA32EL) de Intel permettant d'exécuter des applications x86 sur les processeurs Itanium [10],
— le processeur Denver de NVidia, capable d'exécuter des binaires Arm sur son architecture superscalaire à exécution dans l'ordre [12].

Ces quatre approches sont étudiées en détail dans ce document.

Il existe également des travaux sur la compilation JIT ciblant des architectures VLIW. Par exemple, Agosta et al. [3] ont proposé un outil de traduction du *bytecode* Java vers un VLIW. Leur outil est basé sur une heuristique de *list scheduling* pour l'ordonnancement des instructions et sur une allocation gloutonne des registres. Dupont de Dinechin a également développé un outil de compilation JIT pour le *bytecode* CLI, ciblant un processeur VLIW [26]. Ses travaux se sont concentrés sur l'ordonnancement des instructions, la partie allocation de registres étant déjà résolue dans le flot. L'approche est basée sur l'heuristique du *scoreboard scheduling*, et Dupont de Dinechin propose une extension permettant d'optimiser l'ordonnancement entre des blocs voisins dans le graphe de flot de contrôle.

Il existe également des approches de compilation dynamique ciblant des architectures CGRA. Les CGRA sont des architectures statiquement ordonnancées n'utilisant pas de bancs de registres. Les opérandes sont acheminés sur un réseau d'interconnections programmable. La compilation pour ces architectures demande non seulement d'expliciter le parallélisme d'instructions, mais aussi de déterminer un schéma de routage pour ces opérandes. Ce problème s'apparente au placement routage des LUTs dans ls FPGA et les algorithmes utilisés sont donc très complexes [68].

L'approche de Watkins et al. [90] propose de combiner un processeur à exécution dans le désordre et un CGRA. Cependant, l'approche utilise un outil de DBT pour générer les binaires. Etant donnée la complexité de la compilation pour CGRA, seules les parties les plus critiques de l'application sont migrées sur CGRA. Le reste peut s'exécuter nativement sur le OoO. Les auteurs s'affranchissent ainsi de la pénalité liée au *cold-code*.

Pour limiter le coût de la traduction dynamique, les outils de DBT ciblant les processeurs VLIW se basent sur une traduction incrémentale. L'objectif est le même que dans la compilation dynamique : chaque cycle passé à optimiser du code doit être compensé par l'accélération due à l'optimisation. Ce phénomène est d'autant plus important lorsque l'architecture ciblée est un processeur VLIW, étant

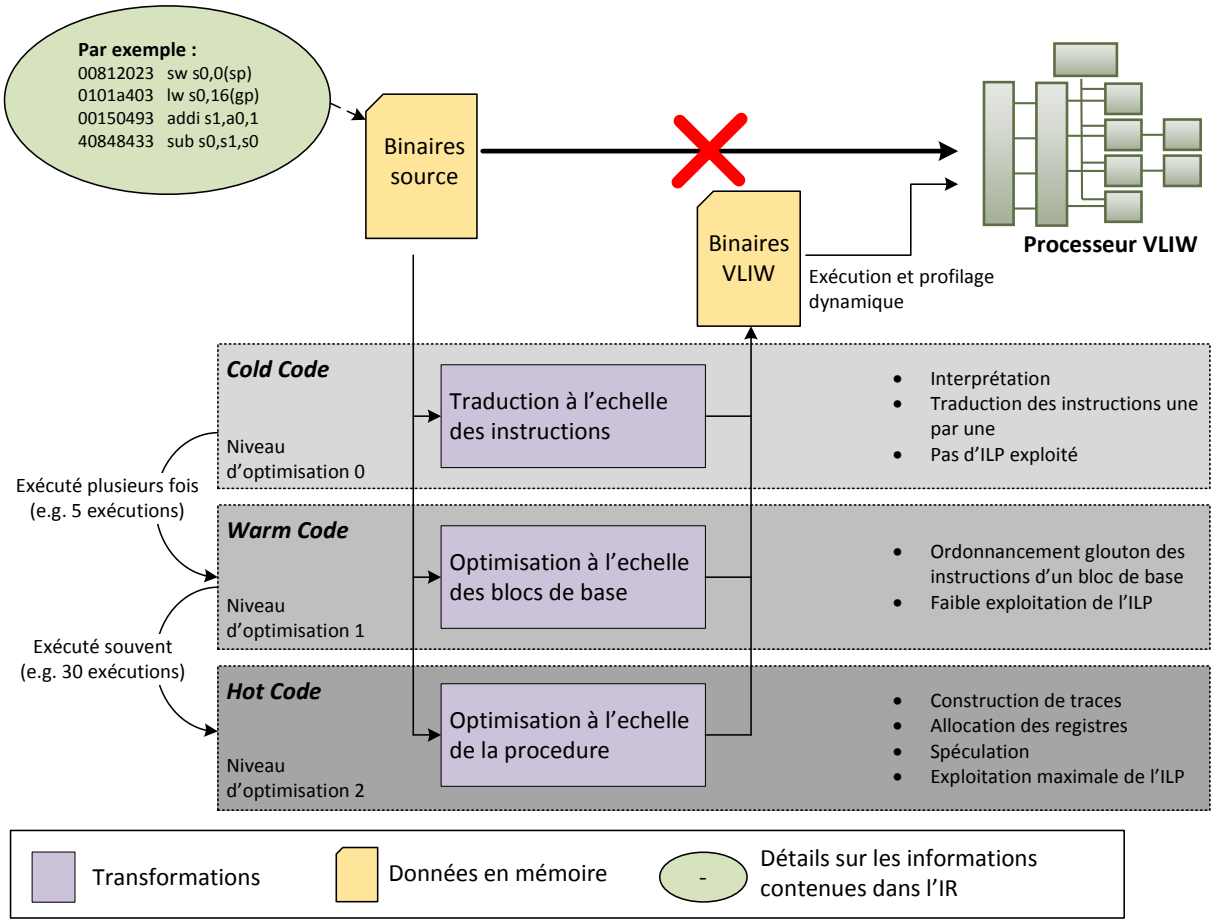


FIGURE 2.3 – Exemple d'un flot standard de traduction dynamique de binaire pour processeur VLIW. Le binaire source n'est pas compatible avec le processeur VLIW et ne peut pas être exécuté directement. L'outil de traduction dynamique de binaire va appliquer différents niveaux d'optimisation aux blocs de ce binaire, selon leur niveau d'utilisation. Chaque niveau d'optimisation est plus coûteux mais mène à de meilleures performances.

donné l'importance et la complexité des phases de compilation. La figure 2.3 donne un exemple d'organisation d'un tel flot : la traduction et l'optimisation du binaire est découpée en trois niveaux. Le niveau d'optimisation 0 s'applique au *cold code* et consiste à commencer l'exécution le plus rapidement possible. Les outils vont alors interpréter les binaires source ou procéder à une traduction naïve de chaque instruction individuellement. Le niveau d'optimisation 1 s'applique au *warm code* et commence à exploiter le parallélisme d'instructions : l'outil réalise un ordonnancement des instructions à l'échelle du bloc de base (une dizaine d'instructions environ). Enfin, le niveau d'optimisation 2 n'est appliqué que sur le *hot code* et réalise les transformations les plus coûteuses car les binaires générés sont susceptibles d'être exécutés souvent. Plusieurs optimisations inter-blocs (déroulage de boucles, construction de *superblock* ou d'*hyperblock*) sont appliquées pour extraire du parallélisme d'instructions et l'outil réalise un ordonnancement des instructions plus fin.

La suite de cette section est consacrée à une étude détaillée de quatre outils de traduction dyna-

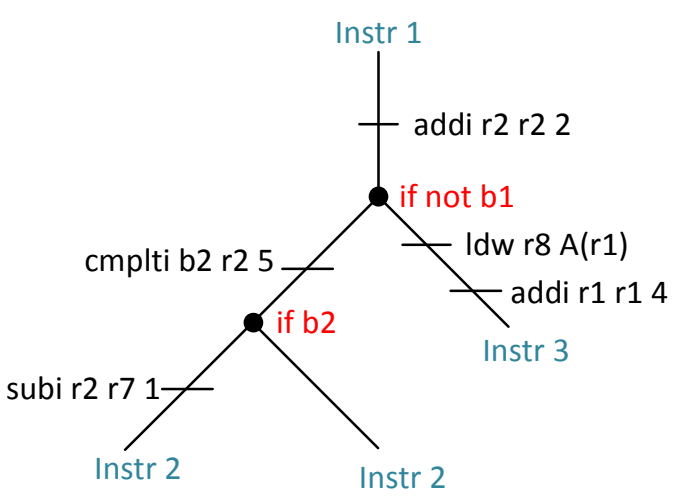
	IBM DAISY	**Transmeta CMS**	**Intel IA32EL**	**NVidia Denver**
Année	1997	2003	2003	2015
ISA Source	PowerPC	x86	x86	Armv8
Architecture de destination	VLIW	VLIW	Itanium	Superscalaire dans l'ordre
Cold Code	Interprétation	Interprétation	Traduction naïve basée sur des patterns	Décodeur matériel
Warm Code	Construction de *Tree Regions*, spéculation et ordonnancement adaptatif	Génération naïve de binaires	Construction d'hyperblocs, analyse du x86, génération et optimisation d'une représentation intermédiaire, ordonnancement	Ordonnancement dans l'ordre
Hot Code		Optimisations inter-bloc logicielles		Optimisations et ordonancement logiciel

TABLE 2.1 – Présentation des principales caractéristiques des outils de traduction dynamique de binaire pour VLIW.

mique de binaires ciblant des architectures VLIW. Afin de faciliter leur comparaison, le tableau 2.1 offre une vue qualitative des différentes approches. Celles-ci sont étudiées et commentées dans l'ordre chronologique de leur dévelopement. Il est important de noter que ces quatre approches sont des outils industriels. Par conséquent, très peu d'informations ont été communiquées sur leur implémentation.

L'outil DAISY de IBM

L'outil de traduction dynamique DAISY de IBM a été proposé dans le but d'offrir une compatibilité binaire entre différentes générations de processeurs VLIW [30, 31, 83]. Contrairement aux autres approches présentées dans cette section, DAISY se base sur une architecture de VLIW arborescente. Les instructions ne sont pas regroupées en *bundles* mais en arbres dont les différentes branches correspondent à différents prédicats.




```
1  addi r2 r2 2 ;
2  if not b1 then
3  |  cmplti b2 r2 5 ;
4  |  if b2 then
5  |  |  subi r2 r7 1 ;
6  |  |  goto instr2 ;
7  |  end
8  |  else
9  |  |  goto instr2 ;
10 |  end
11 end
12 else
13 |  ldw r8 A(r1) ;
14 |  addi r1 r1 4 ;
15 |  goto instr3 ;
16 end
```

FIGURE 2.4 – Exemple d'une instruction VLIW générée par DAISY ainsi que son interprétation.

La figure 2.4 illustre ce principe. Les noeuds correspondent à des tests sur des registres 1 bit, les branches correspondent aux différents résultats du test. La partie droite de la figure donne une description plus précise de la sémantique de l'instruction VLIW présentée. L'utilisation de ce type de VLIW complexifie l'architecture matérielle mais simplifie la spéculation. En effet, l'ordonnancement des *hyperblock* ou des traces est simplifié par les prédicats. De plus, des chaînes de décision complexes peuvent être résolues en un seul cycle grâce à la structure en arbre. L'architecture matérielle offre également des mécanismes pour permettre la spéculation : un mécanisme de vérification d'adresse permet d'exécuter des chargements mémoire avant des écritures mémoires ; un banc de registres deux fois plus grande que celle du PowerPC permet de renommer les registres facilement.

DAISY est organisé en deux étapes : les instructions PowerPC sont tout d'abord interprétées puis traduites et ordonnancées quand la région a déjà été exécutée quelques fois. L'algorithme d'ordonnancement utilisé est une heuristique gloutonne qui considère les instructions source une par une et les place en tant que feuille de l'arbre actuel. Si les opérandes sont disponibles, l'instruction est ordonnancée aussi tôt que possible et son résultat est placé dans un registre renommé. Un *move* est ensuite placé en tant que feuille de l'arbre actuel.

Afin de réduire le coût de la traduction et de l'optimisation, l'ordonnancement généré par DAISY est adapté au cours du temps. Les premiers ordonnancements générés ne sont pas très complexes et, en fonction du comportement dynamique du programme, des techniques plus coûteuses sont appliquées. L'étude expérimentale montre qu'il faut, en moyenne, 4 000 cycles pour traduire une instruction PowerPC. Cette même étude montre que DAISY atteint 80% des performances obtenues grâce à un compilateur statique.

Les différents travaux sur DAISY sont également très précis sur l'organisation du système : organisation de la mémoire pour distinguer les zones traduites, gestion des branchements en utilisant un mécanisme appelé le *VLIW Instruction Translation Lookaside Buffer*, gestion des exceptions précises et du code auto-modifiant et gestion du cache de traduction. Les études expérimentales sont également très précises mais la structure très particulière du VLIW rend toute comparaison difficile.

Le *Code Morphing Software* de Transmeta

L'objectif de Transmeta est d'exécuter des binaires x86 sur une architecture VLIW. Leur outil de traduction dynamique de binaires, le CMS (*Code Morphing Software*) est capable d'interpréter, de traduire et d'optimiser les binaires x86. Le CMS a été utilisé dans deux générations de processeurs VLIW :

— Le processeur Crusoe, un processeur VLIW à quatre voies commercialisé en 2000. [52] ;

— Le processeur Efficeon, un VLIW à huit voies commercialisé en 2003.

Les leçons tirées par David Ditzel, l'un des fondateurs de Transmeta, ont été résumées lors d'une conférence sur la traduction de binaires en 2008 [28]. Une des principales conclusions est que le processeur doit être développé conjointement avec l'outil de traduction afin de permettre une spéculation peu coûteuse et agressive. L'approche adoptée par Transmeta est l'utilisation de *shadow registers* qui correspondent à une copie cachée de chaque registre présent dans le x86. Ces registres permettent de stocker le résultat des instructions dans les zones de spéculation. Si l'hypothèse qui a été prise pour spéculer se révèle fausse, les valeurs stockées dans ces registres peuvent être détruites et les valeurs

stockées dans les registres originaux restent inchangées (on parle de *roll-back*). Si la spéculation est correcte, les valeurs sont copiées dans les registres originaux (on parle de *commit*). Ces mécanismes permettent de simplifier la spéculation logicielle.

Le flot de traduction et d'optimisation de CMS est organisé en quatre niveaux :

— Le niveau 0 correspond à l'interprétation des instructions. L'outil réalise en même temps un profilage de l'application et des différents branchements exécutés.

— Le niveau d'optimisation 1 crée une première traduction des binaires pour des régions de codes allant jusqu'à 100 instructions x86. Des optimisations légères et un ordonnancement glouton des instructions sont réalisés.

— Le niveau d'optimisation 2 consiste à effectuer des optimisations plus complexes (élimination de sous expressions communes, réorganisation des accès mémoire) sur la région. Un ordonnancement plus fin du chemin critique est réalisé.

— Le dernier niveau d'optimisation va fusionner différentes régions et réaliser les optimisations les plus complexes.

Enfin, l'article de Dehnert et al. présentant le fonctionnement de CMS traite également de la gestion de différents points nécessaires pour respecter la spécification du x86 : la gestion correcte des interruptions x86 et la gestion des codes auto-modifiants [25].

L'outil *IA32 Execution Layer* de Intel

L'outil *IA32 Execution Layer* permet d'exécuter des applications x86 sur un processeur Itanium [10] (voir Section 1.2.3 pour plus de détails sur ces processeurs). Avant cet outil, les processeurs Itanium utilisaient un mécanisme matériel pour réaliser une traduction naïve des binaires x86. Contrairement aux autres approches présentées ici, cet outil de traduction dynamique ne travaille qu'au niveau des applications et la partie système est déployée nativement, en IA64.

L'outil est organisé selon deux niveaux d'optimisation :

— Dans un premier temps, pour gérer le *cold code*, l'outil travaille au niveau des blocs de base (4-5 instructions x86). Une première traduction naïve des instructions est générée en se basant sur des motifs prédéterminés. L'outil réalise également une analyse de la durée de vie des différents bits de *flag* utilisés dans le x86. Pendant cette première étape, des instructions de profilage sont insérées pour compter le nombre d'exécutions de chaque bloc ainsi que les décisions des branchements.

— Lorsqu'une partie de l'application est exécutée régulièrement, le second niveau d'optimisation est déclenché. L'outil utilise alors les informations de profilage pour construire un hyperblock (environ 50 instructions x86) sur lequel les optimisations seront appliquées. L'outil ré-analyse ensuite le binaire x86 pour construire une représentation intermédiaire, qui est ensuite optimisée (gestion des accès non alignés, propagation de constantes, analyse de la durée de vie des bits de *flag* et analyse plus fine des instructions SIMD et FP). Enfin, la représentation intermédiaire est ordonnancée dans des *bundles* en suivant les contraintes et les capacités de spéculation des processeurs Itaniums [82].

Les quelques résultats expérimentaux fournis montrent que l'utilisation de IA32El permet d'atteindre

65% des performances obtenues en compilant les applications directement pour Itanium. Contrairement au Denver de NVidia et au CMS de Transmeta, l'outil proposé par Intel est utilisé pour faire fonctionner des binaires x86 sur un processeur existant. Par conséquent, les développeurs n'ont pas pu utiliser une approche de *co-design* logiciel/matériel pour réduire le coût de leur traduction.

L'architecture Denver de NVidia

Dans la continuité de Transmeta, NVidia a introduit en 2011 l'architecture Denver. Le principe est d'exécuter des binaires ARMv8 sur une architecture superscalaire à exécution dans l'ordre à 7 voies [12]. Le jeu d'instructions de ce processeur est différent du ARM et une phase de traduction et d'optimisation de binaires est utilisée. L'une des particularités du Denver est l'utilisation d'un composant matériel pour traduire les instructions ou pour les ordonnancer dans l'ordre.

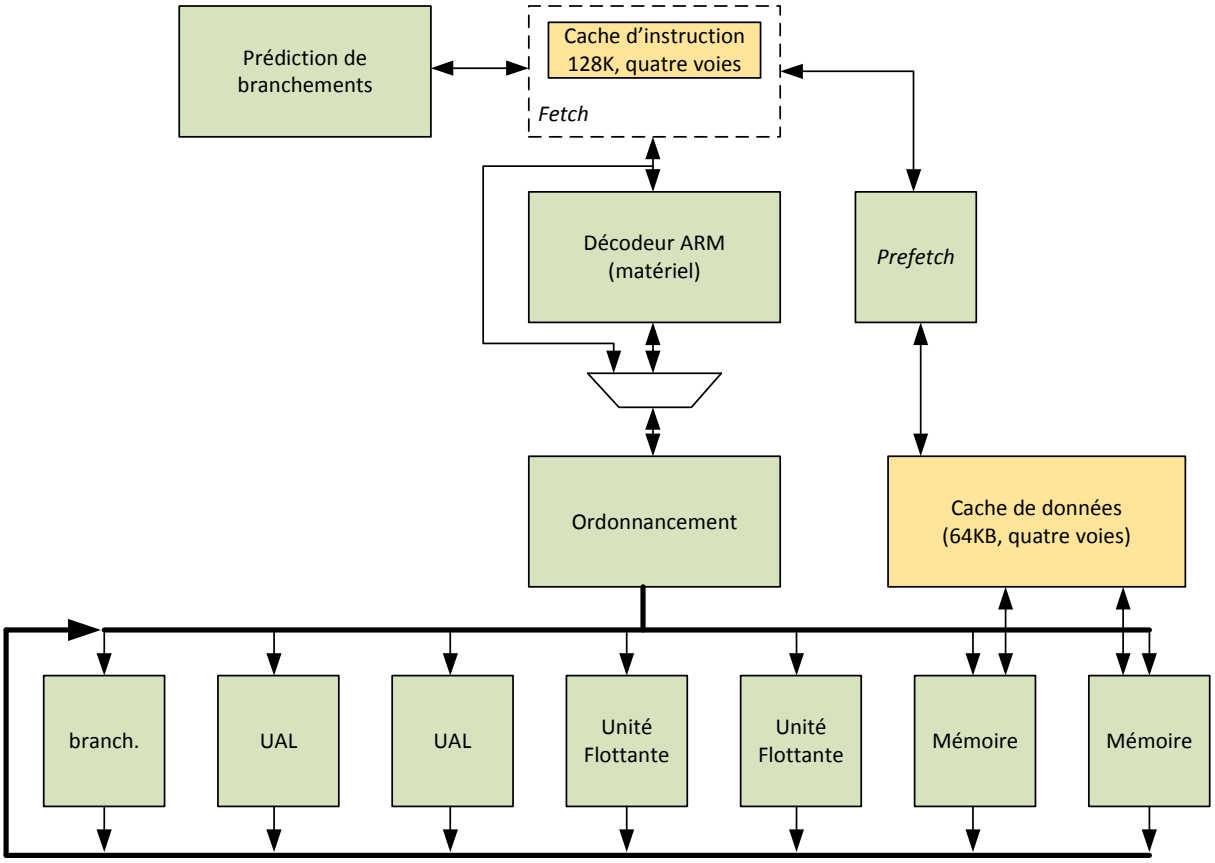


FIGURE 2.5 – Présentation de l'architecture Denver de NVidia. Figure extraite de l'article de Boggs et al. [12]

La figure 2.5 représente l'architecture Denver. Le système se base sur un processeur superscalaire à exécution dans l'ordre disposant de 7 unités d'exécution. L'architecture est également dotée d'un décodeur d'instructions ARMv8 capable de décoder jusqu'à deux instructions par cycle.

Le système de DBT de Denver est organisé en trois niveaux d'optimisation :

— Lorsqu'une portion de binaire est exécutée pour la première fois, le décodeur matériel ARM est utilisé. Il permet de décoder jusqu'à deux instructions par cycle, qui sont ensuite exécutées par l'architecture superscalaire. La version décodée des binaires est sauvegardée dans un cache d'instructions.
— A partir de la deuxième exécution d'une portion de binaires, une version traduite est présente dans le cache d'instructions. Le décodeur ARM est donc désactivé et le mécanisme d'ordonnancement dans l'ordre peut exploiter plus de parallélisme d'instructions.
— Les portions de binaire les plus critiques sont analysées par un processus logiciel. Celui-ci travaille directement sur le cache d'instructions et réordonnance les instructions afin d'augmenter les performances de l'architecture à exécution dans l'ordre.

L'approche utilisée par Denver permet de gérer très efficacement le *cold code* grâce au décodeur matériel et permet également d'exploiter très tôt un parallélisme d'instructions limité, grâce au mécanisme d'ordonnancement dans l'ordre. C'est un exemple d'accélération matériel du procédé de traduction dynamique de binaire. Cette architecture est encore commercialisée et il y a peu de détails sur les optimisations logicielles qui sont réalisées.

Dans ce chapitre, nous avons présenté les techniques de compilation dynamique et, plus spécifiquement, la traduction dynamique de binaires. Nous avons également abordé les principaux défis posés lors de la compilation pour un processeur VLIW et étudié différents outils de DBT ciblant de telles architectures. L'un des défis de la compilation dynamique est de limiter l'impact du temps de compilation sur le temps d'exécution. Dans le prochain chapitre de ce document, nous présentons Hybrid-DBT, un outil de DBT pour processeur VLIW utilisant trois accélérateurs matériels pour réduire le coût des parties critiques de la traduction.

CHAPITRE 3

HYBRID-DBT : COMPILATION HYBRIDE LOGICIELLE/MATÉRIELLE POUR VLIW

Dans les deux chapitres précédents, nous avons étudié la traduction dynamique de binaires et conclu que cette technique permet d'ajouter des processeurs VLIW dans des systèmes multi-cœurs hétérogènes à jeu d'instructions unique. Le surcoût de la DBT doit toutefois être minimisé afin de conserver le niveau de performance et l'efficacité énergétique de ces processeurs.

Dans ce chapitre, nous présentons la principale contribution de cette thèse, une chaîne de traduction dynamique de binaires permettant d'exécuter des binaires RISC-V sur des architectures VLIW. Pour réduire le coût de cette traduction, Hybrid-DBT utilise trois accélérateurs matériels permettant de réaliser les phases critiques de la traduction rapidement et à moindre coût énergétique. De plus, cet outil est *open-source*[1] pour faciliter la reproductibilité des résultats et permettre à la communauté académique de travailler sur ce type de machines.

Nous commencons par présenter une vision globale de l'outil et des hypothèses posées dans la section 3.1. Une présentation plus précise du flot de traduction et d'optimisation est fournie dans la section 3.2. La section 3.3 fournit une description détaillée des trois composants matériels développés pour accélérer la traduction. Enfin, la section 3.4 présente les résultats expérimentaux qui démontrent l'intérêt de notre approche.

3.1 Présentation de l'outil Hybrid-DBT

Dans cette section, nous présentons une vision d'ensemble de notre plateforme. L'objectif est également d'expliciter nos choix d'implémentation. Nous commençons par présenter l'organisation matérielle de la plateforme en décrivant les différents éléments du système, puis nous discutons du choix de partitionnement logiciel/matériel fait dans notre outil. Enfin nous abordons en détail le choix de la représentation intermédiaire.

3.1.1 Vue d'ensemble de la plateforme

La plateforme Hybrid-DBT permet d'exécuter des binaires RISC-V sur une architecture statiquement ordonnancée : des processeurs VLIW. Pour ce faire, elle utilise une combinaison d'accélérateurs matériels et de transformations logicielles.

1. https ://github.com/srokicki/HybridDBT

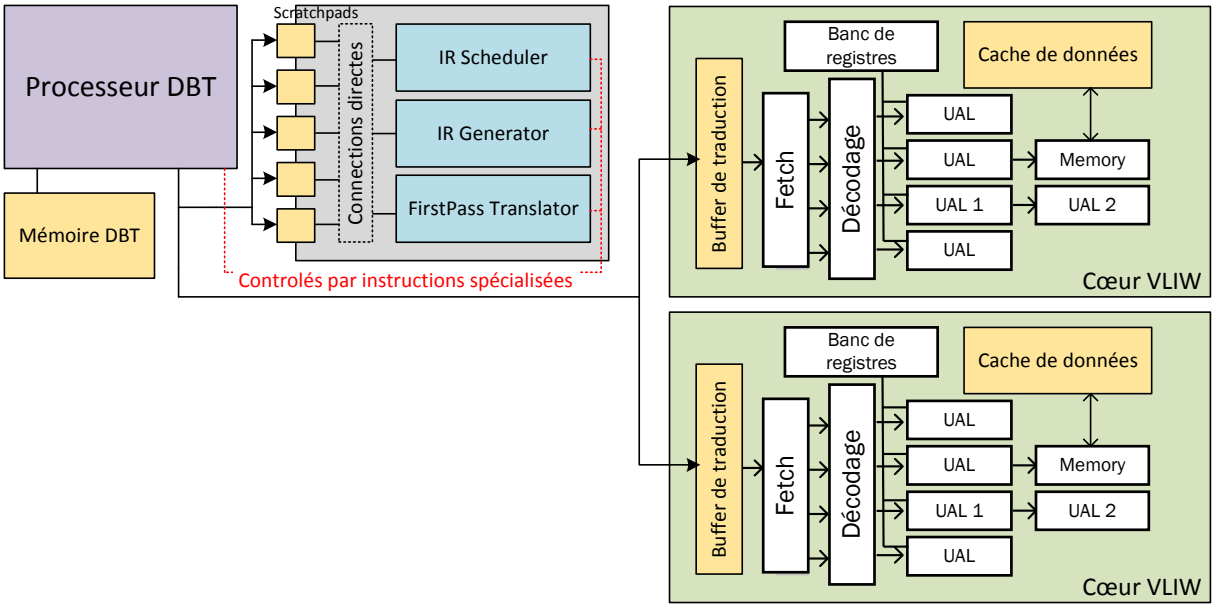


FIGURE 3.1 – Schéma de l'organisation globale d'un système Hybrid-DBT composé du processeur DBT et de ses trois accélérateurs matériels. Il est accompagné par deux processeurs VLIW en charge de l'exécution des applications.

Dans ce système, nous avons choisi d'utiliser un processeur basse consommation dédié à la traduction du binaire. Ce choix permet de réaliser les traductions et les optimisations en avance de phase, pendant que le VLIW exécute une version naïve des binaires et ainsi de masquer une partie du coût de traduction. Cela simplifie également le partage des accélérateurs entre plusieurs VLIW puisque c'est le processeur DBT qui arbitre le partage de la ressource (la DBT pourrait également être gérée par le processeur VLIW en charge d'exécuter l'application, ou par un autre cœur du système). Nous avons utilisé un processeur dédié dans les premières versions de ces travaux et n'avons pas exploré les autres possibilités.

L'architecture de la plateforme est représentée sur la figure 3.1 et est composée des éléments suivants :

— Plusieurs processeurs VLIW sont utilisés pour l'exécution de l'application. Ceux-ci sont conçus sur le modèle défini par Fisher et al. et sont, de fait, très proches d'un ST200. Ils sont composés de 4 voies d'exécution et de 64 registres de 64 bits. Les différentes voies d'exécution sont pipelinées à 4 ou 5 étages selon leur spécialisation. Aucun mécanisme de détection d'aléas de pipeline ou de *forwarding* n'est utilisé. Le système de traduction dynamique de binaires peut être partagé entre plusieurs cœurs d'exécution.
— Le processeur DBT est un processeur pipeliné qui est chargé de la traduction et de l'optimisation des binaires, en parallèle de leur exécution. Ce processeur dispose d'un accès direct au buffer de traduction du VLIW pour y stocker les binaires traduits. Il dispose également d'un accès à la mémoire de données pour y récupérer les informations de profilage.
— Enfin, le système embarque trois accélérateurs permettant de réduire le coût de la traduction.

Nous avons également ajouté un ensemble de mémoires permettant d'échanger des informations entre le processeur DBT et ces accélérateurs. Le processeur DBT peut réaliser des accès mémoire pour modifier leur contenu tandis que les accélérateurs disposent d'un accès direct à chaque banc mémoire. Ces mémoires sont utilisées pour communiquer avec le processeur DBT qui envoie les données d'entrée, démarrer l'accélérateur et accéder aux données de sortie. L'organisation et le fonctionnement de ces différents accélérateurs sont décrits dans la section 3.3.

3.1.2 Partitionnement logiciel/matériel

Comme nous l'avons vu précédemment, Hybrid-DBT utilise trois accélérateurs matériels pour réduire le coût du processus de traduction et d'optimisation. Un des points clés de notre approche est le choix du partitionnement logiciel/matériel au niveau des différentes étapes du flot de traduction. Nous avons suivi plusieurs principes pour prendre ces décisions :

— La compilation dynamique est basée sur le principe que chaque cycle et chaque joule dépensés pour traduire ou optimiser le code doivent être compensés par le gain provenant du code optimisé. Or, lorsque l'outil traduit du *cold-code*, il ne peut pas savoir combien de fois ce code sera exécuté. L'outil dépense donc le moins de temps possible à le traduire et à l'optimiser. Les transformations qui ont été accélérées sont celles utilisées dans les premiers niveaux du processus. En effet, ce sont les parties les plus critiques et les plus largement utilisées.

— Si un accélérateur matériel permet d'obtenir de meilleures performances (à la fois en temps d'exécution et en consommation énergétique), il est cependant beaucoup moins flexible. Ainsi, les transformations ayant de nombreux cas particuliers à gérer ne sont pas de bonnes candidates pour une accélération matérielle. Les transformations accélérées sont basées sur des algorithmes simples et plusieurs hypothèses permettent d'ignorer des cas particuliers.

Dans notre système, l'accélérateur appelé *First-Pass Translator* est chargé de réaliser une première traduction des binaires aussi peu coûteuse que possible. L'accélérateur appelé *IR Generator* est chargé d'analyser ces binaires pour construire une représentation de plus haut niveau de chaque bloc de base de l'application (en construisant notamment son graphe de flot de données). Enfin l'accélérateur *IR Scheduler* va utiliser cette représentation intermédiaire et effectuer un ordonnancement des instructions sur les différentes voies d'exécution du VLIW.

Toutes les autres optimisations de notre flot interviennent à des niveaux d'optimisation plus élevés. Par exemple, la construction de *superblock* ou la réallocation des registres travaillent à l'échelle des sous programmes. Ces optimisations sont plus complexes et ne sont pas utilisées systématiquement (elles dépendent de la forme du code). Par exemple, une optimisation de déroulage de boucle n'est appliquée que lorsqu'une boucle est identifiée dans un *hotspot*. Ces transformations sont donc préférablement appliquées de manière logicielle par le processeur en charge de la DBT.

Un autre élément clé de notre approche est liée au fait que les accélérateurs matériels sont plus efficaces sur des structures de données de type tableaux, impliquant des motifs d'accès relativement simples. Suivant le principe du co-design, nous avons donc défini une représentation intermédiaire dans l'optique de "faciliter" l'utilisation d'accélérateurs. Cette représentation intermédiaire, ainsi que les hypothèses qui ont été posées lors de son développement, sont décrites ci-après, car il est nécessaire

de bien les comprendre avant d'appréhender la mise en œuvre des accélérateurs.

3.1.3 Choix de la représentation intermédiaire

Comme pour tout compilateur, le flot de traduction et d'optimisation de Hybrid-DBT est basé sur l'utilisation d'une représentation intermédiaire (*Intermediate Representation* (IR)). Le choix de cette représentation a été influencé par deux contraintes :

— la représentation intermédiaire est générée et utilisée par des accélérateurs matériels. Par conséquent, elle a été pensée pour être aussi régulière que possible et éviter d'utiliser des structures de données basées sur des objets ou des pointeurs,

— la représentation intermédiaire a été développée dans le but de simplifier certaines transformations logicielles (par exemple le déroulage de boucle, l'allocation de registres ou la construction de traces).

L'ordonnancement des instructions sur les unités d'exécution du VLIW et l'allocation de registres sont des étapes critiques du flot. En effet, ces transformations sont coûteuses et sont exécutées très souvent. L'objectif principal de l'IR est de réduire le coût de ces deux étapes. Par conséquent, le graphe de flot de données (DFG pour *Data-Flow Graph*) de chaque bloc de base a été encodé directement dans la représentation : pour chaque instruction d'un bloc, l'IR encode explicitement la liste des instructions devant être exécutées avant celle-ci ainsi que le nombre d'instructions qui dépendent de celle-ci.

Dépendances de donnée et dépendances de contrôle

La représentation intermédiaire que nous utilisons dans Hybrid-DBT encode complètement le graphe de flot de données d'un bloc (bloc de base, trace, *superblock*, ...). Ainsi, pour chaque instruction, la liste de ses prédécesseurs dans ce graphe est accessible. Deux types de dépendances sont encodés :

— Une dépendance de donnée entre une instruction $i1$ et une instruction $i2$ signifie que $i2$ a besoin du résultat de $i1$. Lors de l'ordonnancement, il faudra s'assurer que $i2$ est ordonnancée après $i1$ à une distance minimale qui dépend de la voie d'exécution sur laquelle $i1$ est placée.

— Les dépendances de contrôle servent à capturer toutes les autres contraintes liées à l'organisation séquentielle du code. Par exemple, elles aident à garantir que les instructions d'accès à la mémoire sont exécutées dans le même ordre que dans le code d'origine. Elles permettent également de s'assurer que certaines instructions sont placées après une instruction de branchement dans le cas des traces. Contrairement aux dépendances de donnée, les dépendances de contrôle n'entrainent pas de distance minimale lors de l'ordonnancement. S'il y a une dépendance de contrôle entre une instruction $i1$ et une instruction $i2$, il faut simplement que $i2$ soit ordonnancée strictement après $i1$.

Les dépendances de nom (WAR pour *Write after Read* et WAW pour *Write after Write*) ne sont pas encodées dans l'IR. Comme nous allons le voir dans la prochaine sous-section, l'allocation de registres est partiellement résolue lors de l'ordonnancement des instructions. Par conséquent, ces dépendances doivent être calculées et respectées pendant l'ordonnancement. Dans l'exemple de la figure 3.4, on peut voir que l'IR ne contient pas de dépendance WAW entre les instructions 2 et 6 qui écrivent dans le

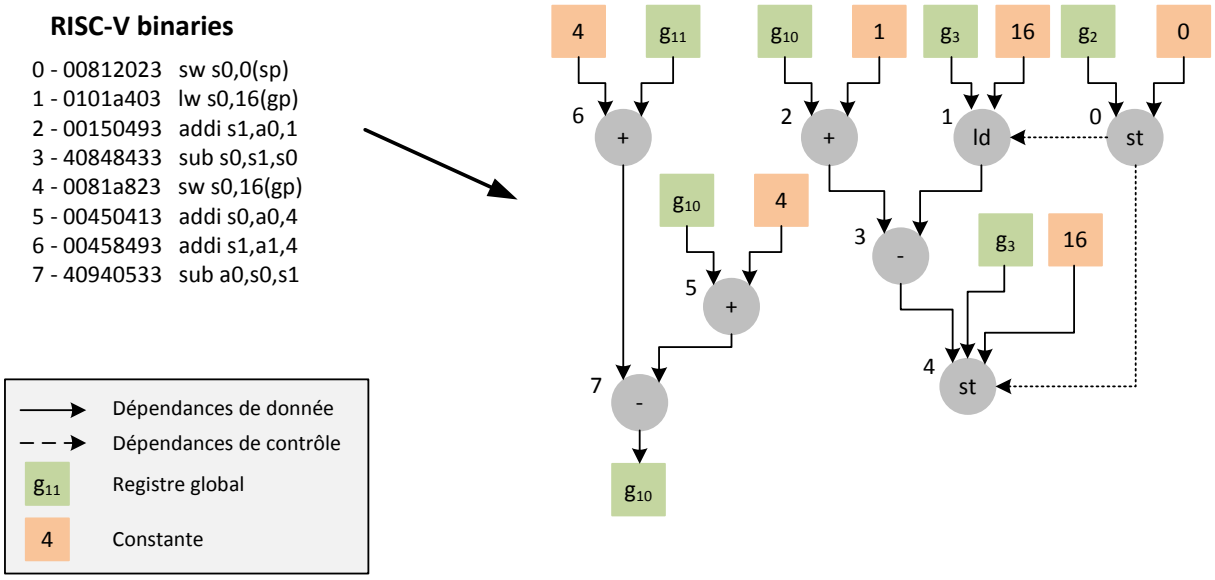


FIGURE 3.2 – Exemple de graphe de flot de données encodé dans un bloc de la représentation intermédiaire. Deux types de dépendances sont représentés : les dépendances de donnée et les dépendances de contrôle.

même registre et pas de dépendance WAR entre les instructions 4 et 5 alors que 5 écrit dans un registre lu par 4.

La figure 3.2 est un exemple de graphe de flot de données. Les dépendances de donnée sont représentées en trait plein tandis que les dépendances de contrôle sont en pointillés. On peut par exemple observer qu'une dépendance de contrôle permet de s'assurer que l'instruction d'écriture en mémoire 0 sera ordonnancée avant l'instruction de lecture en mémoire 1. En effet, puisqu'on ne connait pas les adresses des accès mémoire réalisés par les instructions 0 et 1, il est nécessaire de les exécuter dans leur ordre d'origine pour être conservatif.

Gestion de l'allocation de registre

Dans la représentation intermédiaire, nous avons posé plusieurs hypothèses sur la manière de réaliser l'allocation des registres qui permettent de simplifier sa mise en oeuvre, ainsi que l'ordonnancement des instructions. Ainsi, quand une instruction utilise une valeur, elle peut référencer un **résultat local** qui correspond à un résultat produit dans le même bloc, ou un **résultat global** qui correspond à un résultat produit dans un autre bloc (ou dans une autre itération du même bloc). De même, lorsqu'une instruction produit un résultat, elle peut le stocker dans un **registre temporaire** ou dans un **registre global**. Un registre temporaire ne peut être accédé que dans le bloc courant (l'information du registre physique qui lui a été alloué n'est pas conservée). A l'opposé, un registre global peut être utilisé comme un résultat global dans un autre bloc. Attention, une instruction peut accéder à un résultat local (produit dans le même bloc) stocké dans un registre global (car réutilisé dans un autre bloc).

Par défaut, chaque instruction d'un bloc stocke son résultat dans un registre global, dont l'identifiant est encodé dans l'IR. Initialement, cette allocation des registres globaux est construite à partir des

registres du binaire d'origine. Ce choix d'implémentation facilite les premières étapes de la traduction et fonctionne bien car le RISC-V utilise suffisamment de registres.

Le flot de traduction et d'optimisation offre également la possibilité de ré-allouer les registres temporaires lors de l'ordonnancement des instructions. L'intérêt de cette ré-allocation est de s'affranchir des dépendances de nom. Dans l'IR, les instructions utilisant un registre temporaire sont identifiées par un bit nommé `alloc`.

Lors de l'ordonnancement d'une instruction i utilisant un registre temporaire, deux situations sont possibles. Si des registres physiques sont disponibles, l'outil en alloue un à l'instruction et garde trace de cette allocation. Le nombre de registres physiques disponibles est diminué de 1. Si aucun registre physique n'est disponible, l'instruction utilise le registre global encodé dans l'IR. Dans la suite de l'ordonnancement, si une autre instruction référence le résultat de i, l'outil d'ordonnancement est capable de retrouver le registre physique effectivement utilisé.

Puisque l'allocation des registres peut être modifiée pendant l'ordonnancement des instructions, les dépendances de nom ne sont pas encodées dans l'IR. Ces dépendances doivent être calculées pendant l'ordonnancement en fonction des registres effectivement utilisés.

Cette représentation a plusieurs avantages :

— La séparation entre registres temporaires et locaux permet une optimisation incrémentale du binaire généré. Dans un premier temps, chaque instruction utilise les registres globaux déduits du binaire d'origine. L'ordonnancement généré est fortement contraint par les dépendances de nom. L'outil peut ensuite réaliser une analyse locale permettant d'identifier les registres temporaires, et les identifier grâce au bit `alloc`. Ces registres sont ensuite ré-alloués lors de l'ordonnancement. Enfin, l'outil peut également ré-allouer tous les registres d'une procédure, au prix d'une analyse complexe des durées de vie.

— Les registres globaux ne sont pas modifiés pendant l'ordonnancement. Il est donc possible d'optimiser et d'ordonnancer un unique bloc de l'IR. Cette fonctionnalité est utilisée lors de certaines phases d'optimisation continue.

La figure 3.3 représente deux interprétations possibles d'un bloc, selon la valeur des bits `alloc` des instructions 2 et 3. Le graphe de flot de données du bas correspond au cas où les registres ne sont pas ré-alloués, entrainant ainsi des dépendances WAW et des dépendances WAR. Ces dépendances forcent les instructions 5 et 6 à être ordonnancées après les instructions 2, 3 et 4. Le chemin critique dans ce graphe est de 5. Au contraire, dans le graphe du haut, ces registres temporaires sont ré-alloués et l'outil s'affranchit de ces dépendances de nom. Le chemin critique est alors de 3.

Encodage des instructions

Maintenant que les principales idées de l'IR ont été présentées, nous détaillons l'encodage précis de chaque instruction. La figure 3.4 schématise l'encodage d'une instruction de la représentation intermédiaire.

La représentation contient les éléments suivants :

— Le champ `nbRead` contient le nombre d'utilisations du résultat de l'instruction courante dans le bloc. Cette valeur permet de déterminer quand un registre peut-être libéré, c'est-à-dire quand la

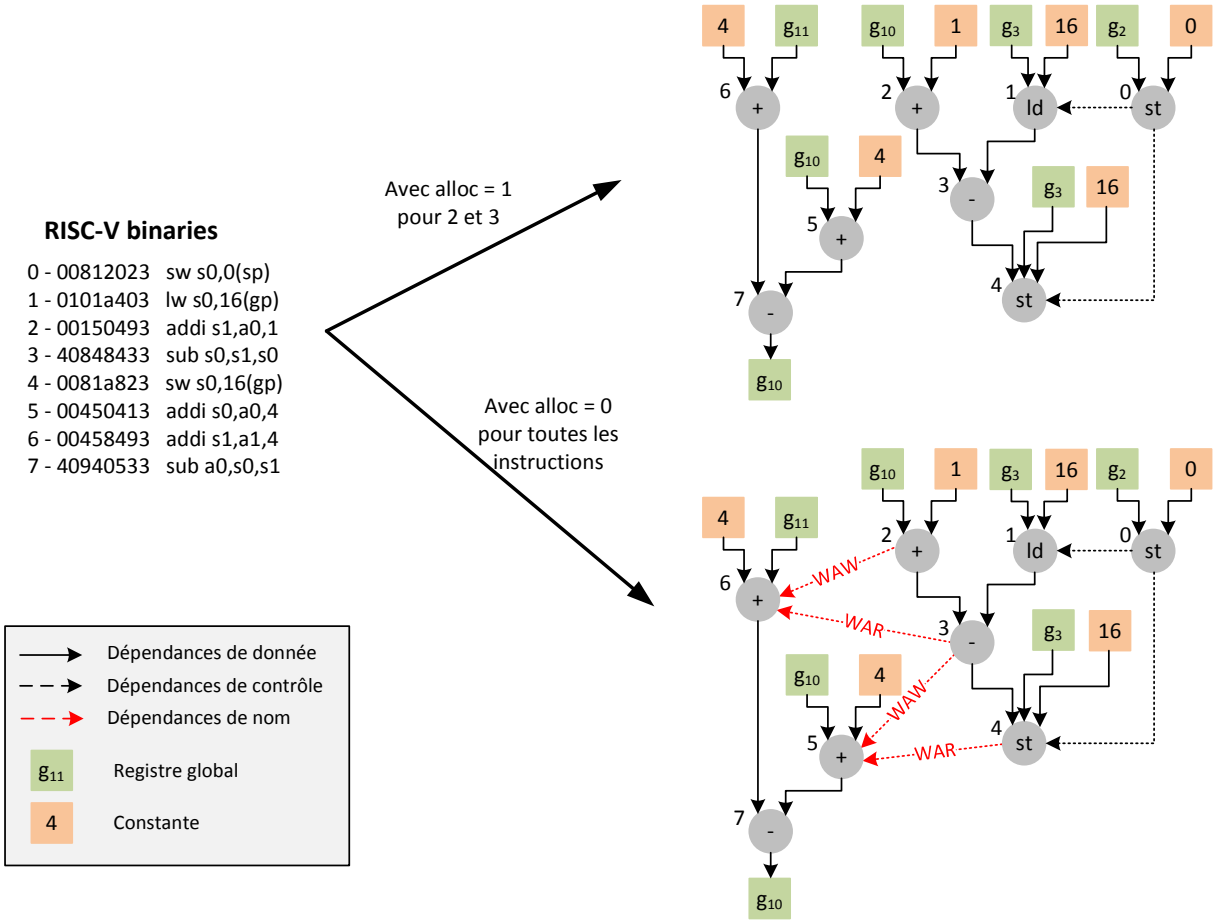


FIGURE 3.3 – Illustration du mécanisme de renommage des registres. Le premier graphe représente la version où le bit `alloc` des instructions 2 et 3 est à un. Par conséquant, il n'y a pas de dépendances de nom. A l'opposé, le deuxième graphe représente la version où le bit `alloc` est à zero. Plusieurs dépendances de nom contraignent l'ordonnancement.

valeur stockée n'est plus utilisée.

— Le champ `predNames` est un tableau contenant jusqu'à 5 prédécesseurs de l'instruction courante. Lors de la phase d'ordonnancement des instructions, il faudra s'assurer que chacun de ces prédécesseurs soit ordonnancé avant l'instruction courante. Ce tableau peut contenir deux types de prédécesseurs : les prédécesseurs de donnée qui correspondent à l'attente d'un résultat et les prédécesseurs de contrôle capturent les autres contraintes. Le nombre de 5 instructions a été choisi de manière à pouvoir encoder plusieurs dépendances tout en gardant la taille de l'IR raisonnable.

— Les champs `nbPred` et `nbDPred` contiennent respectivement le nombre total de prédécesseurs et le nombre total de prédécesseurs de donnée de l'instruction courante. Ces champs sont utilisés pour savoir comment parcourir le tableau `predNames` et comment déterminer le type de chaque dépendance (celles-ci sont triées dans le tableau).

— Le champ `dest` contient l'identifiant du registre dans lequel le résultat de l'instruction sera écrit et

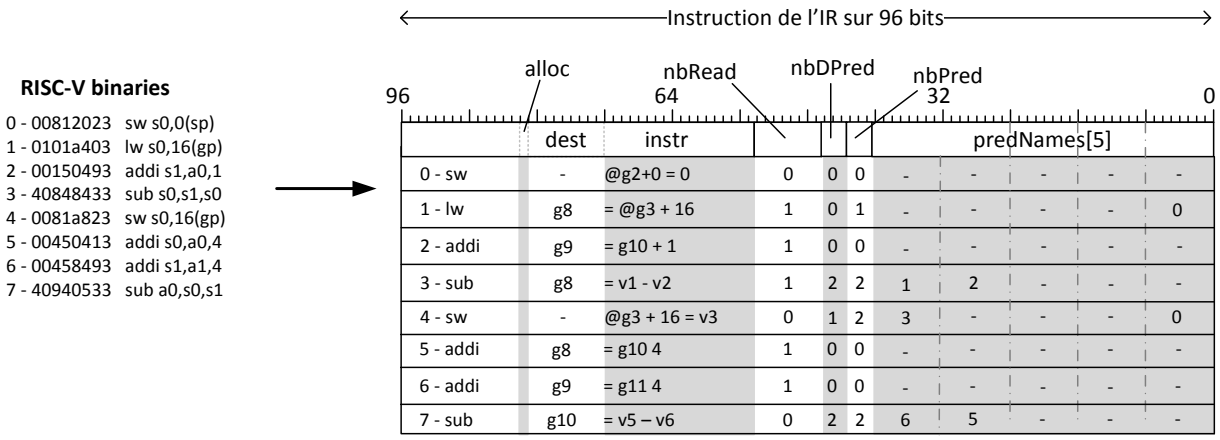


FIGURE 3.4 – Présentation de l'encodage des instructions dans la représentation intermédiaire de Hybrid-DBT.

le champ `alloc` contient un bit permettant de savoir si le registre peut être renommé ou non.

— Le champ `instr` contient toutes les informations sur l'instruction (code de l'opération, constantes utilisées etc.). Il est utilisé pour générer l'instruction binaire une fois qu'elle a été ordonnancée.

Cet encodage permet de représenter le graphe de flot de données d'un bloc. Dans la prochaine sous-section, nous présentons rapidement la représentation des procédures et du graphe de flot de contrôle dans l'IR.

Représentation du graphe de flot de contrôle

Dans les paragraphes précédents, nous nous sommes principalement intéressés à l'encodage du graphe de flot de données à l'intérieur d'un bloc. La représentation intermédiaire utilisée permet également de représenter le graphe de flot de contrôle (CFG pour *Control Flow Graph*) d'une fonction. Toutefois, cette représentation ne sera utilisée qu'aux niveaux d'optimisation les plus élevés et ne sera pas manipulée directement par des accélérateurs. Pour ces raisons, ce graphe est représenté grâce à des listes chaînées de noeuds et de successeurs.

Pour chaque bloc, l'outil a un accès immédiat à la liste de ses successeurs dans le CFG de la procédure. Il a également accès la liste des instructions de branchement, chacune étant associée à un bloc de destination. Cette représentation est utilisée pour construire des traces, identifier et/ou dérouler des boucles ou fusionner spéculativement des blocs de code.

Dans cette section, nous avons présenté les principaux choix de conception de Hybrid-DBT. Nous avons justfié le choix des différents accélérateurs matériels et motivé l'utilisation d'une représentation intermédiaire spécialisée. La prochaine section présente le flot de traduction et d'optimisation de Hybrid-DBT et précise quand et comment les différents accélérateurs sont utilisés.

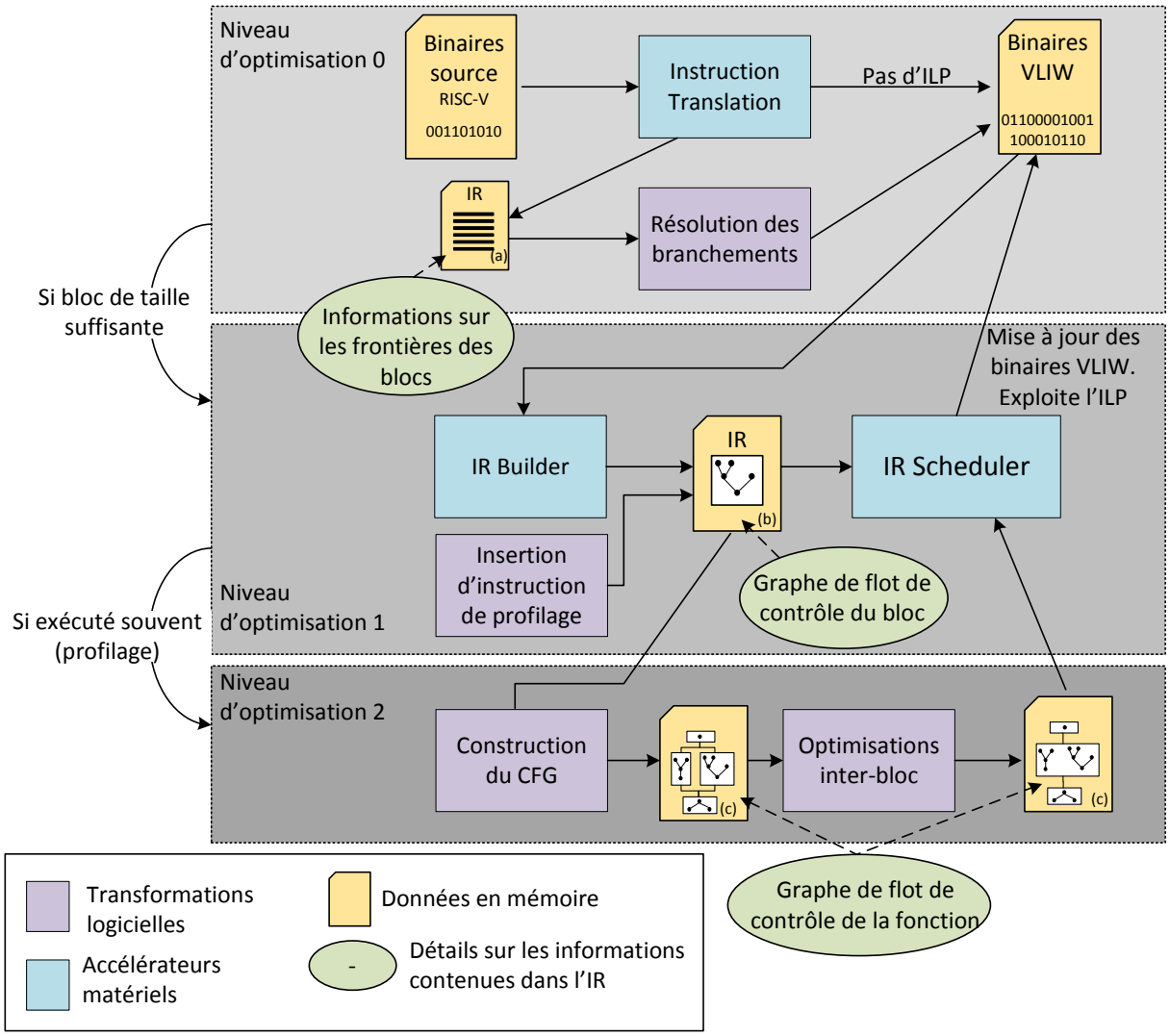


FIGURE 3.5 – Représentation du flot de traduction et d'optimisation d'un binaire RISC-V vers un binaire VLIW. Les différents niveaux d'optimisation décrits dans cette section sont représentés sur cette figure.

3.2 Flot de traduction et d'optimisation

Dans cette section, nous décrivons le flot de traduction et d'optimisation de Hybrid-DBT. Comme pour de nombreux outils de compilation dynamique, nous nous basons sur différents niveaux d'optimisation qui dépendent de la fréquence d'exécution de la région de code. La figure 3.5 illustre ce flot. Le flot est composé de trois niveaux d'optimisation :

— **La traduction des instructions** est le niveau d'optimisation 0 de notre flot. La première fois qu'une région de code est exécutée, l'outil de DBT commence l'exécution aussi vite que possible en cherchant à consommer aussi peu de temps et d'énergie que possible. Ainsi, nous nous contentons de réaliser une traduction naïve de chaque instruction, une par une, en une ou plusieurs instructions VLIW. Ce niveau d'optimisation ne cherche pas à exploiter du parallélisme

d'instructions.

— **La construction des blocs et l'ordonnancement des instructions** constituent le niveau d'optimisation 1 du flot. Quand un bloc a plus de 8 instructions ou s'il possède un branchement arrière, il est considéré comme un candidat éligible pour le premier niveau d'optimisation : la représentation intermédiaire est construite et est utilisée pour ordonnancer les instructions sur les voies d'exécution du VLIW. L'utilisation de deux accélérateurs matériels permet de réduire le coût de ce niveau d'optimisation et donc de le déclencher de manière très agressive, sans avoir besoin de profiler l'exécution. Lors de ce niveau d'optimisation, l'outil insère également des instructions de profilage dans les blocs, permettant de déclencher le niveau d'optimisation suivant.

— **La construction des fonctions et les optimisations inter-blocs** constituent le niveau d'optimisation 2 de notre flot. Quand un bloc a été exécuté plus de 20 fois, ce niveau d'optimisation est déclenché. L'outil de DBT construit le graphe de flot de contrôle de la région environnant le bloc. Ces informations seront alors utilisées pour appliquer plusieurs optimisations visant à augmenter les performances.

Dans la suite de cette section, nous allons présenter en détail les différents niveaux d'optimisation et comment ils utilisent les accélérateurs mis à disposition.

3.2.1 Traduction des instructions

Quand une portion de code est exécutée pour la première fois, l'exécution doit commencer le plus vite possible. De plus, l'outil ne peut prévoir le nombre d'exécutions du code à traduire. Pour ces deux raisons, nous avons choisi de commencer par une première traduction naïve des binaires qui n'essaie pas d'exploiter le parallélisme d'instructions. Ainsi chaque instruction RISC-V sera lue et traduite indépendamment. S'il y a une dépendance de donnée entre deux instructions, des NOP sont insérés pour s'assurer de l'exécution correcte.

Essayer d'exploiter du parallélisme d'instructions pendant cette première traduction peut poser plusieurs problèmes. En effet, la traduction aurait pu être réalisée de manière à exécuter en parallèle certaines opérations. Un mécanisme similaire à ceux utilisés dans les processeurs superscalaires à exécution dans l'ordre aurait rempli cette fonction. Toutefois, nous désirons ici traduire une fois pour toutes les instructions RISC-V. Si, lors de cette traduction, la destination d'un branchement arrière est située entre deux instructions qui ont été regroupées en un seul cycle, les binaires générés sont incorrects. Pour cette raison, nous avons choisi de générer des binaires moins efficaces mais toujours corrects lors de cette première traduction.

Pour réduire encore plus le coût de cette première traduction, nous avons mis au point un accélérateur (appelé `First-Pass Translator`) capable de traduire des portions de binaire avec une cadence importante. Plus de détails sur cet accélérateur sont fournis dans la sous section 3.3.1.

Pendant cette traduction, l'accélérateur extrait deux informations :

— l'emplacement et la destination de tous les branchements rencontrés pendant la traduction (les adresses conservées sont celles des binaires d'origine) ;

— le nombre d'instructions qui sont insérées pendant cette traduction (parce qu'une instruction RISC-V a besoin de plusieurs instructions VLIW ou parce qu'un NOP a été inséré à cause d'une

dépendance). Cette information est utilisée pour traduire une adresse dans les binaires RISC-V en son équivalent dans les binaires VLIW.

En utilisant ces deux informations, l'outil de DBT est capable de calculer les frontières des différents blocs et d'encoder ces informations dans une première version de la représentation intermédiaire (appelée `IR (a)` sur la figure 3.5). Cette représentation intermédiaire ne contient que les informations sur le début et la fin de chaque bloc identifié dans les binaires.

Le flot de DBT devra également gérer les sauts directs (où la destination est connue statiquement) et indirects (c'est-à-dire les sauts vers une adresse stockée dans un registre). Pendant la première traduction, tous les sauts sont remplacés par des sauts vers une procédure permettant de recalculer l'adresse correspondante en utilisant la destination initiale (c'est-à-dire l'adresse RISC-V) et les informations sur les insertions d'instructions. En parallèle de l'exécution, l'outil de DBT examinera les différents sauts directs pour calculer la bonne destination. Les binaires générés seront corrigés en conséquence. Pour les instructions indirectes, l'adresse ne peut être calculée que pendant l'exécution. On passera donc toujours par la procédure qui calcule les adresses.

La gestion des branchements indirects dans les outils de DBT a été étudiée par Hiser et al. [46]. Comme ils le proposent, nous utilisons une table de hachage qui stocke les dernières adresses RISC-V et leur équivalent VLIW. Lors d'un branchement, les derniers bits de l'adresse de destination sont utilisés comme hachage et cette destination est comparée à la destination sauvegardée. Si elles sont identiques, on peut résoudre le saut sans calculer l'adresse. Sinon, la destination effective est calculée et sauvegardée dans la table. D'autres options sont proposées par Hiser et al. pour éviter de polluer les caches d'instructions et de données : au lieu de se déplacer dans la section qui gère les branchements, la résolution avec la table de hachage peut se faire dans le bloc de base contenant le saut ; pour économiser la table de hachage, elle peut être remplacée par une vérification de la destination la plus courante. Ces deux options ne sont pas encore implémentées et constituent des améliorations possibles de notre outil.

3.2.2 Construction des blocs et ordonnancement des instructions

Lors de la traduction des instructions, l'outil a identifié les différents blocs de base de l'application. À cette étape du flot de traduction et d'optimisation, la représentation intermédiaire construite ne contient que le début et la taille de chaque bloc détecté et les binaires VLIW en cours d'exécution n'exploitent pas d'ILP. À ce niveau d'optimisation, l'outil de DBT sélectionne des blocs de taille suffisante, génère l'IR et s'en sert pour ordonnancer les instructions sur les différentes voies d'exécution du VLIW.

Pour générer la représentation intermédiaire décrite dans la sous-section 3.1.3, nous utilisons un accélérateur matériel nommé `IR Builder`. Cet accélérateur lit et analyse les binaires VLIW pour construire le graphe de flot de données et l'encoder dans l'IR. Sur la figure 3.5, cette IR est représentée en tant que IR (b). Plus de précisions sur l'accélérateur sont disponibles dans la sous-section 3.3.2.

Une fois la représentation intermédiaire construite, l'outil de DBT utilise un autre accélérateur matériel, appelé `IR Scheduler`. Celui-ci est chargé de réaliser l'ordonnancement des instructions et l'allocation des registres temporaires de l'IR. Cette étape d'ordonnancement doit respecter les différentes hypothèses posées lors du développement de l'IR : il faut s'assurer que les dépendances de donnée

et de contrôle représentées dans l'IR soient respectées. Il faut également respecter les dépendances de nom qui ne sont pas encodées. Cet accélérateur supporte également l'utilisation du bit `alloc` qui permet la ré-allocation d'un registre temporaire. La sous-section 3.3.3 apporte plus de détails sur cet accélérateur.

Enfin, les binaires générés sont insérés à la place de leurs précédentes versions (celles générées lors de la première traduction). Si le bloc optimisé contient un branchement arrière (ce qui correspond généralement à une boucle), l'outil de DBT insère également une instruction de profilage permettant de déclencher le prochain niveau d'optimisation.

Il faut souligner que ce niveau d'optimisation n'est pas déclenché par du profilage. Il suffit en effet que le bloc contienne suffisament d'instructions pour qu'il soit optimisé. Ce niveau d'agressivité dans le flot d'optimisation est rendu possible par l'utilisation des deux accélérateurs.

3.2.3 Optimisation à l'échelle des fonctions

Quand un bloc est exécuté plus de 20 fois, l'outil de DBT déclenche le niveau d'optimisation 2 qui correspond aux optimisations inter-blocs. L'outil analyse la région de code avoisinant le bloc concerné pour construire le graphe de flot de contrôle de la procédure qui le contient. Plus précisement, l'analyse commence sur le bloc qui était profilé et ajoute, par itérations successives les successeurs et les prédécesseurs de chaque bloc présent dans la procédure en cours de construction. Sur la figure 3.5, cette IR est représentée en tant que IR (c). L'outil insère également des instructions de profilage dans les différents blocs pour déterminer le poids des différents chemins dans la fonction. Ce poids correspond au taux d'utilisation d'un bloc.

Des analyses sur le code permettent de déclencher différentes transformations logicielles qui ont pour but d'exposer plus de parallélisme d'instructions. Ces différentes transformations ne seront pas appliquées automatiquement sur tous les binaires. En effet, des analyses sur le graphe de flot de contrôle et le résultat du profilage vont permettre de choisir où les appliquer. Etant appliquées moins souvent, ces optimisations sont réalisées de manière logicielle et non matérielle.

Dans la suite de cette sous-section, nous présentons les différentes transformations qui sont utilisées pour optimiser les binaires générés.

Création de super-blocs

Un super-bloc est un bloc ne disposant que d'un seul point d'entrée mais de plusieurs points de sortie. Il correspond à un chemin dans le graphe de flot de contrôle. Travailler sur des super-blocs plutôt que sur des blocs de base permet de commencer l'exécution des instructions du bloc suivant avant de savoir s'il faut effectivement les exécuter ou non. En ce sens, cela est très similaire aux mécanismes de spéculation des OoO.

Pour construire un super-bloc, nous utilisons une transformation pour fusionner spéculativement deux blocs :

— Toutes les instructions du premier bloc sont parcourues pour garder une trace de la dernière instruction ayant modifié chaque registre global, des derniers accès mémoire réalisés et de la dernière instruction de branchement rencontrée.

— Le deuxième bloc est ensuite analysé et ses instructions sont ajoutées dans la représentation intermédiaire du premier bloc. Les instructions qui utilisent une valeur globale définie dans le premier bloc référencent maintenant cette valeur comme une valeur locale. Des dépendances sont ajoutées pour s'assurer que les accès mémoire soient toujours effectués dans l'ordre et que les écritures mémoire du deuxième bloc soient bien ordonnancées après le point de sortie éventuel. Des registres sont alloués aux variables temporaires pour permettre l'exécution spéculative. Des dépendances de contrôle sont insérées entre les instructions de branchement et les instructions qui génèrent des variables globales pour que ces dernières soient ordonnancées après le point de sortie.
— Le branchement situé à la fin du premier bloc est modifié pour permettre de sortir dans le cas où le second bloc ne doit pas être exécuté.

Une transformation similaire est utilisée pour dérouler les boucles.

Allocation de registres

Nous exploitons deux niveaux d'allocation de registres dans notre flot d'optimisation : une ré-allocation naïve (renommage) est réalisée lors de l'ordonnancement des instructions et une allocation plus globale permet d'obtenir un registre de destination pour toutes les instructions qui produisent une valeur. Nous avons donc deux analyses de durée de vie dans notre flot :

— La première analyse ne porte que sur un bloc de l'IR et consiste à trouver la liste des registres globaux qui ne sont lus et modifiés que dans le bloc courant (c'est-à-dire la liste des valeurs locales). Ces instructions pourront donc être marquées en utilisant le bit `alloc` de la représentation intermédiaire.
— La deuxième analyse, plus complexe, analyse les binaires à l'échelle des procédures, en déterminant les registres modifiés et lus dans chaque bloc. Les différents chemins du graphe de flot de controle sont ensuite examinés pour définir la durée de vie de chaque registre global. Une fois ces durées de vie connues, il est possible d'identifier un plus grand nombre de registres temporaires (par exemple des registres qui sont écrits dans un bloc, non ré-écrits à la fin du bloc mais pourtant non lus dans chaque chemin quittant le bloc), ou de ré-allouer des registres globaux.

Dans cette section, nous avons présenté le flot de traduction et d'optimisation utilisé dans Hybrid-DBT. Comme nous l'avons vu, ce flot est basé sur trois accélérateurs matériels qui permettent de réduire le coût des premiers niveaux d'optimisation. Dans la prochaine section, nous présentons en détail le fonctionnement de ces différents accélérateurs.

3.3 Description des accélérateurs matériels utilisés

Dans cette section, nous décrivons la structure et le fonctionnement des différents accélérateurs matériels qui ont été développés pour notre flot de traduction. Chacun de ces composants a été développé en utilisant un outil de synthèse de haut-niveau. Nous avons développé une description fonctionnelle des algorithmes en C puis utilisé Mentor Catapult C pour générer la description RTL correspondante.

Pour augmenter les performances des accélérateurs générés, plusieurs optimisations ont été faites sur le code :

- pipeline de boucle pour augmenter le débit des accélérateurs et cacher la latence de traitement d'une instruction ;
- modification des algorithmes pour utiliser des structures de données régulières (pas de structures basées sur des pointeurs) ;
- partitionnement des mémoires pour offrir plus d'accès en parallèle ;
- ajout de mécanismes de *forwarding* explicites afin de retirer des dépendances mémoire entre itérations.

Le reste de cette section présente les trois différents accélérateurs utilisés dans Hybrid-DBT : le `first-pass translator`, le `IR builder` et le `IR Scheduler`.

3.3.1 Description du First-pass Translator

L'objectif du `first-pass translator` est de traduire des blocs d'instructions le plus rapidement possible tout en consommant le moins d'énergie possible. Cet accélérateur n'a pas pour objectif de générer un code fortement optimisé. L'accélérateur que nous avons conçu lit chaque instruction RISC-V indépendamment, la décode (et extrait ainsi les champs `opcode`, `funct3`, `funct7`, `rs1`, `rs2` et `rd`) puis utilise ces informations pour reconstruire une ou plusieurs instructions VLIW. Si la plupart des instructions peuvent être traduites quasi directement, certaines instructions RISC-V nécessitent plus d'une instruction VLIW pour respecter la sémantique. Nous avons restreint le jeu d'instructions du VLIW pour permettre un décodage plus simple. Parmi les différences entre les deux jeux d'instructions, les instructions du processeur VLIW utilisent des champs plus courts pour encoder les valeurs immédiates.

En ce qui concerne l'allocation des registres, le VLIW a accès à 64 registres physiques alors que la spécification du RISC-V ne définit que 32 registres. Pour des raisons de simplicité, nous faisons le choix d'établir une correspondance directe entre les registres RISC-V et les 32 premiers registres du VLIW lors de cette première traduction.

Afin de maximiser la cadence de traduction des instructions, notre accélérateur est pipeliné. Ainsi une nouvelle instruction VLIW est générée à chaque cycle. Le jeu d'instructions du RISC-V et celui du VLIW étant très proches, la conception et la mise en oeuvre de cet accélérateur est relativement simple. S'il avait fallu concevoir ce même type d'accélérateur pour des jeux d'instructions plus complexes, le coût en surface de ce composant pourrait s'en ressentir. Cependant, de par la nature des formats considérés, la complexité (au sens algorithmique) du processus de traduction reste en O(1) et il devrait toujours être possible de produire une instruction VLIW par cycle.

Enfin, il faut noter que l'accélérateur utilise des mémoires internes pour garder trace des insertions réalisées ainsi que de l'emplacement et de la destination des instructions de branchement. Ces informations peuvent être générées en parallèle de la traduction. Comme nous l'avons expliqué auparavant, elles sont utilisées par l'outil de DBT pour calculer la destination des branchements directs et indirects.

3.3.2 Description du IR Builder

L'objectif du deuxième accélérateur matériel, que nous avons nommé `IR Builder`, est encore d'analyser les binaires VLIW mais cette fois-ci dans le but de construire la représentation intermédiaire décrite dans la sous-section 3.1.3. L'accélérateur analyse chaque instruction dans l'ordre séquentiel et construit le graphe de flot de données tout en ajoutant des dépendances entre les *load* et les *stores* pour assurer la cohérence mémoire.

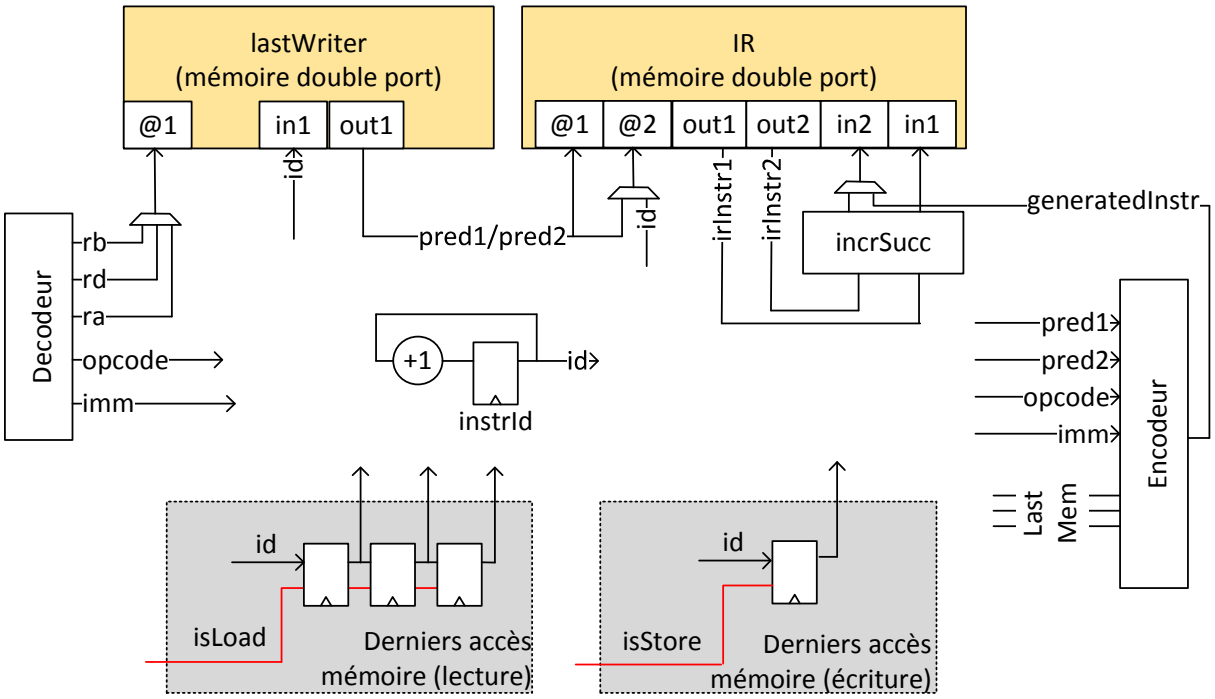


FIGURE 3.6 – Représentation simplifiée de l'organisation interne de l'accélérateur `IR Builder` : on peut y voir les principales mémoires utilisées pour la traduction ainsi que les registres permettant de conserver les derniers accès mémoire analysés. Les blocs `decodeur` et `encodeur` servent à décoder et encoder les instructions.

La figure 3.6 est une représentation simplifiée de l'accélérateur matériel. Les composants suivants y sont représentés :

- Une mémoire appelée `lastWriter` permet de stocker l'identifiant de la dernière instruction ayant écrit dans chaque registre physique. Au début de l'analyse d'un bloc, chaque valeur de la mémoire est initialisée à -1.
- Une mémoire appelée `IR` contient les instructions de la représentation intermédiaire qui sont générées.
- Un registre contient l'identifiant du dernier accès en écriture à la mémoire et trois registres contiennent les identifiants des trois dernières lectures de la mémoire. Ces trois registres sont organisés comme des registres à décalage.

Analyser une instruction VLIW et l'ajouter à la représentation intermédiaire du bloc s'effectue cinq étapes, décrites ci-dessous. Ces étapes sont également représentées sur la figure 3.7 qui décrit un

ordonnancement des différents accès mémoire réalisés pendant ces étapes. En effet, les accès à la mémoire sont le goulot d'étranglement des performances de l'accélérateur.

- L'instruction utilise au plus deux valeurs provenant de registres comme opérandes. L'identifiant de la dernière instruction du bloc ayant écrit dans ces registres est obtenue depuis la mémoire `lastWriter`. Quand cette valeur est -1, cela signifie que le registre n'a pas été mis à jour dans le bloc courant, cette valeur est donc considérée comme globale dans l'IR. Dans le cas contraire (c'est-à-dire lorsque la valeur est différente de -1), l'identifiant de l'instruction source est récupéré et encodé dans l'IR. Ces deux accès à la mémoire `lastWriter` sont notés `lire ra` et `lire rb` sur la figure 3.7.
- L'instruction peut produire un résultat qui sera stocké dans un registre. L'accélérateur doit alors stocker l'identifiant de l'instruction en cours d'analyse dans la case mémoire correspondante de `lastWriter`. Ainsi, les futures instructions qui utilisent cette valeur référencent l'instruction en cours d'analyse. Cet accès, qui est nommé `écrire rd` sur la figure 3.7, doit être fait après les accès en lecture de la même itération mais avant ceux de l'itération suivante.
- L'accélérateur doit également gérer les dépendances de donnée. L'instruction courante peut référencer au plus deux autres instructions dont elle lit le résultat. Or, dans la représentation intermédiaire, le nombre d'utilisations du résultat d'une instruction est explicitement encodé (c'est le champ `nbReads`). L'accélérateur lit donc les instructions IR de ses deux prédécesseurs en faisant des accès à `IR`. Il incrémente ensuite les champs `nbReads` et réécrit l'instruction IR en mémoire. Sur la figure 3.7, ces quatre accès sont appelés `lire p1`, `lire p2`, `écrire p1` et `écrire p2`.
- Enfin, lorsque l'instruction réalise un accès mémoire, l'accélérateur doit insérer des dépendances de contrôle pour assurer la cohérence mémoire.
  - Si l'instruction est une lecture mémoire, une dépendance est ajoutée entre la dernière instruction qui modifie le contenu de la mémoire et l'instruction courante. L'instruction courante est également ajoutée à la liste des trois derniers accès en lecture. S'il y a déjà trois accès sauvegardés dans cette liste, on utilise un mécanisme de chaînage : une dépendance est ajoutée entre le plus ancien accès de la liste et l'instruction courante. Cet ancien accès est ensuite remplacé par la nouvelle instruction.
  - Si l'instruction est une écriture en mémoire, des dépendances provenant des trois derniers accès en lecture sont ajoutées dans l'IR. L'instruction est ensuite sauvegardée comme étant le dernier accès en écriture.
- Enfin, l'instruction sur 96 bits est générée à partir de toutes les informations préalablement rassemblées et est stockée dans la mémoire `IR`. Sur la figure 3.7, cet accès est nommé `écrire instr`.

Comme on peut le voir dans ces différentes étapes, les dépendances de contrôle sont utilisées pour maintenir la cohérence mémoire. Or, à cause du nombre limité de dépendances pouvant être encodées dans la représentation intermédiaire, nous avons choisi de ne garder que les trois dernières lectures mémoire et de chaîner les dépendances quand on en rencontre plus de trois.

La figure 3.7 présente les différents accès à la mémoire réalisés par l'accélérateur ainsi qu'un ordonnancement pipeliné capable de commencer l'analyse d'une nouvelle instruction tous les trois cycles.

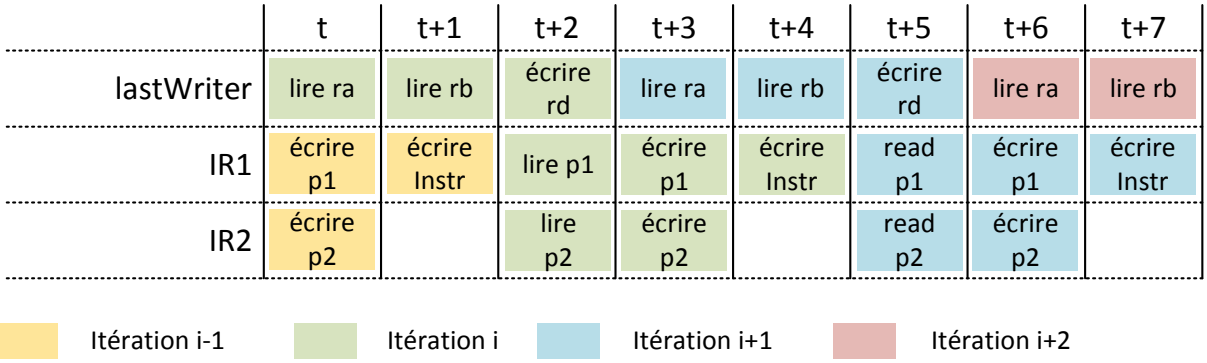


FIGURE 3.7 – Ordonnancement des différents accès mémoire réalisés par l'accélérateur avec un pipeline analysant une nouvelle instruction tous les trois cycles.

3.3.3 Description du IR Scheduler

Le dernier accélérateur conçu dans le cadre de ce travail est le `IR Scheduler`. Celui-ci est chargé de calculer un ordonnancement des instructions et d'allouer des registres pour les valeurs temporaires de l'IR. Deux versions de cet accélérateur ont été développées, basées sur deux heuristiques différentes pour l'ordonnancement des instructions : la première version développée est basée sur le *list-scheduling* et la deuxième sur le *scoreboard scheduling*. Ces deux heuristiques permettent de résoudre le problème différement :

— le *list-scheduling* parcourts chaque cycle de l'ordonnancement et y place une instruction prête ;
— le *scoreboard scheduling* parcourts les instructions du bloc et les place dans une fenêtre d'ordonnancement.

A cause de cette différence, ils ne travaillent pas sur la même représentation intermédiaire. La représentation intermédiaire présentée dans la section 3.1.3 est celle utilisée pour le *scoreboard scheduling* qui est la version de l'accélérateur la plus performante.

Dans le reste de cette sous-section, nous présentons les différences entre les deux représentations intermédiaires et nous tâchons de convaincre que l'étape de génération de l'IR est similaire pour les deux versions. Nous présentons ensuite les deux algorithmes ainsi que les accélérateurs correspondants.

Les deux variantes de la représentation intermédiaire

La représentation intermédiaire utilisée dans le flot de traduction et d'optimisation a été spécialement développée pour accélérer l'ordonnancement des instructions. Son encodage dépend donc fortement de l'algorithme utilisé et des hypothèses retenues. La première version de Hybrid-DBT étant basée sur une heuristique différente, elle utilise également une représentation intermédiaire différente. Toutefois, l'idée principale de l'IR reste la même : le graphe de flot de données du bloc à ordonnancer est encodé directement dans l'IR. La différence réside dans la manière de l'encoder. Les deux versions de la représentation intermédiaire sont :

— **La représentation arrière :** la liste des prédécesseurs de chaque instruction et le nombre d'uti-

lisations du résultat est encodé dans l'IR . Cette version, présentée dans la sous-section 3.1.3, est celle utilisée par défaut dans Hybrid-DBT. Elle est associée à l'algorithme de *scoreboard scheduling*.

— **La représentation avant :** la liste des successeurs de chaque instruction ainsi que le nombre de prédécesseurs est encodé dans l'IR. Cette représentation est utilisée par le *list-scheduling*.

La figure 3.8 présente un graphe de flot de données encodé dans les deux IR. La première représentation correspond à la représentation arrière, la seconde à la représentation avant.

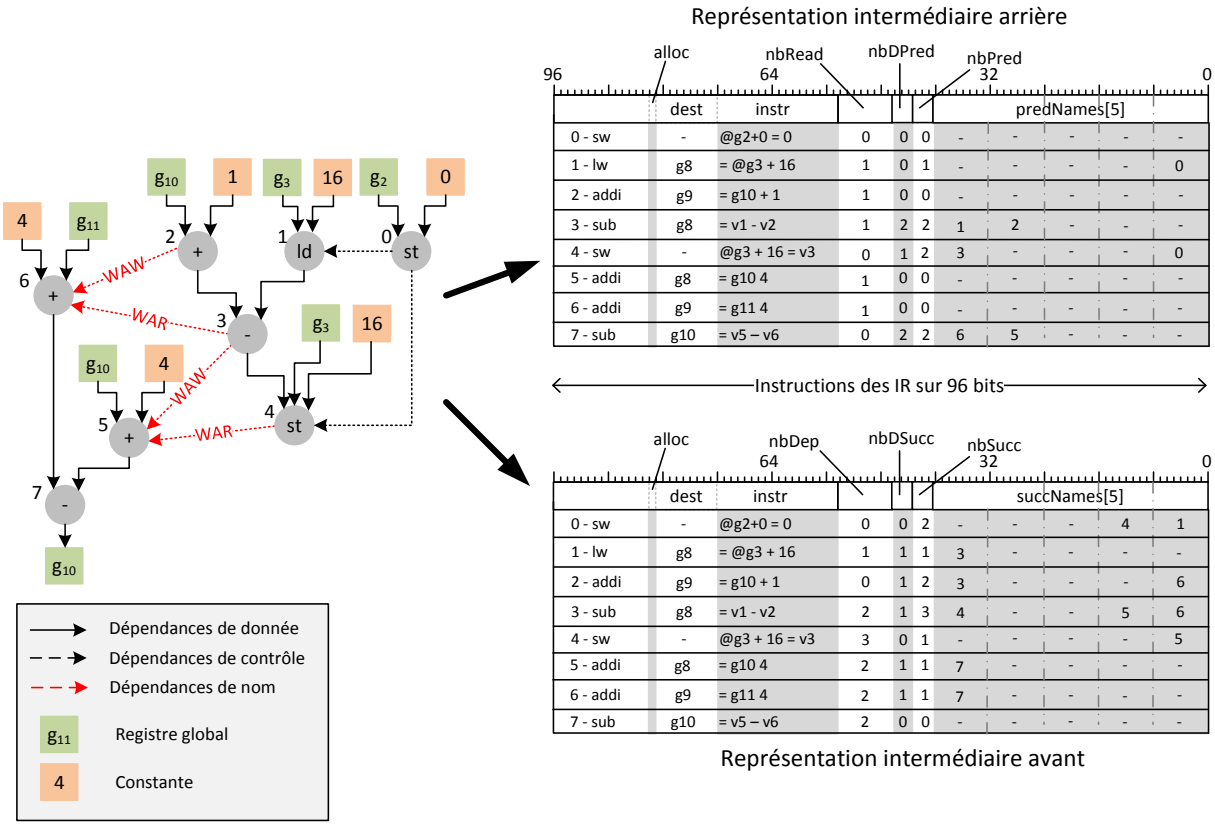


FIGURE 3.8 – Comparaison des deux représentations intermédiaires utilisées dans les deux versions de l'outil. La première version (celle du haut) est la représentation arrière qui a été présentée dans la sous-section 3.1.3 et qui est utilisée pour le *scoreboard scheduling*. La seconde version (en bas) est la représentation avant, utilisée pour le *list-scheduling*.

La représentation avant utilise les champs suivants :

— Le champ `nbDep` contient le nombre total de prédécesseurs de l'instruction dans le graphe de flot de données. Il correspond au champ `nbPred` de la représentation arrière.

— Les champs `nbSucc` et `nbDSucc` contiennent respectivement le nombre total de successeurs et le nombre de successeurs de donnée dans le bloc courant. On peut noter que la valeur de `nbDSucc` correspond au champ `nbRead` de la représentation arrière.

— Les champs `succNames` contiennent la liste de tous les successeurs de l'instruction courante.

Une autre différence importante réside dans la gestion des dépendances de nom : si celles-ci sont

ignorées dans la représentation arrière, elles sont encodées dans la représentation avant. C'est pour cette raison que les valeurs des champs `nbPred` et `nbDep` dans la figure 3.8 sont différentes alors qu'elles représentent la même information. Ce choix est motivé par les algorithmes utilisés pour l'ordonnancement : il est facile de calculer automatiquement les dépendances de nom dans le *scoreboard scheduling* alors que cette tâche est beaucoup plus délicate à réaliser dans le *list-scheduling*.

Les deux représentations sont similaires. Il est donc facile de se convaincre que les quelques différences n'entraînent pas de changements fondamentaux dans les différentes étapes de traduction et d'optimisation. Il y a toutefois des différences subtiles dans la génération de l'IR. En effet, pour ajouter une dépendance de contrôle entre deux instructions $i1$ et $i2$ dans la représentation dite "avant", il faut charger l'instruction de $i1$ et y ajouter $i2$ dans la liste des successeurs. Le nombre de dépendances de $i2$ est incrémenté. Insérer une dépendance de contrôle demande donc à la fois une lecture et une écriture dans la mémoire contenant l'IR. L'ajout des trois dépendances de contrôle assurant la cohérence mémoire est donc coûteuse et le nombre d'accès concurrents à la mémoire stockant l'IR devient trop élevé pour permettre une cadence de pipeline élevée pour l'accélérateur `IR Builder`. De plus, comme les dépendances de nom sont encodées dans l'IR, il faut également les insérer ce qui augmente encore la pression sur les ports de la mémoire `IR`. Par conséquent, notre mise en oeuvre de l'accélérateur permettant de générer la représentation avant ne traite qu'une nouvelle instruction tous les 5 cycles, contre une tous les 3 cycles pour l'accélérateur générant la représentation arrière.

L'algorithme de *list scheduling*

Le premier algorithme qui a été utilisé pour ordonnancer les instructions est le *list-scheduling*. Pour chaque cycle de l'ordonnancement, l'algorithme tente d'assigner une instruction prête à chaque unité d'exécution disponible. Nous avons adapté cette heuristique gloutonne pour exploiter correctement les informations contenues dans la représentation intermédiaire et pour gérer l'allocation de registres.

L'algorithme 1 décrit les différentes étapes réalisées par notre algorithme modifié. La variable `currentCycle` désigne le cycle actuellement considéré par l'algorithme. Il contient les tableaux suivants :

- Le tableau appelé `IR` contient la représentation intermédiaire du bloc à ordonnancer. Pour cet algorithme, la représentation dite "avant" est utilisée.
- Le tableau appelé `reservationTable[][]` est un tableau à deux dimensions contenant l'identifiant des instructions ordonnancées au cycle courant ainsi qu'aux deux cycles précédents, sur les différentes unités d'exécution du VLIW.
- Le tableau appelé `readyList` est un groupe de listes contenant les instructions prêtes à être exécutées pour les différents types d'instructions (arithmétiques simples, mémoires, branchements ou arithmétiques complexes).
- Le tableau `freeRegisters` est une FIFO contenant la liste des registres disponibles pour l'allocation.
- Le tableau `nbRead` contient le nombre de lecture d'un registre dans le bloc courant. Quand ce nombre atteint 0, le registre peut être libéré. Cette valeur est initialisée en utilisant le champ `nbDSucc` contenu dans l'instruction de l'IR qui a produit son résultat dans le registre.
- Le tableau `placeReg` permet de faire le lien entre l'instruction qui produit un résultat et le registre

physique dans lequel ce résultat est stocké.

— Le tableau `nbDeps` contient, pour chaque instruction du bloc, le nombre de prédécesseurs qu'il lui reste avant de pouvoir être ordonnancée. Cette valeur est initialisée grâce au champ `nbDeps` de la représentation intermédiaire.

— Le tableau `binaries` contient les binaires VLIW générés par l'accélérateur.

```
1  while nbInstrScheduled < blockSize do
2      for oneExecUnit in execUnits do
3          instrId = readyList.get(type) ;
4          irInstr = IR[instrId] ;
5          rdest = irInstr.getDest() ; if irInstr.isAlloc then
6              if freeRegisters.isEmpty() then
7                  fails() ;
8              end
9              rdest = freeRegister.get() ;
10         end
11         nbRead[rdest] = irInstr.nbDSucc ;
12         placeReg[instrId] = rdest() ;
13         ra = placeReg[irInstr.dataPred1] ;
14         rb = placeReg[irInstr.dataPred2] ;
15         reservationTable[currentCycle][currentExecUnit] = instrId ;
16         binaries = assembleInstr(ra, rb, rdest) ;
17         nbInstrScheduled++ ;
18     end
19     for oneInstrId in reservationTable[nextCycle] do
20         irInstr = IR[oneInstrId] ;
21         for oneSuccessor in irInstr.succNames do
22             nbDeps[oneSuccessor]– ;
23             if nbDeps[oneSuccessor] == 0 then
24                 type = oneSuccessor.type ;
25                 readyList.insert(oneSuccessor, type) ;
26             end
27         end
28         for oneRegUsed in irInstr.regs do
29             nbRead[oneRegUsed]– ;
30             if nbRead[oneSuccessor] == 0 then
31                 freeRegisters.push(oneSuccessor) ;
32             end
33         end
34     end
35     currentCycle++ ;
36 end
```

Algorithm 1: Description de l'algorithme de *list-scheduling*

Le traitement d'un cycle de l'ordonnancement se fait en deux étapes. La première correspond à la première boucle `for` de l'algorithme 1. Pour chaque unité d'exécution du VLIW, une instruction compatible est prise dans la FIFO des instructions prêtes pour être placées. Si l'instruction a le bit `alloc` à 1, l'algorithme tente de lui allouer un registre. S'il en reste un de libre, il est alloué ; sinon l'ordonnancement

échoue. S'il ne faut pas allouer de registre, le registre de destination encodé dans la représentation intermédiaire est utilisé. Les registres physiques utilisés comme opérandes sont trouvés grâce au tableau `placeReg`. Les différents tableaux `placeReg` et `nbRead` sont mis à jour, puis l'instruction est placée dans la table de réservation et une version encodée est placée dans les binaires VLIW générés.

La seconde étape englobe les deux dernières boucles `for` de l'algorithme et correspond à la validation des instructions ordonnancées. Lorsque les instructions sont retirées de la table de réservation, cela signifie que leur résultat sera prêt au prochain cycle considéré. Ainsi, chaque instruction retirée est analysée : on parcourts tous ses successeurs et on décrémente leur nombre de dépendances. Si le nombre de dépendances d'un successeur de l'instruction atteint 0, il est inséré dans la liste des instructions prêtes. Le même traitement est réalisé pour les registres lus par l'instruction à retirer : leur nombre de lectures restantes est décrémenté. Si un registre temporaire n'a plus de lecture, il peut être inséré dans la liste des registres prêts.

L'accélérateur pour le *list scheduling*

En utilisant la synthèse de haut-niveau, nous avons implémenté l'algorithme de *list-scheduling* décrit dans la partie précédente. La figure 3.9 est une représentation simplifiée du chemin de données de l'accélérateur et des différentes connections. On peut y voir que les différents tableaux utilisés dans l'algorithme 1 sont assignés à des mémoires. Les tableaux les plus fréquement utilisés sont assignés à des mémoires disposant de deux ports lecture/écriture pour augmenter le nombre d'accès en parallèle.

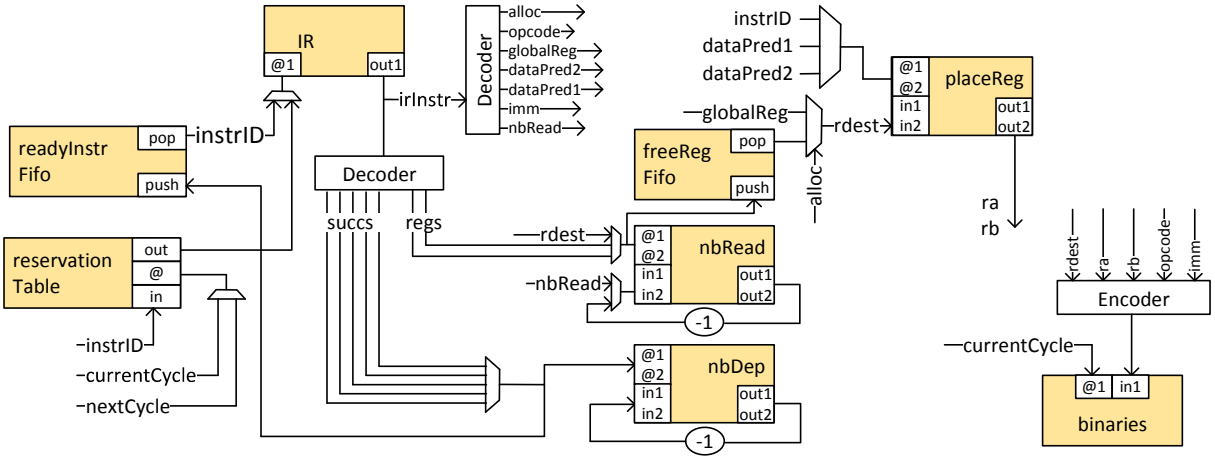


FIGURE 3.9 – Schéma du chemin de données utilisé par l'accélérateur du `list-scheduling`. Les différentes mémoires y sont représentées ainsi que les principales connections.

La figure 3.10 décrit l'ordonnancement des accès mémoire pour les deux boucles lignes 2 et 19 de l'algorithme 1. L'exécution de ces deux boucles est pipelinée : la boucle ligne 2, qui assigne une instruction prête à chaque unité d'exécution, peut commencer une nouvelle itération tous les deux cycles ; la deuxième boucle, qui correspond à la fusion de la boucle ligne 21 traitant les successeurs et de la boucle ligne 28 traitant les registres lus, peut commencer une nouvelle itération à chaque cycle. L'ordonnancement représenté figure 3.10 montre que l'exécution de la première boucle, relativement simple,

est limitée par les accès à la mémoire `placeReg`. La seconde boucle se prête très bien à une exécution pipelinée. Les accès représentés sont ceux réalisés dans le pire cas (lors du traîtement d'une instruction dont les successeurs passent dans l'état prêts et dont le registre de destination se libère). La figure 3.10 représente également de nombreux conflits entre les accès en lecture et en écriture sur les mémoires `nbRead` et `nbDep`. En effet, la première lecture de l'itération i peut se retrouver en conflit avec l'écriture de l'itération $i-1$. Nous avons mis en place un mécanisme de *forwarding* explicite dans le code source : l'adresse des deux accès est comparée et la valeur utilisée est celle de l'itération précédente s'il y a conflit. Une directive spécifique est transmise à l'outil de HLS pour lui indiquer d'ignorer cette dépendance mémoire.

Enfin, même si l'exécution des deux boucles est pipelinée, l'accélérateur passe souvent d'une boucle à l'autre et paye le coût de la latence (4 cycles pour chacune des boucles). Le temps d'ordonnancement d'un bloc est difficile à prédire et dépend de la taille de l'ordonnancement généré puisqu'on parcourt les différents cycles.

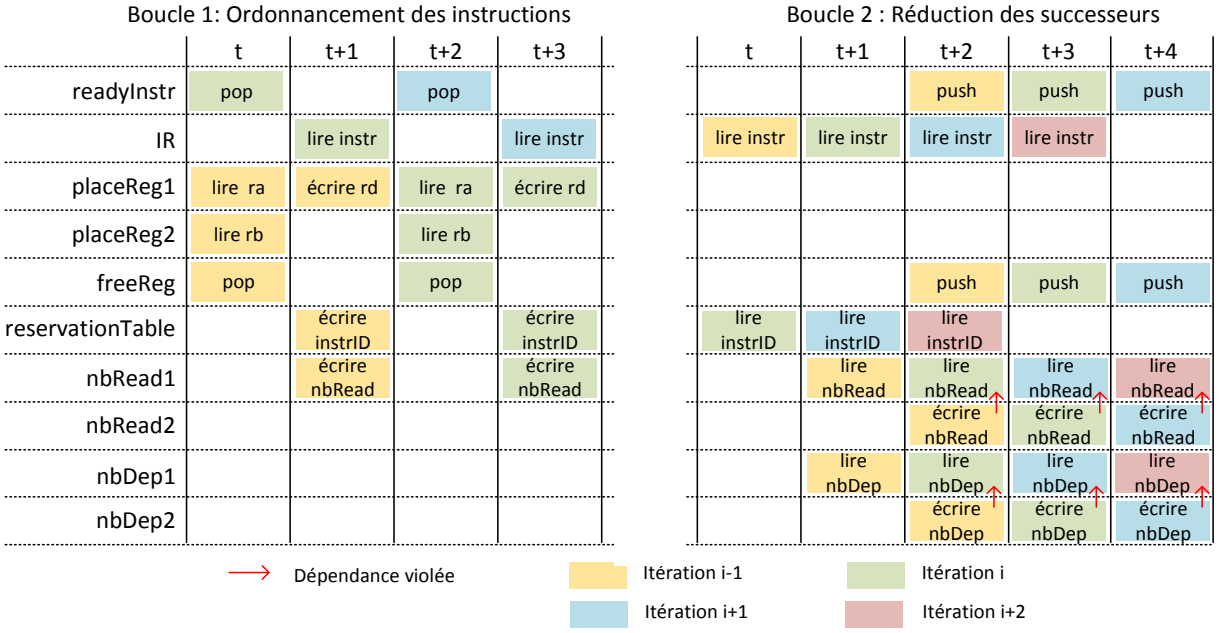


FIGURE 3.10 – Ordonnancement des différents accès mémoire pour l'accélérateur du `list-scheduling`.

L'algorithme de *scoreboard scheduling*

Le deuxième algorithme que nous avons évalué pour l'ordonnancement des instructions est le *scoreboard schedudulin*g. C'est une heuristique gloutonne qui consiste à traiter chaque instruction et tenter de la placer dans une fenêtre de cycles, en respectant ses dépendances. Si l'instruction ne peut être placée dans la fenêtre actuelle, celle-ci est avancée pour libérer une place. Cet algorithme a été modifié pour permettre de prendre en compte les dépendances de nom et l'allocation de registres pour les valeurs temporaires.

L'algorithme 2 décrit les différentes opérations réalisées. Huit tableaux sont utilisés :

— Le tableau appelé `IR` contient la représentation intermédiaire du bloc en cours d'ordonnancement. Cet IR est celle avec la version arrière des dépendances.
— Le tableau `window` est utilisé pour la fenêtre glissante. À chaque fois que l'algorithme place une instruction, tous les emplacements de la fenêtre vont être inspectés pour voir si une unité d'exécution est disponible. Si l'instruction ne peut être placée dans la fenêtre, celle-ci sera décalée de manière à ce que l'instruction puisse être placée au dernier emplacement.
— Le tableau `placeOfInstr` contient le cycle auquel chaque instruction a été ordonnancée. Il est complété au fur et à mesure de l'ordonnancement.
— Les tableaux `lastRead` et `lastWrite` contiennent les identifiants des dernières instructions ayant lu/écrit dans chaque registre physique du VLIW.
— Le tableau `placeReg` permet de faire le lien entre les instructions du bloc et le registre physique dans lequel leur résultat a été stocké.
— Le tableau `nbRead` permet de compter, pour chaque registre physique, le nombre de fois qu'il doit être lu avant de pouvoir être libéré. Cette valeur est initialisée en utilisant le champ `nbRead` contenu dans l'instruction de l'IR qui a écrit son résultat dans le registre.
— Le tableau `binaries` est celui qui contient les binaires VLIW générés.

L'ordonnancement d'une instruction avec l'algorithme 2 peut être divisé en trois étapes. La première correspond à la première boucle *for* qui itère sur chaque prédécesseur de l'instruction et trouve le premier cycle auquel toutes les dépendances de l'instruction sont respectées. Le cycle auquel un prédécesseur a été ordonnancé est obtenu grâce au tableau `placeOfInstr` et un offset y est ajouté en fonction du type de dépendance.

La deuxième étape de l'algorithme permet de gérer le mécanisme d'allocation de registre : si l'instruction à ordonnancer est marquée comme `alloc`, un nouveau registre physique est pris dans une FIFO et lui est alloué. Si le registre ne doit pas être ré-alloué, le registre par défaut est utilisé. Le système adapte ensuite la valeur du premier cycle auquel l'instruction peut être placée en fonction de l'emplacement de la dernière lecture et écriture dans le registre de destination. La valeur de `nbRead` est initialisée grâce à l'IR.

Enfin la troisième étape correspond à la deuxième boucle qui inspecte les différents emplacements de la fenêtre pour trouver le premier emplacement libre étant situé après le point où l'instruction est prête. Si aucun emplacement n'est trouvé, la fenêtre est décalée pour offrir de nouvelles places. Si le problème est qu'aucun emplacement n'est libre, un décalage de 1 suffit. Si le problème est que les dépendances ne pourront pas toutes être résolues dans la fenêtre, la fenêtre est décalée de la différence entre `earliestPlace` et `window.end`. Ainsi, l'instruction devra toujours être placée au dernier cycle de la fenêtre.

Après ces trois étapes, les différents tableaux sont mis à jour : `lastRead`, `lastWrite`, `placeReg` et `nbRead` sont modifiés en fonction des registres utilisés ; l'emplacement dans la fenêtre est marqué comme utilisé et est stocké dans `placeOfInstr`. Enfin, l'instruction est encodée et placée dans les binaires générés à l'adresse adéquate.

```
1  for oneInstruction in IR do
2      earliestPlace = windowStart ;
3      for onePredecessor in oneInstruction.predecessors do
4          if onePredecessor.isData then
5              earliestPlace = min(earliestPlace, placeOfInstr[onePredecessor] + latency) ;
6          else
7              earliestPlace = min(earliestPlace, placeOfInstr[onePredecessor] + 1) ;
8          end
9      end
10     rdest = oneInstruction.getDest() ;
11     if oneInstruction.isAlloc then
12         rdest = freeRegisters.get() ;
13     end
14     earliestPlace = min(earliestPlace, lastRead[rdest], lastWrite[rdest]) ;
15     for oneCycle in window do
16         for oneIssue in oneCycle.issues do
17             if oneIssue.isFree and oneCycle ≥ earliestPlace then
18                 place = oneCycle.address
19             end
20         end
21     end
22     if notFound then
23         window.move(max(1, earliestPlace - window.end)) place = window.end
24     end
25     window[place] = instrId ;
26     accelerators placeOfInstr[instrId] = place ;
27     ra = placeReg[pred1] ;
28     rb = placeReg[pred2] ;
29     placeReg[instrID] = rdest ;
30     lastRead[ra] = place ;
31     lastRead[rb] = place ;
32     lastWrite[rdest] = place ;
33     binaries[place] = assembleInstr(ra, rb, rdest) ;
34 end
```

Algorithm 2: Description de l'algorithme de *scoreboard scheduling*

Organisation de l'accélérateur pour le *scoreboard scheduling*

L'algorithme du `scoreboard scheduling` décrit précédemment a également été implémenté grâce à la synthèse de haut niveau. La figure 3.11 est une représentation simplifiée des différents éléments du chemin de données de l'accélérateur et de leurs intéractions. Des composants mémoire sont utilisés pour la plupart des tableaux de l'algorithme 2, à l'exception de la fenêtre du `scoreboard scheduling` utilise des registres afin d'offrir un accès parallèle à l'ensemble des cases du tableau. Ce choix d'implémentation permet de dérouler entièrement la boucle de la ligne 15 de l'algorithme 2, dont les traitements seront effectués en parallèle.

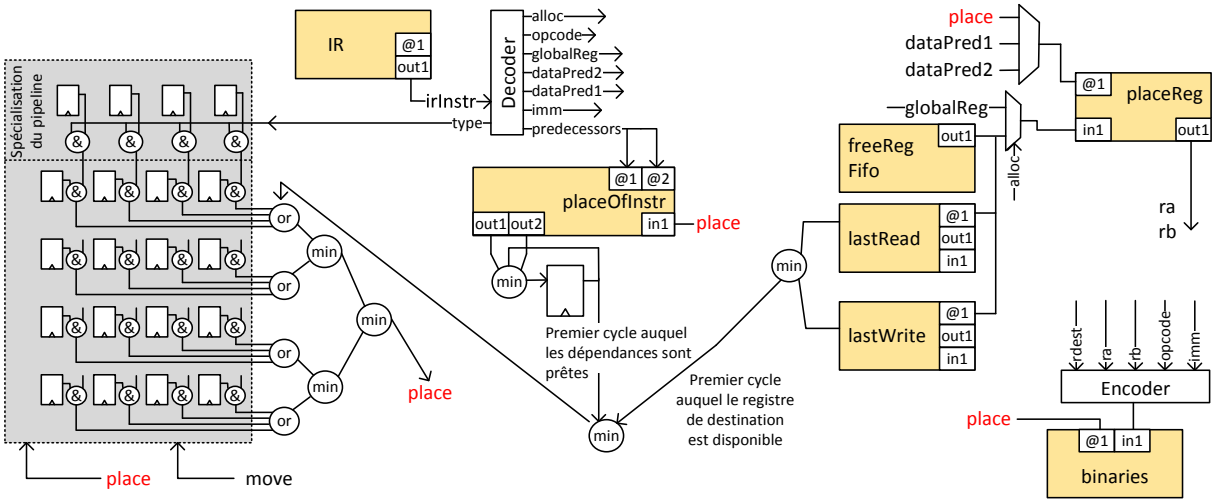


FIGURE 3.11 – Schéma du chemin de données utilisé par l'accélérateur du `scoreboard scheduling`. Les différentes mémoires y sont représentées ainsi que les principales connections.

Les autres étapes de l'algorithme décrivent le processus d'encodage et de décodage des instructions ainsi la mise à jour des mémoires. Toutes ces opérations peuvent être exécutées en parallèle. Cependant, la première boucle demande un accès à la mémoire `placeOfInstr` pour chaque prédécesseur potentiel. Ces accès vont devenir le principal goulot d'étranglement de l'accélérateur.

L'exécution de la boucle principale (ligne 1) qui itère sur toutes les instructions du bloc est pipelinée pour augmenter le débit de l'accélérateur. La figure 3.12 présente un ordonnancement des différents accès mémoire dans le cas ou on ordonnance une nouvelle instruction tous les quatre cycles. Le lecteur peut remarquer que le cycle $t + 4$ de l'ordonnancement est vide, mais il correspond à l'exécution de la boucle de la ligne 15. Comme aucun accès mémoire n'est impliqué dans cette étape, elle n'est pas visible sur cette figure.

Les facteurs limitant la performance de cet accélérateur sont (i) les accès à la mémoire `placeOfInstr`, (ii) les dépendances inter-itérations sur ces valeurs. En effet, l'ordonnancement d'une instruction $i2$ peut dépendre de l'emplacement de l'instruction $i1$ placée à l'itération précédente. Dans ce cas, les différents accès en lecture à `placeOfInstr` doivent avoir eu lieu avant de commencer l'accès en écriture qui marque l'emplacement de l'instruction actuelle. Sur la figure 3.12, au cycle $t + 5$, le premier accès en lecture à `placeOfInstr` ainsi que le premier accès en écriture de l'itération précédente sont réalisés en

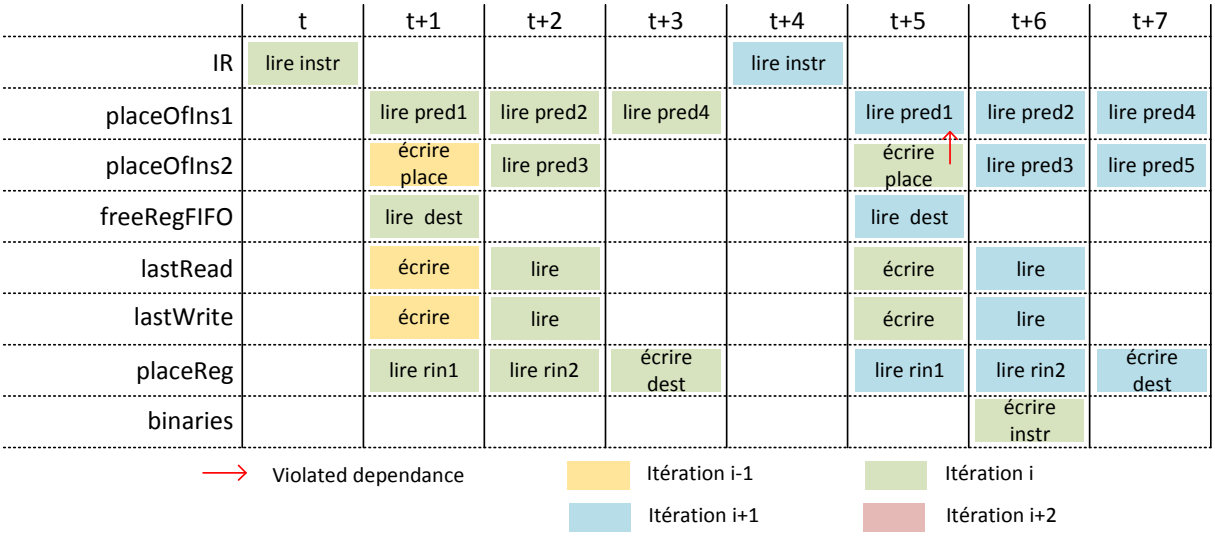


FIGURE 3.12 – Ordonnancement des différents accès mémoire pour l'accélérateur du `scoreboard scheduling`.

parallèle. Pour éviter tout conflit, un système de *forwarding* explicite a été ajouté (comme pour l'accélérateur de *list-scheduling*) : les adresses des deux accès sont comparées et, en cas d'égalité, la valeur de l'écriture est utilisée. Des directives signalent à l'outil de HLS qu'il faut ignorer cette dépendance.

La surface utilisée par l'accélérateur dépend principalement de la taille de la fenêtre. Nous avons utilisé une fenêtre de taille 16 ce qui nous a semblé le meilleur compromis entre la qualité de l'ordonnancement généré et la taille de l'accélérateur. La description utilisée en entrée de l'outil de HLS est générique, et il est donc très facile de modifier cette valeur.

Discussion sur le choix de l'algorithme

Les deux stratégies d'ordonnancement présentées reposent sur des heuristiques très différentes. Une première différence, qui vient de l'approche utilisée par chacune des heuristiques pour appréhender le problème de l'ordonnancement, est que le temps d'exécution du *list-scheduling* dépend de la taille de l'ordonnancement généré alors que celui du *scoreboard scheduling* dépend du nombre d'instructions dans le bloc. La deuxième approche est donc beaucoup plus prédictible puisqu'on connait le nombre d'instructions à l'avance. De plus, les opérations réalisées par le second accélérateur sont plus régulières.

D'autre part, l'utilisation d'une boucle unique dans le *scoreboard scheduling* permet de pipeliner le traitement de chaque instruction, menant à un traitement global plus rapide que pour le *list-scheduling*, qui implique deux boucles internes et nécessite de payer régulièrement le coût de la latence.

Enfin, la principale différence réside dans la gestion de l'allocation de registres. En effet, l'approche basée sur le *scoreboard scheduling* permet de calculer très facilement les dépendances de nom qui sont présentes dans le bloc à ordonnancer. Cela permet donc de ne pas les encoder dans la représentation intermédiaire, ce qui permet de simplifier l'allocation des registres temporaires : s'il n'est pas

possible d'allouer un registre dans l'approche basée sur le *scoreboard scheduling*, il est toujours possible d'utiliser le registre global pour continuer l'ordonnancement. Cette stratégie crée des dépendances de nom mais l'ordonnancement généré restera correct. À l'inverse, pour exploiter correctement les mécanismes de ré-allocation de registres pour les variables temporaires dans le cas du *list-scheduling*, il a fallu éliminer les dépendances de nom disparues en raison de cette ré-allocation. Il n'est alors pas possible de revenir à l'allocation initiale car cela peut mener à des dépendances de nom ignorées. Dans un tel scénario, l'algorithme de *list-scheduling* échoue et il faut utiliser une transformation supplémentaire (mise en oeuvre en logiciel sur le processeur DBT) pour modifier la représentation intermédiaire avant d'essayer de nouveau un ordonnancement.

Au delà des considérations algorithmiques et de l'impact sur la mise en oeuvre des accélérateurs, il faut également évaluer les coûts et les performances obtenus. C'est l'objet de la sous-section suivante.

3.4 Etude expérimentale

Dans cette section, nous présentons les résultats de l'étude expérimentale réalisée. Cette étude a plusieurs objectifs : mesurer l'efficacité des accélérateurs matériels développés pour Hybrid-DBT, étudier le coût en surface et le niveau de performance de notre approche, et enfin démontrer l'intérêt de l'accélération matérielle. Dans cette section, nous discutons à la fois de la performance des accélérateurs matériels, qui impacte le temps de compilation dynamique et de la performance de l'exécution des applications.

Pour chaque expérience, le protocole expérimental est présenté succinctement. L'annexe A de ce document contient une présentation complète de tous les protocoles expérimentaux, ainsi qu'un lien vers un dépôt contenant les sources et les scripts permettant de reproduire ces études expérimentales.

3.4.1 Performance des accélérateurs matériels

Dans cette première étude, nous nous intéressons à la performance des accélérateurs matériels développés. Les algorithmes permettant de réaliser la première traduction, de générer l'IR et d'ordonnancer les instructions ont été implémentés de manière logicielle et compilés vers du RISC-V. Nous avons ensuite utilisé un simulateur de jeu d'instructions RISC-V pour obtenir une estimation du temps d'exécution de chaque étape, que nous avons utilisée pour estimer le coût moyen de traitement d'une instruction. Cette exécution logicielle a été comparée à l'exécution (simulée) des différents accélérateurs.

Les résultats liés au temps d'exécution des accélérateurs sont représentés sur la figure 3.13. La partie gauche de la figure correspond aux facteurs d'accélération obtenus grâce aux différents accélérateurs. La partie droite de la figure représente le temps moyen de traitement d'une instruction par l'accélérateur. Les résultats expérimentaux montrent que le `First-Pass Translator` divise par 250 le temps d'exécution de cette première traduction et traduit une instruction en moins de deux cycles. Cette première traduction consiste principalement à décoder et ré-encoder les instructions, ce qui explique le facteur d'accélération particulièrement élevé. L'accélérateur `IR Builder` permet de construire une

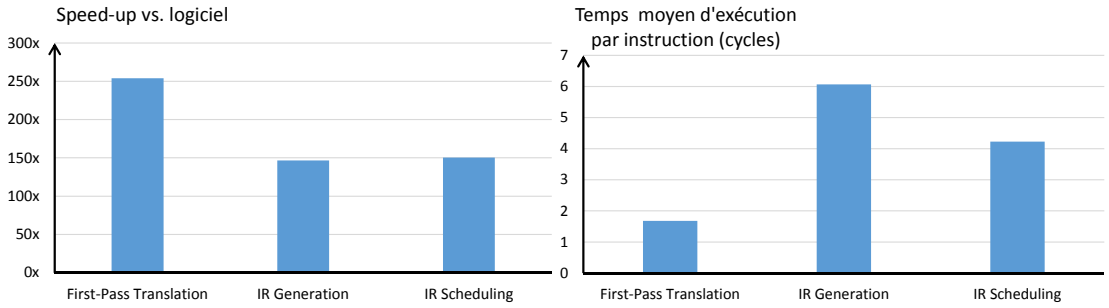


FIGURE 3.13 – Le graphique de gauche représente le facteur d'accélération des différents accélérateurs matériels, en comparaison d'une approche logicielle exécutée sur le processeur DBT. Le graphique de droite représente le nombre moyen de cycles d'exécution requis pour traiter une instruction avec les accélérateurs matériels.

instruction de l'IR en 6 cycles en moyenne et l'accélérateur `IR Scheduler` (la version basée sur le *scoreboard scheduling*) peut en ordonnancer une tous les 4 cycles. Ces deux accélérateurs sont environ 150 fois plus rapides que leur équivalent logiciel, exécuté sur le processeur DBT.

Notons tout de même que les transformations logicielles utilisées n'ont pas été complètement optimisées. En effet, nous avons mis au point une version logicielle des transformations qui n'utilise pas les types spécifiques à la HLS, mais n'avons pas remis en cause la structure globale des transformations. Une version plus travaillée de ces transformation mènerait sans doute à une accélération moindre. Pour fournir un autre élément de comparaison, nous réalisé une estimation du coût de l'ordonnancement dans les travaux de Dinechin [26]. Cette estimation a montrée que l'algorithme d'ordonnancement utilisé par Dinechin (qui s'exécute sur un VLIW à 4 voies) nécessite environs 400 cycles pour ordonnancer une instruction. Notons que l'algorithme utilisé par Dinechin est une version améliorée du *scoreboard scheduling*, ce qui augmente sans doute son côut. Cependant, ce temps d'exécution ne comprend que le temps requis pour l'ordonnancement des instructions, et ne comptes pas le temps nécessaire pour encoder les instructions ou renommer les registres. Notre accélérateur est environs 100 fois plus rapide que l'implémentation logicielle utilisée par Dinechin.

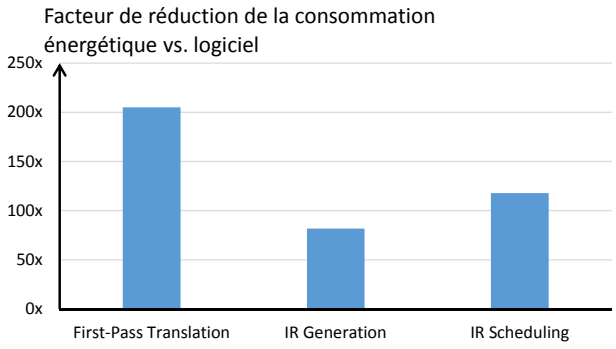


FIGURE 3.14 – Facteur de réduction de la consommation énergétique des différents accélérateurs matériels, en comparaison d'une approche logicielle exécutée sur le processeur DBT.

Un autre aspect important dans le contexte des systèmes embarqués est la consommation énergé-

tique. L'utilisation des différents accélérateurs permet de réduire également l'énergie consommée pour le traitement d'une instruction. Ces facteurs de réduction de la consommation énergétique sont représentés sur la figure 3.14. Là encore, l'utilisation des accélérateurs permet de consommer entre 70 et 200 fois moins d'énergie pour réaliser les traitements. Ces réductions de la consommation énergétique sont principalement liées à la réduction du temps d'exécution discutée précédement.

3.4.2 Performance de l'architecture

La seconde partie de l'étude expérimentale s'intéresse aux performances de Hybrid-DBT sur un ensemble d'applications et à l'importance des différents niveaux d'optimisation. Nous avons exécuté les différentes applications avec la plateforme de DBT, limitée à différents niveaux d'optimisation. Ces exécutions sont labélisées *Hybrid-DBT O0*, *Hybrid-DBT O1* et *Hybrid-DBT O2*. Le niveau d'optimisation O2 contient les optimisations inter-bloc présentées dans la sous-section 3.2.3 ainsi que la spéculation de dépendance mémoire, présentée dans le chapitre 4 de ce document. Nous avons également utilisé un simulateur de processeur RISC-V pipeliné (qui peut s'apparenter aux processeurs *LITTLE* des architectures big.LITTLE), et un modèle GEM5 de processeurs OoO (qui peut s'apparenter à un processeur *big*). Pour chaque application, tous les simulateurs exécutent le même fichier binaire. Ces exécutions sont labélisées respectivement *In-Order* et *OoO*. Pour estimer l'énergie consommée pendant ces différentes exécution, nous avons synthétisé le processeur à exécution dans le désordre BOOM [21] et le processeur *In-Order* Rocket [6] pour la technologie 28 nm de STMicroelectronics. Nous avons ensuite utilisé Design Compiler pour estimer la puissance moyenne dissipée par ces processeurs.

La figure 3.15 représente les niveaux de performance des différentes exécutions, normalisés selon la performance obtenue pour *In-Order*. Une valeur plus élevée signifie un temps d'exécution plus faible. Plusieurs points peuvent être observés sur cette figure :

- Concernant le niveau d'optimisation 0, le niveau de performance est souvent similaire à celui du processeur à exécution dans l'ordre. En effet, la première traduction n'exploite pas de parallélisme d'instructions. Les quelques différences observées (pour `jacobi-2d`, `epic` ou `gsm`) sont provoquées par des différences dans le pipeline des deux processeurs. Par exemple, la pénalité liée à un branchement est plus faible pour le VLIW, ce qui explique les performance de `epic` et `gsm`.
- Concernant le niveau d'optimisation 1, on remarque que l'ordonnancement des instructions impacte grandement la performance de l'architecture. A ce niveau d'optimisation, l'exécution est en moyenne (géométrique) 26% plus rapide que celle basée sur le processeur *In-Order*.
- Le niveau d'optimisation 2 de Hybrid-DBT permet d'être 48% plus rapide que le processeur `In-Order`. Pour les applications `floyd-warshall`, `heat-3d`, `jacobi-1d`, `nussinov` et `seidel` on observe que le niveau d'optimisation 2 apporte nettement moins. Cela est dû aux transformations logicielles implémentées dans ce niveau d'optimisation : l'outil ne peut pas encore créer d'*hyperblock* et, par conséquent, ne peut dérouler que les boucles constituées d'un unique bloc. On constate également que le gain lié au niveau d'optimisation 2 est moins important pour les applications de Mediabench (à partir de `adpcm`) qui cont plus complexes que celles de Polybench. Cela pourrait être corrigé en améliorant la construction de traces réalisée dans le niveau d'opti-

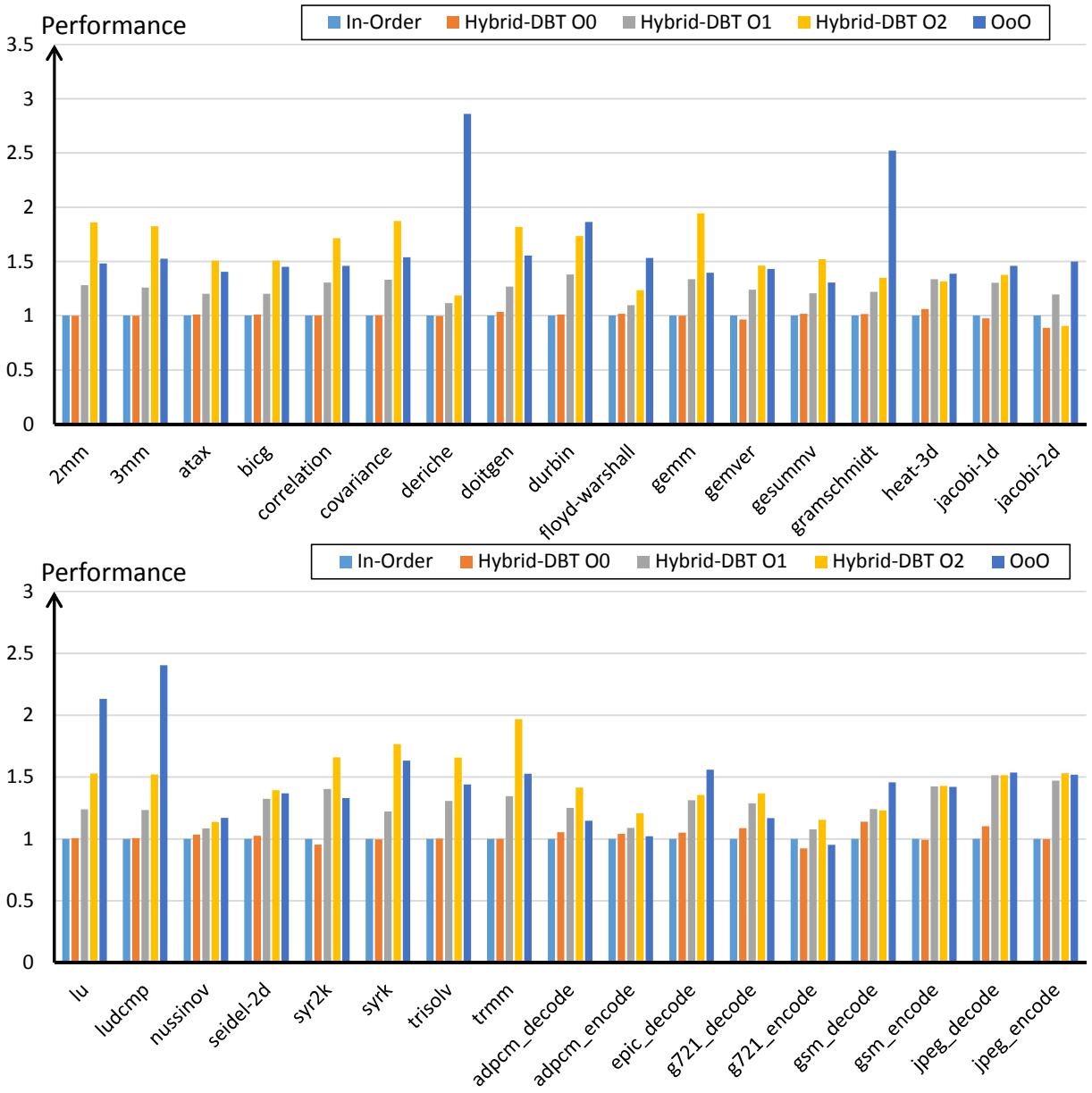


FIGURE 3.15 – Performance lors de l'exécution avec différents niveaux d'optimisation et différentes architectures de processeurs. Les résultats affichés correspondent au temps d'exécution sur le processeur *In-Order* divisé par le temps d'exécution dans le scénario concerné. Une valeur plus élevée représente un temps d'exécution plus faible.

misation 2.

— Enfin, dans la plupart des cas, Hybrid-DBT offre un niveau de performance plus élevé que celui du processeur OoO.

Notons que les résultats obtenus pour `deriche`, `gramschmidt`, `lu` et `ludcmp` avec le processeur `OoO` sont bien supérieurs aux autres. Après une étude plus poussée des résultats, nous avons réalisé que

ces applications sont celles montrant le plus haut taux de *cache miss*. La simulation du cache n'est pas la même dans les modèles GEM5 et dans nos simulateurs. Nous avons donc ré-exécutés les applications en utilisant le modèle GEM5 d'un processeur à exécution dans l'ordre. Ces exécutions ont montré un niveau de performance plus élevé, rendant l'accélération du `OoO` par rapport au *In-Order* semblable aux autres applications. Par conséquent, comparer les résultats de Hybrid-DBT et ceux du processeur *In-Order* a du sens car ils sont basés sur le même modèle. Cependant, comparer les résultats du *OoO* aux autres a moins de sens quand le taux de *cache miss* est élevé.

Les résultats obtenus pour `jacobi-2d` sont également particuliers : le niveau d'optimisation 1 offre un niveau de performance bien supérieur au niveau d'optimisation 2. L'étude des résultats montre que, lorsque le niveau d'optimisation 2 est utilisé, la grande majorité des cycles d'exécution est consacrée à du code généré par le `First Pass Translator`. A l'inverse, si le niveau d'optimisation est limité à 1, l'exécution passe la majorité de son temps sur du code ordonnancé. Cette différence est probablement due à un problème dans l'insertion des branchements vers le code optimisé.

Pendant cette étude expérimentale, nous avons également étudié l'énergie consommée lors de chaque exécution. Ces résultats sont représentés dans la figure 3.16, qui représente l'efficacité énergétique normalisée selon celle du processeur `In-Order` : une valeur plus haute représente une consommation énergétique moindre. On peut voir que l'approche basée sur Hybrid-DBT (au niveau d'optimisation 2) consomme 3,5 fois plus d'énergie que le processeur *In-Order*, mais également 2,6 fois moins que celle basée sur le processeur OoO.

Il est important de noter que les valeurs de consommation énergétique n'incluent pas la consommation des caches. En effet, notre approche ne modifie nullement le nombre d'accès ou le nombre de *cache miss*. Les caches ajoutent donc un surcoût constant pour chaque exécution (*In-Order*, Hybrid-DBT et OoO). La considération de ce surcoût réduit l'écart relatif entre les différentes exécutions. A titre d'indication, nous avons estimé le coût des caches pour l'exécution utilisant Hybrid-DBT au niveau d'optimisation 2. Notre expérience a montrée que les caches augmentent de 50% la consommation d'énergie de ces exécutions.

3.4.3 Intérêt de l'accélération matérielle

La troisième étude expérimentale étudie l'impact de l'accélération matérielle de la DBT sur la performance et l'efficacité énergétique de l'exécution. Pour ce faire, nous avons modélisé le temps d'exécution et la consommation énergétique des différentes phases de Hybrid-DBT. Pour ce faire, nous avons "simulé" la contribution du temps de traduction sur les performances globales, en considérant le cas ou celle-ci est réalisée en logiciel ainsi que le cas ou elle est réalisée en matériel.

La figure 3.17 représente le gain en temps d'exécution de la version accélérée par rapport à la version logicielle. En moyenne, l'exécution est 5% plus rapide pour la version accélérée. On peut noter que la valeur pour `jpeg_decode` se détache des autres. En effet, cette application s'exécute très rapidement (4 millions de cycles) et le modèle logiciel n'a pas le temps d'appliquer toutes les optimisations avant la fin de l'exécution. Cette application est donc très impactée par l'utilisation d'accélérateurs réduisant le coût de l'optimisation. Il est important de rappeler que nous avons fait le choix d'utiliser un processeur dédié à la DBT, qui est en charge d'appliquer les différentes optimisations. Ainsi, le temps de traduction

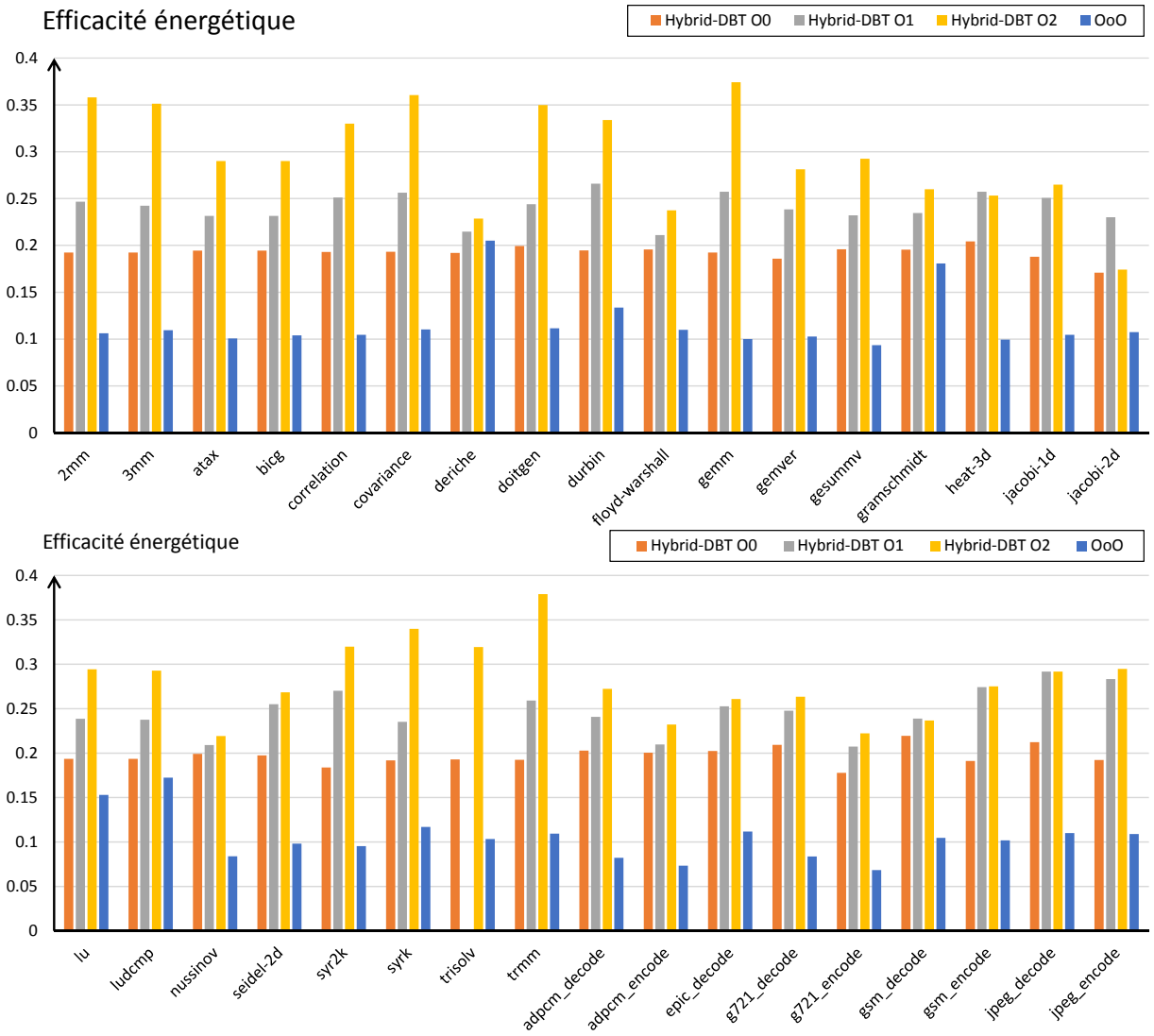


FIGURE 3.16 – Efficacité énergétique lors de l'exécution pour différents niveaux d'optimisation et différentes architectures de processeurs. Les valeurs affichées correspondent à l'énergie consommée par l'exécution sur le processeur *In-Order* divisée par l'énergie consommée dans le scénario concerné. Une valeur plus élevée représente une consommation d'énergie moindre.

n'impacte que faiblement le temps d'exécution d'une application.

Pendant ces exécutions, nous avons également estimé la consommation énergétique du système. Même si l'optimisation est réalisée par un coeur dédié, l'énergie consommée pour effectuer les optimisations est englobée dans l'énergie totale. La figure 3.18 représente la réduction de la consommation énergétique due à l'utilisation des accélérateurs. On peut constater que l'énergie consommée est réduite de 21%. Notons une fois encore que la consommation des caches n'est pas prise en compte dans les différentes études expérimentales réalisées. Leur considération réduirait certainement l'impact de ce gain énergétique. De plus, les résultats présentés ici sont basés sur le flot complet de Hybrid-DBT,

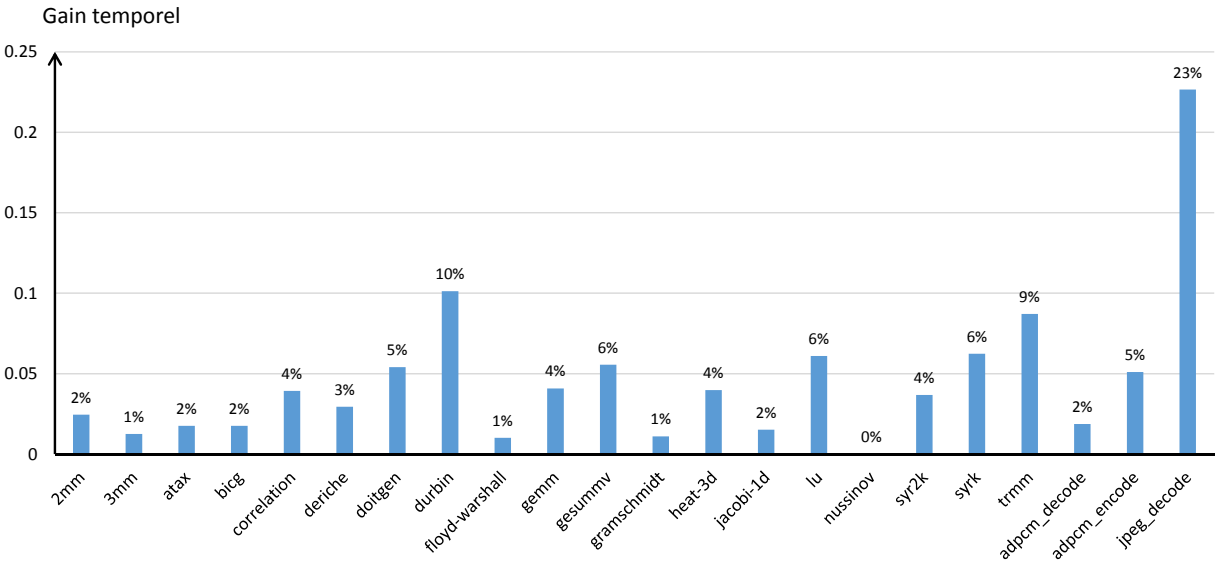


FIGURE 3.17 – Gain en temps d'exécution du à l'utilisation des accélérateurs matériels pour la DBT, comparé au flot de DBT entièrement logiciel.

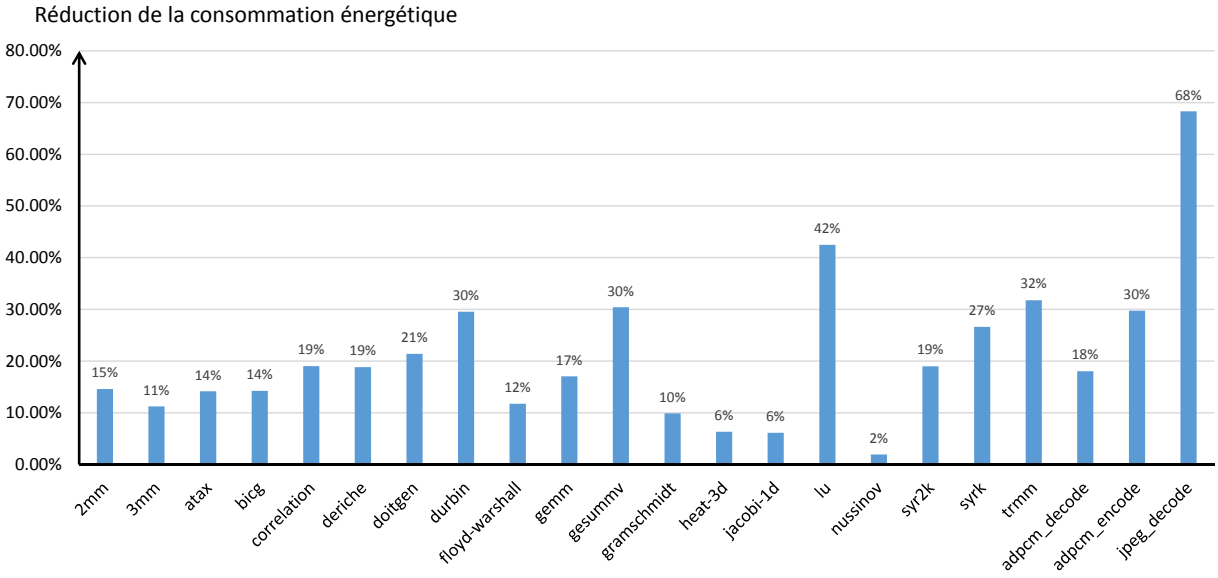


FIGURE 3.18 – Reduction de l'énergie consommée en utilisant les accélérateurs matériels pour la DBT, comparé au flot de DBT entièrement logiciel.

comprenant les phases d'optimisation continue que nous présenterons dans le chapitre 4.

3.4.4 Surcoût en surface

La dernière partie de l'étude expérimentale s'intéresse aux surcoûts matériels liés à la plateforme Hybrid-DBT. En effet, si les résultats précédents ont démontré l'utilité de l'accélération matérielle, il est

important d'évaluer également leur coût matériel. Les différents composants de la plateforme (ainsi que les coeurs Rocket [6] et BOOM [21]) ont été synthétisés pour une technologie 28 nm de STMicroelectronics, en utilisant Synopsys Design Compiler.

Composant	Surface (μm^2)
VLIW	106 621
First Pass Translator	6 146
IR Builder	15 374
IR Scheduler	7 155
Processeur DBT	30 792
Scratchpads	19 300

Processeur/Mémoire	Surface (μm^2)
Rocket	30 792
Hybrid-DBT	185 400
BOOM	513 048
Cache L1	69 290
Cache L2	540 469
Buffer de traduction	51 700

TABLE 3.1 – Surface des différents composants de la plateforme Hybrid-DBT, de la plateforme complète et des processeurs Rocket et BOOM.

Les tableaux 3.1 représentent ces coûts. La partie gauche précise le coût individuel de chaque composant de la plateforme Hybrid-DBT. Le tableau de droite précise le coût total de différents processeurs. Globalement, notre plateforme utilise une surface six fois plus grande que celle du Rocket, mais trois fois inférieure à celle du BOOM. Contrairement aux autres processeurs, le BOOM contient une FPU dont la surface est de $27\,000\,\mu m^2$.

Le tableau de droite contient également la surface des différents caches utilisés, estimée avec l'outil Cacti [8]. Le cache L1 est un cache de 16 ko avec des blocs de 64 octets et une associativité de 4. Le cache L2 est un cache de 256 ko avec des blocs de 128 octets et une associativité de 8. Enfin, nous fournissons également la surface d'un buffer de traduction de 32 ko. Cette dernière surface est donnée à titre indicatif : le système actuel ne modélise pas ce buffer. Pour les processeurs Rocket et Boom, le système comprend deux caches L1 et un cache L2. La plateforme Hybrid-DBT ne contient qu'un cache L1, un buffer de traduction et un cache L2.

3.5 Positionnement par rapport à l'état de l'art

Dans cette section, nous discutons du positionnement de notre approche par rapport à l'état de l'art. Nous abordons notamment les différences avec le CMS de Transmeta et l'architecture Denver de NVidia. Nous insistons également sur les différences de l'approche par rapport à un processeur superscalaire à exécution dans le désordre.

Le *Code Morphing Software* de Transmeta permet d'exécuter des binaires x86 sur un processeur VLIW. Tout comme notre outil, le CMS repose sur le principe du co-développement logiciel/matériel. L'idée est de développer un système en travaillant conjointement sur la partie logicielle et sur la partie matérielle. En effet, des modifications réalisées sur une de ces deux parties peut simplifier le développement de l'autre.

En pratique, l'architecture du VLIW utilisé par Transmeta propose plusieurs mécanismes permettant de simplifier l'optimisation du binaire. Par exemple, des mécanismes de *shadow register* et de *rollback/commit* des valeurs de ces registres permettent de simplifier les optimisations liées à la spéculation. A notre connaissance, Transmeta ne fait intervenir aucun accélérateur matériel dans son flôt de

traduction et d'optimisation. A l'inverse, nous avons choisi d'accélérer complètement les étapes clés de la traduction du binaire. L'une des faiblesses des processeurs Transmeta concernait le surcoût lors du démarrage d'une application. Notre approche permet de réduire drastiquement ce coût.

L'architecture Denver de NVidia partage plus de points communs avec notre approche. Comme nous l'avons vu dans la section 2.3.2, la traduction du *cold code* fait intervenir un composant matériel capable de traduire deux instructions ARM par cycle, offrant ainsi des performances identiques à celles d'un petit processeur superscalaire à exécution dans l'ordre. Ce composant est très similaire au `First Pass Translator` de Hybrid-DBT. La seule différence réside dans la manière d'utiliser cet accélérateur : dans notre approche, l'accélérateur est totalement déconnecté du processeur VLIW et traduit les binaires région par région ; au contraire, le décodeur matériel du Denver est intégré au sein du pipeline du processeur et permet de décoder les instructions juste avant qu'elles ne soient exécutées. L'approche du Denver permet donc de ne traduire que les instructions effectivement exécutées par le processeur. Toutefois, notre approche permet de simplifier la gestion de la DBT puisque toutes les instructions d'une région sont traduites et que l'outil conserve la trace des insertions d'instructions, permettant de convertir toute adresse du binaire d'origine en une adresse dans les binaires traduits.

La gestion de l'ordonnancement des instructions est le deuxième point de comparaison entre Denver et Hybrid-DBT. Le système développé par NVidia est basé sur un processeur superscalaire à exécution dans l'ordre. Par conséquent, un mécanime matériel de *dispatch* permet d'ordonnancer les instructions sur les unités d'exécution sans en modifier l'ordre. Cette approche permet de commencer à exploiter le parallélisme d'instructions dès l'exécution du *cold-code*. L'ordre des instructions ne peut être modifié que par une transformation logicielle, intervenant sur les régions de binaire fréquemment exécutées. Dans notre approche, aucun parallélisme d'instructions n'est exploité lors de l'exécution du *cold code*, mais l'utilisation d'accélérateurs matériels pour la génération de l'IR et pour l'ordonnancement des instructions permet d'arriver beaucoup plus rapidement à l'étape où l'ordre des instructions est remis en cause.

Enfin, le troisième point de notre discussion porte sur le positionement de notre approche par rapport à un processeur superscalaire à exécution dans le désordre (OoO). Tout comme un processeur OoO, notre système utilise un mécanisme matériel permettant de renommer les registres et d'ordonnancer les instructions sur les différentes unités d'exécution. Cependant, parce qu'il réalise une traduction du programme, notre approche permet de réutiliser les ordonnancements générés. En effet, le fait de travailler à l'echelle d'un bloc ayant un unique point d'entrée et le fait de ne pas modifier l'allocation des registres globaux permet d'assurer que l'ordonnancement généré est totalement équivalent au binaire d'origine. D'autre part, notre accélérateur est découplé du pipeline du processeur. Ainsi, une cadence de traduction d'instructions de 4 cycles est acceptable, là où le mécanisme d'ordonnancement d'un processeur OoO doit être capable de gérer plusieurs instructions par cycle.

Hybrid-DBT se présente donc comme une approche intermédiaire entre un processeur VLIW et un processeur OoO. Pour les processeurs VLIW, l'ordonnancement est généré lors de la compilation statique et ne peut pas être modifié ultérieurement. Le compilateur ne peut donc pas utiliser d'information de profilage dynamique pour générer un binaire de meilleure qualité et adapter les optimisations à la phase de l'exécution. A l'opposé, les processeurs OoO génèrent un nouvel ordonnancement à chaque instant de l'exécution, permettant de s'adapter à l'exécution de manière très efficace. Notre approche cherche à combiner le meilleur des deux mondes : les ordonnancements peuvent être générés rapi-

dement, permettant ainsi d'optimiser le binaire en fonction des informations de profilage. Notre outil pourrait être étendu pour construire dynamiquement les traces les plus probables à un instant donné et les ordonnancer (ce point est discuté dans les perspectives).

3.6 Conclusion

Dans cette section, nous avons présenté le fonctionnement du système Hybrid-DBT, qui forme la contribution principale de cette thèse. Cet outil de DBT tire parti de trois accélérateurs matériels afin de réduire le coût de la traduction et de l'optimisation. Les résultats experimentaux ont montré que l'utilisation de ces trois accélérateurs permet de réduire le coût de ces transformations de plusieurs ordres de grandeurs, et réduit également l'énergie consommée lors d'une exécution. La plateforme Hybrid-DBT permet d'atteindre un niveau de performance proche de celui d'un processeur OoO.

Pour aller plus loin, l'accélération et la réduction du coût en énergie du processus de traduction et d'optimisation permettent de développer des stratégies d'optimisation continue plus aggressives. Dans le prochain chapitre, nous présentons des flots d'optimisation continue basés sur Hybrid-DBT et permettant la gestion de processeurs VLIW dynamiquement reconfigurables, ou permettant de spéculer sur des dépendances mémoire. La première approche permet de spécialiser la configuration du VLIW à chaque procédure exécutée, tandis que la seconde permet d'augmenter la performance de Hybrid-DBT.

CHAPITRE 4

OPTIMISATION CONTINUE DANS UN ENVIRONNEMENT DE TRADUCTION DYNAMIQUE DE BINAIRE

Dans le chapitre précédent, nous avons présenté Hybrid-DBT, une chaîne de traduction dynamique de binaires exploitant des accélérateurs matériels ciblant la phase de génération des instructions. L'utilisation de ces accélérateurs permet d'augmenter la réactivité et de réduire la consommation énergétique d'un tel système, mais elle facilite énormément l'utilisation de techniques d'optimisation continue. Le principe de l'optimisation continue est d'analyser les binaires en cours d'exécution et de les ré-optimiser à la volée en tirant parti des informations obtenues dynamiquement. Dans ce chapitre, nous présentons deux flots d'optimisation continue intégrés dans Hybrid-DBT et exploitant le coût réduit de la génération de binaires.

Dans la première partie de ce chapitre, nous présentons rapidement la notion de processeur VLIW dynamiquement reconfigurable. Ces processeurs VLIW peuvent modifier, au cours de l'exécution, le nombre de voies d'exécution ou le nombre de registres disponibles. Chacune de ces configurations offre un compromis différent entre performance et efficacité énergétique. Le flot d'optimisation de Hybrid-DBT a été modifié pour explorer dynamiquement les différentes configurations du VLIW et choisir celle qui est la plus appropriée (la décision se prenant au niveau sous-programme). L'outil se base sur des directives du système pour pondérer la performance et la consommation énergétique. Ce sujet sera traité dans la section 4.1.

Dans la deuxième partie de ce chapitre, nous présentons une approche pour la spéculation de dépendance mémoire. Le principe de cette technique est de spéculer sur le fait que deux accès mémoire sont indépendants, afin de pouvoir ignorer cette dépendance lors de l'ordonnancement. Pendant l'exécution, un composant matériel, qui est similaire à la *Load/Store Queue* (LSQ) des processeurs OoO, est chargé de comparer les adresses des différents accès et de détecter des spéculations incorrectes. L'exécution de l'application doit alors réaliser un *rollback* et ré-exécuter certaines instructions en utilisant les valeurs correctes. L'outil Hybrid-DBT a été modifié pour construire dynamiquement des groupes de spéculation et les profiler afin de décider si la spéculation est rentable ou non. En effet, le gain lié à la suppression de la dépendance doit être supérieur au coût des différents *rollback*. Notre approche se base sur le principe du co-développement logiciel/matériel pour offrir une amélioration des performances de l'exécution tout en conservant un surcoût matériel le plus faible possible. Cet aspect est présenté en détail dans la section 4.2.

4.1 Support pour processeur VLIW dynamiquement reconfigurable

Dans la section 1.1.2, nous avons vu que l'utilisation de systèmes multi-cœurs hétérogènes permet de modifier dynamiquement le compromis entre la consommation énergétique et la performance. Nous avons également mis en avant l'intérêt de l'utilisation de systèmes à jeu d'instructions unique pour faciliter l'adaptation dynamique. Pour augmenter les compromis disponibles dans de tels systèmes, nous proposons d'ajouter des processeurs VLIW, via une traduction dynamique pour conserver l'illusion d'un jeu d'instructions unique.

Les approches basées sur les systèmes multi-cœurs hétérogènes permettent de s'adapter dynamiquement à la charge de travail du système ou à des contraintes extérieures. Cependant, ces techniques ne permettent pas de s'adapter aux caractéristiques de l'application en cours d'exécution. Par exemple, lors de l'exécution d'une application ayant un fort potentiel de parallélisme d'instructions, le système peut tirer parti d'une architecture offrant de nombreuses voies d'exécution. A l'inverse, si l'application présente un graphe de flot de contrôle complexe et des blocs de base de petite taille, le système favorise une architecture efficace énergétiquement. Au sein d'une même application, les différentes procédures peuvent offrir un niveau de parallélisme différent.

Les processeurs VLIW dynamiquement reconfigurables sont capables de modifier dynamiquement le nombre d'unités d'exécution actives ainsi que la taille de leur banc de registres. Pour ce faire, ils utilisent la technique de *power gating*. Le *power gating* permet de désactiver l'alimentation d'une partie de l'architecture, ramenant ainsi leur consommation statique et dynamique à des valeurs très faibles. Puisque le parallélisme d'instruction est explicitement encodé dans le jeu d'instructions, deux configurations du VLIW ne peuvent exécuter le même binaire.

Dans cette section, nous présentons un flot d'optimisation continue permettant d'exploiter des VLIW dynamiquement reconfigurable. Lors de l'exécution, l'outil explore les différentes configurations du VLIW, en fonctionnant au niveau sous-programme. L'outil est ensuite capable de décider de la meilleure configuration à utiliser en suivant différentes directives données par le système pour pondérer la performance et la consommation énergétique.

Dans le reste de cette section, nous commençons par présenter différentes approches permettant d'exploiter le *power gating* pour désactiver des voies d'exécution dans des processeurs. Nous présentons ensuite l'architecture du VLIW dynamiquement reconfigurable utilisée dans ces travaux, ainsi que l'ensemble des configurations utilisables. Ensuite, nous présentons les modifications apportées au flot de traduction et d'optimisation de Hybrid-DBT ainsi qu'aux accélérateurs matériels dans la sous-section 4.1.4. Enfin, nous présentons comment l'utilisation de ces différentes configurations peut permettre d'adapter dynamiquement le compromis entre consommation énergétique et performance de l'architecture avant de valider les résultats avec une étude expérimentale.

4.1.1 Les approches existantes

L'idée d'exploiter le *power gating* pour désactiver les unités d'exécution non utilisées a été étudiée suivant différentes approches, la plupart d'entre d'elles reposant sur une phase de compilation statique pour insérer les instructions pour activer ou désactiver les unités d'exécution.

Les travaux de Rele et al. [76] utilisent le mécanisme de *power gating* pour désactiver les unités d'exécution non utilisées dans le contexte d'un processeur superscalaire à exécution dans le désordre. Leur approche repose sur une première phase de compilation statique, qui consiste à chercher les zones où certains opérateurs sont peu utilisés et à les marquer à l'aide d'instructions spécialisées. Lors de l'exécution, l'outil exploite les mécanismes de prédiction de branchement pour détecter et ignorer les paires d'instructions d'activation et de désactivation trop rapprochées les unes des autres, pour ne pas désactiver une unité qui est utilisée peu de temps après.

L'approche de Roy et al. [78, 79] suit le même principe mais cette fois-ci appliqué à des processeurs embarqués. N'ayant plus accès aux mécanismes dynamiques de l'exécution dans le désordre, l'étape de compilation statique doit être plus précise lors de la détection des phases dans lesquelles une unité d'exécution n'est pas utilisée. Les travaux plus récents de Giraldo et al. [38] suivent la même approche pour désactiver les voies d'exécution ou réduire la taille de la file de registres dans un processeur VLIW. Les travaux de Tabkhi et al. [85] se basent également sur un analyse statique des binaires pour désactiver des parties de le banc de registres.

Anjam et al. [5] ont introduit en 2011 le processeur VLIW polymorphique. Ils montrent dans leurs travaux que l'adaptation du nombre de voies d'exécution à l'application exécutée permet de réduire la consommation énergétique. Dans leurs premiers travaux, un configuration fixe est assignée à chaque application lors de la compilation statique, et les binaires sont générés spécifiquement pour cette configuration. En 2013, Brandon et al. [15] introduisent le concept de binaires génériques capables de fonctionner sur toutes les configurations du VLIW. Ils permettent ainsi de changer la configuration du VLIW à tout moment de l'exécution pour s'adapter à des évènements extérieurs. Le fonctionnement des binaires génériques se base sur une contrainte supplémentaire, ajoutée lors de la compilation, qui impose que toutes les instructions d'un *bundle* puissent être exécutées indépendamment. Il est ainsi possible de couper un *bundle* de huit instructions en deux *bundles* de quatre instructions ou en quatre *bundles* de deux instructions. A partir de ces travaux, ils ont étudié comment prédire efficacement la meilleure configuration du VLIW à utiliser [42] ou encore comment décomposer les huit unités d'exécution d'un VLIW pour former plusieurs processeurs VLIW moins aggressifs, permettant ainsi de trouver un compromis dynamique entre le parallélisme d'instructions et le parallélisme de *threads* [16].

Grâce à l'utilisation d'un binaire générique, les binaires exécutés par les différentes configurations sont exactement les mêmes. Par conséquent, le surcoût nécessaire pour changer la configuration du VLIW est minime et ne mène à aucun problème de cache. Cependant, l'utilisation d'un format binaire spécifique à l'approche proposée peut freiner l'adoption du système, à cause du besoin de maintenir une chaîne de compilation spécifique. De plus, cette technique ne permet pas de développer un système hétérogène utilisant à la fois un processeur à exécution dans le désordre et des processeurs VLIW reconfigurables. Un tel système doit embarquer deux versions des binaires, limitant ainsi les possibilités d'adaptation dynamique.

L'utilisation des binaires génériques ajoute également des contraintes sur le choix de l'architecture du VLIW. En effet, lorsqu'un *bundle* est découpé, les instructions doivent être sur une voie qui est capable de les exécuter. Le VLIW doit donc être symétrique. De plus, ces binaires ne permettent pas de gérer la modification dynamique du nombre de registres utilisés. Notre approche va générer une nouvelle version des binaires spécifique à chaque architecture et est donc très flexible sur les architectures supportées.

4.1.2 Processeur VLIW dynamiquement reconfigurable

Avant de présenter les modifications apportées au flot d'optimisation de Hybrid-DBT, nous présentons dans cette section l'architecture du VLIW reconfigurable utilisée dans ce travail. Nous listons également l'intégralité des configurations exploitables par ce processeur.

Le VLIW reconfigurable que nous utilisons est une extension du VLIW de Hybrid-DBT. Ce processeur dispose d'un total de huit voies d'exécution, trois unités arithmétiques complexes et deux voies d'exécution dédiées aux accès mémoire. Il dispose également d'un banc de 64 registres (non partitionné). En utilisant des mécanismes de *power gating*, il est capable de désactiver la moitié du banc de registres ou des voies d'exécution (à une granularité de deux voies). L'étage de *fetch* dispose également de multiplexeurs permettant de router une instruction sur la voie d'exécution qui est activée.

Nous nous sommes basés sur les hypothèses de Roy et al. qui considèrent que 10 cycles sont nécessaires pour activer ou désactiver une des voies d'exécution [78].

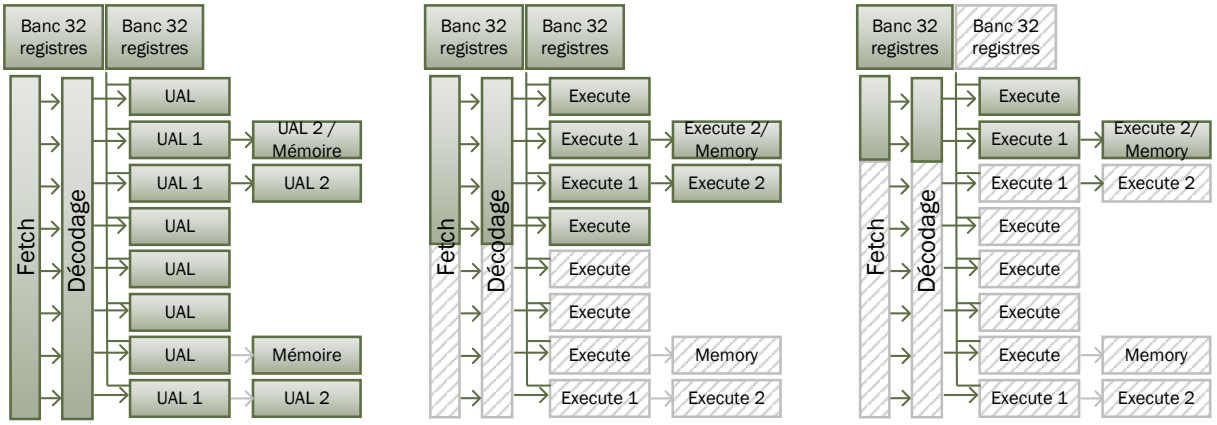


FIGURE 4.1 – Représentation de l'architecture du VLIW reconfigurable fonctionnant sous différentes configurations. Le premier exemple est une configuration haute performance qui utilise les huit unités d'exécuion et les 64 registres disponibles. Le deuxième exemple est une configuration intermédiaire n'utilisant que 4 voies d'exécution. Enfin, le troisème exemple est une configuration basse consommation n'utilisant que deux voies et 32 registres.

La figure 4.1 est une représentation simplifiée du VLIW utilisé dans trois configurations différentes : la partie la plus à gauche est la configuration *haute performance* où toutes les unités d'exécution sont activées, la partie centrale représente une configuration *intermédiaire* avec seulement 4 voies d'exécution et la partie de droite est la configuration *basse consommation*, avec seulement deux unités d'exécution et 32 registres. Notons également que la deuxième voie d'exécution (en partant du haut) permet de réaliser à la fois les multiplications et les accès mémoire. Cela permet d'obtenir une configuration à deux voies, capable d'exécuter l'intégralité des instructions.

L'architecture permet d'exploiter 22 configurations différentes, qui varient sur le nombre d'unités d'exécution, leur spécialisation ou encore sur la taille du banc de registres. La figure 4.2 introduit une notation des configurations qui sera utilisée dans la suite, et détaille toutes les configurations possibles de notre VLIW.

Dans la suite de cette section, nous présentons les différentes étapes du flot permettant d'exploiter

Nommage d'une configuration :
(#Voies, #Mem, #Mult, #Reg)

#Voies : Nombre de voies
#Mem : Nombre d'accès mémoire
#Mult : Nombre de multiplieurs
#Reg : Nombre de registres

Configurations Possibles		
(2,1,1,-)	(4,1,3,-)	(6,2,2,-)
(4,2,1,-)	(4,1,1,-)	(6,2,3,-)
(4,2,2,-)	(6,1,3,-)	(6,1,2,-)
(4,1,2,-)	(6,2,1,-)	(8,2,3,-)

FIGURE 4.2 – Description de notre convention de nommage des configurations et liste de toutes les configurations atteignables avec notre système. Pour chacune des 12 configurations listées ici, le banc de registres peut avoir 32 ou 64 registres.

efficacement les différentes configurations de ce VLIW.

4.1.3 Evaluation et classement des différentes configurations

Le principe de notre approche est de spécialiser la configuration du VLIW à une région de code de l'application. Etant donné que la reconfiguration du VLIW entraine un surcoût, un grain de reconfiguration trop fin risque de dégrader les performances. Dans nos travaux, nous avons fait le choix de spécialiser le VLIW à l'échelle des fonctions de l'application. Par ailleurs, seules les fonctions critiques de l'application profitent de cette spécialisation.

Pour chacune de ces fonctions, le flot modifié explore dynamiquement différentes configurations du VLIW. Pour chacune de ces configurations, un ordonnancement est généré et utilisé pour calculer deux scores reflétant les performances et la consommation énergétique de la configuration choisie.

Le score mesurant le temps d'exécution se base sur la taille des ordonnancements de chaque bloc de la fonction, multipliée par le taux d'utilisation de chacun de ces blocs. Ce taux d'utilisation est obtenu grâce à un profilage dynamique des différents blocs présents dans la fonction. L'objectif ici est de donner plus de poids aux blocs souvent exécutés dans la fonction. Par exemple, les blocs constituant le cœur d'une boucle sont très souvent exécutés et représentent donc une part importante dans le score de performance.

Plus concrètement, la formule utilisée pour obtenir le score d'une fonction f dans une configuration c est la suivante :

$$T_{exec}(f,c) = \sum_{b \in f} \rho_b * taille(b,c)$$

où ρ_b représente la fréquence du bloc b, et $taille(b,c)$ la taille de l'ordonnancement de b dans la configuration c.

Le second score que nous utilisons modélise l'énergie consommée lors de l'exécution de la fonction dans une configuration donnée. Pour ce faire, nous utilisons une liste pré-calculée de la puissance dissipée de chaque configuration du processeur. Cette liste a été estimée statiquement, en utilisant des outils de simulation (*gatelevel*) et d'estimation de la consommation. Le score énergétique est obtenu en multipliant le temps d'exécution estimé par la consommation de la configuration. Concrètement, la formule utilisée pour obtenir le score énergétique d'une fonction f dans une configuration c est la

suivante :

$$E(f,c) = P_c * T_{exec}(f,c) = P_c * \sum_{b \in f} \rho_b * taille(b,c)$$

où P_c est la puissance dissipée par la configuration c du processeur VLIW.

Ces deux scores sont très simples à calculer à partir de l'ordonnancement des différents blocs d'une fonction.

Ces scores obtenus pour les différentes configurations du VLIW, représentent différents compromis atteignables. Pour cette raison, on utilise la notion de front de Pareto : une configuration est sur le front de Pareto si et seulement si, il n'existe pas de configuration offrant des performances strictement meilleures tout en consommant moins d'énergie. La figure 4.3 représente les différents scores obtenus pour une multiplication de matrices avec notre outil. L'axe horizontal représente le temps d'exécution estimé (plus un point est à gauche, meilleures sont les performances) et l'axe vertical représente la consommation énergétique estimée (plus le point est bas, plus l'exécution a été efficace en énergie). Chaque point représente une configuration possible de notre VLIW. Enfin, la courbe rouge représente le front de Pareto obtenu pour cette fonction. Chacun des points sur cette courbe est un optimal au sens de Pareto.

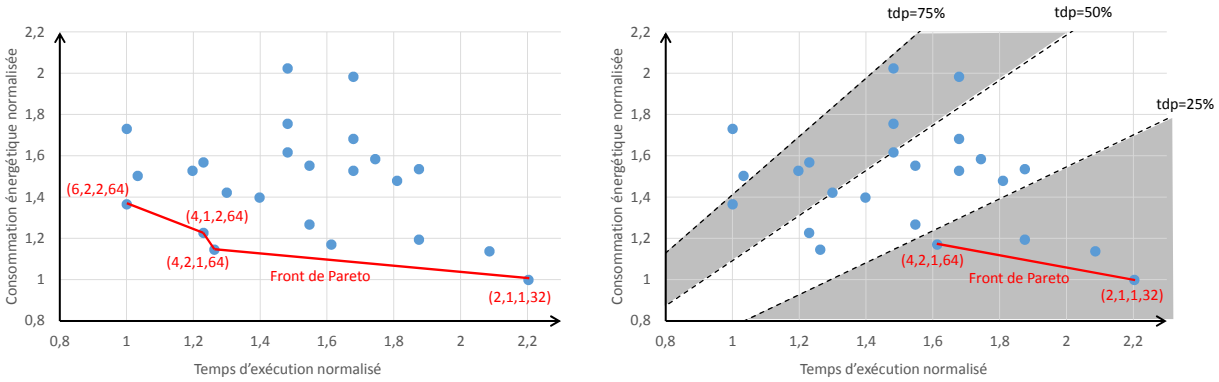


FIGURE 4.3 – Représentation des compromis entre performance et consommation énergétique pour une multiplication de matrices. Le graphique de gauche montre l'ensemble des configurations atteignables ainsi que le front de Pareto complet. Le graphique de droite ajoute la notion d'enveloppe thermique (tdp) et montre le front de Pareto lorsque le tdp est limité à 25% de son maximum.

Pendant l'exécution, le système doit sélectionner une configuration pour la fonction. Afin que le compromis soit acceptable, le point doit faire partie du front de Pareto. Pour choisir lequel utiliser parmi ces points, l'outil d'optimisation reçoit deux directives de la part du système d'exploitation :

— La directive ***energy_ratio*** permet de pondérer le score d'énergie par rapport au score de performance. Cette directive permet de choisir un point parmi le front de Pareto.

— La directive ***max_tdp*** permet de limiter la puissance instantanée dissipée. Cela permet d'interdire des points de fonctionnement dissipant trop de puissance.

La deuxième directive permet de distinguer la puissance dissipée à un moment donné de l'énergie consommée par une application. Si minimiser l'énergie peut sembler plus important, contrôler la

puissance dissipée par un cœur dans un contexte de multi-cœurs peut avoir un intérêt, car il offre une réponse aux enjeux du *dark-silicon*. La partie droite de la figure 4.3 représente les compromis disponibles pour la multiplication de matrices et y ajoute les informations sur l'enveloppe thermique. Quatre zones y sont représentées selon la limite qu'on impose sur le tdp. Le front de Pareto représenté est celui utilisé si la valeur de *max_tdp* était à 25% .

4.1.4 Modifications du flot de Hybrid-DBT

Dans la section précédente, nous avons présenté les différents scores utilisés pour évaluer une configuration donnée, ainsi que le concept de front de Pareto. Cette section est consacrée à la description du flot d'optimisation continue permettant de supporter les VLIW dynamiquement reconfigurables. Les changements apportés au flot de Hybrid-DBT sont illustrés sur la figure 4.4, qui reprend le flot d'origine (voir figure 3.5) de manière simplifiée et schématise les deux nouvelles phases permettant de gérer le VLIW dynamiquement reconfigurable : la phase d'exploration et la phase d'adaptation.

La partie droite de la figure 4.4 montre le mode de fonctionnement du VLIW aux différentes étapes du flot. Lors des deux premiers niveaux d'optimisation, lorsque l'outil n'exploite pas ou peu de parallélisme d'instructions, une configuration *basse consommation* du VLIW est utilisée (deux voies d'exécution, un accès mémoire, un multiplicateur et 32 registres). Lorsque le flot commence à optimiser des fonctions de l'application, il s'autorise à reconfigurer le VLIW pour augmenter le nombre d'instructions pouvant être exécutées en parallèle ou le nombre de registres disponibles.

Deux des trois accélérateurs présentés dans la section 3.3 ont été modifiés pour supporter les différentes configurations du VLIW : le `first-pass translator` et le `IR Scheduler`. Le `first-pass translator` a été légerement modifié pour pouvoir générer du code pour différentes configurations du VLIW. Selon la configuration, les accès mémoire ou les instructions de multiplication traduits ne sont plus placés à la même position dans un *bundle*. Les changements réalisés dans le `IR Scheduler` permettent de générer du code pour toutes les configurations du VLIW. Concrètement, la fenêtre du *scoreboard scheduler* a été agrandie pour supporter huit instructions par cycle et des registres peuvent être modifiés pour configurer l'activation ou la spécialisation de chaque voie. Ceux-ci sont utilisés lors de la phase d'exploration de la fenêtre pour dire si l'emplacement est un emplacement possible ou non.

Lorsque le flot d'optimisation rencontre une fonction fréquement exécutée et déclenche le niveau d'optimisation 2, il entre dans la **phase d'exploration** des différentes configurations. Il sélectionne une configuration parmi les configurations possibles, utilise le `IR Scheduler` pour générer un ordonnancement de chaque bloc et calcule les scores précédemment présentés. En utilisant les directives système (*energy_ratio* et *max_tdp*), l'outil peut calculer un score unique et décider si la configuration actuellement testée est meilleure que celle d'origine. Si elle l'est, les binaires générés sont validés sinon, ils sont abandonnés et une nouvelle configuration est testée.

Des instructions de reconfiguration doivent être insérées dans les points d'entrée du binaire généré pour que le VLIW se trouve dans la bonne configuration lors de l'exécution. Ces instructions doivent également être insérées aux points de sortie (c'est à dire lorsqu'un autre appel ou un retour de fonction est exécuté) pour revenir dans la configuration basse consommation.

En raison du faible coût de l'ordonnancement des instructions, la version actuelle de l'outil explore

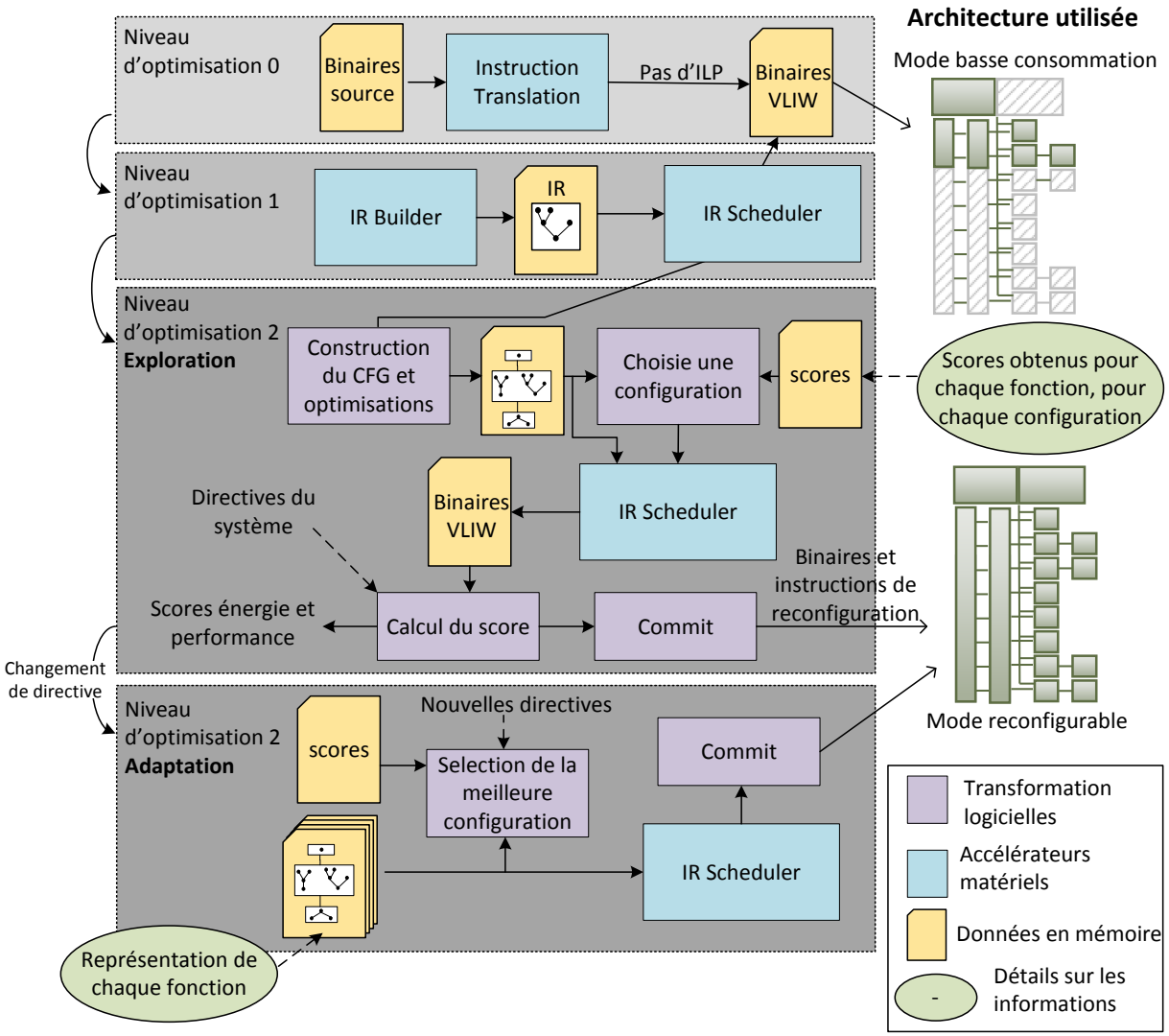


FIGURE 4.4 – Flot de traduction et d'optimisation de Hybrid-DBT modifié pour supporter les VLIW dynamiquement reconfigurables. Les niveaux d'optimisation 0 et 1 ont été simplifiés par rapport à la figure 3.5. Le niveau d'optimisation 2 est séparé en deux phases : la phase d'exploration et la phase d'adaptation. La partie droite de la figure représente le mode de fonctionnement du VLIW aux différents niveaux d'optimisation.

toutes les configurations du VLIW pour construire le front de Pareto. Cependant, il est possible d'affiner ce processus pour n'explorer qu'un sous-ensemble de configurations ayant une forte probabilité d'appartenir au front de Pareto. L'ordre dans lequel les différentes configurations sont explorées dépend des ordonnancements déjà établis. En effet, l'exploration démarre par la configuration basse consommation du VLIW et, à chaque fois qu'un ordonnancement est généré pour une nouvelle configuration, l'outil calcule un score secondaire permettant de mesurer le facteur limitant d'une configuration. Ce score est par exemple le nombre d'accès mémoire divisé par le nombre d'emplacements possibles pour les accès mémoire. Ensuite, à partir de ce score, le système choisit une nouvelle configuration qui augmente le

facteur limitant.

Lorsque toutes les configurations du VLIW ont été testées pour une fonction donnée et les scores correspondants établis, le flot d'optimisation entre dans la **phase d'adaptation**. L'outil conserve les différents scores et, lorsque les directives système changent, détermine la nouvelle meilleure configuration à partir de ces scores. Les binaires VLIW sont de nouveau générés et validés. Grâce aux différents accélérateurs matériels, le système est capable de réagir très rapidement à des changements de directive.

4.1.5 Etude expérimentale

Dans cette section, nous présentons les résultats expérimentaux mesurant l'intéret de la reconfiguration dynamique. Pour chaque expérience, le protocole expérimental est présenté succinctement. L'annexe A de ce document contient une présentation complète de tous les protocoles expérimentaux, ainsi qu'un lien vers un dépôt contenant les sources et les scripts permettant de reproduire ces études expérimentales.

Performance des différentes configurations de VLIW

L'objectif de la première étude expérimentale est de mesurer l'efficacité du flot de reconfiguration dynamique. Pour ce faire, nous avons exécuté les applications sur différentes configurations (fixes) du VLIW. Nous les avons également exécutées en activant le flot de reconfiguration dynamique, en utilisant deux valeurs différentes pour le coefficient *energy_ratio* : 10% et 90%. Pendant ces exécutions, nous avons collecté le temps d'exécution et la consommation énergétique.

La figure 4.5 représente la performance obtenue lors des différentes exécutions. Pour la plupart des applications, on peut observer les écarts de performances entre les configurations à 2, 4, ou 6 voies. Ces écarts démontrent que l'outil est capable d'extraire et d'exposer du parallélisme d'instruction. L'écart de performance entre les configurations 6 ou 8 voies est bien moindre : seule l'application `syrk` tire partie de ces voies d'exécution supplémentaires. Notons également que pour les applications `deriche`, `jacobi-2d` et `gsm` le changement de configuration du VLIW n'affecte que très peu la performance. Ces différentes applications ont déjà été mentionnées dans le l'étude expérimentale du chapitre 3 parce que l'outil ne les gère pas correctement.

La figure 4.6 représente l'efficacité énergétique des différentes exécutions. Une efficacité énergétique plus grande signifie que l'exécution a consommée moins d'énergie. On peut observer que, pour toutes les exécutions, la configuration à 2 voies est celle qui consomme le moins d'énergie. Les configurations à 4 et 6 voies sont souvent équivalentes. En effet, la configuration à 6 voies impacte grandement le temps d'exécution mais ne consomme pas beaucoup plus que la configuration à 4 voies. Enfin, la configuration à 8 voies est la moins efficace énergétiquement.

La première exécution exploitant le mécanisme de reconfiguration dynamique utilise un coefficient `energy_ratio` de 10% : l'outil privilégie la performance par rapport à l'efficacité énergétique. Sur les figures 4.5 et 4.6. On constate que pour la plupart des application, ce mode d'exécution offre un niveau de performance proche de la meilleure configuration, et une efficacité énegétique faible. Quand la configuration à huit voie n'apporte que peu de performance (par exemple l'application `lu`), l'outil n'utilise pas cette configuration et offre une efficacité énergétique plus grande. On constate enfin que, dans plusieurs

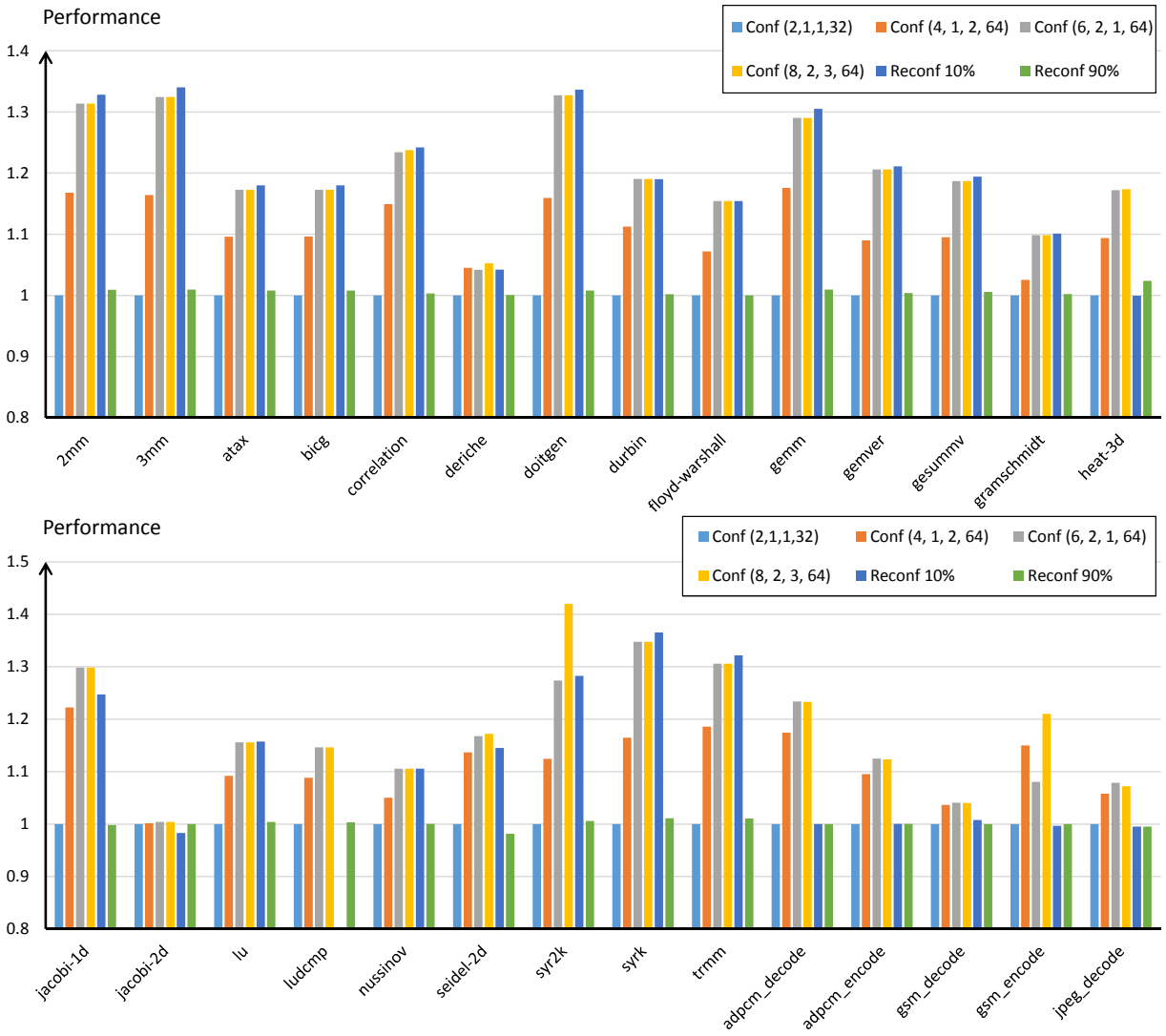


FIGURE 4.5 – Performance obtenue en utilisant différentes configurations du VLIW ou en utilisant le flot de reconfiguration dynamique présenté dans ce chapitre. Les quatre premières exécutions correspondent respectivement à des VLIW ayant deux, quatre, six ou huit voies d'exécution. Le flot de reconfiguration dynamique a été utilisé avec deux valeurs différentes pour ***energy_ratio*** : le premier utilise un coefficient de 10% (la performance est privilégiée), le deuxième utilise un coefficient de 90% (l'efficacité énergétique est privilégiée). Les résultats affichés correspondent au temps d'exécution sur le processeur VLIW à deux voies divisé par le temps d'exécution dans le scénario concerné. Une valeur plus haute signifie un temps d'exécution plus faible.

cas, cette configuration offre un niveau de performance légèrement meilleur que la configuration fixée à 8 voies. Cette variation est due à des différences dans la stratégie d'optimisation.

La seconde exécution exploitant le mécanisme de reconfiguration dynamique utilise un coefficient de 90%, privilégiant ainsi l'efficacité énergétique par rapport à la performance. On peut voir que, une fois encore, la configuration permet d'atteindre la meilleur efficacité énergétique possible, au prix de la

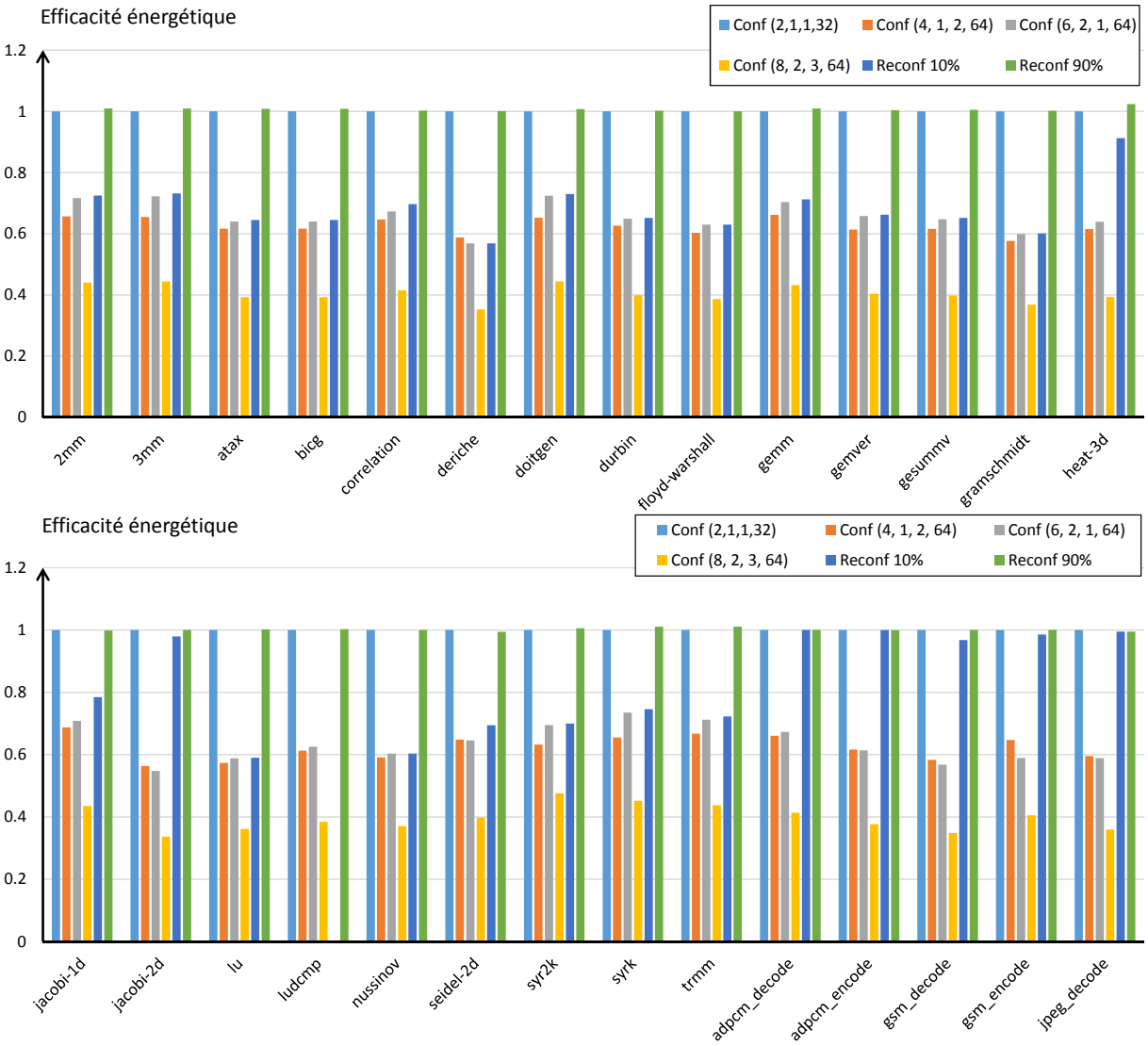


FIGURE 4.6 – Efficacité énergétique obtenue en utilisant différentes configurations du VLIW ou en utilisant le flot de reconfiguration dynamique présenté dans ce chapitre. Les quatre premières exécutions correspondent respectivement à des VLIW ayant deux, quatre, six ou huit voies d'exécution. Le flot de reconfiguration dynamique a été utilisé avec deux valeurs différentes pour ***energy_ratio*** : le premier utilise un coefficient de 10% (la performance est privilégiée), le deuxième utilise un coefficient de 90% (l'efficacité énergétique est privilégiée). Les résultats affichés correspondent à l'énergie consommée par l'exécution sur le processeur VLIW à deux voies divisée par l'énergie consommée dans le scénario concerné. Une valeur plus haute représente une consommation d'énergie moindre.

performance.

Utilisation des différents configurations

La seconde étude expérimentale réalisée mesure l'utilité des différentes configurations du processeur VLIW. Pour chacuns des sous-programmes de chacune des applications, nous avons construit le front de Pareto du compromis performance énergie. Nous avons ensuite comptabilisé le nombre d'apparition de chaque configuration du processeur VLIW dans un front de Pareto.

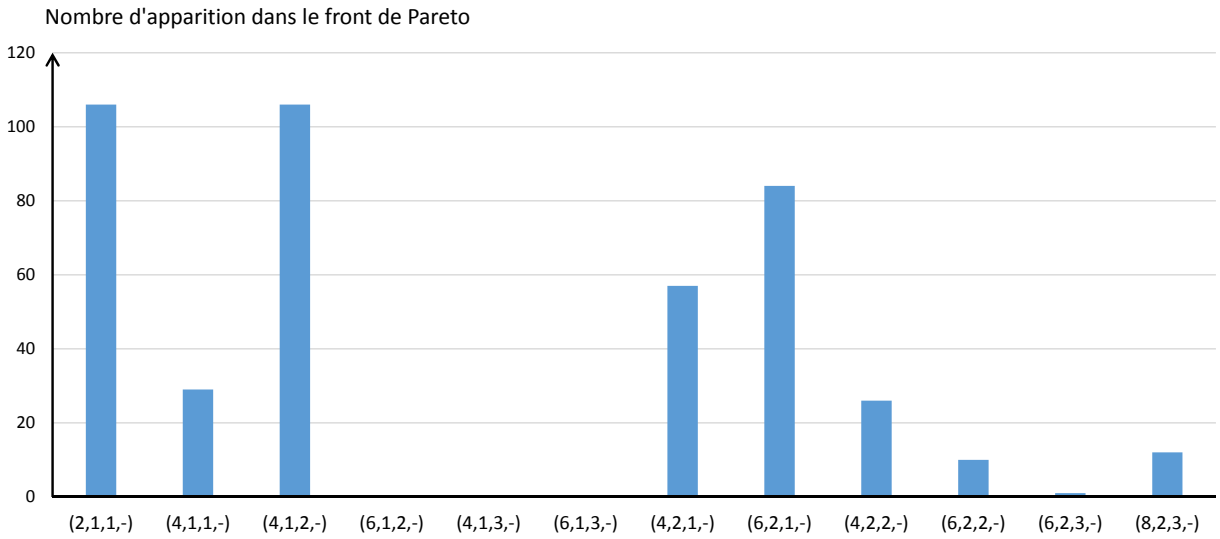


FIGURE 4.7 – Nombre d'apparition de chaque configuration dans le front de Pareto de l'un des sous-programme d'une application.

La figure 4.7 représente les résultats obtenus. On peut voir que la configuration (2, 1, 1, -) et (4, 1, 2, -) sont les plus présentes, sans doute parce que la différence de performance et de consommation entre elles sont très marquées. Les configurations offrant des hautes performances sont globalement utilisées, à l'exception de la configuration (6, 2, 3, -). Les configurations (6, 1, 2, -), (4, 1, 3, -) et (6, 1, 3, -) ne sont jamais utilisées, sans doute à cause du déséquilibre des unités d'exécution. En effet, ces configurations offrent un grand nombre d'unité d'exécution ou de multiplication pour un unique accès à la mémoire. Aucune des applications utilisées ici ne peut tirer partie de ces configurations.

Adaptation dynamique de la configuration

L'objectif de la dernière étude expérimentale est de donner un exemple de changement dynamique du coefficient *energy_ratio*. L'outil Hybrid-DBT a été modifié pour que le coefficient change tous les 7 millions de cycles. La figure 4.8 représente l'évolution du nombre d'instruction exécuté par cycle et de la puissance dissipée au cours du temps. Sur cette figure, on identifie très clairement les changements de configurations et leur impact sur l'IPC et sur la puissance dissipée.

Dans cette section, nous avons présenté le flot d'optimisation continue permettant de gérer les processeurs VLIW dynamiquement reconfigurable. Pendant l'exécution de l'application, l'outil explore les différentes configurations possibles pour chaque fonction. Cette approche permet de contrôler finement

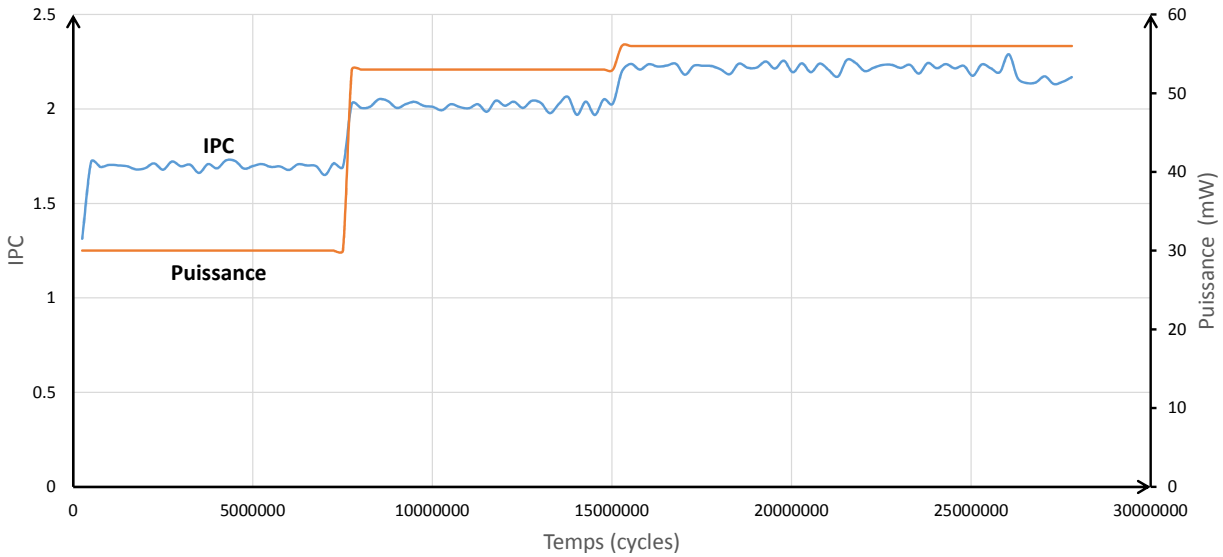


FIGURE 4.8 – Exemple de changements dynamiques de la valeur du coefficient ***energy_ratio***. Le nombre moyen d'instructions exécutées par cycle ainsi que la puissance dissipée sont représentés au cours du temps. Cette courbe a été obtenue avec l'application `trmm` et le coefficient est modifié tous les 7 millions de cycles.

le compromis entre la consomation énergétique et la performance du processeur. La prochaine section est consacrée à l'étude d'un deuxième processus d'optimisation continue permettant de gérer la spéculation de dépendances mémoire.

4.2 Spéculation mémoire

Dans la partie 1.2.2, nous avons présenté les travaux de McFarlin et al. qui analysent les performances des architectures à exécution dans le désordre [62]. Cette étude a démontré que les performances de ce type de micro-architecture proviennent essentiellement de sa capacité à tirer partie de l'exécution spéculative. Dans ces processeurs, les mécanismes de prédiction de branchement permettent d'explorer une large fenêtre d'instructions pour extraire le parallélisme d'instructions. De même, la *Load-Store Queue* permet de réorganiser les accès mémoire afin d'exécuter les chargements mémoire dès que l'adresse est connue. Cette technique est connue sous le nom de spéculation de dépendances mémoire. Pour réduire l'écart de performance entre les processeurs OoO et les processeurs à exécution dans l'ordre, McFarlin et al. prônent le développement d'architectures à exécution dans l'ordre ayant des mécanismes matériels permettant de faciliter la spéculation.

De telles architectures ont été développées dans les années 2000 avec, notamment, l'architecture Itanium, basée sur le jeu d'instructions EPIC. Celui-ci offre à la fois des mécanismes d'ordonancement dynamique simplifié et un support matériel pour la spéculation. Les processeurs Crusoe, Efficeon et Denver utilisent également ce type de mécanismes mais il n'existe que très peu d'informations sur leur fonctionnement.

Dans cette section, nous présentons les modifications apportées à Hybrid-DBT afin de permettre la spéculation de dépendances mémoire. L'objectif est de pouvoir exécuter un chargement mémoire avant une écriture mémoire sans être sûr que les emplacements mémoire ciblés soient différents. Cette technique de spéculation permet d'augmenter la performance délivrée par notre flot en exposant plus de parallelisme à l'ordonnanceur.

Plus précisement, nous avons ajouté un mécanisme matériel simplifié, similaire aux *load/store queue* utilisées dans les processeurs à exécution dans le désordre, qui est controlé par des instructions spécialisés. Nous avons également développé un système permettant de ré-exécuter une portion de code en cas de mauvaise spéculation. Le flot d'optimisation de Hybrid-DBT a été modifié afin d'ignorer certaines dépendances entre des *loads* et des *stores*, permettant ainsi d'exécuter les chargements mémoire plus tôt.

Dans la suite de cette section, nous commençons par présenter les approches existantes de spéculation de dépendances mémoire. Les parties suivantes sont consacrées à la description du flot d'optimisation modifié et des différents mécanismes matériels ajoutés au système. Enfin la partie 4.2.6 présente les résultats de notre étude expérimentale.

4.2.1 Rappels sur la spéculation de dépendances mémoire

Dans cette partie, nous présentons les différents mécanismes de spéculation de dépendances mémoire utilisés dans les processeurs OoO.

Comme nous l'avons mentionné dans la section 1.2.1 de ce document, ceux-ci peuvent changer l'ordre des instructions d'accès à la mémoire [43]. Un mécanisme appelé *Load/Store Queue* (LSQ) permet de vérifier les adresses de ces différents accès et de gérer les éventuels conflits. Dans la suite de cette description, nous utilisons les notations suivantes :

— la fonction age représente l'emplacement d'une instruction dans l'ordre séquentiel d'exécution. Ainsi pour deux instructions i et j, si $age(i) < age(j)$, alors l'instruction i doit être exécutée avant l'instruction j dans l'ordre séquentiel.

— la fonction $addr$ représente l'adresse d'un accès mémoire.

Pendant une exécution dans le désordre, plusieurs types de conflit mémoire peuvent se produire :

1. Une instruction de lecture mémoire l est exécutée avant une instruction d'écriture mémoire e, avec $age(e) < age(l)$ et $addr(e) = addr(l)$. Dans cette situation, la valeur obtenue pour l'instruction l n'est pas correcte.
2. Une instruction d'écriture mémoire $e1$ est exécutée avant une autre instruction d'écriture mémoire $e2$, avec $age(e2) < age(e1)$ et $addr(e2) = addr(e1)$. Dans cette situation, la valeur en mémoire est effacée par la deuxième instruction.

La LSQ permet de détecter et de résoudre ces deux types de conflits mémoire. Ce composant matériel est composé de deux mémoires. La première stocke l'adresse et l'âge de toutes les lectures mémoire réalisées. La seconde stocke l'adresse, l'âge et la valeur de chaque écriture mémoire exécutée.

La figure 4.9 illustre le traitement réalisé par la LSQ lorsqu'une instruction de lecture mémoire l est exécutée. Son adresse $addr(l)$ et son âge $age(l)$ sont stockés dans la *Load Queue*. En parallèle, pour chaque entrée s de la *store queue*, l'adresse et l'âge de s sont comparés à l'adresse et à l'âge de

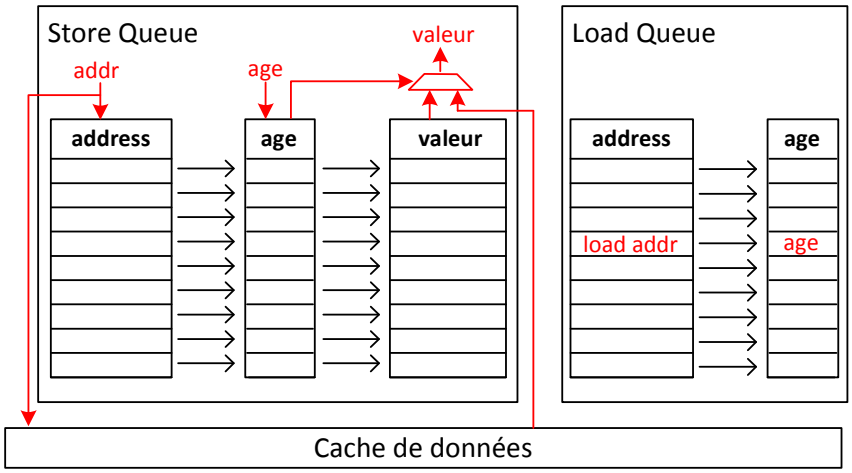


FIGURE 4.9 – Illustration du fonctionnement de la LSQ lors de l'exécution d'une instruction de chargement mémoire. L'adresse et l'âge de l'instruction sont stockés dans la *Load Queue*. Toutes les entrées de la *Store Queue* sont examinées à la recherche d'une éventuelle instruction écrivant à la même adresse mémoire et étant plus ancienne que l'instruction courante. Si une telle instruction est trouvée, la valeur stockée est utilisée comme résultat.

l'instruction l. Si $age(s) > age(l)$ et $addr(s) = addr(l)$, le résultat de la lecture mémoire est la valeur stockée dans la *store queue*. Si cette *store queue* contient plusieurs écritures mémoires à l'adresse demandée, la valeur de l'écriture la plus ancienne est retournée. Si la *store queue* n'en contient aucune, la valeur contenue dans le cache de données est utilisée.

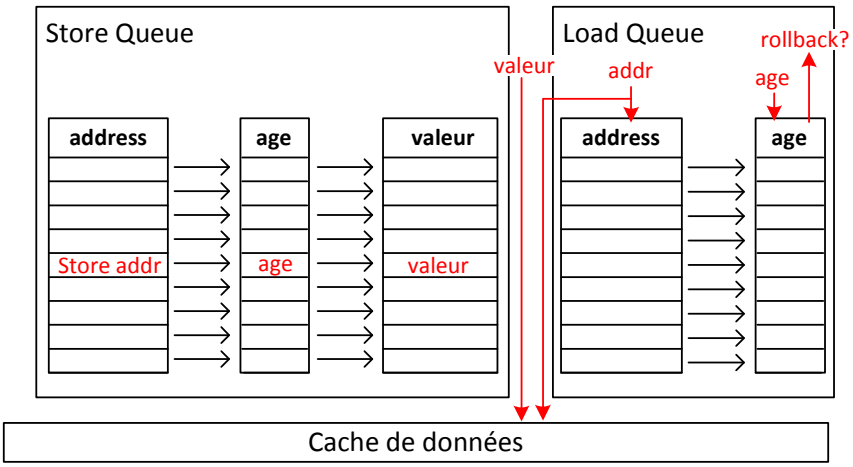


FIGURE 4.10 – Illustration du fonctionnement de la LSQ lors de l'exécution d'une instruction d'écriture mémoire. L'adresse, l'âge et la valeur de l'instruction sont stockés dans la *Store Queue*. Toutes les entrées de la *Load Queue* sont examinées à la recherche d'une éventuelle instruction accedant à la même mémoire et étant plus ancienne. Si une telle instruction est trouvée, un *rollback* est déclenché.

La figure 4.10 illustre le fonctionnement de la LSQ lorsqu'une instruction d'écriture mémoire s est exécutée. Son adresse $addr(s)$, son âge $age(s)$ et sa valeur sont ajoutés dans la *store queue*. En paral-

lèle, pour chaque entrée l de la *load queue*, l'adresse et l'âge de l sont comparés à l'adresse et à l'âge de l'écriture. Si $age(l) > age(s)$ et $addr(l) = addr(s)$, cela signifie que l'instruction de lecture mémoire l n'a pas renvoyé la bonne valeur. Par conséquent, elle doit être annulée, ainsi que toutes les instructions dépendant de cet accès. Cette annulation est possible grâce au mécanisme du *Re-Order Buffer*. L'écriture mémoire devient effective lors de la phase de validation dans l'ordre.

Les instructions de lecture et d'écriture mémoire sont retirées de la LSQ lorsqu'elles ont été validées.

La combinaison de la LSQ et du ROB autorise les processeurs OoO à exécuter les chargements mémoire aussi tôt que possible. Plusieurs travaux proposent des variantes pour réduire la surface de la LSQ en utilisant des structures similaires à des caches ou en filtrant les accès avec des filtres de Bloom [1, 80, 69, 84]. Par exemple, Roth et al. [77] proposent de réduire le coût de la vérification des adresses lors des écritures en ré-exécutant la lecture avant sa validation. Si la valeur a changé, l'instruction peut être invalidée.

Les mécanismes de LSQ et de ROB sont coûteux en surface et réduisent l'efficacité énergétique des processeurs OoO. Des mécanismes similaires ont été développés pour les architectures superscalaires à exécution dans l'ordre. Dans ces approches, des instructions spécialisées sont utilisées pour les accès spéculatifs à la mémoire, et un mécanisme similaire à la LSQ est utilisé pour stocker et comparer les adresses de ces instructions. Par exemple, Gallagher et al. [37] ont proposé un mécanisme appelé *Memory Conflict Buffer*. Celui-ci permet de stocker les adresses des lectures et écritures mémoire spéculatives. Après la spéculation, une instruction spécialisée permet de vérifier que la spéculation est correcte, et provoque un branchement dans le cas contraire. Dans ce type d'approche, le compilateur statique est chargé d'identifier les groupes d'instructions où la spéculation est prometteuse, d'ordonnancer les instructions en conséquence et de générer le code de gestion permettant de corriger une mauvaise spéculation.

Les processeurs Itanium de Intel utilisent une approche similaire appelée *Advances Load Address Table* (ALAT) [82]. Ces processeurs utilisent deux instructions pour vérifier la spéculation. La vérification complexe vérifie les adresses et branche vers une section de code spécifique en cas d'erreur. La vérification simple ré-exécute simplement le chargement mémoire. Cette seconde instruction ne peut être utilisée que si aucune instruction n'a utilisé le résultat du premier chargement mémoire.

Les informations disponibles sur le système CMS de Transmeta et sur le processeur Denver nous apprennent qu'ils utilisent un mécanisme similaire pour exécuter des chargements mémoire de manière spéculative [25, 12]. Cependant, ces architectures étant propriétaires, nous n'avons pas été en mesure d'obtenir des précisions sur la mise en oeuvre exacte de cette technique ni sur son surcoût matériel. Dans la suite de cette section, nous présentons notre mise en oeuvre de ces techniques au sein de Hybrid-DBT. Une étude expérimentale nous a permis d'estimer, à la fois en terme de performance et de surface, les gains obtenus via cette technique.

4.2.2 Aperçu du système de spéculation

Comme expliqué précédement, nous avons intégré un mécanisme de gestion de la spéculation de dépendances mémoires dans Hybrid-DBT. Nous avons ainsi modifié le flot d'optimisation logiciel ainsi que l'architecture matérielle du système.

Plus précisement, nous avons ajouté ou modifié les éléments suivants :

— l'accélérateur matériel `IR Scheduler` pour que l'ordonnancement généré permette d'annuler la spéculation. Celui-ci génère également des masques de *rollback*.

— deux composants matériels sont ajoutés au VLIW pour simplifier l'exécution spéculative : la *Partitioned Load/Store Queue* (PLSQ), qui est utilisée pour vérifier que l'hypothèse spéculée est effectivement correcte, et un mécanisme qui s'apparente à l'exécution prédiquée permet de ne pas exécuter certaines instructions chargées lors du *Fetch*.

— Nous avons modifié le deuxième niveau d'optimisation du flot afin d'identifier et d'extraire des groupes de spéculation.

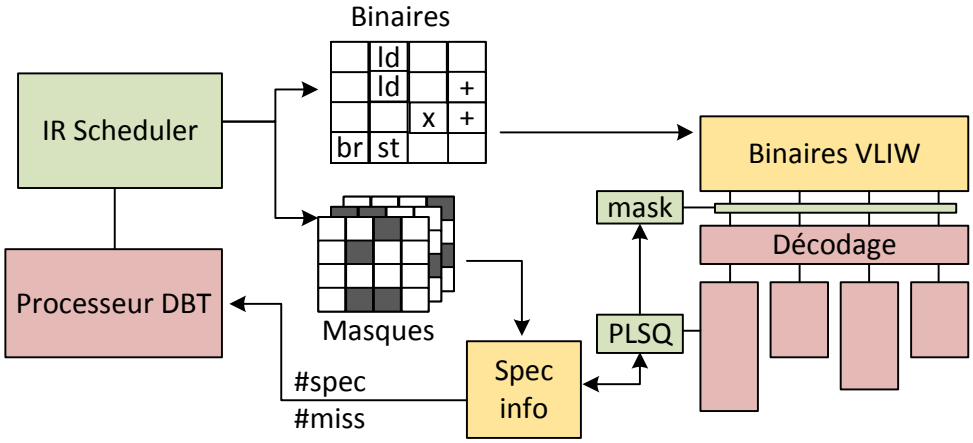


FIGURE 4.11 – Schéma simplifié du système de spéculation de Hybrid-DBT et des différents acteurs : le processeur DBT construit les groupes de spéculation et modifie la représentation intermédiaire, le `IR Scheduler` génère les binaires VLIW (avec spéculation) ainsi que les masques de *rollback*, le processeur VLIW possède deux nouveaux composants pour la spéculation : une PLSQ qui permet de vérifier la spéculation réalisée et de profiler les accès mémoires et un mécanisme pour masquer les instructions exécutées. Enfin une mémoire, appelée *Spec Info* contient les différents masques de *rollback* et les informations de profilage. Cette mémoire sera utilisée par le processeur DBT pour décider s'il faut spéculer ou non.

Le principe général du nouveau système est représenté sur la Figure 4.11. Le flot d'optimisation exécuté par le processeur DBT identifie des groupes de spéculation dans la représentation intermédiaire d'une procédure. Les dépendances de contrôle entre les accès mémoires et les écritures mémoires sont retirées de la représentation intermédiaire. Les instructions du bloc sont ensuite ordonnancées par le *IR Scheduler*, qui génère non seulement le binaire VLIW, mais aussi des masques de *rollback*. Ces masques permettent d'identifier les instructions ayant utilisé une valeur issue d'une instruction spéculée. Les binaires générés par le *IR Scheduler* sont exécutés par le VLIW et, à chaque exécution d'un accès mémoire spéculatif, l'adresse de l'accès est sauvegardée dans la PLSQ et comparée à l'adresse des opérations d'écriture mémoire du groupe de spéculation. Si la PLSQ détecte un alias (c'est-à-dire deux accès réalisés à la même adresse), l'exécution reprend à l'endroit du chargement fautif et ré-exécute le groupe d'instructions en utilisant le masque de *rollback*. Les instructions utilisant une valeur "contaminée" par la spéculation sont ré-exécutées mais pas les autres.

La PLSQ est également utilisée pour profiler les accès mémoire des groupes de spéculation : deux

compteurs sont utilisés pour compter le nombre d'exécutions d'un accès mémoire ainsi que le nombre d'alias détectés. Grâce à ces informations, le système d'optimisation continu peut décider s'il est rentable de spéculer ou non.

Dans la suite de cette section, nous présentons le flot d'optimisation modifié pour supporter la spéculation de dépendance mémoire, puis nous détaillons le fonctionnement de la PLSQ et les modifications apportées au `IR Scheduler`.

4.2.3 Description du flot de spéculation dynamique

Afin d'exploiter la spéculation de dépendances mémoire, nous avons modifié le flot d'optimisation dynamique de Hybrid-DBT. Ce nouveau flot est représenté sur la Figure 4.12. Les niveaux d'optimisation 0 et 1, qui correspondent aux optimisations à l'échelle des instructions ou des blocs de base, n'ont pas été modifiés. Le second niveau d'optimisation, qui consiste à analyser et optimiser à l'echelle du sous-programme, a été séparé en trois étapes : la construction des groupes de spéculation (voir ci-après), l'activation et la désactivation d'une spéculation.

Un groupe de spéculation est une paire d'ensembles (S, L) d'instructions mémoire telle que :

— les instructions de S sont des écritures en mémoire ;
— les instructions de L sont des lectures en mémoire ;
— pour chaque lecture $l \in L$, il existe au moins une écriture $s \in S$ située avant la lecture l dans le binaire RISC-V original ;
— pour chaque écriture $s \in S$, il existe au moins une lecture $l \in L$ située après l'écriture s dans le binaire RISC-V original ;
— une instruction d'écriture s ne peut appartenir qu'à un seul groupe de spéculation.

Plus concrètement, un groupe de spéculation doit permettre de retirer des dépendances mémoire. De fait, il n'est pas utile d'avoir dans un groupe une écriture mémoire sans aucune lecture qui en dépend, ni l'inverse.

Dans le second niveau d'optimisation, l'outil d'optimisation dynamique construit le graphe de flot de contrôle des procédures, déroule certaines boucles et construit des superblocs (voir la partie "Niveau d'optimisation 2" de la figure 4.12). Une fois ces transformations appliquées, l'outil d'optimisation analyse les blocs les plus critiques (les traces et les corps de boucle) pour construire des groupes de spéculation de manière gloutonne. Si le bloc analysé contient plus d'instructions qu'il n'y a d'emplacements dans le groupe de spéculation, un deuxième groupe est créé. Ce sont les caractéristiques de la PLSQ qui dictent la taille maximale d'un groupe de spéculation et le nombre de groupes de spéculation qu'il est possible de créer dans un même bloc. Ensuite, l'outil assigne à chaque groupe une adresse dans la mémoire `Spec Info` qui permettra de stocker les masques de *rollback* et les données de profilage.

Lorsque l'outil d'optimisation dynamique a analysé tous les blocs de la procédure, il va procéder à la génération des binaires VLIW, en utilisant le `IR Scheduler`. Il est important de souligner que, à cet endroit du flot, les groupes de spéculation sont construits mais les dépendances entre les opérations d'un groupe donné n'ont pas été supprimées. Les binaires VLIW générés ici n'utilisent donc pas la spéculation. Dans cette première étape, ils sont dans un mode dit de "**profilage**" : la PLSQ est utilisée pour compter le nombre d'utilisations des instructions du groupe et le nombre d'alias détectés. Ces

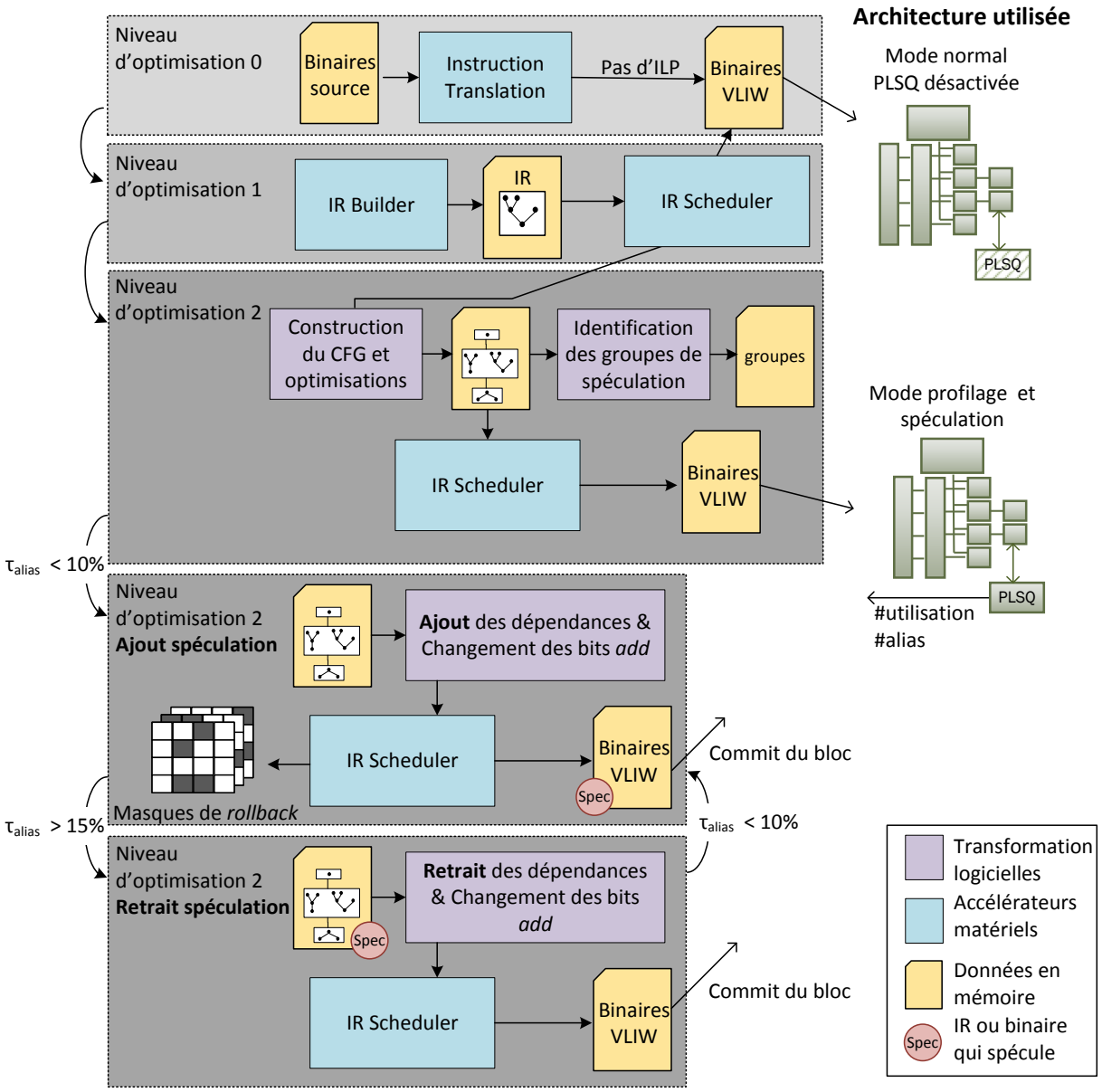


FIGURE 4.12 – Présentation du flot de traduction et d'optimisation exploitant la spéculation mémoire. Les niveaux d'optimisation 0 et 1 restent inchangés, et la PLSQ n'est pas utilisée à ce stade. Le niveau d'optimisation 2 commence par identifier les groupes de spéculations et génère des binaires en mode profilage. Si le taux d'aliasing d'un groupe passe sous les 10%, le système déclenche la spéculation : les dépendances mémoire sont retirées et les bits `add` des *loads* sont mis à 1. Le `IR Scheduler` est utilisé pour générer les binaires VLIW (avec spéculation) et les masques de *rollback*. Si le taux d'aliasing passe au dessus de 15%, le système coupe la spéculation : les dépendances mémoires sont ajoutées de nouveau, et les bits `add` des *stores* sont mis à 1. Puis le `IR Scheduler` est utilisé pour générer des binaires VLIW (sans spéculation).

informations de profilage sont sauvegardées dans la mémoire *Spec Info*, à l'emplacement assigné au groupe de spéculation.

En parallèle de l'exécution des binaires VLIW, l'outil de DBT parcourt régulièrement la liste des différents groupes de spéculation et leurs données de profilage. Si il détecte un groupe en **mode profilage** ayant un taux d'alias inférieur à 10%, il active la spéculation pour le groupe (voir la partie "Niveau d'optimisation 2 - Ajout spéculation" de la figure 4.12). Pour ce faire, il supprime les dépendances de contrôle entre les instructions du groupe de spéculation. Si deux instrutions du groupe possèdent également une dépendance de donnée, celle-ci n'est pas modifiée pour assurer un fonctionnement correct. Une nouvelle version des binaires est alors générée à partir de cette IR modifiée. Puisque qu'il y a des instructions spéculatives dans le bloc, le `IR Scheduler` génère également un masque de *rollback* pour chaque groupe de spéculation utilisé. Ce masque est stocké dans la mémoire `Spec Info`, et sera utilisé si un alias est détecté pendant l'exécution. Le groupe de spéculation passe alors en **mode spéculation**.

Quand l'outil de DBT détecte un groupe de spéculation en **mode spéculation** dont le taux d'alias est supérieur à 15%, il désactive la spéculation pour le groupe (voir la partie "Niveau d'optimisation 2 - Retrait spéculation" de la figure 4.12). Pour ce faire, il ajoute des dépendances de contrôle entre les différentes opérations du groupe de spéculation (seulement entre les écritures et les lectures situées ensuite). Une nouvelle version du binaire et alors générée par le `IR Scheduler`. Le groupe de spéculation passe en **mode profilage**.

Les valeurs de transition entre la phase de profilage et la phase de spéculation (10% et 15%) ont été choisies à partir des quelques exemples étudiés. Idéalement, il est possible de calculer le gain lié à la spéculation (différence entre la taille de l'ordonnancement avec et sans la spéculation) ainsi que le coût d'un *rollback*. L'outil peut ainsi trouver un palier à partir duquel la spéculation est rentable.

Les sous-sections suivantes sont consacrées à la description des modifications matérielles apportées au système.

4.2.4 Fonctionnement de la PLSQ

Le flot d'optimisation modifié de Hybrid-DBT permet de créer des groupes de spéculation, puis d'activer ou de désactiver la spéculation en fonction du taux d'alias mesuré. Dans ce nouveau flot, le composant matériel appelé *Partitioned Load/Store Queue* (PLSQ) est chargé de comparer les adresses des différents accès mémoires des groupes de spéculation dans le but de mesurer le taux d'alias et de déclencher des *rollback* lorsque cela est nécessaire.

L'organisation interne de la PLSQ est similaire à celle d'une *load queue* (cf. 4.2.1). Une des principales différences provient du fait que la PLSQ est partitionnée, c'est-à-dire qu'elle est organisée en plusieurs bancs, chaque banc correspondant à une *load queue* de petite taille. La PLSQ que nous avons développée est controlée de manière logicielle, en utilisant quatre instructions spécifiques ajoutées au jeu d'instructions de notre processeur VLIW :

— L'instruction `specInit` permet d'initialiser la PLSQ avec les informations d'un groupe de spéculation donné (masque de *rollback* et informations de profilage). Elle est composée d'un champ *specID* permettant d'identifier le banc de la PLSQ à utiliser et d'un champ *specInfo* qui correspond à l'adresse du groupe de spéculation dans la mémoire *Spec Info*. Cette instruction doit être terminée avant d'exécuter une instruction d'un groupe de spéculation.

— L'instruction `specClear` permet de remettre à zéro un banc de la PLSQ, c'est-à-dire de remettre

tous les champs "ages" à zéro pour ne pas détecter de faux alias avec les instructions qui seront exécutées ensuite. Cette instruction est uniquement composée d'un champ `specId` permettant d'identifier le banc de la PLSQ qui doit être remis à zéro. De plus, cette instruction déclenche l'incrément des compteurs de profilage de la PLSQ : le compteur d'utilisation d'un groupe de spéculation est incrémenté à chaque fois que l'instruction de remise à zéro est rencontrée, le compteur d'alias est, quant à lui, incrémenté si un alias a été détecté depuis la dernière remise à zéro.

— Les instructions `load` et `store` ont été modifiées : un bit a été ajouté pour déterminer si l'accès mémoire est spéculatif ou non. Si il est spéculatif, un champ `specId` permet d'identifier le banc de la PLSQ à utiliser et un bit `add` permet de décider si l'adresse doit être ajoutée dans la PLSQ ou comparée aux adresses déjà stockées. Ce bit `add` permet d'utiliser la PLSQ pour vérifier la spéculation (les `load` sont alors ajoutés dans la PLSQ) ou simplement pour profiler le groupe de spéculation (les `store` sont alors ajoutés dans la PLSQ).

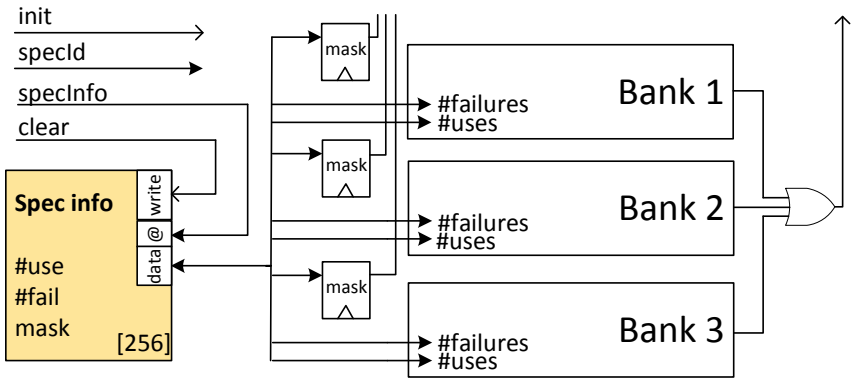


FIGURE 4.13 – Schéma de fonctionnement de la PLSQ, constituée de trois bancs. Outre ces bancs, le schéma représente également la mémoire `Spec Info` qui permet de stocker les masques de spéculation et les informations de profilage ; un registre pour chaque banc contient le masque à appliquer si un alias est detecté.

La figure 4.13 est une représentation simplifiée d'une PLSQ partitionnée en trois bancs. Les principaux signaux permettant de contrôler la PLSQ y sont représentés : `specId` et `specInfo`. A ces deux signaux s'ajoutent les bits `init` et `clear` placés à 1 lorsque l'instruction exécutée correspond à un `specInit` ou un `specClear` respectivement. La figure 4.13 représente également la mémoire `specInfo` et les différents bancs et registres utilisés pour stocker les masques.

Lorsqu'une instruction `specInit` est exécutée, l'adresse encodée dans `specInfo` est utilisée pour accéder au masque de *rollback* et aux informations de profilage. Ces informations seront ensuite stockées dans le banc correspondant, identifié par le signal `specId`. Les autres instructions sont traitées au niveau des bancs de la PLSQ.

La figure 4.14 représente l'organisation d'un banc de la PLSQ. Ce banc contient deux compteurs permettant de profiler les groupes de spéculation et six registres, organisés en registres à décalage, pour stocker des adresses et les bits d'âge ainsi que les différents comparateurs d'adresse. Les signaux d'entrée sont ceux des instructions présentées précédemment : `add`, `specId`, `address` et `clear`.

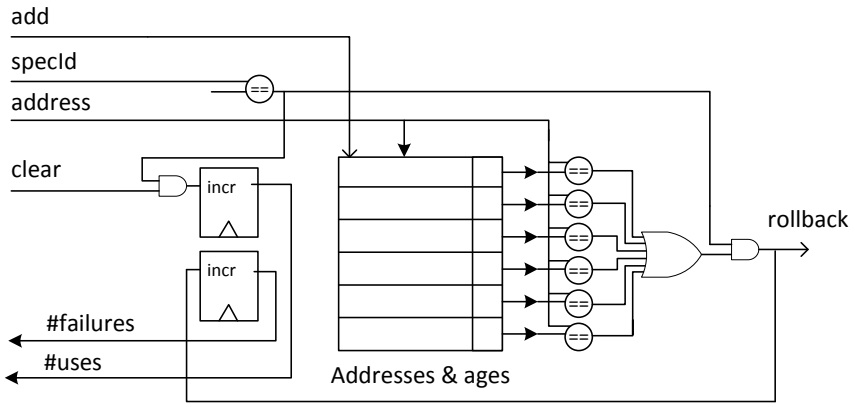


FIGURE 4.14 – Schéma de fonctionnement d'un banc de la PLSQ, capable de stocker 6 accès mémoires. Le schéma représente notamment les registres permettant de compter le nombre d'utilisations et le nombre d'alias, les registres pour sauvegarder les adresses accédées et les différents comparateurs.

Lorsqu'une instruction d'accès à la mémoire est exécutée et que son bit de spéculation est à 1, elle est traitée par un banc de la PLSQ. En fonction de la valeur du bit `add`, deux traitements sont possibles :

— si le bit `add` est à 1, l'adresse de l'instruction est ajoutée en entrée des registres à décalage, avec le bit d'âge mis à 1 ;

— si le bit `add` est à 0, l'adresse de l'accès mémoire est comparée aux adresses stockées dans les registres à décalage, dont le bit d'âge est à 1. Si une de ces adresses est identique à celle de l'accès courant, le signal de `rollback` est déclenché.

Toutefois, l'interprétation du signal *rollback* est effectuée par le processeur de VLIW, qui ne doit pas revenir en arrière si c'est un `store` qui le déclenche (car dans ce cas le groupe de spéculation est en mode profilage et non en mode spéculation).

Lorsqu'une instruction `specClear` est exécutée, tous les âges du banc concerné sont remis à zero et les compteurs sont incrémentés. La nouvelle valeur des deux compteurs est également inscrite dans la mémoire `specInfo`.

4.2.5 Modifications de l'accélérateur *IR Scheduler*

L'accélérateur matériel `IR Scheduler`, en charge d'ordonnancer les instructions de l'IR, a été modifié pour supporter la spéculation de dépendances mémoire. L'accélérateur modifié ordonnance les instructions et place les instructions de chargement mémoire aussi tôt que possible. Les autres instructions sont placées de façon à ce qu'aucun résultat nécessaire pour le *rollback* ne soit perdu et qu'aucune écriture mémoire possiblement fausse ne soit exécutée avant vérification. L'accélérateur construit également le masque de `rollback` qui permet d'identifier toutes les instructions qui doivent être ré-exécutées en cas de mauvaise spéculation.

Pour ce faire, l'algorithme propage des bits de contamination pendant l'ordonnancement afin d'identifier les instructions utilisant ou générant des données possiblement incorrectes. Un ordonnancement fin des instructions permet d'assurer que les binaires générés remplissent les contraintes données.

L'algorithme de *scoreboard scheduling* présenté dans la section 3.3.3 a été modifié pour respecter

```
1  for oneInstruction in IR do
2      earliestPlace = windowStart ;
3      for onePredecessor in oneInstruction.predecessors do
4          if onePredecessor.isData then
5              earliestPlace = min(earliestPlace, placeOfInstr[onePredecessor] + latency) ;
6          else
7              earliestPlace = min(earliestPlace, placeOfInstr[onePredecessor] + 1) ;
8          end
9      end
10     rdest = oneInstruction.getDest() ;
11     if oneInstruction.isAlloc then
12         rdest = freeRegisters.get() ;
13     end
14     earliestPlace = min(earliestPlace, lastRead[rdest], lastWrite[rdest]) ;
15     if poisoned[operand] & !instruction.isAlloc then
           /* Instruction have to be scheduled after last speculative store          */
16     end
17     for oneCycle in window do
18         for oneIssue in oneCycle.issues do
19             if oneIssue.isFree and oneCycle ≥ earliestPlace then
20                 place = oneCycle.address
21             end
22         end
23     end
24     if notFound then
25         window.move(max(1, earliestPlace - window.end)) place = window.end
26     end
27     if isSpecLoad then
28         poisoned[dest] = 1 ;                                        // Poison dest
29         mask[place] = 1 ;                                         // Mask instruction
30     end
31     if isSpecStore then
32         lastStore = ... ;                                  // Update last spec store
33     end
34     if poisoned[operand] then
35         poisoned[dest] = 1 ;                                        // Poison dest
36     end
37     if poisoned[operand] || isBranch then
38         mask[place] = 1 ;                                         // Mask instruction
39     end
40     window[place] = instrId ;
41     accelerators ...
42     ;
43 end
```

Algorithm 3: Modifications apportées à l'algorithme de *scoreboard scheduling* pour supporter le flot de spéculation proposé. Pendant l'ordonnancement, des bits de *poisoning* sont utilisés pour identifier les instructions utilisant et/ou créant des valeurs provenant d'un *load* spéculatif. Cette information est utilisée pour construire le masque de *rollback*.

ces nouvelles contraintes. L'algorithme 3 est la nouvelle version de l'algorithme. Trois nouveaux tableaux sont utilisés pour chaque banc de la PLSQ :

- Le tableau `lastStores` permet de stocker l'emplacement de la dernière instruction d'écriture mémoire ordonnancée pour le groupe de spéculation.
- Le tableau `poisonned` contient 64 bits de contamination, un pour chaque registre physique du VLIW. Ces bits permettent de déterminer si un registre contient une valeur possiblement incorrecte.
- Le tableau `mask` contient le masque de *rollback* en cours de construction. Chaque bit correspond à l'emplacement d'une instruction dans l'ordonnancement en cours. Si une instruction utilise un registre contaminé, le bit du masque correspondant à son emplacement est mis à 1. L'instruction est ainsi ré-exécutée en cas de mauvaise spéculation.

L'algorithme original a été modifié en y ajoutant deux étapes. La première intervient après l'étape d'allocation des registres temporaires et avant le parcours de la fenêtre d'ordonnancement. Le tableau `poisoned` est utilisé pour vérifier si les opérandes de l'instruction à ordonnancer sont contaminés par une opération spéculative. Si c'est le cas, le résultat de l'instruction risque d'être incorrect. Si le résultat est écrit dans un registre temporaire, l'instruction peut être placée sans risque car aucune donnée ne sera perdue. Si l'instruction écrit dans un registre global, il y a un risque d'écraser une valeur vivante (au sens de *liveness*), rendant le *rollback* inefficace. Dans ce cas, une contrainte est ajoutée pour que l'instruction soit ordonnancée après la dernière écriture mémoire du groupe de spéculation, dont l'emplacement est stocké dans le tableau `lastStores`. Si les opérandes de l'instruction ne sont pas contaminés, aucun traitement spécifique n'est requis.

La seconde étape intervient lorsque l'emplacement de l'instruction est connu. Celle-ci consiste à propager les bits de contamination et mettre à jour le tableau `lastStores` et le masque de spéculation. Si l'instruction ordonnancée est une écriture mémoire spéculative, le tableau `lastStores` est mis à jour avec l'emplacement de l'instruction. Si l'instruction est une lecture mémoire spéculative, le registre qui stocke son résultat est marqué comme contaminé. Si une instruction utilise un opérande contaminé, son résultat est contaminé également, et le bit du masque correspondant à son emplacement est marqué à 1.

4.2.6 Etude expérimentale

Dans cette section, nous présentons les résultats expérimentaux montrant l'impact de notre technique de spéculation de dépendances mémoire sur les performances et sur la surface du système. Pour chaque expérience, le protocole expérimental est présenté succinctement. L'annexe A de ce document contient une présentation complète de tous les protocoles expérimentaux, ainsi qu'un lien vers un dépôt contenant les sources et les scripts permettant de reproduire ces études expérimentales.

Performance avec spéculation mémoire

La première étude expérimentale menée mesure la performance de Hybrid-DBT avec et sans la technique de spéculation de dépendance mémoire. Cette expérience a été menée sur un processeur

VLIW à quatre voies (dont un unique accès à la mémoire) ainsi que sur un processeur VLIW à 6 voies (offrant deux accès à la mémoire).

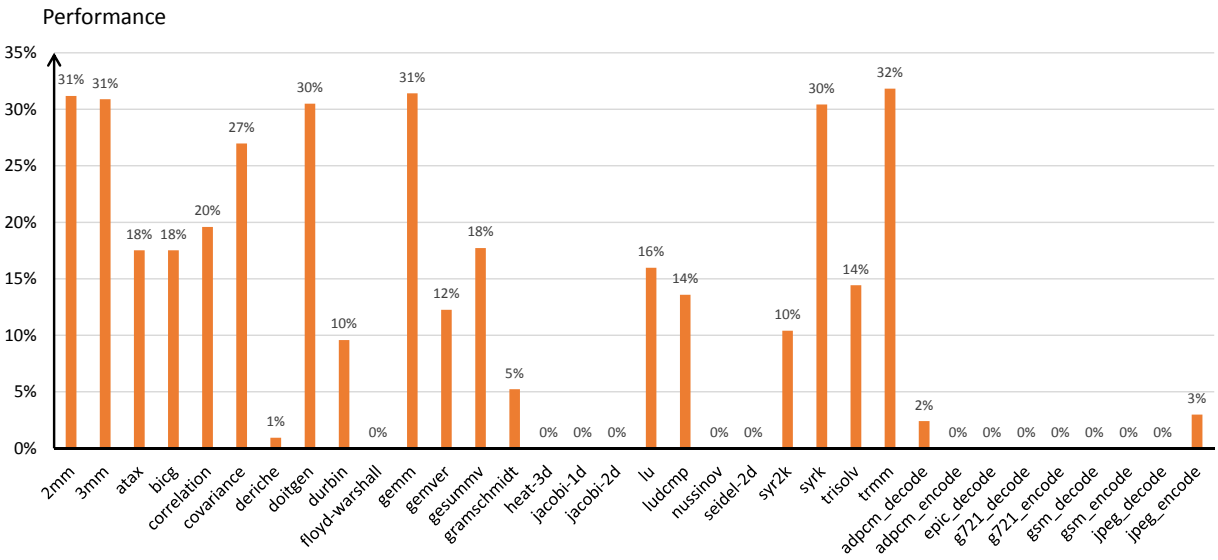


FIGURE 4.15 – Performance obtenue avec et sans le flot de spéculation de dépendances mémoire, pour un VLIW n'offrant qu'une unique accès à la mémoire. Une valeur plus haute signifie un temps d'exécution plus faible.

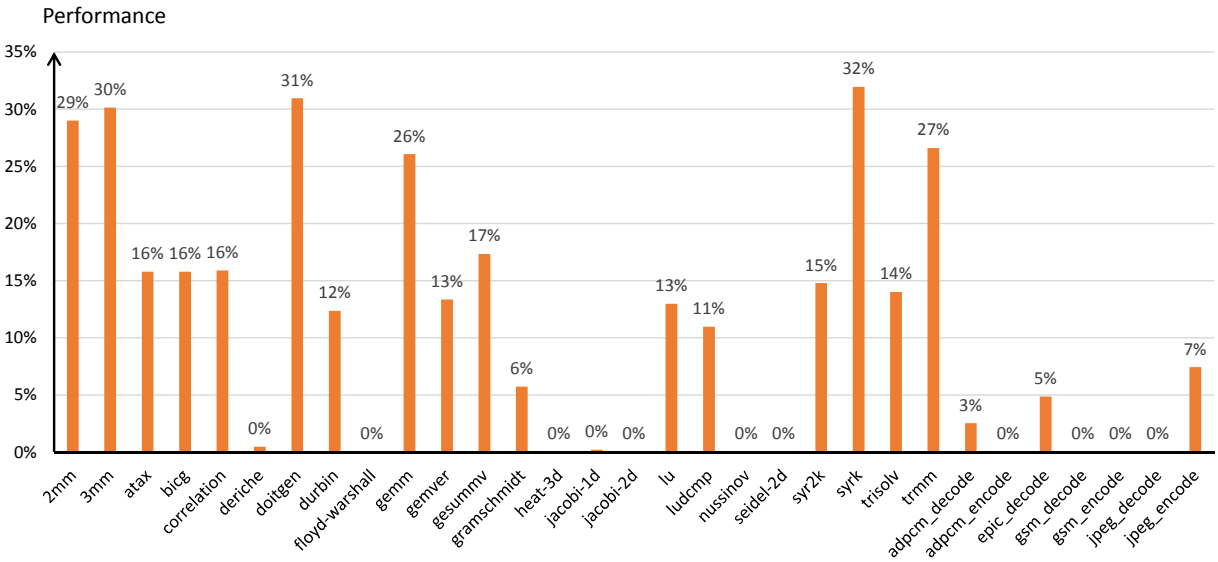


FIGURE 4.16 – Performance obtenue avec et sans le flot de spéculation de dépendances mémoire, pour un VLIW offrant deux accès à la mémoire. Une valeur plus haute signifie un temps d'exécution plus faible.

La figure 4.15 représente la performance des différentes exécutions sur le processeur VLIW à 4 voies. On remarque que la majorité des applications profite grandement de la spéculation de dépendances mémoire. En moyenne, cette technique réduit le temps d'exécution de 10%. On remarque également que plusieurs applications n'en profitent pas du tout. Parmi celles-ci, les applications `deriche`, `floyd-warshall`, `heat-3d`, `jacobi-1d` et `nussinov` ont été identifiées dans le chapitre 3 comme étant mal gérée par l'outil. En effet, Hybrid-DBT n'est pas capable d'identifier la boucle principale et de la dérouler. Une fois encore, cela peut être corrigé en implémentant la construction d'*hyperblock*. Pour les applications de Mediabench (les applications entre `adpcm-dec` et `jpeg-dec`) la structure du code est plus complexe et l'implémentation actuelle de la construction de trace ne permet pas de construire des régions contenant des groupes de spéculation.

La figure 4.16 représente la performance des différentes exécutions sur le processeur VLIW à 6 voies. Ce processeur permet d'exécuter deux instructions mémoire en parallèle. On peut voir que le flot de spéculation de dépendances mémoire apporte le même gain que pour un processeur n'offrant qu'une seule voie d'accès à la mémoire.

Surcoût en surface

La dernière étude expérimentale s'intéresse au surcoût du flot de spéculation de dépendances mémoire.

Composant		Surface (μm^2)
Rocket		30 792
VLIW	(4 issues)	106 621
BOOM		513 048
LSQ	size = 8	17 938
	size = 16	36 692
	size = 32	81 189
PLSQ	size = 4×4	4 144
Spec Info		6 300
IRScheduler		8 630

TABLE 4.1 – Surface des différents composants matériels.

Le tableau 4.1 contient la surface des composants modifiés ou ajoutés pour supporter le flot de spéculation. Nous avons également synthétisé la *Load/Store Queue* du processeur BOOM en modifiant sa taille. On peut observer que la surface utilisée par notre PLSQ est beaucoup plus faible que celle utilisée par la LSQ du BOOM. Deux points expliquent cette différence :

— notre PLSQ ne contient qu'une file alors que la LSQ du BOOM contient une file pour la *Load Queue* et une file pour la *Store Queue* qui stocke également la valeur à stocker en mémoire ;

— notre PLSQ est partitionnée, réduisant ainsi le nombre de comparaisons à réaliser en parallèle.

Globalement, l'ajout du `IR Scheduler`, de la mémoire `Spec Info` et de la PLSQ est moins coûteux que l'ajout d'une LSQ.

4.3 Bilan

Dans ce chapitre, nous avons présenté deux flots d'optimisation continue. Le premier permet de gérer les processeurs VLIW dynamiquement reconfigurables et de choisir dynamiquement la configuration la plus adaptée à chaque procédure d'une application. Le second flot permet de gérer la spéculation de dépendances mémoire pour pouvoir exécuter les chargements mémoire aussi tôt que possible.

Les résultats des différentes études expérimentales ont montré que le premier flot permet d'augmenter les performances ou de réduire la consommation du système en suivant des directives système. Le second flot permet d'augmenter la performance du système de 10% en moyenne, au prix d'un surcoût en surface faible.

Il est important de remarquer que ces deux flots d'optimisation utilisent de manière intensive l'accélérateur `IR Scheduler`. En effet, lors de la phase d'exploration du premier flot présenté, l'accélérateur est utilisé pour générer des ordonnancements pour chacune des configurations du VLIW, permettant ainsi de calculer leurs scores. Dans le deuxième flot, la version améliorée du `IR Scheduler` est utilisée pour passer du mode dit "profilage" au mode spéculation. Nos différents choix de conception reposent sur le niveau de performance de cet accélérateur.

CONCLUSION

Dans ce document, nous avons présenté nos travaux de recherche qui portent sur l'utilisation d'accélérateurs matériels pour la traduction dynamique de programmes binaires. Nous avons proposé un partitionnement logiciel/matériel pour un flot de DBT et développé trois accélérateurs pour permettre de réduire le coût de la traduction. Ceux-ci sont en charge de réaliser la première traduction des binaires, de construire la représentation intermédiaire et d'ordonnancer les instructions sur les différentes voies d'exécution du VLIW. L'utilisation de ces accélérateurs permet de diviser le coût de ces transformations par 150, en comparaison à une implémentation logicielle. Concernant la performance globale de notre système, nos résultats montrent également que l'accélération matérielle réduit le temps d'exécution de 5% et l'énergie consommée de 21%. Sur les applications de Polybench, notre système permet d'être 48% plus rapide qu'un processeur à exécution dans l'ordre et d'atteindre 98% des performances d'un processeur superscalaire à exécution dans le désordre.

Deux autres contributions ont porté sur l'exploitation de ces accélérateurs pour explorer plusieurs types d'optimisations continues. La première contribution consiste à exploiter des processeurs VLIW dynamiquement reconfigurables afin de déterminer la configuration la plus proche des besoins applicatifs. Cette approche permet d'offrir les meilleurs compromis entre la performance et l'efficacité énergétique. La seconde contribution consiste à exploiter des mécanismes de spéculation de dépendance mémoire en déterminant, à l'exécution, les dépendances à spéculer. Ici encore, notre approche logicielle/matérielle permet d'augmenter les performances du système de 10% tout en maintenant le surcoût matériel bas.

Le projet Hybrid-DBT est disponible sur Git-Hub [1]. Il est composé du flot de traduction et d'optimisation, de la description des trois accélérateurs et du processeur VLIW.

L'idée d'un support matériel pour la compilation n'est pas nouvelle : la machine Symbol [57], développée en 1971, est basée sur une approche similaire. Il a cependant fallu près de 10 ans pour la mise en oeuvre et le débogage de ce système. David Ditzel, le co-fondateur de la société Transmeta qui a proposé une analyse *post-mortem* du projet [27], a conclu qu'un compilateur était beaucoup trop complexe pour pouvoir être réalisé en matériel. C'est l'une des raisons qui a motivé le choix de Transmeta d'utiliser une approche logicielle lors du développement du *Code Morphing Software*.

Toutefois, l'amélioration récente des outils de synthèse de haut-niveau a grandement simplifié le développement matériel. En effet, grâce à la HLS, il est possible de décrire le comportement et l'architecture d'un composant et de générer une description matérielle rapidement. L'utilisateur peut ainsi explorer différents choix architecturaux, différents compromis entre la performance, l'énergie ou la surface du composant. De plus, l'utilisation du langage C permet de simplifier le partitionnement logiciel matériel d'un outil. Lors du développement de nos différents accélérateurs, l'intégralité du travail de développement (et de déboggage) a été fait sur la version C du programme. Lorsque le C est correcte,

1. https ://github.com/srokicki/HybridDBT

l'outil de HLS génère la description matérielle et une phase de simulation vérifie le comportement de l'accélérateur. Dans notre plateforme, les différents accélérateurs matériels, le processeur DBT ainsi que le processeur VLIW ont tous été développés via la synthèse de haut-niveau. Pour le processeur VLIW, nous avons en fait réalisé un modèle RTL du processeur en C. Retrospectivement, les travaux que nous avons réalisés n'auraient sans doute pas été possibles sans ces outils.

Perspectives de recherche

Les travaux de cette thèse ont porté sur la traduction dynamique de binaire. Bien qu'une petite communauté continue à étudier cette thématique de recherche, il n'y a que très peu de travaux sur la DBT ciblant les processeurs VLIW, similaires à Transmeta ou à Denver. Un tel outil est très complexe à mettre en oeuvre et demande de combiner une expertise en compilation, en micro-architecture et en conception matérielle (CAO), soit trois communautés disctinctes. Par ailleur, certains de nos résultats (accélération matérielle) se sont avérés difficiles à valoriser dans la communauté "compilation", qui reste focalisée sur les aspects algorithmiques. De même, nous avons pu constater que les enjeux liés à la compilation dynamique étaient souvent mal compris dans la communauté "CAO/systèmes embarqués".

Il est à noter que ces approches de DBT souffrent du fait que les principaux jeux d'instruction (ARM, x86) sont propriétaires, et il est de fait difficile de proposer ce type d'outils sans risquer de litiges. Par exemple, Microsoft a récemment proposé un outil permettant d'exécuter les applications x86 sur les systèmes Arm via une phase de traduction dynamique de binaire. Suite à cette annonce, la société Intel a menacé d'engager des poursuites contre Microsoft. Il est par ailleurs important de comprendre que développer un tel outil sous une licence open-source ne résous pas ce type de problème, bien au contraire (ceci explique peut-être la faible quantité de travaux dans le domaine).

Pour ces raisons, nous avons donc fait le choix d'un jeu d'instruction libre de droit, porté par le consortium RISC-V. Outre que ce choix nous a libéré des contraintes juridiques, il nous a également permis de bénéficier du dynamisme de cette communauté, qui s'est très fortement développée depuis le début de cette thèse. Dans cette logique, nous fait le choix de mettre notre outil à disposition de cette communauté.

Dans la suite de cette section, nous détaillons plusieurs pistes de recherche à court et moyen terme autour de ces problématiques.

Intégration dans un système d'exploitation

Les travaux de cette thèse se sont principalement concentrés sur le flot de traduction et d'optimisation ainsi que sur le développement des accélérateurs matériels. Des aspects essentiels du système n'ont pas été étudiés précisément. Par exemple, la gestion du cache de traduction est un élément important d'un outil de DBT qui n'est pas modélisé dans notre outil. Grâce à l'utilisation d'accélérateurs matériels, le système est capable de traduire une instruction RISC-V en moins de deux cycles, ou d'ordonnancer une instruction de l'IR en quatre cycles. Ce temps de traduction est similaire à celui d'un *cache miss*. La gestion du buffer de traduction est fortement influencée ces résultats de performance. Il serait intéressant de mesurer le niveau de performance du système en utilisant plusieurs tailles pour

le cache de traduction et plusieurs politiques de remplacement. Cette réduction de la taille du cache pourrait être un autre avantage lié à notre approche.

La gestion des codes auto-modifiants est également un aspect important. Non seulement il faut être capable de gérer ce type d'application en détectant les portions de code qui ont été modifiées et en invalidant la traduction qui a été faite, mais également en essayant de gérer ce code de manière efficace. Le fait d'exécuter des applications qui se modifient très souvent (des compilateurs dynamiques par exemple) va nécessiter de retraduire les binaires très souvent. Cela peut également augmenter l'intérêt de nos travaux. C'est d'ailleurs une piste intéressante pour la continuation de nos travaux : certains articles ont étudié les interactions possibles entre différents niveaux de compilation dynamique [88]. Par exemple lors de l'exécution d'une application Android (application déployée en *bytecode* Dalvik puis compilée dynamiquement) sur un processeur Efficeon de Transmeta, le premier compilateur dynamique va analyser le *bytecode* contenant des informations de haut-niveau pour générer des binaires x86 ; ensuite, le *Code-Morphing Software* de Transmeta va analyser ce binaire pour construire une représentation de plus haut niveau qu'il va utiliser pour générer des binaires VLIW. Dans un tel scénario, de l'information est perdue entre les deux outils de compilation dynamique et les auteurs montrent qu'il est possible de faire communiquer les deux outils pour améliorer la qualité du binaire généré. L'utilisation de notre flot *open-source* permettrait d'aller plus loin dans cette direction et peut-être de modifier les première étapes de la DBT pour faire en sorte, par exemple, que les outils de compilation Javascript génèrent une version riche de l'IR.

Construction dynamique de traces

Les résultats des différentes études expérimentales soulignent le besoin de construire des blocs d'instructions plus grands. L'implémentation des techniques classiques de compilation (par exemple la construction d'*hyperblock*) permettrait d'augmenter les performances de certaines applications. Dans la version actuelle, le processeur DBT réalise une construction très basique de *superblocks* avant que ceux-ci ne soient ordonnancés par le `IR Scheduler`.

Les processeurs VLIW et les processeurs OoO ont deux visions opposées de la construction de traces : dans les processeurs VLIW, les traces sont construites statiquement par le compilateur et ne sont jamais remises en question ; à l'opposé, le processeur OoO ré-ordonnance toujours les instructions situées sur sa trace d'exécution actuelle. Notre approche se situe entre les deux autres. En effet, l'utilisation des accélérateurs matériels pour l'ordonnancement des instructions permet de ré-ordonnancer très souvent les instructions.

On peut ainsi imaginer un mécanisme similaire aux prédicteurs de branchements utilisés dans les OoO, permettant de construire dynamiquement les traces les plus probables. Le processeur DBT peut ainsi les consulter régulièrement pour vérifier si une autre trace ne serait pas plus adaptée à l'état actuel de l'exécution. Une approche similaire a été proposée par Yourst et al. [92] sans toutefois qu'ils ne proposent d'implémentation des mécanismes ni d'informations sur le surcoût matériel.

Dans le flot d'optimisation de Hybrid-DBT, la fusion de deux blocs de l'IR est réalisée par le processeur DBT et est beaucoup plus coûteuse que l'ordonnancement des instructions. Si on souhaite ajouter un mécanisme capable de construire et de mettre à jour des traces d'instructions à une granularité très

fine, il faut également réduire le coût de cette transformation. Pour cela, nous pensons généraliser le processus de spéculation mis en oeuvre dans le chapitre 4. En effet, les modifications réalisées sur le `IR Scheduler` ajoutent des bits de contamination et un mécanisme permet de s'assurer qu'une instruction qui altère définitivement l'état de l'exécution (écriture mémoire ou modification d'un registre global) soit placée en dehors de la zone de spéculation. Une généralisation de ce mécanisme permet d'ordonnancer plusieurs blocs dans la même fenêtre d'ordonnancement sans avoir à modifier l'IR à chaque changement de trace.

Failles de sécurité liées aux mécanismes de spéculation

En janvier 2018, Kocher et al. rendent public le fonctionnement de la faille Spectre [53]. L'idée de cette faille est d'exploiter les mécanismes de spéculation et les caches des processeurs à exécution dans le désordre pour lire le contenu de zones mémoire protégées.

```
1  char array1[100], array2[65536] ;
2  int index = @target ;
3  if index<array1.size then
4  |   char a = array1[index] ;
5  |   int index2 = (a*0x100) + 0x200 ;
6  |   if index2<array2.size then
7  |   |   char b = array2[index2] ;
8  |   end
9  end
10 ... ;                    // Cache Side Channel
```

FIGURE 4.17 – Exemple d'attaques de type Spectre, exploitant les mécanismes de spéculation des processeurs à exécution dans le désordre.

La faille Spectre concerne les processeurs superscalaires à exécution dans le désordre capables de spéculation. La version la plus connue de l'attaque exploite le mécanisme de prédiction de branchements. Un exemple d'une telle attaque est donné sur la figure 4.17. Dans cet exemple, l'attaquant contrôle la valeur de $@target$ et fait en sorte que l'adresse $array1 + @target$ pointe vers la donnée qu'il souhaite obtenir. Le code utilisé est constitué de deux accès tableau : le premier accès se fait sur le tableau $array1$ à l'offset choisi par l'attaquant ; le second accès se fait sur le tableau $array2$, à un index calculé à partir du résultat du premier accès.

Dans un premier temps, l'attaquant entraine le prédicteur de branchement pour que le chemin entrant dans les deux if soit privilégié. Il va donc réaliser de nombreux accès à un index correct. Après un certain temps, l'attaquant s'assure que le tableau $array2$ n'est pas présent dans le cache, met l'index à la valeur $@target$ et exécute de nouveau le code. Le prédicteur de branchement prédit que l'exécution entre dans les deux if et exécute spéculativement le premier accès, le calcul du deuxième index, ainsi que le deuxième accès. A un moment donné, le processeur réalise que, cette fois-ci, l'index est inférieur à la taille du tableau et invalide les valeurs de a et de b. Cependant, en mesurant le temps d'accès aux différentes valeurs de $array2$, l'attaquant peut trouver laquelle est déjà présente dans le cache et, par conséquent, déduire la valeur de $index2$ et donc la valeur de a. Cette exploitation du cache pour faire

fuiter la donnée est appelée *Cache Side Channel Attack*.

Une deuxième variante de la faille Spectre exploite le mécanisme de spéculation de dépendances mémoire et la LSQ.

Les failles Spectre étant causées par les mécanismes de spéculation des processeurs OoO, un processeur VLIW ne devrait pas y être sensible. Toutefois, les différents mécanismes de spéculation des processeurs OoO trouvent leur équivalent dans des transformations de compilation : la prédiction de branchement équivaut à la création de traces et la spéculation de dépendance mémoire est également réalisée par certains compilateurs ciblant des processeurs VLIW. Dans notre plateforme, ces transformations sont appliquées dynamiquement. Par conséquent, notre système est vulnérable aux failles Spectre.

Les contre-mesures actuelles consistent à insérer une instruction permettant de stopper la spéculation dans les zones sensibles. L'identification de ces zones est faite statiquement par le compilateur. Cette solution a un impact très important sur la performance des processeurs OoO car la spéculation est la cause principale de leur niveau de performance.

L'étude de ces failles dans le contexte des processeurs VLIW basés sur la DBT est donc une piste de travail intéressante. En effet, nous pensons que ces contre-mesures sont plus simples à mettre en place pour de tels systèmes. La raison est que les outils de DBT ont un contrôle beaucoup plus précis de la spéculation et de l'ordonnancement généré. Ces outils peuvent profiler l'exécution et détecter des indicateurs de faille Spectre, puis bloquer ou limiter la spéculation. De plus, les différentes failles Spectre ne sont exploitables qu'à travers une double indirection (accès à la valeur, calcul d'un index dépendant de cette valeur et accès à la mémoire). Un tel schéma d'accès peut être identifié dans notre flot en propageant les bits de contamination (voir la section 4.2.5) et des contraintes peuvent être ajoutées pour que le deuxième accès soit placé en dehors de la zone de spéculation. Cette solution aurait un impact beaucoup plus faible sur les performances.

Au-delà des failles Spectre, étudier les aspects de sécurité liés à l'utilisation de la DBT est également une piste de recherche prometteuse.

Utilisation de plateformes multi-cœurs hétérogènes avec Hybrid-DBT

Notre principale motivation lors du développement de l'outil Hybrid-DBT est son utilisation dans un système multi-cœurs hétérogène. En effet, notre outil permet d'ajouter un autre type de cœur dans ces systèmes, tout en conservant l'illusion d'un jeu d'instructions unique. Nos résultats expérimentaux montrent que, sur certaines applications, Hybrid-DBT permet d'atteindre un niveau de performance supérieur à celui des processeurs OoO, tout en consommant moins d'énergie. L'utilisation d'une architecture VLIW permet de s'adapter en fonction des caractéristiques de l'application à exécuter, là où les systèmes big.LITTLE actuels ne permettent de s'adapter qu'à la charge de travail.

Dans un tel système, allouer dynamiquement un processeur à une application en cours d'exécution est un problème non trivial. En effet, il est difficile de prévoir le niveau de performance d'un processeur sur une application donnée. La génération dynamique de cette allocation constitue une piste de recherche intéressante. Il faut définir un ensemble d'indicateurs de performance et les extraire pendant l'exécution de l'application sur l'un des processeurs. Par exemple, ces indicateurs peuvent être

la taille moyenne des blocs de base, la proportion d'accès mémoire, le taux de *cache-miss* ou encore des informations liées à la prédiction de branchements. Dans des architectures OoO, des indicateurs plus poussés peuvent capturer le taux de réutilisation d'un ordonnancement ou le taux de mauvaise spéculation. Une fois ces indicateurs réunis, des algorithmes de classification peuvent être utilisés pour décider du meilleur processeur à utiliser. Par exemple, une application ayant des blocs larges, avec peu d'accès à la mémoire et des ordonnancements réguliers est une candidate parfaite pour une exécution sur VLIW. Cette approche peut être complexifiée en ajoutant d'autres types de processeurs.

Les systèmes multi-cœurs hétérogènes offrent également d'autres possibilités. Par exemple, il est possible d'utiliser ces indicateurs de performance pour identifier plus rapidement les points chauds de l'exécution et pour les optimiser en avance de phase. Ainsi, l'application est migrée sur le processeur VLIW quand l'outil de DBT a déjà généré une version efficace des binaires.

Liste des publications

Les travaux de cette thèse ont mené à trois publications dans des journaux ou dans des conférences internationnales, une publication dans une conférence française et une publication dans un *workshop* international. Ces publications sont les suivantes :

- Simon Rokicki, Erven Rohou et Steven Derrien, ***Hybrid-JIT : Hardware Accelerated JIT Compilation for Embedded VLIW Processor*** au *5th International Workshop on Dynamic Compilation Everywhere*, 2016, Barcelone
- Simon Rokicki, Erven Rohou et Steven Derrien, ***Hybrid-JIT : Compilateur JIT Matériel/Logiciel pour les Processeurs VLIW Embarqués*** à *Conference d'informatique en Parallelisme, Architecture et Systeme (Compas)*, 2016, Lorient
- Simon Rokicki, Erven Rohou et Steven Derrien, ***Hardware-Accelerated Dynamic Binary Translation*** à *IEEE/ACM Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2017, Lausanne
- Simon Rokicki, Erven Rohou et Steven Derrien, ***Supporting Runtime Reconfigurable VLIWs Cores Through Dynamic Binary Translation*** à *IEEE/ACM Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2018,
- Simon Rokicki, Erven Rohou et Steven Derrien, ***Hybrid-DBT : Hardware/Software Dynamic Binary Translation Targeting VLIW*** dans le journal *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018

Ces travaux ont également été présentés aux $10^{\text{èmes}}$ Journées de la Compilation en 2016 à Banyuls-sur-Mer, lors du $6^{\text{ème}}$ Workshop RISC-V en 2017 à Shanghai et un poster a été présenté à la conférence *International Symposium on Code Generation and Optimization* en 2016 à Barcelone.

Enfin, un article nommé *Aggressive Memory Speculation in HW/SW Co-Designed Machines* a été soumis à DATE'19 et est actuellement en phase d'évaluation par les pairs. Celui-ci reprend les travaux sur la spéculation de dépendances mémoire présentés dans le chapitre 4.

BIBLIOGRAPHIE

[1] J. ABELLA et A. GONZALEZ. « SAMIE-LSQ : Set-Associative Multiple-Instruction Entry Load/Store Queue ». In : *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*. Avr. 2006, DOI : 10.1109/IPDPS.2006.1639290.

[2] Santosh G. ABRAHAM, Waleed M. MELEIS et Ivan D. BAEV. « Efficient Backtracking Instruction Schedulers ». In : *Parallel Architectures and Compilation Techniques, 2000. Proceedings. International Conference On*. IEEE, 2000, p. 301–308.

[3] Giovanni AGOSTA et al. « JIST : Just-In-Time Scheduling Translation for Parallel Processors ». In : *Sci. Program.* T. 13. Juil. 2005, p. 239–253. DOI : 10.1155/2005/127158.

[4] D. H. ALBONESI et al. « Dynamically Tuning Processor Resources with Adaptive Processing ». In : t. 36. IEEE Computer Society, déc. 2003, p. 49–58. DOI : 10.1109/MC.2003.1250883.

[5] Fakhar ANJAM, Muhammad NADEEM et Stephan WONG. « Targeting Code Diversity with Run-Time Adjustable Issue-Slots in a Chip Multiprocessor ». In : *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011, p. 1–6.

[6] Krste ASANOVIĆ et al. *The Rocket Chip Generator*. Rapp. tech. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, avr. 2016.

[7] R. I. BAHAR et S. MANNE. « Power and Energy Reduction via Pipeline Balancing ». In : *Proceedings 28th Annual International Symposium on Computer Architecture*. 2001, p. 218–229. DOI : 10.1109/ISCA.2001.937451.

[8] Rajeev BALASUBRAMONIAN et al. « CACTI 7 : New Tools for Interconnect Exploration in Innovative Off-Chip Memories ». In : *ACM Trans. Archit. Code Optim.* 14.2 (juin 2017), 14 :1–14 :25. ISSN : 1544-3566. DOI : 10.1145/3085572.

[9] Rajeev BALASUBRAMONIAN et al. « Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures ». In : *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 33. New York, NY, USA : ACM, 2000, p. 245–257. ISBN : 978-1-58113-196-3. DOI : 10.1145/360128.360153.

[10] Leonid BARAZ et al. « IA-32 Execution Layer : A Two-Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-Based Systems ». In : *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 36. Washington, DC, USA : IEEE Computer Society, 2003, p. 191–. ISBN : 978-0-7695-2043-8.

[11] Fabrice BELLARD. « QEMU, a Fast and Portable Dynamic Translator ». In : *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Berkeley, CA, USA : USENIX Association, 2005, p. 41.

[12] D. BOGGS et al. « Denver : Nvidia's First 64-Bit ARM Processor ». In : *IEEE Micro*. T. 35. Mar. 2015, p. 46–55. DOI : 10.1109/MM.2015.12.

[13] E. BORIN et Y. WU. « Characterization of DBT Overhead ». In : *2009 IEEE International Symposium on Workload Characterization (IISWC)*. Oct. 2009, p. 178–187. DOI : 10.1109/IISWC.2009.5306785.

[14] M. BRANDALERO et A. C. S. BECK. « A Mechanism for Energy-Efficient Reuse of Decoding and Scheduling of X86 Instruction Streams ». In : *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. Mar. 2017, p. 1468–1473. DOI : 10.23919/DATE.2017.7927223.

[15] Anthony BRANDON et Stephan WONG. « Support for Dynamic Issue Width in VLIW Processors Using Generic Binaries ». In : *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium, 2013, p. 827–832.

[16] Anthony BRANDON et al. « Exploring ILP and TLP on a Polymorphic VLIW Processor ». In : *30th International Conference on Architecture of Computing Systems*. Avr. 2017.

[17] D. BRUENING, T. GARNETT et S. AMARASINGHE. « An Infrastructure for Adaptive Dynamic Optimization ». In : *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* Mar. 2003, p. 265–275. DOI : 10.1109/CGO.2003.1191551.

[18] Juan Manuel Martinez CAAMANO et al. « APOLLO : Automatic Speculative POLyhedral Loop Optimizer ». en. In : *IMPACT 2017 - 7th International Workshop on Polyhedral Compilation Techniques*. Jan. 2017, p. 8.

[19] David CALLAHAN et Brian KOBLENZ. « Register Allocation via Hierarchical Graph Coloring ». In : *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI '91. New York, NY, USA : ACM, 1991, p. 192–203. ISBN : 0-89791-428-7. DOI : 10.1145/113445.113462.

[20] Alexandre CARBON, Yves LHUILLIER et Henri-Pierre CHARLES. « Hardware Acceleration for Just-In-Time Compilation on Heterogeneous Embedded Systems ». In : *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference On*. IEEE, 2013, p. 203–210.

[21] Christopher CELIO, David A. PATTERSON et Krste ASANOVIC. *The Berkeley Out-of-Order Machine (BOOM) : An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor*. Rapp. tech. UCB/EECS-2015-167. EECS Department, University of California, Berkeley, juin 2015.

[22] Gregory J. CHAITIN. « Register Allocation & Spilling via Graph Coloring ». In : *ACM Sigplan Notices*. T. 17. ACM, 1982, p. 98–105.

[23] Henri-Pierre CHARLES et al. « deGoal a Tool to Embed Dynamic Code Generators into Applications ». en. In : *Compiler Construction*. Sous la dir. d'Albert COHEN. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, p. 107–112. ISBN : 978-3-642-54807-9.

[24] *Chrome V8 - https ://developers.google.com/v8/*. en. https ://developers.google.com/v8/.

[25] James C. DEHNERT et al. « The Transmeta Code Morphing™ Software : Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges ». In : *Proceedings of the International Symposium on Code Generation and Optimization : Feedback-Directed and Runtime Optimization*. IEEE Computer Society, 2003, p. 15–24.

[26] Benoit Dupont de DINECHIN. « Inter-Block Scoreboard Scheduling in a JIT Compiler for VLIW Processors ». en. In : *Euro-Par 2008 – Parallel Processing*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, août 2008, p. 370–381. DOI : 10.1007/978-3-540-85451-7_40.

[27] D. R. DITZEL. « Reflections on the High-Level Language Symbol Computer System ». In : 14.7 (juil. 1981), p. 55–66. ISSN : 0018-9162. DOI : 10.1109/C-M.1981.220530.

[28] David DITZEL. *Experience with Dynamic Binary Translation*. Keynote presentation at ISCA AMAS-BT. Keynote at ISCA AMAS-BT, 2008.

[29] James DUNDAS et Trevor MUDGE. « Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss ». In : *Proceedings of the 11th International Conference on Supercomputing*. ICS '97. New York, NY, USA : ACM, 1997, p. 68–75. ISBN : 978-0-89791-902-9. DOI : 10.1145/263580.263597.

[30] Kemal EBCIOĞLU et Erik R. ALTMAN. « DAISY : Dynamic Compilation for 100% Architectural Compatibility ». In : *Proceedings of the 24th Annual International Symposium on Computer Architecture*. ISCA '97. New York, NY, USA : ACM, 1997, p. 26–37. ISBN : 978-0-89791-901-2. DOI : 10.1145/264107.264126.

[31] Kemal EBCIOGLU et al. « Dynamic Binary Translation and Optimization ». In : *IEEE Trans. Comput.* T. 50. Juin 2001, p. 529–548. DOI : 10.1109/12.931892.

[32] P. FARABOSCHI et al. « Lx : A Technology Platform for Customizable VLIW Embedded Processing ». In : *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*. Juin 2000, p. 203–213. DOI : 10.1109/ISCA.2000.854391.

[33] J. A. FISHER. « Trace Scheduling : A Technique for Global Microcode Compaction ». In : *IEEE Trans. Comput.* 30.7 (juil. 1981), p. 478–490. ISSN : 0018-9340. DOI : 10.1109/TC.1981.1675827.

[34] Joseph A. FISHER, Paolo FARABOSCHI et Cliff YOUNG. *Embedded Computing : A VLIW Approach to Architecture, Compilers and Tools*. en. Elsevier, jan. 2005. ISBN : 978-0-08-047754-1.

[35] Philippe FLATRESSE, Giorgio CESANA et Xavier CAUCHY. *Planar Fully Depleted Silicon Technology to Design Competitive SOC at 28nm and Beyond*. Fév. 2012.

[36] Andreas GAL et al. « Trace-Based Just-in-Time Type Specialization for Dynamic Languages ». In : *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. New York, NY, USA : ACM, 2009, p. 465–478. ISBN : 978-1-60558-392-1. DOI : 10.1145/1542476.1542528.

[37] David M. GALLAGHER et al. « Dynamic Memory Disambiguation Using the Memory Conflict Buffer ». In : *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VI. New York, NY, USA : ACM, 1994, p. 183–193. ISBN : 978-0-89791-660-8. DOI : 10.1145/195473.195534.

[38] J. S. P. GIRALDO et al. « Leveraging Compiler Support on VLIW Processors for Efficient Power Gating ». In : *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. Juil. 2016, p. 502–507. DOI : 10.1109/ISVLSI.2016.70.

[39] James R. GOODMAN et W.-C. HSU. « Code Scheduling and Register Allocation in Large Basic Blocks ». In : *Proceedings of the 2nd International Conference on Supercomputing*. ACM, 1988, p. 442–452.

[40] John John GOUGH et K. John GOUGH. *Compiling for the .Net Common Language Runtime*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 2001. ISBN : 978-0-13-062296-9.

[41] Peter GREENHALGH. « Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7 ». In : *ARM White Paper* (2011).

[42] Q. GUO et al. « Run-Time Phase Prediction for a Reconfigurable VLIW Processor ». In : *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2016, p. 1634–1639.

[43] John L. HENNESSY et David A. PATTERSON. *Computer Architecture, Fifth Edition : A Quantitative Approach*. 5th. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2011. ISBN : 978-0-12-383872-8.

[44] K. HEYDEMANN et al. « UFS : A Global Trade-off Strategy for Loop Unrolling for VLIW Architectures ». en. In : *Concurrency and Computation : Practice and Experience* 18.11 (sept. 2006), p. 1413–1434. ISSN : 1532-0634. DOI : 10.1002/cpe.1014.

[45] A. HILTON, S. NAGARAKATTE et A. ROTH. « iCFP : Tolerating All-Level Cache Misses in in-Order Processors ». In : *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. Fév. 2009, p. 431–442. DOI : 10.1109/HPCA.2009.4798281.

[46] Jason D. HISER et al. « Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems ». In : *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '07. Washington, DC, USA : IEEE Computer Society, 2007, p. 61–73. ISBN : 978-0-7695-2764-2. DOI : 10.1109/CGO.2007.10.

[47] Ding-Yong HONG et al. « HQEMU : A Multi-Threaded and Retargetable Dynamic Binary Translator on Multicores ». In : *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. CGO '12. New York, NY, USA : ACM, 2012, p. 104–113. ISBN : 978-1-4503-1206-6. DOI : 10.1145/2259016.2259030.

[48] Shiliang HU et James E. SMITH. « Reducing Startup Time in Co-Designed Virtual Machines ». In : *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. ISCA '06. Washington, DC, USA : IEEE Computer Society, 2006, p. 277–288. ISBN : 978-0-7695-2608-9. DOI : 10.1109/ISCA.2006.33.

[49] Wen-Mei W. HWU et al. « The Superblock : An Effective Technique for VLIW and Superscalar Compilation ». en. In : *Instruction-Level Parallelism : A Special Issue of The Journal of Supercomputing*. Sous la dir. de B. R. RAU et J. A. FISHER. The Springer International Series in Engineering and Computer Science. Boston, MA : Springer US, 1993, p. 229–248. ISBN : 978-1-4615-3200-2. DOI : 10.1007/978-1-4615-3200-2_7.

[50] Alexandra JIMBOREAN et al. « Dynamic and Speculative Polyhedral Parallelization Using Compiler-Generated Skeletons ». en. In : *International Symposium on High-Level Parallel Programming and Applications, HLPP*. Juil. 2013.

[51] Thomas KISTLER et Michael FRANZ. « Continuous Program Optimization : A Case Study ». In : *ACM Trans. Program. Lang. Syst.* 25.4 (juil. 2003), p. 500–548. ISSN : 0164-0925. DOI : 10.1145/778559.778562.

[52] Alexander KLAIBER. *The Technology Behind Crusoe Processors*. Rapp. tech. Jan. 2000.

[53] Paul KOCHER et al. « Spectre Attacks : Exploiting Speculative Execution ». In : *arXiv :1801.01203 [cs]* (jan. 2018). arXiv : 1801.01203 [cs].

[54] Thomas KOTZMANN et al. « Design of the Java HotSpot™ Client Compiler for Java 6 ». In : *ACM Trans. Archit. Code Optim.* 5.1 (mai 2008), 7 :1–7 :32. ISSN : 1544-3566. DOI : 10.1145/1369396.1370017.

[55] R. KUMAR et al. « Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance ». In : *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* Juin 2004, p. 64–75. DOI : 10.1109/ISCA.2004.1310764.

[56] R. KUMAR et al. « Single-ISA Heterogeneous Multi-Core Architectures : The Potential for Processor Power Reduction ». In : *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* Déc. 2003, p. 81–92. DOI : 10.1109/MICRO.2003.1253185.

[57] T. A. LALIOTIS. « Implementation Aspects of the Symbol Hardware Compiler ». In : *Proceedings of the 1st Annual Symposium on Computer Architecture*. ISCA '73. New York, NY, USA : ACM, 1973, p. 111–115. DOI : 10.1145/800123.803976.

[58] M. LAM. « Software Pipelining : An Effective Scheduling Technique for VLIW Machines ». In : *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. New York, NY, USA : ACM, 1988, p. 318–328. ISBN : 978-0-89791-269-3. DOI : 10.1145/53990.54022.

[59] Etienne LE SUEUR et Gernot HEISER. « Dynamic Voltage and Frequency Scaling : The Laws of Diminishing Returns ». In : *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*. HotPower'10. Berkeley, CA, USA : USENIX Association, 2010, p. 1–8.

[60] Scott A. MAHLKE et al. « Effective Compiler Support for Predicated Execution Using the Hyperblock ». In : *Proceedings of the 25th Annual International Symposium on Microarchitecture*. MICRO 25. Los Alamitos, CA, USA : IEEE Computer Society Press, 1992, p. 45–54. ISBN : 978-0-8186-3175-7.

[61] H. T. MAIR et al. « A 20nm 2.5GHz Ultra-Low-Power Tri-Cluster CPU Subsystem with Adaptive Power Allocation for Optimal Mobile SoC Performance ». In : *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. Jan. 2016, p. 76–77. DOI : 10.1109/ISSCC.2016.7417914.

[62] Daniel S. MCFARLIN, Charles TUCKER et Craig ZILLES. « Discerning the Dominant Out-of-Order Performance Advantage : Is It Speculation or Dynamism ? » In : *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. New York, NY, USA : ACM, 2013, p. 241–252. ISBN : 978-1-4503-1870-9. DOI : 10.1145/2451116.2451143.

[63] Luc MICHEL, Nicolas FOURNEL et Frederic PETROT. « Fast Simulation of Systems Embedding VLIW Processors ». In : *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '12. New York, NY, USA : ACM, 2012, p. 143–150. ISBN : 978-1-4503-1426-8. DOI : 10.1145/2380445.2380472.

[64] S. NEKKALAPU et al. « A Simple Latency Tolerant Processor ». In : *2008 IEEE International Conference on Computer Design*. Oct. 2008, p. 384–389. DOI : 10.1109/ICCD.2008.4751889.

[65] Dorit NUZMAN et al. « Vapor SIMD : Auto-Vectorize Once, Run Everywhere ». In : *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2011, p. 151–160.

[66] Shruti PADMANABHA et al. « DynaMOS : Dynamic Schedule Migration for Heterogeneous Cores ». In : *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. New York, NY, USA : ACM, 2015, p. 322–333. ISBN : 978-1-4503-4034-2. DOI : 10.1145/2830772.2830791.

[67] Shruti PADMANABHA et al. « Mirage Cores : The Illusion of Many Out-of-Order Cores Using In-Order Hardware ». In : *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-50 '17. New York, NY, USA : ACM, 2017, p. 745–758. ISBN : 978-1-4503-4952-9. DOI : 10.1145/3123939.3123969.

[68] Hyunchul PARK et al. « Edge-Centric modulo Scheduling for Coarse-Grained Reconfigurable Architectures ». In : *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2008, p. 166–176.

[69] Il PARK, Chong Liang OOI et T. N. VIJAYKUMAR. « Reducing Design Complexity of the Load/Store Queue ». In : *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 36. Washington, DC, USA : IEEE Computer Society, 2003, p. 411–. ISBN : 978-0-7695-2043-8.

[70] Sanjay J. PATEL et al. « Increasing the Size of Atomic Instruction Blocks Using Control Flow Assertions ». In : *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 33. New York, NY, USA : ACM, 2000, p. 303–313. ISBN : 978-1-58113-196-3. DOI : 10.1145/360128.360160.

[71] Shlomit S. PINTER. « Register Allocation with Instruction Scheduling ». In : *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. PLDI '93. New York, NY, USA : ACM, 1993, p. 248–257. ISBN : 0-89791-598-4. DOI : 10.1145/155090.155114.

[72] Massimiliano POLETTO et Vivek SARKAR. « Linear Scan Register Allocation ». In : *ACM Trans. Program. Lang. Syst.* 21.5 (sept. 1999), p. 895–913. ISSN : 0164-0925. DOI : 10.1145/330249.330250.

[73] Dmitry PONOMAREV, Gurhan KUCUK et Kanad GHOSE. « Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources ». In : *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 34. Washington, DC, USA : IEEE Computer Society, 2001, p. 90–101. ISBN : 978-0-7695-1369-0.

[74] Krishna K. RANGAN, Gu-Yeon WEI et David BROOKS. « Thread Motion : Fine-Grained Power Management for Multi-Core Systems ». In : *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA '09. New York, NY, USA : ACM, 2009, p. 302–313. ISBN : 978-1-60558-526-0. DOI : 10.1145/1555754.1555793.

[75] B. Ramakrishna RAU. « Iterative Modulo Scheduling : An Algorithm for Software Pipelining Loops ». In : *Proceedings of the 27th Annual International Symposium on Microarchitecture*. MICRO 27. New York, NY, USA : ACM, 1994, p. 63–74. ISBN : 978-0-89791-707-0. DOI : 10.1145/192724.192731.

[76] Siddharth RELE et al. « Optimizing Static Power Dissipation by Functional Units in Superscalar Processors ». In : *Proceedings of the 11th International Conference on Compiler Construction*. CC '02. London, UK, UK : Springer-Verlag, 2002, p. 261–275. ISBN : 978-3-540-43369-9.

[77] Amir ROTH. « Store Vulnerability Window (SVW) : Re-Execution Filtering for Enhanced Load Optimization ». In : *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*. ISCA '05. Washington, DC, USA : IEEE Computer Society, 2005, p. 458–468. ISBN : 978-0-7695-2270-8. DOI : 10.1109/ISCA.2005.48.

[78] S. ROY, N. RANGANATHAN et S. KATKOORI. « A Framework for Power-Gating Functional Units in Embedded Microprocessors ». In : *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17.11 (nov. 2009), p. 1640–1649. ISSN : 1063-8210. DOI : 10.1109/TVLSI.2008.2005774.

[79] S. ROY, N. RANGANATHAN et S. KATKOORI. « Exploring Compiler Optimizations for Enhancing Power Gating ». In : *2009 IEEE International Symposium on Circuits and Systems*. Mai 2009, p. 1004–1007. DOI : 10.1109/ISCAS.2009.5117928.

[80] Simha SETHUMADHAVAN et al. « Scalable Hardware Memory Disambiguation for High ILP Processors ». In : *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 36. Washington, DC, USA : IEEE Computer Society, 2003, p. 399–. ISBN : 978-0-7695-2043-8.

[81] Andre SEZNEC. « A New Case for the TAGE Branch Predictor ». In : *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44. New York, NY, USA : ACM, 2011, p. 117–127. ISBN : 978-1-4503-1053-6. DOI : 10.1145/2155620.2155635.

[82] H. SHARANGPANI et H. ARORA. « Itanium Processor Microarchitecture ». In : *IEEE Micro*. T. 20. Sept. 2000, p. 24–43. DOI : 10.1109/40.877948.

[83] Gabriel M. SILBERMAN et Kemal EBCIOGLU. « An Architectural Framework for Supporting Heterogeneous Instruction-Set Architectures ». In : t. 26. Juin 1993, p. 39–56. DOI : 10.1109/2.214441.

[84] Samantika SUBRAMANIAM et Gabriel H. LOH. « Fire-and-Forget : Load/Store Scheduling with No Store Queue at All ». In : *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 39. Washington, DC, USA : IEEE Computer Society, 2006, p. 273–284. ISBN : 978-0-7695-2732-1. DOI : 10.1109/MICRO.2006.26.

[85] H. TABKHI et G. SCHIRNER. « Application-Guided Power Gating Reducing Register File Static Power ». In : *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22.12 (déc. 2014), p. 2513–2526. ISSN : 1063-8210. DOI : 10.1109/TVLSI.2013.2293702.

[86] R. M. TOMASULO. « An Efficient Algorithm for Exploiting Multiple Arithmetic Units ». In : *IBM Journal of Research and Development* 11.1 (jan. 1967), p. 25–33. ISSN : 0018-8646. DOI : 10.1147/rd.111.0025.

[87] Carlos VILLAVIEJA, Jose A. JOAO et Rustam MIFTAKHUTDINOV. « Yoga : A Hybrid Dynamic VLIW/OoO Processor ». In : *HPS Technical Report*. 2014.

[88] Cheng WANG, Youfeng WU et M CINTRA. « Acceldroid : Co-Designed Acceleration of Android Bytecode ». In : *CGO'13* (2013).

[89] Cheng WANG et al. « StarDBT : An Efficient Multi-Platform Dynamic Binary Translation System ». en. In : *Advances in Computer Systems Architecture*. Springer, Berlin, Heidelberg, août 2007, p. 4–15. DOI : 10.1007/978-3-540-74309-5_3.

[90] M. A. WATKINS, T. NOWATZKI et A. CARNO. « Software Transparent Dynamic Binary Translation for Coarse-Grain Reconfigurable Architectures ». In : *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Mar. 2016, p. 138–150. DOI : 10.1109/HPCA.2016.7446060.

[91] Y. WU et al. « A HW/SW Co-Designed Heterogeneous Multi-Core Virtual Machine for Energy-Efficient General Purpose Computing ». In : *2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Avr. 2011, p. 236–245. DOI : 10.1109/CGO.2011.5764691.

[92] M. T. YOURST et K. GHOSE. « Incremental Commit Groups for Non-Atomic Trace Processing ». In : *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. Nov. 2005. DOI : 10.1109/MICRO.2005.23.

Annexes

ANNEXE A

PROTOCOLES EXPÉRIMENTAUX

Plusieurs études expérimentales ont été réalisées dans ce document pour mesurer l'efficacité de notre approche. Un objectif de ces travaux était de mettre en place un outil de recherche *open-source* pour la traduction dynamique de binaires, la reproductibilité des résultats est un point important.

Cependant, de nombreuses expériences demandent des outils qui nécessitent une licence (Catapult HLS, Design Compiler, Modelsim). Ainsi, les études expérimentales qui mesurent la surface, fréquence et consommation d'une architecture ne seront pas facilement reproductibles.

Cependant, toutes les autres expériences, qui visent à mesurer la performance de notre système et à les comparer à d'autres types de processeurs, sont scriptées et peuvent être reproduites facilement. Un dépot Git [1] a été mis en place avec le code source des différentes applications utilisées comme benchmark, des consignes détaillées pour installer les différents outils, et des scripts pour lancer les experiences et réunir les résultats. Dans la suite de cette annexe, nous tâchons de décrire de façon détaillée les protocoles expérimentaux utilisés pour les différentes expériences menées dans ce document.

Experience	Section	Reproductibilité	Outils nécessaires
Performance des accélérateurs	3.4.1	✗	Flot ASIC et Catapult
Performance de Hybrid-DBT	3.4.2	✓	Gem5 et Hybrid-DBT
Intérêt de l'accélération	3.4.3	✓	Hybrid-DBT
Coût en surface des accélérateurs	3.4.4	✗	Flot ASIC et Catapult
Performance de la reconfiguration dynamique	4.1.5	✓	Hybrid-DBT
Utilisation des configurations	4.1.5	✓	Hybrid-DBT
Etude de la reconfiguration dynamique	4.1.5	✓	Hybrid-DBT
Performance de la spéculation mémoire	4.2.6	✓	Hybrid-DBT
Coût en surface de la spéculation	4.2.6	✗	Flot ASIC et Catapult

TABLE A.1 – Liste des études expérimentales menées pour valider les résultats présentés dans ce document, classées selon leur ordre d'apparition. Les expériences marquées comme non reproductibles sont celles qui ne peuvent pas être reproduite sans licence payante.

Protocoles expérimentaux

Pour toutes les expériences impliquant une exécution, les différentes applications de Polybench et de Mediabench ont été compilées pour le jeu d'instructions RISC-V 64 bits, avec le niveau d'optimisation

1. https ://github.com/srokicki/HybridDBT-benchmarks

O3. Ces applications sont également *linkés* statiquement avec la Newlib. Les binaires obtenus sont utilisés pour toutes les exécutions. La suite d'applications Polybench donne accès à des paramètres permettant de contrôler le temps d'exécution des applications. Celles-ci ont été calibrées pour prendre entre 25 et 600 millions de cycles d'exécution. Les temps d'exécution des applications de Mediabench sont distribués entre 4 et 400 millions de cycles. Dans la suite cette section, nous listons les protocoles expérimentaux utilisés dans chacune des expériences reproductibles, listées dans le tableau A.1.

Performance de Hybrid-DBT

Pour cette expérience, les applications ont été exécutées avec trois simulateurs différents :

— Le scénario d'exécution sur un processeur OoO utilise la version RISC-V du simulateur GEM5. Le modèle de processseur utilisé est disponible sur le dépot Git et est inspiré du modèle de processeur *big* fourni dans GEM5.
— Le scénario d'exécution sur un processeur à exécution dans l'ordre utilise un simulateur de processeur pipeliné disponible sur le dépot de Hybrid-DBT.
— Les trois scénarios basés sur Hybrid-DBT utilisent notre similateur de VLIW.

Ces trois outils simulent une hiérarchie de cache composée d'un L1 de 16 ko et d'un L2 de 256 ko. Les simulateurs pour le processeur *in-order* et pour le processeur VLIW utilisent exactement la même modélisation du cache, tandis que la simulation du GEM5 est différente. Cette différence impacte l'exécution de certaines applications.

Les trois scénarios d'exécutions basés sur Hybrid-DBT utilisent des niveaux d'optimisation différents :

— Le niveau O0 n'utilise que le `First-pass Translator` pour réaliser la traduction naïve des instructions RISC-V.
— Le niveau O1 utilise également le `IR Generator` et le `IR Scheduler` pour ordonnancer les instructions à l'échelle du bloc de base.
— Le niveau d'optimisation O2 applique également les optimisations à l'échelle des procédures : la construction de superblocs, le déroulage de certaines boucles et une réallocation des registres globaux. Pour cette expérience, la spéculation de dépendances mémoire est également activée.

Pour estimer l'énergie dissipée lors de ces exécutions, nous avons synthétisé le processeur Rocket, le processeur Boom et notre processeur VLIW pour une technologie 28 nm *Low Power, High Threshold* de STMicroelectronics. Pour notre processeur VLIW, nous avons ensuite réalisé une simulation précise au niveau cycle afin d'obtenir le taux de changement d'états de chaque porte logique du processeur. Avec cette information, DesignCompiler est capable d'estimer la puissance dissipée de manière très précise. Pour les processeurs Rocket et Boom, nous avons utilisé les fichiers VCD générés par Verilator pour dériver le taux de changement d'états des ports de chaque composant. Design Compiler est ensuite capable de propager cette information pour estimer la puissance consommée, mais cette estimation est moins précise que celle basée sur la simulation *gate-level*. Les valeurs obtenues lors de cette étude sont résumées dans le tableau A.2.

	Rocket	VLIW	Boom
Tension	1,0 V		
Fréquence	700 MHz		
Surface (μm^2)	30 851	106 621	598 116
Consommation (mW)	10,3	56,7	203,8

TABLE A.2 – Valeurs utilisées pour l'estimation de la consommation d'énergie des différentes exécutions.

Intérêt de l'accélération matérielle

Dans cette expérience, nous avons mis en avant l'impact des accélérateurs matériels sur la performance des exécutions. Dans le simulateur de Hybrid-DBT, le coût des optimisations est également simulé : le simulateur applique les transformations de manière instantanée, et modélise le temps d'optimisation t_opt à partir du temps de traitement moyen d'une instruction. Le simulateur exécute ensuite t_opt cycles sur le processeur VLIW sans appliquer d'autres optimisations. L'énergie dépensée pour l'optimisation est également estimée à partir de la puissance dissipée par les accélérateurs et par le processeur DBT.

Pour cette étude expérimentale, Hybrid-DBT a été configuré à son plus haut niveau d'optimisation. La spéculation de dépendances mémoire ainsi que le support pour les VLIW dynamiquement reconfigurables sont activés (ce dernier flot utilise un coefficient *energy_ratio* de 100%, privilegiant ainsi l'efficacité énergétique).

Les applications ont été exécutées deux fois, en utilisant des modélisations différentes du temps et de l'énergie consacrés à l'optimisation.

Performance de la reconfiguration dynamique et utilisation des différentes configurations

Cette étude expérimentale utilise uniquement Hybrid-DBT. Chaque application est exécutée six fois :

— Quatre exécutions utilisent des configurations de VLIW fixées à deux, quatre, six ou huit voies d'exécution.

— Les deux autres exécutions ont été réalisées en utilisant la gestion de la reconfiguration dynamique, avec des coefficients de 10% et de 90%.

Pour cette étude expérimentale, toutes les optimisations de Hybrid-DBT ont été activées (y compris la spéculation de dépendances mémoire). Les exécutions utilisant le support pour les VLIW dynamiquement reconfigurables ont également permis de compter le nombre d'utilisations de chaque configuration du processeur VLIW. En effet, pour chaque procédure identifiée, Hybrid-DBT construit le front de Pareto et renvoie le nombre d'apparitions de chaque configuration.

Performance de la spéculation de dépendances mémoire

La dernière étude expérimentale mesure la gain en performance lié à l'utilisation de la spéculation de dépendances mémoire. Pour cette expérience, les différentes applications ont été exécutées avec et sans le flot de spéculation de dépendances mémoire, sur deux configurations différentes du processeur

VLIW (une à 4 voies dont une qui permet d'accéder à la mémoire, l'autre à 6 voies dont deux permettent d'accéder à la mémoire).

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Accélération matérielle pour la traduction dynamique de programmes binaires

Mot clés : Systèmes embarqués, Traduction Dynamique de Binaires, VLIW, Accélération matérielle

Resumé : Cette thèse porte sur l'utilisation de techniques d'accélération matérielle pour la conception de processeurs basés sur l'optimisation dynamique de binaires. Dans ce type de machine, les instructions du programme exécuté par le processeur sont traduites et optimisées à la volée par un outil de compilation dynamique intégré au processeur. Ce procédé permet de mieux exploiter les ressources du processeur cible, mais est délicate à exploiter car le temps de cette recompilation impacte de manière très significative l'effet global de ces optimisations.

Dans cette thèse, nous montrons que l'utilisation d'accélérateurs matériels pour certaines étapes clés de cette compilation (construction de la représentation intermédiaire, ordonnancement des instructions), permet de ramener le temps de compilation à des valeurs très faibles (en moyenne 6 cycles par instruction, contre plusieurs centaines dans le cas d'une mise en œuvre classique). Nous avons également montré comment ces techniques peuvent être exploitées pour offrir de meilleurs compromis performance/consommation sur certains types de noyaux de calculs. La thèse à également débouché sur la mise à disposition de la communauté de recherche du compilateur développé.

Title : Hardware acceleration of Dynamic Binary Translation

Keywords : Embedded Systems, Dynamic Binary Translation, VLIW, Hardware/Software Co-Design

Abstract : This thesis is focused on the hardware acceleration of processors based on Dynamic Binary Translation. Such architectures execute binaries by translating and optimizing each instruction at run-time, thanks to a DBT toolchain embedded in the system. This process leads to a better ressource utilization but also induces execution time overheads, which affect the overall performances.

During this thesis, we've shown that the use of hardware components to accelerate critical parts of the DBT process (First translation, generation of an intermediate representation and instruction scheduling) drastically reduce the compilation time (around 6 cycles to schedule one instruction, against several hundreds for a fully-software DBT). We've also demonstrated that the proposed approach enables several continuous optimizations flow, which offers better energy/performance tradeoffs. Finally, the DBT toolchain is open-source and available online.