N° d'ordre: 3092

THÈSE

présentée

devant l'université de Rennes 1

pour obtenir

le grade de : Docteur de l'université de Rennes 1
Mention Informatique

par

Olivier ROCHECOUSTE

Équipe d'accueil : CAPS
École doctorale : MATISSE
Composante universitaire : IFSIC

Titre de la thèse :

Architecture et bits significatifs

soutenue le 24 Octobre 2005 devant la commission d'examen

M. :	Isabelle	PUAUT	Président
MM. :	Olivier	TEMAM	Rapporteurs
	Daniel	LITAIZE	
MM. :	Pascal	SAINRAT	Examinateurs
	André	SEZNEC	

If you steal from one author it's plagiarism
If you steal from many it's research
Wilson Mizner

Remerciements

Cette thèse n'aurait, assurément, pas pu s'accomplir sans le concours et le support de nombreuses personnes. Quelques lignes ne suffiront, sans doute, pas pour leur exprimer toute ma gratitude. Je tiens, malgré cela, à adresser mes remerciements et mon amitié à celles et ceux m'ayant soutenu et accompagné tout au long de cette expérience.

Mes premiers remerciements doivent être adressés à *André Seznec* pour avoir dirigé mes travaux de recherche et m'avoir accordé le privilège de travailler à ses côtés. Je souhaite notamment saluer sa disponibilité, sa patience, et ses qualités scientifiques qui ont rendu possible l'élaboration des techniques défendues à travers ce document.

En second lieu, je tiens à exprimer ma plus profonde reconnaissance à *Gilles Pokam* pour son étroite collaboration aux études présentées dans ce document. Je remercie Gilles autant pour son support scientifique que pour son soutien moral qui ont été primordiaux jusqu'au terme de cette thèse.

Je voudrais également témoigner ma gratitude aux membres du jury présidé par *Isabelle Puaut* (Professeur à l'Université de Rennes I) et composé de : *Olivier Temam* (Directeur de recherche INRIA Futurs, Rapporteur), *Daniel Litaize* (Professeur à l'Université Toulouse III, Rapporteur), et *Pascal Sainrat* (Professeur à l'Université Toulouse III, Examinateur). Olivier Temam et Daniel Litaize méritent des remerciements plus particuliers pour s'être consacrés à la relecture de ce document.

Une mention spéciale doit être attribuée à celles et ceux m'ayant aidé à surmonter les difficultés associées à la thèse et ayant également contribué à l'élaboration de son pot. Encore merci à Philippe, Valérie, Séverine, Tifenn et Eric.

Enfin, mes remerciements les plus sincères vont à mes parents et à ma sœur qui m'ont accordé un soutien permanent et indispensable tout au long de ces dernières années.

Olivier.

Table des matières

Table des figures

Liste des tableaux

Introduction

La puissance de calcul des processeurs a progressé à un rythme quasi-exponentiel au cours de ces dernières décennies. Cette progression résulte à la fois des améliorations réalisées en *technologie*, *microarchitecture* et *compilation* [1, 61]. L'amélioration des technologies de fabrication a en effet permis de doubler les densités d'intégration et d'élever les fréquences de cadencement à chaque nouvelle génération [60]. Ce complément de ressources physiques a pu alors contribuer au déploiement de nouveaux mécanismes microarchitecturaux visant essentiellement à augmenter le parallélisme d'instructions ou alors à optimiser les accès à la mémoire. Ces divers mécanismes ont alors suscité le développement de techniques de compilation sophistiquées permettant un usage des ressources plus optimal.

Bien que cette stratégie ait longtemps permis la réalisation de processeurs plus rapides, il devient toutefois de plus en plus difficile de l'appliquer à la conception de processeurs modernes. En effet, en contrepartie de meilleures performances, les processeurs sont également devenus de plus en plus complexes. L'exécution dans le désordre [65], l'intégration de plusieurs niveaux de caches, la prédiction de branchements [84] ou le modèle multi-flot simultané [88] témoignent notamment de cette tendance relative à l'augmentation de la complexité du processeur. La complexité d'un composant influe directement sur différents paramètres critiques dont sa *consommation d'énergie*, sa *dissipation de puissance*, son *temps d'accès* et sa *surface de silicium*.

La littérature regroupe plusieurs solutions pour maîtriser un ou plusieurs de ces paramètres dans les processeurs embarqués et à usage général [87]. Minimiser la consommation d'énergie est plus particulier aux systèmes embarqués fonctionnant sur batteries alors que maîtriser la complexité matérielle est fondamental aux processeurs superscalaires. La consommation d'énergie détermine la durée de fonctionnement du système et limite ainsi le nombre de fonctionnalités intégrables. Augmenter la complexité matérielle impacte directement sur le délai de traversée des composants ; ce qui peut alors limiter la fréquence des structures placées dans le chemin critique du processeur. Les autres problèmes relatifs à l'augmentation de la complexité sont une surface de silicium et une dissipation de puissance plus élevées pouvant circonscrire le passage à une nouvelle génération. Tous ces problèmes sous-jacents à la complexité contraignent déjà la conception des processeurs contemporains [65, 87] et sont susceptibles de limiter également celle des prochaines générations.

Contributions

Cette thèse propose deux techniques pour maîtriser la complexité dans les systèmes embarqués et superscalaires modernes. Ces techniques découlent du même constat établi par Brooks et Martonosi [12]. Ces auteurs ont observé que la plupart des applications traitent fréquemment des données comportant un nombre restreint de bits significatifs (c-à-d. inférieur à 16 bits). Les processeurs modernes architecturent toutefois des chemins de données de plus en plus larges (c-à-d. 64-bit) pour satisfaire les besoins en adressage. Ce surdimensionnement impacte directement sur la taille et la complexité des composants critiques déterminant la fréquence du système. Plusieurs propositions [17, 55, 85, 64, 17, 27] ont été avancées pour exploiter la propriété observée par Brooks. Ces propositions sont scindées entre celles [12, 17] exploitant cette propriété pour réduire la complexité du processeur et celles [64, 55] l'utilisant pour accélérer le temps de calcul. Les processeurs contemporains supportent notamment des instructions spécialisées, appelées SIMD[1], pour tirer profit de cette propriété des programmes et pouvoir ainsi accélérer le calcul des tâches multimédia [50]. Ces instructions permettent le traitement concurrent de plusieurs opérations homogènes sur des données de faible dimension ; d'où un usage plus optimal des ressources du chemin de données. Contrairement au contexte de notre étude, la finalité première de ces techniques ne consiste toutefois pas à minimiser la complexité du processeur [12, 71].

Proposition 1

La première proposition présentée dans ce document se rapporte à la réduction d'énergie pour les systèmes embarqués. Cette solution a été récemment publiée dans [71]. L'étude proposée examine un nouveau schéma logiciel/matériel pour réduire l'activité du chemin de données. Contrairement aux propositions antérieures, nous suggérons une approche coopérative logiciel/matériel plus efficace et mieux adaptée aux contraintes de conception des systèmes embarqués. Notre approche consiste à découper un programme en régions regroupant des opérations nécessitant une même largeur d'exécution. Les régions résultantes peuvent ainsi être exécutées avec une largeur plus optimale permettant ainsi de réduire la consommation d'énergie. Le découpage du programme est réalisé à travers une analyse du comportement de ses exécutions passées et est exposé au matériel grâce à une *instruction de reconfiguration*. Cette instruction provoque le redimensionnement du chemin de données à la largeur spécifiée. Notre schéma est toutefois de nature spéculative car nous nous appuyons sur le comportement passé du programme pour prédire la dimension du calcul. Ce modèle spéculatif nécessite donc l'assistance d'un mécanisme de recouvrement dédié. Nous proposons à cet effet un schéma de recouvrement simpliste mais requérant une faible complexité matérielle. Une contribution importante de notre étude est la proposition d'un nouveau modèle pour fichiers de registres : *le fichier de registres en tranches* [2]. Grâce à une modifica-

[1] Single-Instruction Multiple-Data
[2] *Byte-slice Register File*

tion de ses cellules mémoire, ce dernier peut s'adapter à la largeur des données tout en préservant le contenu de cette mémoire. L'évaluation expérimentale du schéma proposé révèle une réduction significative de la consommation statique et dynamique du chemin de données avec un impact relativement faible sur les performances.

Proposition 2

Notre première proposition emploie une instruction de reconfiguration pour indiquer au matériel de redimensionner le chemin de données. La solution proposée n'est toutefois pas directement applicable aux processeurs à exécution dynamique car les instructions de reconfiguration sont limitées à une exécution dans l'ordre. Notre seconde proposition [77, 76] explore une alternative envisageable pour exploiter la propriété observée par Brooks dans le cadre des processeurs superscalaires. Les processeurs superscalaires contemporains font usage de chemins de données 64-bit pour exécuter toutes les instructions composant une application. Nous avons observé qu'une large fraction de ces instructions manipulent uniquement des données de faible dimension tout en étant bien distribuées au long de l'exécution. Pour exploiter ces propriétés dynamiques, nous proposons une microarchitecture capable de découpler le traitement des instructions sur des *clusters* distincts. Nous étudions notamment une microarchitecture composée de deux *clusters* de largeurs distinctes : *un cluster 64-bit capable d'exécuter les opérations complexes et les opérations mémoire*, et *un cluster 16-bit optimisé pour le traitement des opérations manipulant des données de faible dimension*. De plus, l'échange de données entre ces clusters concerne exclusivement les données de faible dimension. Ce nouveau schéma de partitionnement permet de réduire significativement la complexité des composants critiques appartenant au chemin de données : *fichier de registres*, *réseau d'interconnexion* et *unités fonctionnelles*. Dans les microarchitectures clusterisées classiques, deux caractéristiques sont typiquement fondamentales pour les performances : le nombre de communications inter-cluster et l'équilibrage des charges de travail. En exploitant uniquement l'entrelacement dynamique des opérations 16-bit et 64-bit, nous montrons qu'il est possible de réaliser un équilibrage des charges correct sans intervention de mécanismes complexes. Les implémentations suggérées permettent également d'optimiser le nombre de communications inter-cluster pour réduire davantage la consommation des interconnexions. L'évaluation empirique de ce schéma indique des gains substantiels en complexité (consommation, surface de silicium et temps d'accès) avec des performances avoisinant celles des schémas conventionnels.

Organisation du document

Ce document s'articule autour de trois chapitres. Le chapitre 1 présente les diverses notions abordées dans nos deux propositions. Ce chapitre définit ainsi les sources de complexité auxquelles s'attaque cette thèse. Il détaille notamment les sources de consommation dans la technologie actuelle et les optimisations envisageables dérivées dans nos travaux. Ce chapitre détaille également la complexité de certains composants critiques ayant été ciblés par nos études. L'état de l'art des recherches relatives à nos propositions est aussi mis en avant dans ce chapitre. Les chapitres 2 et 3 décrivent les deux propositions émises dans cette thèse. Le chapitre 2 expose le schéma spéculatif proposé dans [71] pour accommoder la largeur du chemin de données à la dimension du calcul. Le chapitre 3 décrit un nouveau schéma de partitionnement pour découpler les opérations manipulant exclusivement des données de faible dimension sur des *clusters* appropriés.

Chapitre 1

État de l'art

1.1 Introduction

Les processeurs modernes supportent des techniques très agressives pour tirer profit du parallélisme d'instructions présent au cœur des applications. L'exécution superscalaire [65], la prédiction de branchements [84] ou l'approche multi-flot simultané [88] illustrent notamment cette tendance. Mais en contrepartie de gains en performance, l'utilisation de ces techniques a toutefois engendré une complexification significative de la structure des processeurs au fil des générations. Cette complexité influe négativement sur plusieurs paramètres dont la consommation d'énergie et la dissipation de puissance du processeur. Accentuer la complexité des composants localisés sur le chemin critique du processeur contribue également à rendre plus difficile la réalisation de générations performantes. Plusieurs travaux [87, 65] ont démontré le besoin de mettre au point de nouvelles techniques pour réduire ces effets indésirables associés à l'augmentation de la complexité.

Diverses solutions [87, 65] ont été exposées dans la littérature pour ce faire. Dans ce document, nous nous basons sur certaines propriétés [12] des applications pour réduire la consommation et/ou la complexité matérielle du chemin de données dans les processeurs modernes. Ces propriétés se rapportent à l'utilisation fréquente des données de faible dimension (c-à-d. inférieure à 16 bits) à l'exécution de la majorité des applications. Les processeurs modernes ont toutefois tendance à architecturer des chemins de données de plus en plus larges pour satisfaire la demande en espace d'adressage. Il en résulte ainsi une utilisation non optimale des ressources pouvant impacter à la fois sur la consommation et la complexité du matériel. Nous avons proposé deux techniques [71, 77] pour optimiser ces deux paramètres dans les architectures embarquées et hautes performances. La première approche permet d'optimiser la consommation d'énergie du pipeline en adaptant sa taille aux besoins du calcul. La seconde approche se base sur une architecture composée de plusieurs clusters d'unités fonctionnelles pour découpler le traitement des opérations selon la dimension des opérandes manipulés.

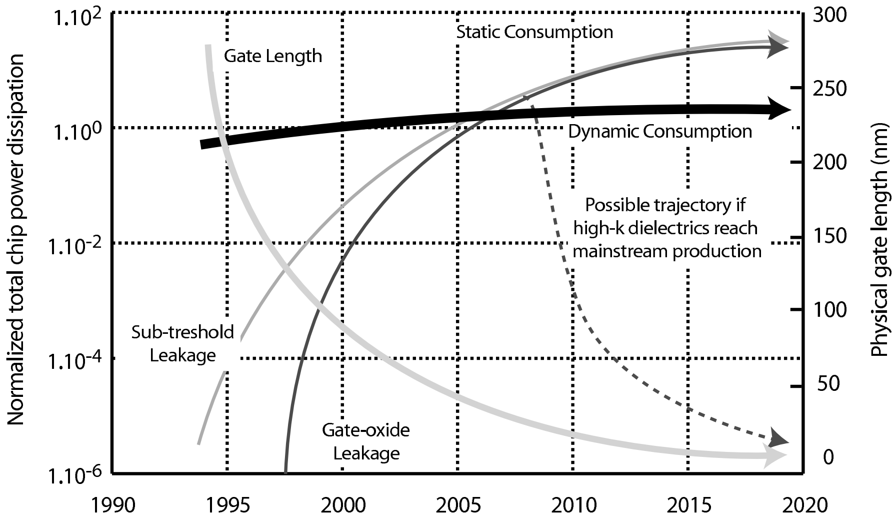


FIG. 1.1 – Projection de la consommation en fonction de la taille du motif [81].

1.1.1 Organisation du chapitre

Nous présentons dans ce chapitre les différentes notions et l'état de l'art des recherches relatives à nos propositions. La section 1.2 décrit l'origine de la consommation électrique dans la technologie de fabrication actuelle tout en exposant plusieurs solutions basiques dérivées dans ce document. La section 1.3 détaille ensuite les composants critiques du processeur sur lesquels nous nous sommes concentrés. Ces composants regroupent le fichier de registres, le réseau de *bypass* et les logiques associées à la fenêtre d'instructions. La section 1.4 présente les travaux relatifs se basant sur l'exploitation de la largeur des opérandes pour réduire la complexité du processeur. Cette section distingue notamment l'implémentation des schémas permettant l'identification de la largeur des données et les microarchitectures exploitant cette propriété pour réduire la complexité. La section 1.5 décrit les architectures utilisant plusieurs clusters d'unités fonctionnelles pour réduire la taille des composants critiques du processeur. À noter que cette section est plus particulière à la proposition abordée dans le chapitre 3.

1.2 Consommation électrique

Les processeurs contemporains sont relativement plus rapides et plus performants que leurs prédécesseurs mais exhibent en contrepartie une consommation électrique largement supérieure. Le terme consommation dissimule en fait deux problèmes critiques circonscrivant la réalisation des processeurs embarqués et hautes performances modernes : la *consommation d'énergie* et la *dissipation de puissance*. Optimiser la consommation d'énergie est plus particulier aux systèmes embarqués fonctionnant sur batteries alors que réduire et/ou homogénéiser la dissipation de puissance est primordial aux systèmes hautes performances. Plusieurs travaux de recherches récents [62, 79, 87, 21, 20, 33] ont mis l'accent sur l'aspect critique que couvrent ces problèmes dans la réalisation de microarchitectures modernes. Cette section permet d'appréhender ces derniers en exposant les sources principales de consommation dans la technologie de fabrication actuelle des composants. Elle présente ainsi les diverses équations permettant d'approximer la consommation et ayant été utilisées dans nos modèles analytiques pour quantifier les gains de nos solutions. Nous décrivons également plusieurs optimisations basiques envisagées pour réduire la consommation et dérivées dans ce document.

1.2.1 Sources de consommation

Il existe deux sources principales de consommation dans un circuit CMOS[1]. D'une part, une puissance est dissipée à chaque fois que le circuit est sollicité : la *puissance dynamique*. D'autre part, une puissance est dissipée lorsque le circuit est dans un état stable : la *puissance statique*. La puissance dynamique constitue la fraction dominante de la puissance totale dissipée par un processeur contemporain, soit environ 90% [87]. Le passage aux géométries très fines (< à 0.13 μm) tend toutefois à inverser de plus en plus cette tendance [46]. Ce phénomène est illustré en figure 1.1 et détaillé en section 1.2.2.1. La conception des prochains processeurs semble donc être limitée à la fois par la puissance dynamique et la puissance statique. Nos deux propositions, décrites aux chapitres 2 et 3, suggèrent des solutions pour réduire ces deux paramètres. Pour mieux appréhender ces notions, cette section décrit les diverses sources de consommation ayant lieu dans un circuit CMOS. Le principe de fonctionnement d'un transistor MOSFET[2] est préalablement rappelé.

1.2.1.1 Transistor MOSFET

Le transistor MOSFET (illustré en figure 1.2) est un dispositif semi-conducteur composé de quatre électrodes : la *source*, le *drain*, la *grille* (*gate*) et le *substrat* (*body*). En appliquant une tension entre la grille et le substrat, des charges électriques sont diffusées entre la *source* et le *drain* qui forment le *canal*. Il existe deux types de transistors MOSFET qui se distinguent par un dopage différent : le transistor à *canal p* (P-MOS) et le transistor à *canal n* (N-MOS). Le principe de fonctionnement de ces transistors est analogue mis à part leurs polarités. En mode commutateur, le transistor possède deux états commandés par l'application d'une tension à la grille. Une tension nulle interdit la circulation des électrons dans le canal (cf. figure 1.2). Appliquer une tension supérieure à un seuil donné permet la circulation des électrons dans le canal. Comme la grille est isolée électriquement du reste du composant, on dit qu'elle agit comme une capacité. Idéalement, une fois cette capacité chargée ou déchargée, aucun courant ne devrait être nécessaire pour maintenir l'état courant. La majorité des paramètres d'un transistor sont fixés par la technologie [16]. Les dimensions latérales et verticales du canal et

[1] *Complementary Metal Oxyde Silicon*
[2] *Metal Oxyde Silicon Field Effect Transistor*

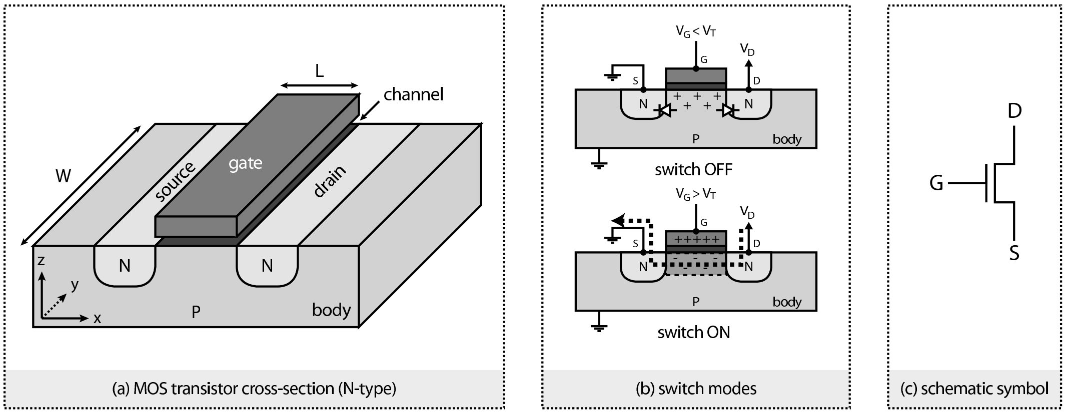


FIG. 1.2 – Transistor N-MOS : (a) vue en coupe ; (b) fonctionnement ; (c) symbole.

la tension de seuil sont modulables (W et L sur la figure 1.2). Un circuit CMOS est une structure composée de deux transistors MOSFET de type complémentaire. Cette structure dissipe une puissance en régime de fonctionnement dynamique et statique.

1.2.1.2 Régime dynamique

La puissance dynamique $P_{dynamic}$ d'un composant CMOS est donnée par l'équation 1.1. Il existe deux types de consommation dynamique : *la puissance de commutation (P_{switch})* et *la puissance de court-circuit ($P_{short-circuit}$)*. Dans la technologie actuelle, la puissance de court-circuit est quantitativement négligeable par rapport à la puissance de commutation [21].

$$P_{dynamic} = P_{switch} + P_{short-circuit} \tag{1.1}$$

Puissance de commutation (P_{switch}) La figure 1.3 illustre les deux sources participant à la puissance de commutation dans un composant CMOS. Nous pouvons ainsi observer que la puissance de commutation est provoquée par la charge et de la décharge de la capacité de charge C_L. Lorsque le circuit effectue une transition du type « *1 → 0* », ceci suscite la charge de la capacité en entrée et la décharge de C_L. Le scénario inverse se produit lors d'une transition de type « *0 → 1* ». La charge d'une capacité engendre un drain d'énergie proportionnel au carré du voltage source et de la capacité C_L : $E = C_L \times V_{dd}^2$. La décharge d'une capacité provoque une dissipation de l'énergie emmagasinée préalablement. L'énergie E_{switch} consommée à chaque commutation d'une porte CMOS est donnée par l'équation 1.2.

$$E_{switch} = \frac{1}{2} \times C_L \times V_{dd}^2 \tag{1.2}$$

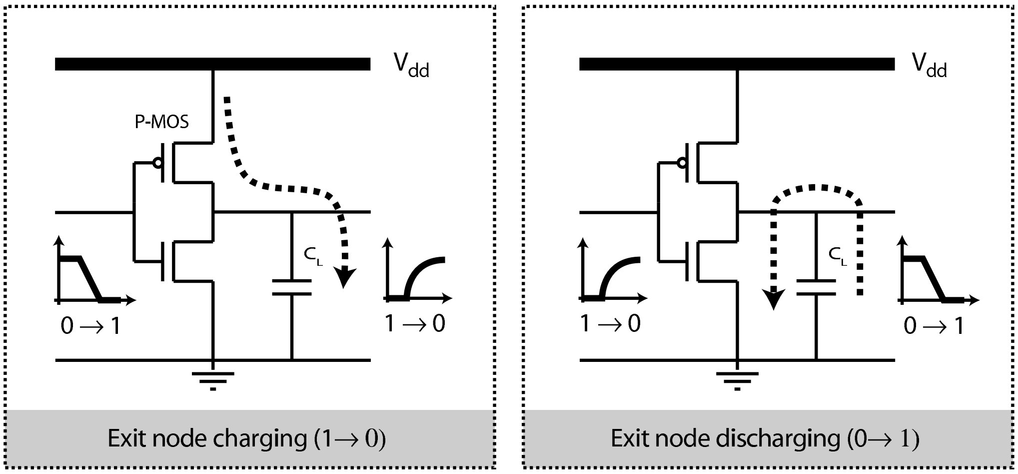


FIG. 1.3 – Puissance de commutation.

La puissance de commutation $P_{gate}^{(dyn)}$ d'une porte CMOS peut être dérivée à partir de l'équation 1.2 pour une fréquence de fonctionnement $f_{switching}$ et un voltage source V_{dd}. La puissance de commutation est ainsi donnée par l'équation 1.3.

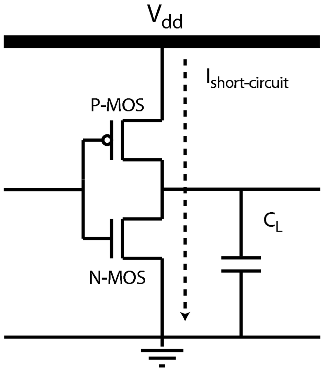
$$P_{gate}^{(dyn)} = \frac{1}{2} \times C_L \times V_{dd}^2 \times f_{switching} \tag{1.3}$$

On peut extrapoler la formule 1.3 au niveau du circuit CMOS complet. La consommation dynamique $P_{circuit}^{(dyn)}$ d'un circuit CMOS est alors donnée par l'équation 1.4.

$$P_{circuit}^{(dyn)} = \alpha \times C_{eq} \times V_{dd}^2 \times f_{clock} \tag{1.4}$$

où : C_{eq} représente la capacité équivalente du circuit CMOS. f_{clock} correspond à la fréquence de cadencement du circuit. α représente le taux d'activité du circuit dont la valeur varie entre 0 et 1. Ce paramètre sera égal à 1 si tous les transistors du circuit sont sollicités simultanément, il sera égal à 0 autrement. Par convention, la valeur de α est souvent fixée à 0.5.

Puissance de court-circuit ($P_{short-circuit}$) La figure 1.4 illustre la puissance de court-circuit dissipée par un composant CMOS. La commutation des composants CMOS devrait théoriquement être effectuée de manière instantanée ; c-à-d. un seul transistor (PMOS ou NMOS) devrait être actif à un instant donné. Il existe toutefois un délai entre l'activation et la désactivation d'un transistor MOS. Les transistors PMOS et NMOS peuvent donc être actifs en même temps et ainsi former un court-circuit entre la source V_{dd} et la masse du composant CMOS. La figure 1.4 illustre ce phénomène en affichant le courant résultant $I_{short-circuit}$. Sur cette figure, le paramètre « *Timeshort* » correspond à la période durant laquelle les transistors N-MOS et P-MOS sont actifs simultanément. La puissance de court-circuit est donnée par l'équation 1.5.



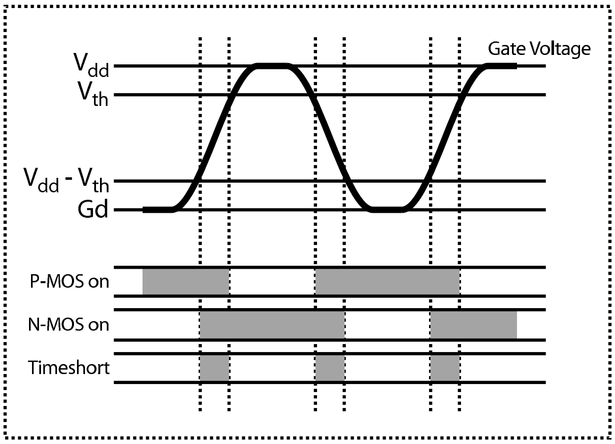


FIG. 1.4 – Puissance de court-circuit.

$$P_{short-circuit}^{(dyn)} = \alpha \times V_{dd} \times I_{short-circuit} \times t \tag{1.5}$$

où : t correspond au délai de propagation du transistor.

1.2.1.3 Régime statique

En théorie, un transistor MOS ne devrait pas dissiper de puissance dans un état stable. Comme cette structure ne modélise pas un interrupteur parfait, il existe toutefois une « *fuite* » du courant. Il existe plusieurs types de courant de fuite. La figure 1.5 illustre les types principaux. On peut discerner : (1) le *courant de fuite en régime sous-seuil* (I_{sub}), (2) le *courant de fuite de jonction* (I_{diode}) et, le *courant de fuite de grille* (I_{ox}).

Le courant de fuite I_{leak} et la puissance consommée en régime statique $P_{leak}^{(stat)}$ sont donnés par la formule 1.6 :

$$I_{leak} = I_{subthreshold} + I_{diode} + I_{ox} \quad et, \quad P_{leak}^{(stat)} = V_{dd} \times (I_{sub} + I_{diode} + I_{ox}) \tag{1.6}$$

Courant de fuite en régime sous-seuil Le courant de fuite en régime de faible inversion ou sous-seuil apparaît lorsque la tension appliquée à la grille est légèrement en dessous du seuil requis pour commuter. Un transistor dans le régime de faible inversion laisse circuler quelques électrons entre le drain et la source ; créant ainsi un courant de fuite. La figure 1.5 illustre mieux ce phénomène. Le courant de sous-seuil dépend de la largeur du canal, de la chaleur, et de la tension de seuil. L'intensité $I_{subthreshold}$ du courant de fuite en régime sous-seuil est approximée par la formule 1.7 :

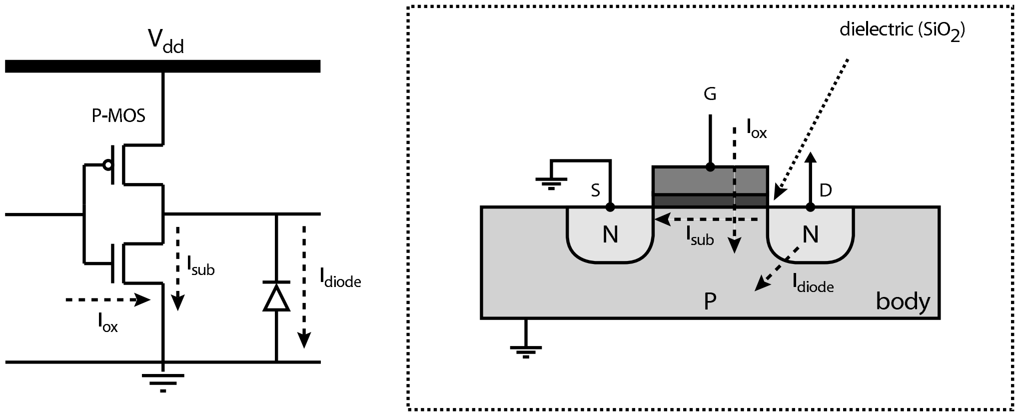


FIG. 1.5 – Courants de fuite.

$$I_{subthreshold} \simeq K_1 \times W \times e^{-V_{th}/\eta V_\theta} \times \left(1 - e^{-V/V_\theta}\right) \quad (1.7)$$

où : K_1 et η sont des constantes. W correspond à la largeur du canal. V_{th} est la tension de seuil. V_θ est la tension thermique ; V_θ se situe à 25mV à la température ambiante et croît de manière exponentielle avec la température [24].

Courant de fuite de jonction Il existe dans les diodes formées entre les électrodes et le substrat un courant de polarisation inverse (cf. figure 1.2). L'intensité de ce courant I_{diode} est donnée par la formule 1.8 :

$$I_{diode} = I_S \left(e^{V/V_\theta} - 1\right) \quad (1.8)$$

où : I_S représente le courant de saturation. V est la tension de polarisation.

1.2.1.4 Courant de fuite de grille

Dans le procédé MOS, la grille est isolée du substrat par une fine couche d'oxyde de silicium (diélectrique). La faible permittivité du silicium, dont la valeur décroît avec l'épaisseur, laisse diffuser des électrons entre la grille et le substrat. L'intensité du courant de fuite de grille I_{ox} est donnée par la formule 1.9.

$$I_{ox} = K_2 \times W \left(\frac{V}{T_{ox}}\right)^2 \times e^{-\alpha T_{ox}/V} \quad (1.9)$$

où : K_2 et α sont des constantes. T_{ox} représente l'épaisseur du diélectrique.

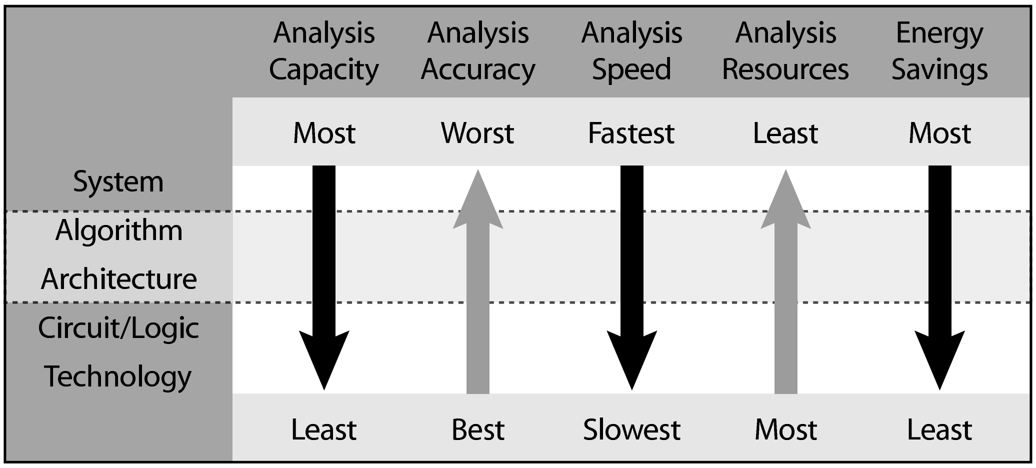


FIG. 1.6 – Niveaux de conception.

1.2.2 Optimiser la consommation

Plusieurs techniques ont été envisagées pour minimiser les puissances dynamiques et statiques dissipées par un processeur [87]. De plus, ces techniques ont été appliquées aux différents niveaux de conception d'un système pour maximiser les gains absolus. La figure 1.6 illustre ces niveaux : *technologique*, *logique*, *architectural*, *algorithmique* et *système d'exploitation*. Les propositions émises dans ce document ont été principalement établies autour des niveaux logique, architectural, et algorithmique. La figure 1.6 met également en avant les propriétés relatives aux outils de mesure pour la consommation. Remarquons que les outils développés ou utilisés dans cette thèse appartiennent au niveau architectural et ces derniers sont couplés à des modèles analytiques. Cette section détaille différentes techniques ayant été employées ou dérivées dans ce document.

1.2.2.1 Niveau technologique

Les équations 1.4 et 1.6 exhibent les deux composantes permettant de réduire efficacement la consommation au niveau technologique. Les consommations en régime statique et dynamique peuvent être ainsi optimisées à travers la diminution du voltage source V_{dd}. La consommation en régime dynamique peut être davantage réduite en diminuant la valeur de la capacité effective C_{eq} du circuit. Comme cette valeur dépend directement des dimensions du composant, il est à noter que sa réduction ne peut uniquement se faire qu'à travers le passage à une géométrie plus fine. Il existe plusieurs autres moyens de produire des composants à faible consommation. Le procédé de fabrication SOI (*Sulfate On Insulator*) permet notamment d'éliminer le courant de fuite de jonction mais en contrepartie d'un coût de production plus élevé [2]. Dans ce document, nous décrirons uniquement la réduction de consommation pour les processeurs implémentés à partir de la technologie CMOS.

Réduire le voltage source Comme stipulé par l'équation 1.4, la consommation due aux commutations est proportionnelle au carré de la source (V_{dd}^2). L'optimisation de ce paramètre peut donc résulter en une diminution quadratique de l'énergie consommée dynamiquement. Le passage du voltage source V_{dd} de 5,0 à 3,3 volts (44% réduction) permet, par exemple, une réduction de 56% de l'énergie consommée [87]. La réduction du voltage source est toutefois limitée par la tension de seuil V_{th} et a un impact direct sur la fréquence maximale du circuit. L'équation 1.10 exprime la relation linéaire qui existe entre la fréquence f_{max} et le voltage source V_{dd}. On remarque ainsi que réduire le voltage source du quart diminue la fréquence de moitié.

$$f_{max} \simeq \frac{Vdd - Vth^2}{Vdd} \tag{1.10}$$

Réduire le voltage V_{dd} implique également une réduction de la tension de seuil afin de conserver un fonctionnement correct [62]. Réduire la tension de seuil influe cependant sur la consommation en régime statique comme mis en évidence dans l'équation 1.7.

Affiner le procédé de gravure La capacité d'un circuit CMOS est déterminée par les dimensions latérales et verticales (W et L) du canal. Le passage à un procédé de gravure plus fin est la méthode la plus efficace pour réduire la taille de la capacité du circuit et donc diminuer la consommation dynamique. Affiner le procédé de gravure a trois objectifs principaux : c-à-d. (1) *réduire les délais de propagation*, (2) *augmenter les densités d'intégration* et, (3) *réduire la consommation*. En théorie, pour se conformer aux conjectures de Moore [60], la miniaturisation doit réduire les dimensions latérales et verticales du composant de 30% (W et L sur la figure 1.2) ; résultant ainsi en une réduction proportionnelle de la capacité. Ce qui implique : (1) la réduction des délais de propagation de *25 à 30%* ; d'où une augmentation de la fréquence opératoire de *33 à 43%*, (2) le doublement des densités d'intégration et, (3) la réduction de *30 à 65%* de l'énergie nécessaire à une commutation (en fonction de V_{dd}). À noter que la surface de la puce augmente également de *25%* à chaque nouvelle génération pour satisfaire la loi de Moore.

La figure 1.1 met en perspective la contribution des diverses sources de consommation en fonction du procédé de gravure. Cette projection [81] met en évidence : (1) une augmentation sensible de la consommation dynamique et (2) une augmentation brutale de la consommation statique pour les générations futures. Pour une taille de motif inférieure à 65nm, la consommation statique pourrait ainsi dominer la consommation globale du processeur. À noter que ce phénomène s'explique essentiellement par (1) le rapprochement des électrodes source et drain (*short-channel effect*) et, (2) la réduction de l'épaisseur de la couche isolant la grille du substrat (*Gate oxide tunneling* ; I_{ox}) [46]. La figure 1.1 montre également la diminution du phénomène (2) si un matériau à

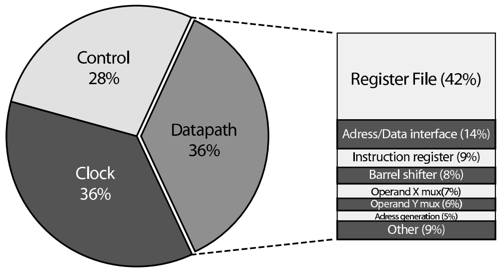


FIG. 1.7 – Distribution pour le processeur Motorola M.Core [80].

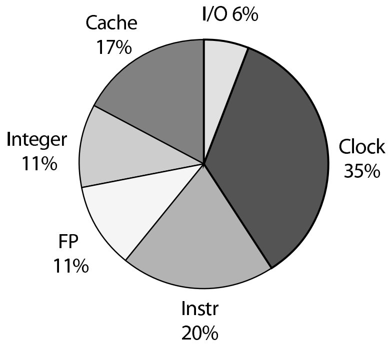


FIG. 1.8 – Distribution pour le processeur Alpha 21264) [36].

forte permittivité (*high-k dielectric*) remplace l'oxyde de silicium utilisé actuellement. Par conséquent, comme les processeurs modernes tendent à implémenter des structures mémoires plus larges, il devient autant essentiel de minimiser leur consommation en régime statique.

1.2.2.2 Distribution de la consommation

Les optimisations appliquées à des niveaux de conception supérieurs concernent généralement les structures dissipant la puissance la plus significative. Cette section détaille la distribution de la consommation dynamique dans deux architectures dédiées à l'embarqué et à l'usage général : le processeur Motorola M.Core [80] et le processeur Alpha 21264 [44]. Les figures 1.7 et 1.8 affichent la distribution de consommation par composant pour ces deux systèmes. À noter que le processeur Motorola M.Core implémente un pipeline RISC 16/32 bits et est dépourvu de caches. Le processeur Alpha 21264 est un processeur 4-voie superscalaire à exécution dans le désordre. Ce processeur peut exécuter jusqu'à 6 instructions par cycle et sa microarchitecture est mieux détaillée dans le chapitre 3.

Nous pouvons remarquer que, pour les deux implémentations, le réseau de distribution de l'horloge (*clock*) participe au tiers de la consommation dynamique totale. Une consommation élevée de ce composant se comprend aisément en sachant que la plupart des structures sont synchronisées par le signal d'horloge à chaque cycle. D'autre part, nous pouvons remarquer que certains composants du chemin de données (*datapath*) sont également fortement sollicités. On peut notamment discerner : le fichier de registres, les unités de calcul et les divers bus de données. Une consommation importante du fichier de registres s'explique par le modèle *load/store* implémenté par les deux processeurs. Les unités mémoires (*caches*) consomment également une quantité considérable de puissance dynamique. La consommation par accès à une donnée mémoire dépend es-

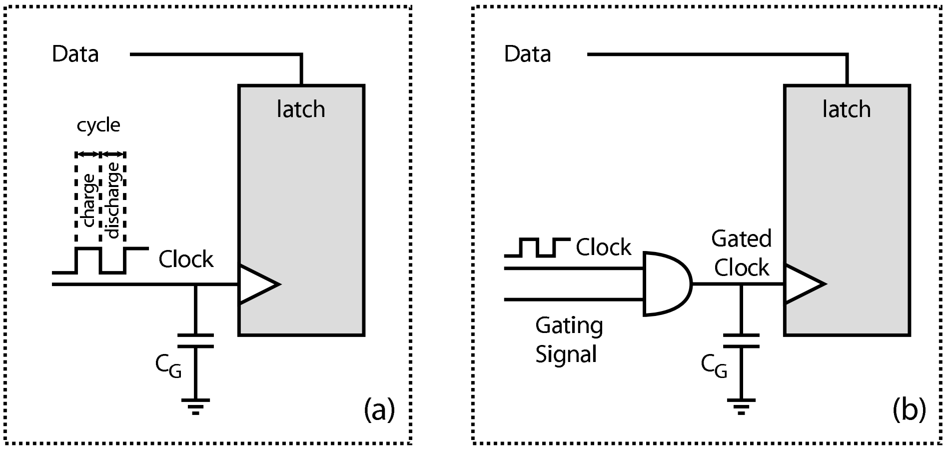


FIG. 1.9 – (a) implémentation conventionnelle, (b) implémentation du *clock-gating*.

sentiellement de la largeur et du nombre de ports de la structure de mémorisation [43]. Les techniques abordées dans ce document tentent de réduire l'activité des éléments ainsi cités.

1.2.2.3 Niveau circuit/logique

Cette section décrit deux techniques implémentées au niveau circuit pour réduire les consommations dynamique et statique du processeur. La première solution inhibe le signal d'horloge à certaines structures pour réduire leur consommation dynamique. La seconde solution propose d'adapter la tension de seuil pour réduire la consommation statique.

Inhiber le signal d'horloge (*clock-gating*) Chaque circuit composant un processeur consomme une puissance à chaque cycle de l'horloge. Pour une logique statique, ceci est dû à la charge et la décharge des capacités en entrée de circuit. La figure 1.9 (a) illustre le phénomène pour un verrou. C_G représente la capacité de charge du verrou qui est directement reliée au signal d'horloge. Nous pouvons ainsi observer que cette capacité sera sollicitée à chaque cycle indépendamment des données à mémoriser. Bloquer le signal d'horloge en amont peut donc permettre d'éliminer cette activité. La technique permettant d'inhiber le signal d'horloge à une structure donnée est appelée : « *clock-gating* » [33]. La figure 1.9 (b) expose les modifications matérielles nécessaires pour la mise en œuvre de cette technique pour une logique statique. Le *clock-gating* requiert essentiellement l'ajout d'une porte logique et d'un signal qui commande la mise en veille des composants (*Gating Signal*). Ce signal peut être cadencé plus faiblement pour minimiser la consommation. Le *clock-gating* parvient à réduire efficacement l'activité car les capacités des éléments ajoutés sont négligeables. La technique de *clock-gating* peut également être appliquée à une logique dynamique [53]. Son utilisation peut toutefois

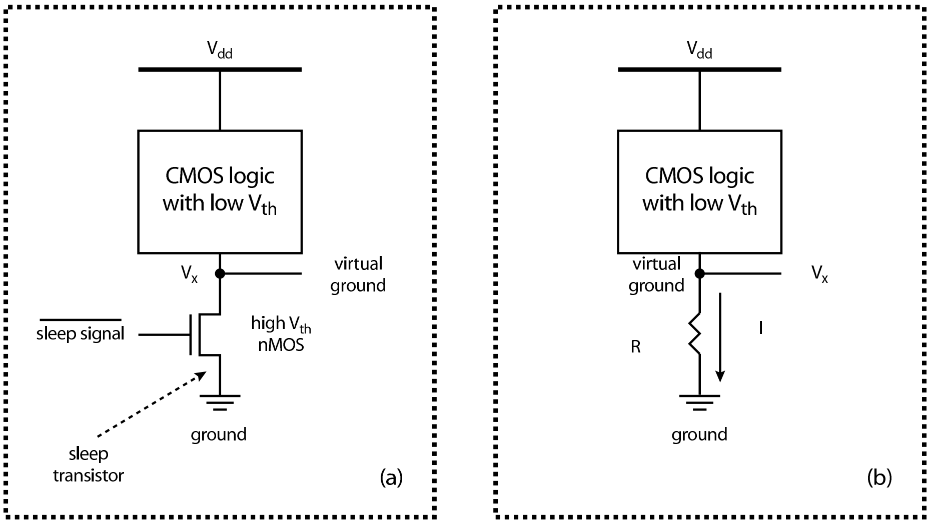


FIG. 1.10 – circuit MTCMOS (a) mode actif (b) mode inactif.

générer des impulsions parasites (*glitches*) ainsi que des déphasages dans l'horloge (*clock skew*). De plus, une sollicitation trop fréquente de ces mécanismes peut impacter de manière négative sur la consommation de part les phénomènes d'induction [87, 74].

Adapter la tension de seuil Comme exposé en section 1.2.1.3, le courant de fuite est fortement dépendant de la tension de seuil. Le courant de fuite en régime de faible inversion augmente notamment de façon exponentielle par rapport au voltage de la tension de seuil (cf. section 1.2.1.3). Au niveau circuit, plusieurs approches ont considéré l'adaptation de la tension de seuil pour minimiser le courant de fuite [46]. La technique MTCMOS (Multi-threshold CMOS) permet notamment de réaliser un circuit possédant de telles caractéristiques. Le principe de cette technique est de coupler deux catégories distinctes de transistors au sein du même circuit : des transistors à seuil élevé et des transistors à seuil faible. L'activation d'une catégorie dépend ensuite du contexte d'utilisation. Les transistors à seuil élevé permettent notamment de limiter le courant de fuite. Les transistors à faible seuil procurent des délais de transition rapides. La figure 1.10 illustre l'implémentation d'un composant MTCMOS. On remarque ainsi que la logique *sleep signal* permet de contrôler l'activation des transistors. Dans le mode « *sleep* = 0 », les transistors à seuil élevé sont activés afin de procurer une commutation rapide. Inversement, dans le mode « *sleep* = 1 », les transistors à seuil faible sont activés afin de permettre une limitation du courant de fuite.

1.2.2.4 Niveau architectural

De nombreux travaux de recherche ont attaqué le problème de consommation au niveau microarchitectural [87, 62, 21, 32, 58]. Au niveau de la réduction d'énergie,

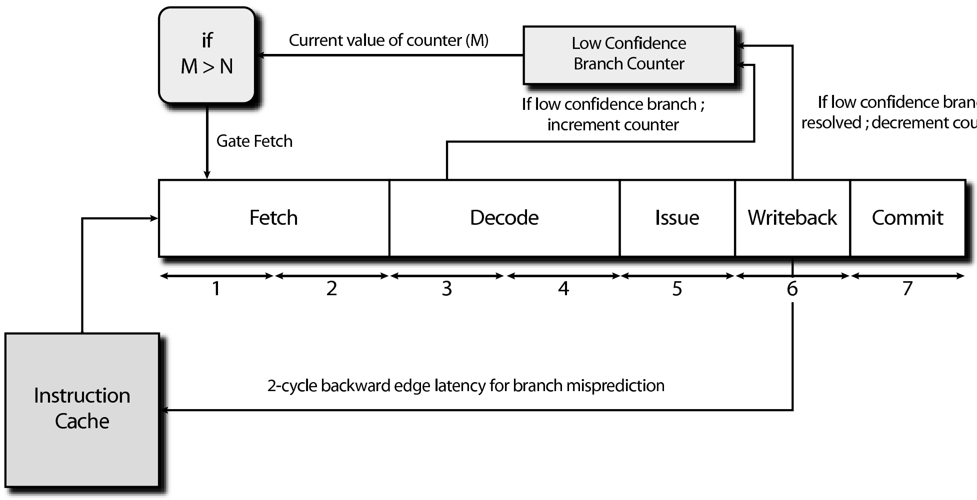


FIG. 1.11 – Gel du pipeline [58].

on peut essentiellement discerner deux grandes catégories. La première regroupe les méthodes envisageant la réduction d'énergie via la désactivation des portions inutilisées. La seconde catégorie rassemble les techniques suggérant de minimiser l'énergie à travers une utilisation plus optimale des ressources. Cette section décrit deux propositions académiques s'appuyant sur les techniques logiques décrites dans la section 1.2.2.3. La première technique s'appuie sur le *clock-gating* pour limiter l'activité inutile due aux mauvaises spéculations. La seconde technique s'appuie sur l'adaptation du voltage source pour réduire la consommation statique du cache de données.

Geler les étages du pipeline [58] L'utilisation de techniques spéculatives dans les processeurs contemporains permet de séquencer un nombre supérieur d'instructions par cycle. Ces techniques peuvent néanmoins avoir un impact négatif sur la consommation d'énergie. Manne et al. [58] ont proposé de limiter la consommation superflue engendrée par une mauvaise spéculation. Ces auteurs définissent ainsi les instructions issues d'un mauvais chemin de contrôle comme un *travail superflu* (*Extra Work*). Le travail superflu E_w pour un étage du pipeline est donnée par la formule 1.11 :

$$E_w = \left(\frac{nb_{seenInsn} - nb_{commitedInsn}}{nb_{commitedInsn}} \right) \tag{1.11}$$

où : $nb_{seenInsn}$ correspond au nombre total d'instructions procédées par un étage du pipeline. $nb_{commitedInsn}$ correspond au nombre total d'instructions accomplies par le processeur. Par exemple, $E_w = 20\%$, si uniquement 100 sur 120 instructions exécutées appartiennent à un chemin de contrôle correct.

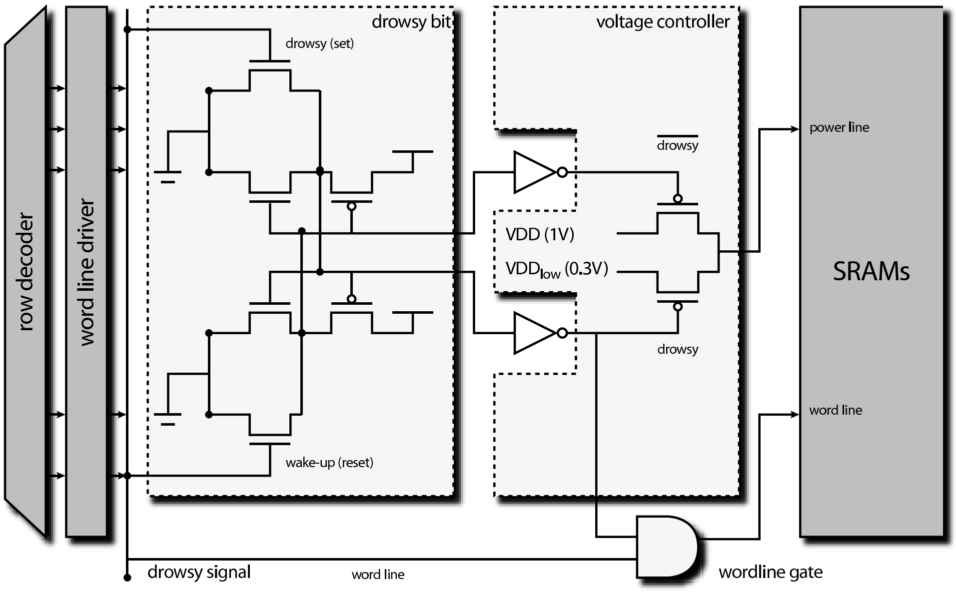


FIG. 1.12 – Implémentation du mode *drowsy* pour une ligne de cache.

Manne et al. [58] proposent de minimiser la quantité E_w en renforçant la précision des prédictions de branchements. Ces auteurs suggèrent à cet effet d'agrémenter le prédicteur de branchements par un *estimateur de confiance* [37]. Cet estimateur associe une valeur à chaque prédiction permettant d'augmenter leur précision. Une valeur élevée correspond à la forte probabilité que la direction du branchement soit correcte. Inversement, une valeur faible correspond à la forte probabilité que la prédiction soit incorrecte. Dans ce dernier cas, les auteurs proposent de geler les étages du pipeline (chargement et décodage) afin de prévenir l'insertion des instructions appartenant à un mauvais chemin de contrôle. Ce qui permet ainsi au processeur d'éviter un travail superflu. L'architecture envisagée par Manne et al. est présentée en figure 1.11. La prédiction de branchements a lieu à l'étage de chargement et la résolution des branchements à l'étage d'exécution. Les auteurs suggèrent aussi l'ajout d'un compteur pour éviter un gel trop fréquent du pipeline.

Mettre en veille les voies du cache (*drowsy caches* [32]) Une fraction dominante de la consommation statique est dissipée par les unités de mémorisation. Comme le suggère l'équation 1.6, un moyen efficace pour réduire celle-ci est de considérer le voltage de la source V_{dd}. Powell et al. [73] ont proposé une telle approche pour minimiser la consommation statique des caches. Leur approche consiste à adapter dynamiquement les dimensions du cache en fonction des besoins de l'exécution. Pour ce faire, les auteurs proposent d'éteindre les bancs du cache les moins accédés à l'exécution. Bien que cette technique s'avère efficace, les informations contenues dans le cache sont perdues et nécessitent d'être rechargées en cas de nouvelle sollicitation. Flautner et al.

[32] ont adopté une approche alternative qui permet de réduire la consommation du cache de données tout en préservant l'état des cellules mémoire. Au lieu de déconnecter les voies du cache, les auteurs optent plutôt pour la mise en mode *drowsy* des cellules mémoire. Le mode *drowsy* ou mode sommeil consiste à appliquer une tension suffisamment faible aux cellules afin de préserver l'état des données. À noter que les cellules mémoire dans un tel état ne peuvent être accèdes ; un accès peut en effet détruire le contenu de la cellule. Une manière d'implémenter le mode *drowsy* est de considérer une technologie ABB-MTCMOS [46]. Cette technologie n'est toutefois pas adéquate car elle sollicite un voltage source plus fort et des délais plus longs. Flautner et al. ont proposé une nouvelle implémentation matérielle du mode *drowsy*. La figure 1.12 illustre les modifications à apporter à une ligne de cache pour supporter un tel mode. On peut ainsi remarquer que le mode *drowsy* est implémenté à travers l'adaptation de la tension d'alimentation (V_{DD} et V_{DDLow}). Un signal (*drowsy bit*) commande notamment le fonctionnement d'une ligne de cache. On remarque aussi que la précharge d'une ligne est bloquée (*wordline gate*) lorsque la cellule est dans un mode *drowsy*. Flautner et al. proposent plusieurs politiques exploitant le mode *drowsy* pour réduire la consommation statique du cache de données. Il semble que la politique la plus simple soit la plus efficace : c-à-d. mettre périodiquement les lignes en mode *drowsy* sans considérer l'historique des accès. À noter que les auteurs supposent qu'un cycle est nécessaire pour « *réveiller* » une cellule « *endormie* ». À noter également que les étiquettes des lignes de cache ne sont pas mises en mode *drowsy* pour éviter le cycle de pénalité associé à la comparaison.

1.3 Complexité du chemin de données

Cette section détaille la complexité des principaux composants critiques du chemin de données attaqués à travers ce document. Ces composants sont principalement constitués du fichier de registres, du réseau de *bypass*, des logiques de séquencement et des interconnexions.

1.3.1 Fichier de registres

Le fichier registres est une structure fondamentale aux processeurs modernes qui sont majoritairement basés sur un modèle RISC [69]. Dans ce modèle, les seules instructions pouvant accéder à la mémoire sont du type *load/store*. Le reste des opérations ne peuvent uniquement accéder à leurs opérandes qu'à travers le fichier de registres. L'exécution d'un programme génère donc une forte sollicitation de cette structure ; d'où une dissipation de puissance élevée [91]. Par ailleurs, les microarchitectures modernes font appel à des techniques de plus en plus agressives pour extraire le maximum de parallélisme entre les instructions. Ces techniques nécessitent de stocker un nombre supérieur d'informations tout en sollicitant des accès plus fréquents au fichier de registres. L'approche naïve pour satisfaire ces demandes consiste à concevoir un fichier

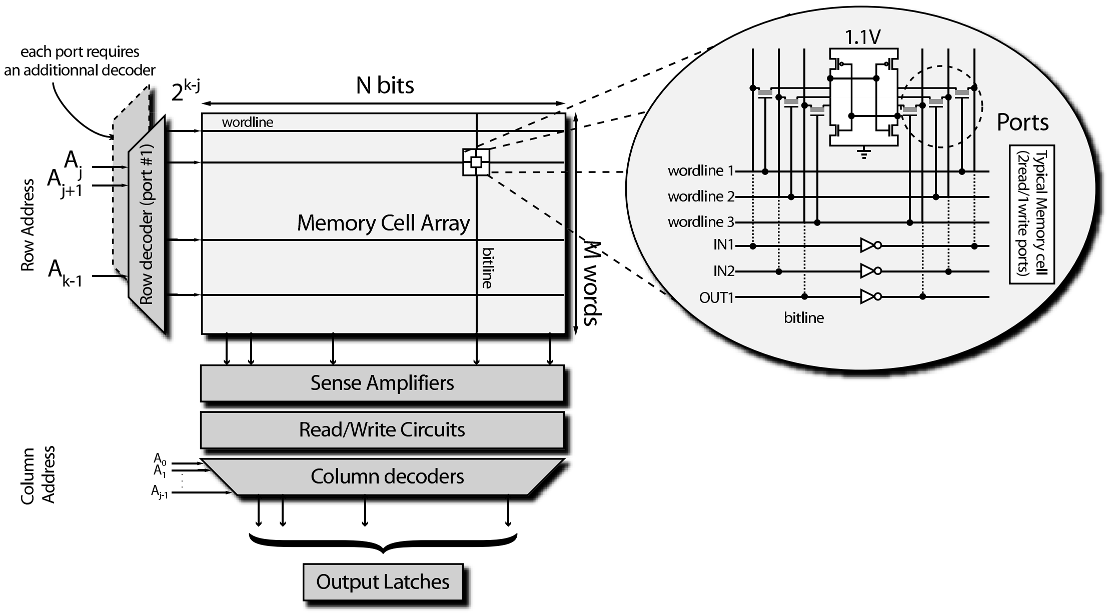


FIG. 1.13 – Organisation classique du fichier de registres avec les détails du point mémoire (2 ports lecture/1 port écriture).

de registres aux dimensions plus larges et pourvu d'un nombre de ports d'accès plus important. Cette approche devient toutefois de plus en plus difficile à mettre en œuvre sans impacter sur le temps de cycle, la surface et la consommation du processeur. Cette section détaille l'implémentation physique d'un fichier de registres monolithique et analyse la complexité de ce dernier.

1.3.1.1 Organisation conventionnelle du fichier de registres

La figure 1.13 illustre la structure physique pour un fichier de registres monolithique constitué de points mémoire implémentant 2 ports d'accès en lecture et 1 port d'accès en écriture. Plusieurs blocs fonctionnels constituent un fichier de registres :

- la matrice de points mémoire (*Memory cell array*)
- les décodeurs d'adresse de ligne et de colonne (*read/write column/row decoders*)
- les lignes de mot et de bit (*wordlines and bitlines*)
- les amplificateurs différentiels (*sense amplifiers*)
- les circuits d'entrée/sortie (*read/write circuits*)

La matrice de points mémoire est chargée du stockage des données sous la forme binaire. Cette dernière est généralement architecturée selon une grille composée de M lignes de mot par N lignes de bits. Lors d'un ou plusieurs accès au fichier de registres, les opérations de lecture/écriture se font simultanément sur tous les points mémoire des rangées décodées (chaque rangée correspond à un registre). En d'autres termes,

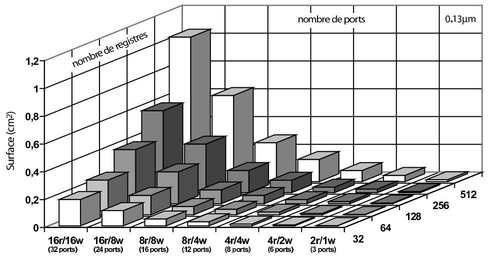


Fig. 1.14 – Surface de la matrice mémoire ($cm2$).

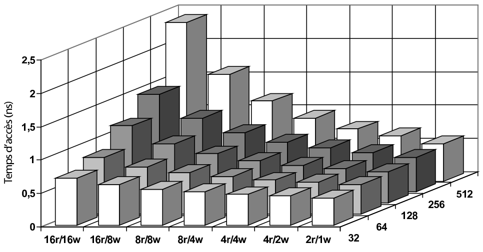


Fig. 1.15 – Temps d'accès (ns).

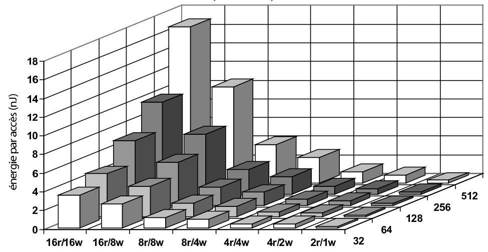


Fig. 1.16 – Énergie par accès (nJ).

une ou plusieurs lignes de mot peuvent être préchargées alors que toutes les lignes de bit doivent être activées. Les données sont alors lues ou écrites dans ou depuis les points mémoire grâce aux circuits d'entrée/sortie. Les décodeurs d'adresse sont responsables de déterminer quelles sont les rangées à procéder. Chaque port d'accès en lecture/écriture nécessite un décodeur d'adresse car plusieurs accès peuvent être établis de manière simultanée. Les amplificateurs différentiels permettent de convertir le signal détecté sur une ligne de bit en la valeur binaire correspondante.

1.3.1.2 Surface, temps d'accès et consommation du fichier de registres

La complexité du fichier de registres est essentiellement caractérisée par trois facteurs : la *surface de silicium*, le *temps d'accès* et la *consommation*. Cette section détaille ces facteurs tout en illustrant leur corrélation avec le nombre de registres et de ports d'accès du fichier de registres. Les mesures, affichées en figure 1.14, 1.15 et 1.16, ont été obtenues à l'aide d'une version modifiée[3] de CACTI 3 [43, 83] pour un procédé de gravure CMOS de $0.13\mu m$. Dans ces figures, l'axe des abscisses représente le nombre de ports d'accès en lecture/écriture et l'axe z représente le nombre de registres total.

[3] la partie relative aux étiquettes a été évincée

Surface de silicium Pour un fichier de registres monolithique composé de N_{read} ports en lecture, N_{write} ports en écriture, un total de N_{read} lignes de bit, N_{read} lignes de mot, $2 * N_{write}$ lignes de bit et N_{write} lignes de mot doivent traverser chaque cellule mémoire. L'équation 1.12 exprime la surface de silicium $W_{registerfile}$ occupée par un fichier de registres comportant N_{regs} registres de largeur R_{width}. Dans cette équation, le facteur ω correspond à la largeur d'une ligne plus la distance séparant deux lignes, soit $(width + spacing)$ de la figure 1.19 [91] .

$$W_{registerfile} = N_{regs} \times R_{width} \times \underbrace{\omega^2 \times (N_{read} + N_{write}) \times (N_{read} + 2 \times N_{write})}_{taille\ cellule\ memoire} \quad (1.12)$$

L'équation 1.12 montre ainsi que la surface de silicium occupée par le fichier de registres est fonction du produit du nombre de registres N_{regs}, du nombre de bits R_{width} par registre et de la taille d'une cellule mémoire. La surface de silicium augmente donc linéairement avec le nombre de registres et le nombre de bits alors qu'elle croît de manière quadratique avec le nombre de ports d'accès en lecture/écriture. La figure 1.14 montre sur l'axe des ordonnées l'augmentation de la surface de silicium pour différentes configurations en nombre de ports et de registres. La corrélation entre la surface et le nombre de bits n'est pas représentée sur cette figure mais est similaire à celle de l'augmentation du nombre de registres.

Temps d'accès Le temps d'accès au fichier de registres en lecture ou en écriture est principalement déterminé par le délai de deux composants : (1) *le délai de propagation dans les interconnexions* (c-à-d. le délai pour propager la donnée à travers les lignes de mot et de bit) et (2) *le temps de traversée des structures de décodage et d'entrée/sortie.* Le passage aux géométries très fines tend à majorer le temps d'accès au fichier de registres par celui du délai des lignes de mot et de bit [75, 29, 65]. Les délais de (1) et (2) sont dépendants à la fois du nombre de registres et du nombre de ports. Doubler le nombre de ports d'accès affecte deux fois plus le temps d'accès au fichier de registres [29]. Le fait de doubler le nombre de ports implique en effet le doublement du nombre de lignes de bit et de mot alors que doubler le nombre de registres implique uniquement le doublement du nombre de lignes de mot. La figure 1.15 met en avant cette variation du temps d'accès pour différentes configurations en nombre de ports et de registres.

Consommation d'énergie L'énergie dissipée par un fichier de registres correspond à la somme des capacités des éléments sollicités à chaque accès : c-à-d. décodeurs d'adresse, amplificateurs différentiels, circuits d'entrée/sortie, matrice de points mémoire (cf. section 1.3.1.1). Zyuban et al. [91] ont proposé un modèle détaillant la consommation des fichiers de registres multiports. Chaque accès au fichier de registres entraîne la précharge d'une ou plusieurs lignes de mot et la précharge de toutes les lignes de bit. La capacité des lignes de mot majore ainsi la consommation du fichier de registres. Pour un fichier de registres comportant un nombre large de ports d'accès, cette capacité correspond principalement à celle des interconnexions. Pour un fichier de registres avec un faible nombre de ports d'accès, la capacitance des lignes de bit correspond à celle des transistors composant la cellule mémoire [75]. La figure 1.16 illustre la variation de la consommation d'énergie du fichier de registres pour différentes configurations en nombre de ports et de registres.

1.3.1.3 Optimiser la consommation du fichier de registres

De nombreux travaux [23, 7, 82, 68, 91, 65] ont abordé le sujet de la réduction de consommation dans les fichiers de registres. Au niveau microarchitectural, nous pouvons discerner deux catégories principales. D'une part, nous pouvons distinguer les techniques s'attaquant au problème par la réduction du nombre de registres. D'autre part, nous pouvons distinguer les techniques essayant de diminuer la consommation par la réduction du nombre de ports. Les propositions abordées dans cette thèse introduisent une troisième dimension pour réduire la consommation du fichier de registres : c-à-d. à travers la réduction du nombre de bits par registre.

Réduction du nombre de registres Diverses méthodes ont été proposées pour parvenir à la réduction du nombre de registres. Dans [23, 7], les auteurs proposent, pour ce faire, l'implémentation d'un fichier de registres hiérarchique. Dans une philosophie similaire à celle des caches, les auteurs considèrent l'implémentation de deux niveaux de registres. Un premier niveau dont le temps d'accès et la consommation sont écourtés car il est constitué uniquement de quelques registres. Les auteurs font alors en sorte de stocker les valeurs appartenant au chemin critique de l'exécution dans ces registres. Un second niveau plus large en registres et contenant des données moins critiques à l'exécution. Cette technique permet de réduire la consommation totale du fichier de registres car le premier niveau est plus souvent sollicité et ses caractéristiques permettent une dissipation de puissance inférieure.

Réduction du nombre de ports Dans [82, 68], les auteurs proposent de réduire le nombre de ports au moyen de techniques de partitionnement du fichier de registres en clusters ou en bancs. Il est à rappeler que la consommation du fichier de registres est quadratique par rapport au nombre de ports en lecture/écriture (cf. section 1.3.1.2). Ces techniques ont donc un impact considérable sur la consommation électrique. Park et al. [68] ont observé que la majorité des instructions obtiennent leurs opérandes depuis le *bypass* de données. Ces auteurs ont alors proposé un mécanisme permettant de prédire la disponibilité des opérandes pour réduire le nombre d'accès en lecture sur le fichier de registres et donc le nombre de ports de ce type. Les auteurs ont également suggéré le partitionnement du fichier de registres en bancs pour diviser le nombre de ports en écriture. Ces deux techniques permettent une réduction considérable de la consommation tout en requérant une complexité minimale. Partitionner le fichier de registres en bancs nécessite toutefois la duplication de certaines structures (décodeurs d'adresse)

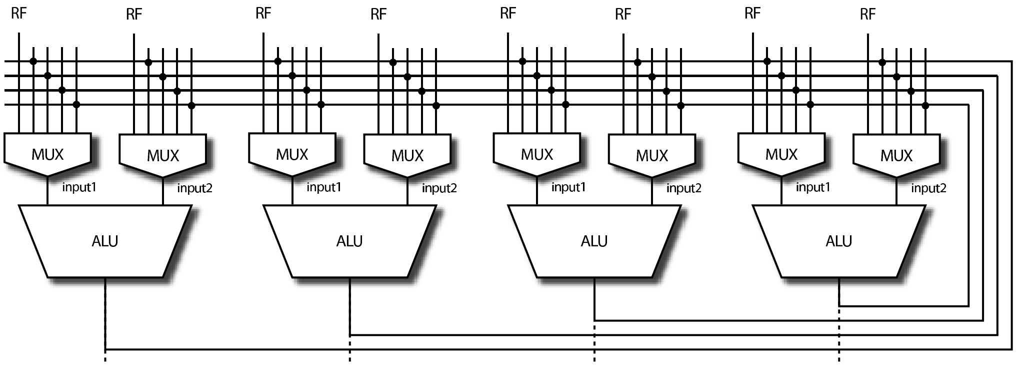


FIG. 1.17 – Réseau du *bypass* de données pour processeur superscalaire de degré 4. [70]

et peut ainsi engendrer une surface de silicium et une consommation supérieures. Une approche alternative pour réduire le nombre de ports d'accès consiste à distribuer et/ou partitionner le fichier de registres entre différents clusters. Cette approche est discutée en section 1.5.

1.3.2 Logique du *bypass* de données

Le mécanisme de *bypass* de données permet de propager un résultat dès sa sortie des unités de calcul sans attendre son écriture dans le fichier de registres. Ce mécanisme autorise ainsi le séquencement « *au plus tôt* » des opérations dépendantes du résultat produit[4]. La logique réalisant le *bypass* de données est composée de deux types distincts de structures : *la logique de propagation des données* et *la logique de contrôle*. La logique de propagation des données est essentiellement composée des bus de données sur lesquels la donnée est disponible et de multiplexeurs permettant de sélectionner la source pour un opérande. La logique de contrôle permet de commander l'état de ces multiplexeurs par le biais des *tags* propagés lorsqu'un résultat est produit.

La complexité de la logique réalisant le *bypass* de données dépend ainsi, à la fois, du nombre de ressources de calcul N et du nombre de cycles X pendant lesquels la donnée est rendue disponible [65]. Pour un processeur implémentant un réseau de *bypass* complet et un fichier de registres monolithique, il existe ainsi $X * N + 1$ sources possibles pour chaque opérande. La complexité du réseau de *bypass* pour un processeur superscalaire de degré 4 est illustrée en figure 1.17. Le délai du *bypass* de données est essentiellement déterminé par la longueur et la largeur des bus de données et des multiplexeurs. Le passage à l'échelle irrégulier des interconnexions tend à majorer le délai du *bypass* de données dans les processeurs modernes. Le délai de cette structure est pratiquement équivalent à celui des unités fonctionnelles dans le processeur Intel Itanium [70].

[4]Cette technique correspond au terme « *back-to-back execution* »

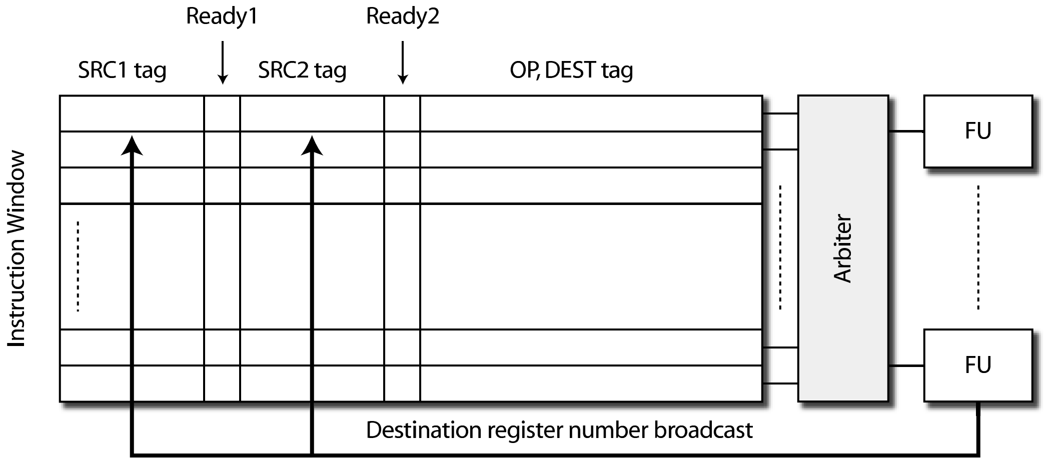


FIG. 1.18 – Illustration du fonctionnement des logiques de séquencement.

1.3.3 Logique de séquencement des instructions

Le séquencement des instructions dans un processeur à exécution dans le désordre est réalisé par deux logiques : *la logique de réveil* et *la logique de sélection*. La logique de réveil permet de marquer les instructions en fonction de la disponibilité des opérandes sources. La logique de sélection permet de résoudre les conflits éventuels de ressources et de séquencer les instructions prêtes à s'exécuter. La figure 1.18 illustre le fonctionnement de ces mécanismes pour une fenêtre d'instructions implémentée à partir d'une logique associative CAM [45, 41]. Chaque entrée de la fenêtre d'instructions contient le code opératoire et les numéros physiques (autrement appelés *tags*) des registres sources et destination. Chaque opérande source est associé à un *ready bit* qui informe la logique de sélection de l'état de l'instruction. Lorsque tous les *ready bits* de cette instruction sont correctement positionnés, la logique de sélection (*Arbiter*) peut envoyer celle-ci sur une unité de calcul appropriée. Lorsque cette instruction a terminé son exécution, le numéro physique du registre destination est propagé à la fenêtre d'instructions pour réveiller les opérations dépendantes. Chaque entrée de la fenêtre d'instructions compare ce numéro pour positionner les *ready bits* correspondants.

La complexité de la logique de séquencement est ainsi déterminée par le nombre de résultats que le processeur peut produire par cycle. Si chaque instruction peut procéder jusqu'à deux opérandes et le processeur possède une largeur d'exécution N, il en est effet nécessaire d'implémenter $2{*}N$ comparateurs pour chaque entrée de la fenêtre d'instructions [65]. Augmenter le degré du processeur affecte ainsi le délai et la consommation de la fenêtre d'instructions. De manière plus particulière, le délai de la logique de réveil est déterminé par le temps requis pour propager les *tags* aux instructions dépendantes et pour comparer les différents opérandes sources. Le délai de la logique de sélection dépend de la taille de la structure d'arbitrage (*Arbiter*) et donc du nombre d'entrées de la fenêtre d'instructions [65]. Dans les processeurs modernes, il devient de plus en plus difficile de réaliser en un cycle les tâches de séquencement des opérations [59, 14, 45].

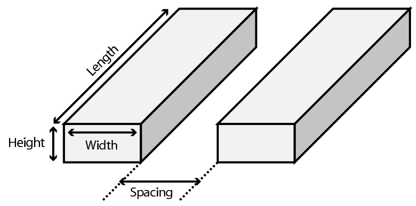


FIG. 1.19 – Implémentation physique des interconnexions.

1.3.4 Les interconnexions

Plusieurs travaux [40, 9] ont mis en évidence les problèmes liés au dimensionnement des interconnexions par rapport à celui des transistors. L'affinement plus difficile de ces structures de communication engendre en effet plusieurs problèmes dont un délai, une surface et une consommation plus élevés pour les futures générations. Une projection récente [81] indique ainsi un délai pouvant avoisiner les 30 cycles pour communiquer une donnée aux extrémités de la puce dans les générations prochaines de processeurs. Magen et al. [56] ont également montré que les interconnexions participent à 50% de la consommation dynamique dans les processeurs contemporains basés sur une architecture Intel Pentium 4. Ce document propose une nouvelle solution pour réduire significativement la consommation, la surface et le temps de cycle des interconnexions grâce à une microarchitecture à clusters optimisés en fonction de la taille des opérandes. Cette section détaille la surface de silicium, le délai et la consommation liés aux interconnexions.

1.3.4.1 Surface de silicium

La surface de silicium occupée par les interconnexions dans un processeur est fonction de plusieurs paramètres : le nombre, la longueur et la largeur des interconnexions, et l'espace de séparation entre deux interconnexions. La figure 1.19 illustre ces divers paramètres. L'implémentation physique des interconnexions impose une distance de séparation minimale entre deux interconnexions (*spacing*) pour préserver le temps de propagation des signaux. Le rapprochement des interconnexions augmente en effet la capacité de ces structures et donc le délai associé [40]. Le chapitre 3 montre comment il est possible de faire varier ces paramètres pour réduire la surface tout en préservant les performances [9].

1.3.4.2 Délai de propagation

Les interconnexions possèdent trois caractéristiques électriques fondamentales : la résistance, la capacité et l'inductance. L'inductance est mieux détaillée dans [40]. La résistance R_{wire} exprime la faculté des interconnexions à véhiculer les charges électriques. Ce paramètre est généralement approximé par l'équation 1.13 [40] où ρ représente la permittivité du matériel et *barrier* correspond à la barrière diélectrique prévenant la diffusion des électrons hors des interconnexions. Les autres paramètres sont illustrés en figure 1.19.

$$R_{wire} = \frac{\rho}{(Height - barrier)(width - 2 * barrier)} \tag{1.13}$$

La capacité C_{wire} des interconnexions représente le nombre de charges électriques requises pour changer leur potentiel électrique. Cette capacité est exprimée par l'équation 1.14 [40] où les paramètres ϵ_x sont des constantes associées aux diélectriques, K correspond à l'effet Miller, $lspacing$ est la distance de séparation entre deux couches métalliques, et $fringe$ est une constante.

$$C_{wire} = \epsilon_0(2K\epsilon_{horiz}\frac{Height}{Spacing} + 2\epsilon_{vert}\frac{width}{lspacing}) + fringe(\epsilon_{horiz}, \epsilon_{vert}) \tag{1.14}$$

Le délai de propagation des signaux dans les interconnexions est proportionnel au produit résistance $*$ capacité (RC). Il est donc possible d'améliorer ce délai en augmentant la largeur (*width*) des interconnexions et l'espacement (*spacing*) entre ces dernières sur la puce. Le chapitre 3 présente une telle méthode pour implémenter des interconnexions hétérogènes [9].

1.3.4.3 Consommation

La consommation dynamique dans les interconnexions est similaire à celle des transistors. Cette consommation P_{wire} est généralement exprimée par l'équation 1.15 [56] où a représente le facteur d'activité, f la fréquence de cadencement, N_{wire} le nombre de bus, V le voltage source. Le chapitre 3 expose une solution permettant de réduire la consommation des interconnexions à travers le paramètre N_{wire}.

$$P_{wire} = a * f * N_{wire} * C_{wire} * V^2 \tag{1.15}$$

1.4 Identifier et exploiter les données tronquées

Les sections précédentes ont exposé les notions relatives à la complexité des processeurs contemporains. Les sections suivantes exposent les travaux relatifs à nos propositions permettant de réduire la complexité à travers la propriété des programmes observée par [12]. Celle ci concerne le traitement fréquent des données de faible dimension à l'exécution. Plusieurs travaux [12, 17, 85, 64, 55, 27] ont profité de la forte présence des données de faible dimension à l'exécution pour (1) *réduire la complexité du processeur* ou (2) *augmenter le parallélisme d'instructions*. Quelque soit la finalité des approches considérées, exploiter cette propriété des programmes requiert la mise en œuvre d'un mécanisme spécialisé pour identifier la largeur des données à l'exécution.

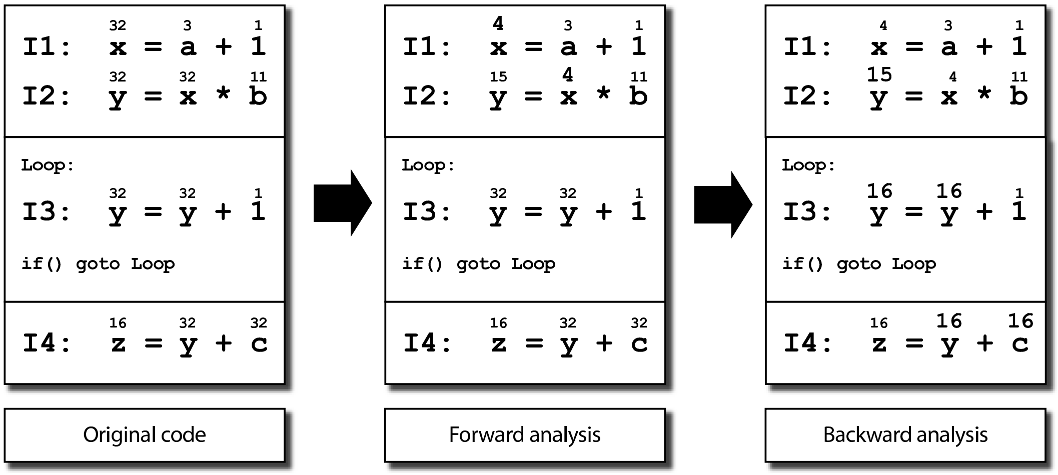


FIG. 1.20 – Déduction de la taille des variables par propagation [57].

Cette section détaille, d'une part, les divers moyens envisagés dans la littérature pour réaliser cette tâche. Elle expose ensuite les supports matériels suggérés pour réduire la complexité du processeur grâce au contenu des données dynamiques.

Terminologie *Par la suite, nous désignerons toutes les données comportant un nombre de bits significatifs inférieur à 16 bits par : « données tronquées ». Par convention, les opérations qui manipulent uniquement des opérandes tronqués seront référées comme : « opérations tronquées ». À noter que la terminologie définie s'applique exclusivement aux données et opérations du type entier.*

1.4.1 Schémas de détection

La littérature regroupe deux catégories distinctes de schémas pour identifier les données tronquées manipulées par un programme. Ces catégories se distinguent essentiellement par le support matériel utilisé pour ce faire ; nous pouvons ainsi distinguer (1) *les approches logicielles* et (2) *les approches matérielles.*

1.4.1.1 Approches logicielles

Plusieurs travaux ont proposé de recourir à une analyse statique pour attribuer une dimension plus optimale aux variables d'un programme. Ces travaux reposent globalement sur un même concept. Stephenson et al. [85] généralisent ce concept comme un problème de propagation de valeurs. Le principe est de renforcer le compilateur par un algorithme de propagation standard mais qui diffuse la dimension des variables au lieu de leurs valeurs. Les dimensions disponibles initialement sont ainsi propagées à l'ensemble du programme. Un processus itératif est ensuite utilisé pour propager

```
uint32 x ;
for (i = 0 ; i < 25 ; i++) {
   x += i ; /* 8-bit : 88% ; 16-bit : 12% */
   .....
}
```

FIG. 1.21 – Aspect restrictif des approches logicielles.

cette connaissance, ainsi que celle inférée à chaque itération, au reste du programme. L'algorithme s'arrête lorsqu'un état stable ou un nombre arbitraire d'itérations est atteint. Plusieurs variantes de cet algorithme ont été présentées dans la littérature [15, 18, 57, 85].

La méthode proposée par Mahlke et al. [57] est illustrée en figure 1.20. Le programme original regroupe quatre instructions : $I1$ à $I4$. Les tailles connues statiquement sont annotées au dessus de chaque opérande. Les auteurs considèrent deux étapes pour assigner une taille plus optimale aux variables du programme. D'une part, l'analyse en avant permet d'inférer la taille maximale des opérandes résultats à partir de la sémantique des opérateurs et de la taille des opérandes sources. Ainsi, le résultat x de l'instruction $I1$ aura une taille maximale de 4-bit à l'exécution. D'autre part, l'analyse en arrière permet d'optimiser la taille de chaque opérande source selon la dernière valeur produite. La taille de la variable y dans $I3$ peut ainsi être déclarée sur 16-bit étant donnée sa dernière écriture dans $I2$.

Ces techniques de détection logicielles sont toutefois relativement limitées. Elles doivent en effet attribuer une dimension maximale aux variables pour préserver la sémantique du programme. Le code C exposé en figure 1.21 témoigne de cette limitation. Dans cet exemple, nous supposons que la variable x n'est plus utilisée par la suite. Cette variable est déclarée comme un entier non signé 32 bits. Elle s'exécute la majeure partie du temps (88%) sur une largeur de 8 bits. Elle requiert toutefois une largeur de 16 bits dans les dernières itérations de boucle (12%). Une approche logicielle est contrainte d'utiliser un entier 16 bits pour déclarer la variable x afin de conserver un comportement correct à l'exécution. Cette approche ne peut toutefois prendre en compte le fait que la variable x n'excède pas 8 bits la majeure partie de l'exécution. Il existe plusieurs cas similaires qui démontrent la nature limitée des approches logicielles [12]. À la différence, une approche matérielle permet de détecter la totalité des données tronquées manipulées à l'exécution.

1.4.1.2 Approches Matérielles

Plusieurs schémas de détection matériels ont été proposés dans la littérature [12, 22, 17, 55, 27]. Ces schémas regroupent principalement *les approches conservatrices* [12, 22, 17] et *les approches spéculatives* [55, 27]. Les approches conservatrices emploient une

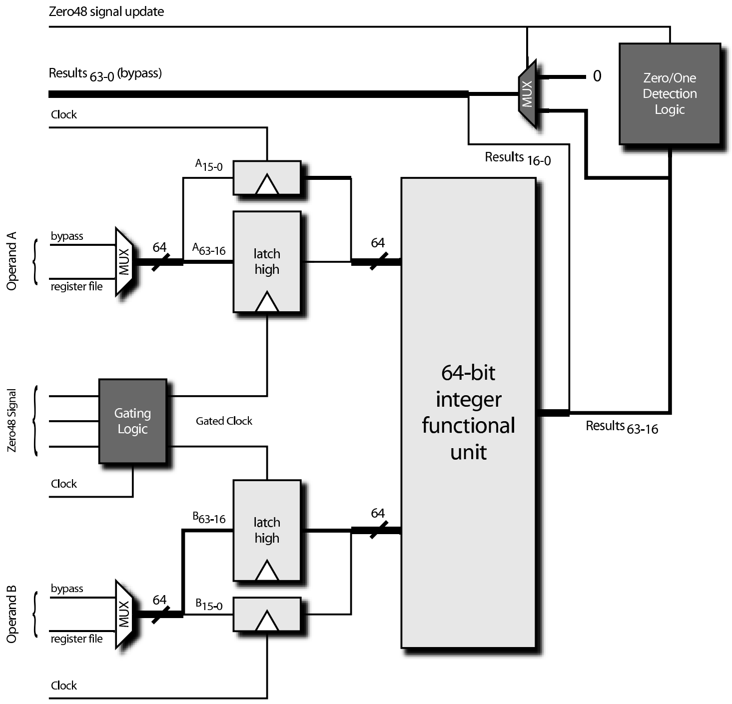


Fig. 1.22 – Logique de détection postérieure au calcul [12].

logique spécialisée pour déterminer la largeur des données. Les approches spéculatives sont assistées par un mécanisme dédié pour prédire la largeur des données.

Logique de détection (postérieure au calcul) [12] Dans les travaux de Brooks et Martonosi [12], la logique de détection est localisée à la sortie des unités d'exécution et des unités mémoires. Le mécanisme matériel mis en œuvre par ces auteurs est présenté en figure 1.22. En localisant ainsi cette logique, l'information relative à la détection doit être stockée afin d'éviter une redondance à la lecture. Brooks et al. ont suggéré à cet effet de stocker l'information dans les registres et stations de réservation au moyen de bits spécifiques. Cette information est transitée au moyen du signal *zero48* qui est routé aux structures concernées. À noter que le réseau de *bypass* bénéficie également de ces informations. À la lecture des opérandes, le signal *zero48* permet de déterminer si l'opération est tronquée ou non. D'un point de vue technique, l'implémentation proposée requiert l'ajout matériel de trois structures. Ces dernières sont mises en évidence sur la figure 1.22. Une logique à l'entrée de l'unité fonctionnelle (*Gating Logic*) permet de désactiver la portion haute des verrous. La logique de détection (*zero/one detect logic*) est physiquement composée de deux logiques agissant en parallèle. La première logique permet de détecter si les bits de poids fort sont positionnés à *zéro*. Ceci permet notamment de détecter une valeur entière positive nulle. La seconde logique permet de détecter si les bits de poids fort sont tous à *un*. Ceci permet inversement de détecter une valeur entière signée, en notation complément à deux. La dernière structure est

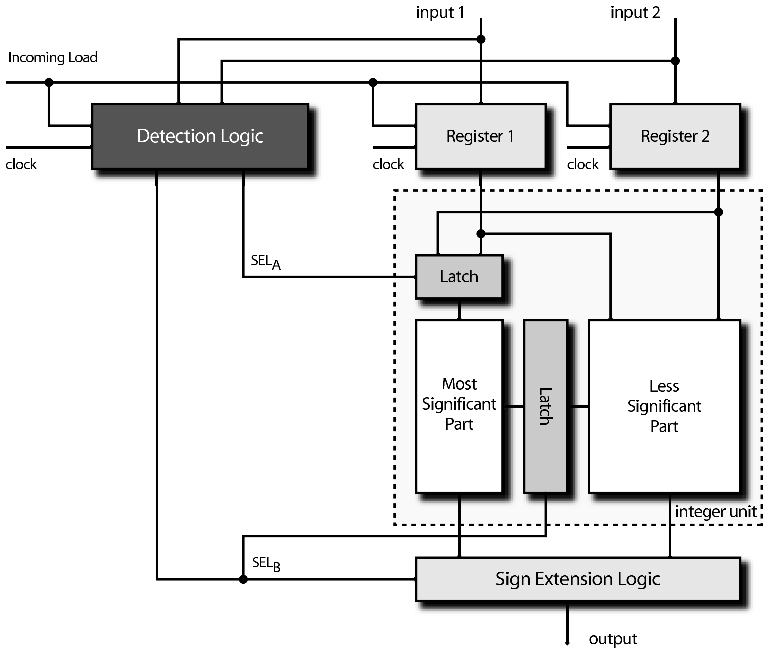


FIG. 1.23 – Logique de détection antérieure au calcul [22].

un multiplexeur. Selon le résultat de la détection et le type d'opération, ce dernier multiplexe une valeur nulle (zéro ou un) aux bits de poids fort.

Logique de détection (antérieure au calcul) [22] Choi et al. [22] ont préféré situer la logique de détection à l'entrée des unités fonctionnelles. À noter que cette logique permet aussi de détecter les valeurs signées. À la différence de l'étude [12], localiser ainsi la logique de détection permet de minimiser la complexité du matériel. Il n'est en effet pas nécessaire d'user de bits spécifiques pour sauvegarder le résultat de la détection. La logique propageant cette information est par conséquent inutile. Le multiplexeur n'est plus requis car les données sont directement écrites dans les structures concernées. La logique de détection ne nécessite pas d'être dupliquée au niveau de l'unité mémoire. Malgré de tels avantages, un tel schéma peut toutefois avoir un impact conséquent au niveau des performances. Un cycle supplémentaire est effectivement nécessaire pour détecter les opérations tronquées. De plus, ce schéma est exclusivement limité à la réduction d'énergie au niveau de la lecture des opérandes. Ceci peut être significatif en sachant que les accès en écriture sont généralement plus coûteux en termes de consommation [91].

Identification des octets significatifs [17] Canal, Gonzáles et Smith [17] ont proposé un schéma amélioré permettant d'utiliser les donnés tronquées pour optimiser l'activité de l'ensemble du pipeline. Canal et al. qualifient différemment une donnée tronquée. Les études [12, 22] supposent qu'une donnée est tronquée lorsque sa largeur est inférieure à 16 bits. Dans l'étude de Canal et al., une donnée est organisée en plusieurs tranches ; une tranche correspond à un octet. Le modèle microarchitectural proposé est capable de procéder uniquement les tranches significatives au calcul. Considérons par exemple la valeur décimale **56** codée sur 8 bits : « **00 11 10 00** ». Dans les approches de [12, 22], cette valeur n'est pas estimée comme une donnée tronquée (2 bits). L'approche de Canal est ainsi supérieure car elle peut identifier et optimiser un tel scénario. Pour préciser au matériel les tranches à procéder, les registres sont étendus par des *bits d'extension*. Ces auteurs considèrent l'ajout de deux et trois bits d'extension. À noter que la tranche de poids faible est toujours accédée. Deux bits d'extension permettent d'identifier trois tranches significatives d'une donnée 32 bits. La notation « **00 00 1F 75 : 10** » indique ainsi que les octets de poids fort ne sont pas significatifs. De même pour la valeur signée « **FF FF 1F 75 : 10** ». Trois bits d'extension permettent de différencier les tranches significatives des données mixtes ; c-à-d. des données ayant des octets positionnés à **00** ou **FF**. Ces données sont fréquemment utilisées par les adresses mémoires. Dans un schéma à trois bits d'extension, chaque bit indique l'utilité d'une tranche. Ainsi, « **FF 00 1F 75 : 001** » signifie que seuls les bits de poids faible doivent être accédés. La génération des bits d'extension utilisent une logique spécialisée similaire à [12].

Prédiction matérielle des données tronquées [55] Au lieu d'employer une approche conservatrice comme dans [12, 17], Loh [55] propose de prédire la largeur que nécessitent les opérandes à l'exécution. Cette approche est viable car Loh a démontré la forte tendance des opérations à procéder des opérandes de même dimension au cours de l'exécution. L'auteur propose deux prédicteurs matériels pour déduire la largeur des données à l'exécution. Le premier mécanisme est un prédicteur de largeur avec compteur à remis-à-zéro. Le second mécanisme est un prédicteur trimodal agrémenté par un compteur à saturation. La différence essentielle entre ces prédicteurs réside dans le taux de mauvaises prédictions effectives et conservatrices. Ces notions sont mieux détaillées dans le chapitre 3. Loh suppose qu'une mauvaise prédiction est détectée à la fin de l'étage d'exécution et dans ce cas les instructions présentes dans le pipeline doivent être rejouées.

Prédicteur avec compteur à remis-à-zéro La figure 1.24 illustre ce prédicteur. Deux informations sont maintenues pour chaque instruction du programme. La première information concerne la largeur de la dernière instance dynamique de l'instruction. La seconde information représente le nombre d'occurrences où l'instruction a exhibé la même largeur au cours de l'exécution. L'auteur suppose trois largeurs distinctes : 1-16 bits, 17-33 bits et, 34-64 bits. Un compteur k bits est utilisé pour maintenir la seconde information. Ce compteur k bits renforce la précision des prédictions. Si le compteur indique une valeur inférieure à $2^k - 1$, le prédicteur indique que l'instruction n'est pas une instruction tronquée. Autrement, la prédiction est faite selon la largeur de la dernière instance dynamique de l'instruction. En cas de mauvaise prédiction ou si l'instruction

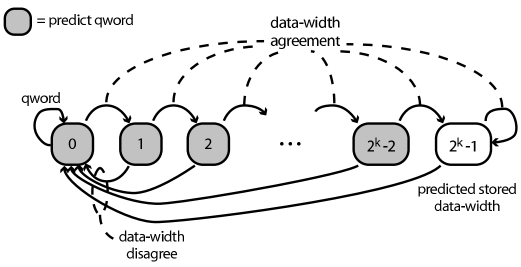


FIG. 1.24 – Prédicteur avec compteur à remis-à-zéro [55]

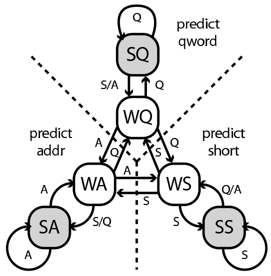


FIG. 1.25 – Prédicteur trimodal à saturation [55]

courante n'est pas tronquée, le compteur est remis à zéro. Autrement, si la prédiction est bonne, l'estimateur est incrémenté jusqu'à ce qu'il y ait saturation.

Prédicteur trimodal à saturation La figure 1.25 représente ce prédicteur. Ce dernier possède trois modes représentant la largeur potentielle de l'instruction : *short*, *addr* et *quad-word*. Chaque mode est subdivisé en un état fort et un état faible. L'état fort dénote la forte tendance de l'instruction à exhiber la largeur indiquée. Réciproquement, l'état faible dénote la faible tendance de l'instruction à exhiber la largeur indiquée. Le prédicteur dans un état $*X$ prédit qu'une instruction aura une largeur X. Par exemple, $*A$, englobant les états **SA** et **WA**, signifie que le prédicteur prédit une largeur de type *addr*. Une mauvaise prédiction dans un état faible change l'état de l'automate à l'état faible de la dernière dimension de l'instruction. Une prédiction correcte dans l'état faible permet à l'automate d'entrer dans l'état fort correspondant. Une mauvaise prédiction dans un état fort implique le passage à l'état faible correspondant.

1.4.2 Supports matériels pour réduire la consommation d'énergie

Plusieurs études ont proposé des moyens matériels pour exploiter la propriété des programmes à s'exécuter avec une fraction significative de données tronquées. Certaines études [12, 17, 27] ont utilisé cette propriété pour réduire la consommation alors que d'autres [55] l'ont employée pour améliorer les performances. Le paradigme SIMD [50] est une illustration des techniques ayant exploité les données tronquées pour accélérer le temps de calcul. Comme notre étude concerne principalement la réduction d'énergie, nous nous limiterons toutefois aux travaux ayant cette finalité. Il faut cependant remarquer que les extensions SIMD peuvent être également utilisées pour réduire la consommation du processeur [26]. L'objectif du paradigme SIMD est en effet d'optimiser l'usage du chemin de données tout en réduisant le temps de calcul ; une réduction de l'énergie peut alors résulter. L'implémentation du paradigme SIMD suscite toutefois un support matériel de complexité non négligeable ainsi que le recours à un compilateur sophistiqué. Les instructions SIMD nécessitent, en effet, que le programme cible exhibe un haut degré de parallélisme d'instructions et de données ; ce qui implique généralement le recours à des techniques de vectorisation complexes [72]. De plus, comme les opérations d'une instruction SIMD sont homogènes, les techniques SIMD peuvent conséquemment négliger un nombre considérable d'optimisations [64]. Quoiqu'il en soit, l'approche envisagée dans ce document est orthogonale au modèle SIMD ; il est donc possible de combiner les deux schémas pour bénéficier d'un meilleur compromis performance/consommation.

1.4.2.1 Redimensionnement dynamique des unités fonctionnelles [12]

Brooks et Martonosi [12] ont proposé d'exploiter les données tronquées pour optimiser l'activité des unités entières de calcul. Ces auteurs suggèrent un schéma agressif dans lequel un mécanisme matériel permet d'accommoder la largeur des unités aux besoins du calcul. Brooks et al. perçoivent un entier 64 bits comme la juxtaposition de : *16 bits de poids faible significatifs* et *48 bits de poids fort, éventuellement, non significatifs*. Les auteurs proposent ainsi la mise en œuvre matérielle d'unités entières de calcul pourvues de deux modes de fonctionnement : 16 et 64 bits. Les opérations dont tous les opérandes sources sont inférieurs à 16 bits sont exécutées dans le mode 16 bits. Les opérations dont, au moins, un opérande source excède 16 bits sont exécutées dans le mode le plus large.

Du point de vue technique, le redimensionnement des unités fonctionnelles se fait au moyen de circuits utilisés pour le *clock-gating* (cf. section 1.2.2.3). Contrairement à un schéma traditionnel, la logique de désactivation agit à la granularité de l'opérande au lieu de celle de l'opération. Le synoptique de la microarchitecture proposée dans [12] est exposé en figure 1.22. On remarque sur cette figure que seules les portions hautes des unités d'exécution (c-à-d. les 48 bits de poids fort) sont soumises à la logique de désactivation ; la désactivation étant commandée par les signaux *zero48* et *one48* (détaillés en section 1.4.1.2). À noter que les 16 bits de poids faible sont procédés de manière ordinaire. En sortie d'unité, on peut aussi remarquer la présence d'un multiplexeur. La fonction de ce dernier est de sélectionner les bits à écrire selon le résultat de la détection. Si les portions hautes des opérandes sont nulles, il est en effet inutile de les traiter mais il est, toutefois, nécessaire de positionner les bits de la donnée finale ; ce qui entraîne ainsi une consommation dans l'unité de mémorisation associée.

Au niveau de la réduction d'énergie, le mécanisme proposé par Brooks est relativement efficace car la consommation d'une ALU entière est quasi-linéaire par rapport au nombre de bits traités. En théorie, une addition sur 16 bits devrait consommer environ quatre fois moins qu'une addition sur 64 bits. Dans [13], Brooks et Martonosi ont présenté une version étendue de [12] où, cette fois-ci, l'unité entière possède trois modes de fonctionnement différents : 16, 33, et 64 bits. Les auteurs justifient l'ajout du mode 33 bits par la fréquente utilisation de cette dimension par les calculs d'adresses. La modification nécessaire à cette nouvelle implémentation révèle toutefois la complexité et l'inflexibilité du modèle envisagé par Brooks. Cette implémentation requiert,

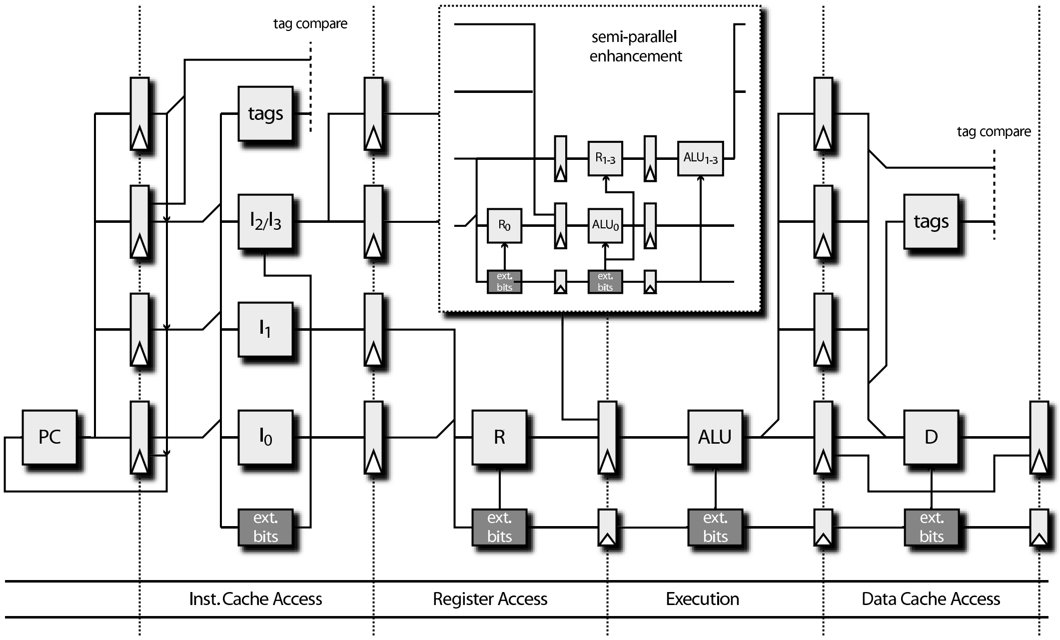


FIG. 1.26 – Implémentation sérialisée du pipeline [17]

en effet, l'ajout de signaux supplémentaires, pour commander la désactivation, et celui de multiplexeurs plus larges pour écrire les données finales. Dans le chapitre 2, nous découvrirons que les approches agressives peuvent également augmenter, de manière significative, la consommation maximale du processeur. En effet, une sollicitation trop fréquente du mécanisme de *clock-gating* peut provoquer des pics dans la consommation ; c-à-d. le phénomène d'induction ou *di/dt* [74]. Par la suite, nous mentionnerons comment notre approche peut permettre de limiter un tel phénomène.

1.4.2.2 Pipeline de largeur fixe implémentant une exécution sérialisée [17]

Canal, Gonzáles et Smith [17] ont suggéré plusieurs modèles architecturaux pour minimiser la consommation dynamique via les opérations tronquées. Ces auteurs ciblent plus particulièrement le contexte des systèmes embarqués. Les modèles proposés sont dérivés d'une architecture classique 32-bit 5 étages. À la différence de Brooks et al. [12], l'approche envisagée par Canal et al. s'applique à l'ensemble du pipeline. Il est à savoir que ces auteurs perçoivent une donnée 32 bits comme la juxtaposition de plusieurs tranches de 8 bits. Chaque tranche peut notamment contenir des informations significatives ou non ; des bits d'extension attachés aux données permettent de caractériser l'utilité des données (cf. section 1.4.1.2). Les auteurs envisagent plusieurs implémentations présentant un compromis différent en termes de consommation et performance. On peut notamment discerner l'implémentation séquentielle et l'implémentation semi-parallèle.

Implémentation séquentielle (8 bits) Le premier modèle est illustré en figure 1.26. Ce modèle implémente un chemin de données de largeur fixe 8 bits mais opère toutefois sur des données 32 bits. En d'autres termes, chaque étage du pipeline ne peut procéder qu'un seul octet par cycle. Si les bits d'extension (présents à chaque étage) révèlent qu'un étage doit opérer sur une donnée supérieure à 8 bits, l'exécution se fait alors sur plusieurs cycles. Du point de vue architectural, le fichier de registres, le cache de données et le cache d'instructions sont scindés en quatre bancs de 8 bits ; un seul banc étant accédé par cycle. La consommation dynamique de ces structures est réduite car l'accès à un banc 8 bits sollicite environ quatre fois moins d'énergie que l'accès à un banc 32 bits. À noter que les bancs non accédés restent actifs afin de préserver l'état des données. Les auteurs proposent également de réduire l'activité du cache d'instructions grâce à un encodage spécifique des instructions. Cet encodage ordonne les champs d'une instruction (opcode, registres, immédiat) selon les informations étant critiques à l'exécution ; e.g. opcode et registres afin de pouvoir décoder et lire les registres au plus tôt. Le nouveau format permet notamment de regrouper les champs les plus significatifs dans les trois premiers octets. Les auteurs considèrent aussi que l'accès au cache d'instructions se fait sur les trois premiers bancs pour ne pas ralentir l'exécution.

Implémentation semi-parallèle (16 bits) L'implémentation séquentielle réduit significativement l'activité du processeur mais au détriment des performances. Canal et al. proposent une implémentation intermédiaire où le facteur performance est privilégié. Cette implémentation est présenté en figure 1.26. Cette implémentation est relativement similaire à l'implémentation séquentielle. La différence essentielle réside dans l'élargissement de certains étages du pipeline. Le fichier de registres peut maintenant être accédé sur deux bancs par cycle. Ce qui permet de lire et écrire les données 32 bits deux fois plus vite. Les unités fonctionnelles sont également élargies pour pouvoir traiter l'octet supplémentaire. Les autres étages du pipeline restent similaires à l'implémentation séquentielle.

1.4.2.3 Compactage des données tronquées [27, 48]

Deux études récentes [27, 48] ont suggéré de compacter les données tronquées au sein du même registre pour optimiser l'usage de ces structures à la façon des techniques SIMD [50]. Ergin et al. [27] proposent notamment le partitionnement des registres 64-bit en portions de 16 et 32-bit. Chaque registre 64-bit du processeur peut ainsi contenir différentes données 16 et 32-bit dont la répartition au sein du registre peut être non contiguë ; l'association d'un masque permettant de distinguer les opérandes stockées. Ce compactage des données permet donc au processeur de percevoir un nombre de registres physiques plus important. Mais, ceci impose également le maintien de listes annexes (*free lists*) pour indiquer au mécanisme de renommage les portions des registres physiques allouables.

Deux méthodes sont proposées pour compacter les données au sein des registres. La méthode conservatrice assigne un registre physique à toute opération productrice. La détection des bits significatifs permet ensuite de décider si la valeur produite doit être allouée à une nouvelle portion de registres ou non. Lorsque la valeur est assignée à un nouveau registre, le matériel doit notifier ces changements au mécanisme de séquencement des instructions. La méthode spéculative permet d'allouer les résultats à une portion du registre selon la taille prédite pour éviter ainsi ce besoin de notification.

La solution proposée par Ergin et al. accentue néanmoins la complexité de plusieurs composants critiques du chemin de données. Plusieurs listes annexes sont en effet nécessaires pour maintenir les portions de registres physiques libres. Ceci peut alors poser un problème de passage à l'échelle car la taille de ces listes est dépendante du nombre de registres physiques. Pour la technique conservatrice, la logique de réveil des instructions doit être modifiée pour considérer la réallocation des données à une nouvelle portion de registres. Cette logique fait typiquement partie des structures déterminant le temps de cycle des processeurs superscalaires (cf. section 1.3.3). Chaque registre doit également être associé à un masque pour pouvoir distinguer les opérandes stockés. Ce qui implique une complexification du fichier de registres et du réseau de *bypass* des données.

1.5 Le modèle clusterisé

Le chapitre 3 de ce document adresse le problème de complexité à travers la proposition d'un modèle clusterisé dimensionné en fonction de la taille des données traitées. Le principe des microarchitectures clusterisées est de distribuer les ressources critiques du processeur entre plusieurs entités distinctes, appelées *clusters*. Chaque cluster contient un sous-ensemble des ressources du processeur tels que les unités de calcul, les queues d'instructions et les fichiers de registres. Ce partitionnement des ressources permet alors de réduire la structure matérielle des composants concernés et autorise ainsi une fréquence d'horloge très élevée. Par surcroît, étant donné que la largeur d'exécution est distribuée entre les clusters, ceci permet également l'implémentation de processeurs superscalaires à très large degré d'exécution. Contrairement à un modèle monolithique, une microarchitecture clusterisée peut néanmoins présenter un niveau de performances inférieur. Les résultats produits sur un cluster doivent en effet être communiqués aux clusters distants pour permettre l'exécution des opérations dépendantes. Comme les interconnexions utilisées pour propager les résultats entre les clusters sont plus lentes, une dégradation de l'IPC peut alors résulter.

Les performances d'une microarchitecture clusterisée sont donc fortement corrélées au mécanisme de distribution des instructions. L'heuristique, employée à cet effet, doit ainsi être capable de minimiser le nombre de communications inter-cluster tout en maintenant l'équilibrage des charges. Une situation de mauvais équilibrage des charges peut en effet affecter davantage l'exécution de part une utilisation non optimale des ressources d'exécution. Plusieurs modèles de microarchitectures clusterisées accompagnés par leurs heuristiques d'allocation ont été proposés dans la littérature [10, 19, 30, 65, 8]. Plusieurs modèles de communication, topologies d'interconnexion, et degré de parti-

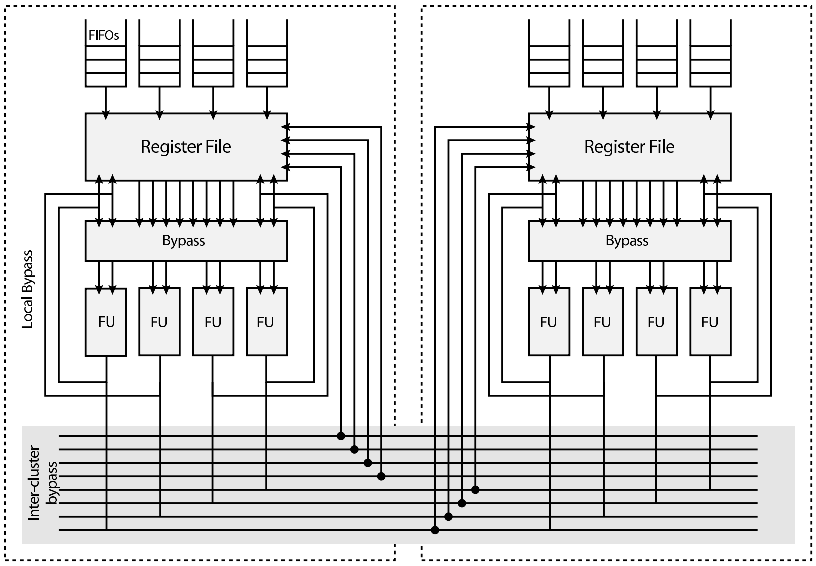


FIG. 1.27 – Microarchitecture clusterisée (Palacharla et al.[65])

tionnement ont également été étudiés [86, 66]. Les processeurs modernes partitionnent notamment le cœur d'exécution entre un cluster entier et un cluster flottant (e.g. Alpha 21264 [44]). Cette section décrit plusieurs propositions de la littérature [65, 30, 82, 35] se distinguant par l'implémentation du fichier de registres. L'étude décrite dans [65] considère ainsi la duplication du fichier de registres, celle dans [30] évalue la distribution de cette structure alors que notre solution [82] examine un schéma hybride. Gonzalez et al. [35] ont présenté, dans une étude très récente, un modèle similaire à notre proposition, détaillée au chapitre 3. Nous décrivons ce modèle tout en anticipant

1.5.1 Approche de Palacharla [65]

Dans [65], Palacharla et al. ont proposé de partitionner le cœur d'exécution d'un processeur superscalaire de degré 8 entre deux clusters homogènes . Chaque cluster peut procéder jusqu'à 4 instructions par cycle. L'implémentation envisagée par ces auteurs est illustrée en figure 1.27. Chaque cluster se compose de 4 files d'instructions (FIFOs), 4 unités fonctionnelles et d'une copie du fichier de registres. Les FIFOs modélisent la fenêtre d'instructions d'une implémentation conventionnelle. À la différence, les instructions appartenant à une même FIFO sont contraintes à s'exécuter dans l'ordre alors que les opérations dépendantes doivent être assignées à la même FIFO. L'utilisation des FIFOs permet de réduire la complexité de la fenêtre d'instructions car la propagation des résultats concerne uniquement les instructions provenant d'une même FIFO. Chaque cluster implémente une copie intégrale du fichier de registres pour faciliter le mécanisme de distribution des instructions décrit ci-après. Les résultats produits par une unité fonctionnelle sont propagés à chaque fichier de registres. Quelques cycles supplémentaires sont toutefois nécessaires pour mettre à jour le contenu de la copie

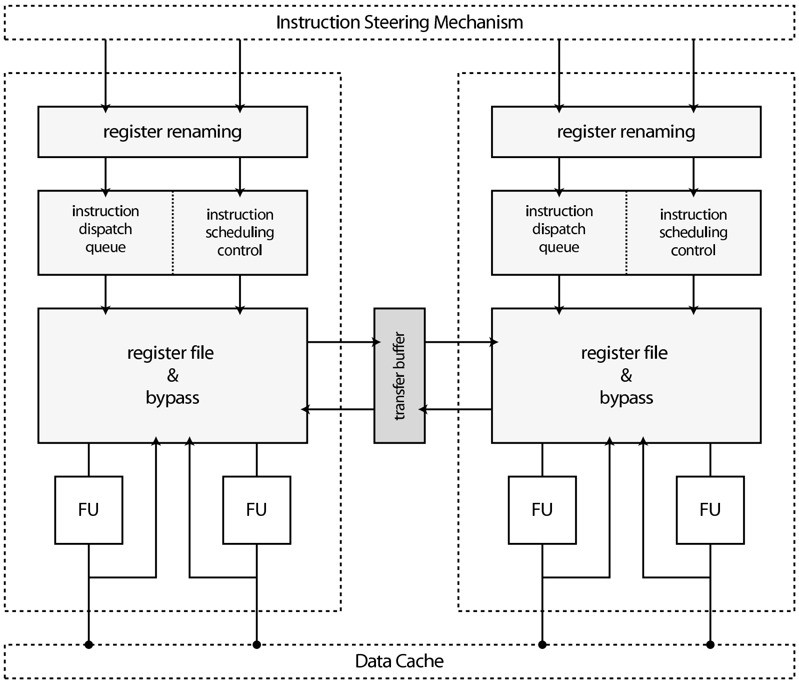


FIG. 1.28 – Microarchitecture clusterisée (Farkas et al.[30])

du cluster distant. L'utilisation d'un réseau d'interconnexion complet [86] implique l'intégration de 4 ports en écriture supplémentaires par fichier de registres. Le nombre de ports en écriture est donc équivalent à celui architecturé par un fichier de registres monolithique. En revanche, le partitionnement du fichier de registres permet de diviser le nombre de ports en lecture par 2 ; ce qui peut permettre de réduire le temps d'accès à cette structure.

Après renommage, les instructions sont assignées à un cluster selon la localisation de leurs opérandes. Soit I, l'instruction à assigner. L'heuristique considérée par Palacharla est la suivante :

- *Tous les opérandes de I résident dans le fichier de registres.* Assigner I à une FIFO sélectionnée parmi une liste de FIFOs libres.
- *I est en attente d'un opérande produit par une instruction assignée à la FIFO Fa.* Assigner I à la FIFO *Fa* si aucune instruction dépendante n'a déjà été assignée à *Fa*, sinon assigner I à une nouvelle FIFO.
- *I est en attente de deux opérandes produits par une instruction I1 et I2, résidant dans Fa et Fb respectivement.* Appliquer la règle précédente à l'instruction I1 ou à l'instruction I2 en cas d'échec.

1.5.2 Approche de Farkas [30]

Farkas et al. [30] ont proposé un schéma de partitionnement similaire à celui de Palarcharla [65] mais dans lequel le fichier de registres est distribué entre les clusters. Ce qui signifie que le résultat d'une instruction séquencée dans un cluster donné sera écrit dans le fichier de registres local à ce dernier ; chaque résultat est donc associé à un seul registre physique. Cette technique permet de réduire à la fois le nombre de registres physiques et le nombre de ports d'accès car les écritures ne doivent pas être systématiquement propagées. Néanmoins, distribuer le fichier de registres peut avoir des conséquences négatives sur les performances. Lorsqu'une instruction nécessite un opérande distant, ce dernier doit en effet être explicitement communiqué par le matériel. Les auteurs proposent alors d'exécuter ce type d'instructions sur les deux clusters. Un cluster effectue ainsi le calcul alors que l'autre communique la donnée grâce à une mémoire spécialisée comme illustré en figure 1.28 (*transfer buffer*). La tâche de chaque cluster est déterminée par l'affiliation des opérandes sources et du résultat à produire. Au niveau de la distribution des instructions, les auteurs suggèrent une approche statique. L'heuristique proposée prend ainsi en compte les dépendances de données tout en ayant une estimation de la répartition dynamique des instructions grâce à un profil des applications. Les résultats expérimentaux révèlent que l'équilibrage des charges est très sensible à la génération du code.

1.5.3 Modèle WSRS[5] [82]

Dans [82], nous avons examiné le modèle WSRS : une approche intermédiaire entre celle de Palacharla [65] et Farkas [30]. Le modèle WSRS partitionne le cœur d'exécution entre plusieurs clusters homogènes ; chacun d'eux implémentant une copie du fichier de registres. Notre modèle se distingue toutefois des approches utilisant plusieurs copies du fichier de registres [65, 44] en proposant la réduction du nombre de ports d'accès en écriture et en lecture sur cette structure. Pour ce faire, nous proposons le recours aux *spécialisations en écriture et en lecture* pour chaque cluster d'unités fonctionnelles.

La spécialisation en écriture consiste à limiter la propagation des résultats au fichier de registres associé au cluster producteur. En limitant ainsi les interconnexions, il est possible de réduire de manière proportionnelle le nombre de ports d'accès en écriture sur le fichier de registres. Pour un modèle à 4 clusters où chaque cluster peut produire 3 résultats par cycle, chaque copie du fichier de registres peut être implémentée avec 3 ports en écriture au lieu des 12 ports suscités par un modèle clusterisé classique (e.g. Alpha 21264 [44]). La spécialisation en écriture peut toutefois mener à une situation d'interblocage mais peut être aisément résolue en augmentant le nombre de registres physiques par copie. La spécialisation en lecture consiste à restreindre les clusters à lire leurs opérandes depuis un sous-ensemble prédéfini des copies du fichier de registres. Elle permet de réduire ainsi le nombre d'interconnexions et le nombre de ports en lecture pour chaque cluster.

[5] Write Specialization Read Specialization

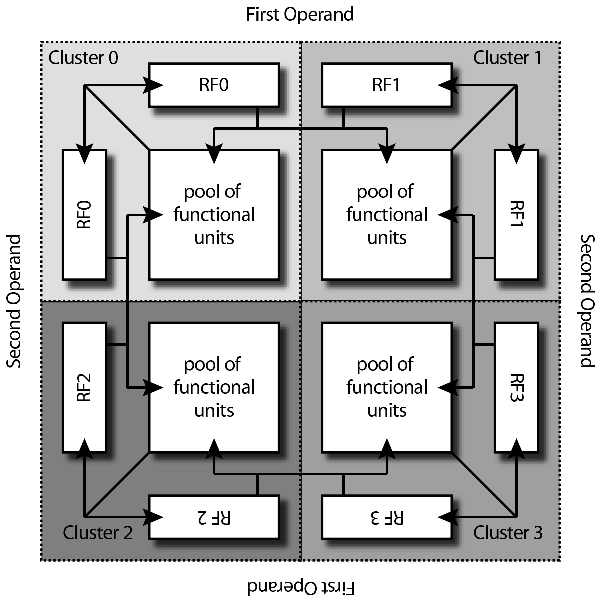


FIG. 1.29 – Modèle WSRS à 4 clusters [82].

La figure 1.29 présente une implémentation à 4 clusters du modèle WSRS. Chaque cluster est connecté en lecture/écriture au fichier de registres local et est connecté en lecture aux fichiers de registres associés aux clusters adjacents. Pour réduire davantage la complexité du matériel, nous proposons de limiter les accès en lecture en fonction de la disposition des opérandes. La figure 1.29 illustre ce concept. Dans notre étude, nous avons évalué la performance de plusieurs stratégies d'allocation matérielles, simples de mise en œuvre, pour distribuer les instructions vers les clusters. Les résultats expérimentaux révèlent des performances proches de celles déployées par un modèle clusterisé conventionnel. Le modèle WSRS réduit toutefois la complexité de plusieurs composants critiques : la logique de réveil, le réseau de *bypass* de données et le fichier de registres.

1.5.4 Approche de Gonzalez [35]

Gonzalez et al. [35] ont récemment examiné un schéma de partitionnement basé sur la dimension des données et des opérations manipulées à l'exécution. Cette étude présente certaines similitudes avec la proposition décrite au chapitre 3 mais se positionne toutefois dans une optique opposée : celle des performances. Pour ce faire, ces auteurs proposent une organisation asymétrique du processeur dans laquelle le cœur d'exécution entier est distribué entre deux clusters : (1) *un cluster 64-bit classique composé de deux ALUs* et (2) *un cluster 20-bit composé d'une seule ALU mais cadencé deux fois plus vite*. Il est notamment possible de cadencer ainsi le cluster 20-bit car celui-ci est moins complexe au niveau des composants distribués : fichier de registres, *bypass*, fenêtre d'instructions. Les performances du modèle proposé par [35] sont donc étroitement liées au nombre d'instructions assignées au cluster 20-bit de part sa fréquence de cadencement. Les auteurs proposent d'utiliser un mécanisme d'allocation des instructions similaire à celui de notre proposition mais relativement plus basique et ne distinguant pas les différents types de mauvaises prédictions de largeur. Contrairement à notre approche, les auteurs montrent toutefois que les calculs d'adresses peuvent être aussi effectués dans le cluster 20-bit en considérant les adresses aux bits de poids forts invariants [34]. Ceci permet au cluster 20-bit de traiter en moyenne 75% des instructions du programme (au lieu des 40-45% dans notre approche) mais en contrepartie d'un mauvais équilibrage des charges. Malgré que ce modèle puisse améliorer les performances (3% par rapport à un modèle monolithique et 8% par rapport à un modèle clusterisé classique), il est toutefois plus difficile de quantifier son impact sur la complexité du processeur. Cadencer deux fois plus rapidement le cluster 20-bit implique en effet une logique plus complexe et influe de manière proportionnelle sur la consommation électrique. De plus, plusieurs artefacts complexes sont proposés pour augmenter les performances (mécanisme de recouvrement, TLB, ...) mais aucune étude ne détaille leur complexité matérielle et/ou leur consommation d'énergie. Par ailleurs, il est également incertain que les performances de leur modèle soient équivalentes pour des applications avec un IPC supérieur car le mauvais équilibrage des charges implique une sous-utilisation de la largeur d'exécution. À la différence, le chapitre 3 expose un schéma de partitionnement exploitant les données tronquées pour réduire la complexité du processeur. Une analyse détaillée démontre la faisabilité de notre schéma avec une faible dégradation des performances. Par surcroît, la complexité de notre modèle permet aussi d'utiliser les hypothèses exposées dans [35] pour cadencer deux fois plus vite le cluster 16-bit et ainsi obtenir des gains substantiels en performance.

1.6 Conclusions

Plusieurs facteurs contraignent la conception des processeurs modernes. Ce chapitre en a exploré certaines notions particulières à ce document. Nous avons ainsi déterminé l'origine et les différentes sources de consommation liées à la technologie de fabrication des processeurs contemporains. Plusieurs solutions basiques pour résoudre ce problème de consommation ont été décrites et seront abordées dans les chapitres suivants. Nous avons également détaillé la structure et la complexité des composants critiques du chemin de données ciblés par nos propositions. Ces composants regroupent notamment le fichier de registres, le réseau de *bypass* des données, les logiques de séquencement et les interconnexions. Nous avons aussi exposé l'état de l'art des recherches relatives à nos études, c-à-d. l'exploitation des données tronquées pour réduire la complexité du processeur. Cet état de l'art discerne les études ayant proposé des mécanismes pour identifier la taille des données de celles ayant détaillé un support matériel permettant d'exploiter cette propriété des programmes. Le chapitre 3 propose le recours au partitionnement des ressources matérielles pour réduire la complexité du processeur. Ce chapitre a introduit les notions nécessaires pour appréhender ce modèle microarchitectural. Le chapitre suivant décrit la technique logicielle/matérielle proposée dans [71] pour réduire les consommations dynamique et statique du chemin de données dans un processeur embarqué.

Chapitre 2

Redimensionnement spéculatif du chemin de données pour réduire la consommation d'énergie *

2.1 Introduction

Les systèmes embarqués modernes intègrent de plus en plus de nouvelles fonctionnalités suscitant une puissance de calcul considérable. Cette tendance résulte notamment de la forte demande en applications multimédia sollicitant des ressources de calcul toujours plus importantes : e.g. *compression/décompression vidéo ou reconnaissance vocale*. Plusieurs facteurs viennent cependant contraindre la conception de ces systèmes. Parmi ces facteurs, la consommation d'énergie constitue l'aspect le plus critique car elle borne implicitement le nombre de fonctionnalités et la durée de fonctionnement des systèmes fonctionnant par batteries. Il devient donc primordial pour les systèmes embarqués modernes de maximiser le meilleur compromis entre la performance et la consommation. Plusieurs techniques [87, 62] ont été envisagées pour parvenir à un tel effet. Dans la technologie actuelle, un composant dissipe une puissance significative lorsqu'il est sollicité. Une technique classique consiste alors à le désactiver quand il est inutilisé pour économiser l'énergie associée. Dans une implémentation conventionnelle, la logique de mise en veille (*clock-gating*) agit au niveau de l'opération. Ainsi, seules les unités nécessaires au calcul de l'opération sont activées à un moment donné de l'exécution.

L'application de cette technique a été étudiée à différentes granularités. Manne et al. [58] ont proposé de réduire l'impact dû aux mauvaises spéculations en désactivant certaines portions du pipeline. Brooks et Martonosi [12] ont pris en compte une granularité plus fine pour optimiser l'activité du processeur. Ces auteurs considèrent la dimension

*Publié à **LCTES 2004** : « *Speculative Software Management of Datapath-Width for Energy Optimization* »[71].

des opérandes pour désactiver les portions des unités fonctionnelles non nécessaires à l'exécution. Ces auteurs ont en effet observé qu'une fraction importante des données dynamiques, manipulées par les programmes entier et multimédia, ne comportent qu'un nombre restreint de bits significatifs (inférieur à 16 bits). Ceci est cohérent car la plupart des applications de ce type s'exécutent avec des données de faible précision (e.g. encodage audio au format CD sur 16 bits). Les microarchitectures modernes ont toutefois tendance à implémenter un chemin de données plus large pour satisfaire les besoins en adressage. La consommation des divers éléments du chemin de données est directement dépendante de cette largeur. Optimiser l'usage du chemin de données pourrait donc avoir un impact significatif sur la consommation.

Divers travaux annexes [17, 55, 57, 85, 64] se sont appuyés sur la propriété observée par Brooks. Ces derniers ont principalement exploité cette propriété pour réduire la consommation d'énergie ou augmenter les performances du processeur. Ce document se focalisera sur les travaux ayant privilégié la réduction d'énergie. Les solutions proposées à cet égard peuvent être différenciées selon la stratégie adoptée pour détecter la dimension des données. La littérature regroupe deux catégories distinctes. La première catégorie se compose des solutions basées sur une analyse statique pour inférer la dimension maximale des données : *les approches logicielles* [57, 85, 17, 18]. La seconde catégorie rassemble les solutions reposant sur un mécanisme dynamique pour ce faire : *les approches matérielles* [12, 17, 55].

Les approches logicielles sont particulièrement adaptées aux systèmes embarqués car elles ne nécessitent aucun matériel particulier. Faute d'une connaissance du comportement dynamique du programme, ces approches peuvent néanmoins omettre un nombre conséquent d'opportunités. Les approches matérielles disposent de cette connaissance et peuvent donc déterminer avec précision la dimension des données. En revanche, ces dernières sollicitent une complexité matérielle difficilement envisageable dans le contexte des systèmes embarqués. Nous avons récemment proposé une approche alternative [71] mieux adaptée aux contraintes des systèmes embarqués. Notre schéma logiciel/matériel transgresse la nature conservatrice limitant les approches logicielles pour identifier une plus large fraction de données de faible dimension. Pour ce faire, nous déduisons le comportement des données manipulées par un programme à partir de ses exécutions passées. Notre approche est viable car la dimension des données possède la propriété de forte prédictabilité [55]. Si une opération produit un résultat sur 16 bits, il existe en effet une forte probabilité que sa prochaine instance dynamique produise un résultat de même dimension.

2.1.1 Proposition et contributions

Dans ce chapitre, nous montrons que les données de faible dimension sont fortement concentrées au sein des blocs de base. Nous proposons d'exploiter cette propriété pour réduire la consommation du processeur. Pour ce faire, nous proposons une méthodologie pour partitionner un programme en régions nécessitant une même largeur d'exécution. Pour établir ces régions, nous spéculons sur la dimension que nécessitent les opérations en s'aidant du profil des exécutions antérieures de l'application. L'insertion d'une instruction de reconfiguration permet d'exposer ce découpage logique au matériel. La nature spéculative de notre schéma impose toutefois le recours à un mécanisme dédié pour corriger des mauvaises prédictions. Nous considérons un mécanisme naïf mais requérant une faible complexité matérielle pour ce faire. Ce mécanisme est le facteur principal de dégradation des performances. Pour limiter cet impact, nous proposons diverses optimisations logicielles qui réduisent la fréquence d'appel au recouvrement grâce à un support adéquat du jeu d'instructions. Pour exploiter ce découpage logique des applications, nous décrivons un chemin de données redimensionnable à la largeur des régions. Ceci permet une réduction significative de la consommation d'énergie statique et dynamique. Plus particulièrement, nous introduisons un nouveau modèle pour fichier de registres qui est adéquat aux pipelines reconfigurables dynamiquement. Ce dernier peut s'adapter à la dimension du calcul tout en préservant les données stockées grâce au mode *drowsy* [32].

2.1.2 Organisation du chapitre

Ce chapitre est architecturé de la manière suivante. La section 2.2 motive notre approche en révélant l'usage prédominant des données tronquées dans les applications embarquées. La section 2.3 examine la distribution des opérations tronquées à l'exécution et évoque la présence de régions tronquées dans le programme. La section 2.4 présente une stratégie pour découper un programmes en régions requérant une même largeur d'exécution. La section 2.5 décrit plusieurs optimisations logicielles pour éviter le recouvrement de l'exécution. La section 2.6 détaille le support architecturé pour réduire la consommation du chemin de données. La section 2.7 présente l'évaluation expérimentale en termes de consommation, performance, et taille de code. La section 2.8 conclut ce chapitre en exposant les directions futures de ces travaux.

2.2 Motivations

Il a été récemment observé [12] que la moitié des données manipulées à l'exécution comportent un nombre restreint de bits significatifs. Cette propriété des programmes a été observée pour une machine 64 bits exécutant des applications du domaine entier (SPECint95) et multimédia (MediaBench). Nous avons procédé à des expériences complémentaires pour vérifier la validité de cette propriété dans le contexte des systèmes embarqués. La caractérisation a été effectuée à partir d'un sous-ensemble de programmes qui composent la suite PowerStone [80]. Ce sous-ensemble est plus amplement détaillé dans la section 2.7.3.

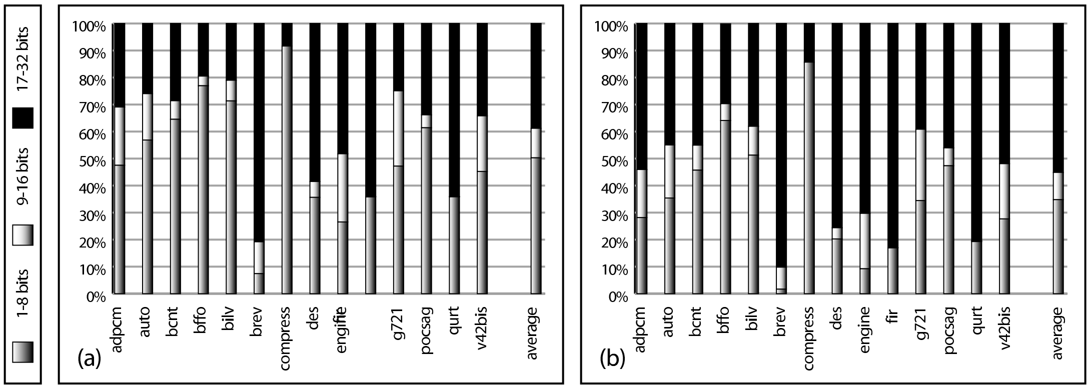


Fig. 2.1 – Distribution des opérandes (a) et des opérations (b) tronqués à l'exécution des applications Powerstone [80].

2.2.1 Quantification des données tronquées

Les figures 2.1 (a) et (b) affichent les résultats obtenus. La figure 2.1 (a) expose la proportion des opérandes manipulés dans les programmes en fonction du nombre de bits significatifs que comportent ces derniers ; ceci pour une dimension variant de 8 à 32 bits. La figure 2.1 (b) présente la proportion des opérations exécutées en fonction de la dimension maximale de ses opérandes. En moyenne, nous pouvons constater que la moitié des opérandes manipulés à l'exécution sont tronqués ; en considérant un nombre de bits significatifs inférieur à 16 bits. Nous pouvons également noter qu'environ 45% des opérations dynamiques exhibent le même comportement à l'exécution. Les mesures recueillies s'apparentent à celles obtenues par Brooks [12] pour une machine à usage général 64 bits. Par conséquent, ces résultats peuvent démontrer l'intérêt d'une approche exploitant les données tronquées pour réduire la consommation d'énergie. En fait, si le schéma de détection permet d'identifier la totalité des données tronquées, l'activité du chemin de données pourrait être théoriquement réduite de moitié.

2.3 Distribution dynamique des opérations tronquées

Le profilage des applications a révélé que la plupart des applications s'exécutent avec une forte proportion en opérations tronquées. La figure 2.1 montre ainsi que la moitié des données manipulées par le programme *adpcm* sont tronquées à l'exécution. Cette section examine la distribution des opérations tronquées au cours de l'exécution. Nous considérons la granularité du bloc de base pour ce faire.

La figure 2.2 affiche une tranche aléatoire de l'exécution du programme *adpcm*. Chaque point sur l'axe des abscisses représente une instance dynamique d'un bloc de base du programme. Chaque valeur pourvue sur l'axe des ordonnées représente la fraction en opérations tronquées présentes dans un bloc de base. Nous pouvons noter que la majorité des blocs de base exécutés se composent en moyenne d'une fraction

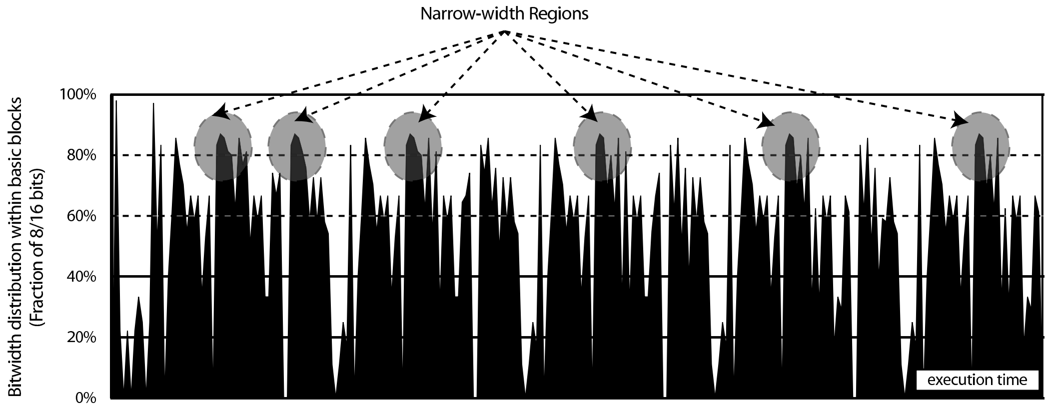


FIG. 2.2 – Répartition dynamique des opérations tronquées par bloc de base pour le programme *adpcm*.

en opérations tronquées supérieure à 60%. Ceci peut dénoter l'existence d'une forte localité des opérations tronquées au sein des blocs de base du programme. Ceci peut donc suggérer d'accommoder la largeur du chemin de données à la dimension de ces blocs de base. Ce redimensionnement permettrait de réduire de façon significative la consommation du chemin de données.

2.3.1 Stabilité des régions

La figure 2.2 suggère de considérer la granularité du bloc de base pour accommoder la dimension du chemin de données. Considérer une telle granularité permet de désactiver les structures du processeur sur une période relativement longue pour réduire les impacts sous-jacents au *clock-gating* [74]. La variation du comportement des données exécutées dans les blocs de base doit toutefois être analysée au préalable. L'intérêt de cette analyse est de prouver la stabilité des régions tronquées pour éviter un appel abusif au recouvrement. La stabilité moyenne des régions tronquées exécutées a été évaluée comme suit. Soit un seuil arbitraire en opérations manipulant des opérandes tronquées. Pour chaque bloc de base exécuté, nous avons calculé le nombre de fois où ce seuil est maintenu. Une moyenne est ensuite établie pour les blocs de base affichant le même comportement. La figure 2.3 présente les résultats obtenus. Cette figure expose la variation moyenne de la dimension des données exécutées dans les blocs de base du programme. Une valeur haute indique la stabilité des régions au cours de l'exécution. Ainsi, la figure 2.3 indique que la plupart des programmes exhibent des régions tronquées stables à l'exécution. Ces résultats sont valides pour un seuil fixé à 80%. En moyenne, 90% des blocs de base ne fluctuent pas au cours de l'exécution, mis à part pour *qurt*. Ceci indique que la formation statique des régions tronquées peut être appliquée car celles-ci exhibent un comportement dynamique stable.

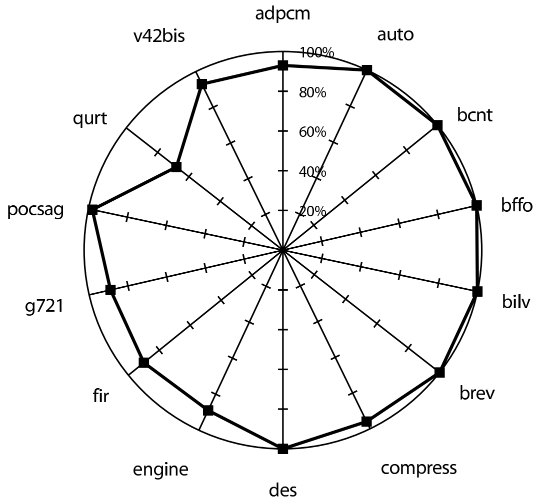


FIG. 2.3 – Stabilité moyenne des régions tronquées.

2.4 Partitionnement du programme en régions tronquées

La section 2.3 a mis en avant la propriété de forte concentration dynamique des opérations tronquées au cœur des blocs de base du programme. De plus, il a été montré que plusieurs blocs de base consécutifs dans le code, définissant la notion de région, affichent un comportement similaire à l'exécution. Cette section propose une stratégie pour découper un programme en régions requérant une même largeur d'exécution. Par convention, une région s'exécutant avec une forte proportion en opérations tronquées sera désignée comme *région tronquée*.

2.4.1 Sélection des régions tronquées

La stratégie adoptée pour extraire les régions tronquées du programme consiste à analyser son comportement antérieur. Néanmoins, la figure 2.2 révèle que peu d'instances dynamiques des blocs de base affichent une proportion absolue en opérations tronquées. Nous proposons de compenser cette insuffisance en considérant les régions du programme s'exécutant avec une proportion dominante en opérations tronquées. Pour ce faire, nous définissons deux seuils fixés à 60 et 80% pour qualifier une région tronquée. Remarquons que la figure 2.2 corrobore à ces choix. Considérer de telles régions peut toutefois impliquer un recours plus fréquent au mécanisme de recouvrement. Notons que si le seuil est réduit alors la probabilité de se tromper est supérieure. Les seuils choisis correspondent respectivement à 1 et 2 opérations sur 5 sollicitant un recouvrement. L'impact de ces choix sur les performances est discuté en section 2.7.5.3.

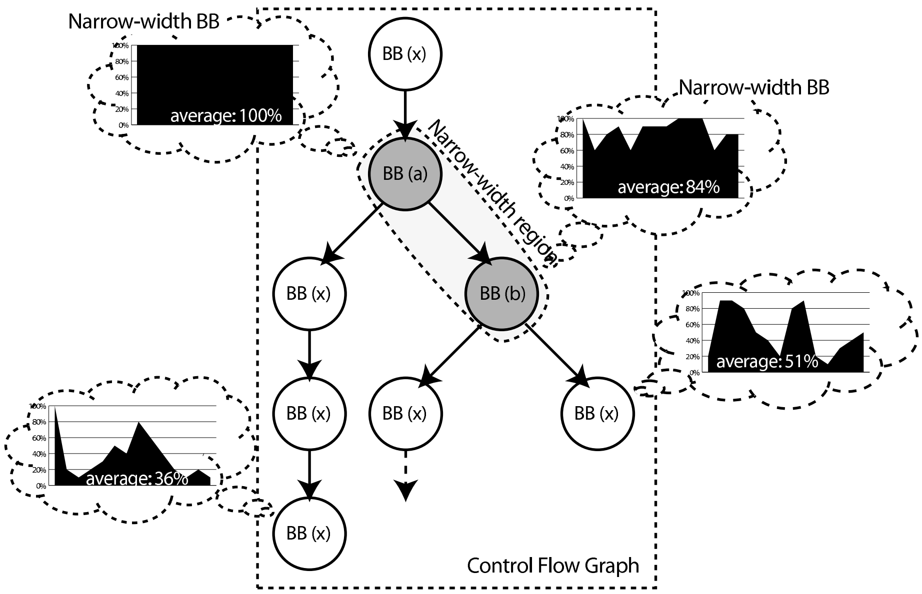


FIG. 2.4 – Principe de l'algorithme de sélection ; chaque bloc de base est pondéré par son comportement tronqué.

2.4.2 Méthodologie

La figure 2.4 expose la méthodologie proposée pour former les régions tronquées. Un seuil fixé à 80% est considéré. La répartition dynamique des opérations tronquées est représentée pour chaque bloc de base. Nous pouvons remarquer que le bloc de base *BB(a)* traite exclusivement des opérations tronquées pour le jeu de données considéré. D'autre part, le bloc de base *BB(b)* affiche un taux en opérations tronquées supérieur au seuil fixé. Par conséquent, ce bloc de base sera traité comme une région tronquée. Les blocs de base ne satisfaisant le seuil requis ne sont pas pris en considération.

2.4.3 Participation des régions tronquées à l'exécution

La figure 2.5 affiche la proportion des régions tronquées qui participent à l'exécution ; ceci pour une valeur de τ fixée à 80%. Nous remarquons que les régions prédites tronquées participent en moyenne au tiers de l'exécution ($\sim 30\%$). Les applications *compress* ou *bilv* exhibent notamment un comportement fortement tronqué ($> 80\%$). Inversement, les applications *brev* ou *bcnt* exhibent un comportement faiblement tronqué ($< 5\%$). Ceci peut suggérer, d'une part, que l'on peut augmenter la valeur de τ pour les applications *compress* ou *bilv* afin d'obtenir un compromis différent en terme de consommation/performance. Ceci suggère d'autre part qu'il faut diminuer la valeur de τ pour les applications *brev* ou *bcnt* afin d'obtenir un pondération dynamique acceptable. À noter qu'un axe futur de recherches consistera à ajuster la valeur de τ selon le meilleur compromis consommation/performance. À cet effet, il est possible de faire

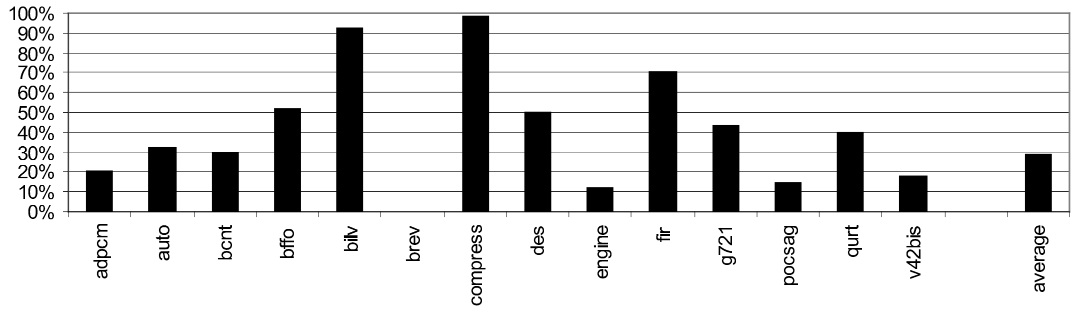


FIG. 2.5 – Participation des régions tronquées à l'exécution pour $\tau = 80\%$.

varier le produit énergie-délai [33] en fonction de la proportion dynamique des régions tronquées pour obtenir une valeur plus optimale de τ.

2.5 Transformation des régions tronquées

Cette section propose deux méthodes pour réduire la fréquence d'appel au mécanisme de recouvrement. D'une part, nous proposons de découpler le calcul d'adresses de l'accès mémoire. D'autre part, nous proposons de déplacer les opérations 32 bits présentes dans les régions tronquées vers des régions voisines.

2.5.1 Composition interne des régions tronquées

Nous avons considéré les régions composées d'une fraction élevée en opérations tronquées pour renforcer le nombre de régions candidates. La figure 2.6 caractérise la composition moyenne des régions exécutées pour un seuil fixé à 80%. Le graphe représente la proportion en opérations tronquées, en opérations manipulant des adresses mémoires et en opérations requérant 32 bits. Nous pouvons ainsi noter que ce sont les instructions accédant à la mémoire qui requièrent toute la largeur du chemin de données. Cette observation suggère de découpler le calcul d'adresses du processus de transfert de données pour éviter le recours au recouvrement. Nous proposons un support microarchitectural et logiciel adéquats pour ce faire.

2.5.2 Découplage du calcul d'adresses de l'accès mémoire

Dans une microarchitecture RISC, seules les instructions de type load/store peuvent accéder à la mémoire. Nous supposons que ces dernières sont découpées explicitement en un calcul d'adresses suivi par l'accès aux données. Pour ce faire, de manière informelle, une instruction *load/store* manipulant des opérandes 32 bits est découpée

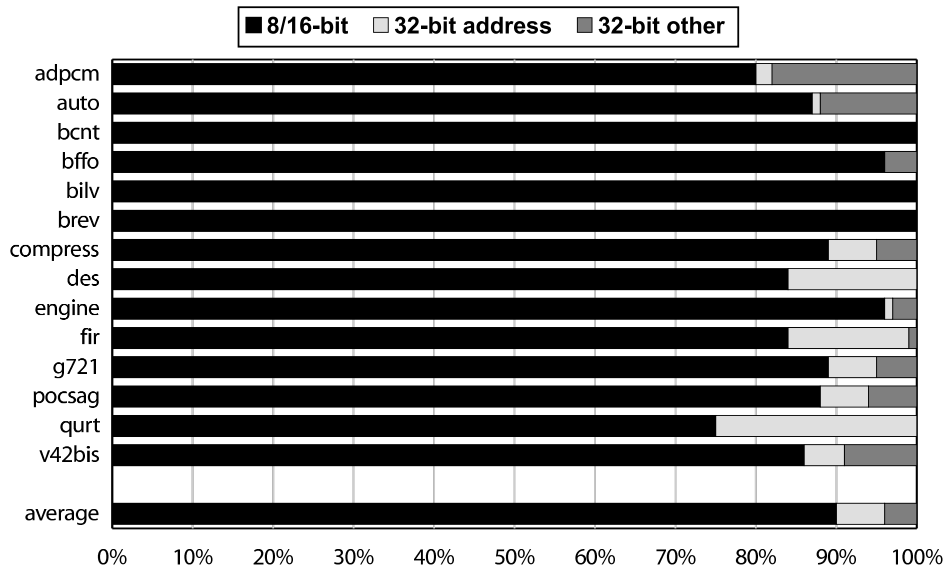


FIG. 2.6 – Composition interne des régions tronquées.

en deux opérations. La première opération permet d'effectuer le calcul d'adresses selon le type d'adressage considéré. À noter que cette opération peut être elle-même scindée en plusieurs micro-opérations et qu'un nombre trop élevé de micro-opérations impacte de manière néfaste sur la taille de code. Pour réduire cet impact, nous avons considéré une heuristique simple mais permettant d'obtenir un compromis acceptable en performance/consommation/taille de code. La seconde opération permet d'effectuer le chargement ou le stockage des données en mémoire.

2.5.2.1 Partitionnement du graphe de flot de données (DFG)

La substitution des opérations *load/store* par la série équivalente de micro-opérations peut être apparentée à un problème de partitionnement de graphe. Soit le graphe direct acyclique $G < N, A >$ correspondant au graphe de flot de données (DFG). Un nœud n de G représente une opération du bloc de base B. Deux nœuds n et n' de G sont connexes via un arc a s'il existe une dépendance de données entre ces opérations. Le partitionnement du DFG consiste à déterminer, pour chaque nœud *load/store* manipulant des adresses 32 bits, l'ensemble de nœuds N' réalisant le calcul d'adresses (cf. section 2.6.1.2) tout en préservant une taille de code acceptable. Pour ce faire, nous proposons une heuristique relativement naïve qui considère à la fois l'impact de l'expansion du code sur les performances et sur la consommation. De manière plus précise, pour chaque région tronquée, notre algorithme décide si l'expansion du code est acceptable en fonction de la fréquence d'apparition de la région. Dans le cas contraire, la région n'est pas traitée. Intuitivement, l'heuristique proposée permet de se concentrer

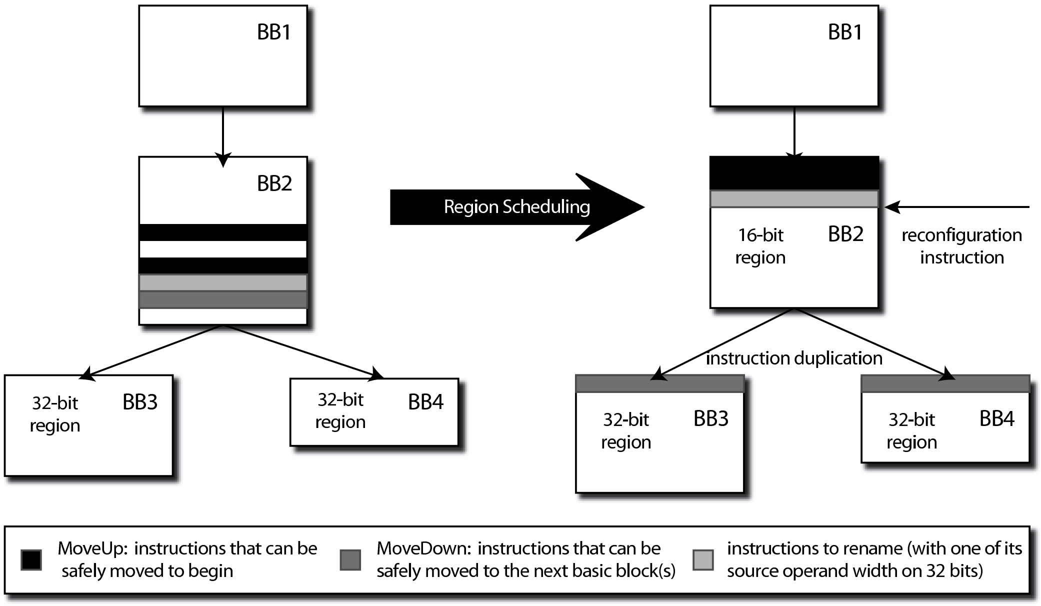


Fig. 2.7 – Restructuration du code assembleur.

sur les régions les plus fréquemment exécutées et, par extension, peut ainsi permettre de réduire la fréquence d'appel au mécanisme de recouvrement.

2.5.3 Restructuration du code assembleur

Dans la figure 2.6, il apparaît également que des opérations différentes de celles accédant à la mémoire manipulent des opérandes sur 32 bits. Cette section propose une technique pour restructurer le code assembleur afin de décharger les régions tronquées de ces opérations 32 bits. Informellement, les opérations 32 bits qui appartiennent à la région tronquée r doivent être déplacées vers la région 32 bits r' la plus proche, sans violer les dépendances originales. La figure 2.7 illustre mieux le concept décrit ci-après.

2.5.3.1 Algorithme de réordonnancement

L'algorithme de réordonnancement est composé de trois étapes. La première étape consiste à renommer tous les opérandes résultat des instructions possédant au moins un opérande source sur 32 bits et pouvant être utilisées antérieurement. De cette manière, le nombre de candidats 32 bits est augmenté grâce à la suppression des fausses dépendances de données. La seconde étape consiste à former les ensembles *MoveUp* et *MoveDown*. Ces ensembles correspondent respectivement aux instructions 32 bits qui peuvent être déplacées, sans violer les dépendances originales, vers le début de la région tronquée et vers les régions 32 bits successeurs. La dernière étape consiste à déplacer les opérations 32 bits des ensembles *MoveUp* et *MoveDown* dans les régions concernées.

Il est à remarquer que cet algorithme est susceptible d'augmenter la taille du code. Les instructions 32 bits doivent effectivement être dupliquées dans tous les blocs de base successeurs. Ceci est illustré dans la figure 2.7. Une autre conséquence se situe au niveau de l'allocation de registres. Les instructions déplacées exercent en effet une pression supplémentaire sur le fichier de registres. Au pire des cas, du *spill code* peut apparaître. Néanmoins, ce dernier peut être évité en considérant un nombre de registres important comme supposé dans cette étude.

2.6 Microarchitecture proposée

Cette section décrit le support matériel envisagé pour réduire la consommation du processeur par le biais des opérations tronquées. De manière similaire à [12], nous supposons, pour ce faire, une microarchitecture reconfigurable qui peut s'adapter dynamiquement à la largeur des opérations traitées. À la différence de [12], le schéma proposé ne se restreint pas exclusivement aux unités fonctionnelles et s'applique aux autres éléments du chemin de données ; c-à-d. fichier de registres et logique de *bypass*. Dans la section 2.4, nous avons proposé de découper statiquement les applications en régions tronquées. Cette section présente le modèle adopté pour réaliser un tel partitionnement. Une instruction de reconfiguration permet notamment d'exposer au matériel nos prédictions logicielles. Dans le cas d'une mauvaise spéculation, un mécanisme de recouvrement permet de réadapter la largeur du chemin de données à une dimension correcte. Étant donné le contexte de cette étude, le mécanisme de recouvrement est limité à une mise en œuvre matérielle relativement simpliste. Une autre contribution originale de cette étude réside dans l'introduction d'un nouveau modèle de fichier de registres ; c-à-d. le fichier de registres en tranches (*byte-slice register file*). Un tel fichier de registres peut optimiser l'activité et la consommation statique d'une telle structure tout en préservant les données préalablement stockées.

2.6.1 Support du jeu d'instructions (ISA)

Cette section détaille le support requis au niveau du jeu d'instructions pour notifier au matériel nos optimisations logicielles. Nous supposons que ceci est strictement réalisable si le jeu d'instructions cible dispose d'un nombre suffisant en code opératoires libres. À vrai dire, l'ajout d'une seule instruction est impérative au fonctionnement de notre schéma ; c-à-d. l'instruction de reconfiguration. Les autres instructions peuvent être omises mais sont toutefois à considérer car elles permettent de limiter l'appel au mécanisme de recouvrement.

2.6.1.1 Instruction de reconfiguration

Dans notre contexte, l'utilisation d'une instruction de reconfiguration est recommandée pour spécifier au matériel le découpage logiciel. Cette approche est en effet re-

Instruction syntax	**Instruction semantic**
MOVACC Rd	ACC $\rightarrow$ REG[Rd]
MOVREG Rs	REG[Rs] $\rightarrow$ ACC
OPACC Rt	OP((ACC), (REG[Rt])) $\rightarrow$ ACC

TAB. 2.1 – Opérations pour le calcul d'adresses

lativement simple à mettre en œuvre, à la fois, au niveau de la microarchitecture et au niveau du compilateur. Le rôle de cette instruction consiste à accommoder, au décodage, la largeur du chemin de données à celle spécifiée par l'utilisateur. Nous supposons, à cet égard, qu'un registre de statut est conformément positionné pour que la largeur courante du pipeline soit conservée. Les instructions exécutées subséquemment seront alors exécutées avec cette dimension. Lorsqu'une nouvelle instruction de reconfiguration est décodée et spécifie une nouvelle dimension, le pipeline est alors conformément reconfiguré ; autrement, cette instruction n'affecte pas la largeur courante.

2.6.1.2 Instructions pour le calcul d'adresses

Comme détaillé en section 2.5.1, les instructions, requérant typiquement toute la largeur d'exécution dans les régions tronquées, sont celles manipulant des adresses. Afin d'écarter de telles instructions des régions tronquées et ainsi obtenir un meilleur degré de prédiction, nous proposons de découpler le calcul d'adresses de l'accès mémoire. À cet effet, nous proposons de réaliser le calcul d'adresses dans un additionneur spécialisé 32 bits relié à un petit groupe d'accumulateurs. Nous supposons que cet additionneur est présent au sein de l'unité mémoire et peut être désactivé en cas d'inutilisation. À noter que l'on aurait pu également spécialiser un banc de registres à cet effet. Néanmoins, la première solution semble plus adaptée car elle sollicite une complexité matérielle moindre. Dans [42], Jain et al. usent également d'un tel mécanisme pour réduire la pression sur le fichier de registres. Afin de pouvoir manipuler les différents modes d'adressage, le jeu d'instructions original doit être augmenté par de nouvelles opérations ayant accès à l'accumulateur. Le tableau 2.1 présente les extensions possibles permettant de manipuler une gamme d'adressages RISC. Les opérations MOV{ACC, REG} permettent notamment de transférer une valeur depuis l'accumulateur vers les registres. L'opération OPACC permet de manipuler les adresses présentes dans l'accumulateur. Par exemple, pour l'adressage indirect, l'opérateur OP sera une addition.

2.6.2 Fichier de registres en tranches

Les propositions de la littérature exploitent principalement les opérations tronquées pour réduire l'activité des unités de calcul. Notre proposons d'appliquer ces dernières pour optimiser l'activité du fichier de registres. L'intérêt de notre proposition est considérable car une forte quantité de l'énergie est dissipée par cette structure [91]. Canal et al. [17] ont récemment considéré un schéma similaire.

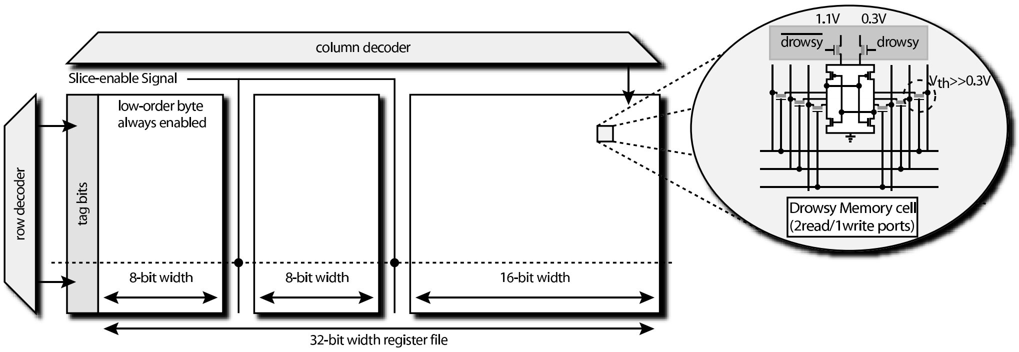


FIG. 2.8 – Fichier de registres en tranches (*byte-slice register file* [71]).

2.6.2.1 Travaux relatifs

La section 1.4.1.2 présente la microarchitecture proposée par Canal et al. [17]. Pour réduire la consommation du fichier de registres, ces auteurs considèrent une structure 32-bit scindée en quatre bancs de 8-bit. L'activité du fichier de registres est ainsi réduite car un banc est accédé une seule fois par cycle. Ce qui est imposé par la largeur fixe du chemin de données. Cette implémentation impose néanmoins le maintien en activité des bancs de registres pour préserver l'état des données. Cette approche peut donc présenter un intérêt limité pour une microarchitecture redimensionnable dynamiquement. Plusieurs bancs peuvent être en effet accédés dans une telle microarchitecture pour lire ou écrire les données. Considérer une structure multi-bancs nécessite aussi la duplication des décodeurs d'adresses pour chaque banc. Ce qui peut alors engendrer un décodage redondant de la même adresse. La surface de silicium, la consommation et le temps d'accès au fichier de registres peuvent alors être augmentés.

2.6.3 Modèle proposé

Le modèle proposé est illustré en figure 2.8. Le fichier de registres en tranches (*byte-slice register file*) peut être vu comme une structure conventionnelle de largeur 8, 16 ou 32 bits à un instant donné de l'exécution. Du point de vue logique, le fichier de registres est scindé en trois tranches de largeurs respectives : 8, 8 et 16 bits. La tranche de poids faible 8 bits (représentée à gauche sur la figure 2.8) est toujours alimentée. Les deux autres tranches peuvent être désactivées grâce au signal *slice-enable*. Étant donné que les registres peuvent contenir des informations de largeur supérieure à 8 bits, il est indispensable de préserver de telles données afin de maintenir une exécution correcte. Une solution envisageable consiste à sauvegarder l'état des cellules à désactiver dans la mémoire. Cette solution peut toutefois impacter à la fois sur les performances et la consommation de manière substantielle. Une autre solution consiste à modifier les points mémoire comme proposé par Ayala et al. [3]. Ces auteurs décrivent une implémentation du mode *drowsy* [32] appliqué aux fichiers de registres. Une cellule en

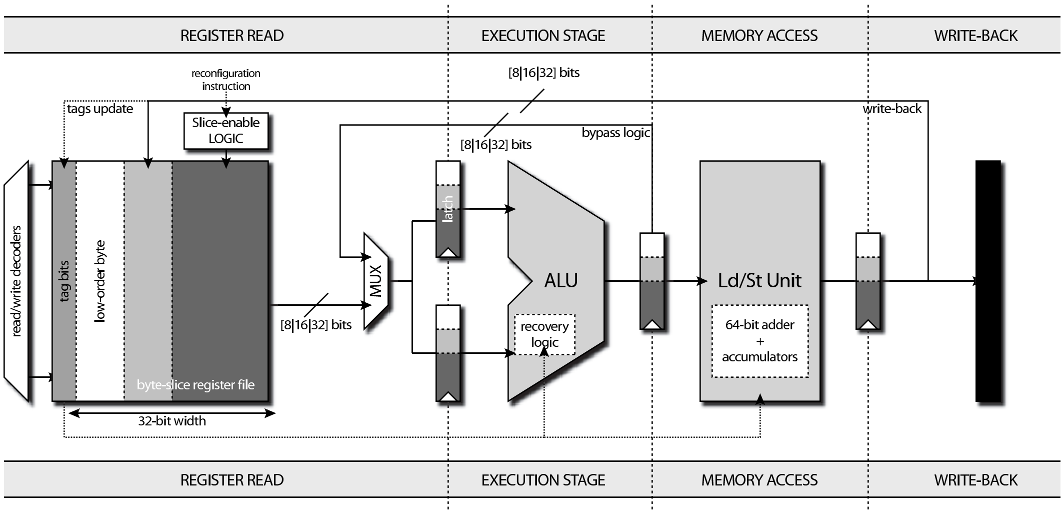


FIG. 2.9 – Chemin de données reconfigurable.

mode *drowsy* ou *sommeil* possède la capacité de préserver l'état des données tout en minimisant la consommation statique. Une cellule dans un tel état ne peut toutefois pas être accédée en lecture ou écriture. Du point de vue technique, le mode sommeil est réalisé par l'application de deux tensions d'alimentation : une tension faible et une tension classique. La tension faible permet notamment de maintenir l'état des données en minimisant les courants de fuite (cf. section 1.2.2.4). Les modifications nécessaires pour le mode sommeil sont présentées en figure 2.8.

2.6.4 Chemin de données reconfigurable

La figure 2.9 illustre le chemin de données proposé pour exploiter les régions tronquées afin de réduire la consommation d'énergie. Comme illustré sur ce synoptique, le chemin de données peut être redimensionné dynamiquement au moyen du signal *Slice-enable*. Ce signal est contrôlé par l'instruction de reconfiguration (cf. section 2.6.1.1). Le redimensionnement des verrous et des unités fonctionnelles se fait grâce à l'ajout de circuits de *clock-gating* comme proposé dans [12]. Dans une logique CMOS statique, le fait d'inhiber le signal aux verrous éliminent pratiquement la consommation en régime dynamique. Dans une logique CMOS dynamique (domino), le même effet peut être obtenu en inhibant le signal aux structures contrôlant les phases de précharge et d'évaluation du circuit [13]. Il est à noter que la granularité de contrôle des circuits de désactivation se fait sur une période assez longue. Un des futurs travaux de recherche est de mesurer l'impact de notre schéma sur le phénomène d'induction di/dt [74].

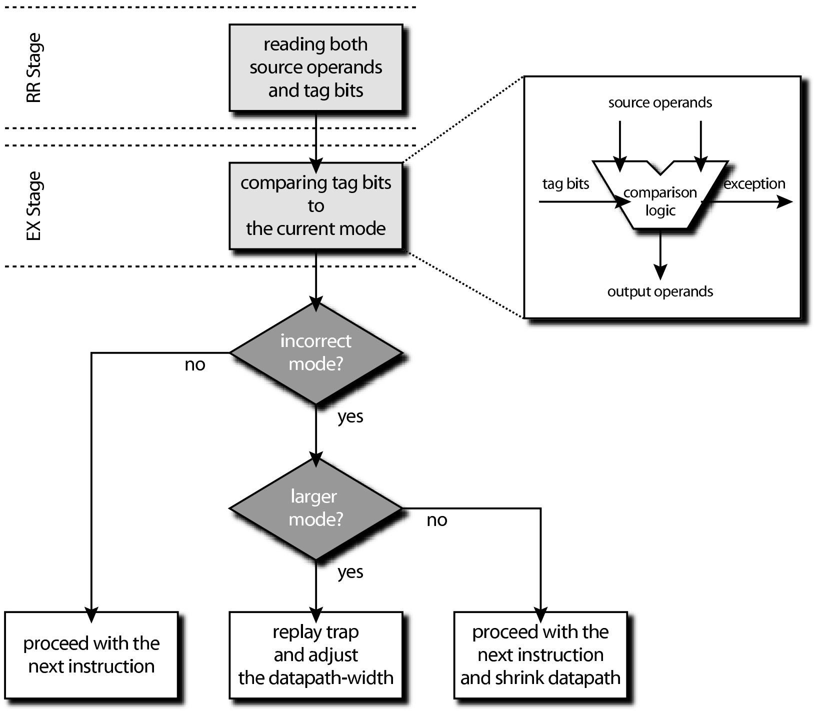


FIG. 2.10 – Principe du mécanisme de recouvrement.

2.6.5 Mécanisme de recouvrement

Le profilage d'une application permet de percevoir son comportement dynamique sans recours à un support matériel complexe. Néanmoins, le profil obtenu ne peut refléter précisément ce comportement dynamique car il est généralement établi avec un nombre limité de jeux de données. Ainsi, si le comportement dynamique des instructions dépend des entrées du programme, les régions tronquées formées peuvent alors être erronées. Nous avons utilisé plusieurs jeux de données pour écarter les régions très sensibles. Nous avons également considéré les régions comportant une large fraction d'opérations tronquées pour augmenter le nombre de régions tronquées. Ces régions tronquées sont davantage susceptibles de contenir des opérations 64-bit. Nous proposons un mécanisme matériel simple pour ramener le processeur dans un état correct lorsqu'une mauvaise prédiction est détectée.

2.6.5.1 Présentation du mécanisme

Le mécanisme de recouvrement proposé est relativement simple mais approprié aux systèmes embarqués de part sa simplicité de mise en œuvre matérielle. Le principe de ce mécanisme est de capturer la taille effective des données dans le fichier de registres à travers quelques bits. Nous considérons ainsi que chaque registre est étendu par 2 bits, désignés *tag bits*. Canal et al. [17] ont également utilisé des *tag bits* pour sérialiser l'exécution. À la différence, notre étude se base sur les *tag bits* pour identifier les mauvaises prédictions de largeur.

Génération des *tag bits* Les *tag bits* sont générés par les unités fonctionnelles lors d'une exécution ou par l'unité mémoire lors d'un chargement. Nous supposons que les unités fonctionnelles sont pourvues des fonctionnalités nécessaires pour ce faire [55]. Ce qui peut être notamment réalisé par la logique de génération des retenues. En ce qui concerne l'unité mémoire, nous considérons également qu'une fois la donnée chargée, les *tag bits* sont calculés avant de stocker les informations dans le fichier de registres. À noter que la latence induite par un tel schéma a été prise en compte dans l'évaluation expérimentale.

2.6.5.2 Principe de fonctionnement

L'organigramme illustré en figure 2.10 résume le principe de fonctionnement du mécanisme de recouvrement. Lorsqu'une instruction lit ses opérandes depuis le fichier de registres, les données et les *tag bits* sont envoyés à l'unité fonctionnelle. Lors de l'exécution, une logique de comparaison, agissant en parallèle, contrôle si le mode courant est correct. Dans le cas contraire, deux cas de figure peuvent se produire. Premièrement, le mode courant est plus large que celui prédit. Ceci n'a pas d'impact particulier sinon celui du repositionnement du registre de statut contenant la largeur courante du pipeline (cf. section 2.6.1.1). Deuxièmement, le mode courant est plus petit. Dans ce second cas, les instructions du pipeline doivent être exécutées à nouveau. En d'autres termes, l'exécution doit être arrêtée, le pipeline vidé, et le chemin de données réadapté à une largeur correcte.

2.7 Évaluation expérimentale

Cette section présente l'évaluation du modèle microarchitectural proposé dans cette étude. Nous décrivons, préalablement, l'infrastructure employée pour segmenter un programme en un ensemble de régions tronquées ainsi que les conditions expérimentales considérées. Nous introduisons ensuite un modèle analytique permettant d'approximer les apports en énergie suscités par notre schéma. L'évaluation de notre schéma est présentée selon trois aspects : (1) la réduction en énergie statique et dynamique du chemin de données, (2) la dégradation en performance due aux mauvaises prédictions et, (3) l'expansion de la taille du code résultant des optimisations logicielles.

2.7.1 Infrastructure logicielle

Une représentation du synoptique de l'infrastructure logicielle utilisée pour former les régions tronquées est exposée en figure 2.11. Cette infrastructure a été développée autour de la bibliothèque *Salto* [11] qui permet une manipulation générique du code

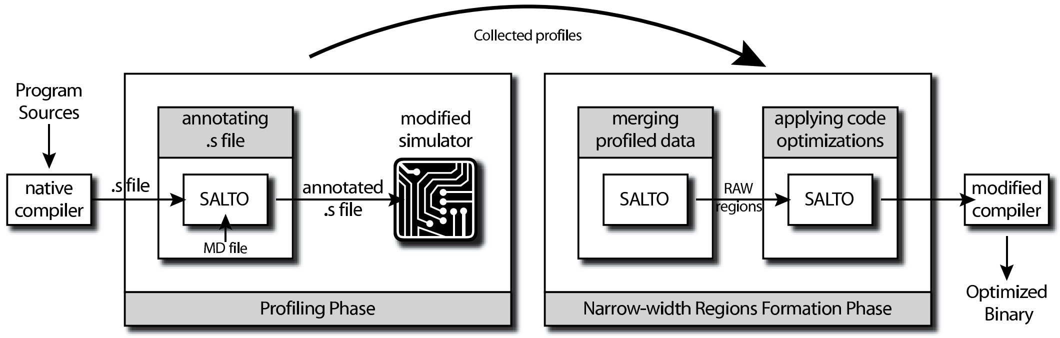


Fig. 2.11 – Méthodologie pour le partitionnement en régions tronquées.

assembleur. Deux phases distinctes peuvent être distinguées lors du partitionnement en régions tronquées. La première phase consiste à récolter le comportement dynamique des données : la *phase de profilage*. La seconde phase emploie les informations profilées pour construire les régions tronquées : *la phase de formation*.

2.7.1.1 Phase de profilage

L'étape antérieure au profilage consiste en l'assemblage des fichiers sources du programme à optimiser. Un outil de compilation industriel similaire à *gcc* a été utilisé à cet effet. Les fichiers assemblés sont soumis à Salto qui construit une représentation CFG du programme. Ce dernier instrumente alors chaque bloc de base pour leur attribuer une identité unique. Un outil de simulation spécifique recueille ainsi le comportement de ces derniers à l'exécution pour différents jeux de données. Plusieurs profils d'exécution sont ainsi obtenus.

2.7.1.2 Phase de formation

Les informations de ces profils sont intersectées de manière à produire un profil unique dont le contenu fortement sensible aux jeux de données est évincé. Salto établit ensuite la correspondance entre ce profil et les instructions du programme pour pouvoir annoter chaque bloc de base avec son comportement dynamique. Les blocs de base consécutifs qui présent un même comportement sont ainsi regroupés en régions. Le code assembleur des régions tronquées est ensuite restructuré comme explicité en section 2.5. Salto insère alors les instructions de reconfiguration pour exposer au matériel le découpage logique du programme. Un outil spécifique permet la génération du binaire optimisé qui sera émulé sur la plate-forme décrite ci-après.

Application	Description
adpcm	voice encoding/decoding
auto	automotive control code
bcnt	bit count
bffo	find first zero
bilv	shift, and, or operations
brev	bit reverse operations
compress	data compression
des	data encryption
engine	engine control application
fir	integer FIR filter
g721	protocol for voice transmission
pocsag	communication protocol for paging
qurt	root computation of a quadratic equation
v42bis	modem encoding/decoding

Tab. 2.2 – Descriptif des applications Powerstone sélectionnées [80].

2.7.2 Environnement expérimental

Le modèle de machine considéré dans nos expériences est dérivé d'une microarchitecture VLIW 32 bits similaire à [28]. Une instruction longue peut contenir jusqu'à quatre instructions indépendantes. La microarchitecture est dotée d'un pipeline à 6 étages : chargement des instructions (IF), décodage des instructions (ID), accès aux registres (RR), 2 étages d'exécution (EX1, EX2) et, la mise à jour des registres (WB). Il existe trois niveaux de *bypass* : EX1-EX2, EX2-EX1, EX2-RR. Le fichier de registres contient 64 registres généraux 32 bits.

2.7.3 Applications représentatives

Les applications de la suite Powerstone [80] ont été sélectionnées pour représenter la charge de travail des systèmes embarqués. Un descriptif de ces applications est présenté dans le tableau 2.2. Chaque application a été compilée avec le niveau d'optimisations *O*3. Ces dernières ont été exécutées entièrement et pourvues de différents jeux de données.

2.7.4 Modèle analytique de consommation

Le modèle architectural proposé en section 2.6 a été conçu pour limiter la consommation de certaines portions du processeur ; c-à-d. les unités du calcul entier et les fichiers de registres. L'ajout de circuits de désactivation (*clock-gating*) a notamment permis d'inhiber le signal d'horloge aux verrous d'entrée (*latches*) et aux portions hautes des ALUs ; c-à-d. une réduction de la consommation dynamique. L'implantation du mécanisme *drowsy* [32] au sein du fichier de registres a permis de réduire à la fois la consommation dynamique et statique de cette structure. À noter que pour simplifier le modèle, nous supposons ici uniquement deux modes de fonctionnement : c-à-d. 16 et 32 bits.

2.7.4.1 Consommation des unités entières

On peut approximer la consommation des unités entières en fonction de la fréquence à laquelle elles fonctionnent dans un mode donné (16 ou 32 bits) tout en y intégrant la consommation due aux mauvaises spéculations. Nous supposons que la consommation est linéaire par rapport au nombre de bits traités [12]. La formule 2.1 modélise la consommation des unités entières : $E_{ALU}^{16/32}$.

$$E_{ALU}^{16/32} = E_{ALU}^{32} * (\underbrace{(1-r)}_{32-bit} + \underbrace{\frac{r}{2}}_{16-bit} + \underbrace{m * r * \alpha}_{recovery}) \tag{2.1}$$

où : E_{ALU}^{32} correspond à la consommation d'une ALU classique en mode 32 bits. Le paramètre r correspond à la fraction de régions tronquées découvertes par notre analyse. m correspond au taux de mauvaises prédictions. α est une constante reflétant le coût supplémentaire dû au recouvrement (e.g. coût du redimensionnement du chemin de données). Dans cette évaluation, nous considérons $\alpha = 1$; le coût en énergie résultant d'une mauvaise prédiction équivaut ainsi à rejouer l'instruction dans le mode 32 bits.

2.7.4.2 Consommation du fichier de registres

Le mode *drowsy* [32] a été principalement introduit afin de réduire la consommation statique d'une unité mémoire. Ce mode permet également de limiter la consommation dynamique car seules les cellules actives sont sollicitées lors d'un accès. À noter qu'une sollicitation des cellules dans un état de veille peut détruire leur contenu.

Consommation dynamique La consommation dynamique $E_{RF}^{16/32(dyn)}$ du fichier de registres peut être approximée par la formule 2.2 :

$$E_{RF}^{16/32(dyn)} = \underbrace{N_{rw(32)} * E_{access(32)}}_{32-bit\ cost} + \underbrace{N_{rw(16)} * E_{access(16)}}_{16-bit\ cost} \tag{2.2}$$

où : les paramètres $N_{rw(16)}$ et $N_{rw(32)}$ correspondent respectivement aux nombres d'accès 16 et 32 bits en lecture/écriture au fichier de registres. $E_{access(16)}$ et $E_{access(32)}$ représentent le coût énergétique d'un accès 16 et 32 bits en lecture/écriture au fichier de registres. À noter que les accès en lecture et écriture sont confondus ; le modèle suppose en moyenne 95% d'accès en lecture et 60% d'accès en écriture [91].

Technological parameter	**Value**
Technology process	$0.18\mu m$
Frequency	1 GHz
# of read/write ports	8/4
$E_{access(16)}$ (16-bit byte-slice)	0.45 nJ
$E_{access(32)}$ (32-bit byte-slice)	0.80 nJ
$E_{access(monolithic)}$ (32-bit monolithic RF)	0.72 nJ
P_{leak}^{drowsy} (drowsy leakage power/cell)	2.34 pW
P_{leak}^{active} (normal leakage power/cell)	9.47 pW

TAB. 2.3 – Paramètres considérés pour l'évaluation.

Consommation statique La consommation statique $E_{RF}^{16/32(stat)}$ du fichier de registres peut être approximée par la formule 2.3 :

$$E_{RF}^{16/32(stat)} = \frac{1}{f} * (\underbrace{N_{cycle}^{drowsy} * N_{cell}^{drowsy} * P_{leak}^{drowsy}}_{drowsy\ mode} + \underbrace{N_{cycle}^{active} * N_{cell}^{active} * P_{leak}^{active}}_{active\ mode}) \quad (2.3)$$

où : N_{cycle}^{drowsy} correspond au nombre de cycles pendant lesquels la moitié (16 bits) du fichier de registres est en mode sommeil. N_{cycle}^{active} correspond inversement au nombre de cycles pendant lesquels le fichier de registres (32 bits) est en mode actif. N_{cell}^{drowsy} correspond au nombre de cellules mémoire dans le mode drowsy. N_{cell}^{active} correspond au nombre de cellules mémoire actives. P_{leak}^{drowsy} correspond à la consommation de puissance en mode drowsy pour une cellule. P_{leak}^{active} correspond à la consommation de puissance en mode actif pour une cellule. f est la fréquence à laquelle la structure est cadencée.

2.7.5 Résultats expérimentaux

Cette section présente les résultats expérimentaux relatifs au schéma proposé. Plusieurs points sont ainsi discutés : (1) la réduction en énergie dynamique et statique du chemin de données, (2) la dégradation des performances résultant de l'usage du mécanisme de recouvrement et, (3) l'augmentation de la taille du code due aux optimisations logicielles.

2.7.5.1 Réduction de la consommation dynamique

Le fait de redimensionner le fichier de registres, les unités fonctionnelles et les verrous aux besoins du calcul implique une réduction de la consommation dynamique de ces structures. Étant donné qu'il est difficile d'obtenir certains paramètres technologiques, les résultats empiriques ont été transposés sur une architecture analogue [89] ; ceci afin

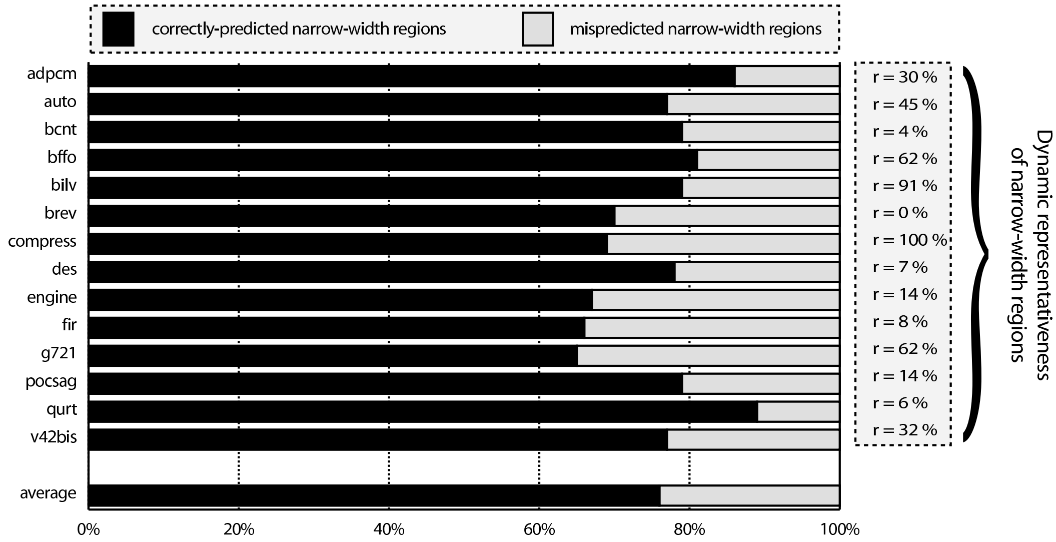


FIG. 2.12 – Précision des prédictions pour τ fixé à 60%.

de mieux mettre en avant les gains en énergie de notre modèle. Dans [89], les auteurs considèrent que les verrous participent à 36%, le fichier de registres à 28% et les ALUs à 27% de la consommation dynamique.

Fichier de registres en tranches L'équation 2.2 a été évaluée pour pouvoir quantifier la consommation du fichier de registres en tranches $E_{RF}^{16/32(dyn)}$. Les nombres d'accès au fichier de registres en mode 16 bits $N_{rw(16)}$ et en mode 32 bits $N_{rw(32)}$ ont été estimés grâce à notre outil de simulation cycle par cycle. Les énergies par accès en mode 16 bits $E_{access(16)}$ et en mode 32 bits $E_{access(32)}$ ont été approximées grâce à une version modifiée de *Cacti* 3 [43]. Le tableau 2.3 expose la consommation par accès et les divers paramètres technologiques fournis à *Cacti*. Nous pouvons ainsi remarquer que la consommation d'une cellule mémoire implémentant le mode *drowsy* consomme une quantité d'énergie légèrement supérieure par rapport à une cellule mémoire classique.

Unités fonctionnelles et verrous L'équation 2.1 a été évaluée afin de quantifier la consommation des unités arithmétiques et logiques $E_{ALU}^{16/32}$. Les paramètres r et m ont été mesurés pour différentes valeurs de τ fixées à 80% et 60%. Les résultats sont exposés en figures 2.13 et 2.12. En moyenne, nous pouvons observer un taux de mauvaises prédictions m de l'ordre de 6% pour $\tau = 80\%$ et de l'ordre de 23% pour $\tau = 60\%$.

Résultats La figure 2.14 expose la réduction en énergie de notre modèle par rapport à la microarchitecture décrite dans [89]. Une valeur de τ de 80% a été considérée. Les gains en énergie par composant sont illustrés dans la partie haute de la figure 2.14. Les gains en énergie par rapport au chemin de données sont affichés dans la partie basse

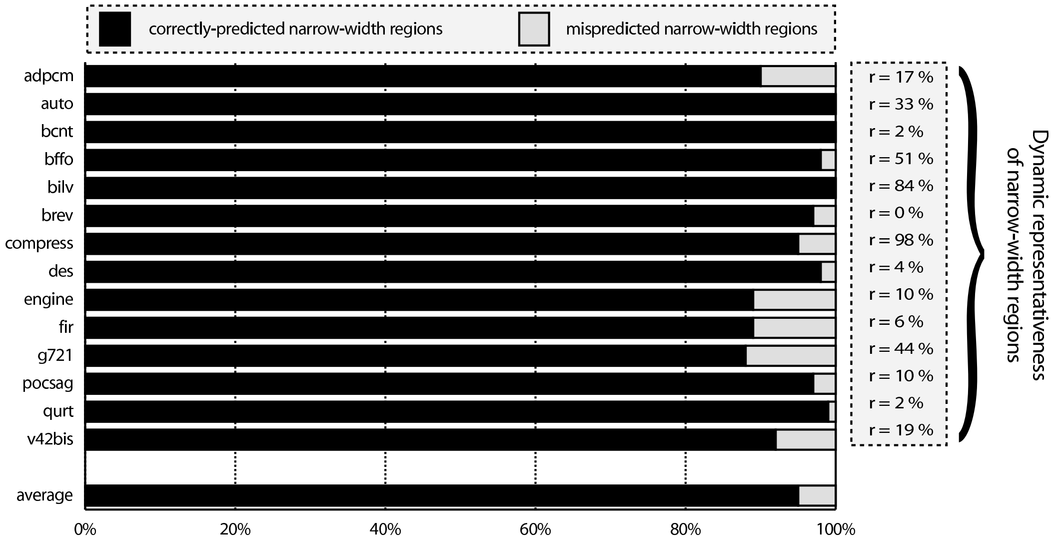


FIG. 2.13 – Précision des prédictions pour τ fixé à 80%.

de cette figure. En moyenne, le modèle proposé permet de réduire l'activité du chemin de données de 17%. Sur les applications *bilv* ou *compress*, notre approche permet une réduction considérable de l'énergie du chemin de données ; c-à-d. de l'ordre de 40 et 70%. En revanche, sur les applications *bcnt* ou *brev*, le modèle proposée n'apporte pas de gains substantiels. Ceci s'explique notamment par la faible proportion en régions tronquées r découvertes statiquement : 2% et 0% pour $\tau = 80\%$ (cf. figure 2.13). Il est possible d'obtenir de meilleurs résultats en faisant varier τ plus finement sans impacter sur les performances. Une méthode envisageable pour attribuer une telle valeur à τ est de considérer la métrique énergie-délai [33]. Cette méthode sera évaluée lors de prochains travaux.

2.7.5.2 Réduction de la consommation statique

Le fichier de registres en tranches (cf. section 2.6.2) présente une consommation statique réduite par rapport à une structure conventionnelle. Ceci est notamment dû à la mise en mode *drowsy* [32] des tranches de poids fort lors de l'exécution d'une région tronquée. L'équation 2.3 permet de quantifier la réduction d'énergie en régime statique de notre modèle. Les paramètres P_{leak}^{drowsy} et P_{leak}^{active} ont été estimés à partir de l'outil *Hotleakage* [90]. Le tableau 2.3 affiche la valeur de ces paramètres. La figure 2.15 résume le gain en consommation statique du fichier de registres en tranches. Nous pouvons ainsi observer un gain moyen en consommation statique de l'ordre de 20% pour les applications considérées.

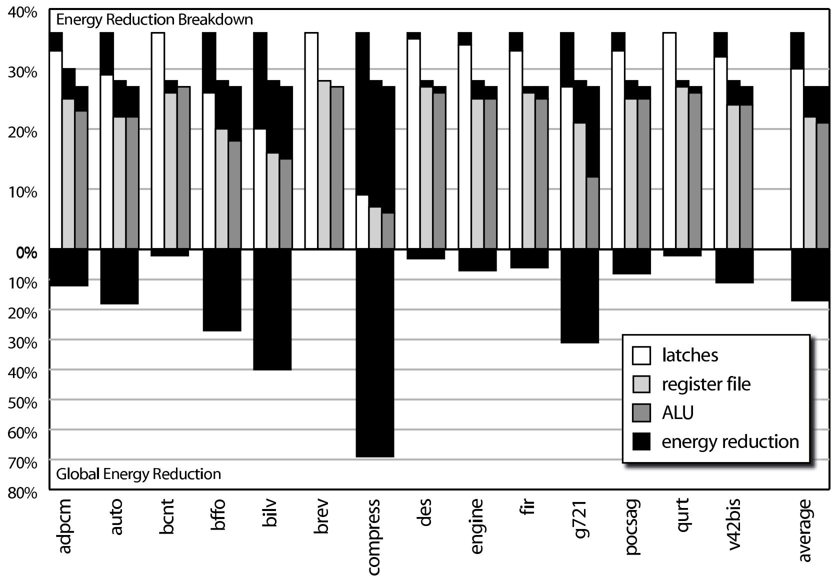


FIG. 2.14 – Réduction d'énergie par composant et réduction d'énergie totale pour τ fixé à 80%.

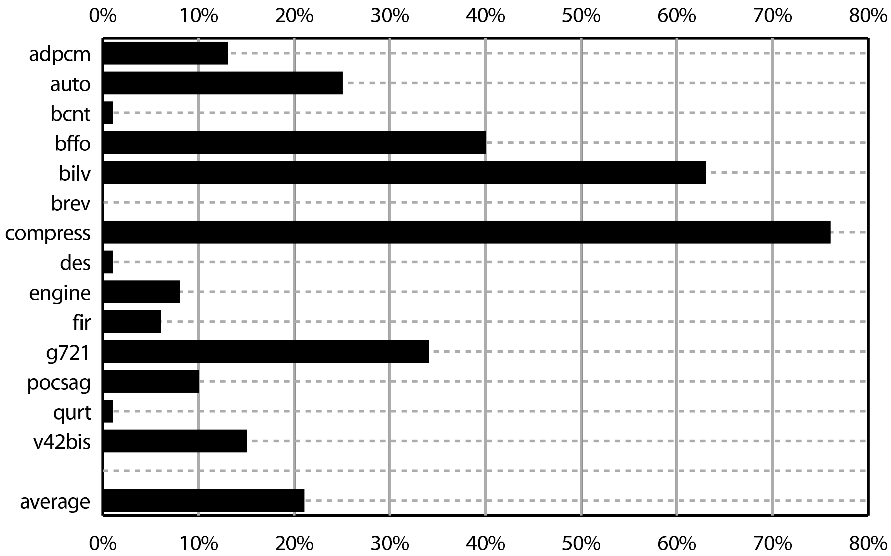


FIG. 2.15 – Réduction de consommation en régime statique du fichier de registres.

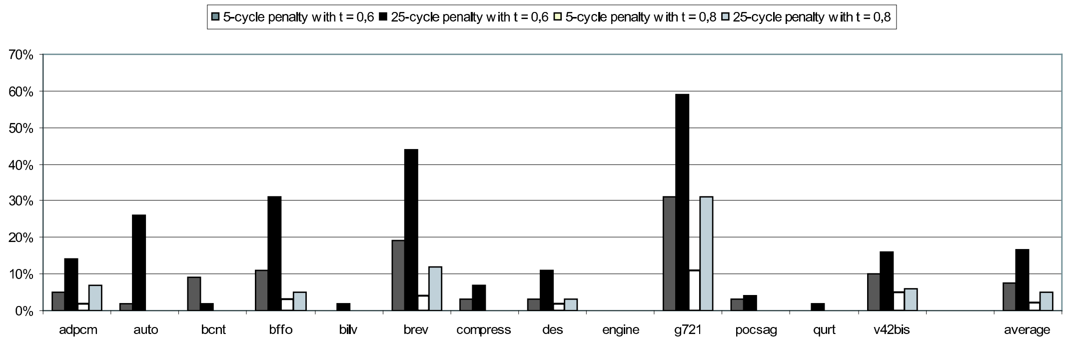


FIG. 2.16 – Dégradation des performances pour différentes valeurs de τ et p.

2.7.5.3 Impact sur les performances

La nature spéculative de notre schéma impose le recours à un mécanisme de recouvrement en cas de mauvaises prédictions. Dans un tel scénario, le pipeline doit être vidé et le chemin de données réadapté à une largeur correcte. Le surcoût en cycles résultant du recours au mécanisme de recouvrement est approximé par la formule 2.4.

$$Cost = n_{bb} * r * m * p \tag{2.4}$$

où : n_{bb} est le nombre de blocs de base exécutés. r est la proportion en régions tronquées exécutées. m est le taux de mauvaises prédictions. p est le coût, exprimé en nombre de cycles, du mécanisme de recouvrement.

Le paramètres m et r ont été estimés pour différentes valeurs de τ : 60 et 80% ; les résultats sont présentés sur les figures 2.12 et 2.13. Le paramètre p peut être vu comme : (1) *le nombre de cycles nécessaires pour vider le pipeline* et, (2) *le nombre de cycles nécessaires pour réadapter le chemin de données à une largeur correcte.* Étant donné qu'une mauvaise prédiction est détectée à l'étage d'exécution, nous supposons un coût de 3 cycles pour (1). Le coût de (2) peut toutefois varier selon la complexité du processeur. Nous supposons alors un coût de recouvrement de 5 cycles pour le meilleur scénario et de 25 cycles au pire des cas. La figure 2.16 présente la dégradation des performances pour différentes valeurs de p et de τ. En moyenne, pour une valeur de p fixée à 5 cycles, le recouvrement a un impact négligeable sur les performances. En revanche, pour une valeur de p fixée à 25 cycles, les performances peuvent être dégradées de manière substantielle ; jusqu'a 30% pour τ = 80% et de 60% pour τ = 60%. Ces observations dénotent l'importance de la précision des prédictions dans notre modèle.

2.7.5.4 Augmentation de la taille du code

Les transformations logicielles (cf. section 2.5) proposées pour réduire la fréquence d'appel au mécanisme de recouvrement peuvent avoir un impact négatif sur la taille du code. La séparation du calcul d'adresses de l'accès mémoire peut augmenter la taille du code si un nombre important de micro-opérations est atteint. Ce nombre dépend essentiellement du mode d'adressage. Le réordonnancement du code peut également augmenter la taille du code si il est nécessaire de dupliquer l'instruction à déplacer. La duplication a lieu lorsqu'une instruction est déplacée vers un bloc de base externe et que plusieurs chemins de contrôle existent. Des heuristiques simples qui privilégient le compromis performance/taille de code ont été considérées pour restructurer le code. Pour une valeur de τ fixée à 80%, nous avons mesuré une augmentation moyenne de 3% pour l'ensemble des applications concernées. Il est possible d'améliorer ce facteur à travers des heuristiques plus élaborées. Ceci sera évalué lors de futurs travaux.

2.8 Conclusions et directions futures

Ce chapitre a décrit la proposition logicielle/matérielle émise dans [71] pour redimensionner le chemin de données aux besoins du calcul. La majorité des travaux relatifs à notre modèle ont considéré une approche logicielle ou matérielle pour détecter la dimension des données. Ces approches ne sont toutefois pas suffisamment efficaces ou adéquates pour adresser la réduction d'énergie dans les systèmes embarqués. Les approches logicielles sont relativement limitées de part leur aspect conservatif. Les approches matérielles requièrent une complexité matérielle difficilement envisageable dans le contexte embarqué.

Nous avons proposé une solution intermédiaire présentant un compromis acceptable en termes de consommation/complexité/performance. Le principe de notre approche est d'inférer la dimension optimale des données d'un programme grâce à l'analyse de ses exécutions antérieures. Nous avons montré que les données tronquées sont fortement concentrées au sein des blocs de base du programme. Le découpage statique du programme en régions, exposant la même largeur d'exécution, a été ainsi proposé. Nous avons décrit un support spécifique pour exécuter ces régions tronquées. Nous avons proposé un nouveau concept pour fichiers de registres adapté à notre modèle. La dimension de ce dernier peut être accommodé aux fluctuations du calcul tout en préservant l'état des données. En couplant cette structure à un chemin de données redimensionné à la largeur des régions, nous avons obtenu une réduction significative de la consommation en régime dynamique et statique. L'évaluation du modèle proposé révèle, en effet, une réduction de 17% de la consommation dynamique du chemin de données et une réduction de 22% de la consommation statique du fichier de registres.

Directions futures

Plusieurs autres voies d'exploration sont envisageables à notre étude :

- **Adopter un schéma de partitionnement du cache de données similaire à celui du fichier de registres**. Les caches de données dissipent une fraction dominante des puissances dynamique et statique de la puce (e.g. 40% pour le processeur alpha 21264 [44]). Il est possible d'optimiser ces puissances en accommodant la largeur du cache à celle des régions du programme. Les modifications matérielles nécessaires à cette approche sont similaires à celles employées pour le fichier de registres en tranches. Les politiques de redimensionnement peuvent toutefois différer selon les performances recherchées.

- **Optimiser le seuil qualifiant une région tronquée pour chaque application**. Par convention, nous avons considéré deux seuils arbitraires (60% et 80%) pour qualifier une région tronquée. Une approche plus optimale serait de déterminer ce seuil à travers un profil des applications mettant en jeu le compromis performance/consommation. La métrique *énergie/délai* peut être utilisée à cet effet [33]. Il serait alors possible d'obtenir un seuil optimal en fonction du but premier, soit augmenter les performances ou réduire la consommation.

- **Appliquer la formation des régions et les optimisations logicielles au niveau binaire**. Cette étude a considéré le niveau assembleur pour appliquer La formation des régions et les optimisations logicielles. Une approche plus efficace serait de considérer le niveau binaire. De cette manière, les librairies associées au programme pourraient aussi être optimisées. Une analyse préliminaire basée sur un outil similaire à [63, 5] montre que ce schéma augmenterait le nombre de régions de 15% en moyenne. Ce qui prouve l'intérêt de cette étude. Un autre argument en faveur de cette méthode est de rendre indépendant le processus de formation des régions du compilateur.

- **Utiliser la compilation dynamique**. Cette étude s'appuie sur des informations obtenues par *profilage* pour inférer la largeur d'exécution des blocs de base du programme. Cependant, ces informations peuvent différer selon le jeu de données utilisé en entrée. Dans notre contexte, chaque application a été exécutée avec différents jeux de données et seules les informations pertinentes ont été reportées au sein du profil final. Une approche plus efficace serait toutefois de recourir à la compilation dynamique. La compilation dynamique [6] est une technique qui permet d'appliquer des optimisations logicielles à l'exécution. Il serait donc possible de l'utiliser pour construire et optimiser les régions tronquées à la volée. L'intérêt de cette méthode est de pouvoir adapter les régions à l'exécution, ce qui pourrait réduire la fréquence d'appel au mécanisme de recouvrement, tout en profitant des avantages de la méthode binaire, c-à-d. optimiser les librairies et s'abstraire du code source.

- **Adapter l'approche pour un modèle à exécution dans le désordre**. Dans notre modèle, le redimensionnement du chemin de données se fait grâce à une instruction insérée par l'utilisateur. Cette approche n'est pas directement applicable à une architecture superscalaire à exécution dans le désordre car le traitement des instructions de reconfiguration doit se faire dans l'ordre. Une approche naïve pour adapter notre modèle est de considérer deux modes d'exécution commandés par l'instruction de reconfiguration : un mode dans le désordre et un mode dans l'ordre. Cette architecture à bascule [4] pourrait cependant impacter de manière significative sur les performances. Une approche plus efficace consisterait à considérer deux chemins de données taillés aux dimensions des opérations. Cette approche est examinée dans le chapitre suivant.

Chapitre 3

Traitement découplé des opérations tronquées sur des opérateurs dédiés *

3.1 Introduction

Les processeurs superscalaires modernes implémentent des structures de plus en plus larges pour maximiser le parallélisme d'instructions tout en supportant des fréquences d'horloge élevées. Bien que cette stratégie ait suscité la conception de processeurs plus performants, il devient toutefois plus difficile de l'appliquer sans surmonter plusieurs nouvelles difficultés. Parmi ces difficultés, le délai de propagation des signaux dans les interconnexions tend de plus en plus à limiter la réduction du temps de cycle avec le passage aux géométries très fines [39]. Les autres difficultés, liées au développement des processeurs superscalaires, concernent le fichier de registres, le réseau de *bypass* ainsi que les logiques associées à la fenêtre d'instructions [65]. De plus, comme la complexité de ces structures augmente significativement avec la largeur d'exécution, la consommation et la surface de silicium apparaissent également comme des problèmes majeurs.

Plusieurs solutions ont été avancées dans la littérature pour maîtriser ces difficultés de conception. Certaines études ont suggéré le recours au *partitionnement des ressources de calcul* en composants simplifiés, appelés *clusters*, pour ce faire [65, 30, 8]. Dans ces études, le partitionnement des ressources est employé pour minimiser la complexité des structures appartenant au chemin critique du processeur. Les clusters résultants implémentent des structures de taille réduite leur permettant d'être cadencés à une fréquence élevée. Ce gain se fait toutefois en contrepartie d'une dégradation du nombre d'instructions exécutées par cycle (IPC). Cette dégradation de l'IPC dépend essentiellement de la stratégie adoptée pour assigner les instructions aux différents clusters. La

*Disponible en rapport de recherche INRIA [76] et en cours de parution au journal **ACM TACO** : « *A Case for a Complexity-Effective, Width-Partitioned Microarchitecture* »[77].

logique réalisant cette tâche est à la fois responsable de la répartition des charges et du nombre de communications entre clusters. De plus, les interconnexions utilisées pour communiquer les données dissipent une fraction importante de la puissance totale [56]. Pour la performance et la consommation de ces microarchitectures, il est donc fondamental de minimiser le nombre de communications inter-cluster tout en maintenant une répartition des charges balancée.

D'autres études ont proposé d'optimiser la mise en œuvre matérielle des structures critiques pour maîtriser cette complexité. Les optimisations proposées sont généralement fondées sur des observations empiriques concernant le comportement dynamique des programmes. Brooks et al. [12] ont récemment montré la propriété des programmes à s'exécuter avec une forte proportion d'opérations tronquées. De nombreuses propositions ont exploité cette propriété pour réduire la consommation d'énergie [12, 71, 17, 48] ou améliorer les performances [27, 54, 78, 55, 64]. Bien que les propositions émises permettent de réduire la consommation de certaines structures (e.g. le fichier de registres), il est difficile de mesurer leurs impacts sur la performance et la consommation totale du processeur. Ces propositions architecturent en effet des mécanismes complexes qui peuvent dissiper une puissance non négligeable et être placés sur le chemin critique du processeur.

3.1.1 Proposition et contributions

Dans ce chapitre, nous examinons un nouveau schéma de partitionnement basé sur la propriété des programmes observée par Brooks. Le fondement de notre approche repose sur l'analyse de la distribution dynamique des opérations s'exécutant exclusivement avec des opérandes tronqués. Nous avons remarqué que ces opérations tronquées et le reste des instructions du programme sont fortement entrelacées au cours d'une exécution. Cette propriété a été observée pour les applications MediaBench [52] et SPEC2000 [38]. Il a été également mis en avant qu'une large fraction de ces opérations tronquées, en moyenne 40%, participent à l'exécution des applications considérées. Nous suggérons d'user de ces propriétés pour partitionner les ressources de calcul du processeur en clusters de largeur distincte. Le nouveau schéma de partitionnement proposé permet de réduire considérablement la complexité des processeurs superscalaires. Ce schéma est désigné WPM[1].

Dans un WPM, le traitement des opérations tronquées est découplé de celui du reste des instructions du programme. Faire ainsi résulte en une simplification considérable de la structure matérielle des composants critiques présents dans chaque cluster : fichier de registres, réseau de *bypass*. De plus, la propriété relative à l'entrelacement dynamique des opérations de dimensions distinctes permet de répartir correctement les charges de travail sans ajout d'artefacts complexes. Cette étude montre également la capacité du modèle WPM à réduire efficacement la complexité du réseau d'interconnexion. Dans un WPM, les communications inter-cluster concernent uniquement les échanges de données

[1]WPM correspond à *Width-Partitioned Microarchitecture*

tronquées. Par conséquent, un dimensionnement respectif du réseau d'interconnexion permet de réduire substantiellement la consommation, le temps d'accès et la surface de ce composant.

Ce chapitre détaille une implémentation simple du modèle WPM pour un processeur superscalaire de degré 4. L'implémentation considérée partitionne le cœur d'exécution entier en deux clusters distincts : (1) un cluster capable d'exécuter les opérations 64-bit, les opérations load/store et les opérations complexes du programme et, (2) un cluster dimensionné au traitement des opérations 16-bit. Le mécanisme d'allocation des instructions aux clusters est basé sur un prédicteur matériel de largeur. Différents prédicteurs de largeur sont discutés et évalués. Une analyse approfondie du modèle WPM révèle un gain substantiel de la consommation, de la surface de silicium et des performances voisines d'un modèle clusterisé classique.

3.1.2 Organisation du chapitre

Ce chapitre est structuré de la manière suivante. La section 3.2 motive notre approche en présentant une taxinomie des opérations exécutées par un programme en fonction de la dimension des opérandes manipulés. La section 3.3 décrit de manière détaillée l'implémentation du modèle WPM proposé. La section 3.4 présente le mécanisme d'allocation des instructions ainsi que différentes implémentations du prédicteur de largeur. La section 3.5 propose une analyse approfondie de la complexité du WPM par rapport à une microarchitecture clusterisée conventionnelle. La section 3.6 propose l'évaluation du WPM en termes de performance et consommation. La section 3.7 conclut notre étude en exposant les applications futures du modèle WPM.

3.2 Motivations

Différentes études [12, 55, 71, 27] ont mis en avant la propriété des programmes entiers et multimédia à procéder une fraction considérable de données tronquées à l'exécution. Diverses granularités et diverses définitions d'une opération s'exécutant avec des données tronquées ont été considérées par ces études. Dans les travaux de Brooks et al. [12], les auteurs qualifient comme une opération tronquée, toute instance dont tous *les opérandes sources* ne comportent au plus 16 bits significatifs. Dans le chapitre précédent, nous considérons une granularité plus importante pour définir la notion de *région tronquée*. Dans cette étude, une opération est considérée comme tronquée lorsque *tous ses opérandes comportent un nombre de bits significatifs inférieur à 16 bits ; ceci incluant l'opérande destination.*

3.2.1 Classification des opérations

Nous avons analysé la distribution des opérations tronquées dans diverses applications composant les suites MediaBench [52] et SPEC2000 [38]. Cette analyse concerne

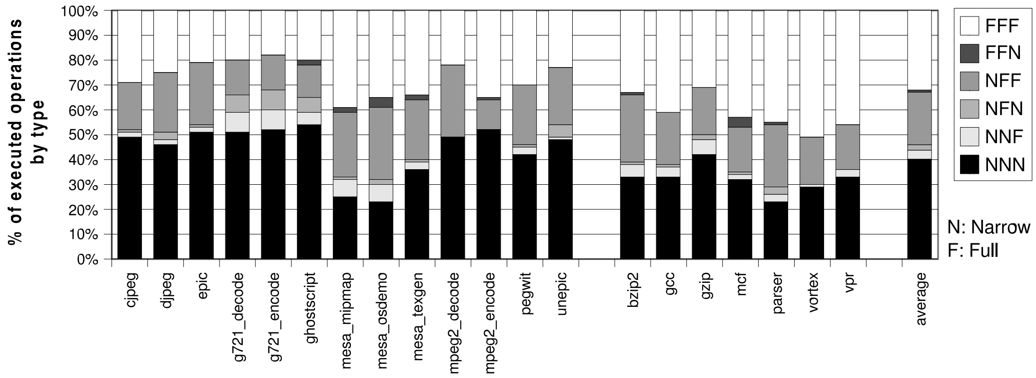


Fig. 3.1 – Classification et distribution des opérations tronquées à l'exécution.

exclusivement les opérations procédées par les unités entières d'exécution ; ceci incluant les calculs d'adresses mémoire. Les résultats présentés tiennent également compte des opérations manipulant des données signées dont la valeur absolue est tronquée. La figure 3.1 expose la classification et la distribution des opérations tronquées à l'exécution. Par convention, nous noterons N un opérande tronqué et F un opérande non tronqué. Nous utiliserons un notation à 3 lettres pour catégoriser une opération : les deux premières lettres désignent la taille des opérandes sources et la dernière celle de l'opérande destination. Par exemple, NFN désigne une opération qui procède en source un opérande tronqué ainsi qu'un opérande non tronqué et produit un opérande tronqué. Pour les opérations monadiques, nous considérons que tous les opérandes sources sont de même dimension.

La figure 3.1 révèle ainsi qu'une fraction importante de l'exécution entière est dédiée au traitement des opérations tronquées (NNN) ; soit 40% en moyenne. Ces résultats corroborent aux observations de Brooks relatives à l'usage dynamique intensif des données tronquées dans le calcul entier. Ceci peut suggérer un schéma dans lequel le traitement des opérations tronquées est découplé sur un chemin de données dimensionné en conséquence. La complexité des certains composants appartenant au chemin critique du processeur pourrait ainsi être significativement réduite. Dans cette étude, nous recommandons un modèle à clusters pour découpler l'exécution des opérations tronquées. Chaque cluster implémente un chemin de données de largeur distincte. Nous désignons ce nouveau modèle de partitionnement par WPM : *width-partitioned microarchitecture*. Comme pour une microarchitecture clusterisée conventionnelle, le modèle WPM requiert la mise en œuvre d'un mécanisme pour distribuer les instructions vers les clusters. Il est primordial que les heuristiques, considérées pour ce faire, puissent équilibrer les charges de travail correctement tout en minimisant le nombre de communications entre les clusters.

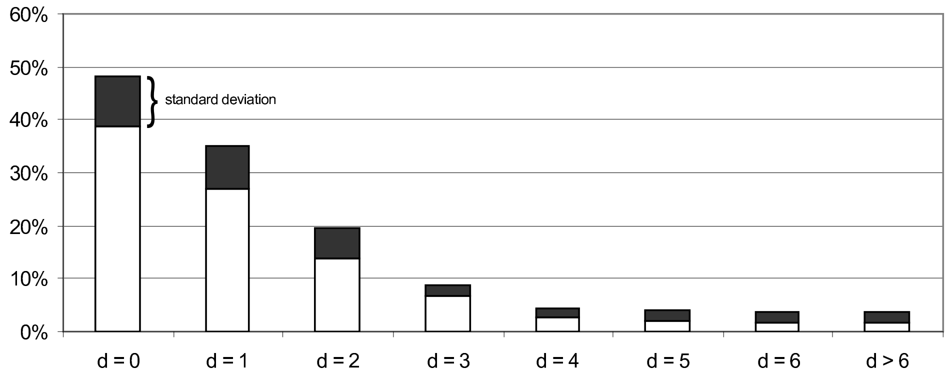


FIG. 3.2 – Moyenne des distances séparant les opérations tronquées des autres instructions du programme à l'exécution.

3.2.2 Communication inter-cluster

La figure 3.1 affiche également une estimation du nombre de communications susceptibles de se produire dans un WPM. Une communication inter-cluster peut uniquement avoir lieu lorsqu'une opération nécessite un opérande produit par un cluster distant (**NNF, NFN, NFF**) ou qu'une opération produit un résultat qui doit être propagé au cluster distant (**NFN, FFN**). La figure 3.1 montre que ces divers cas s'appliquent à 20% des opérations entières. Ceci pourrait alors se traduire en un scénario où 1 opération procédée sur 5 déclenche une communication inter-cluster. Néanmoins, ceci est une borne maximale car le nombre de communications est fortement lié à la distribution des opérations et aux dépendances de données entre ces opérations. Les opérations tronquées ne produisent pas systématiquement un résultat utilisé par une opération non tronquée. Les résultats de notre évaluation montrent en effet que le nombre de communications inter-cluster est largement inférieur à cette borne maximale.

3.2.3 Équilibrage des charges

Une caractéristique fondamentale du mécanisme d'allocation des instructions est de garantir l'équilibrage des charges entre les clusters. Nous avons approximé cette caractéristique pour le modèle WPM. Ceci a été fait de la manière suivante. Pour chaque opération, nous avons collecté la distance séparant une opération tronquée de la prochaine opération s'exécutant avec des opérandes plus larges (64-bit) et celle séparant une opération 64-bit d'une opération tronquée. La figure 3.2 affiche la moyenne des distances les plus fréquemment observées à l'exécution des applications MediaBench et SPEC2000. L'écart-type de la moyenne est également reporté et met en évidence un comportement dynamique similaire entre les applications. La figure 3.2 montre ainsi la prédominance des distances courtes à l'exécution. Ceci peut être dû au fait que nous avons aussi considéré les calculs d'adresses mémoire dans nos expériences. Les calculs d'adresses sollicitent généralement toute la largeur d'exécution. Cette prédominance des distances courtes à l'exécution pourrait ainsi signifier que les occurrences des opérations tronquées sont fortement entrelacées avec celles des autres instructions du programme. Ceci peut donc indiquer que le mécanisme d'allocation des instructions du modèle WPM peut s'appuyer sur des heuristiques très simples pour réaliser un équilibrage des charges satisfaisant.

3.3 Implémentation du modèle WPM

La section 3.2 a montré l'usage fréquent des opérations tronquées à l'exécution des programmes entiers et multimédia. Il a été aussi observé que les opérations tronquées sont fortement entrelacées avec les autres instructions du programme durant l'exécution. Cette section examine un nouveau schéma de partitionnement, le modèle WPM, pour exploiter ces propriétés dynamiques. Notre implémentation du WPM est dérivée d'une machine superscalaire de degré 4. Cette implémentation considère deux clusters aux largeurs distinctes capables d'exécuter jusqu'à deux instructions par cycle. Il est possible de considérer une implémentation du WPM mettant en œuvre un nombre de clusters et/ou une largeur d'exécution supérieurs. Le modèle d'inter-communication doit cependant être modifié de manière appropriée [66] pour préserver la complexité de notre approche. L'étude du passage à l'échelle du WPM sera examinée ultérieurement.

3.3.1 Modèle de référence

Le modèle pris en référence pour comparaison est dérivé du processeur Alpha 21264 [44]. Ce processeur intègre deux clusters de calcul 64-bit et peut procéder les instructions dans le désordre. Un tampon de réordonnancement (ROB) global permet de préserver l'ordre séquentiel du programme. Nous supposons que les opérations flottantes sont traitées séparément dans un cluster dédié dont la description sera omise. La figure 3.3 présente le synoptique de cette microarchitecture de référence. La partie frontale du pipeline d'exécution (chargement, décodage, renommage) et le cache de données sont partagés par tous les clusters. De manière similaire au processeur Alpha 21264 [44], nous supposons que chaque cluster possède une queue d'instructions découplée du tampon de réordonnancement. Les autres structures matérielles distribuées au sein des clusters regroupent les unités fonctionnelles et le fichier de registres. Chaque cluster implémente une copie intégrale du fichier de registres.

Au niveau de l'exécution, chaque cluster peut produire jusqu'à 2 résultats par cycle. Toutes les unités de calcul 64-bit peuvent traiter des opérations complexes telles que les multiplications ou les opérations de décalage. Le séquencement des opérations mémoires (*load/store*) est contraint à un seul cluster. Nous supposons aussi que les unités *load/store* sont capables de traiter les opérations arithmétiques et logiques. Autrement, nous considérons un réseau d'interconnexion complet entre les clusters. Chaque

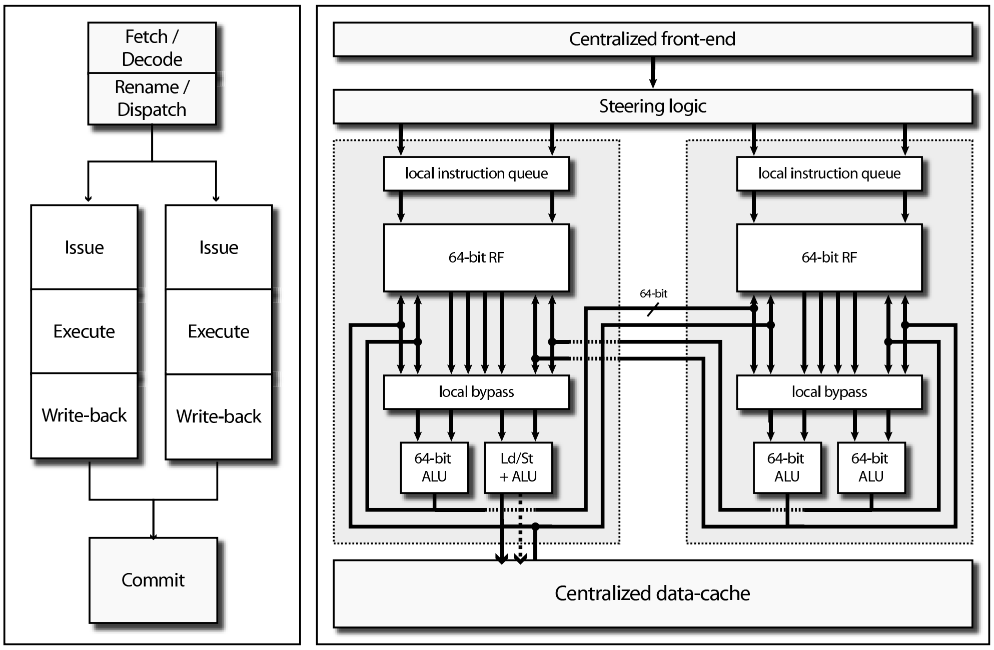


FIG. 3.3 – Modèle de référence.

unité fonctionnelle est ainsi connectée en écriture au fichier de registres distant. Il est possible de considérer cette topologie d'interconnexion car notre implémentation se compose d'un nombre restreint de clusters [66]. Ceci permet ainsi d'éliminer les contentions de ressources pour communiquer les données entre les clusters. Pour supporter cette topologie, chaque fichier de registres doit intégrer un nombre de ports en écriture équivalent au nombre d'unités fonctionnelles, soit 4 ports en écriture par cluster.

Les instructions chargées puis décodées sont dirigées vers l'étage de renommage. À cette étage, le mécanisme d'allocation doit déterminer le cluster sur lequel les instructions seront assignées. Le mécanisme d'allocation des instructions considéré est basée une heuristique similaire à [19]. Cette heuristique assigne les instructions au cluster produisant la majorité de ses opérandes. Ceci permet intuitivement de limiter le nombre de communications inter-cluster. L'heuristique considère aussi l'équilibrage des charges en assignant les instructions sans opérandes au cluster le moins chargé. La lecture des opérandes d'une instruction est limitée au fichier de registres local. Nous supposons que tous les résultats produits par un cluster sont systématiquement propagés au cluster distant. Le résultat est disponible sur le *bypass* local du cluster producteur dans le même cycle. Ceci permet le séquencement immédiat des instructions dépendantes appartenant au cluster producteur. La disponibilité du résultat sur le réseau de *bypass* du cluster distant nécessite quelques cycles supplémentaires.

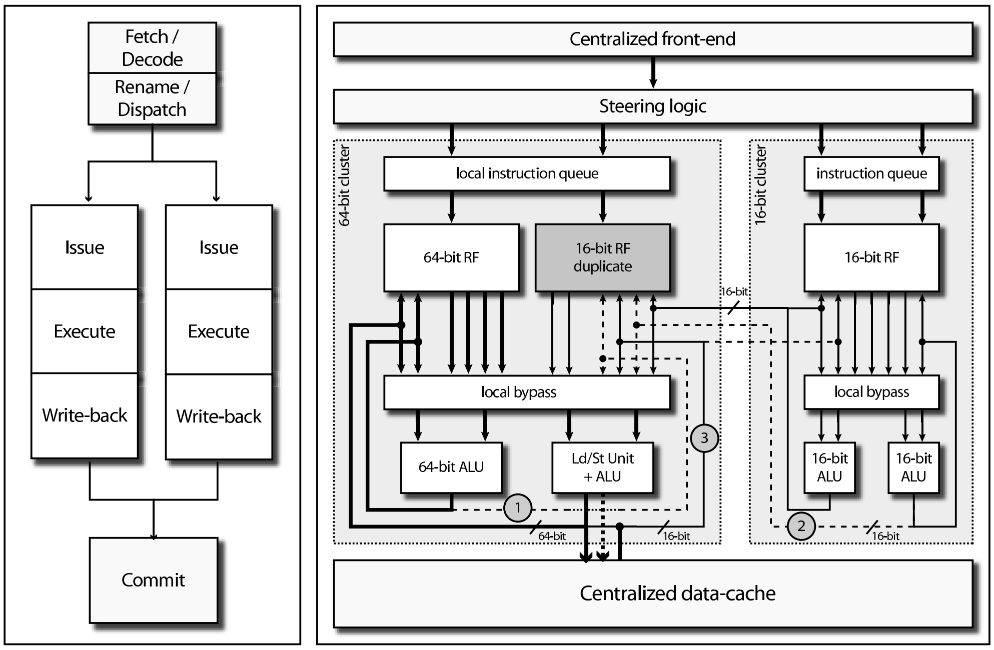


Fig. 3.4 – Modèle WPM.

3.3.2 Implémentation du modèle WPM

Nous proposons une implémentation basique du modèle WPM qui partitionne le cœur d'exécution entier en deux clusters distincts : (1) *un cluster 64-bit implémentant un chemin de données 64-bit* et (2) *un cluster tronqué implémentant un chemin de données 16-bit*. Chaque cluster est composé d'un ensemble d'unités fonctionnelles (FUs) et d'un fichier de registres local (RF). Le cluster tronqué intègre deux unités de calcul 16-bit et un fichier de registres 16-bit. Découpler le traitement des opérations tronquées sur ce cluster peut considérablement réduire la complexité du processeur. Aussi, aucun mécanisme spécifique ne doit être architecturé pour redimensionner le chemin de données, contrairement aux propositions antérieures [12, 71]. La section 3.5 discute des gains en complexité suscités par cette implémentation du WPM. Le fichier de registres 16-bit comporte 4 ports en lecture et 2 ports en écriture pour permettre l'exécution de deux opérations tronquées par cycle.

Le cluster 64-bit contient une unité de calcul 64-bit capable de traiter les opérations complexes et une unité *load/store* capable de traiter les opérations arithmétiques et logiques simples. Le fait d'intégrer l'unité *load/store* au cluster 64-bit est cohérent avec notre approche car les calculs d'adresses mémoire sollicitent souvent toute la largeur d'exécution. Le fichier de registres 64-bit implémente 4 ports en lecture et 2 ports en écriture pour supporter l'exécution de deux opérations 64-bit par cycle. La figure 3.4 détaille l'organisation de cette implémentation du WPM. Comme pour le modèle de référence, la partie frontale du pipeline d'exécution et le cache de données ne sont pas

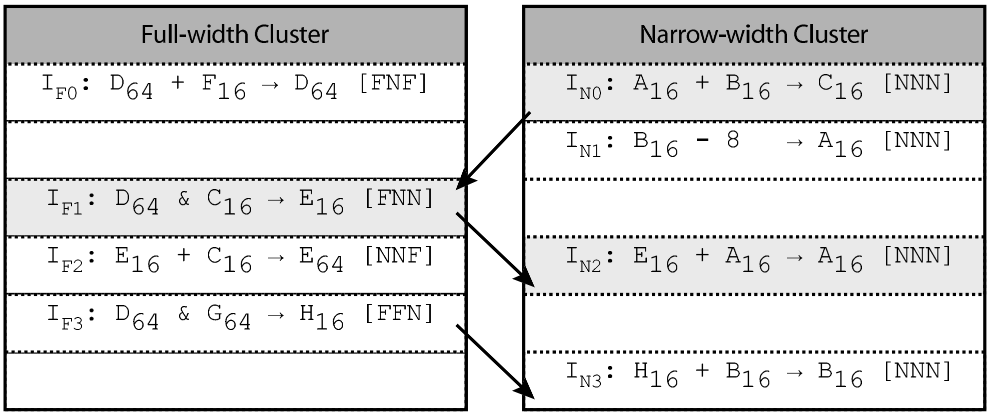


FIG. 3.5 – Scénario de communications inter-cluster.

partitionnés. Le mécanisme pour communiquer les données entre les clusters 16-bit et 64-bit est toutefois optimisé en fonction des propriétés dynamiques observées en section 3.2.

3.3.2.1 Communications inter-cluster

Le nombre de communications inter-cluster est déterminé par les dépendances de données existant entre les opérations tronquées et les autres instructions du programme. L'exemple proposé en figure 3.5 illustre un scénario basique de communications inter-cluster. Chaque opérande est préfixé par son nombre de bits significatifs. L'opération I_{N0} s'exécutant sur le cluster tronqué produit ainsi un résultat qui est consommé par l'opération I_{F1} séquencée sur le cluster 64-bit. Le résultat de cette dernière opération est ensuite consommé par l'opération I_{N2} s'exécutant sur le cluster tronqué. Sur cette figure, les flèches symbolisent les dépendances de données entre les instructions appartenant à des clusters distincts. Leur nombre correspond au cardinal de l'intersection entre l'ensemble regroupant les opérations tronquées et l'ensemble regroupant les autres instructions du programme. Ce cardinal représente la borne maximale du nombre de communications inter-cluster se produisant dans le WPM. L'exemple de la figure révèle ainsi trois communications de données.

Une approche naïve pour communiquer les données entre les clusters consiste à considérer un réseau d'interconnexion complet. Cette topologie requiert l'ajout de 4 ports en écriture supplémentaires sur les fichiers de registres : 2 ports pour connecter l'unité de calcul 64-bit et l'unité load/store et, 2 ports pour connecter les deux unités de calcul 16-bit. Cette approche peut toutefois avoir un impact négatif sur la consommation et le temps de cycle du processeur. La section 3.2 a montré que 20%, en moyenne, des opérations suscitent une communication inter-cluster. La duplication du fichier de registres 16-bit[2] a été envisagée pour tenir compte de cette propriété. Le contenu de

[2]le duplicata du fichier de registres 16-bit sera référé comme *duplicata 16-bit*

ce *duplicata 16-bit* est synchronisé avec celui du fichier de registres 16-bit implémenté par le cluster tronqué. Considérer ce *duplicata 16-bit* permet d'optimiser la surface, la consommation et le temps d'accès du processeur, comme démontré en section 3.5.

Implémentation détaillée du cluster tronqué Nous supposons que tout résultat produit par une unité de calcul 16-bit est systématiquement propagé au *duplicata 16-bit.* Nous considérons donc que toutes les unités 16-bit sont connectées en écriture au *duplicata 16-bit.* Le *duplicata 16-bit* doit disposer de deux ports en écriture supplémentaires pour ce faire. Il existe plusieurs raisons de considérer ce modèle d'inter-communication. Premièrement, la figure 3.5 affiche qu'une fraction importante des opérations séquencées sur le cluster 64-bit (**NFF**, **NNF**) consomment des données tronquées. Aussi, nous avons mesuré qu'une fraction importante des opérations *store*, s'exécutant sur le cluster 64-bit, opèrent avec des données tronqueés. Étant donnée cette utilisation fréquente des données tronquées par le cluster 64-bit, une approche de mise à jour des registres à la demande [66, 19] pourrait donc impacter négativement sur les performances. Le modèle d'interconnexion considéré privilégie les performances en contrepartie d'une consommation supérieure. La section 3.3.2.2 expose une nouvelle optimisation maximisant le compromis performance/consommation.

Implémentation détaillée du cluster 64-bit Le *duplicata 16-bit* comporte deux ports en lecture et deux ports en écriture supplémentaires pour permettre aux unités de calcul 64-bit et load/store de procéder des opérandes tronqués. Nous considérons toutefois l'ajout d'un seul port en écriture sur le fichier de registres 16-bit pour propager les résultats tronqués au cluster distant. Ceci se justifie par le fait que peu d'opérations s'exécutant sur le cluster 64-bit produisent des résultats tronqués. La figure 3.1 révèle en effet que les opérations **FFN** et **NFN** représentent, en moyenne, une portion négligeable de l'exécution. De plus, notre analyse a montré que les opérations *load* chargent fréquemment des données tronquées depuis le cache de données. Ceci est la raison justifiant l'interconnexion de l'unité load/store au cluster distant. Étant donné l'usage fréquent des opérations *load* à l'exécution, nous supposons que ces opérations mettent systématiquement à jour les deux fichiers de registres 16-bit pour préserver les performances.

Dans notre schéma, l'unité de calcul 64-bit peut également produire des résultats tronqués. Nous pouvons cependant constater que seule l'unité load/store est connectée au cluster distant. Il est donc possible que le contenu du fichier de registres 16-bit ne soit pas cohérent à un moment donné de l'exécution. Pour maintenir cette cohérence, nous supposons que le matériel peut générer automatiquement une *instruction de copie* [66]. Cette instruction permet de propager une valeur arbitraire du cluster 64-bit au cluster tronqué. Cette instruction est insérée dans le tampon de réordonnancement (ROB) global ainsi que dans la queue d'instructions du cluster producteur. Cette dernière est traitée de manière similaire aux instructions du programme pour simplifier la logique de séquencement et la gestion des exceptions. Le processus de renommage est toutefois différent. Un registre physique libre est alloué dans le cluster consommateur mais la table de correspondance préserve l'entrée précédente pointant sur le registre physique 64-bit. Ce qui assure la cohérence du contenu du fichier de registres 16-bit. Nous pouvons toutefois remarquer que l'utilisation des instructions de copie est rare à l'exécution. La figure 3.1 montre en effet que les seules opérations (**FFN** et **NFN**) produisant un résultat tronqué dans le cluster 64-bit contribuent, en moyenne, à moins de 3% de l'exécution. Les autres opérations produisant un résultat tronqué (**NNN**) seront théoriquement séquencées sur le cluster tronqué. Le *duplicata 16-bit* agit donc à la fois comme un médium de communications et un fichier de registres 16-bit local. Contrairement aux propositions antérieures qui reposent un schéma complexe pour optimiser l'activité du fichier de registres [27, 71], notre schéma permet une réduction significative de la complexité sans ajout de mécanismes extérieurs, comme exposé en section 3.5.

3.3.2.2 Optimisation de la connexité inter-cluster

Nous avons également examiné une implémentation alternative du WPM pour optimiser la connexité entre les clusters. Cette nouvelle organisation, désignée *WPM optimisé*, permet de simplifier davantage la complexité du modèle WPM. Dans cette organisation, le nombre de ports en écriture du *duplicata 16-bit* est réduit par un facteur 2, soit 2 ports en écriture. Le bus de données connectant une unité de calcul 16-bit au *duplicata 16-bit* est ainsi supprimé (chemin étiqueté (2) sur la figure 3.4). Ce qui signifie qu'une seule unité 16-bit peut désormais propager les valeurs produites dans le cluster tronqué au cluster distant. Notre analyse a montré que peu d'opérations (**NFN** et **FFN**) s'exécutant sur le cluster 64-bit produisent des résultats tronqués. Il est donc consistant de supprimer le bus de données reliant l'unité de calcul 64-bit au *duplicata 16-bit* (chemin étiqueté (1) sur la figure 3.4). Faire ainsi contraint toutefois les opérations produisant des résultats tronqués à s'exécuter sur l'unité *load/store*. L'unité de calcul 64-bit peut toujours produire des résultats tronqués mais ces derniers devront être écrits sur le fichier de registres 64-bit. Bien que ce nouveau schéma puisse permettre une meilleure utilisation des ressources, il doit être toutefois noté que ce dernier peut aussi diminuer le nombre d'opportunités.

Il est également possible d'optimiser le nombre de communications inter-cluster car une faible partie des opérations s'exécutant sur le cluster 64-bit produisent des résultats tronqués. Nous proposons de recourir à un modèle par instructions de copie [66] pour ce faire. Ce modèle permet une mise à jour à la demande du contenu des fichiers de registres 16-bit. Cette approche peut réduire significativement la consommation des fichiers de registres et celle des interconnexions. En revanche, cette dernière peut également avoir un impact négatif sur les performances. L'exécution des instructions dépendantes de clusters distants doit être en effet différée jusqu'à ce le résultat soit disponible localement. Contrairement à un modèle par instructions de copie classique, nous supposons que ce sont les instructions consommatrices qui commandent la mise à jour du fichier de registres. Nous considérons toutefois que les opérations *load* sont systématiquement propagées pour réduire l'impact sur les performances, comme pour le WPM basique. Néanmoins, une approche plus efficace serait de considérer un *prédicteur d'usage* qui puisse prédire sur quel cluster une valeur est susceptible d'être consommée. Ce schéma éviterait le recours aux instructions de copie qui peuvent complexifier la logique de séquencement des instructions. L'étude de cette approche est toutefois réservée à des recherches ultérieures.

3.4 Allocation des instructions aux clusters

Diverses stratégies [10, 19, 30, 65, 8] ont été envisagées pour assigner une instruction à un cluster donné. Certaines études [30, 65] se basent sur un schéma statique alors que d'autres [10, 19, 8] considèrent un schéma dynamique pour ce faire. Les divers mécanismes d'allocation proposés reposent toutefois sur des heuristiques similaires qui tentent de minimiser le nombre d'inter-communication et le mauvais équilibrage des charges. La section 3.2 a montré que, pour le WPM, ces deux facteurs sont fortement corrélés à la distribution des opérations tronquées durant l'exécution. Ceci pourrait ainsi indiquer que la fonction unique de notre mécanisme d'allocation consisterait donc à révéler les opérations tronquées du programme. Plusieurs études [55, 71] ont mis en évidence la forte prédictabilité de la dimension des données. Ces études montrent que des schémas très simples réalisent des taux de prédiction satisfaisants, en moyenne 95%. Cette section considère des prédicteurs de largeur simples pour exposer au matériel les opérations tronquées. Il est possible d'appliquer des techniques plus élaborées telles que celles employées pour la prédiction de branchements [84]. Ceci n'est toutefois pas la finalité de notre étude. Cette section détaille plusieurs implémentations matérielles de prédicteurs de largeur. L'intégration de ce prédicteur au cœur du mécanisme d'allocation ainsi que le support pour recouvrer l'exécution sont également décrits.

3.4.1 Prédicteurs de largeur

Dans notre étude, le prédicteur de largeur permet de discriminer les opérations produisant un résultat tronqué du reste des instructions du programme. Ce dernier est utilisé pour prédire le comportement des opérations procédées par les unités entières de calcul ainsi que celui des opérations *load*. Comme la prédiction de largeur est basée sur l'adresse des instructions, nous supposons que l'accès à la table du prédicteur est localisé hors du chemin critique du processeur. Notre étude distingue deux types de mauvaises prédictions. Une mauvaise prédiction *conservatrice* se produit lorsque la dimension du résultat est plus large que celle ayant été prédite. Une mauvaise prédiction *effective* correspond à la prédiction d'une dimension plus faible que celle du résultat courant. Nous avons évalué plusieurs prédicteurs de largeur exhibant un compromis différent entre les mauvaises prédictions conservatrices et effectives.

3.4.1.1 Prédicteur de largeur « *comme la dernière fois* »

Le schéma de prédiction de largeur « *comme la dernière fois* » consiste à présumer que le résultat de l'opération traitée aura la même largeur que son dernier résultat produit. Ce schéma est relativement basique mais démontre la forte prédictabilité de la largeur des données dynamiques. L'implémentation de ce prédicteur de largeur sollicite une complexité matérielle simple. Ce prédicteur utilise en effet une table unique dont les entrées contiennent 1-bit. Ce bit permet de différencier les résultats tronqués des autres résultats produits. La table de prédiction est indexée à travers une fonction de hachage sur l'adresse des instructions. Nous supposons initialement que chaque entrée de la table indique une largeur 64-bit. Ces entrées sont mises à jour en fonction de la largeur du résultat produit.

3.4.1.2 Prédicteur de largeur bimodal

Le principe du prédicteur de largeur bimodal est similaire à celui employé dans la prédiction de branchements [84]. Ce prédicteur implémente un automate à deux modes permettant de différencier un résultat tronqué d'un résultat plus large. Chaque mode est subdivisé en un état fort et un état faible. Le prédicteur implémente une table de compteurs à saturation pour modéliser cet automate. Cette table est indexée à travers une fonction de hachage sur l'adresse des instructions. La position initiale de l'automate correspond à une dimension conservatrice 64-bit. L'automate est mis à jour de la manière suivante. Il passe ou reste à l'état fort du mode courant si la prédiction est correcte. Si la prédiction est mauvaise, deux cas sont possibles. Il passe à l'état faible du mode courant si l'état précédent était fort. Il change de mode autrement. L'intérêt de ce prédicteur par rapport au précédent est de renforcer la précision des prédictions grâce à un compteur à saturation.

3.4.1.3 Prédicteur de largeur avec compteur de remis-à-zéro

Ce prédicteur de largeur implémente une table de compteurs à saturation indexée à travers une fonction de hachage sur l'adresse des instructions. Chaque compteur est initialisé avec une valeur nulle. Tant que le compteur n'est pas arrivé à saturation, la largeur est prédite 64-bit. Autrement, le résultat est prédit tronqué. Le compteur est incrémenté à chaque fois qu'une opération produit un résultat tronqué. Le compteur est réinitialisé lorsqu'une opération produit un résultat 64-bit. L'intérêt d'un tel schéma par rapport aux prédicteurs précédents est de minimiser le taux de mauvaises prédictions effectives. Ce schéma augmente cependant le taux de mauvaises prédictions global.

3.4.1.4 Évaluation des schémas de prédiction

Nous avons évalué ces prédicteurs de largeur pour différentes tailles de tables de prédiction. Un nombre d'entrées variant de 4096 à 16392 a été ainsi considéré. Les fi-

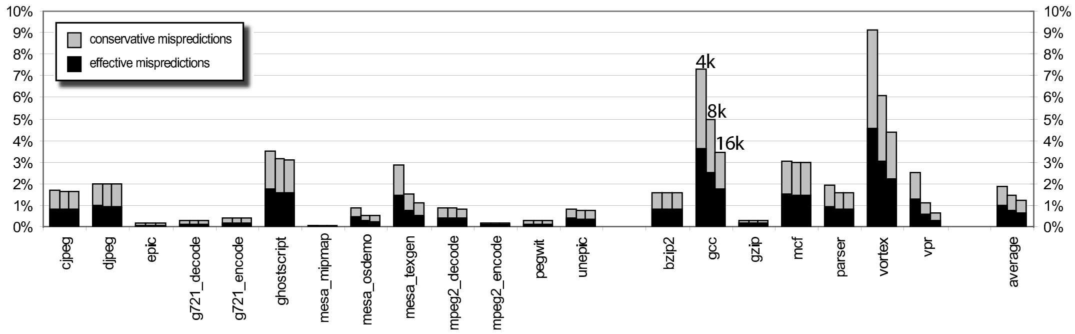


FIG. 3.6 – Taux de mauvaises prédictions pour un prédicteur « *comme la dernière fois* ». Chaque barre correspond respectivement à un nombre d'entrées de 4K, 8K, et 16K.

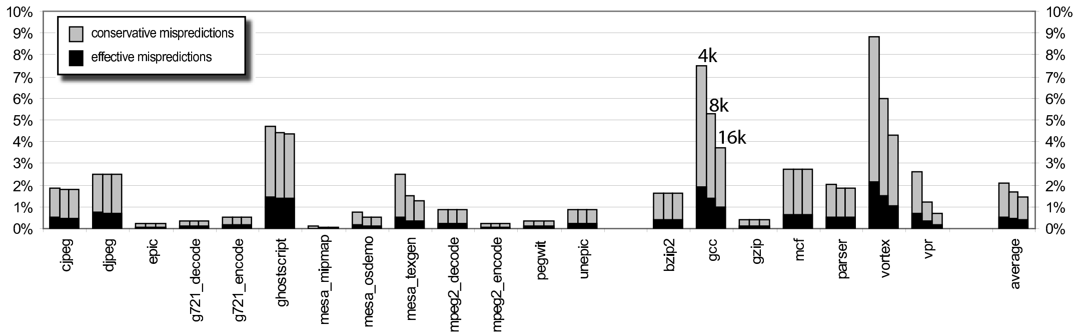


FIG. 3.7 – Taux de mauvaises prédictions pour un prédicteur 2-bit bimodal. Chaque barre correspond respectivement à un nombre d'entrées de 4K, 8K, et 16K.

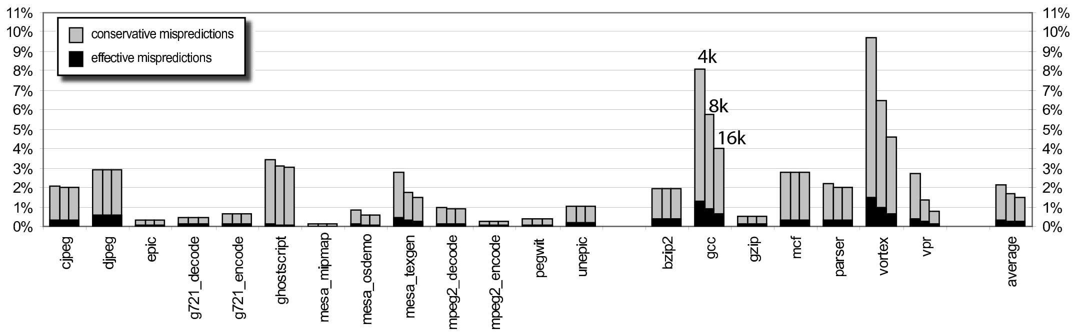


FIG. 3.8 – Taux de mauvaises prédictions pour un prédicteur 2-bit bimodal avec remis-à-zéro. Chaque barre correspond respectivement à un nombre d'entrées de 4K, 8K, et 16K.

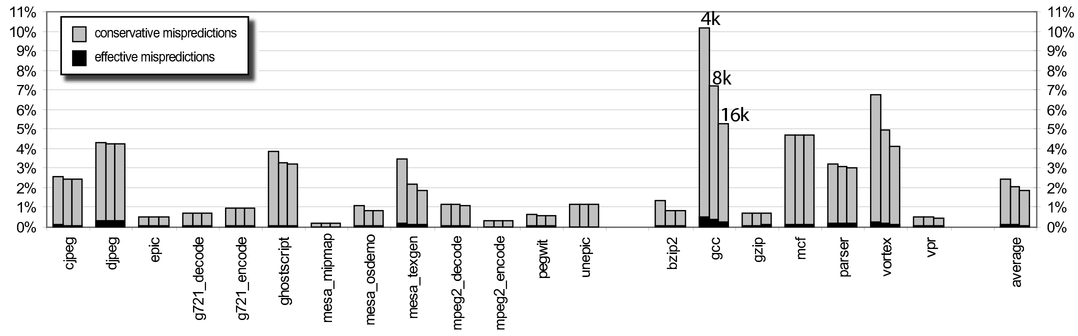


FIG. 3.9 – Taux de mauvaises prédictions pour un prédicteur 3-bit bimodal avec remis-à-zéro. Chaque barre correspond respectivement à un nombre d'entrées de 4K, 8K, et 16K.

gures 3.6, 3.7, 3.8 et 3.9 présentent les résultats obtenus. Dans chaque figure, nous avons distingué les mauvaises prédictions conservatrices des mauvaises prédictions effectives. Pour le schéma de prédiction « *comme la dernière fois* », nous remarquons que le taux moyen de mauvaises prédictions effectives avoisine 1% pour une table à 4096 entrées. Notons également que les taux de mauvaises prédictions conservatrices et effectives sont pratiquement équivalents. Pour le schéma de prédiction bimodal, nous observons un taux moyen de mauvaises prédictions effectives de l'ordre de 0.5% pour une table à 4096 entrées. Nous pouvons remarquer ce prédicteur exhibe toutefois un taux global de mauvaises prédictions supérieur au schéma précédent mais notre étude ne considère uniquement que les mauvaises prédictions effectives. Pour le dernier prédicteur, les taux moyens en mauvaises prédictions effectives sont largement inférieurs : 0.2 et 0.1% pour une table à 4096 compteurs 2-bit et 3-bit respectivement. Le schéma avec compteur à remis-à-zéro permet en effet d'éviter les mauvaises prédictions effectives en contrepartie d'un taux global en mauvaises prédictions supérieur. Nous avons aussi mesuré le taux de mauvaises prédictions pour des compteurs n-bit avec $n > 3$ mais ces derniers s'avèrent équivalents. Remarquons également que plusieurs applications exhibent un taux de mauvaises prédictions conservatrices élevé. Nous avons observé que ces dernières peuvent être optimisées en considérant des tables de prédiction plus larges.

Nous avons aussi analysé la *couverture*, c-à-d. *le ratio entre le nombre d'opérations tronquées correctement prédites et le nombre d'opérations tronquées totales à l'exécution*, réalisée par le schéma de prédiction de largeur bimodal avec compteur de remis-à-zéro. La table 3.1 affiche les résultats obtenus pour une configuration à 4096 entrées dotées de compteurs 3-bit. Nous pouvons ainsi remarquer que le schéma de prédiction avec remis-à-zéro offre une couverture relativement large (en moyenne 94%) pour une configuration raisonnable en nombre d'entrées. Au niveau du WPM, ceci indique qu'un schéma simple peut permettre d'identifier une large partie des opérations tronquées tout en limitant l'impact au niveau de la complexité. La section 3.6.5 exhibe de manière plus

Mediabench	jpeg	epic	g721	ghostscript	mesa	mpeg2	pegwit
Predictor coverage	.95	.98	.98	.95	.99	.84	.99
Spec2000	bzip2	gcc	gzip	mcf	parser	vortex	vpr
Predictor coverage	.98	.84	.99	.93	.97	.84	.95

Tab. 3.1 – Couverture réalisée pour un prédicteur de largeur à 4096 entrées 3-bit (remis-à-zéro).

détaillée l'impact du prédicteur de largeur à 4096 entrées 3-bit sur la consommation du processeur.

3.4.2 Allocation spéculative des instructions

Nous supposons que la décision relative à l'allocation des instructions est effectuée à l'étage de renommage. En effet, l'affiliation physique des opérandes registres d'une instruction détermine, en partie, le cluster auquel elle sera associée. Le résultat de la prédiction de largeur complète ensuite cette décision. Remarquons que l'heuristique proposée ne se base sur aucune information concernant l'équilibrage des charges. À vrai dire, ceci est consistant avec notre approche car nous avons montré que les opérations tronquées sont bien distribuées tout au long de l'exécution. Le mécanisme d'allocation du WPM pourrait ainsi répartir les charges de manière équilibrée sans avoir recours à une logique complexe. Il aurait été possible de considérer des heuristiques plus sophistiquées basées sur les dépendances entre instructions [66]. Cette approche est toutefois susceptible de placer le mécanisme d'allocation des instructions dans le chemin critique. Les implémentations du modèle WPM proposées considèrent une seule unité mémoire *load/store*. De ce fait, le mécanisme d'allocation assigne systématiquement les opérations mémoire au cluster 64-bit. Notons que cette contrainte peut influer sur l'équilibrage des charges. Notons que le processus de renommage a connaissance de l'affiliation des registres logiques aux fichiers de registres physiques. Ceci peut être réalisé à travers une numérotation spécifique des registres physiques, e.g. numéro physique pair/impair [65].

L'heuristique proposée pour assigner les instructions vers les clusters est la suivante. Soit I, l'instruction à assigner. Selon l'affiliation des opérandes registres de I, deux cas sont possibles :

– *Tous les opérandes sources de I résident dans un fichier de registres 16-bit.* Dans ce cas, si le prédicteur indique un résultat tronqué, *I* doit être assignée au cluster tronqué. Autrement, *I* sera assignée au cluster 64-bit. À noter que le résultat de *I* sera écrit dans le fichier de registres 16-bit.
– *Un opérande source de I réside dans le cluster 64-bit.* Dans ce cas, nous prenons la décision conservatrice d'assigner *I* au cluster 64-bit. Si le prédicteur indique un résultat tronqué, *I* écrira son résultat dans le *duplicata 16-bit*. Autrement, le résultat de *I* sera écrit dans le fichier de registres 64-bit.

3.4.3 Mécanisme de recouvrement

Nous suggérons de recourir à un mécanisme relativement simpliste pour recouvrir des mauvaises prédictions de largeur. Ce mécanisme permet de détecter et traiter ces mauvaises prédictions à la fin de l'étage d'exécution. Nous supposons que la détection est réalisée à travers une logique de détection des zéros disponible dans plusieurs implémentations commerciales [12]. Contrairement à [55], il est impossible de recourir uniquement à la logique de retenue des ALUs pour la détection car nous devons aussi prédire la largeur des données chargées depuis la mémoire. Le mécanisme de recouvrement proposé est basé sur un *refetch replay scheme* [47]. Aussi, nous considérons que le recouvrement de l'exécution est nécessaire uniquement lors de mauvaises prédictions effectives. Les mauvaises prédictions conservatrices n'affectent pas l'exécution mais la mise à jour de la table de prédiction est toutefois nécessaire. Dans le cas d'un recouvrement, le pipeline de chaque cluster est vidé pour libérer les ressources afin d'éviter un interblocage éventuel de l'exécution. Les entrées concernées de la table de prédiction sont alors mises à jour et les instructions réassignées au cluster adéquat. Ce mécanisme de recouvrement est relativement similaire à celui utilisé pour la prédiction de branchements. L'impact sur les performances équivaut donc à celui sollicité par une mauvaise prédiction de branchements. De plus, il est possible de partager la logique, réalisant le recouvrement, pour rendre négligeable la complexité requise par notre mécanisme. Il faut également noter que l'implémentation d'un mécanisme de recouvrement plus avancé [35] peut permettre de réduire l'impact sur les performances mais au détriment de la complexité du processeur.

3.5 Complexité du modèle WPM

Le modèle WPM bénéficie des avantages en complexité suscités par un modèle clusterisé conventionnel mais également de plusieurs autres points permettant de réduire davantage la complexité du processeur. Cette section analyse ces divers points par rapport au modèle clusterisé de référence détaillé en section 3.3.1. Cette analyse considère, pour ce faire, les deux implémentations du WPM : le *WPM basique* et le *WPM optimisé* dont les configurations sont décrites en section 3.3.2 et 3.3.2.2. Nous comparons ainsi la complexité de notre modèle en ce qui concerne le fichier de registres, le réseau de *bypass* de données, les logiques de séquencement et les interconnexions.

3.5.1 Fichier de registres

La complexité du fichier de registres est essentiellement déterminée par trois paramètres présentés en section 1.3.1 : la surface de silicium, le temps d'accès et la consommation. Le modèle WPM permet de réduire efficacement chacun de ces paramètres par le biais du découplage des opérations tronquées sur un chemin de données de taille plus appropriée. Ce découplage contraint aussi les opérations tronquées à lire et écrire leurs opérandes depuis un fichier de registres 16-bit. Les résultats présentés dans cette section ont été obtenus à partir d'une version modifiée de CACTI dont la partie relative aux étiquettes a été omise [43]. Pour ces résultats, nous supposons un procédé de gravure CMOS 0.13μm.

Surface de silicium L'équation 3.1 fait ressortir la surface de silicium occupée par le fichier de registres en fonction du nombre de ports N_{read} en lecture, du nombre de ports N_{write} en écriture, du nombre N_{regs} de registres et du nombre de bits R_{width} par registres (cf. section 1.12).

$$N_{regs} \times R_{width} \times \underbrace{\omega^2 \times (N_{read} + N_{write}) \times (N_{read} + 2 \times N_{write})}_{taille\ du\ point\ memoire} \tag{3.1}$$

Dans le modèle WPM, le fichier de registres est partitionné physiquement en trois sous-fichiers distincts dont les dimensions correspondent à 64-bit, 16-bit et 16-bit. Ces fichiers implémentent une largeur et un nombre de ports réduits de part les propriétés observées en section 3.2. La table 3.2 expose les gains en surface réalisés par notre stratégie de partitionnement. Pour les modèles WPM basique et optimisé (*limited WPM*), la surface de silicium occupée par le fichier de registres du cluster tronqué est réduite de 81%. Ceci est dû à la réduction par un facteur 4 de la taille des registres (16-bit au lieu de 64-bit) et à la réduction du nombre de ports en écriture sur cette structure (3 au lieu de 4). Pour le WPM basique, la surface occupée par le fichier de registres est réduite de 34% car le nombre de ports en écriture sur le fichier 64-bit est divisé par 2 ; de même pour le nombre de ports en lecture sur le *duplicata 16-bit* dont la dimension est divisée par 4. Pour le WPM optimisé, la surface de silicium est davantage réduite car le nombre de ports d'accès en lecture/écriture sur le *duplicata 16-bit* est divisé par 2. Nous pouvons ainsi remarquer que le découplage des données tronquées permet de réduire significativement la surface du fichier de registres par rapport à un modèle conventionnel malgré un nombre plus important de sous-fichiers de registres physiques. Ce modèle impacte toutefois sur la complexité du réseau de *bypass* de données.

	Conventional		WPM basic		WPM limited	
cluster	0	1	0	1	0	1
RF	1	1	2	1	2	1
reg. width (in bits)	64	64	64/16	16	64/16	16
nb of registers	80	80	80/80	80	80/80	80
(R,W) ports	(4,4)	(4,4)	(4,2)/(2,4)	(4,3)	(4,2)/(2,2)	(4,3)
RF area (x ω^2)	491520	491520	245760 + 76800	89600	245760 + 30720	89600
Area reduction	-	-	34%	81%	43%	81%
RF access time	0.6326	0.6326	0.6000/0.5278	0.5342	0.6000/0.4916	0.5342
Access time reduction	-	-	min(6%,16%)	15%	min(6%,22%)	15%
RF nJ/access	0.5431	0.5431	0.4267/0.1977	0.3500	0.4267/0.1571	0.3500
Energy reduction	-	-	21%/63%	35%	21%/71%	35%

TAB. 3.2 – Estimation de la complexité du fichier de registres. La consommation d'énergie correspond à l'énergie moyenne par accès en lecture/écriture.

Temps d'accès Le temps d'accès au fichier de registres est dépendant des délais de propagation des signaux dans les lignes de bit et de mot (cf. section 1.3.1.2). Le modèle WPM permet de diminuer ces délais en réduisant le nombre de ports et le nombre de bits par registre. La table témoigne 3.2 de cette réduction réalisée par notre proposition et révèle ainsi une réduction moyenne du temps d'accès de 15% pour les modèles WPM basique et optimisé. La réduction du délai est plus importante dans le cluster 64-bit du WPM optimisé pour des raisons analogues citées ci-dessus.

Consommation d'énergie La consommation du fichier de registres est dépendante à la fois du nombre de registres, du nombre de ports d'accès et du nombre de bits par registres (cf. section 1.3.1.2). Dans le modèle WPM, le nombre de ports d'accès et le nombre de bits par registres sont réduits de part le découplage des opérations tronquées. La table 3.2 affiche ainsi un gain moyen en consommation de 35% pour les modèles WPM basique et optimisé par rapport au modèle de référence. La réduction du nombre de ports permet de réduire la longueur des interconnexions et donc leurs capacités. La réduction du nombre de bits permet également un tel effet tout en réduisant le nombre de lignes de bit sollicitées à chaque accès. Nos résultats prouvent que le modèle WPM consomme moins d'énergie malgré l'utilisation des copies d'instructions et du mécanisme de recouvrement.

3.5.2 Réseau de *bypass*

La complexité du réseau de *bypass* est fonction de la profondeur et du nombre de bus implémentés pour propager la donnée aux étages antérieurs. Dans le WPM basique, chaque opération peut obtenir ses opérandes depuis 6 sources distinctes : l'unité de calcul 64-bit, l'unité mémoire, les deux unités 16-bit du cluster tronqué, le *duplicata 16-bit* et le fichier de registres 64-bit. Ceci implique donc que la taille des multiplexeurs et le nombre de bus seront plus importants que ceux implémentés dans une configuration conventionnelle : e.g. 6 entrées au lieu de 5 par multiplexeur. Ceci peut donc avoir un effet négatif sur le délai, la consommation et la surface du réseau de *bypass* dans le cluster 64-bit. Pour le WPM optimisé, la complexité du réseau de *bypass* est équivalente à celle d'une implémentation clusterisée conventionnelle. Pour le cluster tronqué, la complexité du réseau de bypass est en revanche significativement diminuée car le nombre de sources est réduit à 4 au lieu de 5 et la largeur des bus est divisée par un facteur 4. Même si notre modèle augmente sensiblement la complexité du réseau de *bypass* de données pour le WPM basique, nous montrons que les gains en complexité réalisés dans les autres composants peuvent rendre négligeable cette augmentation.

3.5.3 Logiques de séquencement

La complexité des logiques de séquencement dépend de plusieurs paramètres (cf. section 1.3.3) dont le nombre de sources à monitorer par cycle. Pour un jeu d'instructions dyadique et un processeur capable de produire jusqu'à N instructions par cycle, $2 * N$ comparateurs sont ainsi nécessaires pour marquer les instructions prêtes à être séquencées. Pour le WPM basique, 4 sources distinctes doivent être monitorées sur le cluster 64-bit alors que seulement 3 sont nécessaires sur le cluster tronqué. Notre modèle est donc supérieur à une implémentation conventionnelle dont le nombre de comparateurs est équivalent à 4 pour chaque cluster. Pour le WPM optimisé, notre modèle réduit davantage la complexité de la logique de réveil en enlevant une source à monitorer (cf. figure 3.4) ; réduisant ainsi le nombre de comparateurs de 4 à 3.

3.5.4 Réseau d'interconnexion inter-cluster

Le modèle WPM permet une réduction significative de la surface, la consommation et le délai du réseau d'interconnexion chargé de communiquer les données entre le cluster 64-bit et le cluster tronqué. Les échanges de données entre ces deux clusters concernent en effet uniquement les données tronquées. Les interconnexions peuvent donc être dimensionnées en conséquence pour réduire les paramètres cités. La surface de silicium est ainsi réduite par un facteur 4 (16-bit au lieu de 64-bit) en se référant à l'équation reportée en section 1.3.4. La consommation des interconnexions est également considérablement réduite car le nombre de bus est restreint ; d'où également une activité de commutation vraisemblablement diminuée. Il est possible de réduire davantage la consommation et le délai des interconnexions en se référant à l'étude de Balasubramonian et al. [9]. Ces derniers ont exhibé différentes configurations des interconnexions offrant un compromis consommation/performance distinct. Pour les données tronquées, ces auteurs montrent qu'il est ainsi possible d'utiliser des bus de données à la fois performants et à faible consommation. Ceci est possible en augmentant la largeur des bus et l'espace de séparation entre ces derniers (cf. section 1.3.4). Ceci permet de réduire les délais et la consommation en contrepartie d'une surface de silicium pratiquement équivalente à un modèle conventionnel. Balasubramonian et al. montrent que des bus de données 20-bit aux propriétés citées peuvent être implémentés dans une microarchitecture moderne. Ces auteurs reportent une réduction de 70% du délai de propagation

Parameter	Configuration	Parameter	Configuration
Fetch width	4	FUs (16-bit)	2 ALUs
Decode width	4	Register file size	80 per RF
Issue width	2/cluster	ROB size	64
Retire width	4	LSQ size	16
Fetch queue size	16	Branch predictor	16k-entry GShare
Issue queue size	16/cluster	Data-width predictor	4k-entry 3-bit bimodal
FUs (64-bit)	ALU+LD/ST	Misprediction penalty	min. 6 cycles

TAB. 3.3 – Paramètres de simulation.

Homogeneous interconnect delay	2 cycles
Heterogeneous interconnect delay	1 cycle
Homogeneous interconnect relative energy	1.0
Heterogeneous interconnect relative energy	0.84
4k-entry 3-bit data-width predictor (1R/1W ports) energy	0.112 nJ

TAB. 3.4 – Délai et énergie considérés pour les expérimentations.

et de 16% de la consommation dynamique à travers l'utilisation de ces interconnexions hétérogènes. Le modèle WPM peut implémenter ces interconnexions hétérogènes pour bénéficier d'un réseau d'interconnexion inter-cluster performant et à faible consommation. Les résultats expérimentaux exposés dans la section suivante corroborent à ces attentes.

3.6 Évaluation expérimentale

Les sections précédentes ont démontré la manière par laquelle le modèle WPM peut réduire la complexité du matériel par rapport à un processeur clusterisé conventionnel. Dans cette section, nous présentons une évaluation du modèle WPM pour comparer ses performances et sa consommation par rapport au modèle de référence, présenté en section 3.3.1.

3.6.1 Méthodologie

Toutes nos expériences ont été effectuées à partir d'une version modifiée de MASE, simulateur de microarchitectures dérivé de SimpleScalar 3.0 [51]. Notre version modélise un processeur superscalaire dont les cœurs d'exécution entier et flottant sont partitionnés entre différents clusters. Les microarchitectures considérées prennent également en compte les contentions au niveau des interconnexions, des queues d'instructions, des fichiers de registres et du renommage physique. Le mécanisme d'allocation des instruc-

Mediabench		
Application	**Description**	**Input**
epic	image compression	test_image.pgm [.pgm.E]
g721	voice compression	clinton.g721 [.pcm]
ghostscript	postscript interpreter	test.ppm
jpeg	image compression	testout.ppm [.jpeg]
mesa	3D graphics library	–
mpeg2	video compression	mei16v2.m2v out.m2v
pegwit	encryption algorithm	pegwit.enc [.dec]
SPECInt2000		
Application	**Description**	**Input**
bzip2	data compression	input.graphic
gcc	C compiler	scilab.i
gzip	data compression	input.random
mcf	combinatorial optimization	inp.in
parser	word processing	ref.in
vortex	database	lendian1.raw
vpr	FPGA place route	place.in

Tab. 3.5 – Applications Mediabench et SPEC2000.

tions est basé sur un prédicteur de largeur implémentant une table de 4096 compteurs 3-bit avec remis-à-zéro 3.4.1.3. La table 3.3 détaille les divers paramètres microarchitecturaux considérés lors de cette évaluation. Les délais de traversée et la consommation des interconnexions homogènes et hétérogènes ont été dérivés de [9] et sont présentés en table 3.4. Deux implémentations du modèle WPM sont comparées à la microarchitecture de référence. Ces implémentations concernent le WPM basique (section 3.3.2.1) et le WPM optimisé (section 3.3.2.1).

3.6.2 Applications représentatives

Nous avons mesuré les performances et la consommation du WPM à partir d'un ensemble d'applications sélectionnées parmi les suites MediaBench et SPEC2000. Chaque application a été compilée avec la version de gcc portée au jeu d'instructions PISA de SimpleScalar. Le jeu d'instructions PISA correspond à une version académique du jeu d'instructions MIPS. Chaque binaire a été généré avec les optimisations : *-02* et *-funroll-loops*. La table 3.5 présente le détail des benchmarks sélectionnés ainsi que les jeux de données fournis au simulateur. Les applications s'exécutant avec moins de 300 millions d'instructions ont été entièrement émulées. Pour les autres, la tranche initiale regroupant 100 millions d'instructions n'a pas été traitée puis une tranche de 200 millions d'instructions a été émulée.

3.6.3 Équilibrage des charges

L'équilibrage des charges est un critère essentiel à la performance des microarchitectures clusterisées. Un mauvais équilibrage des charges correspond à la situation où un cluster est surchargé alors que l'autre cluster possède des ressources disponibles pour le calcul. Cette situation peut en effet entraîner une dégradation des performances car la capacité d'exécution du processeur n'est pas exploitée de façon optimale. Nous avons estimé la répartition moyenne des charges pour le modèle WPM à partir de la métrique proposée par [19]. La répartition des charges est ainsi estimée par la différence du nombre d'opérations prêtes à s'exécuter dans chaque cluster à chaque cycle. Une métrique alternative aurait été de mesurer la différence du nombre d'instructions allouées aux clusters à chaque cycle. Cette métrique ne capture toutefois pas le degré de parallélisme présent dans la fenêtre d'instructions à un moment donné de l'exécution. La métrique [19] considère ainsi un équilibrage des charges parfait lorsque la différence du nombre d'opérations prêtes à s'exécuter est nulle. Autrement, la différence mesurée exprime une situation de mauvais équilibrage : une petite différence ($<$ à 5 dans notre cas) correspond à un équilibrage des charges correct alors qu'une différence plus grande implique une sous-utilisation des ressources du processeur.

L'équilibrage des charges pour le modèle de référence sont présentés en figure 3.10(a) et 3.10(b). Nous nous sommes basés sur une heuristique similaire à [19] pour assigner les instructions aux clusters du modèle de référence. Cette heuristique (*Balanced RBMS*) tente, à la fois, de minimiser le nombre d'inter-communication et le mauvais équilibrage des charges. Pour ce faire, les opérations dépendantes sont séquencées sur le même cluster alors que les opérations dépourvues d'opérandes registre sont affectées au cluster le moins chargé. Les résultats relatifs à l'équilibrage des charges du WPM basique sont présentés en figure 3.10(c) et 3.10(d). Ces résultats ont été obtenus avec un prédicteur de largeur bimodal de 4096 compteurs 3-bit avec remis-à-zéro.

La figure 3.10 montre ainsi que la répartition des charges est relativement similaire pour le modèle de référence et pour le WPM. Nous pouvons expliquer cette similitude par le fait que notre étude considère un modèle clusterisé asymétrique. Ainsi, nous pouvons remarquer que la répartition des charges est correctement équilibrée pour 60% de l'exécution en moyenne : (différences [0-2]). En moyenne, pour 80% de l'exécution, notre schéma réalise une répartition des charges relativement satisfaisante (différences [0-5]). Étant donnée la nature asymétrique du WPM, ces résultats semblent très prometteurs. Notre heuristique pour assigner les instructions aux clusters ne repose en effet sur aucune information relative à l'équilibrage des charges. Il serait donc possible d'améliorer ainsi les résultats obtenus. Ceci peut toutefois réduire le nombre d'opportunités relatif au séquencement des opérations tronquées.

3.6.4 Impact sur les performances

Pour une microarchitecture clusterisée conventionnelle, la dégradation des performances - mesurée en IPC - dépend essentiellement de l'équilibrage des charges et du

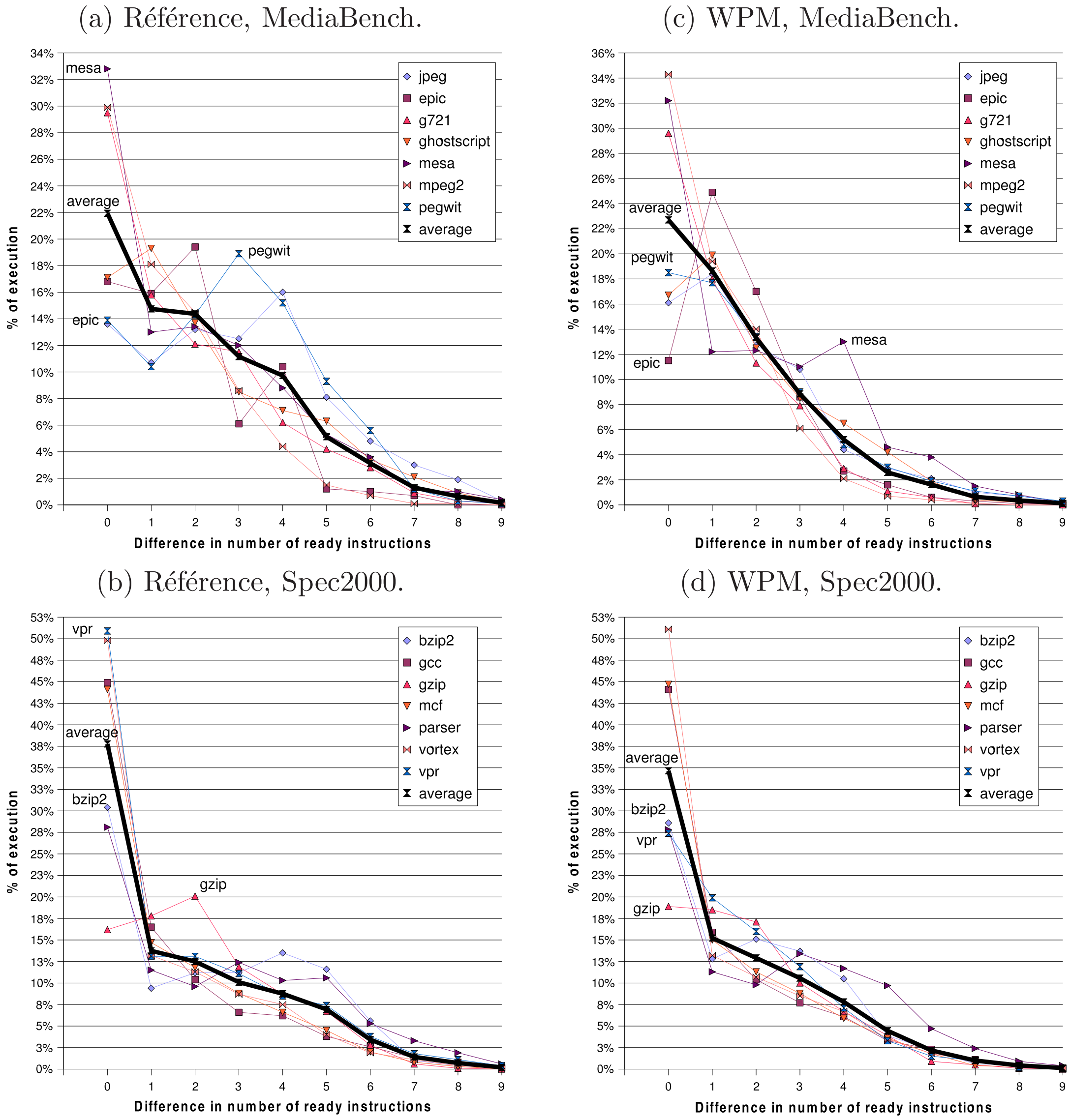


FIG. 3.10 – Répartition des charges pour les applications MediaBench et SPEC2000.

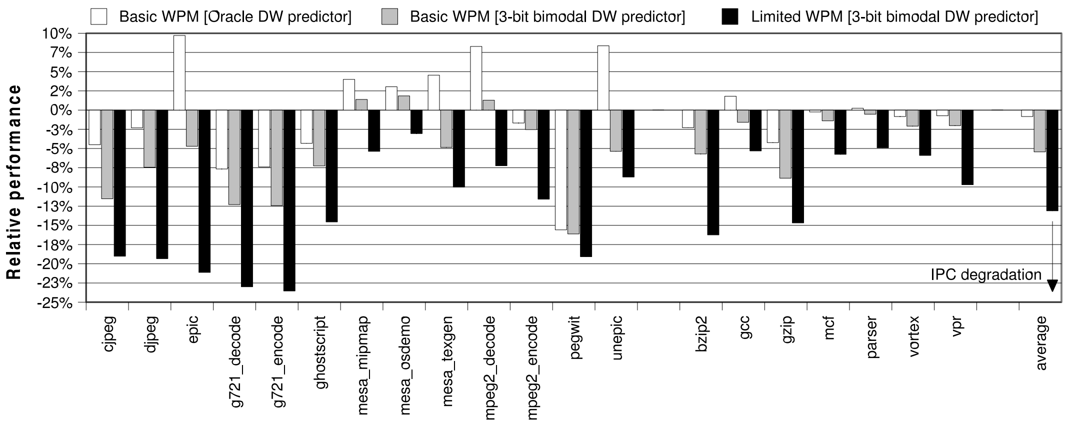


FIG. 3.11 – Impact sur les performances avec un prédicteur de largeur à 4096 entrées 3-bit.

nombre d'inter-communication. Pour le modèle WPM, les mauvaises prédictions de largeur constituent une source additionnelle de dégradation. Plusieurs cycles sont en effet nécessaires pour ramener le processeur à un état correct. Nous avons considéré un WPM basique implémentant un prédicteur de largeur parfait, ou oracle, pour mettre en évidence l'efficacité du modèle WPM sans l'impact dû aux mauvaises prédictions. La figure 3.11 affiche la dégradation des performances de différentes implémentations du WPM par rapport au modèle de référence. Le prédicteur de largeur utilisé pour générer ces résultats est le prédicteur bimodal 3-bit avec remis-à-zéro (*bim-rst wp*). Nous ne présentons pas dans cette étude les mesures de performance relatives aux autres prédicteurs de largeur.

WPM basique + oracle Nous observons que les performances de l'implémentation « *WPM basique + oracle* » avoisinent celles du modèle de référence. Quelques applications (epic, mpeg2_decode, unepic) exhibent même des performances supérieures avec cette implémentation du WPM. Nos expériences montrent que ceci est dû au mécanisme d'allocation du modèle référence qui réalise un mauvais équilibrage des charges sur ces applications. Ce phénomène peut s'expliquer par la nature asymétrique du WPM basique contraignant la majorité des instructions à s'exécuter sur le cluster contenant l'unité *load/store*. Les performances du « *WPM basique + oracle* » révèle ainsi le potentiel de notre modèle à réduire la complexité tout en préservant des performances satisfaisantes. Pour l'implémentation considérée, une heuristique d'allocation basée sur la distribution dynamique des opérations tronquées s'avère ainsi aussi efficace qu'une heuristique plus complexe.

WPM basique + prédicteur de largeur bimodal 3-bit avec remis à zéro Considérer un prédicteur de largeur réaliste influe toutefois légèrement plus sur les

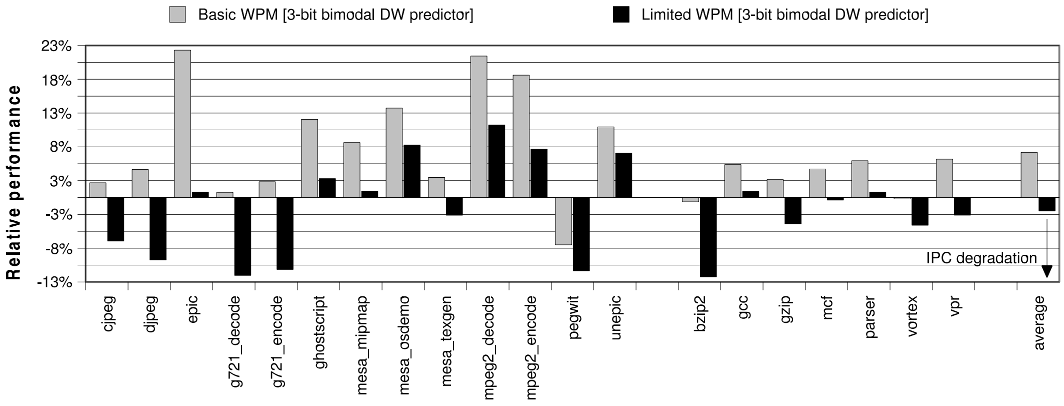


FIG. 3.12 – Impact sur les performances avec un prédicteur de largeur à 4096 entrées 3-bit et des interconnexions hétérogènes.

performances du modèle WPM. La figure 3.11 montre en effet que les performances sont dégradées de 6% en moyenne pour un WPM basique implémentant le prédicteur décrit en section 3.4.1.3. D'une part, cette dégradation des performances peut être attribuée au coût du recouvrement déclenché par une mauvaise prédiction de largeur. D'autre part, cette dégradation dépend aussi du nombre d'opérations tronquées identifiées par le prédicteur de largeur. Ce nombre affecte directement l'équilibrage des charges du WPM. Nous pouvons également observer que certaines applications (gcc, mcf, vortex) exhibent un taux élevé de mauvaises prédictions conservatrices. Les performances du WPM sur ces applications ne sont toutefois pas affectées car nous considérons exclusivement le recouvrement des mauvaises prédictions effectives. La figure 3.10 révèle aussi un équilibrage des charges correct sur ces applications.

WPM optimisé La dégradation additionnelle des performances observée pour le WPM optimisé est due à l'utilisation des instructions de copie. Ces instructions de copie permettent de synchroniser le contenu des fichiers de registres 16-bit. Ces dernières sont générées lorsqu'une opération nécessite un opérande distant. Le traitement de cette opération est alors différée du nombre de cycles nécessaires pour transiter les données. La latence des instructions de copie dépend ainsi directement de celle des interconnexions. La figure 3.11 montre que ce schéma a un impact négatif sur les performances. Nous observons une dégradation de 13% en moyenne pour les applications considérées. Cette dégradation peut toutefois être minimisée en considérant le *prédicteur d'usage* décrit en section 3.3.2.2.

Interconnexions hétérogènes Nous avons également mesuré les performances d'une implémentation composée d'un réseau d'interconnexion hétérogènes. Nous supposons que les données sont propagées deux fois plus vite dans cette implémentation. La figure 3.12 montre ainsi que les performances du WPM basique sont améliorées, en moyenne, de 6% en considérant des interconnexions hétérogènes. Les performances du WPM optimisé sont aussi proches de celles du modèle de référence. Les résultats obtenus considèrent pourtant une réduction pessimiste du délai de traversée des interconnexions. Balasubramonian et al. [9] considèrent notamment que la latence des interconnexions peut être réduite par un facteur 3 avec une largeur similaire à notre modèle. Ceci sous-entend alors un gain supérieur de notre modèle avec un réseau d'interconnexion hétérogènes.

3.6.5 Estimation de la consommation

Le partitionnement du processeur en clusters permet de simplifier considérablement la structure matérielle de la plupart de ses composants critiques. Faire ainsi résulte alors en une diminution de sa consommation d'énergie. Notre modèle WPM profite de tous les gains en énergie suscités par un modèle clusterisé conventionnel. De plus, le modèle WPM permet de réaliser des gains additionnels en énergie au niveau de plusieurs éléments du chemin de données. Dans cette évaluation, nous nous sommes focalisés sur les éléments critiques suivants : unités de calcul, fichier de registres et interconnexions.

En optimisant les unités de calcul pour le traitement des opérations tronquées, une réduction linéaire de la consommation est ainsi réalisée [71]. Dans notre modèle WPM, le nombre de ports en écriture et la largeur du fichier de registres ont été réduits en fonction de la distribution et des propriétés des opérations tronquées. La section 1.3.1 a montré que la consommation du fichier de registres est très sensible à ces deux paramètres. Les communications inter-cluster se produisant dans un WPM concernent uniquement la propagation de données tronquées. Ceci implique ainsi des communications inter-cluster moins fréquentes. La diminution du nombre de communications couplée au redimensionnement des interconnexions permettent alors une réduction significative de la consommation du réseau d'interconnexion.

Le modèle WPM nécessite toutefois des structures annexes pour assigner les instructions aux clusters et propager leurs résultats vers les fichiers de registres. Pour minimiser l'impact du prédicteur de largeur, nous avons limité la taille de la table d'historiques de manière à ce qu'elle dissipe une puissance limitée [67]. Nos résultats expérimentaux montrent que la consommation suscitée par ces composants additionnels est négligeable par rapport aux gains réalisés en énergie.

Réduction d'énergie La figure 3.13 révèle les gains en énergie par composant réalisés pour le modèle WPM basique et le modèle WPM optimisé vis-à-vis du modèle de référence. La première barre du graphe représente la réduction de la consommation pour les unités fonctionnelles (ALU). Nous assistons ainsi à une réduction moyenne de 20% pour le WPM basique et de 13% pour le WPM optimisé. Cette différence de gains s'explique notamment par la sollicitation plus fréquente des ALUs du WPM optimisé pour pouvoir traiter les instructions de copie. Il serait ainsi possible d'améliorer

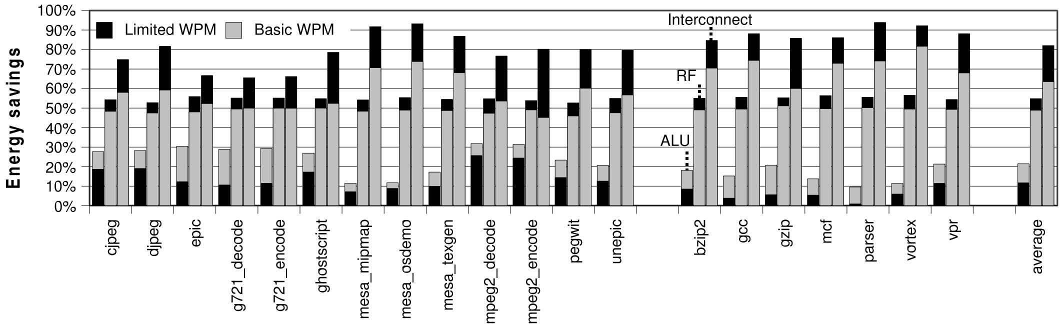


FIG. 3.13 – Power savings breakdown (ALU, RF, interconnect (IC)).

les gains du WPM optimisé en considérant une heuristique d'allocation plus optimale vis-à-vis du nombre de communications inter-cluster.

La seconde barre du graphe indique une réduction moyenne de 50% pour la consommation du fichier de registres dans les WPMs basique et limité. La différence de gains est relativement insignifiante car les instructions de copie du WPM optimisé sollicitent davantage le fichier de registres pour lire les données à propager.

La réduction de consommation réalisée dans les interconnexions représente les gains les plus importants dans notre modèle WPM. En effet, nous pouvons assister à 60% de réduction pour le WPM basique et 80% pour le WPM optimisé. Remarquons que ces résultats ne concernent pas des interconnexions à très faible consommation. Ces dernières peuvent ainsi réduire davantage la consommation d'énergie par un facteur 3 [9]. Étant donné que les processeurs modernes dissipent une fraction importante de l'énergie dans les interconnexions (50% dans le Pentium 4 [56]), notre modèle WPM pourrait donc avoir un impact significatif au niveau de la consommation totale.

Notons également que le modèle WPM peut également réduire la consommation statique des éléments cités. Balasubramonian et al. [9] montrent notamment que la consommation statique dans les interconnexions peut être divisée par un facteur variant de 1.26 à 3 en fonction de la largeur des bus de communications. La consommation statique du fichier de registres et des ALUs peut être aussi réduite de manière similaire à [71] mais sans le support de mécanismes complexes.

Consommation additionnelle Le modèle WPM peut induire une consommation additionnelle due à l'utilisation du prédicteur de largeur. Nous avons approximé la consommation d'un prédicteur à 4096 entrées 3-bit grâce à CACTI [43]. Nous avons considéré une table composée de 2 ports d'accès (1 en lecture et 1 en écriture) en supposant que 4 compteurs 3-bit peuvent être accédés simultanément. Nous considérons aussi que l'accès au prédicteur et sa mise à jour sont réalisés à l'étage de chargement (*fetch*) et de complétion (*commit*) du pipeline, respectivement. La table 3.4 affiche l'énergie par accès pour ce prédicteur de largeur. L'absence d'un outil approprié pour mesurer la consommation du prédicteur rend toutefois difficile la quantification de l'impact du prédicteur sur la consommation totale. Nous nous proposons de comparer la consommation du prédicteur de largeur à celle du fichier de registres pour mettre en avant sa contribution au niveau de la consommation. Nous considérons ainsi un fichier composé de 80 registres 64-bit implémentant 4 ports d'accès en lecture et 4 ports d'accès en écriture. En moyenne, nous avons mesuré que la consommation du prédicteur de largeur représente une consommation additionnelle de 6% par rapport à celle du fichier de registres. Ceci s'explique par le fait que le nombre d'accès au prédicteur sont très inférieurs par rapport à ceux effectués sur le fichier de registres, c-à-d. 12% en moyenne. Il est ainsi possible de négliger la consommation de composant par rapport aux gains réalisés par le WPM. Il faut noter qu'il aurait été possible de considérer un prédicteur de largeur basé sur une structure banquée pour réduire davantage l'impact sur la consommation du processeur.

3.7 Conclusions et directions futures

Dans ce chapitre, nous avons évalué un nouveau schéma de partitionnement basé sur la dimension des données pour réduire la complexité du processeur. Les propositions relatives à l'exploitation des données tronquées considèrent généralement un chemin de données unique pour appliquer leurs optimisations. Cette stratégie sollicite toutefois l'addition de plusieurs circuits spécialisés sur le chemin de données dont l'impact sur la complexité est souvent difficile à mesurer. Notre proposition se distingue de ces études en considérant plusieurs chemins de données taillés aux dimensions des opérations manipulées. Notre analyse des applications a en effet révélé la forte présence à l'exécution des opérations manipulant exclusivement des opérandes tronqués. Nous avons également remarqué que ces opérations sont relativement bien distribuées au long de l'exécution.

Pour exploiter ces propriétés inhérentes aux applications, nous avons proposé de découpler le traitement des opérations tronquées de celui des autres instructions du programme. Nous avons ainsi proposé le modèle clusterisé WPM et en avons évalué une configuration composée de deux clusters dimensionnés distinctement : un cluster 64-bit procédant les opérations complexes et mémoire et un cluster 16-bit optimisé pour le calcul des opérations de même dimension. Deux caractéristiques reflètent typiquement les performances des microarchitectures clusterisées : le nombre de communications et l'équilibrage des charges entre les clusters. Grâce aux propriétés observées, nous avons montré que la mise en œuvre d'une heuristique de distribution simple s'avère performante. Cette heuristique consiste uniquement à révéler les opérations tronquées du programme par le biais d'un prédicteur de largeur. Nous avons considéré plusieurs mécanismes de prédiction sollicitant une faible complexité matérielle mais exhibant déjà un taux satisfaisant en bonnes prédictions.

Nous avons également analysé en détails la complexité du modèle WPM. Le modèle WPM permet de limiter davantage la complexité du fichier de registres, des unités de calcul, des logiques de séquencement et du réseau d'interconnexion entre les clusters par rapport à un modèle clusterisé conventionnel. Notre évaluation expérimentale indique des performances avoisinant celles du modèle de référence et une consommation d'énergie significativement réduite. Les consommations du fichier de registres et des interconnexions sont pratiquement réduites de moitié alors que la dégradation des performances est légèrement inférieure à 7%. Nous avons détaillé une configuration plus optimisée du modèle WPM qui permet de réduire davantage la consommation pour des performances équivalentes au modèle clusterisé conventionnel.

Directions futures

Plusieurs travaux postérieurs peuvent être menés pour améliorer la complexité ou la performance du modèle WPM :

- **Affiner le mécanisme de prédiction de largeur.** Nos résultats expérimentaux ont montré que le prédicteur de largeur influe directement sur les performances du modèle WPM. Sur certaines applications (gcc, mcf, vortex), nous avons remarqué que le nombre d'interférences est relativement considérable. Une solution pour réduire ce nombre d'interférences consiste à considérer deux tables de prédiction associées à une fonction de hachage distincte. Une analyse plus approfondie de la sensibilité du prédicteur peut également

- **Évaluer le passage à l'échelle du modèle WPM.** Cette étude a considéré une configuration très simple du WPM pour démontrer son potentiel. Il est intéressant d'examiner comment le modèle WPM peut passer à l'échelle avec un nombre de clusters et une largeur d'exécution plus importants. Pour un nombre supérieur en clusters, le modèle de communications inter-cluster doit être révisé de manière à conserver la complexité de notre approche. Plusieurs modèles présentés dans la littérature [66, 86] peuvent être adaptés pour ce faire.

- **Partitionner le cache de données.** Nous nous sommes limités au partitionnement des ressources de calcul dans cette étude. Il est cependant envisageable d'appliquer nos optimisations au cache de données pour réduire sa complexité. Balasubramonian et al. [9] ont proposé plusieurs techniques pour ce faire. La nature asymétrique de notre modèle permet toutefois de faciliter le partitionnement du cache car seul un cluster 64-bit peut avoir accès à cette structure ; d'où une simplification du mécanisme de désambiguïsation mémoire.

- **Implémenter et évaluer le prédicteur d'usage.** La section 3.3.2.2 a présenté le recours à un prédicteur d'usage pour anticiper les opérandes requis par un cluster et ainsi optimiser le nombre de communications inter-cluster. Plusieurs approches présentant une complexité matérielle différente sont possibles pour ce faire. L'utilisation du compilateur ou d'un mécanisme matériel similaire à [31] peuvent notamment aider à identifier les instructions dépendantes.

- **Appliquer le modèle WPM aux processeurs embarqués**. L'implémentation du modèle WPM exposée dans cette étude a été essentiellement pensée pour une microarchitecture superscalaire. Le modèle clusterisé est toutefois aussi répandu dans les systèmes embarqués contemporains (e.g. le processeur VLIW Lx [28]). Il est donc possible de revisiter notre implémentation pour l'adapter aux processeurs VLIW. La distribution des instructions vers les clusters s'apparente ainsi à un problème d'ordonnancement pouvant être réalisé de manière similaire à [30]. Ceci implique alors que le mécanisme de prédiction de largeur doit être également statique. Il est possible de considérer le schéma décrit dans le chapitre précédent pour parvenir à un tel effet.

- **Considérer un processeur multicœur**. Nous avons appliqué les propriétés des programmes observées à un modèle clusterisé. Il est possible d'étendre notre modèle WPM aux processeurs multicœur (CMP) [25, 49]. L'application du WPM au CMP peut ainsi prendre différentes directions dont l'exploration des meilleures configurations en nombre de cœurs 16-bit et 64-bit pour améliorer la performance ou la consommation du processeur. Le modèle WPM impose toutefois le partage du cache d'instructions pour distribuer une application entre les divers clusters. Il est possible d'éviter ce dernier à travers l'exécution du même programme sur les différents cœurs.

Conclusions et perspectives

Les progrès incessants de l'industrie du semi-conducteur ont largement contribué au développement de processeurs performants lors de ces dernières décennies. Les concepteurs ont, en effet, pu profiter des ressources disponibles à chaque nouvelle génération pour intégrer des mécanismes toujours plus sophistiqués ; ce qui a ainsi favorisé l'expansion des performances. Néanmoins, appliquer cette stratégie de développement a également engendré la conception de processeurs dont la *complexité matérielle* et la *consommation électrique* n'ont cessé d'augmenter au fil des générations. L'amplitude de ces problèmes est telle aujourd'hui, qu'il devient difficile de concevoir des systèmes largement plus performants que leurs prédécesseurs.

Cette thèse se propose d'adresser ces problèmes de complexité dans les processeurs embarqués et superscalaires. L'essence de nos propositions provient de la démocratisation des jeux d'instructions 64-bit dans les systèmes informatiques contemporains. Cette tendance peut être notamment illustrée par le récent attrait des fabricants de processeurs Intel ou AMD pour les microarchitectures 64-bit. Du point de vue matériel, ceci se traduit toutefois en un dimensionnement respectif de la plupart des composants critiques du processeur. Cela implique donc une consommation et une complexité matérielle plus élevées pour les microarchitectures modernes.

De manière paradoxale, l'analyse des applications modernes révèle qu'une fraction considérable des données manipulées à l'exécution contiennent un nombre restreint de bits significatifs (*données tronquées*). La tendance actuelle pour exploiter cette propriété des programmes est d'opérer sur ces données de manière simultanée à travers les instructions multimédia (SIMD). Les techniques SIMD sont toutefois difficiles à mettre en œuvre et n'exploitent pas la totalité des données tronquées du programme. Plusieurs travaux académiques ont proposé des solutions pour tirer profit de cette propriété des programmes observée par Brooks [12]. Ces derniers se distinguent principalement selon la finalité ciblée, c-à-d. : (1) *la recherche de performances à travers l'augmentation du parallélisme d'instructions* ou (2) *la réduction de consommation à travers le redimensionnement des structures*. Nous avons proposé deux nouvelles solutions exploitant cette propriété pour réduire la complexité du chemin de données.

La première solution détaillée dans [71] se focalise sur la réduction des consommations dynamique et statique dissipées par le chemin de données. Cette solution est plus particulière aux systèmes embarqués. Nous avons observé que les opérations manipulant des données tronquées sont fortement concentrées au sein des blocs de base du programme. Nous avons proposé d'exploiter cette nouvelle propriété à travers le découpage du programme en régions sollicitant une même largeur d'exécution. Pour identifier les régions du programme, nous nous sommes appuyés sur une approche logicielle augmentée par les informations relatives aux exécutions antérieures du programme. Cette stratégie permet ainsi d'outrepasser le potentiel des techniques purement logicielles mais requiert un support adéquat pour recouvrer des mauvaises spéculations. Nous avons décrit un mécanisme matériel très simple pour ce faire, de part le contexte de notre étude. Nous avons également proposé un support matériel complet permettant de redimensionner le pipeline à la largeur du calcul. Notre microarchitecture se démarque par la proposition d'un nouveau modèle pour fichiers de registres adapté aux pipelines reconfigurables dynamiquement. Ce modèle permet de conserver les données tout en exposant des consommations dynamique et statique réduites. Notre évaluation expérimentale indique que notre approche peut réduire de 17% la consommation dynamique du chemin de données et de 22% le consommation du fichier de registres avec un impact négligeable sur les performances.

La seconde proposition de cette thèse s'emploie à la réduction de la complexité dans le cadre des processeurs superscalaires. Nous avons observé qu'une fraction importante des instructions manipulent exclusivement des opérandes tronqués à l'exécution. Notre analyse a également révélé que ces opérations tronquées sont entrelacées avec les autres instructions du programme au cours de l'exécution. Nous avons proposé de découpler le traitement des opérations tronquées pour exploiter ces propriétés des applications. Plus particulièrement, nous avons présenté le modèle clusterisé WPM qui permet de réduire significativement la complexité du processeur. Une des particularités du modèle WPM est de ne solliciter aucun effort particulier pour distribuer les instructions vers les clusters. Les propriétés observées indiquent effectivement que la mise en œuvre d'un prédicteur de largeur suffit à préserver l'équilibrage des charges tout en limitant le nombre de communications inter-cluster. Nous avons évalué une implémentation relativement basique de ce modèle mais qui exprime déjà le fort potentiel sous-jacent à cette approche. Une configuration à 2 clusters montrent en effet qu'il est possible de réaliser des gains substantiels en énergie, c-à-d. de l'ordre de 50% pour le fichier de registres et les connexions inter-cluster, avec un impact acceptable sur les performances.

Perspectives

Les perspectives de nos études sont multiples et sont mieux détaillées dans leurs chapitres respectifs. Une première voie d'exploration possible pour notre première proposion concerne le redimensionnement du cache de données à la largeur des régions tronquées du programme. L'intérêt premier de ce schéma est de réduire davantage la consommation du processeur. Ensuite, nous proposons d'optimiser l'algorithme de formation des régions tronquées pour réduire l'appel au mécanisme de recouvrement. Enfin, il est possible d'appliquer notre schéma au niveau binaire ou d'employer la compilation dynamique pour améliorer les performances et la consommation tout en s'abstrayant du code source.

D'autre part, les perspectives de notre seconde étude concernent principalement l'application du modèle aux processeurs VLIW et aux architectures multicœur. Dans le contexte VLIW, le schéma présenté pour les processeurs superscalaires peut être appliqué à condition de déléguer l'allocation des instructions au compilateur. Pour ce faire, il est possible de nous appuyer sur le schéma spéculatif décrit dans la première étude. Pour les architectures multicœur, le travail consiste à explorer les configurations hétérogènes présentant le meilleur compromis performance/complexité et de mettre au point un mécanisme efficace pour communiquer les données tronquées.

Bibliographie

[1] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC : the end of the road for conventional microarchitectures. In *Proceedings of the 27th annual International Symposium on Computer architecture*, 2000.

[2] Dimitri A. Antoniadis. SOI CMOS as a mainstream low power technology : a critical assessment. In *Proceedings of the 1997 International Symposium on Low power electronics and design*, 1997.

[3] Ayala, J.L., López, V.M., Veidenbaum, A., and López C.A. Energy aware register file implementation through instruction predecode. In *Proceedings of the International Conf. on Application-Specific Systems, Architectures and Processors*, June 2003.

[4] Yu Bai and R. Iris Bahar. A low-power in-order/out-of-order issue queue. *ACM Transaction on Architecture and Code Optimization*, 2004.

[5] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo : a transparent dynamic optimization system. *SIGPLAN Not.*, 35(5), 2000.

[6] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo : a transparent dynamic optimization system. In *PLDI '00 : Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, 2000.

[7] Balasubramonian, R., Dwarkadas, S., Albonesi, D. Reducing the Complexity of the Register File in Dynamic Superscalar Processor. In *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001.

[8] Balasubramonian, R., Dwarkadas, S., Albonesi, D. Dynamically Managing the Communication-parallelism Trade-off in Future Clustered Processors. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[9] Balasubramonian, R., Muralimanohar, N., Ramani, K., and Venkatachalapathy, V. Microarchitectural Wire Management for Performance and Power in Partitioned Architectures . In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, February 2005.

[10] Baniasadi, A., and Moshovos, A. Instruction Distribution Heuristics for Quad-Cluster,Dynamically-Scheduled, Superscalar Processors. In *Proceedings of the 33th International Symposium on Microarchitecture*, Dec. 2000.

[11] Bodin, F., Rohou, E., and Seznec, A. Salto : System for assembly-language transformation and optimization. In *Proceedings of the Sixth Workshop on Compilers for Parallel Computers*, 1996 Dec.

[12] David Brooks and Margaret Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proceedings of the The Fifth International Symposium on High Performance Computer Architecture*, 1999.

[13] David Brooks and Margaret Martonosi. Value-based clock gating and operation packing : dynamic strategies for improving processor power and performance. *ACM Trans. Comput. Syst.*, 2000.

[14] Mary D. Brown, Jared Stark, and Yale N. Patt. Select-free instruction scheduling logic. In *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture*, pages 204–213. IEEE Computer Society, 2001.

[15] Budiu, M., Goldstein, S., Sakr, M., and Walker, K. BitValue inference : Detecting and exploiting narrow bitwidth computations. In *Proceedings of the 6th European Conference on Parallel Computing (EuroPar)*, Aug. 2000.

[16] J. Adam Butts and Gurindar S. Sohi. A static power model for architects. In *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*, pages 191–201. ACM Press, 2000.

[17] Canal, R., Gonzalez, A., and Smith, J.E. Very low power pipelines using significance compression. In *Proceedings of the 33th International Symposium on Microarchitecture*, Dec. 2000.

[18] Canal, R., Gonzalez, A., and Smith, J.E. Software-controlled operand-gating. In *Proceedings of the International Symposium on Code Generation and Optimization*, March 2004.

[19] Canal, R., Parcerisa, J-M., Gonzalez, A. Dynamic Cluster Assignment Mechanisms. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, Jan. 2000.

[20] A. Chandrakasan and R. Brodersen. Minimizing power consumption in digital cmos circuits. In *Proceedings of the IEEE*, pages 498–523, April 1995.

[21] Anantha P. Chandrakasan and Robert W. Brodersen. *Low Power Digital CMOS Design.* Kluwer Academic Publishers, 1995.

[22] Choi, J., Jeon, J., and Choi, K. Power minimization of functional units by partially guarded computation. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 2000.

[23] José-Lorenzo Cruz, Antonio González, Mateo Valero, and Nigel P. Topham. Multiple-banked register file architectures. In *Proceedings of the 27th annual International Symposium on Computer architecture*, pages 316–325. ACM Press, 2000.

[24] Vivek De and Shekhar Borkar. Technology and design challenges for low power and high performance. In *Proceedings of the 1999 International Symposium on Low power electronics and design*, pages 163–168. ACM Press, 1999.

[25] Romain Dolbeau and André Seznec. CASH : Revisiting hardware sharing in single-chip parallel processor. *Journal of Instruction Level Parallelism*, April 2004.

[26] Nathalie Drach and Julien Sebot. Simd isa extensions : Tradeoff between power consumption and performance on a superscalar processor. In *Kool Chips Workshop, Micro 33*, December 2000.

[27] Oguz Ergin, Deniz Balkan, Kanad Ghose, and Dmitry Ponomarev. Register packing : Exploiting narrow-width operands for reducing register file pressure. In *MICRO-37 : Proceedings of the 37th International Symposium on Microarchitecture*, pages 304–315. IEEE Computer Society, 2004.

[28] Faraboschi, P., Brown, G., Fisher, J.A., Desoli, G., and Homewood, F. Lx : A Technology Platform for Customizable VLIW Embedded Processing. In *Proceedings of the 27th International. Symposium on Computer Architecture*, June 2000.

[29] Keith I. Farkas, Paul Chow, and Norman P. Jouppi. Register file design considerations in dynamically scheduled processors. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA '96)*, page 40. IEEE Computer Society, 1996.

[30] Farkas, K. I., Chow, P., Jouppi, N. P., and Vranesic, Z. The Multicluster Architecture : Reducing Cycle Time Through Partitioning. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[31] Brian Fields, Shai Rubin, and Rastislav Bodik. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th annual International Symposium on Computer architecture*, pages 74–85. ACM Press, 2001.

[32] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches : simple techniques for reducing leakage power. In *Proceedings of the 29th annual International Symposium on Computer architecture*, pages 148–157. IEEE Computer Society, 2002.

[33] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9) :1277–1284, September 1996.

[34] Ruben Gonzalez, Adrian Cristal, Daniel Ortega, Alexander Veidenbaum, and Mateo Valero. A content aware integer register file organization. *SIGARCH Comput. Archit. News*, 2004.

[35] Gonzalez, R., Cristal, A., Veidenbaum, A., Pericas, M. and Valero, M. An Asymmetric Clustered Processor based on Value Content. In *Proceedings of the 19th ACM International Conference on Supercomputing*, June 2005.

[36] Michael K. Gowan, Larry L. Biro, and Daniel B. Jackson. Power considerations in the design of the alpha 21264 microprocessor. In *Proceedings of the 35th annual conference on Design automation*, pages 726–731, 1998.

[37] Dirk Grunwald, Artur Klauser, Srilatha Manne, and Andrew Pleszkun. Confidence estimation for speculation control. In *Proceedings of the 25th annual International Symposium on Computer architecture*, pages 122–131. IEEE Computer Society, 1998.

[38] John L. Henning. Spec cpu2000 : Measuring cpu performance in the new millennium. *Computer*, 33(7) :28–35, 2000.

[39] Ho, R., Mai, K. W., and Horowitz, M. A. The Future of Wires. *Proceedings of the IEEE*, 89(4) :490–504, Apr. 2001.

[40] M. Horowitz, R. Ho, and K. Mai. The future of wires, 1999.

[41] Michael Huang, Jose Renau, and Josep Torrellas. Energy-efficient hybrid wakeup logic. In *ISLPED '02 : Proceedings of the 2002 international symposium on Low power electronics and design*, pages 196–201, New York, NY, USA, 2002. ACM Press.

[42] Prabhat Jain, G. Edward Suh, and Srinivas Devadas. Embedded intelligent sram. In *Proceedings of the 40th conference on Design automation*, pages 869–874. ACM Press, 2003.

[43] N. Jouppi and S. Wilton. Cacti : an enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5) :677–688, May 1996.

[44] R.E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2) :24–36, March 1999.

[45] Ilhyun Kim and Mikko H. Lipasti. Macro-op scheduling : Relaxing scheduling loop constraints. In *36th Annual Internation Symposium on Microarchitecture (MICRO-36'03)*, 2003.

[46] Nam Sung Kim, Todd Austin, David Blaauw, Trevor Mudge, Krisztián Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnanshnan Narayanan. Leakage current : Moore's law meets static power. *Computer*, 36(12) :68–75, 2003.

[47] Kim, I., and Lipasti, M. H. Understanding Scheduling Replay Schemes. In *HPCA '04 : Proceedings of the 10th International Symposium on High Performance Computer Architecture*, 2004.

[48] Kondo, M., and Nakamura, H. A Small, Fast and Low-Power Register File by Bit-Partitioning. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, February 2005.

[49] Rakesh Kumar, Norman P. Jouppi, and Dean M. Tullsen. Conjoined-core chip multiprocessing. In *Proceedings of the 37th annual International Symposium on Microarchitecture*, 2004.

[50] Larsen, S., and Amarasinghe, S. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2000.

[51] Larson, E., Chatterjee, S., and Austin, T. Mase : A novel infrastructure for detailed microarchitectural modeling. In *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, November 2001.

[52] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench : A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[53] Hai Li, Swarup Bhunia, Yiran Chen, T. N. Vijaykumar, and Kaushik Roy. Deterministic clock gating for microprocessor power reduction. In *Proceedings of the The Ninth International Symposium on High-Performance Computer Architecture (HPCA'03)*, page 113. IEEE Computer Society, 2003.

[54] Lipasti, M. H., Mestan, B. R., and Gunadi, E. Physical Register Inlining. In *Proceedings of the 31th International Symposium on Computer Architecture*, June 2004.

[55] Gabriel H. Loh. Exploiting data-width locality to increase superscalar execution bandwidth. In *Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture*, pages 395–405. IEEE Computer Society Press, 2002.

[56] Nir Magen, Avinoam Kolodny, Uri Weiser, and Nachum Shamir. Interconnect-power dissipation in a microprocessor. In *SLIP '04 : Proceedings of the 2004 international workshop on System level interconnect prediction*, pages 7–13, New York, NY, USA, 2004. ACM Press.

[57] Mahlke, S., Ravindran, R., Schlansker, M., Schreiber, R., and Sherwood, T. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11), Nov. 2001.

[58] Manne, S., Klauser, A., and Grunwald, D. Pipeline gating : Speculation control for energy reduction. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[59] Pierre Michaud and André Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *HPCA '01 : Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 27, Washington, DC, USA, 2001. IEEE Computer Society.

[60] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[61] A. Moshovos and G. S. Sohi. Microarchitectural innovations : Boosting microprocessor performance beyond semiconductor technology scaling. In *IEEE Proceedings*, volume 89, pages 1560–1575, January 2002.

[62] Trevor Mudge. Power : A first-class architectural design constraint. *Computer*, 34(4) :52–58, 2001.

[63] Robert Muth, Saumya Debray, Scott Watterson, and Koen de Bosschere. alto : A link-time optimizer for the DEC alpha. Technical Report TR98-14, Wednesday, 9 1998.

[64] Nakra, T., Childers, B.R., and Soffa, M.L. Width-Sensitive Scheduling for Resource-Constrained VLIW Processors. In *Proceedings of the 3th ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2000.

[65] Palacharla, S., Jouppi, N. P., and Smith, J. E. Complexity-effective Superscalar Processors. *ACM SIGARCH Computer Architecture News*, 25(2) :206 – 218, May 1997.

[66] Parcerisa, J-M., Sahuquillo, J., Gonzalez, A., and Duato, J. Efficient Interconnects for Clustered Microarchitectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2002.

[67] Parikh, D., Skadron, K., Zhang Y., Barcella, M., and Stan, M.R. Power issues related to branch prediction. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, 2002.

[68] Il Park, Michael D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture*, 2002.

[69] David A. Patterson and Carlo H. Sequin. RISC I : A Reduced Instruction Set VLSI Computer. In *Proceedings of the 8th annual Symposium on Computer Architecture*, 1981.

[70] D.S. Wills P.G. Sassone. Multicycle broadcast bypass : Too readily overlooked. In *Proceedings of the Workshop on Complexity Effective Design (WCED)*, June 2004.

[71] Gilles Pokam, Olivier Rochecouste, André Seznec, and François Bodin. Speculative Software Management of Datapath-width for Energy Optimization. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2004.

[72] Pokam, G., Bihan, S., Simonnet, J., and Bodin, F. SWARP : A Retargetable Preprocessor for Multimedia Instructions. *Concurrency and Computation : Practice and Experience*, 16(2-3) :303–318, February/March 2004.

[73] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-vdd : a circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the 2000 International Symposium on Low power electronics and design*, 2000.

[74] Michael D. Powell and T. N. Vijaykumar. Pipeline damping : a microarchitectural technique to reduce inductive noise in supply voltage. In *Proceedings of the 30th annual International Symposium on Computer architecture*, 2003.

[75] Scott Rixner, William J. Dally, Brucek Khailany, Peter R. Mattson, Ujval J. Kapasi, and John D. Owens. Register organization for media processing. In *HPCA*, 2000.

[76] Olivier Rochecouste, Gilles Pokam, and André Seznec. A Case for a Complexity-Effective, Width-Partitioned Microarchitecture. RR 5677, IRISA/INRIA, September 2005.

[77] Olivier Rochecouste, Gilles Pokam, and André Seznec. A Case for a Complexity-Effective, Width-Partitioned Microarchitecture. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2006. To appear.

[78] Sato, T., and Arita, I. Table Size Reduction for Data Value Predictors by Exploiting Narrow Width Values. In *Proceedings of the 14th international conference on Supercomputing*, May 2000.

[79] Manfred Schlett. Trends in embedded-microprocessor design. *Computer*, 31(8) :44–49, 1998.

[80] Jeff Scott, Lea Hwang Lee, John Arends, and Bill Moyer. Designing the low-power m-core architecture. In *Proceedings of Power Driven Microarchitecture*, pages 145–150, July 1998.

[81] The international technology roadmap for semiconductors, `http://www.semichips.org`, 2001.

[82] André Seznec, Eric Toullec, and Olivier Rochecouste. Register write specialization register read specialization : a path to complexity-effective wide-issue superscalar processors. In *Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture*, pages 383–394. IEEE Computer Society Press, 2002.

[83] Shivakumar, P., and Jouppi, N. Cacti 3.0 : An integrated cache timing power, and area model. Technical report, DEC Western research Lab, 2002.

[84] Smith, J. E. A Study of Branch Prediction Strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 135–148, 1981.

[85] Stephenson, M., Babb, J., and Amarasinghe, S. Bitwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 2000.

[86] Andrei Terechko, Erwan Le Thenaff, Manish Garg, Jos van Eijndhoven, and Henk Corporaal. Inter-cluster communication models for clustered vliw processors. In *HPCA '03 : Proceedings of the The Ninth International Symposium on High-Performance Computer Architecture (HPCA'03)*, 2003.

[87] Vivek Tiwari, Deo Singh, Suresh Rajgopal, Gaurav Mehta, Rakesh Patel, and Franklin Baez. Reducing power in high-performance microprocessors. In *Proceedings of the 35th annual conference on Design automation*, pages 732–737, 1998.

[88] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading : Maximizing On-chip Parallelism. *SIGARCH Comput. Archit. News*, 1995.

[89] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower. In *Proceedings of the 27th annual International Symposium on Computer architecture*, pages 95–106. ACM Press, 2000.

[90] Zhang, Y., Parikh, D., Sankaranarayanan, K., Skadron, K., and Stan, M. Hotleakage : A temperature-aware model of subthreshold and gate leakage for architects. Technical Report CS-2003-05, University of Virginia, Department of Computer Science, Mar. 2003.

[91] Zyuban, V., and Kogge, P. The Energy Complexity of Register Files. In *Proceedings of the International Symposium on Low Power Electronics and Designs*, Aug. 1998.

Résumé

Cette thèse porte sur la réduction de la consommation d'énergie et de la complexité matérielle dans les processeurs modernes. L'approche proposée repose sur la propriété des programmes à s'exécuter avec une fraction importante en *données tronquées*, celles ne comportant qu'un nombre restreint de bits significatifs. Les microarchitectures modernes continuent, cependant, à évoluer vers des chemins de données très larges, sans en tenir compte. Ce surdimensionnement influe directement sur la consommation et la complexité des composants du processeur. Dans un premier temps, nous proposons d'exploiter cette propriété pour gérer la consommation dans les systèmes embarqués. Notre approche consiste à redimensionner dynamiquement les opérateurs et le fichier de registres en se basant sur des spéculations logicielles. Un mécanisme matériel assure le recouvrement de l'exécution en cas de mauvaise spéculation. Dans un second temps, nous montrons que les *opérations tronquées*, celles manipulant exclusivement des données tronquées, sont fréquentes et bien distribuées au cours de l'exécution. Nous proposons le modèle clusterisé WPM pour découpler le traitement de ces opérations sur des opérateurs dédiés ; réduisant ainsi, à la fois, la consommation d'énergie et la complexité matérielle des processeurs superscalaires.

Mots clés *consommation électrique, complexité matérielle, système embarqué, processeur superscalaire, architecture clusterisée , fichier de registres, donnée tronquée.*

Abstract

This thesis is concerned with reducing power consumption and hardware complexity of modern processors. Central to our proposals is the observation that most program executions are comprised of a large amount of narrow-width data. Modern microarchitectures continue, however, to feature large datapath-width, regardless of that property ; thus making them power and complexity inefficient. First, we propose to exploit this program property for managing power consumption in embedded systems. Our approach consists in dynamically adapting the datapath components width (*functional units and register file*) based on software speculations. Program correctness is guaranteed at runtime by hardware means, recovering the execution in case of a data-width mispeculation. Second, we show that *narrow-width operations*, i.e. operations exclusively handling narrow-width data, are prevalent and well distributed across a program run. We propose to resort to a *Width-Partitioned Microarchitecture* (WPM) to decouple the treatment of these narrow-width operations onto dedicated operators for mastering both power consumption and hardware complexity of superscalar processors.

Keywords *power consumption, hardware complexity, embedded system, superscalar processor, clustered architecture, register file, narrow-width data.*