N^o d'ordre: 2995

THÈSE

présentée

devant l'université de Rennes 1

pour obtenir

le grade de : DOCTEUR DE L'UNIVERSITÉ DE RENNES 1
Mention INFORMATIQUE

par

Gilles POKAM

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

Titre de la thèse :

Techniques de compilation pour la gestion et l'optimisation de la consommation d'énergie des architectures VLIW

soutenue le 15 Juillet 2004 devant la commission d'examen

M. :	Christophe	WOLINSKI	Président
MM. :	Nigel	TOPHAM	Rapporteurs
	Pascal	SAINRAT	
MM. :	Erven	ROHOU	Examinateurs
	André	SEZNEC	
	François	BODIN	

Ainsi, le progrès scientifique nous montre tous les jours que ce qui semblait être un trait spécifique de l'esprit humain n'était qu'une habitude mentale dont on se défait difficilement.

Cheikh Anta Diop, *Civilisation ou Barbarie*

To my son, Nufi
To my wife, Carine

Acknowledgments

I would like to thank all the members of my thesis committee, Prof. Christophe Wolinski, Prof. Nigel Topham, Prof. Pascal Sainrat, Dr. Erven Rohou, Dr. André Seznec and Prof. François Bodin for their patience in reading my dissertation and for their fruitful comments and suggestions that make this thesis more readable and accurate.

I would like to thank more particularly Prof. François Bodin who directed this thesis. He offered me the opportunity to join IRISA in order to conduct this research in the compiler and architecture group, CAPS. I am very thankful to him for his fruitful advice and continuing support. He also deserves special thanks for the opportunity he gave me to develop some of the ideas contained in this thesis.

I am also very indebted to Dr. André Seznec, the head of the CAPS group. He devoted much of his time providing me with advice regarding some of the ideas developed in this thesis. His advice have been invaluable. I sincerely express to him all my gratitude.

Many other people contributed to make this thesis possible. Olivier Rochecouste deserves special thanks for the close collaboration we had on the *speculative bit-width management scheme* paper. I also want to thank Karine Heydemann; without her help it would have not been possible to organize the "*pot de thèse*" in time. Blanche Manikeu was really amazing. She has prepared all the delicious meals of the "*pot de thèse*". Fatou Traoré has prepared the famous "*bissap*", a kind of african juice based on "*fleurs d'oseil*". Clotilde Hamza and Awa Traoré helped me look after my son Nufi when I was preparing my presentation. Thank you to all of them.

My best thanks go to my wife Carine and to my son Nufi for having provided me with the support necessary for overcoming the frustrating phases of this work. Without their love and permanent support, this thesis would not have been possible.

Finally, I would like to thank my beloved parents, Marie-Madeleine and Daniel Pokam, for their unconditional support and unshakable belief in me. They have been giving me incredible support during this past three years. To entirely disclose my gratitude to them will remain an unachievable task.

Contents

List of Figures

Résumé étendu de la thèse

L'intérêt récent des systèmes informatiques pour la réduction de la consommation d'énergie relève principalement de deux constats. Tout d'abord, l'essor rapide du marché des systèmes embarqués repose sur l'emploi de batteries d'alimentation; il devient donc indispensable de trouver de nouvelles solutions visant à minimiser le sur-coût induit par l'utilisation de ces ressources (durée de vie, fiabilité, capacité des batteries, etc). De façon plus générale, on ne peut plus concevoir de nos jours des processeurs rapides sans tenir compte des problèmes de dissipation d'énergie liés à un haut niveau d'intégration et à l'utilisation de hautes fréquences.

Afin de produire des systèmes à basse consommation d'énergie, de nombreuses solutions matérielles ont été proposées. Cependant, la consommation en énergie d'un processeur ne dépend pas uniquement de son architecture, mais aussi du code exécuté. En particulier, la consommation d'énergie pour une tâche donnée dépend fortement de l'efficacité du code produit par le compilateur. Dans le cas des architectures VLIW, ceci est d'autant plus critique que la gestion du parallélisme d'instructions est confiée au compilateur.

Objectifs de la thèse

Cette thèse s'emploie principalement à réduire la consommation d'énergie des architectures VLIW tout en essayant de préserver les performances. Contrairement à certaines approches logicielles tendant à favoriser l'optimisation de code pour obtenir des gains en énergie, nous présentons des arguments en faveur d'une approche synergique intégrant matériel et logiciel à la fois. L'idée principale défendue tout au long de cette thèse repose sur le fait que seule une compréhension avançée du comportement dynamique d'un programme au niveau du compilateur est susceptible de produire un meilleur contrôle de la gestion de la consommation d'énergie. Pour cela, nous introduisons une technique d'analyse statique du comportement dynamique d'un programme afin de parvenir à identifier et caractériser les chemins les plus fréquemment exécutés ("chemins chauds"). L'objectif visé étant la réduction de la consommation d'énergie, nous montrons par la suite que sur directive du compilateur, l'architecture de la machine peut être modifiée pour s'adapter à un état dynamique du programme. Nous présentons les conditions d'une telle reconfiguration ainsi que les éventuelles modifications à apporter à l'architecture, à la fois pour le système des caches que pour le chemin de données d'un processeur VLIW.

Contributions de la thèse

Analyse du compromis énergie/performance

Dans de nombreux cas, optimiser le temps d'exécution réduit aussi la consommation d'énergie et de manière générale, la problématique de la réduction de la consommation d'énergie est similaire à celle de la réduction du temps d'exécution [107]. En effet, il s'agit de mettre en oeuvre des transformations de programme qui minimisent les opérations entraînant une forte consommation d'énergie telles que les accès mémoires ou une consommation inutile telle que celle provoquée par des cycle d'attente ("stall"). Ces transformations sont généralement évaluées sur la base d'une fonction objective que l'on peut définir à partir d'un modèle de consommation d'énergie.

Dans le Chapitre 2, nous avons considéré une fonction objective, PTE, qui est l'équivalent de l'inverse du produit énergie-délai [40]. Ce produit est une mesure efficace de la consommation électrique et permet de tenir compte de la performance et de l'énergie à la fois.

$$PTE = \frac{1}{E_{op} \times Cycle_{op}} = \frac{Performance}{Energy} \tag{1}$$

Dans l'équation (1), le terme E_{op} désigne l'énergie par opération alors que $Cycle_{op}$ est le temps de cycle associé à l'exécution de cette opération. La fonction objective PTE sert à comparer deux instances différentes d'une même application au niveau logiciel. De ce fait, PTE permet de mesurer les besoins liés à une augmentation des performances par rapport à ceux liés à l'accroissement de la consommation d'énergie. Cette fonction objective rend donc aisé la comparaison du ratio performance/énergie de deux instances différentes d'un même programme.

Le modèle de dissipation d'énergie employée dans cette étude est celui d'une architecture VLIW capable d'exécuter N opérations en parallèle. Il s'exprime de la manière suivante [14]:

$$EPB_{w_n} = E_c + IPC_{w_n} \cdot E_{op} + m \cdot p \cdot E_s + l \cdot q \cdot E_{miss} \tag{2}$$

Dans l'équation (2), EPB_{w_n} désigne l'énergie associée à l'exécution d'une instruction longue w_n; E_c est le coût de base associé à l'exécution d'une instruction, IPC_{w_n} le nombre d'instructions différent d'un *nop* au sein d'une instruction longue. La troisième et la quatrième sous-expressions dans (2) représentent l'énergie correspondant à la pénalité due à un défaut dans le cache de données et d'instructions, respectivement. Dans ces deux dernières sous-expressions, m et l désignent le nombre de ***stall*** et de ***nop*** suite aux défauts dans le cache de données et d'instructions, respectivement; la variable p et la variable q la probabilité associée à chacun de ses évènements, et E_s et E_{miss} l'énergie dissipée par le processeur lors d'une pénalité due à un défaut dans le cache de données et d'instructions, respectivement.

Ce modèle met en évidence deux points fondamentaux. Tout d'abord, l'estimation de la consommation dépend de données de ***profile*** sur l'exécution du programme. Ces données ne sont pas nécessairement stables d'un jeu de données à l'autre. Enfin, il ne suffit pas d'améliorer localement la performance d'un fragment de code (i.e. une boucle) pour obtenir une diminution de la consommation totale d'un programme. En effet, la plupart des optimisations de code liées à la transformation de programme pour machines VLIW introduisent une expansion de la taille du code. Outre la mémoire d'instructions consommée, cette expansion de la taille du code induit également un accroissement des défauts de cache d'instructions qui a un impact direct sur les performances et sur la consommation d'énergie.

Afin de mieux saisir l'impact qu'une telle optimisation de code peut avoir sur la consommation électrique d'un processeur, nous avons entrepris l'étude d'une transformation de code, les ***hyperblocs*** [68], qui vise à accroître les performances d'un programme en introduisant une certaine expansion de la taille du code. La fonction objective PTE a été utilisée pour évaluer l'impact des ***hyperblocs*** sur la consommation d'énergie.

Les transformations hyperblocs s'opérant au niveau du graphe de contrôle de flot, nous avons considéré pour la suite de cette étude le bloc de base comme étant la plus petite granularité d'ordonnancement possible. Dès lors, la fonction objective PTE peut être récrite de la manière suivante:

$$PTE = \frac{1}{E_{BB} \times Cycle_{BB}} = \frac{\overline{IPC}}{N \times E_{BB}} \tag{3}$$

Dans la nouvelle formulation de PTE ci-dessus, $\overline{IPC}$ représente le nombre moyen d'opérations exécutées par instruction longue et N le nombre total d'opérations dans le bloc de base considéré. L'énergie consommée par le bloc de base est définie par le terme E_{BB}.

Une transformation hyperbloc transforme une région R du graphe de contrôle de flot en un large bloc de base H. Pour qu'une telle transformation soit valide, nous posons la condition suivante:

$$PTE_H > PTE_R \tag{4}$$

De l'inégalité ci-dessus on peut exprimer la fonction de transformation F_{ilp} d'une région R du graphe de contrôle de flot en un ***hyperbloc*** H de la manière suivante:

$$F_{ilp}(\overline{IPC_R}) = \frac{A \cdot \overline{IPC_R}}{B + C \cdot \overline{IPC_R}} \tag{5}$$

où

$$\begin{cases} A = f_H \cdot N_H \cdot n_H \cdot E_c + N_H \cdot \overline{s_H} \cdot E_s \\ B = m \cdot N_R \cdot \overline{f_R} \cdot \overline{n_R} \cdot E_c + N_R \cdot \overline{s_R} \cdot E_s \\ C = (m \cdot N_R \cdot \overline{f_R} \cdot \overline{n_R} - f_H \cdot N_H \cdot n_H) \cdot E_{op} \end{cases} \tag{6}$$

Les variables f, N, n, et s désignent la fréquence d'exécution, le nombre d'opérations, le nombre d'instructions longues et le nombre de stall pour une région R ou un hyperbloc H. La variable m représente le nombre de blocs de base au sein d'une région R avant transformation. Les mêmes variables avec une barre au dessus désignent une moyenne.

L'analyse du compromis entre l'application de la transformation en hyperbloc en vue d'accroître les performances et la réduction de la consommation d'énergie varie en fonction du signe de la variable C. Les cas $C >= 0$ sont ceux qui sont susceptibles de produire de bon résultats, la quantité $m{\cdot}N_R{\cdot}\overline{f_R}{\cdot}\overline{n_R}$ étant en effet supérieure à $f_H{\cdot}N_H{\cdot}n_H$ qui décrit les caractéristiques de l'hyperbloc résultant de la transformation de la région R. La transformation devient cependant problématique dès que $m \cdot N_R \cdot \overline{f_R} \cdot \overline{n_R} > f_H \cdot N_H \cdot n_H$, c'est à dire dès que $C < 0$. Le signe de C dans ce cas de figure peut s'expliquer par plusieurs raisons, l'une d'entre elles étant une augmentation de la valeur de n à cause de certaines contentions pour des ressources partagées telle que l'accès à des ports mémoire, ou alors une augmentation de la valeur de N à cause de l'ajout de code de compensation pour réparer les effets de bord dus à l'élimination des instructions de branchement.

L'évaluation de ce modèle a montré que la prise en compte de l'heuristique de transformation d'une région du graphe de contrôle de flot en un hyperbloc peut rapporter 17% de gain du produit énergie-délai. Ce résultat, toutefois, n'a été évalué que sur un petit sous ensemble d'applications Powerstone [95] qui présentaient déjà un nombre important de régions susceptibles d'être transformées en hyperbloc. Ce constat montre bien les limites qu'une telle approche peut avoir, tant les opportunités de transformations en hyperbloc restent mesurées. Ceci étant dû, entre autres, au très faible parallélisme d'instructions inhérent à plusieurs applications [79]. D'où notre ambition d'accroître les opportunités d'optimisation en exposant certains aspects du comportement dynamique d'un programme au compilateur.

Vers une analyse statique du comportement dynamique de programme

Afin d'identifier et de caractériser les chemins les plus exécutés d'un programme (dans la suite appelés "chemins chauds"), il est nécessaire de s'attaquer à l'étude statique du comportement dynamique d'un programme. Les techniques employées à cet effet procèdent souvent par profilage. Cependant, si le profilage ne se limite qu'à la collecte des chemins du programme confinés à une procédure [10], à une boucle ou à une fonction, il est impossible d'obtenir une vision globale de l'ensemble de l'application. Il s'en suit donc que les optimisations qui en découlent sont de portée très souvent limitée. Aussi est-il indispensable de collecter les informations relatives aux chemins de contrôle avec une granularité plus grosse afin d'obtenir de meilleur opportunités d'optimisation.

L'approche que nous proposons dans le Chapitre 3 ambitionne de collecter les chemins de contrôle d'un programme pour une exécution complète de l'application. Elle est semblable à celle proposée par Larus [60] en cela que nous nous attaquons à la même granularité du programme, c'est à dire à toute l'application. Cependant, l'approche proposée par Larus se fonde sur un algorithme de compression nommé SEQUITUR [78] pour réduire la taille des informations collectées. SEQUITUR étant un algorithme de compression "*online*", il est toujours important mais difficile de réduire la taille de la trace de sorte à ne pas compromettre le temps d'exécution de l'application. En plus, la représentation de la trace utilisée, qui est un graphe orienté acyclique (DAG), ne permet pas de discriminer entre plusieurs instances dynamiques d'un même chemin statique; chaque représentation dynamique d'un même chemin statique étant confondue par un même et seul noeud dans le DAG.

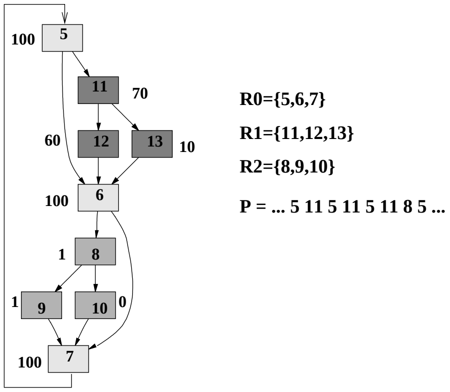


Figure 1: Exemple de CFG avec régions.

Pour les raisons citées ci-dessus, plus particulièrement pour celles liées au temps d'exécution, nous avons envisagé une approche "*offline*". Une première phase dans notre approche consiste à collecter les chemins exécutés. Il s'en suit donc un problème de taille de la trace. Nous nous sommes donc intéressés à la collecte d'un sous-ensemble seulement des blocs exécutés (voir Figure 1). Ces blocs ont étés choisis de façon très judicieuse de sorte que chaque bloc représente un sous-graphe d'exécution du graphe de contrôle de flot. Un des avantages de cette approche est que l'identification des régions les plus fréquemment exécutées se fait au détriment des blocs de base les plus exécutés, ce qui conduit à réduire considérablement la taille de la trace. Les blocs représentant des boucles ont été compressés de manière à ne retenir qu'une seule instance dynamique des différentes exécutions. La combinaison de ces techniques nous a permis d'atteindre un taux de compression de la trace de l'ordre de 74% sur des applications de MiBench

[42].

Toute suite de blocs de base qui se répète dans la trace est conservée dans un sous-ensemble de référence de blocs de base, appelé BBWS, représentant les blocs de base statiques. Afin de pouvoir distinguer deux chemins dynamiques dont les blocs de base sont issus du même BBWS, on associe à chaque BBWS une annotation qui le décrit de manière unique. Les informations que l'on peut trouver dans une annotation sont très variées. Pour nos expériences, nous avons considéré un identificateur unique par région, le nombre de cycles requis pour exécuter une région, le nombre de défauts dans le cache de données et d'instructions.

L'originalité de notre approche réside dans la technique employée pour détecter les BBWS. Nous nous basons sur des tableaux de suffixe [71], généralement utilisés en bioinformatique, pour trier les BBWS dans la trace. Nous montrons que l'emploi de table de suffixes offre plusieurs avantages en comparaison à une représentation DAG:

1. permet des opérations de tri en $O(\mathrm{p} + \ln(N))$, où p est la longueur du pattern recherché et N la taille de la trace;

2. permet de chercher la plus longue séquence répétée de BBWS dans la trace, $lmax$;

3. permet de chercher toutes les séquences répétées de taille n dans la trace, $n \leq lmax$;

4. permet de déterminer la fréquence de distribution de chaque BBWS de taille n dans la trace;

5. permet d'identifier exactement les différentes positions de chaque BBWS de taille n dans la trace.

L'algorithme que nous avons implémenté est une adaptation intelligente de l'algorithme de Karp, Miller et Rosenberg [54] utilisé pour la recherche de séquences répétées dans un *string*. Nous montrons que ce problème est analogue à celui de la recherche des chemins chauds et que de simples modifications apportées à l'algorithme de base permettent d'identifier rapidement tous les BBWS de la trace.

Tous les BBWS d'une trace ne sont cependant pas automatiquement des chemins chauds. Il est donc nécessaire d'identifier parmi les BBWS ceux qui ont la propension à le devenir. Nous introduisons trois critères qui permettent de discriminer parmi les BBWS ceux qui représentent des chemins chauds, c'est-à-dire des chemins fréquemment exécutés (Voir Figure 2).

Le premier critère concerne la *couverture locale* d'un BBWS. Ce critère permet de prendre une mesure du temps d'exécution de la région ou aussi du nombre d'instructions dynamiques exécutées dans la région, avant qu'une transition à une nouvelle région se produise.

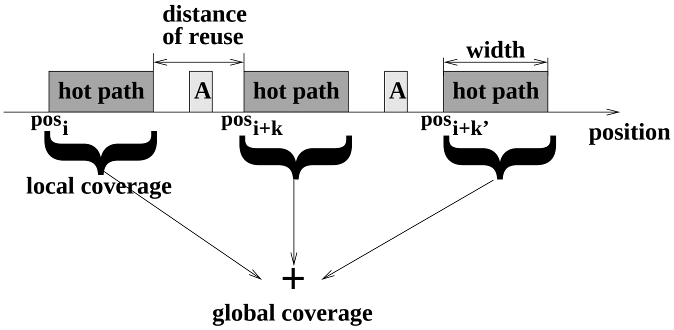


Figure 2: Caractéristiques des chemins chauds.

Le deuxième critère est la *couverture globale* d'un BBWS. Ce critère est relatif au poids de la région sur l'ensemble de l'exécution de l'application. C'est en fait le produit de la *couverture locale* par la fréquence d'exécution dynamique de la région.

Enfin, le dernier critère est la *distance de réutilisation* d'un BBWS. Ce dernier permet d'obtenir une approximation de la température d'un BBWS. En effet, plus la *distance de réutilisation* est grande, plus petite est la probabilité pour qu'un BBWS soit un chemin chaud. En supposant que *Position* désigne l'ensemble des positions dans la trace d'un BBWS mais qui ne se chevauchent pas, la *distance de réutilisation* moyenne d'un BBWS peut se calculer comme suit:

$$\overline{D} = \frac{\sum (pos_{i-1} + width) \ \% \ pos_i}{|Position|} \tag{7}$$

Dans l'équation (7), *width* représente la taille d'un BBWS et % la fonction de modulo. Les chemins chauds sont choisis en fonction de leur couverture locale qui doit être relativement élevée, mais aussi de la distance de réutilisation qui elle doit être relativement petite.

Nous avons entrepris de valider notre approche sur un large sous-ensemble d'applications MiBench [42], la plate forme expérimentale étant composée de SimpleScalar [22] et de SALTO [16]. En somme, nous avons été dans la mesure de couvrir 48% du code dynamique par la détection des chemins chauds. Résultat impressionnant, compte tenu du fait que cette proportion du code représente seulement 0.15% du code statique. Afin de démontrer les avantages qu'une telle approche peut offrir, nous nous sommes aussi attelés à évaluer une technique d'optimisation visant à réduire la consommation d'énergie sous directive du compilateur. Nous avons notamment entrepris d'adapter la taille du cache de données en fonction de la taille des chemins chauds. Cette expérience a montré que 12% de gain en énergie pouvait être gagné en ajustant la taille du cache de données à celle d'un chemin chaud.

Techniques de reconfiguration du cache

Les caches mémoire permettent d'améliorer les performances en maintenant une fraction importante du code source et des données sur la puce, réduisant ainsi la consommation d'énergie liée aux trafics mémoires. Cependant, à cause du haut niveau d'intégration, un cache peut occuper jusqu'à 50% de la surface d'une puce, dissipant ainsi une large part d'énergie. Afin de palier ce problème, des caches mémoire configurables ont donc été progressivement introduits dans certains systèmes [44, 70]. L'intérêt des caches mémoire configurables pour les systèmes embarqués est cependant encore assez limité. En effet, actuellement la plupart des solutions proposées offrent la possibilité de configurer un cache mémoire qu'une seule et unique fois avant l'exécution de l'application.

Nous proposons dans le Chapitre 4 de modifier la structure d'un cache configurable afin de permettre au compilateur de le reconfigurer en fonction des changements de phase dynamique de l'application. Nous montrons que le modèle de cache que nous introduisons offre un fort potentiel de réduction de la consommation d'énergie, tout en se gardant également de trop dégrader les performances.

L'idée de ce chapitre découle de la constatation que au cours d'une exécution, l'utilisation du cache varie beaucoup et donc que sa configuration optimale n'est pas toujours la même. De ce fait, il est indéniable qu'un cache reconfigurable sur la base de phases d'exécution, et non sur la base d'une exécution complète de l'application, est susceptible d'apporter la flexibilité nécessaire pour réduire la consommation d'énergie.

Nous réutilisons deux techniques récemment introduites pour faciliter la reconfiguration du cache. Toutes ces deux techniques adressent la reconfiguration du cache au niveau de l'application.

1. La technique *selective cache way* proposée par Albonesi [1]. Cette technique propose de partitionner un cache associatif le long de ses étiquettes et de ses données. L'énergie peut être sauvée en éteignant certains bancs du cache sur demande, suivant la taille du cache requise pour l'exécution de l'application. Les modifications apportées au matériel sont simples, avec seulement un masque de registre pour activer/déactiver des bancs de cache. Cependant, cette approche peut seulement être adaptée à des caches totalement associatifs;

2. La technique *way-concatenation* proposée par Zhang et al. [117]. Dans cette technique, les auteurs proposent d'exploiter l'arrangement en bancs d'un cache pour parvenir à le reconfigurer comme un cache à correspondance directe ou un cache associatif mais avec un degré d'associativité moindre. L'arrangement proposé exploite une technique appelée le *way-concatenation* qui permet de fusionner des bancs du cache tout en maintenant toujours toute la capacité du cache. Cette approche réduit l'énergie dynamique puisque, avec la même taille de cache, des caches d'associativité inférieure opèrent moins d'activités de commutation que ceux à plus fort degré d'associativité. De plus, le coût d'implémentation s'est avéré minimal.

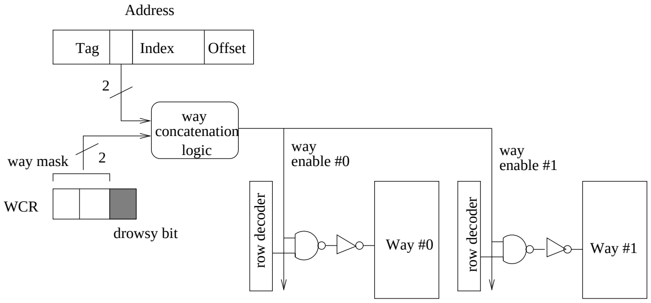


Figure 3: Modèle de base (associativité de degré deux).

L'idée proposée est donc d'employer une combinaison de ces deux techniques afin de pouvoir modifier la taille et/ou l' associativité d'un cache. Le modèle de cache sur lequel nous nous basons et que nous allons progressivement modifier pour parvenir à un cache reconfigurable par-phase est celui de [117] (cf. Figure 3)..

Nous avons introduit une première modification qui vise notamment à préserver la cohérence des données entre deux reconfigurations de cache. En effet, le problème posé est que en passant d'une première configuration A à une deuxième configuration B, plusieurs lignes de cache peuvent être dupliquées, causant ainsi une pollution dans le cache. Pour éviter ces effets de bord, il est impératif d'invalider à chaque écriture toutes les lignes de cache qui ont été dupliquées. Nous avons émis l'idée de préserver les étiquettes de toute reconfiguration; ceci afin de pouvoir contrôler tous les accès en écriture sur les bancs de données et donc provoquer une invalidation sur toutes les autres lignes dupliquées.

La seconde modification consiste à préserver les données dans les bancs éteints entre deux reconfigurations de cache. En effet, ce qui se passe est que, chaque fois que le cache doit être redimensionné suivant sa taille, un banc doit être déconnecté, causant ainsi la perte des données qui y sont stockées. Pour palier ce problème, nous avons modifié la structure du cache pour l'adapter à un mode dit de *sommeil* [38]. Dans ce mode, la tension d'alimentation est réduite pour permettre la préservation du contenu des cellules mémoires (cf. Figure 4). En plus de préserver le contenu des cellules mémoires, cette technique présente l'avantage de réduire l'énergie statique de manière non négligeable.

Nous avons étudié le comportement de plusieurs applications de MiBench et de Powerstone afin d'extraire certaines caractéristiques dynamiques liées aux accès au cache de données. L'étude du comportement de ces applications nous a permis de déceler certaines affinités entre la taille de cache et le degré d'associativité pour différentes configurations de cache possible (voir Figure 5).

Ce que l'on peut constater à travers cette figure c'est que certaines applications restent insensibles à un changement de la taille du cache, la différence dans ce cas de figure se faisant surtout au niveau du degré d'associativité. Pour d'autres applications cependant, c'est le contraire qui se produit; le degré d'associativité important peu par

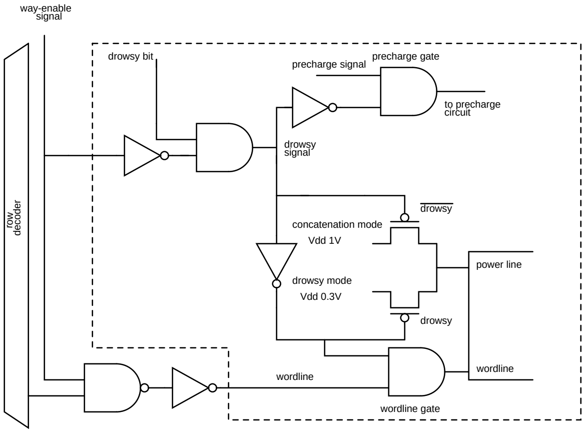


Figure 4: Ligne de cache en mode sommeil.

rapport à la taille du cache. Suite à ces observations, nous avons adapté la taille et/ou l'associativité du cache en fonction des caractéristiques de l'application. Les gains d'énergie dynamique restent, somme toute, modérés, allant de 5% à 12%. Nous avons aussi noté une dégradation des performances pouvant atteindre 31%; ceci étant dû à l'emploi de technique de mise en veille, le réveil coûtant de l'ordre de un à deux cycles. De très nets gains en énergie statique ont cependant été notés pour la majorité des applications, le meilleur gain s'élevant à 80%.

Techniques de reconfiguration du chemin de données

Avec l'accroissement des performances, plusieurs processeurs des nouvelles générations arrivent sur le marché équipés de dispositifs architecturaux très agressifs dont le but est de récolter le maximum de performance. Cette tendance s'est également accompagnée d'une augmentation continue de la taille de mot. Une des raisons qui explique cela est de satisfaire aux besoins d'un espace d'adressage plus grand. On trouve aussi une justification du côté d'une plus grande largeur de bande passante mémoire. Cette tendance qui s'était tassée autour d'une largeur du chemin de données de 32 bits croît progressivement, soutenant maintenant jusqu'à 64-bit pour de nouveaux processeurs tel que l'Itanuim. Cette augmentation de la largeur du chemin de données a surtout été bénéfique pour la grande majorité des applications dominées par le traitement scalaire sur des types de données entier de 32 bits. Cependant, avec la récente confluence des applications à usage universel et multimédia dans les processeurs embarqués modernes, ceci n'est plus du tout le cas; plusieurs de ces dernières applications opèrent maintenant sur des largeurs de données plus étroites, par exemple 8/16 bits de données, ouvrant ainsi de nouvelles opportunités de réduction de la consommation d'énergie.

Une récente étude menée par Brooks et al. [18] propose de tirer profit de ces opérandes de petite taille pour réduire la consommation d'énergie du chemin de données d'un

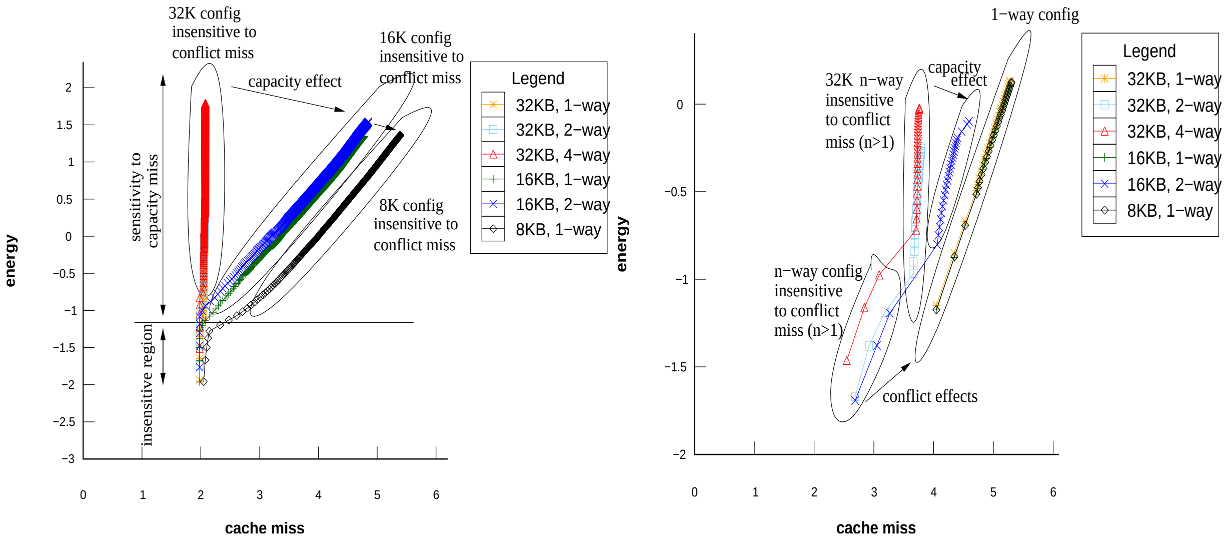


Figure 5: (a) gsm energy/performance profile; (b) fft energy/performance profile.

processeur. L'approche proposée par ces auteurs est basée entièrement sur le matériel et exploite les opérandes de petite taille de façon dynamique. Au niveau du compilateur, cependant, aucune technique similaire n'existe pour le moment. Certaines approches logicielles essaient tout de même de déterminer de manière statique la taille de variables de programme [21, 105, 26, 69]. Ces techniques sont cependant de portée très limitée puisque les contraintes d'analyse statique imposent des restrictions très sévères au niveau du choix des variables à analyser et donc des transformations à appliquer. Par ailleurs, ces transformations doivent en plus se conformer de façon très stricte à la sémantique du programme, réduisant un peu plus les opportunités d'optimisation.

Dans le Chapitre 5, nous proposons une nouvelle approche pour évaluer les opérandes de petite taille au niveau du compilateur. Cette approche intègre logiciel et matériel à la fois. L'idée principale dans ce chapitre est d'exposer les opérandes de petite taille obtenues par le biais du profilage au compilateur afin de lui permettre de basculer de manière spéculative dans un mode d'exécution à taille de mot réduite. L'intérêt que présente cette approche est que le compilateur peut désormais s'attaquer à un plus grand sous-ensemble d'opérandes de petite taille, contrairement à une approche purement logicielle. Comme cela se fait généralement avec une granularité beaucoup plus grosse au niveau du compilateur, il y a donc là la possibilité d'éteindre certains éléments du processeur sur des périodes beaucoup plus longues que cela ne se fait avec une approche matérielle. Ce mécanisme étant fondamentalement spéculatif, un dispositif matériel permet de revenir à un mode d'exécution normal en cas d'erreur.

Notre approche se fonde sur le fait que certaines instances dynamiques de bloc de base présentent un nombre élevé d'opérations s'exécutant avec des données de petite taille, comme le montre la Figure 6 obtenue par le biais du profilage. Pour tirer profit de ces opérations au niveau du compilateur, il a fallu apporter des modifications à

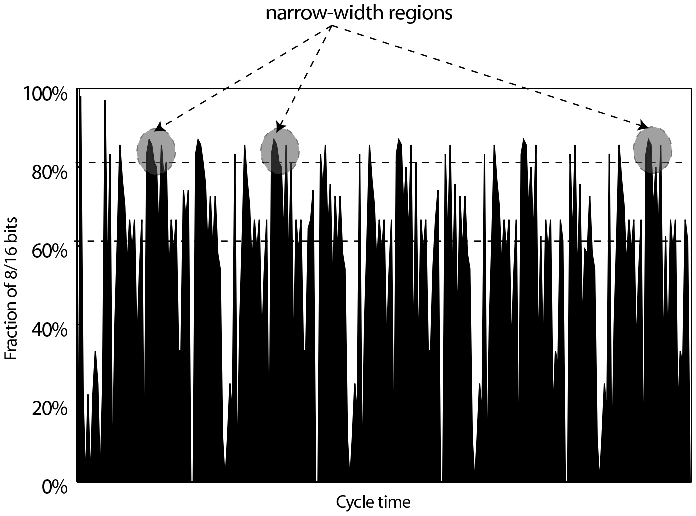


Figure 6: Distribution dynamique des opérandes de petite taille au niveau des blocs de base*adpcm*.

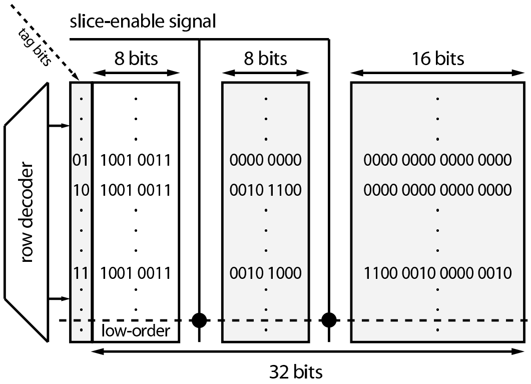


Figure 7: Fichier de registre byte-slice.

l'architecture et développer de nouvelles techniques d'optimisation.

Le premier support architectural que nous avons introduit s'attaque au fichier de registre. L'objectif ici est de modifier la largeur du fichier de registre afin de l'adapter à des modes d'exécution sur des tailles de mot de 8, 16, ou 32 bits. L'architecture *byte-slice* proposée offre cette fonctionnalité. Le principe est de partitionner de façon logique un fichier de registre conventionnel en 3 bancs de registres de taille de mot de 8, 8, et 16 bits, respectivement. Un signal *slice-enable* permet d'activer ou de désactiver les deux derniers bancs logiques du fichier de registre, le premier banc étant maintenu actif de façon permanente. Lorsque le signal assurant l'activation d'un banc logique est désactivé, celui-ci est mis en mode *sommeil* afin de préserver le contenu des cellules mémoires. La reconfiguration du chemin de données du processeur peut également s'étendre au pipeline tout entier. Pour ce faire, nous avons entrepris la mise en oeuvre

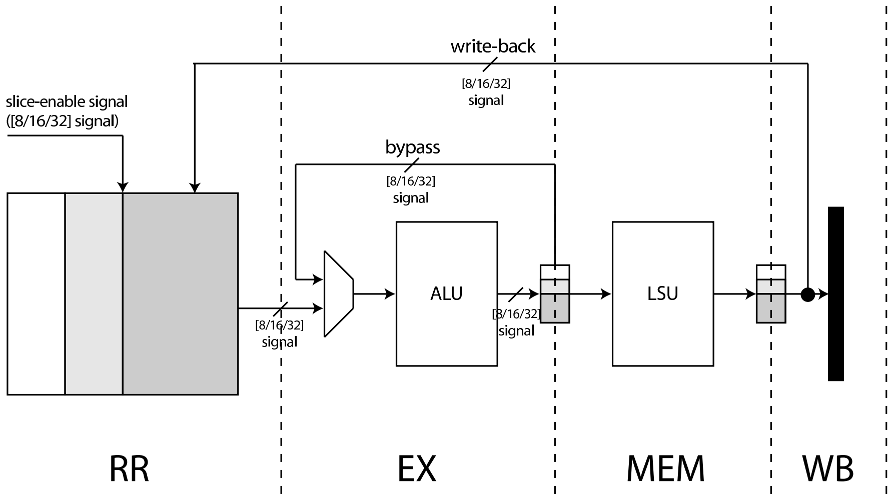


Figure 8: Pipeline.

des techniques de *clock-gating* sur certains composants du pipeline comme les *latch* et les unités arithmétiques et logiques. Le schéma du fichier de registre ainsi que celui du pipeline sont donnés sur la Figure 7 et la Figure 8, respectivement.

Les erreurs de prédiction du mode d'exécution sont gérées par le matériel. Un dispositif simple de recouvrement d'erreurs assure le basculement du mode d'exécution du processeur opérant sur une petite taille de mot à un mode d'exécution exploitant toute la largeur du chemin de données, c'est-à-dire avec une taille de mot sur 32 bits. Des étiquettes rattachées à chaque registre servent de support pour la mise en oeuvre de cette fonctionnalité (cf. Figure 7). Ces étiquettes ont pour but notamment d'indiquer la largeur normale du chemin de données. Elles sont générées à chaque fois qu'un nouveau résultat est produit par l'unité arithmétique et logique et à chaque fois aussi qu'une donnée est lue en mémoire. La vérification du mode d'exécution est entreprise à l'étage d'exécution du pipeline, au niveau de l'unité arithmétique et logique, comme l'indique la Figure 9.

Afin d'augmenter les chances d'obtenir des régions contenant un nombre élevé d'opérandes de petite taille, nous avons considéré le cas particulier des instructions d'accès mémoire. Ces instructions manipulent des adresses mémoire de taille de mot très souvent supérieure à 8 ou 16 bits. Nous avons donc considéré une architecture dans laquelle le calcul d'adresse est séparé de l'accès mémoire, un peu à l'image d'une architecture découplée. Cependant, au contraire d'une architecture découplée, nous avons supposé l'existence de registres spéciaux entièrement dédiés au calcul d'adresse (par exemple des accumulateurs).

La formation des régions contenant des opérandes de petite taille est entreprise par le compilateur. Ces régions sont préalablement annotées lors d'une phase de profilage et sont transformées de la manière suivante:

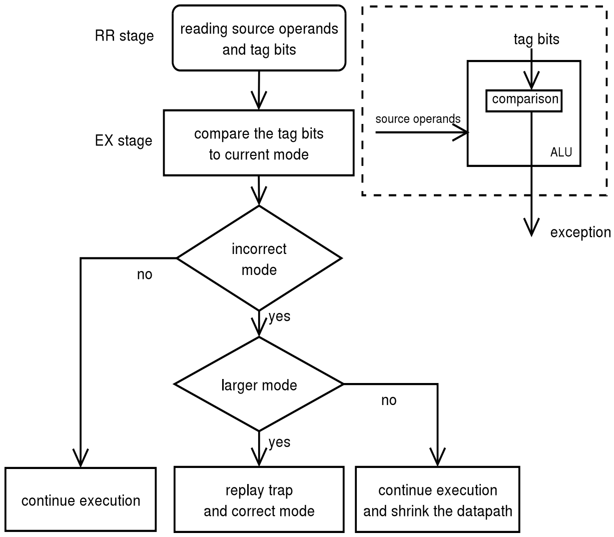

RR stage
reading source operands and tag bits
EX stage
compare the tag bits to current mode
tag bits
comparison
source operands
ALU
exception
incorrect mode
no
yes
larger mode
no
yes
continue execution
replay trap and correct mode
continue execution and shrink the datapath

Figure 9: Mécanisme de recouvrement d'erreur.

1. les instructions d'accès mémoire dont l'un des opérandes au moins est représenté sur une taille de mot de plus de 16 bits sont transformées en une instruction de calcul d'adresse et une instruction d'accès à la mémoire via un registre accumulateur.

2. toutes les instructions d'un bloc de base sont réordonnancées de telle sorte que chaque instruction ayant au moins une opérande de 32 bits est déplacée dans un bloc de base voisin (voire Figure 10).

Nous avons mené des expérimentations en utilisant des applications Powerstone. Nos expériences ont montré que les dégradations de performance étaient fortement liées à la pénalité associée à une mauvaise prédiction. Pour des pénalités faibles, de l'ordre de 5 cycles, les dégradations sont négligeables et n'affectent presque pas les performances. En revanche, pour des pénalités élevées, de l'ordre de 25 cycles, les dégradations peuvent très vite atteindre 30% ou 60% selon que le taux de prédiction est faible ou fort. De manière générale, on a constaté un gain de 17% d'énergie dynamique et de 22% d'énergie statique sur l'ensemble des applications.

Conclusions

En conclusion, cette thèse a abordé différentes techniques pour trouver un compromis entre performance et consommation d'énergie pour des processeurs embarqués du type VLIW. Après avoir étudié les limites imposées par une approche logicielle, notre recherche nous a conduit à considéré des solutions mixtes, intégrant le logiciel et le matériel à la fois.

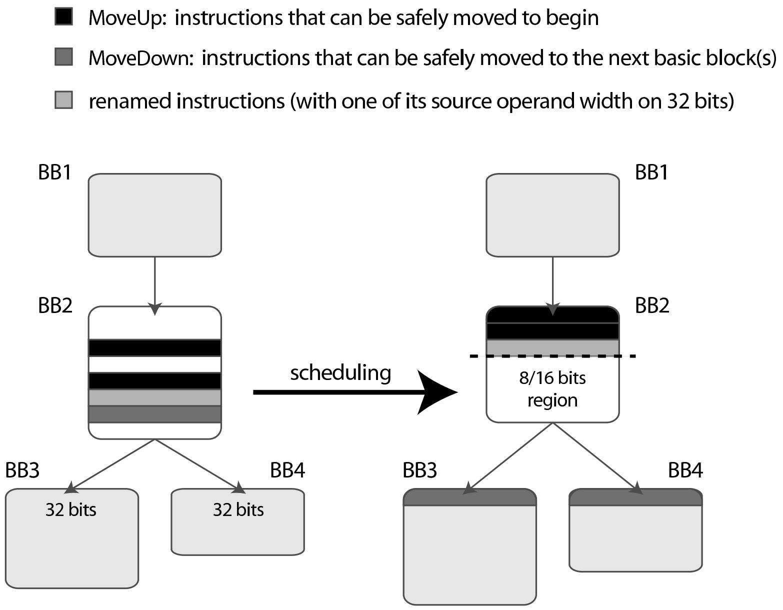

MoveUp: instructions that can be safely moved to begin
MoveDown: instructions that can be safely moved to the next basic block(s)
renamed instructions (with one of its source operand width on 32 bits)
BB1
BB2
BB3
BB4
32 bits
32 bits
scheduling
BB1
BB2
8/16 bits region
BB3
BB4

Figure 10: Ordonnancement de code à l'intérieur d'une région.

Dans un premier temps, nous nous sommes d'abord intéressés à l'étude statique du comportement dynamique d'un programme. L'intérêt d'une telle approche est de pouvoir identifier les chemins les plus fréquemment exécutés d'un programme, autrement dit les chemins chauds. A cet effet, nous avons introduit une nouvelle méthode statique d'identification des chemins chauds qui se base sur des tableaux de suffixe. L'approche proposée montre que 48% du code dynamique peut être couvert par les chemins chauds, la taille du code statique équivalent ne représentant que 0.15% du programme. Cette approche, comparée aux techniques actuellement proposées, permet l'implémentation d'algorithmes de recherche de pattern dans une trace en temps $O(\ln(N))$, N étant la taille de la trace à analyser.

Avec l'identification des chemins chauds, on peut extraire certaines caractéristiques dynamiques liées à l'exécution du programme. Au regard de cette caractérisation des chemins chauds, on peut se permettre d'adapter le matériel en fonction des exigences dynamiques de l'exécution du programme. Dans un deuxième temps donc, nous nous sommes intéressés à la reconfiguration du matériel sous directive du compilateur. Plus particulièrement, nous nous sommes intéressés à la reconfiguration des caches mémoire et du chemin de données du processeur. Concernant les caches mémoire, nous avons montré que par rapport à une approche de reconfiguration du cache au niveau de l'application, un modèle de cache reconfigurable sur la base de phases d'exécution d'un programme peut apporter des gains substantiels en énergie dynamique (allant de 5 à 12%) et en énergie statique (de l'ordre de 80%). De la même manière, on a montré qu'en exposant les opérandes dynamiques de petite taille au compilateur, celui-ci pouvait de façon spéculative adapter la taille du chemin de données à celle des opérandes contenues dans une région donnée. Nous avons montré qu'un gain de 17% d'énergie dynamique pouvait ainsi être gagné et qu'une économie d'énergie statique de 22% était également réalisable sur le fichier de registre.

Thesis fundamental

Over the past decade, the explosion of the embedded applications market has favored a large dissemination of a variety of embedded devices. Our everyday life is overwhelmed of a plethoric number of such devices that are capable of handling various data types including digital video, audio, graphics and text, to quote only those. These devices range from very small, low-end, systems such as those used in telemetry counter sensors, to mid-range portable systems such as mobile phones or PDAs, up to high-end embedded systems such as those found in radars or set-top boxes. Obviously, the need for the software developers to continuously improve the performance of these applications has put a strong emphasis on the processing power demand of modern embedded processors. This enthusiastic quest for performance has been translated into increased architected functionalities that have been added to these processors in order to reap maximum performance, providing them with performance levels that were unaffordable a decade ago, even for their counter-part general purpose processors.

It is tempting to optimize a processor along one system design aspect to achieve high-performance. However, when dimensioning an embedded system to meet this performance demand, the ultimate achievable performance level may not be the only target. There is an important tradeoff among the system design parameters that makes that it becomes practically impossible to optimize one parameter in isolation to the others without risking to worsening overall performance. As a consequence, embedded system designers must in addition address emerging key design issues such as that of power consumption, which has a growing impact on the overall system cost. As for example, when designing for high-performance, it is usual to increase the clock frequency or augment the degree of instructions level parallelism (ILP), e.g. through the duplication of functional units. While this does actually improves performance, it comes at the cost of a growing power consumption envelope. In fact, whereas increasing the clock frequency has a linear effect on power, the power growth factor due to increased hardware complexity can raise more than quadratically with respect to the issue width [80] as shown in Equation (8). In this equation, γ is a power growth constant which depends on the hardware structure and IPC the average number of executed instructions per cycle which is closely tight to the issue width. For values of $\gamma > 2$, such as that corresponding to multiported register files [120], the power growth factor can be quite significant.

$$Power \sim (IPC)^{\gamma} \tag{8}$$

Considering the power consumption when designing embedded systems is therefore becoming a compelling problem such that it is even seen as one of the major challenges of the decades to come [76]. The reason why this challenge will keep up to date may be evidenced by the ever-increasing need for the processors manufacturers to stay competitive and offer attractive products. As a matter of fact, reducing the energy consumption of a battery-powered appliance has a direct consequence on the product's worth to the consumer. As an example, it can translate into a lengthened autonomy of the appliance. In addition, it can also guarantee a lowering of the overall system cost since it may then be possible to design the battery more efficiently, e.g. with a less weight for example.

In more general terms, the power consumption problem appears to be a real stumbling block in the application of the Moore's law since it directly sits on the path towards achieving higher performance. Indeed, with the Moore's law driving the semiconductor industry ahead, the processor's performance limits are pushed at the forefront every one and half to two years on average, causing the integration density to continue to double in the same time span. However, as the integration density doubles, so too do the number of active gates on a chip, raising the dynamic power consumption in an exponential manner. The net effect is a decreasing processor reliability due to the concern of power dissipation, which has much to do with maintaining the processor operating mode below a given temperature level. This has a direct influence on the packaging cost and hence on the overall system cost.

The other implication that the power consumption problem has with the Moore's law is a matter of keeping it alive for a while, at least for the few next generations of embedded processors. As the technology scales down below 100 nanometer, maintaining the Moore's law up to date is becoming a very challenging problem, essentially because of technology limits [56]. In fact, as the technology shrinks, the power supply voltage must also be reduced in order to maintain the dynamic power consumption within an acceptable level. This reduction of the power supply voltage must also be accompanied with a reduction of the threshold voltage to provide sufficient noise margins to insure a reliable functioning of the processor. However, as the threshold voltage, V_{th}, is lowered, the current, I_{leak}, that leaks through the transistor when it is switched off increases exponentially, exacerbating the static power consumption problem even more. This relationship is illustrated in Equation (9), where k is a parameter depending on the device, a a constant slightly greater than 1 and V_T the thermal voltage [47].

$$I_{leak} \sim k * e^{\frac{-V_{th}}{a*V_T}} \tag{9}$$

Henceforth, to reduce the power consumption has since evolved as a very active field of research. Until recently, the vast majority of the research devoted to this topic have mainly focussed on mastering the power consumption at the circuit and the architecture levels. A good survey of these techniques is described in [13]. This has been done at the detriment of software approaches. Albeit the power consumption is strongly dependent on the processor architecture, it is principally the execution of a program that causes power to be consumed. This is essentially true in the context of embedded systems, as it is evidenced in [107]. This later observation stems from the fact that it is the dynamic execution of programs that exercises the hardware, causing it to operate certain amount of transitions, thereby dissipating power. These transitions often account for the so-called switching activity factor, denoted by a, whose relation to the dynamic power consumption is given by the first expression of Equation (10), under the term *dynamic power*. In this expression, C models the gate capacitance of a CMOS node, V_{dd} the power supply voltage and f the processor clock frequency. The second expression models the static energy due to the current that leaks through the transistor when it is turned off.

$$P = \underbrace{a * C * V_{dd}^2 * f}_{dynamic\ power} + \underbrace{V_{dd} * I_{leak}}_{static\ power} \tag{10}$$

For many embedded systems where the compiler is in charge of delivering a good code quality, Equation (10) may suggest us a power control mechanism for the compiler to effectively leverage the switching activity factor, namely by mastering the program code to produce an energy-oriented schedule that minimizes the value of a. On the other hand, an effective way to tackle the static energy at compile time may be to reduce the impact due to the leakage current. This may be achieved by tracking hardware resources in software that may temporally be put out of use at runtime, shutting off the power supply voltage to these resources [91]; thereby reducing the impact due the I_{leak} term, i.e. leakage power. There may even be some more interesting opportunities to improve further the overall system's energy-efficiency by exploiting the interplay between the code being executed and the exercised architectural components. This may involve looking at synergistic architecture-compiler approaches that integrate hardware and software aspects.

This thesis fits generally to this context. The question that we will be always referring to throughout this thesis is how to efficiently master power consumption at the compiler level, while also addressing sustainable performance levels.

Terminology

We assume working with constant frequency. It follows immediately that power and energy can be used interchangeably. Hence, throughout this thesis, otherwise specified, these terms will both refer to the same notion. In this sense, when speaking about energy-efficiency, we also refer to power-efficiency. In the same way, energy-performance or power-performance will reveal the same concept.

Power-performance tradeoff

The processor power consumption can be reduced by tackling any of the parameters shown in Equation (10). Consider, for instance, the expression of the dynamic power consumption. Reducing the power supply voltage to almost a half may reduce the dynamic power consumption by almost a quarter. In the same manner, reducing the clock speed may also reduce the power consumption. Therefore, one can think of leveraging the value of V_{dd} or f at the compiler level to lower the amount of power consumption. Unfortunately, since the clock frequency is approximatively linear in V_{dd} [56], as shown in Equation (11), reducing either one of these values may directly impair on performance.

$$f \sim \frac{V_{dd} - V_{th}}{V_{dd}} \tag{11}$$

To obtain a significant power reduction without compromising performance, it may be requested to leverage parameters that may influence each other. Adjusting the power supply voltage and the processor clock frequency in this way, for instance, has already been the subject of many studies done at the compiler level, some of which are well described in [48, 94].

To consider the power-performance tradeoff may then result to solve an optimization problem that consists in finding the point of balance where any reduction of one parameter is compensated with an equivalent adjustment of the other parameters such that any performance impairment due to the first is nullified by the others. This steady quest for balancing between power consumption and performance will underly our definition of energy-efficient compilation throughout the rest of this thesis.

Objectives of this thesis

The goal of this thesis is to develop a deep understanding of the main sources of power consumption, viewed from both a hardware and software perspectives, in order to come out with architecture-compiler symbiosis solutions that can better improve overall energy-efficiency, without jeopardizing performance.

Our analysis standpoint lies in the compiler. Therefore, we will consider program characteristics in order to converge towards architecture-compiler synergistic solutions that can exercise the hardware and the software. Our technical approach to this problem stems from the fact that programs exhibit dynamic execution patterns [97] that can stress the hardware in different ways. Identifying these patterns in software and then looking precisely on how they may exercise the underlying hardware components may lead us to establish effective power control mechanisms. This will imply from us to leverage the values of the activity factor a in consonance with the other parameters such as the power supply voltage V_{dd}. We propose to address these issues by investigating four main areas of research:

1. **ILP compilation**. A focus is put on understanding the main effect of ILP compilation on the energy by trying to trace back the main sources of the diminishing energy returns when applying these techniques;

2. **Program behaviors**. By attempting to understand the behavior of programs, we expect to reduce the discrepancy between the quality of the code being produced by the compiler and the underlying hardware capability. The goal is to characterize a program execution behavior to achieve a better power-performance tradeoff;

3. **Cache subsystem**. Caches account for a large portion of the chip area and therefore the energy [45]. We study the cache requirements of different applications, expecting to isolate some behavioral software/hardware particularity common to both the applications and the cache components to better leveraging the power-performance tradeoff;

4. **Processor datapath**. The processor datapath becomes increasingly critical as the pipeline is widened to process multiple instructions simultaneously. We investigate the challenging issue of managing this complexity at the compiler level in order to balance the energy and the performance in a convenient manner.

Contributions of this thesis

In order to address the goals stated above, this thesis investigates several key research contributions related to each of the topics mentioned in the objectives.

1. We propose a thorough investigation of the energy-performance tradeoff of ILP compilation techniques. The main objective here is to understand the classical view of optimizing for power which mainly relies on performance optimization to achieve this goal. We challenge this classical view of power management by proposing an analytical methodology which essentially exploits the variations in program performance to identify conditions leading to energy consumption increase;

2. We propose a novel approach based on suffix arrays for characterizing dynamic program behaviors via the static identification of hot program subpaths. We show the potential of such an approach to improve the compiler efficacy to selectively applying some optimizations on portions of code based on the underlying program characteristics;

3. We introduce a potential phase-based reconfigurable cache scheme that can be exploited at compile time to leverage the energy-performance tradeoff. This proposal

is based on a thorough analysis of the dynamic cache requirements of programs. The main goal here is to characterize some application-specific cache architectural parameters that can be managed relatively easily at compile time for improving the energy-performance tradeoff;

4. We introduce bit-width speculation, principally as a means for managing the energy consumption of a processor datapath at the compiler level. We advocate changes in the ISA to virtualize both the processor datapath and the register file widths in order to exploit narrow-width operands at the software level more efficiently. We achieve this by taking advantage of the strong bitwidth locality available in many embedded applications in order to speculatively accommodate the execution of narrow-operands regions on narrower datapaths.

Organization of this thesis

This thesis is organized into five chapters.

- **Chapter one** first describes the general implementation framework of VLIW processors. In particular, we lay emphasis on the processor implementation specifics and on the role played by the compiler. A comprehensive understanding of the main power consumption issues is then introduced. We present some terminology associated with power consumption, describe the metrics used to characterize power as well as some available power evaluation models and tools. Particular attention is paid to the opportunities that can be exploited at compile time to leverage the power-performance tradeoff;

- **Chapter two** investigates in more detail the ILP related power issues;

- **Chapter three** describes our approach for characterizing program behaviors at compile time and evidences the potential of such an approach by exhibiting some selected optimizations techniques that may improve the overall energy-performance tradeoff;

- **Chapter four** introduces a potential phase-based reconfigurable cache design that can be managed at the compiler level to reduce energy;

- **Chapter five** addresses a new hardware/software approach to speculatively control the processor datapath power consumption at the compiler level;

Chapter 1

Background

Historically, there have been two major orthogonal directions for improving processor performance: processor designers have either increased the clock frequency or they have augmented the processor's capability of executing more than one instruction per cycle. In the former case, improvement in the clock frequency has followed an approximate rate of 30% every two to three years [17], primarily because of the implementation of ever deeper pipelines [104] with less logic per pipeline stage and, to a lesser extent, of technology scaling [17]. In the latter case, processor designers have exploited the increasing chip integration density to enhance the instruction throughput by incorporating more hardware to process multiple instructions per cycle [103]. Unfortunately, as we are nearing closer to technological limits, it will soon become very difficult to continue achieving ever narrower process scaling. This will make increasing the clock frequency and the chip integration density a very challenging task. Therefore, new approaches for realizing more performance are becoming urgent. This observation is shared among members of the research community since the interest to invest much more efforts in the study of complexity-effective designs that can potentially better exploit available processor resources is growing.

One of the most promising approaches for improving processor performance without taking much care of technological improvements is exploiting instruction parallelism. Basically, instruction parallelism exploits program characteristics to better take advantage of the available chip area by increasing the number of instructions that can be executed simultaneously. There are two distinct approaches for doing that. In the first approach, conventional superscalar processors [103] dynamically search the sequential instructions stream for on-the-fly instructions that can be executed out-of-order. In the second approach, the responsibility of finding the instructions that may be executed in parallel falls by the compiler, and is therefore done entirely in software. This approach is implemented with Very Long Instructions Word (VLIW) processors [37]. More and more, however, there are many new approaches that tend to exploit this instruction parallelism by integrating hardware and software techniques even more tightly. In particular, some designs have emerged which try to explicitly encode the parallelism found at compile time in the instructions. This design is known as Explicitly Parallel Instruction Computing (EPIC). One such processor is the Itanium [96] from Intel.

The concept of VLIW architecture is becoming very popular in the embedded computing domain, principally because it provides a very low-cost, high-performance alternative solution compared to a conventional general purpose design such as a superscalar processor. For these reasons, VLIW architectures are starting to be adopted in a variety of processors, especially in the DSP domain. But also in general purpose domain with the Itanium [96] processor from Intel. The Philips Trimedia processor [34], the Texas Instrument TMS320C62xx [51], and the Lx processor from HP-LAB/STMicroelectronics [35] are examples of DSP processors that build on VLIW technology to accelerate program execution time. Basically, a VLIW processor allows to reap maximum program performance with a lower power budget and less design effort than a superscalar processor can do, especially because much of the circuit complexity found in the latter, which makes it possible to dynamically take advantage of the instruction parallelism, is withdrawn from the processor's critical path and placed into the compiler.

A VLIW processor exploits instructions level parallelism (ILP) by looking for program instructions at compile time that can be directly exposed to the hardware via the use of very long instructions word. Typically, a very long instructions word embeds from 2 up to 28 independent operations [67] that can be issued in parallel at each cycle. As such, VLIW processors offer a potentially powerful solution to sustain increased performance at low cost. In this chapter, we will provide a deep understanding of the main functioning of a VLIW processor. We will first look at the hardware side, describing the peculiarities of commonly found VLIW architectures. Then, we will look into the software aspects of VLIW architectures. In particular, we will be addressing the issues of exploiting instruction parallelism at the compiler level. Finally, we will provide a thorough analysis of the power consumption issues associated with such a VLIW processor.

The rest of this chapter is structured as follows. In the next two sections, we introduce the general framework of a VLIW processor, followed by the description of some essential techniques used to uncover instructions parallelism at compile-time. These two sections provide the fundamental background of VLIW processors. In Section 1.3, we introduce some basic notions of power consumption of CMOS circuits. Power consumption models and tools that will be subject to use throughout this dissertation are also described in this section. In Section 1.4, we address the sources of power consumption that we will look into throughout the rest of this thesis. Finally, Section 1.5 concludes this chapter.

1.1 VLIW architecture

A VLIW processor operates at the granularity of several independent operations packed into a very long instruction word, often called a bundle. The number of such independent operations varies from one target to another, but can easily reach more than 28

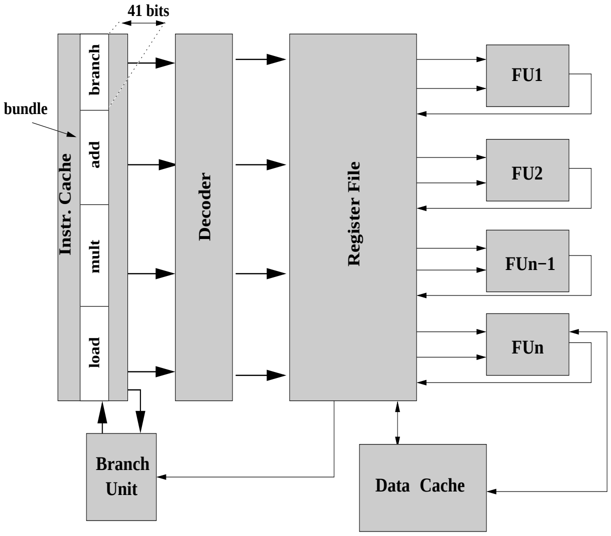


Figure 1.1: Logical view of a generic 4-issue width VLIW architecture.

operations as described in [67]. Each such independent operations within a bundle occupies a set of fields, called a slot. It is the compiler that is in charge of filling each bundle with sufficiently enough independent operations to maintain the hardware busy, i.e. reduce the number of empty slots. Figure 1.1 depicts the logical view of a generic 4-issue width VLIW processor.

A VLIW processor comprises a control unit that issues bundles, one at each execution cycle. As illustrated in the Figure 1.1, a VLIW processor is not equipped with a mean of hardware that can dynamically track runtime data dependences among instructions operands. It returns to the compiler to guarantee that no such dependences can occur at runtime. When such a bundle is issued, e.g. 4 operations as shown in the figure, it usually initializes several independent operations simultaneously. Typically, any such independent operation may be carried out on any available functional unit that is dedicated to this purpose. It is possible to increase the number of such available functional units to enhance the ILP degree. This comes however at the cost of additional complexity which may also have an impact on the power consumption. The format of a very long instruction word comprises a large number of bits that directly control the actions of the multiple functional units. Assuming, for instance, that each operation contained in the hypothetical bundle shown in Figure 1.1 is encoded with 41 bits, a very long instruction word will require a total of 128 bits to be represented in memory. Each very long instruction usually requires a small number of cycles to be carried out. This number of cycle can be predicted at static time in a relative easy way. A VLIW processor may also combine pipelining with parallelism to improve performance. To do this, a VLIW processor may comprise several pipelined functional units to augment instruction throughput. Generally, an instruction bundle on a VLIW processor appears compressed in memory to save code size. This is because, in the absence of sufficient

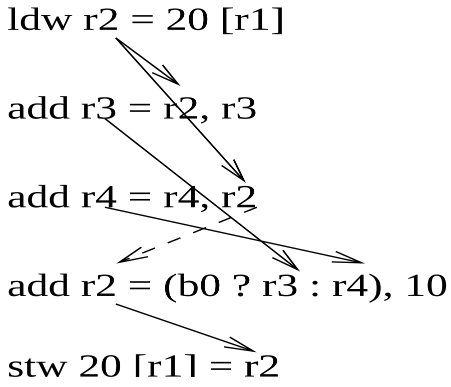


Figure 1.2: Original scalar code.

$ldw\ r2 = 20[r1]$	nop	nop	nop
nop	nop	nop	nop
nop	nop	nop	nop
$add\ r3 = r2, r3$	$add\ r4 = r4, r2$	nop	nop
$cmp\ b0 = r3, r4$	nop	nop	nop
$add\ r2 = (b0\ ?\ r3 : r4), 10$	nop	nop	nop
$stw\ 20[r1] = r2$	nop	nop	nop
nop	nop	nop	nop
nop	nop	nop	nop

Table 1.1: VLIW schedule for the scalar code of Figure 1.2.

parallelism, the empty bundle slots are filled with *NOP* instructions in an uncompressed encoding scheme. As these instructions may waste memory space, VLIW processor designers have engineered several compressed encoding techniques [65, 64, 15] to reduce this overhead, one of them being the use of stop bits.

1.2 VLIW compilation techniques

The role of the compiler in a VLIW processor is of a crucial importance since it entirely returns to him to reveal the parallelism hidden in the code as well as to resolve potential data and structural hazards that may occur. To do so, the compiler must search in the sequential stream of instructions for independent operations that may then be packed together into a very long instruction word. This process is known as instruction scheduling. Instruction scheduling poses a fundamental challenge to VLIW compilers. The challenge is to schedule as many independent operations as there are available empty slots in a bundle; yet with the additional constraints of minimizing the program execution time and using the hardware resources optimally.

Let us consider the sequential code shown in Figure 1.2. If we assume that all instructions take 1 cycle to execute, except those belonging to the class of load/store instructions, which will require 3 cycles, we obtain the VLIW code presented in Table 1.1. It is assumed that for this example, the VLIW machine used is that shown in Figure 1.1.

The code scheduled on the VLIW machine takes one cycle less to execute than that of the scalar machine. However, as it can be observed in the figure, the slot occupancy rate achieves only 17% of its potential, which is a relatively poor number. It is the compiler responsibility to fill the empty slots with operations to improve both the slots occupancy rate and the program execution time. This responsibility is very challenging since it involves to reveal sufficiently enough candidate operations to be scheduled within each single bundle as well as to guarantee the resolution of all data and structural hazards between these operations, e.g. RAW and WAR hazards as illustrated in Figure 1.2 with strong and dotted line, respectively. In the example shown in Table 1.1, to do so the compiler will have to look for at least 30 candidate operations to fill empty VLIW slots. If we assume an average basic block length of 5 instructions, the compiler will have to monitor operations from at least 5 different basic blocks in order to obtain a satisfactory slot occupancy rate. This severely complicates the scheduling task.

Fortunately, there have been several ILP enhancing techniques that have been engineered to circumvent that problem. All of the proposed solutions involve a kind of cross-block scheduling. These techniques essentially differ in the manner by which the compiler combines the different basic blocks together to increase its scheduling scope. In this section, we will only look at some of the major techniques that will be relevant for the remainder of this thesis.

1.2.1 Trace scheduling

Typically, when a program executes, it goes through different sets of basic blocks across the CFG. Fisher [36] has termed this a *trace* (see Figure 1.3) to denote a loop-free path of likely executed basic blocks in the CFG, for a given input data. Since a trace can include up to several basic blocks, it can therefore reveal much of the potential program parallelism than a single basic block can do. In the case the exposed parallelism is still not sufficient, the size of the trace can be enhanced with techniques such as loop unrolling or inlining [75] to include even more instructions. Since, during a program run, some basic blocks are been executed more often than the others, traces can also be arranged according to their execution frequency. Consequently, to form a trace it is therefore important to rely on profiling information to gather basic block execution counts, as shown in Figure 1.3. In that figure, B1, B3 and B5 are the most executed basic blocks. Since this constitutes a path in the CFG, they may be good candidates for forming a trace.

The process of forming a trace involves to choose a seed basic block with the highest execution frequency which has not yet been scheduled before. After that, the trace is grown backward and forward from the seed in the following manner. In the forward

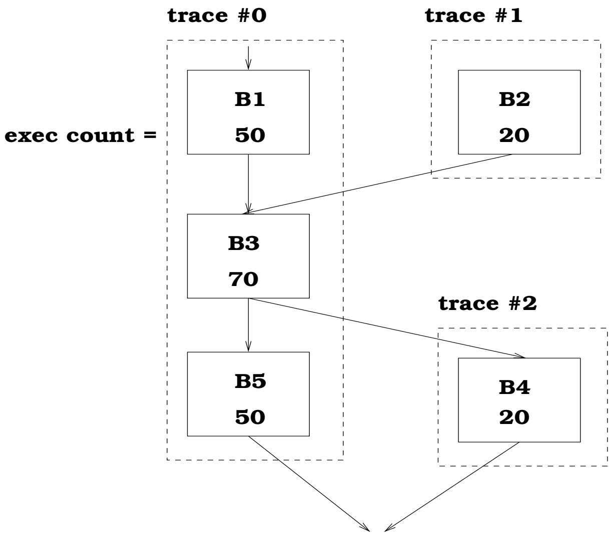


Figure 1.3: Example of a trace.

direction, the block at the head of the most frequently executed edge of the last basic block of the trace is added to the trace as long as this block has not been selected before. The trace is grown in the backward direction in an analogous manner.

Once the traces have been formed, the scheduling is done relative to the execution frequency of each trace, beginning with the trace with the highest execution frequency to that with the less. The scheduling can be done with a relatively simple list scheduling algorithm, such as that used when considering a single basic block.

However, a disadvantage with this approach comes from the fact that a trace can span over several basic blocks; thus forcing the compiler to insert some form of compensation codes to correct the inevitable effect of considering the entire trace as a single basic block. These cases arise as soon as an instruction in the trace is scheduled ahead or behind side exits (e.g. branch going out of the trace) or side entrances (e.g. branch coming into the trace). As for example, if an instruction preceeding a side exit is moved past this point, then a copy of this instruction must be inserted on the path leaving that point. Conversely, an instruction that is moved ahead of a side entrance must also be copied on the path coming into that point to correct this effect. We have illustrated these two possible scenarios in Figure 1.4.

As pointed out earlier, a serious disadvantage of trace scheduling is due to the amount of bookkeeping codes that is inserted in order to account for the effect of code motion before/past side entrances/exits of a trace. This can eventually slow down program execution. A typical scenario occurs when no dominant traces can be found in the program or when the resulted basic blocks (after trace formation) execute with nearly the same frequency. What happens is that some basic blocks may be slowed down due to executing these additional instructions. Since these latter instructions may be transparent to program execution, this can be eventually translated into program overhead. We

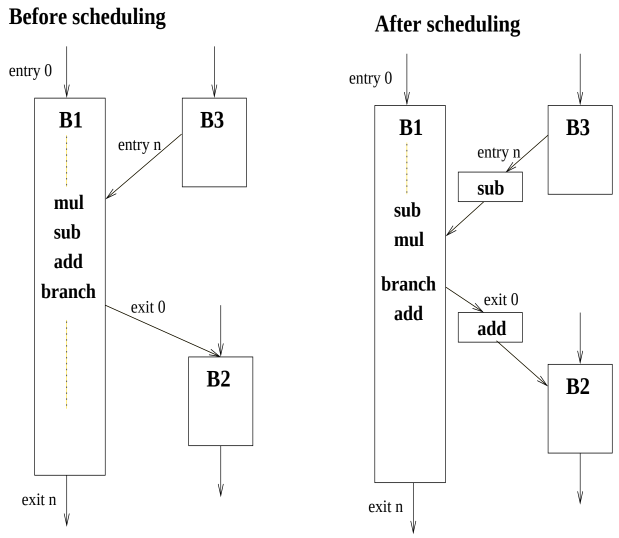


Figure 1.4: Example of insertion of compensation code.

Benchmarks	**IPC**		**ICache miss rate (%)**	
	Without traces	With traces	Without traces	With traces
adpcm coder	1.81	1.97	0.158	0.166
adpcm decoder	2.38	2.59	0.172	0.187
autcor	1.07	1.74	0.366	0.882
fir4	1.05	1.72	0.254	0.647
dct	1.47	1.80	0.962	1.773

Table 1.2: Effect of trace scheduling on the performance using a 4-issue width VLIW machine.

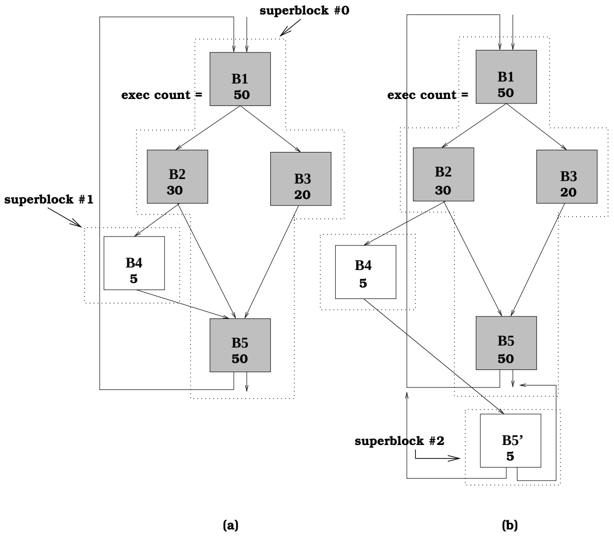


Figure 1.5: Superblock example, (a) trace selection, (b) after tail duplication.

have illustrated in Table 1.2 the effect of applying trace scheduling on some multimedia kernels, assuming a 4-issue width VLIW machine similar to that described in [35]. As it can be seen from the table, it is obvious that applying trace scheduling improves the performance since the *IPC* is increased for all benchmarks. However, when looking at the impact of trace scheduling on the instruction cache, one can observe that the miss ratio is degraded in almost all cases. This can be primarily attributed to the effect of executing the additional compensation codes. We will see in the next chapter that the impact of this compensation code can mitigate program performance when energy consumption is of a concern.

1.2.2 Superblock scheduling

The superblock [49] is an improved version of a trace in which no side entrance is allowed. A superblock is therefore akin to a trace, except that it can only be entered from the top. The process of forming a superblock proceeds similarly to a trace. The code is first profiled in order to gather basic blocks execution statistics to form a trace, as described in Section 1.2.1. Yet, however, in order to avoid any side entrance effects, a second step called tail duplication is applied [28]. The effect of tail duplication is to remove side entrance edges such as that shown in Figure 1.5(a), i.e. edge B4-B5. In this example, after applying tail duplication, basic block B4 points to the new duplicated node B5'. Because a superblock does include no side entrance, the application of some optimization such as constant propagation can be done more easily. Similarly with a trace, however, the amount of compensation code due to tail duplication can be non-negligible.

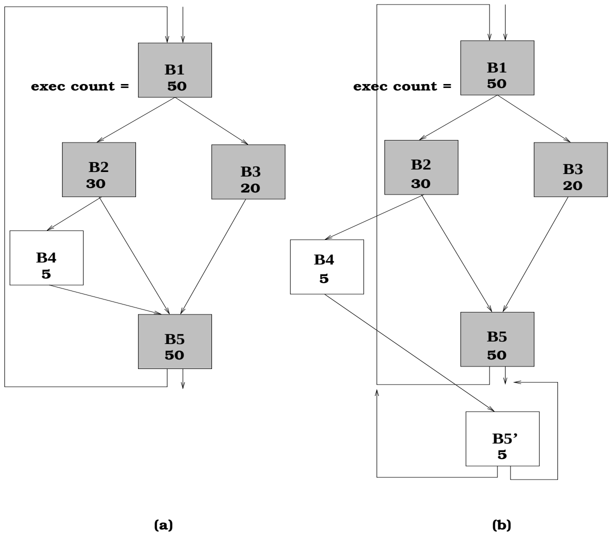


Figure 1.6: Hyperblock example, (a) after basic blocks selection, (b) after tail duplication.

1.2.3 Hyperblock scheduling

Hyperblock [68] is an alternative cross-block scheduling approach to trace and superblock. It improves on these latter techniques by providing compilers with a large scheduling scope that can span over multiple control flow paths. The formation process involves to select a set of basic block candidates from which only one basic block is designated as the entry block. Control flow may then reach the hyperblock region via this point only. It may however leave it from several other locations in the hyperblock region. The selected basic blocks are then fused into a large predicated basic block to form a hyperblock. Following is a brief description of the main steps involved during this process.

Basic block candidates for inclusion in a hyperblock are selected by means of profiling. Priority is given to loop regions with heavily executed basic blocks. This is done in order to exclude basic blocks along the less frequently executed paths so that they do not penalize the execution of those belonging to the frequent path. In addition, only basic blocks not exceeding a given size are choosing. The idea behind that is to reduce the scarcity of processor resources upstream by avoiding the inclusion of too large basic blocks in a hyperblock. This may prevent smaller basic blocks from being penalized by the execution of larger basic blocks. A last criterion often considers the type of the instructions contained in the basic block candidate. Priority is assigned to basic blocks that do not contain hazardous instructions such as procedure calls, unresolvable memory accesses, etc. Figure 1.6(a) shows the hyperblock selection process for a small number of basic blocks. As shown in the figure, B1-3,B5 are the most executed basic blocks in the loop. Hence, they are selected as candidates for inclusion in the hyperblock. In contrast, B4 is excluded from the hyperblock because it belongs to an infrequent path

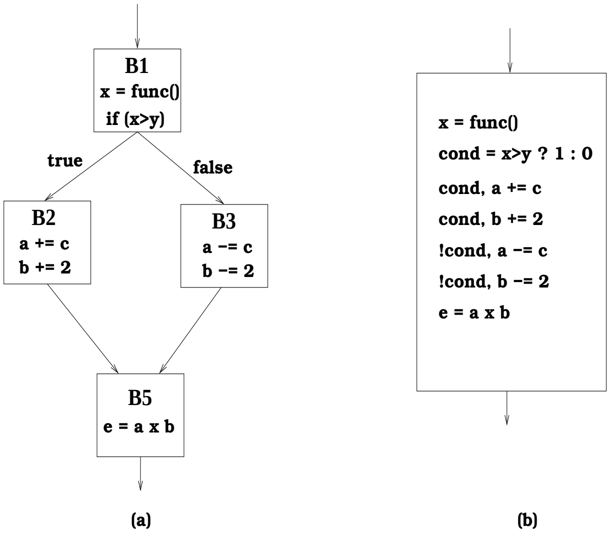


Figure 1.7: *if-conversion* example, (a) basic block candidates, (b) *if-converted* code.

in the CFG.

The formation of the hyperblock principally involves fixing problems related to side entrance and then merging the selected basic blocks into a large predicated block. Side entrance problems arise as soon as the hyperblock region can be entered without passing through the entry block. In such a case, tail duplication [28] is applied to fix this. An illustrative example is given in Figure 1.6(b), showing the duplication of basic block B5 in order to avoid the side entrance effect caused by basic block B4. Note that a drawback of this approach is the relative increase of the program code size.

The last important step involves creating the hyperblock by fusing the selected basic blocks into a large predicated basic block. Predication refers to the process of removing conditional branch instructions. A possible implementation framework for predication is *if-conversion* [2]. *if-conversion* replaces conditional branch instructions with equivalent instructions, e.g. *compare* instructions, that set some predicate registers. These predicate registers then serve as guards for conditional instructions evaluation. This requires that the ISA is augmented with instructions providing a guard register like in the Itanium [96] or in the Trimedia [34] processor, for instance. Figure 1.7 exemplifies the *if-conversion* process for the hyperblock region shown in Figure 1.6(b). The code after *if-conversion* is shown in Figure 1.7(b). Note that, the conditional branch instruction, i.e. the *if-test* in the code, has been replaced with a select instruction that sets a predicate register (register *cond*) according to the outcome of the *if-test*. The evaluations of the subsequent instructions are then conditioned by the value of the predicate register *cond*.

Because basic blocks belonging to different control flow paths can be merged together to form a single large predicated basic block, the hyperblock scheme offers a larger scheduling scope unit than that provided by either one of the trace or the superblock approach. As a result, the hyperblock potential for exploiting instructions parallelism is higher than any of these prior studied cross-block scheduling techniques.

1.3 CMOS Power consumption basics

Modern processors are implemented with CMOS (Complementary Metal-Oxide-Silicon) technology. CMOS circuits have gained their popularity principally from their low price and from their relative good power/performance tradeoff. However, with continuing technology scaling, it is very uncertain whether this power/performance tradeoff will be preserved. With the high integration density, more and more transistors are integrated into a chip, making CMOS designs increasingly complicated. This has been brought to such an extent that the power consumption of CMOS devices has raised significantly, impacting on the overall system power consumption. This section explains the fundamentals of power consumption CMOS circuits. We first introduce the main sources of power consumption in a CMOS device. Then, some metrics relative to power consumption that we will be using throughout this thesis are discussed. Lastly, we introduce some power consumption evaluation models currently available in the literature.

1.3.1 Sources of power consumption

There are three main contributors to the power consumption of a CMOS device: the dynamic power consumption, the static power consumption and the power dissipated due to the short-circuit effects. Because of the negligible impact of the power consumption due to short-circuit effects, we will only limit us to study the power consumption due to the first two components.

1.3.1.1 Dynamic power consumption

The dynamic power consumption is the primary contributor to the power consumption of a CMOS gate. It is estimated to represent as much as 90% of the total power for current process technology. Two events may lead a CMOS gate to dissipate dynamic power. These events are associated with the charging and the discharging of the load capacitance of a CMOS gate. Both of these events take place upon a transition operated by the output node voltage of the CMOS gate.

During a charging operation, the input node voltage of the CMOS gate operates a 1 to 0 transition. As a result, the load capacitance of the output node is charged up with a charging current $I_{charging}$, as indicated in Figure 1.8. This operation draws an energy from the power supply voltage that is proportional to the square of the power supply voltage and the capacitative load of the gate output's node, as indicated in Equation (1.1).

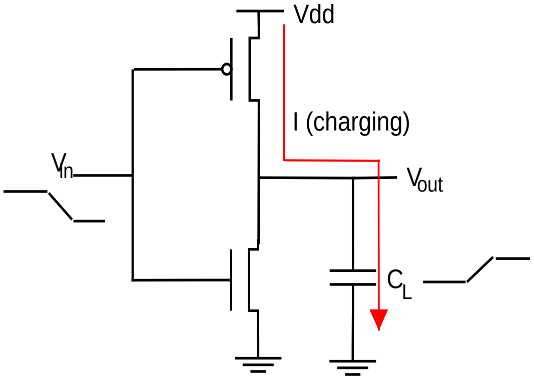


Figure 1.8: Charging operation.

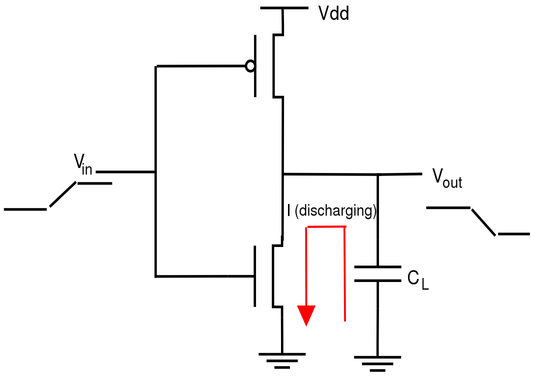


Figure 1.9: Discharging operation.

$$E = C_L * V_{dd}^2 \tag{1.1}$$

On the other hand, a 0 to 1 transition is operated upon a discharging operation occuring at the input node voltage of the CMOS gate. This causes a discharging current, $I_{discharging}$, to dissipate the energy stored in the load capacitance of the output node, as indicated in Figure 1.9. This operation does consequently incurs no energy drawn from the power supply voltage. Therefore, given that a CMOS gate can transition up to *a* times during a clock cycle, the average energy consumed per cycle can be expressed as shown in Equation (1.2).

$$E_{cycle} = \frac{1}{2} * a * C_L * V_{dd}^2 \tag{1.2}$$

The power dissipation of a CMOS gate is usually defined as the rate at which the energy is transformed into heat. Hence, assuming that the CMOS gate operates with a clock frequency *f*, the average power dissipated by such a node is given by Equation (1.3).

$$\begin{aligned} P_{dynamic} &= E_{cycle} * f \\ &= \frac{1}{2} * C_L * V_{dd}^2 * a * f \end{aligned} \tag{1.3}$$

It follows immediately from Equation (1.3) that the dynamic power consumption of a CMOS gate is independent of a transistor characteristics and sizes. What is determinant for a power consuming transition to take place at the output node of a CMOS gate are the values at the input node voltage and the rate at which these values are changing. Hence, the dynamic power consumption is fundamentally dependent on a program input

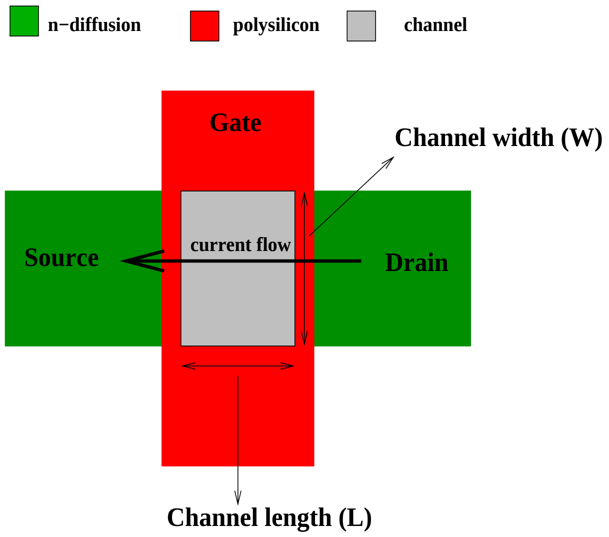


Figure 1.10: n-MOS transistor layout.

data pattern, and can therefore be managed at the software level, as we will demonstrate later in this thesis.

1.3.1.2 Static power consumption

CMOS transistor basics Let us first proceed to a description of the basic functioning of a CMOS transistor before going more deeply into the root causes of static power dissipation.

A CMOS gate, as that shown in Figure 1.8 or Figure 1.9, is composed of a combination of n-channel (lower part) and/or p-channel (upper part) transistors. Figure 1.10 shows the layout of such an n-channel transistor. In general, either n-channel or p-channel transistor is composed of three main terminals, as depicted in Figure 1.10. These are the drain, the gate, and the source.

Conduction usually occurs between the *drain* and the *source* terminals via a *channel*. This conduction is controlled by applying a base voltage at the *gate* terminal which then causes some charge carriers to be attracted at the other terminal end, thereby enabling current flow between the *drain* and the *source* terminal. This current flow is made possible by isolating the *gate* terminal from the conducting *channel* by a thin layer of gate oxide, as shown in the figure. Applying a voltage at the *gate* produces an electrical field that then influences the current flow. Note that, depending on the electric field that is applied on the *gate*, the direction of the current flow can be reversed, e.g. from *source* to *drain*.

Generally, the voltage that is applied at any terminal is measured relative to the *source* terminal. We call the threshold voltage, denoted by V_{th}, the voltage applied at

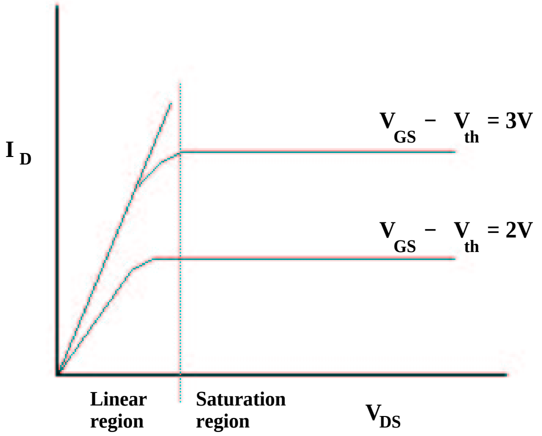


Figure 1.11: I_D as a function of V_{GS}, V_{th} and V_{DS}.

the *gate-source* terminal, denoted by V_{GS}, at which current begins to flow between the *drain* and the *source*. This current is called the *drain* current, denoted by I_D, and is strongly dependent on the quantity $V_{GS} - V_{th}$, i.e. the amount by which the *gate-source* voltage exceeds the threshold voltage, as shown in Figure 1.11. Since the *gate* terminal is assumed to be isolated from the rest of the device, once the gate has been brought to one of its polarities, i.e. charged or discharged, no current is supposed to flow between the *drain* and the *source* terminal, meanwhile; hence, no power is consumed, ideally.

Effects of subthreshold leakage Subthreshold leakage effects arise for small values of the drain current I_D, i.e. values induced by channel voltages which are below the threshold voltage required for conduction. Under this voltage threshold, a current still leaks through the transistor, between the *drain* and the *source*, although the transistor is turned off. This leakage current draws a static power that is proportional to the power supply voltage V_{dd}, as indicated in the Equation (1.4).

$$P_{static} = V_{dd} * I_{leak} \tag{1.4}$$

The leakage current I_{leak} is closely related to some transistor characteristics such as its width, its length, the device operating temperature, and many other parameters. However, the most important property of the leakage current is given by its relationship to the threshold voltage V_{th}, as it is expressed in Equation (1.5). In this equation, W and L are the transistor width and length, I_S a constant current and V_T the thermal voltage [47].

$$I_{leak} \sim \frac{W}{L} * I_S * e^{\frac{-V_{th}}{a*V_T}} \tag{1.5}$$

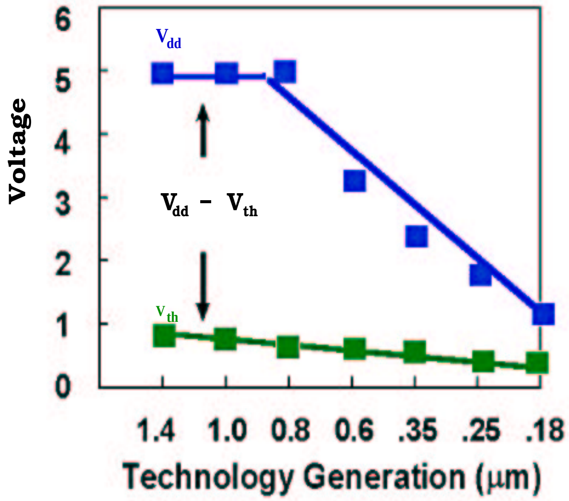


Figure 1.12: Trend in power supply and threshold voltage scaling.

It can be seen from Equation (1.5) that the leakage current grows exponentially with decreasing threshold voltage. This has a drastic consequence on power consumption since the static power increases linearly with I_{leak}, as shown in Equation (1.4). The continuous reduction of the threshold voltage is to be put on the account of the process scaling which is at the forefront of the Moore's law. In fact, as the technology continues to shrink, the power supply voltage must also be reduced in order to maintain the dynamic power consumption within an acceptable level. It is this reduction of the power supply voltage that motivated designers to lower the threshold voltage in order to provide sufficient noise margins to ensure a reliable functioning of the CMOS device. This effect can be clearly illustrated in Figure 1.12, where we plotted trends for the power supply and the threshold voltage scaling[1]. In Figure 1.13, we also have indicated current trend for the dynamic and the static power consumption as technology is scaled down to 0.18 μm. As shown in the figure, the trend for the static power consumption is closely nearing that of the dynamic power as we move toward narrower process sizes. It is even expected that below 0.07 μm, the leakage power consumption will represent as much as 50% of the total chip power consumption [99].

1.3.2 Power metrics

Many metrics have been proposed in the literature [19] that serve as a measure of the power-efficiency of a system. Choosing the right one that is best appropriated to a particular purpose is of a crucial importance since each metric is usually associated with a precise meaning regarding the energy, the power and/or the performance.

When deciding among two alternative optimizations, that which dissipates the less

[1] The values in Figure 1.12 and Figure 1.13 are adapted from [106]

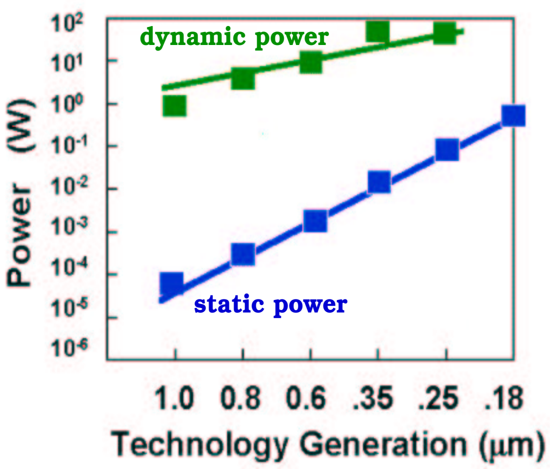


Figure 1.13: Dynamic power and static power consumption trends.

energy, power is usually the best available metric. Power is defined in Watts, and it measures the rate at which energy is dissipated. Power is often used in various circumstances to describe different things. The peak power, for instance, is essential when reliability or packaging cost is of concern. This is dictated by the fact that a high peak power may reduce the lifetime of processor components and cause failures. The average power, on the other hand, is a metric that is often used to measure the average energy required to execute a task over a given time. It is useful for estimating battery life, for example. These power metrics can be however very misleading if performance has to be taken into account. The main reason is that power rises linearly with the clock frequency. Slowing down the clock speed may then reduce power, but it might also impair performance. In this sense, power is usually not effective for measuring the energy-efficiency.

The energy per operation, $\frac{E}{op}$, is another metric of the power-efficiency. It defines the energy required to execute an operation and it is measured in $\frac{J}{op}$. The term operation may take various appellation, ranging from an instruction to a complete program run. Relative to power, this metric can also be expressed in terms of $CPI * W$. Because of this latter expression, this metric is also called the power-delay product. Since power and performance are bound to one common expression, this metric is very effective to measure power-efficiency. It has however some limitations. Namely, it cannot be used as a measure of goodness of power-efficiency whenever voltage scaling is applied. This is mainly because, since the clock frequency is approximately linear in V_{dd}, as indicated in Equation (11), a reduction of the latter may slow down program as a consequence of lowering the former. Hence, this metric is not adapted for designs that use a form of voltage scaling. On the other hand, however, this metric is very appropriate for cases where frequency or activity variations are considered as levers of power reduction.

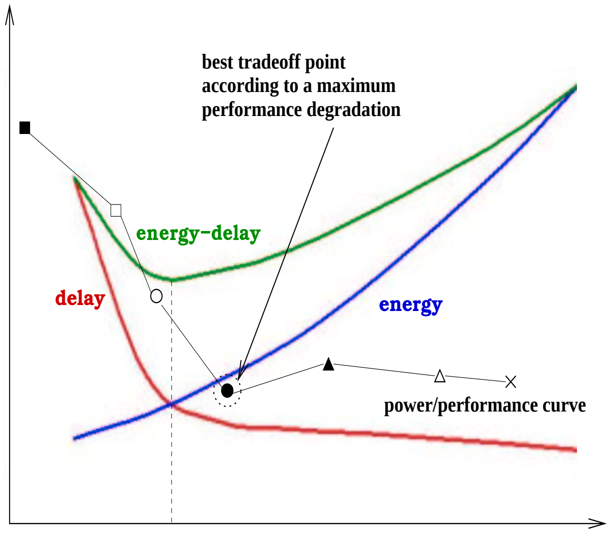


Figure 1.14: Power/performance tradeoff.

Another approach to consider power and performance simultaneously is to have these values plotted into one curve in a plan. Potentially, there are many ways of realizing this. One such alternative is to simply consider a given performance threshold, and then selecting a corresponding point in the plan with power estimates that best meet the performance constraints. We have illustrated this in Figure 1.14. Among the various points of the power/performance curve which represent each one an alternative design choice, that which is highlighted in the curve corresponds to the best tradeoff point that meets a given performance constraint. Another alternative consists of considering the product of the energy and the performance [40], as it is shown in Figure 1.14 under the name *energy-delay*. This last metric is practical for designs that consider varying power supply voltages since the effect of scaling the voltage on the performance can be easily monitored by the product of the energy savings and the obtained performance gain.

Depending on which parameter is subject to change, e.g. activity variation, frequency or voltage scaling, we will always consider one of the last three metrics as a measure of the power-efficiency throughout this thesis.

1.3.3 Power modeling and evaluation

Power can be addressed at various abstraction levels of the design hierarchy, including the circuit level, the transistor level, the architecture level and the system level, as indicated in Figure 1.15. At each abstraction level, there have been numerous works that have tackled the problem of estimating the power consumption of a system. In general, the lower is the level at which the power analysis is performed, more accurate are the obtained results. In contrast, a higher energy saving is expected to be gained from a top down analysis approach. In this section, we only consider analyzing power

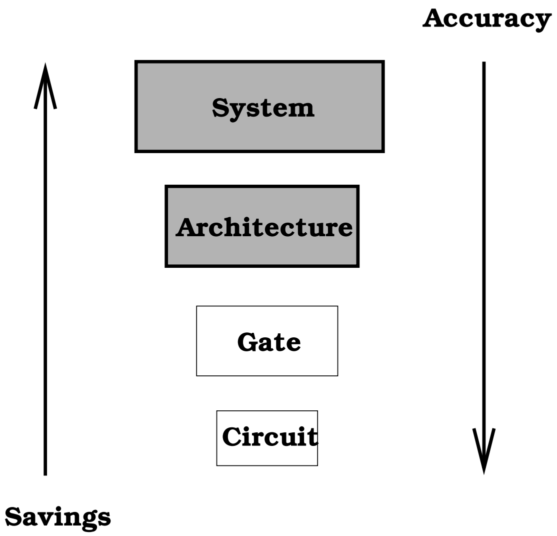


Figure 1.15: Power analysis levels.

consumption at the upper two levels of the design hierarchy shown in Figure 1.15. We do not particularly take care about the energy results accuracy that one can have at these levels. What matters, according to us, is that power-efficient decisions can be easily taken when choosing among alternative implementations.

1.3.3.1 System-level

In order to estimate power at the system-level, a model for the power consumption of a program must be provided. One such model is given by an instruction-based power modeling approach [108, 93, 14, 63]. Basically, instruction-level power models rely on the fact that the power consumption of a program can be computed as the sum of the energy dissipated by each instruction of the program as it proceeds through execution. To do so, a processor power model associates to each ISA instruction an empirical energy cost. This cost can be determined in one of several ways, among which are direct current measurements, HDL simulation at the granularity of functional blocks, or gate level simulation. When attributing a per-instruction energy cost, some important considerations must be taken into account. First, there is usually a per-instruction energy cost which is independent of an instruction execution context. This *energy base cost* accounts for the processor overhead and therefore excludes any energy cost associated to a stall in the pipeline or a data or instruction cache miss. Last, a per-instruction energy cost can also strongly depend on a prior processor state which has resulted from a prior instruction execution context. This inter-instruction effect, also called *circuit state effect* following the terminology adopted by Tiwari et al. [108], can model various things such as the switching activities on the bus and on the control lines for example. Some instruction-based power models also extend their scope to provide power estimations of the whole system by including base energy cost for accessing off-chip components such as caches. This is for instance the case of the instruction-based power model proposed in [14].

1.3.3.2 Architecture-level

The impact of applying a given optimization on the energy can also be evaluated at the compiler level by means of relying on detailed architectural power models. This may help providing in-depth understanding of the impact of a given optimization on the various components of a processor system.

Processor core Micro-architectural power models are largely dependent on processor architectural simulators since they rely on these latter to provide overall power consumption estimates. Basically, an architectural processor simulator provides detailed analysis of the different processor components that are accessed during a cycle. This information can then be incorporated into a power model to provide a per-cycle power estimates of the system. The precision of the power estimates depend on the analysis details of the processor simulator as well as on the power model used to describe the different processor structures. As for example, estimating the dynamic power cost associated with a given processor unit accessed during a cycle is usually found with Equation (1.3). Whereas V_{dd} and f can be determined relatively easily assuming a given technology parameter, estimating the values of C for each of the different processor components is very complicated since this may depend on various parameters such as circuit complexity, transistor sizings, etc. This makes the power evaluation very dependent on the level of analysis used to describe each processor constituent. Examples of micro-architectural power simulators that use this approach are Wattch [20] and SimplePower [112]. These two simulators are based on SimpleScalar [22], which is an architectural processor simulator that keeps track of the different units that are accessed during each execution cycle.

Memory components Memory components (e.g. DRAM, caches, registers, or buffers) represent a significant fraction of the power consumption of a processor system. For this reason, many studies have devoted them much attention in the past decade. The approach for modeling power on these components proceeds similarly to that used for a common processor structure. Based on a given memory configuration, an architectural simulator provides detailed timing analysis for the different memory constituents (e.g. SRAM cells, wordlines, bitlines, comparators, sense amplifiers, decoders, etc) that are exercised as an access is made to the component. A power model can then be associated with each of the different memory constituents to evaluate the power consumption of the accessed memory component. As previously stated, the accuracy of the power results depends on the analysis details used to model the different values of the capacitative load C, as illustrated in Equation (1.3). The CACTI tool [98] implements such an approach and provides detailed timing, power and area estimates for parameterizable cache memory structures. It can also be modified to model register files as well as buffers or queues.

So far, we have only considered modeling the dynamic power consumption of processor or memory structures. This is warranted since dynamic power consumption contributes for the largest fraction of the total power consumption, at least for current technology generation. However, this is not guaranteed to hold for future technology generations, where the gap between the dynamic and the static energy is becoming narrower, as we have already noted it in Section 1.3.1.2. Therefore, modeling the impact of the static power consumption has since evolved as a hot topic of research. In this sense, a recent proposal by Butts et al. [23] evaluates the static energy at the architectural level by capturing the effects of technology and design parameters into a relatively simple formula as shown in Equation (1.6).

$$P_{static} = V_{dd} * N * k_{design} * I_{leak} \tag{1.6}$$

In this equation, N denotes the number of transistors in the modeled circuits, whereas k_{design} is a circuit dependent parameter that captures effects such as transistor sizings, number of switched off transistors, etc. Recently, [118] has proposed an evaluation tool, called Hotleakage, that built on an improved version of the static power model of [23]. This evaluation tool is interfaced with SimpleScalar to provide static power estimates for various processor components. It also integrates CACTI to provide static power estimates for the various cache constituents. As such, it can be easily extended to feature register files as well as other memory components.

1.4 Power-efficient compilation opportunities

We are now better armed for comprehending in more precise terms the thorny problem of power reduction of modern processors. From what has preceded in Section 1.3, the power consumption of a processor is amenable to two principal components. Namely, the dynamic and the static power consumption, respectively modeled by Equation (1.3) and Equation (1.4). Let us combine these two equations into a single one to provide a general expression for the power consumption, as shown in Equation (1.7).

$$P_{total} = \frac{1}{2} * C_L * V_{dd}^2 * a * f + V_{dd} * I_{leak} \tag{1.7}$$

The activity factor a and the frequency f can be combined together to provide an effective activity factor $\alpha = a * f$. In this case, α models the effective switching frequency, i.e. up to a number of switching transitions occur at a data rate of f. This is the effective rate at which dynamic power is dissipated. A new expression for P_{total} can then be given by Equation (1.8).

$$P_{total} = \frac{1}{2} * C_L * V_{dd}^2 * \alpha + V_{dd} * I_{leak} \tag{1.8}$$

We have stated that the effective activity factor α dictates the rate at which dynamic power is dissipated. Recall again from Section 1.3.1.1 that dynamic power is consumed principally as a result of a charging/discharging operation that takes place at the output node capacitance C_L. Hence, in absence of a charging/discharging of the capacitative load, no dynamic power is supposed to be dissipated. Therefore, we can define the effective capacitance C_{eff} as the average capacitative load that is charged/discharged during each data rate *f*, yielding

$$C_{eff} = \frac{1}{2} * C_L * \alpha \tag{1.9}$$

The new expression for P_{total} can then be rewritten as

$$P_{total} = C_{eff} * V_{dd}^2 + V_{dd} * I_{leak} \tag{1.10}$$

Viewed from a compiler standpoint, it follows from Equation (1.10) that the different power levers on which one can act in order to obtain an energy savings reduce to C_{eff} and V_{dd}. This means that either one of the voltage or the effective capacitance scaling techniques will be effective in achieving some energy savings.

Voltage scaling goes against the performance objectives followed in this thesis as far as dynamic power consumption is concerned. Since we are looking for power-efficient solutions that tend to preserve performance, we will then favor effective capacitance scaling techniques to attack the problem of reducing dynamic power. This leads us at least to one practical evidence. Namely that activity reduction techniques have a significant potential for effectively reducing dynamic power consumption. The primary goal of this thesis is therefore to focus on compiler-controlled mechanisms that can achieve a high level of activity reduction, e.g. compiler-controlled processor reconfiguration or adaptation. Since this activity reduction directly turns out into dynamic power reduction, we expect to gain some energy savings while still providing acceptable performance levels. The range of techniques addressing this reduction of activity is very large. We will thus tackle only a small subset of them throughout this thesis. In particular, we will lay emphasis on the processor core (Chapter 2), the cache subsystem (Chapter 4), and the processor datapath (Chapter 5).

On the other hand, voltage scaling accommodates well to the objectives of static power reduction when regarding performance. In fact, since static power is dissipated even when no activity is observed, we can apply any of the voltage scaling techniques on the unused portions of the chip to reduce static power; thereby preserving performance. Therefore, keeping in mind this fact, we will selectively conjugate activity reduction techniques with voltage scaling techniques to achieve reduction of both the dynamic and the static power. This is addressed in the Chapter 4 and the Chapter 5, when looking at the cache subsystem and the processor datapath, respectively.

1.5 Summary

This chapter has introduced some fundamentals of VLIW machines. We have described the architecture of VLIW machines and explored techniques usually deployed to exploit instructions parallelism at compilation time. A deep understanding of power consumption has also been provided to acknowledge the problem related to the power dissipation in VLIW processors. We have motivated the objectives of this thesis, which are located around the power-efficiency problematic, and provided some research directions that we will look into in the remainder of this thesis. Notably, we have highlighted two main directions associated with the reduction of switching activity and voltage scaling. The next chapters will then address each of these issues in a more precise manner.

Chapter 2

Low-power ILP compilation issues

Modern processors rely heavily on instruction level parallelism (ILP) techniques to achieve high performance. With statically scheduled VLIW processors, this is capitalized by the duplication of functional units that provide support for the concurrent execution of multiple instructions. On these machines however, the main challenge essentially lies on the compiler which is in charge of finding enough instructions to fully utilize the underlying hardware parallelism. For this purpose, many optimizing VLIW compilers are built with sophisticated ILP transformation techniques such as superblocks [49], trace scheduling [36], and hyperblocks [68].

These techniques are based on the principle that a larger scheduling scope unit is formed as the result of merging several basic blocks into a single one. While these techniques undoubtedly improve ILP, their impact on the energy consumption is mitigated by the increase in the total instruction count they usually cause. On embedded systems where a large portion of the dissipated energy emanates from unning programs [107], an increase in the energy consumption can have a dramatic impact on the system energy efficiency.

Common approaches used to tackle these issues at the software level rely heavily on performance optimization, following an unstated rule that energy consumption is roughly proportional to the total execution time. This however neglects the effects of computational and architectural overhead, which mainly result from wasted computation and the diminishing performance returns of incrementally applying some optimizations.

This chapter explores the energy-delay tradeoff of applying ILP optimization techniques on a statically scheduled VLIW processor. Our goal is to develop a theoretical understanding of the main energy issues involved in ILP transformation techniques. To this aim, we developed an analytical energy-efficiency model to investigate the issues between energy and performance. The model capitalizes on the fact that monitoring the variations in program performance can be achieved on many modern processors through some form of prediction mechanism or profiling at the software level. We exploit this approach to show that there exists a threshold above which optimizing for ILP may turn into diminishing energy reduction returns. We translated these results into heuristics to measure the energy-efficiency of ILP transformations on a set of embedded applications.

The remainder of this chapter is organized as follows. In Section 2, we define the metric we used throughout this study. In Section 3, the energy model of a VLIW-core processor is introduced, followed by the analytical model for the energy-efficiency. In Section 4, an hyperblock formation case study is presented and evaluated. Section 5 discusses previous work and Section 6 concludes.

2.1 Power/Performance evaluation metric

We consider the energy-delay product [40] to define another equivalent metric; the ratio performance-to-energy (PTE) that we use to compare two different instances of an application at the software level. In comparing instances of the same application, the PTE ratio attempts to measure the need of increase performance against low energy. The PTE ratio is expressed as the inverse of the energy-delay product as shown in (1).

$$PTE = \frac{1}{E_{op} \times Cycle_{op}} = \frac{Performance}{Energy} \tag{2.1}$$

The advantage of the PTE ratio expressed above is that, given an energy budget, one can look at different performance values that potentially improve the PTE ratio without worsening the energy. In this way, we can focus on the range of values that optimize the energy-delay product with a given target. This approach can serve compilers since it does not rely on special hardware support. We only need to track the changes in processor performance. This however can be mastered in a relatively easy manner at the software level.

2.2 Energy-efficiency analysis

This section first introduces the VLIW-core energy model that serves us as a foundation for the energy-delay tradeoff exploration. Then, at the end of this section, we discuss the theoretical model developed to allow compile-time energy-delay evaluation of the applied ILP transformation.

2.2.1 Energy model

We use a VLIW-core energy model [14] that features an embedded VLIW processor capable of issuing up to N parallel operations in a very long instruction called a bundle. In this model, the energy, EPB, dissipated by the execution of a bundle w_n is modeled as follows:

$$EPB_{w_n} = E_c + IPC_{w_n} \cdot E_{op} + m \cdot p \cdot E_s + l \cdot q \cdot E_{miss} \tag{2.2}$$

In (2.2), E_c refers to a constant average energy base cost, IPC_{w_n} is the number of operations different from NOP in the bundle, and E_{op} is the average energy consumption associated with the execution of an operation. Note that, in contrast to superscalar processors, a VLIW processor does not provide aggressive architectural features [80] such as rename and wakeup logic, which otherwise should have largely contributed to increase the energy consumption per operation. In such a case, we could expect the average energy consumed per operation (E_{op}) in a VLIW processor to be roughly the same for almost all operations, not counting operations involving a memory access. This is however not exactly true, since the circuit state effect [107] due to the switching activity on the bus may make the energy base cost of one instruction higher than expected.

The third expression in (2.2) refers to the additive energy consumption due to a miss event on the D-cache, where m is the average number of additional stall cycles per bundle due to a D-cache miss, p the probability that a D-cache miss occurs, and E_s the core energy consumption during a pipeline stall. The fourth expression is related to the I-cache misses, where l is the average number of additional NOP operations per bundle introduced during a I-cache miss, q the probability per bundle that this event occurs, and E_{miss} the energy consumption of the core during an I-cache miss.

Some dynamic program behaviors can not be easily captured at compile time. This includes for instance the data or instruction cache miss count. In order to simplify the above model without loosing to much accuracy, we restricted the study on computation-intensive program regions, e.g loops. The above model can therefore be simplified, namely by neglecting the impact due to the I-cache misses. This consideration is indeed true as long as the undertaken ILP transformations do not increase the size of the loop region in such a way that it overwhelms the I-cache. The energy model described in (2.2) can then be rewritten as follows:

$$EPB_{w_n} = E_c + IPC_{w_n} \cdot E_{op} + m \cdot p \cdot E_s \tag{2.3}$$

We also assume for this study that the presumed machine model does no prefetching. This is necessary to exclude the unpredictable effects of prefetching on the data cache which can impact performance and energy, thereby biasing the analysis. On some machine, a compiler option is sometimes provided to disable prefetching. We assume this is the case for the rest of this study.

2.2.2 Cost model

In order to estimate the energy consumption impact for an ILP transformation at the compilation level, we express the PTE ratio as a function of both the energy and the performance. For the rest of this study, we consider the smallest scheduling scope granularity to be the basic block. We then obtain an expression for PTE as shown in (4).

$$PTE = \frac{1}{E_{BB} \times Cycle_{BB}} = \frac{\overline{IPC}}{N \times E_{BB}} \tag{2.4}$$

N and $\overline{IPC}$ represent the number of operations contained in the basic block and the average number of operations executed in each bundle respectively. The basic block energy term E_{BB} in (4) can be decomposed as the sum of the energy required to execute an instruction stream of n consecutive bundles. This can formally be expressed as follows:

$$\begin{aligned} E_{BB} &= f \cdot \sum_{i=1}^{n} EPB_i \\ &= f \cdot n \cdot (E_c + E_{op} \cdot \overline{IPC}) + \overline{s} \cdot E_s \end{aligned} \tag{2.5}$$

In (5), $\overline{s}$ is the average number of stall cycles due to data cache miss in the basic block, whereas f represents the basic block execution frequency. If we consider a coarser granularity, the energy consumption of a region R composed of m basic blocks can be modeled in a similar manner, as shown in (6).

$$\begin{aligned} E_R &= \sum_{i=1}^{m} E_{BB_i} \\ &= m \cdot \overline{f_R} \cdot \overline{n_R} \cdot (E_c + E_{op} \cdot \overline{IPC_R}) + \overline{s_R} \cdot E_s \end{aligned} \tag{2.6}$$

In (6), $\overline{f_R}$ and $\overline{n_R}$ are the average execution frequency and the average number of bundles of the region R respectively.

Recall that the rationale behind the PTE ratio is to lay emphasis on the range of performance values that improve the energy efficiency. This is necessary to compare program instances corresponding to the states before and after an ILP transformation. Therefore, the IPC term of the PTE ratio can be viewed as a kind of gear mechanism for energy regulation purpose. In this way, we can keep track of the IPC values that improve the energy efficiency and then only select those candidate regions that lead to this IPC improvement. Then, given a candidate region R composed of m basic blocks, we call a particular ILP transformation function, F_{ilp}, energy efficient if the PTE ratio of the resulting ILP block H is such that $PTE_H > PTE_R$. The inequality can be solved at $\overline{IPC_H}$ to yield a new inequality shown in (7).

$$\overline{IPC_H} > F_{ilp}(\overline{IPC_R}) \tag{2.7}$$

The expression for F_{ilp} is equivalent to

$$F_{ilp}(\overline{IPC_R}) = \frac{A \cdot \overline{IPC_R}}{B + C \cdot \overline{IPC_R}} \tag{2.8}$$

with

$$\begin{cases} A = f_H \cdot N_H \cdot n_H \cdot E_c + N_H \cdot \overline{s_H} \cdot E_s \\ B = m \cdot N_R \cdot \overline{f_R} \cdot \overline{n_R} \cdot E_c + N_R \cdot \overline{s_R} \cdot E_s \\ C = (m \cdot N_R \cdot \overline{f_R} \cdot \overline{n_R} - f_H \cdot N_H \cdot n_H) \cdot E_{op} \end{cases} \tag{2.9}$$

The process of evaluating the above transformation function requires to characterize the candidate region R as well as the target ILP block H as indicated by the substitution variables B and A of the function F_{ilp}. This latter variable is however not obvious to determine, since it requires that we know in advance what the target ILP block H will look like. This is mainly due to the fact that several transformations may be applied on the resulting ILP block as a side-effect of some ILP-based optimizations, causing the number of bundles and executed operations to vary. At this stage, we therefore anticipated the formation of the ILP block by integrating a region ahead scheduling pass to predict the characteristics of the target ILP block, yielding a rough indication of the ILP block parameters.

2.2.3 Tradeoff analysis

We have studied the energy-delay tradeoff in terms of the amount of ILP needed to compensate for an increase in the energy dissipation. The increase in energy is measured as the additional overhead and/or wasted energy. By wasted energy we mean essentially the energy dissipated due to the execution of needless operations. Such operations result from if-converting basic blocks along the taken and the not-taken paths of a branch instruction. These operations consume extra resources since the processor still have to fetch, decode and execute them, although no machine state change is guaranteed to occur. In the ILP transformation function F_{ilp}, the wasted energy can be termed by the variable C. As one can observe, the wasted energy is proportional to the product of the average energy per operation and the dynamic operation count difference between the candidate region and the target ILP block. This operation count difference constitutes the extra operations executed by the processor. We can analyze the effect due to the execution of these extra-instructions under 3 different aspects.

Case $C > 0$. In this case, we can expect the amount of wasted energy to be negligible, because the executed extra work is marginal. Obviously, this case corresponds to the optimal execution scenario. Typically, the condition $C > 0$ is true when the value of the ILP block term is quantitatively less than that of its corresponding original schedule. This may be primarily a consequence of the fact that applying some types of optimization on the resulted ILP block may produce more impact in terms of reduction of the number of operations N_H and static scheduled length n_H compared to

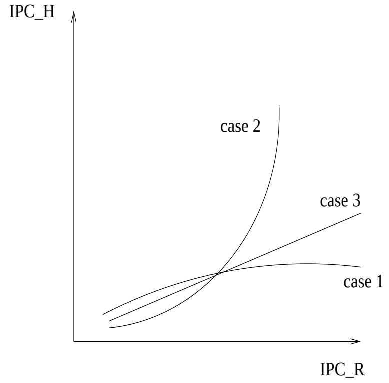


Figure 2.1: Shape of the curves corresponding to the cases $C < 0$, $C = 0$, and $C > 0$ of the inequality (7) for various values of IPC_H and IPC_R.

its original schedule. Examples of such optimization may include instruction merging and instruction renaming which respectively reduces the number of operations N_H and diminishes the impact of dependence height by allowing instructions to be scheduled earlier, thereby reducing the static scheduled length n_H. The class of energy-efficient solutions for which the transformed block yields the best energy-delay tradeoff lies above the curve labeled $case1$ in Figure 2.1. In this figure, it can be seen that the curve has a logarithmic shape. This indicates that the probability to get some values of $\overline{IPC_R}$ that will feet well to the inequality (7), providing lower values for N_H and n_H, is high.

Case $C = 0$. In some extent, this case can be assumed to be similar to the previous one because there is no additional extra computation than what is required for the normal computation. However, at equal schedule length and/or operation count, deciding on the energy-efficiency of the transformation is not as straightforward as it seems to be. Some energy side effects may render the ILP block less energy-efficient than its original schedule. A typical situation is when the circuit effect in the transformed ILP block is no longer negligible. This may be principally due to the inter-instruction effects in the new resulting schedule. If the compiler can evaluate this impact, the decision can be made simpler. If we consider this circuit effect to represent a given fraction of the ILP block energy, we can then expect the set of solutions for which the ILP transformation function yields a good energy-delay tradeoff to be at a corresponding distance above the curve labeled *case 3* shown in Figure 2.1.

Case $C < 0$. In this case, the amount of wasted energy is no longer negligible because the extra computation cost start to be the dominant factor. In the variable C, this is reflected by the fact that the value of the ILP block term is quantitatively higher than that of its corresponding original schedule. In another words, it means that the values of N_H and n_H increase non-linearly with respect to their corresponding values in the original schedule. There are many scenarios which can potentially lead to degrade the values of N_H and/or n_H. Consider the case of a dependence height mismatch. The static scheduled length n_H may increase as a result of lengthening the execution time of the taken path, thereby offsetting the ILP benefit by degrading the average energy consumption of that path. In a intensively executed loop region, this may not be negligible at all. Consider again a machine with scarce resources, e.g one memory port as in our machine model. The resulting schedule length n_H may also increase if resource contention becomes an important issue when paths are overlapped. On the other hand, the increase of the instruction count N_H is more concerned with the amount of compensation code that is introduced in order to repair the effect of the elimination of branch instructions. These effects, combined with the fact that applications usually show not enough available parallelism [79], can rather turn the ILP benefit into additive computation cost and therefore increase energy consumption. This scenario is illustrated in Figure 2.1, in the curve labeled *case 2*. The exponential shape of the curve demonstrates that the range of energy-efficient solutions which satisfies the inequality (7) is quite small because IPC_H grows exponentially with increasing IPC_R. This means that, for an ILP transformation to be energy-efficient given a value of $\overline{IPC_R}$, the transformation function should yield a value for $\overline{IPC_H}$ which is above the aforementioned curve.

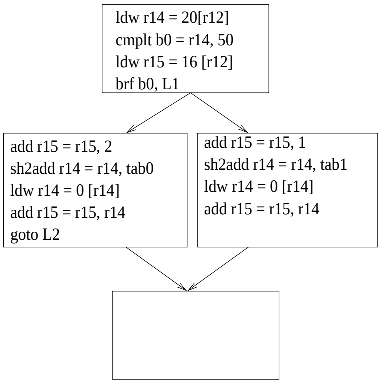
The energy overhead is principally due to the processor core activity, independently of the execution context of a bundle. It is therefore directly related to the architectural implementation cost of the processor. Because of its tight relation with the processor implementation, the energy overhead is difficult to evaluate without a detailed architectural-level simulator. We can however have a theoretic rough estimate of its impact if we again consider the transformation function F_{ilp}. Under optimal data cache conditions, the curve of the ILP transformation function has an asymptote which is proportional to the ratio $\frac{E_c}{E_{op}}$, as shown in (10).

$$\lim_{\overline{IPC_R} \to \infty} \left(\frac{A \cdot \overline{IPC_R}}{B + C \cdot \overline{IPC_R}}\right) = k \cdot \frac{E_c}{E_{op}} \qquad (2.10)$$

If this ratio is made too large (higher E_c values), then the function quickly converges near the asymptote as IPC_R increases. Therefore, the range of energy-efficient solutions that satisfies the inequality (7) (values above the curve) is also expected to narrow as we gradually move towards larger IPC values. Smaller values for the ratio are therefore preferable in order to keep the range of feasible energy-efficient solutions within reasonable IPC values.

2.3 Case study on hyperblocks

This section analyzes the energy issues involved by constructing hyperblocks. We first introduce the hyperblock framework model used throughout this chapter. Then, based on the previously discussed energy model, a set of heuristics is proposed to drive the formation of energy efficient hyperblocks. Finally, an evaluation of the proposed heuristics is provided at the end of this section.

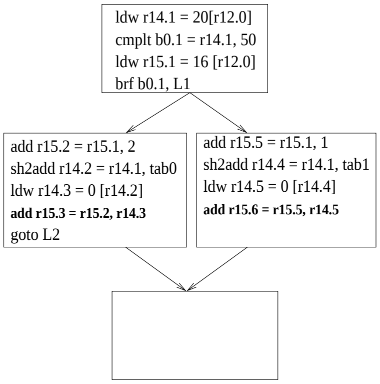


```
ldw r14 = 20[r12]
cmplt b0 = r14, 50
ldw r15 = 16 [r12]
brf b0, L1
```

```
add r15 = r15, 2
sh2add r14 = r14, tab0
ldw r14 = 0 [r14]
add r15 = r15, r14
goto L2
```

```
add r15 = r15, 1
sh2add r14 = r14, tab1
ldw r14 = 0 [r14]
add r15 = r15, r14
```

Figure 2.2: Original CFG.

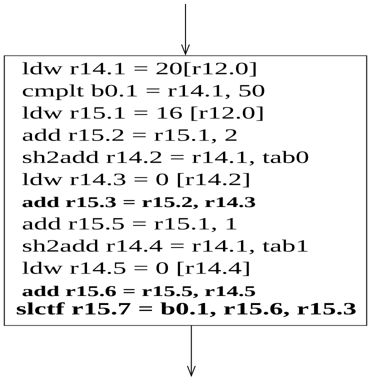


```
ldw r14.1 = 20[r12.0]
cmplt b0.1 = r14.1, 50
ldw r15.1 = 16 [r12.0]
brf b0.1, L1
```

```
add r15.2 = r15.1, 2
sh2add r14.2 = r14.1, tab0
ldw r14.3 = 0 [r14.2]
add r15.3 = r15.2, r14.3
goto L2
```

```
add r15.5 = r15.1, 1
sh2add r14.4 = r14.1, tab1
ldw r14.5 = 0 [r14.4]
add r15.6 = r15.5, r14.5
```

Figure 2.3: CFG in SSA.



```
ldw r14.1 = 20[r12.0]
cmplt b0.1 = r14.1, 50
ldw r15.1 = 16 [r12.0]
add r15.2 = r15.1, 2
sh2add r14.2 = r14.1, tab0
ldw r14.3 = 0 [r14.2]
add r15.3 = r15.2, r14.3
add r15.5 = r15.1, 1
sh2add r14.4 = r14.1, tab1
ldw r14.5 = 0 [r14.4]
add r15.6 = r15.5, r14.5
slctf r15.7 = b0.1, r15.6, r15.3
```

Figure 2.4: *if-converted* CFG.

2.3.1 Hyperblock framework model

We assume a guarded instruction model that relies on the use of a select instruction. The format of the select instruction is as shown below:

$$slctf\ dest = cond,\ src1,\ src2$$

The *slctf* instruction writes the value of $src1$ into *dest* if *cond* is 0, otherwise $src2$ is written into *dest* if *cond* takes the value 1. In this model, the predication process goes through 2 steps. In the first step, the candidate basic blocks are selected for *if-conversion*. In the second step, the *if-converted* region is formed by eliminating the branch instructions and then inserting the appropriate select instructions into the code whenever there are more than one definition of the same register that flow into the resulted *if-converted* region. This last step puts a strong emphasis on the register names, requiring that each register be uniquely named throughout the program. Prior to *if-conversion*, the original CFG is therefore transformed into a SSA [30] form to guarantee the uniqueness of each register name. This is illustrated in Figure 2.2-2.4.

The limited predication scheme presented above differs from the one presented in Section 1.2.3 of Chapter 1 in that we do not allow an hyperblock to contain multiple exit points or branches. Therefore, the regions to be considered for inclusion in a hyperblock consist solely of set of consecutive basic blocks or hammocks. An hyperblock in this framework is thus akin a large basic block of predicated instructions with single entry and exit points.

2.3.2 Understanding the energy issues

We consider the inequality (7) as the main heuristic for deciding whether or not to transform a set of candidate blocks into a hyperblock. In order to address the cost-effectiveness of applying such a transformation, we also consider a profit heuristic. We define the profit of performing a transformation as follows:

$$Profit = \frac{PTE_{transformed} - PTE_{original}}{PTE_{original}} \tag{2.11}$$

The profit is used to weight the gain of the transformation. To this end, the profit is compared against an arbitrary threshold value which puts a minimal bound on the potential gain required to effectively complete the transformation. The hyperblock transformation process traverses the nodes of the loop tree in postorder to insure that innermost loops are processed first. Then, for each loop, candidate regions for if-conversion are selected and sorted from the innermost to the outermost to guarantee that no region is transformed that contains a nested non-if-converted region. Thereon, each region is checked against the if-conversion test, i.e the inequality (7) in the model. This step is preceded by the region ahead scheduling pass from which the target hyperblock characteristics are extracted. The profit of the transformation is then evaluated against the threshold value if the if-conversion test succeeds. In order to capture the global effect of the transformation on the program, the profit is also evaluated at the CFG level. Each region is then annotated with its potential gain. The region that exhibits the highest gain is selected for if-conversion. The CFG is then immediately updated to reflect this new change and the profit for each region is anew computed until no more change occurs. This guarantees that the performance and the energy are globally well balanced.

2.3.3 Application on the Lx VLIW processor

In the following subsections, we present an evaluation of the hyperblock formation heuristics described above. We first introduce the evaluation platform and the simulation methodology used. Then, at the end of this section, we present and discuss our experimental results.

2.3.3.1 Platform and simulation methodology

Simulation platform

Our simulations were carried out on the Lx platform [35]. The Lx platform belongs to a family of customizable multi-cluster VLIW architectures. The implementation used in this study features a 4-issue width processor in which each cluster is composed of 4 ALUs, 2 multipliers, 1 Load/Store and 1 Branch unit. The register bank in each cluster includes a set of 64 32-bit general purpose registers and 8 1-bit branch registers to store the branch condition, the predicates and the carrys.

Simulation methodology

The Lx platform is provided with a complete software tool-chain, where no visible changes are exposed to the programmer. The tool-chain includes, among other things, an aggressive ILP compiler, called the Lx compiler. We used this Lx compiler to generate an input assembly prior that any aggressive ILP transformations are been performed onto the code and to run a Lx binary executable. This optimization level actually corresponds to the stage where all traditional scalar optimizations have been undertaken.

The extracted assembly code is then processed by SALTO [16] and ABSCISS [3]. SALTO is a general, compiler-independent tool that makes the manipulation of the assembly code at the CFG level easier. SALTO is also machine independent and can be parameterized to feature a specific target by providing it with a machine description file. We use SALTO essentially to modify the input assembly code by adding new compilation passes. In particular, we added two new passes: a hyperblock formation and optimization pass and a list-based scheduling pass that produces the final assembly code. Currently, the hyperblock optimization pass comprises instruction renaming, instruction merging and instruction promotion. Prior to SALTO, the input assembly code is first processed by ABSCISS, which is a SALTO-based compiled simulator. ABSCISS is mainly used to profile the assembly code at the basic block level and annotate it with runtime informations including the data miss count and the execution frequency.

The machine specific simulation parameters E_c, E_{op}, and E_s were obtained directly from [14], where the authors also provided a validation of the energy model used in this chapter for a simulation platform that is similar to ours. According to the authors [14], one must count with an error range that approaches an average of 5.2% compared to the real RTL power estimates.

2.3.3.2 Results

We realized our experiments on a subset of the Powerstone benchmarks [95]. The selected benchmarks were chosen according to the amount of hyperblock formation opportunity found in them. The characteristics of the selected benchmarks are shown

Benchmarks	Description	region representativeness
adpcm coder	voice encoding	45%
adpcm decoder	voice decoding	43%
bffo	implementation of find first zero	59%
des	data encryption	38%
g3fax	fax decoding	33%

Table 2.1: Benchmarks description.

in Table 2.1. The last column in the table gives the percentage of basic blocks contained in the candidate regions compared to the total number of basic blocks in the CFG.

The energy-delay values are presented in Figure 2.5 (top left). Each bar in the figure corresponds to the original CFG, the CFG with the hyperblock formation heuristics and the CFG in normal hyperblock form, respectively. We can observe from the figure that, in almost all cases at the exception of the last one, the energy-delay product has a better energy-efficiency when the energy heuristics are integrated as part of the hyperblock formation process. On average, we observe an improvement of the system energy-efficiency of more than 17%, which, given the same power consumption budget, practically signifies that the autonomy period of a battery-powered system is lengthened.

These energy-delay values can be better understood by looking at the other results presented in the same figure. In this figure, we plotted the values corresponding to the static schedule length, i.e the number of cycles not counting the stall cycles due to the Dcache and Icache misses, the total operation count and the resulted IPC. The static schedule length and the total operation count aim at estimating the performance cost of transforming a region into a hyperblock. We can indeed observe from the figure that the energy-delay value of the hyperblock scheme is closely correlated with the increase of the static schedule length and the number of executed operations. When the static schedule length and the operation count of the hyperblock scheme are higher than their values in the scheme with the integrated energy heuristics, the corresponding energy-delay product is also not better. This is especially the case when the corresponding increase in the IPC is not large enough to compensate for this, as shown in Figure 2.5 (bottom right figure). We observe this phenomenon by the *adpcm coder*, the *adpcm decoder*, the *bffo* and the *des* benchmarks where the IPC improvement only averages 9% for a total of 15% degradation of the static schedule length and less than 6% increase of the total operation count. The main reasons why the static schedule length and the total operation count increase are due to the effect of the dependence height mismatch and resource contention on the one side, and the insertion of the select instructions on the other side.

When the ILP transformation does not considerably degrade the static schedule length and the total operation count, the energy heuristics act transparently to the

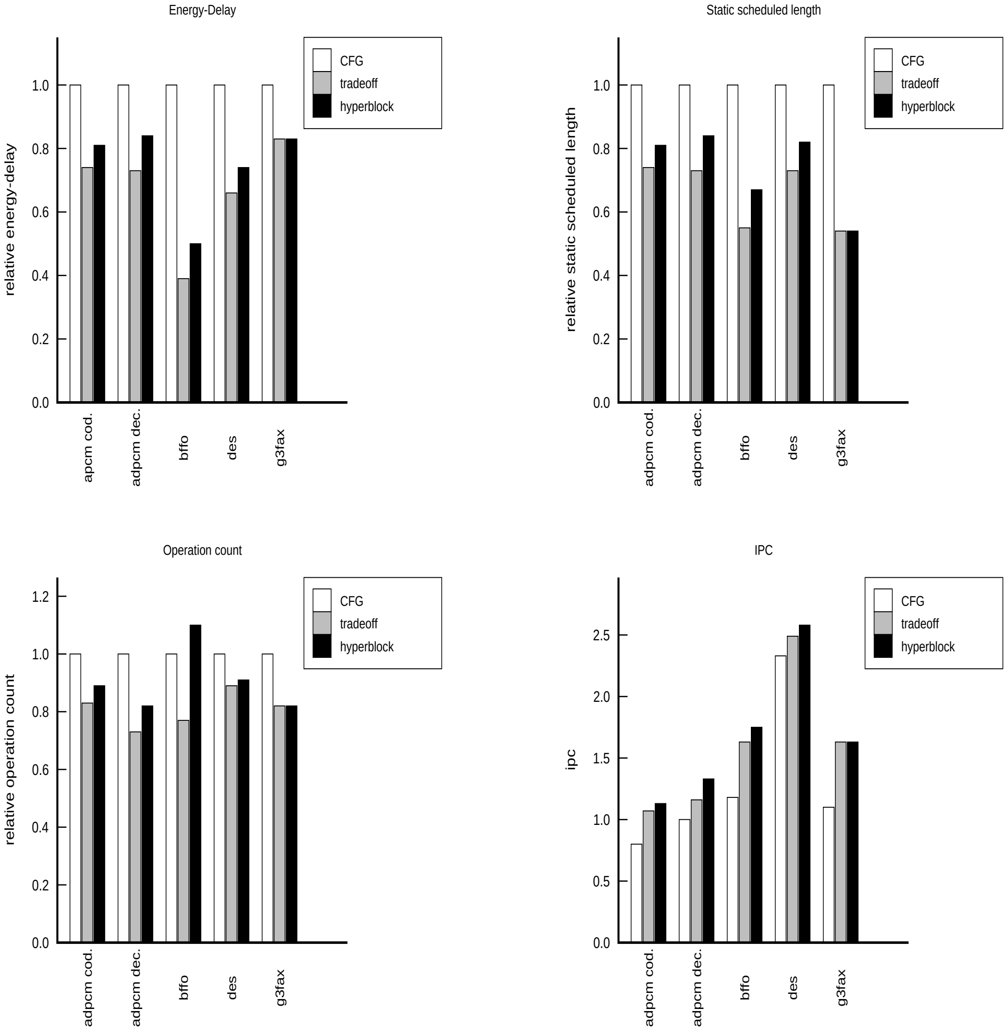


Figure 2.5: Energy-delay (top left), static scheduled length (top right), operation count (bottom left) and IPC (bottom right).

normal hyperblock formation process. This explains why the energy-delay values of the last benchmark are nearly the same in both scenarios. Definitively, an integrated energy heuristic can be used concurrently to the ILP formation process to prevent the compiler from doing optimization when the application has reached a given IPC threshold. This offers the opportunity to discriminate among optimizations that stress the ILP from those that do the same, but in a less efficient manner.

2.4 Related work

There have been many works that have addressed the issues of reducing the energy/power consumption on general purpose or embedded systems. Most of these works have however principally benefit the computer designer community. Until recently, many researches have started focusing on software optimization techniques to provide an alternative solution to that problem. In [81], the authors present a study of the impact of instruction scheduling on energy and performance. In this study, the authors highlight the existing tradeoff between performance and energy by experiencing several list-based scheduling algorithms. They arrive at the conclusion that most scheduling heuristics are not energy-efficient and need therefore to be revisited. The methodology used consisted in scheduling instructions in a DAG based on a energy cost table similar to the model proposed in [108]. Similar work on instruction scheduling include the study by Lee et al. [61] which reduces the total power dissipated by minimizing the switching activity on the bus.

Other works which are closely related to ours include the study by [73] and [111]. In [73] for instance, the author proposes to rely on code profiling to annotate program instructions that can be executed concurrently, hoping in this way to reduce the activity on the fetch issue unit. The approach in [111] follows the same goal, but proposes instead to rely on compiler-based IPC prediction to reduce the activity on the fetch unit. In both cases, the energy consumption can be reduced if the activity on the fetch issue unit can be decreased, because few functional units will be exercised in this way. Our approach is to some extent similar to these two works because we also indirectly seek at reducing the activity on the fetch unit by trading the IPC for energy reduction. However, this can be seen as a "bonus" to our approach since our primarily intention is to enable/disable the application of aggressive ILP optimization techniques whenever this can result to a better energy-delay product. These approaches are therefore complementary each one to each other.

2.5 Summary

A theoretical framework for the energy-delay analysis of ILP transformation techniques on a VLIW-based embedded system has been proposed and experimented. The main purpose of this study was to point out that it is rather questionable if applying aggressive ILP optimization techniques results into an improved energy-efficient system. In particular, we have made clear that aggressive ILP optimization techniques, in addition to improving the ILP, may also emphasize the execution of needless operations, making the benefit obtained by the first to be cancelled as a result of the increasing wasted energy consumption due to the second. Our results have shown that up to 17% energy-delay improvement can be achieved by a compiler that is aware of this. This comfirmed our hypothesis that exposing architecture-based energy features to the compiler may help improving the overall energy efficiency, especially for embedded systems. First, however, we will look at possibilities for improving the compiler efficacy by scrutinizing a program dynamic behavior in order to highlight portions of code that can better benefit from certain optimizations. This is the subject of the next chapter. In the subsequent chapters, e.g. Chapter 4 and Chapter 5, we will concentrate on architecture mechanisms that can be exposed to the compiler to make it achieve a better power/performance tradeoff.

Chapter 3

Program paths analysis

The increasing processor complexity makes the optimization process a compelling task for software developers. These latter usually face the difficult problem of predicting the impact of a static optimization at runtime. One approach used to meet this challenge is to rely on path profiling to collect statistics about dynamic program control flow behavior. While this has proven to be very effective to assist program optimization [53, 116], the way this information is recorded fails to reveal much insight about a dynamic program behavior. One main concern with current path profiling techniques is that they are often restricted to record intra-procedural paths only [10].

More recently, Larus [60] has proposed an efficient technique for collecting path profiles that cross procedure boundaries. In his proposed approach, an input stream of basic blocks is compacted into a context-free grammar using SEQUITUR [78] to produce a DAG representation of a complete program. SEQUITUR, however, is a compression algorithm that proceeds online; hence, the grammar production rules are far from being minimal such that in practice, the achieved compression ratio is likely to incur a high runtime overhead. In addition, as each grammar rule is processed into a DAG, the information pertaining to a particular dynamic path is lost since all dynamic instances of a given path are fused into a unique DAG node.

Proposed approach In this chapter, we propose to collect and analyze whole-program paths offline. In this way, we make it possible to manage reasonable trace sizes, while shifting the cost of online processing off-line. Since, however, the relatively large sizes of the trace may render the paths analysis cumbersome, an approximation of the trace is needed which also can enable efficient path analysis techniques. We introduce a novel program trace representation to deal with path analysis in an efficient way. In particular, our approach stems from the fact that the data retrieval nature of the path analysis problem makes it very tempting to consider pattern-matching algorithms as a basis for paths identification. One such approach is given by *suffix-arrays* [71], which have already proven to be a very efficient data structure for analyzing biological data or text. Conceptually, looking for DNA sequences in biological data, or patterns in a text, is an analogous problem to searching for hot paths in a trace; thus making *suffix-array*-based searching techniques appropriate for path analysis. In addition, in contrast to a DAG representation, a *suffix-array* provides the advantage of treating each dynamic sub-path differently from one other.

Chapter contribution This chapter makes two contributions. The first and the foremost contribution of this chapter is to demonstrate the efficacy of using *suffix-array*-based techniques for analyzing hot program paths. More specifically, we show the appropriateness of suffix arrays to represent program paths, and to identify and characterize the exact occurrences of hot sub-paths in a trace. One particular strength of suffix arrays which make them very attractive for this purpose is their low computational complexity, which usually requires $O(\ln(N))$ time, N being the length of the input trace. The second contribution of this chapter is to indeed illustrate the efficacy of the proposed path analysis technique to guide power-related compiler optimizations. For this purpose, an adaptive cache resizing strategy is used and its potential benefits evaluated at the hot program paths frontiers.

Chapter organization The remainder of this chapter is organized as follows. Section 3.1 introduces the background on suffix arrays. The profiling scheme used to collect paths is described in Section 3.2. In Section 3.3, we introduce the offline algorithm used to identify the sequence of basic blocks that appears repeated in the trace, while in Section 3.4 we show how these sequences can be qualified as hot paths. In Section 3.5, we present our experimental results and discuss a practical application of our scheme to reducing power consumption. Related work is presented in Section 3.6, while Section 3.7 concludes this chapter.

3.1 Suffix arrays background

Suffix arrays have been intensively used in several research areas such as genome analysis or text editing to look for DNA or text patterns. However, despite their widespread use in these domains, we are not aware of any attempt to use this technique in the context of program path analysis. In this section, we briefly introduce the background of suffix arrays and discuss why they may be an efficient data structure for analyzing a whole-program trace.

Given an N-length character string S, the suffix array of S, denoted by Pos in the remainder of this chapter, is defined as the sorted array of the integer indices corresponding to all the N suffices of S. Hence, $Pos[i]$ denotes the string starting at position i in S which extends until the end of the string. Figure 3.1 illustrates a simple example representing a program execution trace T which is first processed into an initial suffix array data structure and then sorted according to a lexicographical ordering.

One key characteristic of suffix arrays is that they can enable computation of search

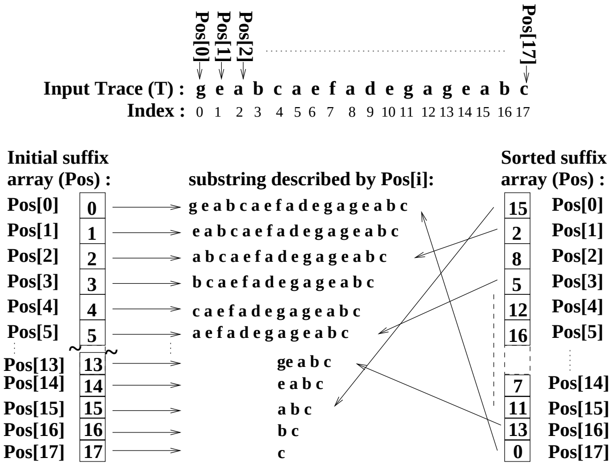

Pos[0]
Pos[1]
Pos[2]
Pos[17]
Input Trace (T) : g e a b c a e f a d e g a g e a b c
Index : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
Initial suffix array (Pos) :
substring described by Pos[i]:
Sorted suffix array (Pos) :
Pos[0] 0 g e a b c a e f a d e g a g e a b c
Pos[1] 1 e a b c a e f a d e g a g e a b c
Pos[2] 2 a b c a e f a d e g a g e a b c
Pos[3] 3 b c a e f a d e g a g e a b c
Pos[4] 4 c a e f a d e g a g e a b c
Pos[5] 5 a e f a d e g a g e a b c
Pos[13] 13 ge a b c
Pos[14] 14 e a b c
Pos[15] 15 a b c
Pos[16] 16 b c
Pos[17] 17 c
15 Pos[0]
2 Pos[1]
8 Pos[2]
5 Pos[3]
12 Pos[4]
16 Pos[5]
7 Pos[14]
11 Pos[15]
13 Pos[16]
0 Pos[17]

Figure 3.1: Example of processing of an input trace T into an initial suffix array representation Pos.

queries in time complexity $O(\text{p} + \ln(N))$, where p is the length of the searched pattern and N the length of the string; making it very convenient to implement very fast searching algorithms. The query computation principally undergoes a binary search phase on the sorted suffix array, taking advantage of the fact that every substring is the prefix of some suffix. In addition to matching the searched query, suffix arrays also permit to compute the frequency of the queried pattern along with the exact positions of all of its occurrences in the string. As for instance, in the given example shown in Figure 3.1, the basic block sequence *geabc* appears 2 times in the trace, respectively at position $i = 16$ and $i = 17$ in the sorted suffix array Pos. By generalizing this concept on variable length substrings, several interesting items of information pertaining to dynamic program behavior can be efficiently retrieved. These include, for instance:

1. finding the longest repeated sequence of basic blocks in a trace, *lmax*;
2. finding all n-length repeated sequences of basic blocks in a trace, $n \leq lmax$;
3. determining the distribution frequency of each specific n-length basic blocks sequence in a trace
4. identifying the positions of each different n-length basic blocks sequence in a trace;

Many of the above items may be of interest for several program optimizations. For instance, item 4 can be used for grouping hot sub-paths together to drive the formation of ILP regions. In addition, if two neighbor hot paths have associated distinct dynamic profiles, this information can be used to decide if their respective profile can be merged or not. This may be helpful for inferring a common configuration to adjacent hot paths in case of an adaptive compilation strategy scheme.

Although suffix arrays present very interesting properties regarding program path analysis, they still have some drawbacks. The most noticeable of them is the memory space required to construct the suffix array, which is linear with the size of the processed trace. This latter issue has since been the subject of intensive studies and some compression algorithms have already emerged that significantly reduce the amount of memory space required [41]. While this work can also be accommodated with such a compression scheme, this is not our main concern in this study.

3.2 Profiling scheme

In this section, we describe our general profiling scheme. In particular, we describe how the whole-program path trace is collected, and what kind of dynamic information can be appended to the trace.

3.2.1 Collecting the trace

Profiling can be used in a straightforward manner to collect a whole-program trace by instrumenting each basic block of the CFG. This approach is however very costly in terms of memory space. A more efficient approach will require to instrument only a subset of the executed basic blocks to capture nearly the same amount of control flow information. Therefore, instead of instrumenting individual basic blocks, we can instrument only a small subset of them, each one representing an individual acyclic basic block region, denoted by *bb-region*. When not specified, a region will refer to either a *bb-region* or a basic block.

We derive the definition of a *bb-region* from the definition of a *strong region* introduced in [11], and from that of the control dependence relationship obtained from the control dependence graph described in [31]. Let $CFG(V, E)$ denotes the directed flow graph with nodes set V and edges set $E \subseteq V \times V$.

Definition 1 *Nodes $v, w \in V$ belong the same strong region iff v and w appear the same number of times in any control flow path occurring from entry point to exit point of the CFG.*

Definition 2 *Node $v \in V$ is said to be control dependent on node $w \in V$ iff:*

1. *there is a non-empty path from v to w in CFG such that w post-dominates each block on the path except v;*

2. *w is either the same as v, or w does not post-dominates v*

Based on **Definition 1** and **Definition 2**, we can define a *bb-region* as given in **Definition 3**.

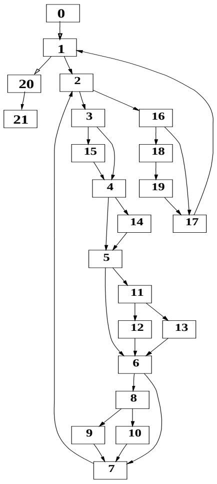


Figure 3.2: CFG.

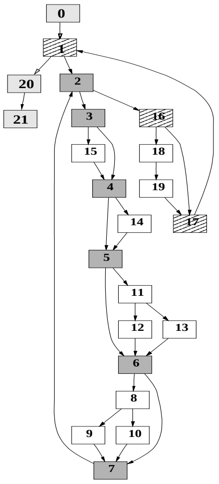


Figure 3.3: CFG with three strong regions.

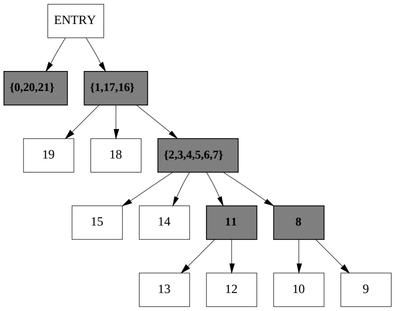


Figure 3.4: CDG with instrumented nodes.

Definition 3 *Node $v \in V$ belongs to a bb-region iff:*

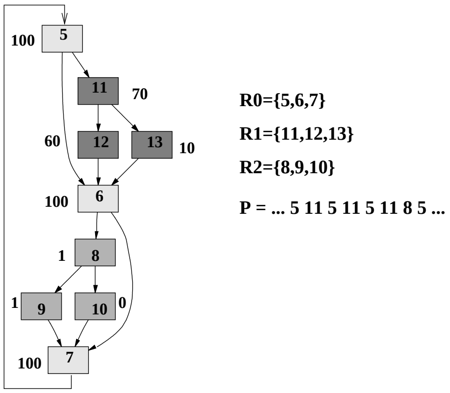
1. *v belongs to a strong region in CFG,*
2. *or, v is the control dependent predecessor of some node u in the control dependence graph CDG, and v does not belongs to any strong region in the CFG.*

The first property in the definition of the *bb-region* given above allows us to identify regions in the CFG in which all basic blocks execute with nearly the same dynamic frequency, whatever the taken control flow path is. In this respect, any node belonging to such a *bb-region* can be used to capture the dynamic control flow path induced by other nodes of the same region. This drastically reduces the number of instrumented basic blocks, as shown in Figure 3.3.

For the other basic blocks that do not fit in any strong region, we proceed as follows. We traverse the *CDG* upwards, selecting each time the control dependent predecessor block that was not previously selected during strong region identification. The idea is to instrument only at each control condition block. This is illustrated in Figure 3.4. As shown in the example, the number of instrumented nodes reduces from 22 to 5 overall, providing up to 80% reduction of the number of instrumented basic blocks.

We applied another compression technique to further reduce the size of the trace. Th is technique principally targets cyclic regions such as loops in which basic blocks execute repeatedly. In such a case, it is not necessary to record all the back-to-back dynamic occurrences of the same region. Instead of that, we can choose to record in the trace the last such dynamic occurrence with all attached information updated accordingly. Combined with our profiling scheme, this shows a real improvement in the compression ratio, typically up to 47%, on average, for our benchmark sets.

3.2.2 Control-flow information accuracy



R0={5,6,7}

R1={11,12,13}

R2={8,9,10}

P = ... 5 11 5 11 5 11 8 5 ...

Figure 3.5: Example of sub-path.

With the node abstraction introduced with the profiling scheme described in the previous section, it is not always easy to reconstruct a copy of the original control flow path. This is however of less a concern since we are more interested to know which *bb-regions* are executed more often than of knowing precisely which of the nodes occur in the trace. Such a hierarchical path profiling approach is at the advantage of the program optimizer since it may permit to focus the analysis only on the predominant paths in the trace. In an another phase, however, the *bb-region* can be investigated more closely to identify individual hot basic blocks.

Consider for instance the example shown in Figure 3.5. Three *bb-regions* are identified, labeled from $R0$ to $R2$. Only the first node of each *bb-region* is instrumented. The corresponding dynamic execution trace is shown with set name P in the figure. This simple example indicates that sub-path $5, 11$ is predominant. In the figure, we also show the cumulated execution count of each node. It is then straightforward to derive from the sub-path $5, 11$ the exact set of the most representative basic blocks by excluding those which execute less frequently, i.e. node 13. In our abstraction, nodes $6, 7, 12, 13, 9, 10$ appear subsumed by the control dependence relation. While this effectively reduces the space, it also emphasizes the rapid identification of the main sub-path $5, 11$. This is central to our program sub-path detection technique.

3.2.3 BBWS signature

When a sequence of basic blocks appears repeated in the trace, we denote by *basic block working set* (BBWS) the set of static basic blocks that constitutes this sequence. The annotation attached to each such sequence is called a basic block working set signature. This annotation can be used to describe such a sequence in a unique manner, depending on the kind of dynamic information that is appended to it. For our experiments, we have considered the region id, *reg-id*, which identifies each instrumented *bb-region* or basic block, the performance parameters *cyc, dyn, dmiss, imiss* which represent the number of elapsed cycles, the dynamic instructions count, the number of data and instruction cache misses attached to each region, respectively. This information will become more apparent during the formation of the hot program sub-paths, to determine the pertinence of a candidate hot region.

3.3 Identifying BBWS

The key idea to search for BBWS is to rely on the suffix array data structure to implement an efficient suffix sorting algorithm. We employ an adapted version of the KMR [54] algorithm used in genetic and text querying systems to achieve this.

3.3.1 KMR algorithm

The KMR (for Karp, Miller and Rosenberg) algorithm is a well known algorithm for computing the occurrence of repeated patterns in a string. The idea is dictated by the observation that each suffix in P can be defined as the k-length suffix of another suffix starting at position i. This implies that, at the j-th stage of the sorting algorithm, $j \geq 1$, the suffix array indices $i + 2^{j-1}$ computed at stage $j - 1$ are used to initially sort each suffix i obtained at stage j. This technique allows to double the suffix length at each stage, requiring only $O(\log(N))$ processing time. The ordering relation used in the KMR sorting algorithm is based on the definition of an equivalence relation over the suffix positions of the path P. Given a path $P = p_1 p_1 p_2 ... p_n$, two suffices starting at positions i and j in P are said to be k-equivalent, $k < n$, denoted by $i \ E_k \ j$, if and only if the path of length k starting at these positions are the same.

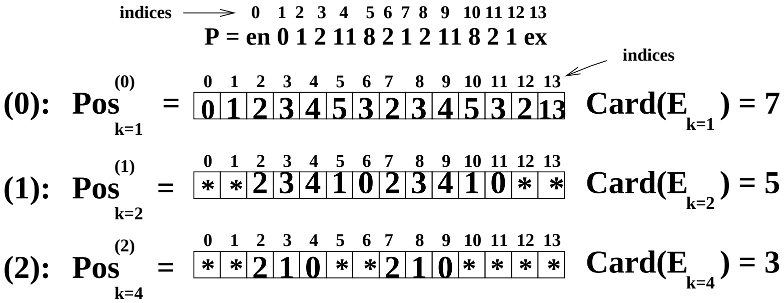

indices
0 1 2 3 4 5 6 7 8 9 10 11 12 13
P = en 0 1 2 11 8 2 1 2 11 8 2 1 ex
indices
(0): Pos (0) k=1 = 0 1 2 3 4 5 3 2 3 4 5 3 2 13 Card(E k=1) = 7
(1): Pos (1) k=2 = * * 2 3 4 1 0 2 3 4 1 0 * * Card(E k=2) = 5
(2): Pos (2) k=4 = * * 2 1 0 * * 2 1 0 * * * * Card(E k=4) = 3

Figure 3.6: Example of BBWS identification

3.3.2 Sorting algorithm description

We can easily make an analogy between the suffix array $Pos_k^{(j)}$ obtained at the j-th stage of the sorting algorithm described in the previous section and a partition of all E_k equivalent integer indices obtained from $Pos_k^{(j)}$, k being the length of the expanded suffix at that stage. Interestingly, the number of elements in the partition gives the actual number of BBWS of length k, whereas their integer indices in the suffix array $Pos_k^{(j)}$ gives their position in the trace P. Hence, it becomes straightforward to identify a BBWS according to its size (i.e length k), its dynamic frequency of occurrence (i.e cardinal of the partition E_k) as well as its dynamic coverage time (i.e start position in the trace until the position where a new BBWS is encountered).

The algorithm used to sort the suffix array Pos is shown in Algorithm 4. The alphabet is composed of the set of *bb-regions* and basic blocks encountered in each CFG. The input search space P represents the execution trace. In line 6 of the algorithm, we first build the partition E_n corresponding to the set of BBWS with maximal repeated occurrence of length n. As each element of the partition is identified, it is hashed into a table of BBWS partitions with the hash key being the length of the BBWS. This is done for E_n as well as for the other partition elements used to iteratively compute it (see Algorithm 3). As shown in lines 8-10 of the algorithm, the program terminates as soon as the set of BBWS identified so far is representative enough of the whole trace P. We address this issue in the next section. In the other case, the algorithm undergoes a binary search to look for other BBWS as shown in lines 11-19. At the end of the algorithm, the partition table T contains, for each valid entry k, the set of all k-length BBWS that appear repeated in the trace. The processing time for this algorithm is quite feasible. For instance, a 40MB trace size requires less than a few minutes to process, whereas for trace sizes ranging from several hundreds of MB to a GB, the processing time is within the order of hours.

Algorithm 1 Initialization

```
Require: $P$ : control flow path defined over $\Sigma^m$
Construct the suffix array $Pos^{(0)}_{k=1}$
Add class elements $E_1$ to $T[1]$
```

Algorithm 2 Construct suffix array $Pos^{(j)}_{k}$ from $Pos^{(j-t)}_{k'}$

```
repeat
    Use $Pos^{(j-t)}_{k'}$ to construct $Pos^{(j-t+1)}_{k'+1}$
    Add class elements $E_{k'+1}$ to $T[k'+1]$
    $k' := k' + 1$
until $k' < k$
```

Algorithm 3 Construct suffix array $Pos^{(N)}_{k=max}$ and E_{max}, N number of processing steps

```
Use Algorithm 1 to initialize the suffix array $Pos^{(0)}_{k=1}$
repeat
    $r = r' + 2^{j-1}$
    Construct $Pos^{(j)}_{k=r}$ from $Pos^{(j-1)}_{k=r'}$
    Add class elements $E_r$ to $T[r]$
until $Pos^{(j)}_{k=r}$ is unchanged
return $r$
```

Algorithm 4 Basic block working set partitioning

```
$n, k$ : Integer $:= 0$
$T$ : BBWS partition table
$\Sigma := \{$Set of reg-id$\}$, $|\Sigma| = m$
$P : p_1\ p_2\ p_3\ \ldots\ p_m \in \Sigma^m$

Use Algorithm 3 to suffix sort the array $Pos$, obtaining $n$, the longest repeated BBWS

if all BBWS are representative of P then
    stop here
end if
for $k = n - 1$ to 2 do
    if $T[k]$ is empty then
        Find $T[d]$ such that $d < k$, $d$ is a power of 2 and $T[d] \neq \emptyset$
        Use Algorithm 2 to iteratively construct $E_k$ from $E_d$
    end if
    if all BBWS are representative of P then
        stop here
    end if
end for
```

3.3.3 Sorting example

Let us consider the example shown in Figure 3.6. We illustrate next the different processing steps involved when searching for the longest repeated BBWS. Step 0 shows the suffix array $Pos_{k=1}^{(0)}$ that corresponds to the initial sorting stage with suffix length $k = 1$. The partition elements of the equivalent class $E_{k=1}$ are deduced directly from $Pos_{k=1}^{(0)}$. The cardinal of the partition gives the number of BBWS of length $k = 1$. At the next iteration step, the array $Pos_{k=2}^{(1)}$ is computed from the array $Pos_{k=1}^{(0)}$ in the following way. The suffix positions of P that correspond to the integer indices in $Pos_{k=1}^{(0)}$ are sorted into buckets of same equivalent class. For instance, suffix positions $2, 7, 12$ in P will belong the same bucket since their integer indices in $Pos_{k=1}^{(0)}$ are identical (i.e. 2). Note that, at this stage, the number of elements in each bucket yields the dynamic execution frequency of the considered BBWS in P. From here, each bucket is sorted according to the b-equivalent relation, where $b = k^{(j)} - k^{(j-1)}$, i.e. $b = 1$ at stage 1. The result of this sorting is a new set of buckets where two suffix positions belong the same bucket iff they are E_b equivalent with regard to their integer indices in $Pos_{k=1}^{(0)}$. For instance, suffix positions $2, 7$ belong the same bucket because they are E_1 equivalent with respect to $Pos_1^{(0)}$. The array $Pos_k^{(j)}$ is obtained by renumbering the integer indices of the suffix positions contained in a bucket list with a same equivalent class number if they satisfy to the b-equivalent relation. Note that BBWS that appear only once are systematically discarded from the suffix array since we are only interested in identifying those that appear at least twice in the trace. This explains the stars in the arrays $Pos_{k=2}^{(1)}$ and $Pos_{k=4}^{(2)}$.

3.4 Qualified BBWS for hot sub-paths

Not all BBWS that are identified with the algorithm described in the previous section are of interest. Of course, there are some BBWS that effectively appear repeated in the trace but which inherently bring no value for the optimization. To distinguish among the BBWS those who are the most representative, we apply three selection criteria, as illustrated in Figure 3.7.

The first criterion is the *local coverage*. This metric is an indication of the number of elapsed cycles in the region, or the dynamic instructions count of that region, before a transition to another region occurs. Either one of the number of cycles or the dynamic instructions count can be directly obtained from the trace, as indicated in Section 3.2.3.

The second criterion is the *global coverage*. This metric is related to the *local coverage* by the dynamic execution frequency of a BBWS, $global\_coverage = frequency \times local\_coverage$. With respect to the overall program execution, this metric assigns to each potential hot path a global cycle weight or a dynamic instructions count weight.

The last criterion is the *distance reuse*, measured in number of dynamic basic blocks. The distance reuse is an approximation of the temperature of a BBWS. As the distance

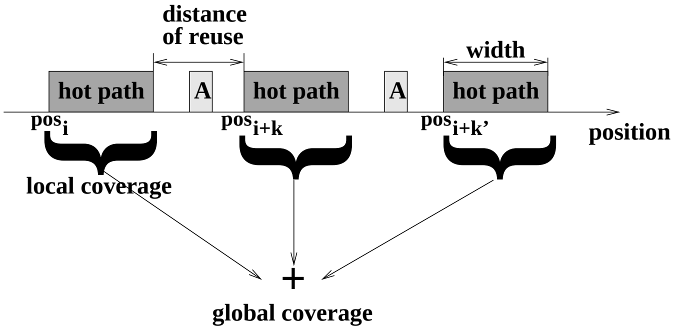


Figure 3.7: Hot path characteristics.

reuse gets larger, the probability that the underlying BBWS is a hot path lowers. This can be mainly attributed to the fact that, although the BBWS appears repeated in the trace, it is not too often executed to infer a hot temperature. In contrast, tighter distance reuses indicate a high probability that the considered BBWS is a hot path. A consequence of this is that cold blocks in the vicinity of a hot path may also be inferred a hot temperature since the heat may propagate to them indirectly. In Figure 3.7 for instance, if the distance reuse of the highlighted hot path is below a given threshold, block A can be included in the BBWS induced by the nodes of the hot path to form a coarser region. Assuming *Position* designates the set of all consecutive, non-overlapping positions of a BBWS in the trace, the average distance reuse $\overline{D}$ is computed as shown in Equation (3.1), where *width* refers to the size of the BBWS (number of basic blocks) and % represents the modulo function.

$$\overline{D} = \frac{\sum (pos_{i-1} + width)\ \%\ pos_i}{|Position|} \tag{3.1}$$

Note that, as the basic block representing a *bb-region* is included in the computation of $\overline{D}$, it must be expanded to the number of basic blocks that is contained in the region. Hence, the expression of $\overline{D}$ provides only an approximation of the distance reuse value. A hot path candidate is then formed by selecting BBWS with a relatively high local coverage and low distance reuse. The global coverage serves as an indication of the hot paths weight in the program.

3.5 Experimental evaluation

This section presents an evaluation of the proposed approach. We first introduce the simulation platform and the benchmarks used in Section 3.5.1. Then, in Section 3.5.2, we evaluate and discuss our results.

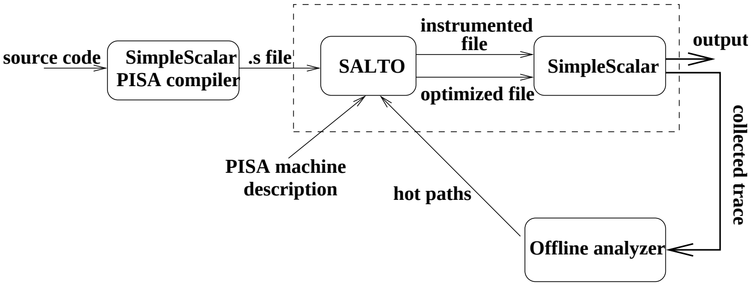


Figure 3.8: Simulation framework.

Bench.	description	trace size (MB)	compr. ratio
dijkstra	shortest path	110	65%
adpcm	code modulation	148	67%
bf	symmetric block cypher	55	74%
fft	fast fourier transform	6	85%
sha	secure hash algorithm	11	77%
bmath	math. calculation	6	78%
patricia	IP traffic	21	77%

Table 3.1: Benchmarks

3.5.1 Experimental methodology

We conducted our experiments using applications collected from MiBench [42] as illustrated in Table 3.1. The applications are first compiled with the PISA compiler from the SimpleScalar [22] tool suite, with optimization level 3, to obtain an input assembly file. Each assembly file is then processed by SALTO [16], which is a general, compiler-independent tool that makes the manipulation of the assembly code at the CFG level easier. SALTO is used essentially to instrument the code, using the SimpleScalar annotation feature, and to add new compiler optimization passes. The produced executable is processed by SimpleScalar to extract the compressed trace which is then fed to the offline analyzer. After the hot paths have been identified, this information can be re-injected into SALTO to drive the various compiler-dependent optimization passes. An overview of the different processing stages is shown in Figure 3.8.

Our measurements were performed with SimpleScalar, which we use to model a 5-stage in-order issue processor such as those encountered in the embedded computing domain, e.g. the Lx processor [35]. Details on the processor configuration parameters used in this study are shown in Table 3.2.

Issue	in-order 4-issue
Integer ALU	4
Multiplication units	2
Load/Store unit	1
Branch unit	1
instr. cache	32K 1-way
data cache	32K 4-way
cache access latency	1 cycle
data cache replacement policy	LRU
memory access latency	100 cycles

Table 3.2: Baseline microarchitecture parameters.

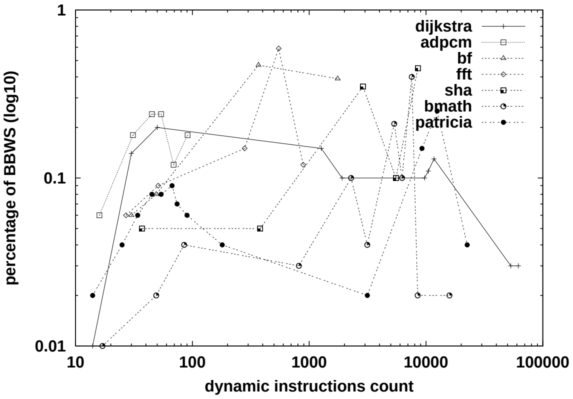


Figure 3.9: Local coverage.

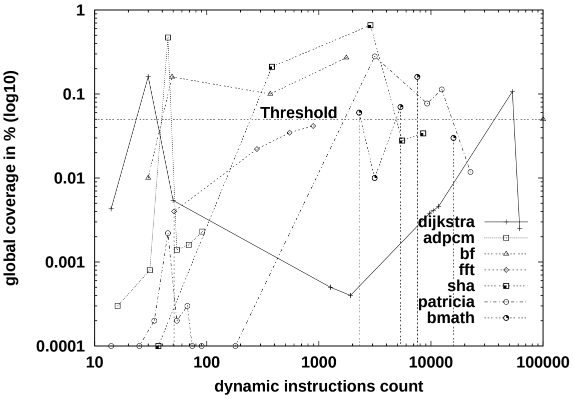


Figure 3.10: Global coverage.

3.5.2 Evaluation

This section presents the evaluation results of using our scheme on the set of benchmarks described in the previous section. The evaluation consisted to measuring the relative compression ratio achieved by our approach and to analyzing the quality of the detected hot paths with respect to the criteria introduced in Section 3.4.

3.5.2.1 Trace size

The last column of Table 3.1 gives an estimate of the compression ratio achieved with our approach. Note that the size of the trace depends strongly on the information encoded with each trace line. For this experiment, we used 20 bytes for each trace line, one byte each for recording the region id, the number of cycles, the number of dynamic instructions, the number of data and instruction cache misses associated with each region respectively. More elaborate trace line representations can be imagined to reduce further the trace size; however, this is not the scope of this chapter. The column labeled *trace size* shows the original size of the trace. As it can be seen from the table, the trace size can be reduced by up to 74% on average. This compression ratio includes the compaction of back-to-back occurrences of loop paths (see Section 3.2.1), which accounts for about 47% of the trace reduction.

3.5.2.2 Local coverage

This metric measures the time spent in a BBWS, or the number of dynamic instructions executed within that BBWS. Figure 3.9 shows the distribution of the local coverage for some representative BBWS when considering the dynamic instructions count. As it can be observed from the figure, some applications have their BBWS which extend from a few tens of instructions to a few hundreds or more, e.g. *adpcm, fft, bf, dijkstra, patricia, bmath*. These applications are therefore best candidates for local optimizations such as instructions coalescence that reduces a region's critical path, or local strength reduction which replaces expensive operations with cheaper ones. In the figure, some BBWS whose sizes extend beyond a few thousand of instructions are also distinguishable, e.g. *bmath, dijkstra, patricia, sha*. As these applications tend to spend a large amount of their execution time within a single region, they may best benefit from memory re-layout techniques such as cache-conscious placements or resizing. The local coverage is however not sufficient enough for deciding on the pertinence of a BBWS.

3.5.2.3 Global coverage

The local coverage must be interpreted in the light of global coverage to yield a fair understanding of the pertinence of a BBWS. Such a comprehensive reading can be provided with help of Figure 3.10. For this experiment, we have fixed an arbitrary threshold at 5% of the total instructions count as indicated in the figure with the *threshold* line. With regard to the local coverage of each BBWS, the points above the threshold line are therefore these that are likely to provide substantial performance benefits across the whole program run. Of most concern are all the applications at the exception of *fft*, which has a global coverage value slightly below the threshold. Some applications such as *sha* and *patricia* exhibit BBWS whose sizes extend from a few hundreds to a few thousands of instructions, with a fairly good distribution among the two. This is an indication that these BBWS are good candidates for both local and global optimizations.

3.5.2.4 Distance reuse

The last criterion that qualifies a BBWS as a hot path is the distance reuse. This metric measures the heat of a BBWS by estimating the average number of accesses to different basic blocks between non-overlapping occurrences of this BBWS. Clearly, the

Bench.	qualified BBWS (%)	local cov. (%)	global cov. (%)	dist. reuse (avg)
dijkstra	2.81	0.09	47	1.74
adpcm	5.88	< 0.005	90	0.00
bf	27.01	0.06	24	85.00
fft	11.7	< 0.005	7	4.21
sha	20.0	0.06	72	0.75
bmath	15.22	0.05	37	19.21
patricia	5.85	0.15	65	24.84

Table 3.3: Qualified BBWS as hot paths.

larger is the distance reuse, less is the probability that it is a hot path. This trend can be well observed in Figure 3.11 where we show the distribution of the distance reuse for our benchmarks set. Applications with BBWS whose sizes extend to a few tens of instructions tend to have distance reuse values distributed among a few tens (e.g. *bmath, fft, bf*) to a few hundreds (e.g. *dijkstra, bmath, fft, sha, adpcm*) and thousands (e.g. *patricia, dijkstra, adpcm*) of basic blocks. Medium sized BBWS which extend from a few hundreds to a few thousands of instructions constitute the other category with distance reuse values less than a thousand, at the exception of *patricia, dijkstra* and *bmath*. Finally, as it is to be expected, very large BBWS tend to have also poor distance reuse values as evidenced with *patricia* and *dijkstra*. Tough, an exception with *dijkstra* is to be noted as a few number of these BBWS exhibit very good distance reuse value with $\overline{D} \approx 1$.

We summarize our experimental results in Table 3.3. The values were computed with a global threshold at 5%. This table presents results obtained by combining all the selection criteria together in order to qualify a BBWS as a hot path. As illustrated in the table, from 7% to 90% of the program dynamic instructions can be covered using our approach, with only as much as 0.15% of the dynamic instructions being executed within a single region.

3.5.3 Application example: adaptive cache reconfiguration

The cache hierarchy is the typical example where the power/performance tradeoff plays a central role. While a large cache permits significant improvements in performance, only a small fraction of it is usually accessed during a program run. Henceforth, to address this source of inefficiency, much recent work has focused on the design of configurable caches [7, 117]. A key point with such work is deciding when to perform such a reconfiguration. With general purpose processors, this can be done dynamically with some mean of hardware, or at software following procedure boundaries [7]. With embedded systems, this is often done once on a per-application basis [117].

In this section, we examine the possibility of reconfigurating a cache at the hot path boundaries. To do so, we assume a scheme similar to that presented in [117] in which the

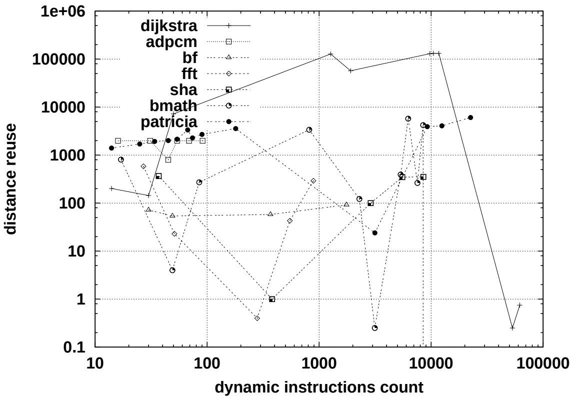


Figure 3.11: Distance reuse.

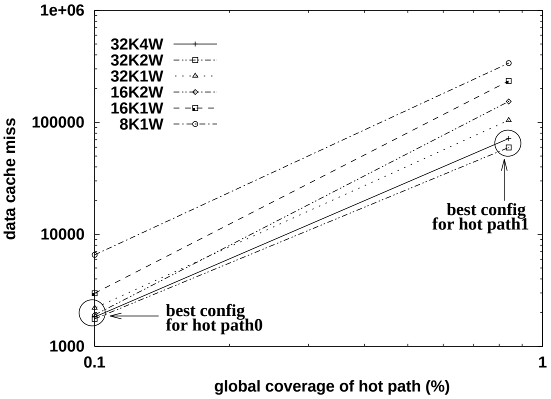


Figure 3.12: Dcache miss distribution (*dijkstra*).

cache configuration	energy per access
32K4W	1.00
32K2W	0.58
32K1W	0.37
16K2W	0.55
16K1W	0.35
8K1W	0.35

Table 3.4: Relative energy ratio.

associativity of a cache can be modified while still preserving the whole cache capacity. Furthermore, we also assume an extension of this scheme, proposed in [85], in which the associativity as well as the size of a cache can be adapted at runtime with the help of a reconfiguration instruction. We illustrate how a cache reconfiguration can be guided with *dijkstra*, since it has the best BBWS profiles with larger local and global coverage and low distance reuse.

Figure 3.12 shows the cumulative distribution of the number of data cache misses, using varying cache configurations, for the two most representative hot paths of *dijkstra* with global coverage at 10% and 83%, respectively. As indicated in the figure, each hot path has a set of cache configuration candidates which vary according to either of the selection criteria introduced in Section 3.4. The first hot path, for instance, has a distance reuse of ≈ 0 and a local coverage of 0.09%, whereas the second occurs practically each 4 blocks with a relative low local coverage (≈ 0.004%). The first hot path is therefore more regular than the second, which could explain the larger choice for the former. Table 3.4 shows the relative energy per access obtained by means of CACTI [98] for each cache configuration. Each xKyW stays for a cache of size x and associativity y. The best configuration for *hot path0* is given by 32K1W which is from far more energy-efficient than the 32K4W case. On the other hand, for ***hot** path1*, 32K2W yields the best energy-performance ratio. However, although both hot paths yield substantial energy savings, only the first one may be of interest because it has a near 0 distance reuse value, which infers that reconfiguration will take place very infrequently. This is crucial for performance as each reconfiguration instruction consumes extra cycles and energy. The energy savings obtained in this way is in the order of 12% with almost no performance slowdown (less than 1%).

3.6 Related work

Many work have been proposed to collect profiling information. In [9], Ball and Larus propose to collect profile information via edges profiling. They extended their work in [10] to include path profiling information that are restricted to intra-procedural paths. Bala [6] then augmented the intra-procedural path profiling scheme to capture inter-procedural paths as well. A similar work has been proposed by Larus [60] which relies on a online compression scheme, SEQUITUR [78], to produce a compact representation of a whole-program paths. Our scheme is to some extent similar to [60] in that we also provide a representation of a whole-program paths. However, unlike the DAG representation used in [60], we rely on a suffix array representation that permits the implementation of very fast searching algorithms, allowing quick offline processing; thereby offsetting the high runtime overhead of Larus's scheme. In addition, this also permits us to treating each dynamic path distinctly from one other and consider large trace sizes. The performance of the proposed scheme can be rather significantly improved, namely by using other compression techniques which are complementary to that proposed in this chapter. For instance, a direct improvement can be obtained by encoding the suffix array compression scheme described in [41]. Compression techniques such as that describe in [83] can also be used to further reduce the size of the trace to less than a fraction of a bit per reference.

3.7 Summary

While suffix arrays have been widely used in biological data analysis or text editing, we are not aware of any prior published work that shows its application to compiler optimization. In this chapter, we presented a first attempt to apply suffix array to the compiler domain. In particular, we showed how a suffix arrays can be used to represent a whole-program paths and to accurately identify hot program paths. Our evaluation results revealed that up to 48% of a program code can be covered by hot paths, with each hot path representing about 0.15% of the total instructions. Practical application of our approach has confirmed its efficacy to reduce power consumption. We showed that up to 12% energy savings can be obtained with a hot-path-directed adaptive cache resizing strategy that used our technique. Because of its power to precisely model program paths (distance reuse, local and global coverage), we believe that suffix arrays can be of a crucial aid to assist a programmer during the optimization process.

Chapter 4

Power-efficient reconfigurable cache

When compiling for low-power, the cache hierarchy is the typical example where the power/performance tradeoff takes a great significance. On one hand, a large cache allows to maintain an important fraction of the embedded code and the data workload on-chip; thus reducing the amount of memory traffic and thereby improving the performance and the power consumption. On the other hand, however, typical cache memory accounts for up to 80% of the total transistor count and it is usual to devote about 50% of the total chip area [45] to host a cache. This makes the cache memory subsystem an important source of power dissipation.

Problem summary Recent researches in this area have focused on the design of configurable caches [44, 70, 1, 117, 7, 115] to effectively tackle the problem of power dissipation. The main motivation behind a configurable cache is to allow one to adapt the cache size requirement of a running program to a desired power/performance trade-off. Energy is saved because fewer switching transitions take place in the cache as it is resized to smaller sizes. However, former configurable cache proposals for embedded systems [44, 70, 117] have only considered configuration on a per-application basis. A drawback with this approach is that an optimal cache size, viewed from a performance standpoint, has not yet been shown to exist, whereas each application simply exhibits varying dynamic cache behaviors [97, 101]. Moreover, in the context of embedded systems, compilers play a central rôle in obtaining good performance; it is thus important to consider configuration schemes that improve compiler's effectiveness as well.

Chapter contribution This chapter explores new solutions to the problems explained above. First, a model of a hybrid configurable cache design is proposed as a alternative to two current proposals. With this model, we allow a cache to be reconfigured on a per-phase basis rather than at the application level, with only minor hardware modifications, keeping the design complexity simple. Second, based on the proposed model, the potential benefits of a fine-grain cache size adaptation scheme is explored that can be used at the compiler level for automatically characterizing the different cache size requirements of a program phase. The proposed model considers a great degree of flexibility, providing the compiler with the opportunity of resizing a cache along its size and/or degree of associativity.

Chapter organization The remainder of this chapter is organized as follows. Section 4.1 provides a review of configurable cache designs. In Section 4.3, we detail our model of a hybrid reconfigurable cache architecture. The compilation support to our hybrid cache model is presented in Section 4.4. Experimental results are presented in Section 4.5. Section 4.9 discusses related work and Section 4.10 concludes.

4.1 Design space exploration of configurable caches

Prior to reconfigurable caches, researchers have sought to reduce the amount of switching activity on a cache access. Several proposals were made to accomplish this. Way-prediction was proposed in [50] and then improved in [90]. The principal idea in way-prediction is to allow ways of a set-associative cache to be accessed speculatively. In the case of a prediction hit, this reduces the number of ways that are concurrently accessed on each cache lookup, and therefore the energy consumption. However, when a prediction miss occurs, the hit/miss cache access time is lengthened because the remaining ways need to be checked again for the valid data. A similar idea is the phased-lookup cache [43]. A cache access is split into two phases. A first phase corresponding to a tag check allows to select the correct data way. The second phase corresponds to a delayed data access where only the selected data way is activated. As in the previous case, this technique also suffers a high penalty in case of a miss in the cache. Another approach involves preventing frequently accessed working set of smaller sizes from hiting in the L1 cache [57, 12]. A smaller, and thus more energy efficient, L0 cache is placed between the processor core and the L1 cache. A hit to the L0 cache reduces the energy because the amount of switching activities is lowered due to the small size of the L0 cache. However, a miss in the L0 cache increases both the cache access time and the energy.

Configurable caches offer a powerful alternative for reducing the energy dissipation of conventional caches. The basic idea is to permit a cache memory system to adapt to the cache size requirement of a running program. The various proposals of configurable cache architectures principally differ in their resizing granularity and their design complexity.

In [1], Albonesi proposes to partition a set-associative cache along its tag and data ways. Energy can be saved by allowing cache ways to be disabled/enabled on demand, according to the cache size requirement of the application. The hardware implementation is simple, with only a software register mask that enables/disables cache ways. However, this approach can only be accommodated to set-associative caches. The configurable cache design proposed in [82] is somewhat similar to selective-ways. However,

size/#sets	1024	512	256
32K	DM	2-way	**4-way**
16K		DM	2-way
8K			DM

Table 4.1: Possible cache size granularities for a 4-way, 32B line base cache with 8K per bank.

instead of disabling the unused cache sections, the authors suggest to transfer useful tasks to them (e.g. instruction reuse for media processing).

Other approaches of configurable caches consist in partitioning a cache along its sets [115] or at the granularity of the cache line [55, 119]. In contrast to [1], these approaches can be accommodated to direct-mapped caches as well. However, the required implementation cost can be much more expensive. For instance, resizing a cache to the smallest and/or largest addressable number of sets with [115], requires to maintain a number of tag bits that often exceeds the one found with a conventional cache of equal size[1].

A more recent work by Zhang et al. [117] proposes to exploit the way partitioning scheme of a set-associative cache to reconfigure it as either a direct-mapped cache or a set-associative cache of lower degree of associativity. The proposed configuration scheme exploits a technique called "way-concatenation" which permits cache ways to be merged, while still retaining the full cache capacity but with reduced set-associativity. This approach reduces the dynamic energy since, with same cache size, lower associativity caches perform fewer switching activities than higher associativity caches. In addition, the implementation cost has been shown to be minimal.

4.2 Motivating a phase-based resizable cache scheme

The motivation behind the proposed model is to emphasize the most critical application-specific cache architectural tradeoffs involved during program execution. To do so, we consider a cache with a fixed line size and modulus mapping function. Our main observation is that the performance of such a cache is mainly dictated by its size and degree of associativity. Therefore, from a software perspective, we would like to select the configuration with the lower energy consumption that minimizes the miss ratio (i.e. the one with lower degree of associativity and smaller size). However, since programs have varying dynamic cache behaviors, they must also feature varying dynamic cache size and/or degree of associativity. Thus, instead of selecting these parameters on a per-application basis, we would then prefer to tune them according to a program dynamic phase.

[1] e.g. to upsize the number of sets from 256 to 1024 in a 32K 4-way cache with 8 tag index bits, 10 tag bits must be maintained instead.

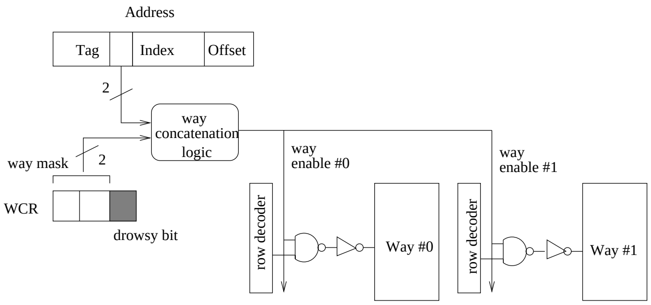


Figure 4.1: Baseline architecture of a 2-way associative cache.

The idea is to use a combination of schemes that permits to reconfigure a cache along its size and associativity in order to provide both in one. We do this by exploiting the variability in the cache size and the degree of associativity provided by combining the selective-way scheme [1] and the way-concatenation technique [117]. Such a hybrid scheme can provide fine-grain cache sizes at various degrees of associativity. Table 4.1 shows a subset of some possible cache configurations that can be exposed to the compiler. For example, starting from a 4-way 32K baseline cache configuration, we move to the 16K direct-mapped configuration by either concatenating 2 banks (32K 2-way) and then selecting only one of the two, or selecting two (16K 2-way) active banks and then concatenating them.

4.3 Potential cache model

4.3.1 Baseline model

Our model builds upon the way-concatenation scheme introduced in [117] and extends it in order to include a flexible selective-way scheme to resize a cache at runtime. Basically, with the way-concatenation scheme, one can select the number of cache ways m that can be activated on each cache lookup. In this scheme, each selected way is virtually a multiple of the size of a cache way in the n-way case, n being the number of available cache ways. For instance, if one way is active, it has virtually four times the size of the 4-way case. A configuration register is provided to set the number of active ways m. A way concatenation logic is in charge of carrying the active/inactive way-enable signal to each of the n cache ways. The baseline architecture is depicted in Figure 4.1. In this figure, the two high-order bits of the way concatenation register (WCR) are used as configuration register to fix the number of active cache ways.

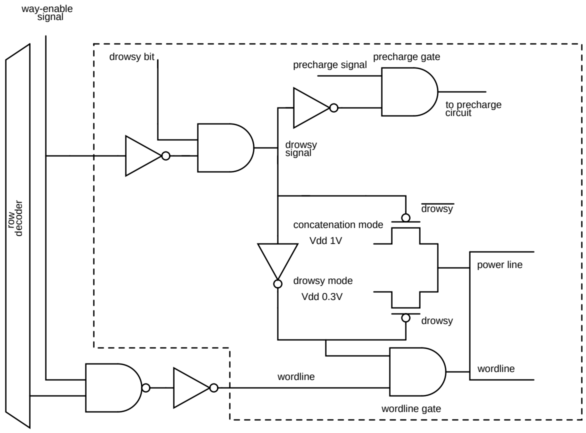


Figure 4.2: Drowsy cache line circuit.

4.3.2 Architectural modifications

4.3.2.1 Associativity dimension

The main issue of concern we address at this stage is preserving the cache coherency across different cache configurations, while minimizing the reconfiguration time. Consider, for instance, the reconfiguration scenario illustrated in Figure 4.3. In phase i, corresponding to a 2-way cache configuration, the way-concatenation logic activates bank 0 and bank 2 when @A is referenced. In this case, @A hits in bank 0. In phase $i+1$, however, the cache configuration changes to a direct-mapped cache and @A is write accessed in bank 1. At this stage, there are two possible locations for @A, the old one in bank 0 and the new one in bank 1.

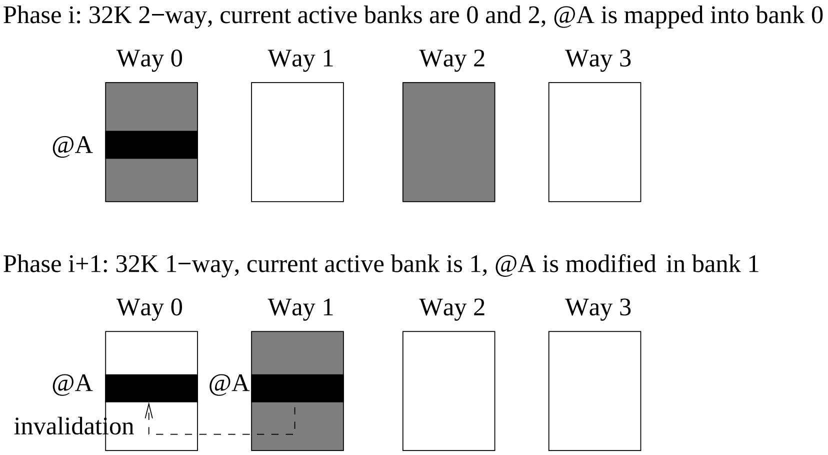

Phase i: 32K 2−way, current active banks are 0 and 2, @A is mapped into bank 0
Way 0
Way 1
Way 2
Way 3
@A
Phase i+1: 32K 1−way, current active bank is 1, @A is modified in bank 1
Way 0
Way 1
Way 2
Way 3
@A
@A
invalidation

Figure 4.3: Reconfiguration scenario.

A possible way to overcome the cache coherency problem illustrated above is to maintain the tag and status arrays always accessible. This implies that only the data array is activated/deactivated by the way-concatenation logic. The tag and status arrays therefore still continue to behave like in a conventional cache. The actions of the cache controller can then be modified to access all tag arrays on each write request to set the corresponding status bit as invalid whenever the referenced address hits in one of the bank. This scenario is illustrated in Figure 4.3 with the dotted arrow line indicating the action of the invalidation signal in the tag array. Future accesses to the invalidated data will cause the new data to be provided by the upper level cache hierarchy. For sake of simplicity, we assume a write-through cache policy. This guarantees data coherency whenever a cache line is provided from the upper level memory hierarchy. This implementation can be done via a special instruction in software to force this behavior or it can also be done transparently in hardware.

4.3.2.2 Cache size dimension

We augmented the way-concatenation architecture to include a drowsy bit [38], represented by the low-order bit of WCR shown in Figure 4.1. The drowsy bit is intended to control the activation/deactivation of the selective-way scheme (drowsy mode). We assume for this mode that the machine supply voltage can be dynamically scaled to higher values of the threshold voltage V_T. As in the gated-Vdd scheme [91], this mode reduces the leakage energy by using higher threshold supply voltages that cause the leakage current to be reduced as a by-side of the short-channel effects. In contrast to the gated-Vdd scheme, however, the supply voltage is scaled in such a way that the state of the memory cell is preserved. Therefore, when reducing the cache size, we do not need to completely disconnect a cache bank which may otherwise cause the loss of the data stored into it. Note that, the drowsy mode only applies to the data array, as explained above. This solution has been preferred in order to avoid the one cycle wake-up delay needed to bring a tag-way out of the drowsy mode with each write access. The fact of continuously maintaining the tag array in a non-drowsy mode has a negligible impact on the leakage energy since the tag ramcells count for less than 4.2% of the total area of our base cache.

Figure 4.2 reflects the changes introduced into the cache line in order to accommodate the drowsy mode. The drowsy bit is ANDed with the way-enable signal of each cache line. An entire cache way may be put into drowsy mode depending on the status of the way-enable signal and the drowsy bit. In particular, this happens when the drowsy bit is set and the corresponding way-enable signal is unset. In such case, the supply voltage for each cache line switches to the lower voltage, putting the entire bank into drowsy mode. With the other cases, the supply voltage for each cache line is set to the normal voltage, bringing the entire cache bank out-of drowsy mode.

way-mask value	drowsy bit state	cache config.
0	0/1	32K1W/8K1W
1	0/1	32K2W/16K1W
2	0/1	32K2W/16K2W
3	0	32K4W

Table 4.2: Effects of the **MOVWCR** instruction.

4.3.3 Design cost

To drive the drowsy signal of each cache line, an inverter and a AND gate have been added. By assuming a memory cell dimension of 1.84 x 3.66 um, this results to approximatively 2 memory cells per cache line. According to [38], the voltage controller adds about 3.35 memory cells, assuming a memory cell layout of 6.18 x 3.66 um. The two inverters, one in the voltage controller and the other in the precharge circuit, add an equivalent of 1 more memory cell per cache line. Finally, the wordline gating circuit accounts for 1.5 additional memory cells, making a total of 7.85 memory cells overhead per cache line. Overall, for a cache size of 32K and a line size of 32B, this makes an area overhead of less than 3%. Note that, in comparison to the circuit shown in [38], there is no need to use a drowsy bit on each cache line since the drowsy signal is directly derived from the way-enable signal which is driven to each cache way. Using a drowsy cache adds however some performance penalty. When a drowsy cache way is activated, the voltage controller to each cache line simultaneously retires from the low voltage to set the memory cell power line to the normal voltage. This takes one additional clock cycle.

4.3.4 ISA support

The ISA support for the presented model can be resumed to a simple WCR modify instruction, denoted by **MOVWCR**, to read/write the content of WCR shown in Figure 4.1. Given that such an instruction is provided by the ISA, Table 4.2 illustrates how this instruction can be used to feature the different cache configurations shown in Table 4.1.

4.4 Trace-based program addresses analysis

The characterization of the cache size requirement of a dynamic program phase is performed on a trace of address references previously extracted by means of profiling. Since an embedded system is often designed to run a few types of applications, it is worth to spend a fraction of time optimizing each embedded application intensively. In such case, the time required to profile and pre-process the embedded applications can be justified.

Our approach to program profiling and trace processing consists in collecting the dynamic LRU-stack profiles $P_{\Pi_i}(map_j(x))$ and $E_{\Pi_i}(map_j(x))$, explained later, of the running program, at some fixed sample interval Π_i. The variable x in the previous expressions corresponds to the LRU-stack depth. By exploiting the cache inclusion property, assigning different values to x permits to simultaneously evaluate alternative cache memory configurations that share the same set-mapping function map_j. Thus, by also varying the set-mapping function map_j, we can increase the range of alternative cache configurations that can be simultaneously evaluated in a one pass simulation through the address trace [46]. We assume for the rest of this study that the caches we model support no prefetching, have the same block size and use the LRU replacement policy.

4.4.1 Cache size performance profile, $P_{\Pi}(map(x))$

At each sample interval Π_i, $P_{\Pi_i}(map_j(x))$ defines the performance profile of a cache with set-mapping function map_j and LRU-stack distance x. This performance profile can be seen as the number of dynamic references that hit in all cache configurations with the same set-mapping function map_j and with the LRU-stack distance $d \leq x$.

4.4.2 Cache size energy profile, $E_{\Pi}(map(x))$

Similarly, the expression $E_{\Pi_i}(map_j(x))$ defines the dynamic energy profile of a cache with set-mapping function map_j and LRU-stack distance x. We define the dynamic energy per sample interval as follows:

$$\begin{aligned} E_{\Pi_i}(map_j(x)) &= P_{\Pi_i}(map_j(x)) * E_c \\ &+ |\Pi_i| * E_T + N_{\Pi_i} * E_d \\ &+ (|\Pi_i| - Read_{\Pi_i}(map_j(x))) * E_m \end{aligned} \tag{4.1}$$

where

$$E_m = energy\_ratio * E_c \tag{4.2}$$

In (1), E_c is the dynamic energy on each cache access, E_m the dynamic energy per memory access, E_d the dynamic energy per drowsy transition, E_T the dynamic energy per each tag access, and N_{Π_i} the number of transitions to/from drowsy mode within the sample interval Π_i. N_{Π_i} is measured by means of monitoring all bank transitions within two consecutive dynamic cache memory accesses, reporting only those bank transitions whose prior state was set to drowsy. The first expression in (1) models the dynamic energy due to a hit in the cache. The second and third expressions respectively model the energy due to accessing the tag part and the energy due to the drowsy mode transitions. Finally, the last expression models the energy due to the memory access on a read miss event and on each write access to the cache. In (2), we estimated the $energy\_ratio$ constant to be 50.

Parameter	**Value**
Issue width	4
Integer ALU	4
Multiplication units	2
Load/Store unit	1
Branch unit	1
data cache	32K 4-way
data cache line size	32B
data cache access latency	1 cycle
data cache replacement policy	LRU
memory access latency	20 cycles

Table 4.3: Lx microarchitecture parameters.

Benchmark	**Suite**	**Datasets**
fft	MiBench	large
gsm	MiBench	large
susan	MiBench	large
mpeg	mediabench	test
epic	mediabench	test_image
summin	Powerstone	custom
whestone	Powerstone	custom
v42bis	Powerstone	custom

Table 4.4: Benchmarks used and number of accesses to data cache (in million).

4.5 Experimental setup

This section presents a preliminary evaluation of the cache size adaptation scheme introduced in Section 4.3 and Section 4.4.

Our simulations were carried out on the Lx platform [35]. The Lx platform belongs to a family of customizable multi-cluster VLIW architectures. The implementation used in this study features a 4-issue width processor. The details of the processor microarchitecture parameters are shown in Table 4.3. We evaluated our cache size adaptation scheme with different applications collected from MiBench [42], Mediabench [62] and Powerstone [95] suites. All the chosen applications were compiled with the Lx native compiler, with the optimization level 3, and then run until completion. Table 5.2 shows an overview of each benchmark together with the datasets used. Some of the benchmarks from these suites that we did not considered were not able to be compiled with the Lx native compiler or were exhibiting close behaviors to some applications that we already selected.

Our simulation parameters were obtained by means of CACTI [98] and Hotleakage

Parameter	Value
process technology	0.07 um
normal supply voltage	0.9 V
drowsy supply voltage	0.3 V
memory access latency	100 cycles
processor clock speed	5.6 GHz
drowsy transition latency	1 cycle
32k 4-way dynamic energy/access	0.294 nJ
32k 2-way dynamic energy/access	0.173 nJ
32k 1-way dynamic energy/access	0.110 nJ
16k 1-way dynamic energy/access	0.104 nJ
16k 2-way dynamic energy/access	0.164 nJ
8k 1-way dynamic energy/access	0.104 nJ
drowsy energy/transition	0.256 pJ
gated-Vdd leakage energy/cell	0.245 fJ
drowsy leakage energy/cell	0.308 pJ
normal leakage energy/cell	0.835 pJ

Table 4.5: Simulation parameters.

[118]. In particular, we extended CACTI to include the leakage energy functions of the Hotleakage tool. We then employed the resulted modified CACTI tool to estimate the dynamic energy per cache access for each simulated cache configuration, as well as the leakage energy per cell for each simulated leakage energy reduction technique. The dynamic drowsy transition energy was derived based on the results published in [38]. Table 4.5 gives an overview of the full simulation parameters that apply to this study.

4.6 Study of program behavior

After empirical evaluations, we have chosen a sample interval size of $\Pi_i = 100K$ cycles to record the energy and performance values for each cache configuration, as described in Section 4.4.2. In order to emphasize the different energy and performance tradeoffs between the cache configurations, we graphed in Figure 4.4 and Figure 4.5 the cumulated energy values vs the number of cumulated cache misses encountered in each sample interval of program execution. The x-axis records the logarithmic scale of the cumulated number of cache misses obtained in each sample interval (e.g. a 3 in the x-axis means 10^3 misses). Each point of the x-axis is associated with a value in the y-axis corresponding to the cumulated amount of dynamic energy consumed (also in logarithmic scale) up to that sample interval. Let us for instance consider the *gsm* and *fft* applications shown in Figure 4.4(a) and Figure 4.4(b), respectively.

In Figure 4.4-a, we can observe that there exists a threshold at which the different cache configurations are clustered according to their size, independently of the degree

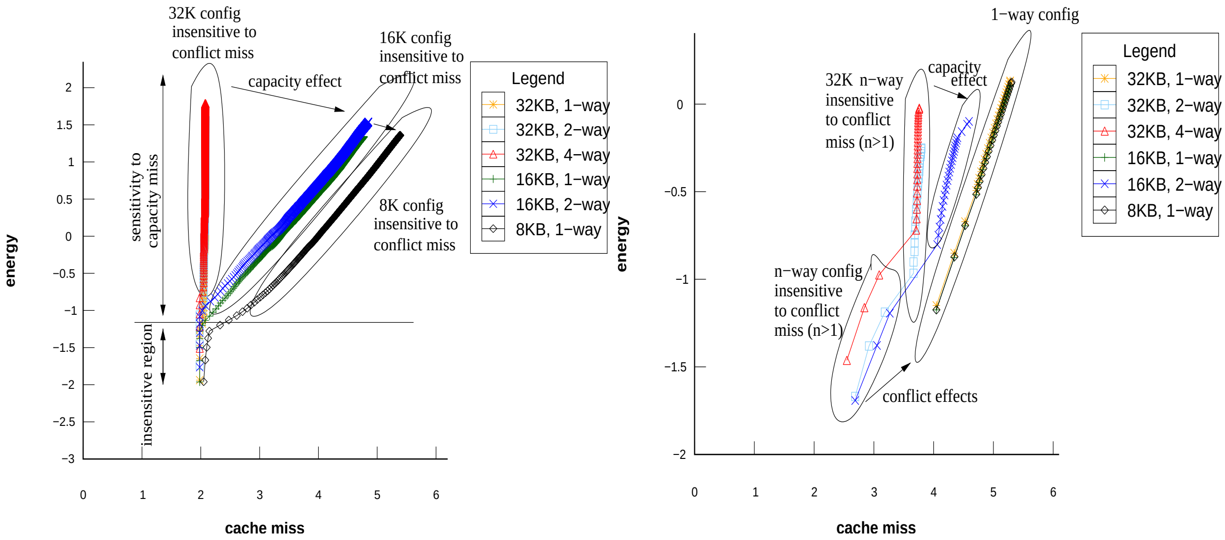


Figure 4.4: (a) gsm energy/performance profile; (b) fft energy/performance profile.

of associativity. In particular, we can distinguish three clusters: one with 32K configurations, another with 16K configurations, and the last with the 8K configuration. Configurations that belong to the same cluster are insensitive to the degree of associativity. Within each cluster, each cache configuration distinguishes itself from the other by the dynamic energy consumption due to the cache hits since the miss ratio is nearly the same. In such a case, the amount of dissipated energy is tightly coupled to the architecture of the cache configuration and is mostly a function of the cache size. A desired energy/performance tradeoff can then be achieved by moving from one cluster to the other, as indicated by the arrow shown in that figure. This comes at the cost of some performance degradation.

In Figure 4.4-b, we can also observe that two main cache clusters can be distinguished: one including the n-way cache configurations with $n > 1$ and the other including the direct mapped cache configurations. These two clusters differ essentially in the degree of associativity. As the program execution proceeds, the increasing effect of the capacity misses forces the first cluster to be further splitted into two distinct clusters: one which includes the n-way 32K cache configurations with $n > 1$ and the other with the 2-way 16K cache configuration. In this case, the clusters are splitted according to the cache capacity, independently of the degree of associativity. Again, in each cluster, the energy consumption due to the hits also serves as the main distinguishing factor.

We have observed that this property is rather common to most programs, as it can be seen in Figure 4.5. These examples prove that the dynamic working set of a program can be arranged so as to take benefit of the inherent working set sensitivity to the conflict or capacity miss, to save more energy. This property is exploited in the next section to construct regions of different energy/performance tradeoffs.

4.7 Approach for managing reconfigurability

The objectives of achieving a partitioning of the application's working set into clusters of cache configurations may be mainly motivated by two facts. First, there is the need of keeping the number of reconfiguration points smaller enough in order not to impact the performance and the energy. In the worst case, the cache may be reconfigured at the beginning of each sample interval, which is unacceptable especially if too much unnecessary reconfigurations take place (e.g. each one for each sample point). Finally, the performance and the energy consumption may be also impaired by the excessive number of inserted reconfiguration instructions - which may grow the code size and raise the energy consumption - and the additional number of cache misses induced by changing cache configurations.

The partitioning is done relative to the base cache configuration. In particular, a cluster of cache configurations with identical sensitiveness to the conflict/capacity miss is constructed from two cache configuration points B and C if each one of them belongs to the closest vicinity of the other, with respect to a reference point A of the base cache configuration.

Let us consider C_{base} and C_k as being the set of values collected at each sample interval Π_i for the base cache configuration C_{base} and for each simulated cache configuration C_k, respectively. The expressions of C_{base} and C_k are defined as follows:

$$C_k = \{(P_i^k, E_i^k), 0 < i \leq N\} \tag{4.3}$$

$$C_{base} = \{(Pbase_i, Ebase_i), 0 < i \leq N\} \tag{4.4}$$

In the above expressions, N represents the number of sample intervals Π_i. In order to partition the working set into similar sensitive cache configurations, we use a Manhattan distance vector V_k, as follows:

$$V_k = (v_1^k, v_2^k, ..., v_N^k), \tag{4.5}$$

where

$$v_i^k = |P_i^k - Pbase_i| + |E_i^k - Ebase_i| \tag{4.6}$$

Two cache configuration points (P_i^{k1}, E_i^{k1}) and (P_i^{k2}, E_i^{k2}) belong to the same cluster if their respective Manhattan distance value is related by the following relation:

$$|v_i^{k2} - v_i^{k1}| < \tau \tag{4.7}$$

τ is a threshold value that is used to decide when to cluster or not. Once the cache configuration points have been clustered into partitions of equal sensitiveness, each partition is chosen a representative cache configuration based on the best performance to energy ratio of each cluster of cache configurations.

The performance to energy ratio of each cluster is computed based on the value of the last sample point ($i = N$). The idea is to capture the relative amount of performance degradation corresponding to a given energy budget. For this, the ratio $\frac{P-Pbase}{Ebase-E}$ of each cache configuration belonging to a cluster is evaluated against each other. A smaller ratio is preferred since this would imply that for a given power budget, the performance is better. Then, for two clusters that span the same working set size, we choose the cache configuration of the representative partition element which has the best performance to energy ratio. The ISA instruction introduced in Section 4.3.4 can then be inserted at the appropriate working set frontier to enable the corresponding cache configuration.

4.8 Results

This section presents the results obtained by evaluating the proposed cache resizing scheme. The evaluation discussed in the remainder of this section is centered around three different performance aspects: the dynamic energy reduction, the leakage energy reduction and the performance degradation estimated in terms of increased total cycle counts.

4.8.1 Dynamic energy reduction

The leftmost side of Figure 4.6 shows the dynamic energy consumption results for the proposed cache resizing scheme. For the purpose of comparison, we also evaluated the dynamic energy consumption of the best performing cache configuration. The best cache configuration can be configured once before application's execution. Therefore, this configuration models the approach proposed by Zhang [117]. It can be seen from the figure that the proposed hybrid scheme can indeed reduce the energy consumption in some cases. Looking precisely at them (*gsm*, *susan*, *summin*, *mpeg*, *epic* and *v42bis*), we observe that they correspond, to some extent, to the cases where there exists a source of working set size variation in the program execution. This mainly explains why the working set can be ideally partitioned into clusters of different cache configurations. In these examples, the energy consumption can be further reduced from 5 to 12% compared to the best performing cache configuration. However, in the other cases where the working set shows little or no variation, the proposed hybrid scheme provides no benefit. This is the case of *fft* and *whestone*.

4.8.2 Leakage energy reduction

We estimated the leakage energy, E_{Leak}, of a program as follows:

$$E_{Leak} = e\_leak_i * Ncell * T_{cyc}$$

In the above expression, e_leak_i represents the leakage energy per cell for each one of the simulated leakage reduction technique i, Ncell the number of cells in the cache and T_{cyc} is the total number of cycles to execute the given program. T_{cyc} is computed as the sum of the number of cycles required to execute the program without any data cache stalls, plus the estimated data cache miss times the miss penalty. Figure 4.7 illustrates the leakage energy of the cache configurations show in Figure 4.6, left. We calculate the leakage energy of the best performing cache configuration by employing a gated-Vdd-based technique. It can be observed from the figure that the proposed hybrid scheme can reduce the static energy by more than 80%. This is a substantial reduction since the leakage energy of future caches generation are predicted to consume as much as 50% of the total power consumption [118]. The advantages of the hybrid scheme are best highlighted on *fft*, *gsm*, *susan*, *summin*, *epic* and *v42bis*. The best cache configuration is however superior to our scheme whenever the capacity of the cache can be reduced over the entire program run. This is the case of *whestone*. In this latter example, the gated-Vdd scheme considerably reduces the static energy compared with the drowsy mode. However, because this case is not the common, we believe our proposed hybrid scheme offers a more flexible alternative for many other applications.

4.8.3 Performance degradation

We evaluated the performance degradation in terms of the number of additional clock cycles required to execute a program. The simulated results are shown in the rightmost side of Figure 4.6. The primary causes of performance degradation in the proposed scheme are due to the one cycle delay of the drowsy transitions and the cache misses induced by changing a configuration. Our results are relatively high in some cases because we actually have considered the worst case in which a drowsy transition may occur even within a single phase. This is indeed inherent to the architecture since two data addresses may eventually be mapped to different combinations of cache bank in the same phase, causing unnecessary drowsy transitions. This is mainly reflected in *susan* and *mpeg* where the degradations are the worst, 35% and 31% respectively. From within this additional number of cycles, more than 65%, in average, are due to the drowsy transitions, the remaining part being due to the additional number of cache misses. A more efficient solution will therefore consist in choosing the set of invariant cache banks that will remain active throughout a complete program phase. This solution provides the benefit of eliminating the superfluous drowsy transitions, but at the cost of increasing the number of cache misses due to the invalidated data that may eventually be accessed in other configurations.

4.9 Related work

Our work is primarily concerned with research related to cache size adaptivity. In this sense, the work in [7, 114, 32] bear some similarities with our own. These researches share the particularity that some means of hardware adaptation scheme is required to allow a search of the optimal solution. In [7], the authors adapt the cache size of a L1/L2 or L2/L3 memory hierarchy in reaction to the sensitivity of a running program to some performance metrics collected dynamically, involving the IPC, the cache miss ratio and the branch frequency. The authors rely on the selective-way architecture [1] to accordingly enable/disable cache ways. This work is intended to general purpose systems featuring several levels of cache memory hierarchy. The proposed cache resizing algorithm can however be used as well in the context of embedded systems, provided that some means of dynamic performance monitoring is available. In the same order, [32] reformulates the cache resizing adaptivity algorithm of [7] to use instead working set signatures, to capture phase changes and to estimate the size of a working set. This solution however also requires an extra hardware effort. Yang et al. [114] proposed a work similar to ours. They rely on the cache resizing schemes of [1] and [115] to propose a hybrid cache of superior resizing granularity than either one of them. The hardware implementation cost of the selective-set scheme is however very expensive to be integrated on a embedded system. In addition, the proposed hybrid cache has a more restrictive cache resizing granularity for direct-mapped cache configurations (8K in the paper for a cache size of 32K). In addition to this, we should notice that our work primarily addresses embedded systems. We seek therefore a low-cost, software-based cache resizing adaptivity scheme. Though our objectives are the same, the different application domains impose us to look for different solutions.

Zhang et al. [117] also presented how the way-concatenation scheme could be used together with a selective-way-based scheme to further reduce energy. The main difference between our approach and the one proposed by Zhang is essentially on the applicability of cache resizing. Zhang et al. look for the cache configuration that gives the best performance on a per-application basis, and therefore only configure the program once at startup time. Thus, the variations in cache size requirements within the running program are not taken into account. We seek instead to adapt the cache memory to meet the dynamic cache size requirements of the running program. This difference infers some divergences in the implementation decisions of the selective-way scheme. Zhang et al. propose to use a circuit technique called gated-Vdd [91] to implement their selective-way scheme. Gated-Vdd permits to reduce the cache leakage energy by gating off the supply voltage to the unused cache memory cells. However, this technique does not preserve the memory cell state, causing the stored data to be loss. This, in addition to the coherency problem we addressed in Section 4.3, are serious hindrance to make their scheme reconfigurable on a per-phase basis.

4.10 Summary

This chapter has proposed to modify the structure of a configurable cache in order to offer embedded compilers the possibility to reconfigure the underlying cache memory according to the cache size requirement of a dynamic program phase. The objective followed was the reduction of the switching activity on the cache subsystem. We showed that with the proposed cache resizing scheme, some reduction in the dynamic and static energy can be realized. In essence, we proved that this energy reduction is significant for applications showing a dynamic working set size variation. In particular, we showed that in such cases, the application's working set can be classified according to a program property we called conflict/capacity miss insensitiveness. We presented simulation results that demonstrated this property is rather common with most programs. In the light of this model, we explored a compiler strategy that may take advantage of this property to partition the application's working set into clusters of similar cache sensitive configurations that can save more energy.

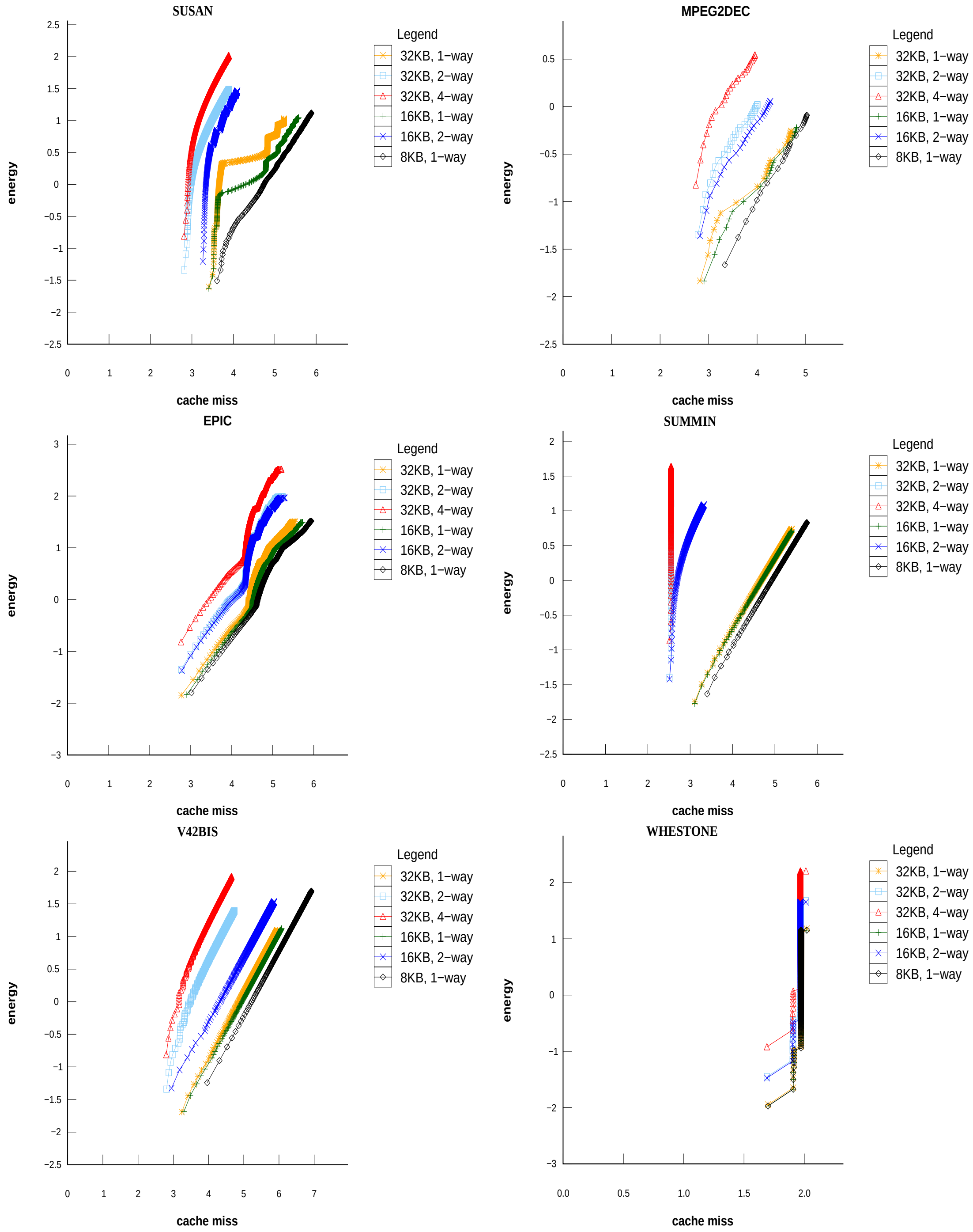


Figure 4.5: Energy and performance profiles

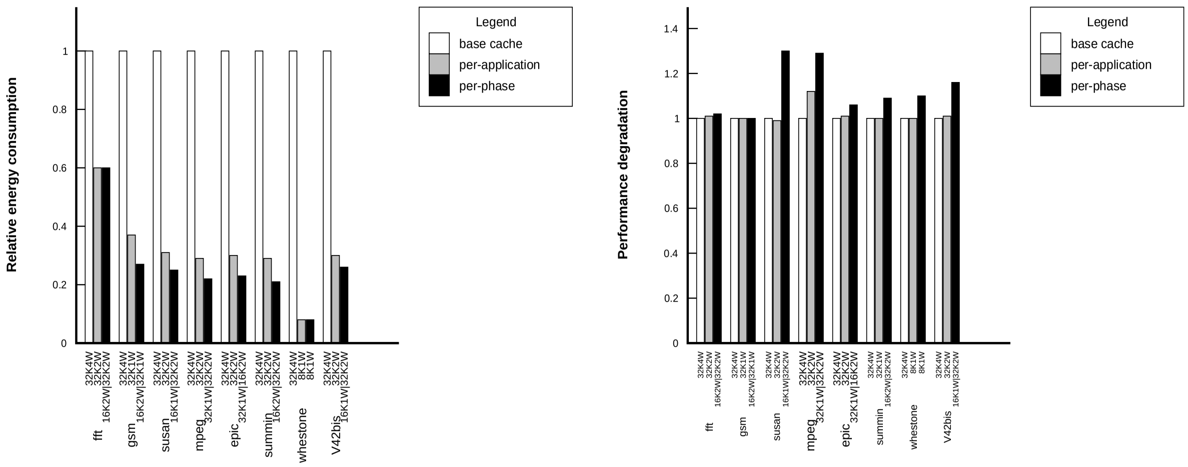


Figure 4.6: (left) Dynamic energy consumption; (right) Performance degradation.

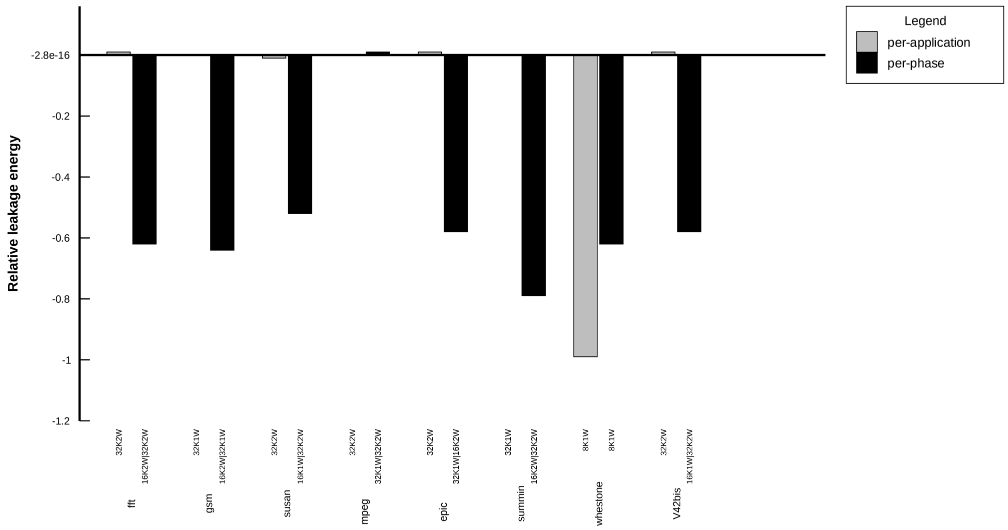


Figure 4.7: Relative leakage energy compared with the base cache.

Chapter 5

Power-efficient reconfigurable processor datapath

With the trend toward high-performance, new generation processors have evolved to support aggressive architected features to reap maximum performance. This trend has also been accompanied with a continuing increase of the bit-width size, principally because of the need of larger addressing space and memory bandwidth. Whereas in many case this trend had settled on 32-bit wide datapath, it is now keeping growing, supporting up to 64-bit datapath width for new generation processors such as the Itanuim. This bit-width growth has been set forth to benefit first to the vast majority of applications, which until recently were largely dominated by scalar processing on 32-bit integer data types. However, with the recent confluence of general purpose and multimedia applications on modern embedded processors, this is no more the case, since many of these latter applications operate on narrower data widths, e.g. 8- and 16-bit data. Brooks et. al. [18] have recognized this opportunity. They observed that with a 64-bit Alpha-like processor, more than 50% of the instructions had their operands with 16-bit or less while executing the MiBench programs suite [42]. A significant fraction of the processor's power-efficiency is thus wasted when operating with these narrow-width operands.

Many techniques have been deployed to reduce the energy consumption in modern processors [109]. However, using bit-width optimization as a means to tackle this problem is a relatively new topic of research and only few works have already emerged in this direction. Basically, employing bit-width optimization for the purpose of energy reduction requires that a few common operations be implemented, which involve:

(1) detecting the useful bit-width of the operand and,

(2) clock-gating the insignificant bit portions of the operand being operated on by a functional unit.

A taxonomy of bit-width-based optimizations can then be arranged according to the issues involved when considering the first operation. Schemes that rely on a form of compiler analysis to discover ranges of useful bit-width values [21, 105, 26, 69, 25] can be classified as software proposals, since the bit-width detection operation is done statically. On the other hand, schemes that rely on some means of architected feature to infer operand bit-width [18, 29, 24] can be deemed as hardware proposals.

Problematic and approach Software approaches offer a low-cost solution to bit-width-based optimizations and can therefore be easily adapted to existing embedded systems. However, due to their inadequacy to exploit the dynamic variations of operand sizes, they may also miss a substantial number of bit-width optimization opportunities. In contrast, architectural approaches are designed to take this variation into account and can therefore provide a more accurate solution to (1). However, these architectural schemes also require complex hardware mechanisms that are often not affordable in the embedded system context. As an alternative, we propose in this chapter to explore a new synergistic hardware/software approach for better exploiting an embedded system with narrow-width data. The principal idea of this chapter is reducing the power consumption of a processor by minimizing the amount of switching transitions on the datapath. Energy is saved because few switching transitions may occur in the datapath as it is resized to lower bit-width sizes. To do so, we assume that the ISA is augmented with an instruction indicating that the subsequent code might be executed through a narrower path; thus requiring only narrow datapath and narrow register operands. This instruction is just a hint. Then, at hardware-level, a simple exception management allows to recover instructions executing with full datapath-width on an incorrect hint. In this sense, we introduce datapath-width speculation principally as a means to predict, at the software level, the operand bit-width of certain program regions; this in order to anticipate the reconfiguration of the processor resources. We show that our proposed scheme enables lower hardware complexity, while exploiting the full software potential for bit-width detection.

Chapter contributions The contributions of this chapter are two-folded. First, we provide evidence that there exists static code regions corresponding to dynamic program instances, where the vast majority of the operands execute with a narrow-width. We then present techniques to uncover these regions at code generation time. Second, we present the architectural support that exploits these regions at runtime. The idea is to reconfigure the datapath width as well as the register file width when such a narrow-width operand region is encountered. Central to our approach is the *speculative nature* of the execution width of a region: we present a simple and efficient recovery mechanism for handling such width mispredictions.

Chapter organization The remainder of this chapter is organized as follows. Section 5.1 discusses the prevalence of narrow-width operands in programs and motivates this research. In Section 5.2, we review software and hardware approaches for exploiting

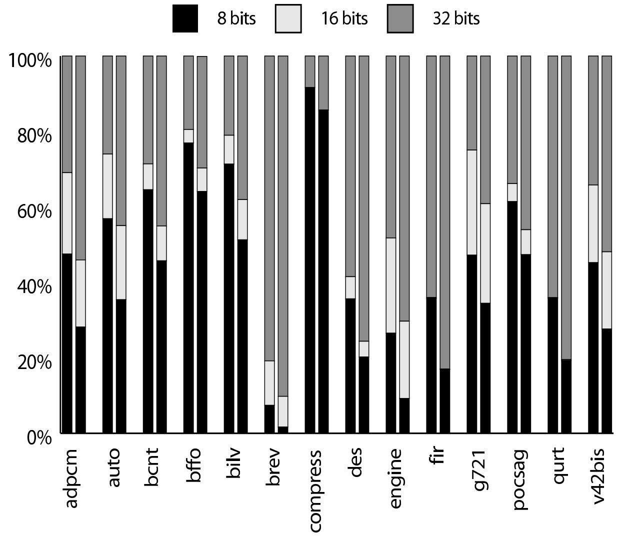


Figure 5.1: Cumulated distribution of operands bit-width. The first bar shows results for one operand; the second bar shows results when both operands are considered.

narrow-width operands. In particular, we discriminate among the issues that make these techniques less attractive for high-performance VLIW processors. We show evidence of narrow-width operands regions in Section 5.3. The architectural support that permits to exploit these narrow-width operands regions is detailed in Section 5.4. Then, in Section 5.5, we present our strategy to detect them at code generation time. Our solution overview is detailed in Section 5.6. Our methodology and experimental results are presented in Section 5.7 and Section 5.8, respectively. Section 5.9 concludes this chapter.

5.1 Prevalence of narrow-width operands

Brooks et. al. [18] were the first to emphasize the availability of narrow-width operands in programs. They conducted their experiments with a 64-bit Alpha-like processor. We have performed equivalent experiments on typical embedded applications, e.g. Powerstone benchmarks [95], running on a 32-bit RISC-like embedded processor [35]. The results shown in Figure 5.1 illustrate that, on average, 45.5% of the instructions have their operands with less than 16-bit or equal. This is already in accordance with Brooks's estimations, which found that about 50% of the integer instructions execute with narrow-width operands in multimedia applications running on a general purpose system.

Unlike a general purpose system, however, in an embedded system, the compiler plays a central role in achieving high performance. It is therefore of importance to improve the compiler's effectiveness to manage both power and performance. Since the basic block is the natural compiler granularity, we introduced the possibility to master

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

Figure 5.2: C code example.

narrow-width operands at the basic block level. At this granularity, the compiler can even achieve better energy/performance tradeoff, rather than relying solely on the hardware, since much more hardware components can be "turned off" over a longer period of time. Moreover, in contrast to the dynamic approach proposed in [18], considering bit-width regions at the compiler level provides the additional advantage of reducing the overhead due to clock-gating on a cycle-by-cycle basis.

5.2 Approaches for exploiting narrow-width operands

In this section, we review some of the approaches often deployed to take benefit of narrow-width data. We mainly focus on the approaches that try to reduce the energy consumption using bit-width optimization.

5.2.1 Exploiting narrow-width operands via SIMD compilation

Exploiting the processor's datapath with short data-types is not a new topic of research. The SIMD programming paradigm has been introduced primarily as a means to take advantage of the full processor's data-width for improving the multimedia performance [92, 59, 87]. As a side-effect of applying SIMD techniques, some studies have shown that energy consumption can also be reduced [33]. This can be primarily attributed to the reduction in the number of executed instructions. Many issues however make the exploitation of SIMD techniques very difficult to realize. In particular, the natural approach to exploit SIMD compilation is to rely on vectorizing technology as a basis for discovering SIMD parallelism. Since many instruction set imposes some constraints on memory data alignment, an alignment analysis phase is often added to take this issue into account. Finally, recognizing the right instruction to use is a difficult task because many multimedia instructions implement complex operations dedicated to computation intensive kernels such as FIR filters. For instance, the TriMedia [34] instruction set offers an instruction capable of simultaneously computing four averages on bytes data.

5.2.2 Exploiting narrow-width operands in software

Most of the recent compiler research devoted to bit-width analysis rely on dataflow analysis techniques to determine the maximum bit-width of operands. In order to do so, the compiler must compute the possible ranges of bit-width values a variable can take during execution, while remaining overly conservative to preserve program correctness. An algorithm reminiscent to constant propagation is used to propagate initial value ranges over the program control flow paths, along with the corresponding refinements obtained at each point of variable definitions. Variants of this algorithm have been implemented in the literature [105, 26, 69, 25]. In either case, the initial value ranges are collected for some set of variables and forward and backward compilation passes are applied iteratively in order to propagate the refinements obtained at each point of definition to the next point of use. This is done until no changes in the value ranges of a variable occur or after that some fixed number of iterations have elapsed.

However, since the purpose of these algorithms is to assign the *maximum* bit-width range a variable can take during execution, they may miss a significant number of optimizations opportunities. In order to evidence this, let us, for instance, consider the piece of code shown in Figure 5.2. This code illustrates the limitations of a bit-width analysis approach regarding the representation of the variable x. In the example, the variable x can be represented with at most 16-bit for a correct execution. A compiler analysis technique will therefore adjust the variable bit-width size to a more appropriate bit-width, i.e. 16-bit. Nevertheless, as depicted in Figure 4.3, the variable x requires most of the time (i.e. 88%) only one byte for executing. However, since the bit-width transformation must preserve program correctness, it may not be possible to adjust the bit-width size to less than 16-bit; thus missing a non-negligible amount of bit-width optimization opportunities, as already pointed out in [18].

5.2.3 Exploiting narrow-width operands in hardware

As highlighted in previous section, software approaches exclude a significant portion of bit-width optimizations since they must preserve program correctness. By contrast, hardware schemes permit to cover the entire bit-width potential since the size of data are detected at runtime. Some works have been proposed in this sense [18, 29, 24] that dynamically exploit narrow-width operands. The main distinction between these latter lies in the manner by which narrow-width values are recognized.

In the approach proposed by Brooks et. al. [18], a narrow-width operand is detected upon evaluating the high-order data wire signals to zero. This evaluation is performed each time an ALU result is produced. Depending on the outcome of the evaluation, a zero-bit signal is generated and stored along with the result operand whose upper bit values are provided either via hardwired zeros or via the result bus, through the use of a multiplexor. By reading an operand from the register file, this signal is used to selectively latching the upper bits of the operand; thereby saving energy in both the latches and the functional unit. Apart the zero detection logic that the authors assume

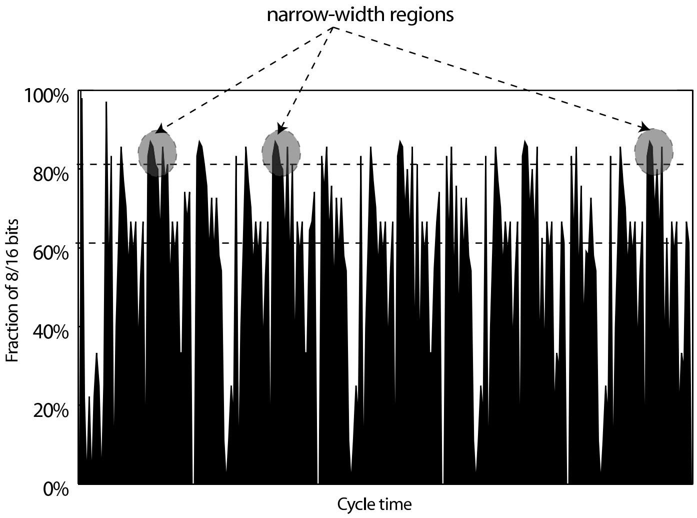


Figure 5.3: Dynamic bit-width distribution at the granularity of a basic block for *adpcm*.

being available in modern machines, this approach can as well be applied in our context.

The approach proposed by Choi [29] is to some extent similar to that of Brooks. The ALU is divided into a least significant part and a most significant part, the latter part being activated via a guarded latch whose control falls to a zero detection logic that acts every time an operand is presented to the ALU input latches. Compare to [18], the detection logic operates just before the ALU computation takes place; therefore adding to the delay and clock cycle time.

In [24], Canal et. al. have considered a byte-serial (8-bit) or a semi-parallel (16-bit) pipeline to exploit narrow-width data at the architecture level. The idea relies on appending extension bits to data residing in caches and registers in order to reflect which part of the processing data is significant. Only the useful bytes are loaded, stored or computed on, and therefore a significant fraction of the switching activities can be reduced. However, the fixed nature of the processor's datapath incurs a high performance penalty when processing operands of a larger bit-width, e.g. 32-bit or more. This performance degradation can simply not be afforded on performance-critical systems.

5.3 Narrow-width operands distribution analysis

Through profiling, we collected some statistics on the width of operands for applications from the Powerstone benchmarks suite [95] on various input data sets. For instance, Figure 5.1 illustrates that narrow-width operands can be of a large number on *adpcm*. In this section, we consider their availability at the basic block level and we propose to examine their distribution across a program run.

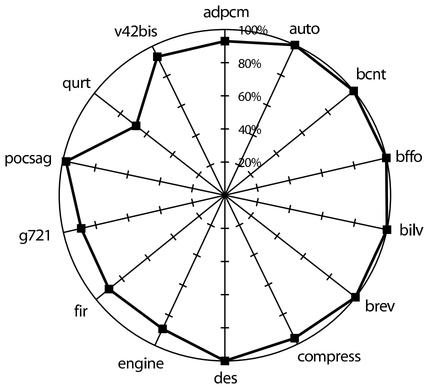


Figure 5.4: Average operand bit-width convergence.

Figure 5.3 captures a snapshot of the dynamic operand bit-width profile of the *adpcm* benchmark. Each point of the x-axis identifies a dynamic instance of a given basic block, while the value associated with the y-axis represents the availability of the narrow-width operands within that basic block. It can be seen from the figure that a sufficiently large number of basic blocks execute with more than 60% of their operands having 16-bits or less. This last point may lead to two main observations. First, this indicates that a strong narrow-width operand locality exists for the considered granularity. Second, this suggests that executing these basic blocks on a narrower datapath-width may increase the compiler opportunities for saving the energy. Hence, we may try to take advantage of this to speculatively accommodate the width of the processor's datapath to the operand's width of a basic block or region.

However, before we may exploit this fact, we must ensure that basic blocks exhibiting such a behavior verify a property we call *bit-width convergence*. Bit-Width convergence refers to the fact that, for a given basic block, its operands width may not vary frequently enough during execution. The rationale behind this property is to prevent the compiler from optimizing on very sensitive narrow-width operands regions. We estimated the bit-width convergence in the following manner. When a basic block is found to execute with 16-bit or less (according to a defined threshold), we record for each future execution of the same basic block the number of times we are wrong. We then average this value on all the basic blocks of concern. This provides us with an estimate of the average bit-width convergence for a given application. Typically, a high value indicates that bit-width transitions occur very infrequently from one dynamic instance of a region to another. The results of the bit-width convergence, considering an 80% narrow-width operands availability, are shown in Figure 5.4. On most applications in our benchmarks set, basic blocks execute with constant operand's bit-width on our data set inputs.

5.4 Architectural support for speculative execution

In this section, we examine a potential architectural support for exploiting narrow-width operands regions. One beneficial approach may consist in reducing the pipeline's activity while achieving acceptable performance. Therefore, we present a reconfigurable architecture that may dynamically adapt itself to an application's bit-width behavior.

5.4.1 Hardware-exposed datapath-width reconfiguration instruction

In order to benefit from narrow-width data elements at the software level, we propose to enhance an ISA with an hardware-exposed datapath-width reconfiguration instruction. The effect of this instruction can be deemed only as a hint to predict the execution width of subsequent regions. Via the use of this instruction, the compiler may speculatively cause the execution of a region to accommodate on a narrower datapath-width (8-bit, 16-bit or 32-bit mode). Then at runtime, a simple hardware-based exception mechanism will allow to recover instructions executing with full datapath-width in case of a misprediction.

5.4.2 Reconfigurable register file model

5.4.2.1 Related Work

Previous research on reducing the register file activity focused on either, limiting the number of registers [8] or, limiting the number of ports [110]. Only few studies attempted to capitalize on narrow-width data for the same purpose. Canal et al. [24] proposed to load, store or compute only significant bytes in the whole pipeline stages. To do so, they designed a byte-serial pipeline where the data are processed on 8-bit slices. In order to provide this 8-bit access, they considered a 32-bit register file partitioned into 8-bit banks. In their study, as only one bank is requested per cycle, this multi-banked approach permits to reduce the register file activity. In contrast to their work, we are considering a data-path that is dynamically resizable according to the application's needs. As a matter of fact, in a multi-banked model, the row decoders are replicated on each bank. Therefore, accessing a wide data would generate redundant decoding and thus, useless power consumption.

5.4.2.2 Our Approach

We introduce a novel register file organization, the *byte-slice* register file. This energy-aware design permits to dynamically resize the register file width so that it can be viewed as a 8-, 16- or 32-bits conventional register file, as depicted in Figure 5.5. The register file is logically splitted into three slices: the first slice, representing the low-order data byte, is always enabled, whereas the others are controlled by means of a "slice-enable" signal. In our scheme, at anytime, the registers can hold different bit-width data; and

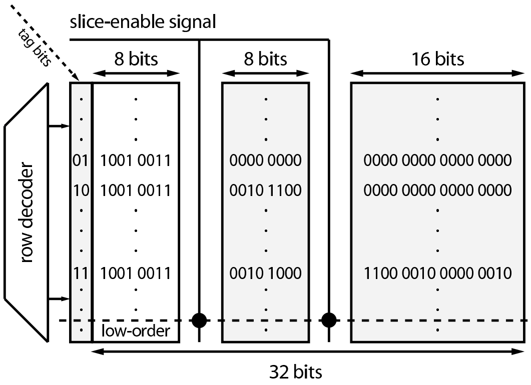


Figure 5.5: Byte-slice register file.

thus, it is not possible to turn off unused slices, unless there is a way to recover the lost informations. Considering this fact, the slices are turned off in a *low-power* mode, achieved by the drowsy state [38]. In order to support such a state, we assume that the memory cells are modified as described in [4]. Technically, the drowsy circuitry is a state-preserving circuit that relies on voltage scaling for leakage reduction. A slice in a low-power mode preserves its data, although, it must switch back to the normal mode to get the correct information. The tag bits illustrated in Figure 5.5 provide this feature. We will get back to this later when we will discuss the recovery mechanism.

In contrast to [24], the adaptability and the simplicity of the *byte-slice* concept provide the advantages of being well suited to dynamically reconfigurable pipelines. In addition, the drowsy circuitry, which represents only a small area overhead [4], makes our design inherently low-power.

5.4.3 Reconfigurable datapath

In this section, we describe a power-effective pipeline that may take advantage of the narrow-width regions. As depicted in Figure 5.6, the datapath has the ability to adapt to the bit-width behavior of an application. This reconfigurable aspect is done via the clock-gating technique [18]. Clock-gating is a well-known scheme used to reduce the dynamic power consumption in today's processors [72]. In our approach, the coarser clock-gating granularity (at region level) reduces the amount of dynamic power dissipated by the clock-gating circuitry [5].

5.4.4 Recovery mechanism

To tackle the disadvantages of a static compiler analysis, as pointed in [18], we propose to statically construct narrow-width regions by using runtime informations. In order to increase the number of these regions, we also consider the ones that verify the *bit-width convergence* property; thus introducing datapath-width speculation. However, since

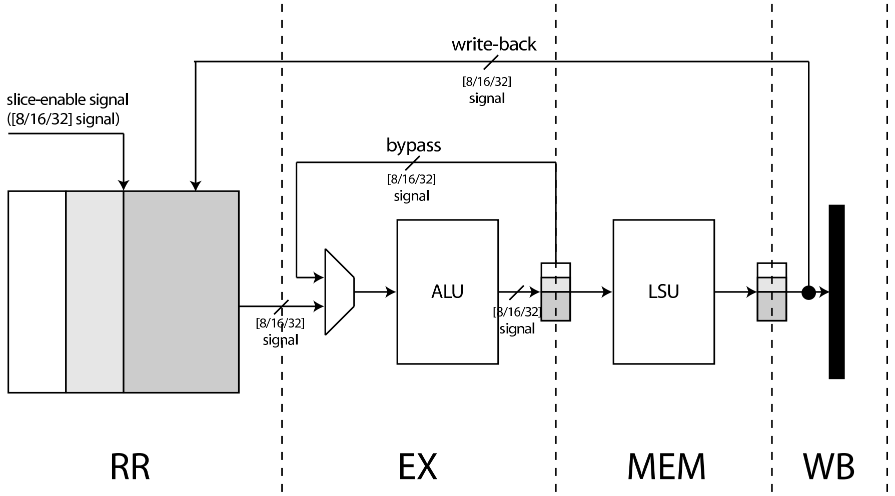


Figure 5.6: datapath.

it is not realistic to profile each application for each input data, or due to a dynamic event, a datapath-width misprediction may occur. In this section, we present a recovery mechanism that identifies the malformed regions and acts accordingly.

The main idea is to use a few tag bits to decide whether the current narrow-width region has been correctly predicted. In this respect, we use two tag bits appended to each register (see Figure 5.5) in order to discriminate between the different datapath modes (i.e. 8, 16, 32-bit mode). With a 32-bit width register file, this represents a negligible area overhead, with only 6% of the area being devoted to the tag bits. These tag bits reflect the true data-width and are generated by the functional unit, upon completion of an operation, and by the memory unit, upon a load instruction. [24] uses a similar scheme, however, we employ the tag bits in a different manner. While in [24] they act as a way to serialize the execution, in our proposal the tag bits dictate the use of the recovery mechanism.

The flow chart shown in Figure 5.7 illustrates the basic concept of this recovery mechanism. When an instruction reads its source operands from the register file, both the data and the tag bits are fed to the functional unit. A simple comparison logic, located at the execute stage, detects whether the current operating mode is correct or not. If it appears that the current mode is narrower than the one expected, the current instructions are replayed, i.e. the pipeline is flushed and the correct width is enabled. When an instruction produces a result larger than the current mode, the pipeline is stalled while switching to the correct width. Although this mechanism may relatively impact on performance, its hardware simplicity fits well into the embedded context.

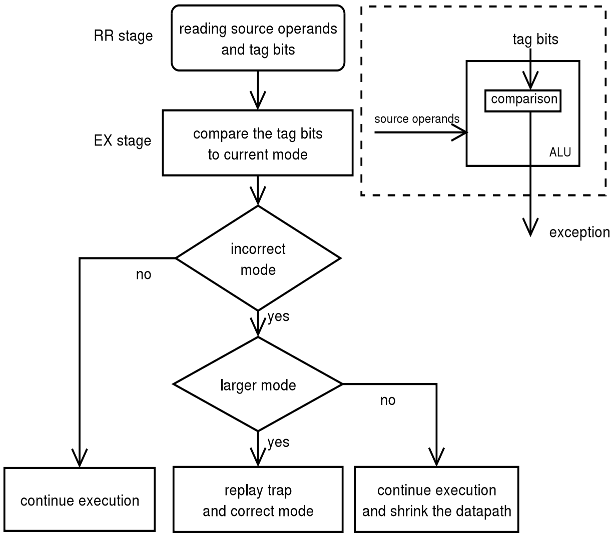

RR stage
reading source operands and tag bits
EX stage
compare the tag bits to current mode
incorrect mode
no
yes
larger mode
no
yes
continue execution
replay trap and correct mode
continue execution and shrink the datapath
tag bits
comparison
source operands
ALU
exception

Figure 5.7: Recovery mechanism.

ISA extension	Description
MOVACC Reg	ACC ⇒ Reg
MOVREG ACC	Reg ⇒ ACC
LDACC Reg	(ACC) ⇒ Reg
STACC Reg	Reg ⇒ (ACC)
ADDACC Reg	Reg + ACC ⇒ ACC
SUBACC Reg	ACC - Reg ⇒ ACC

Table 5.1: Basic address instructions.

5.4.5 Handling address instructions

Address instructions, e.g. *load* and *store*, must be handled separately, since they usually require a larger bit-width to represent memory addresses. We may address this problem by using a dedicated register file for memory addresses, in a way which is reminiscent to a decoupled architecture approach [102]. This feature is already integrated on some modern embedded processors [74]; they may therefore directly benefit from our scheme. For the processors that do not provide support for this feature, we suggest using special purpose registers, e.g. accumulator registers, for hosting and computing memory addresses. Along with the accumulator registers, the ISA must also permit the data transfers between the register file and the accumulators. Table 5.1 shows a possible subset of basic instructions that must be provided to support this scheme. In the table, the load and store instructions must have their base address residing in an accumulator register. The arithmetic instructions might be needed for computing new addresses.

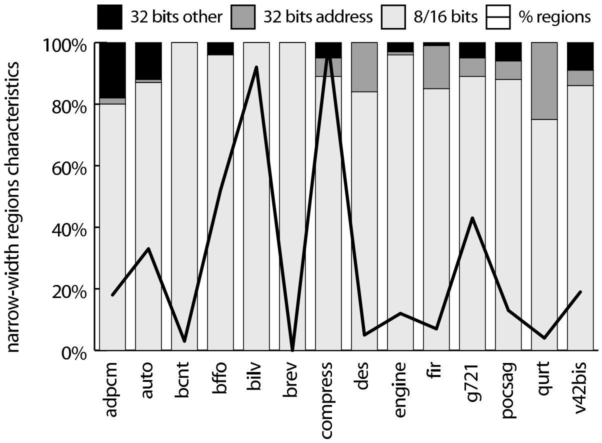


Figure 5.8: Average narrow-width regions characteristics (bargraph) and regions representativeness in program (linepoint).

5.5 Detecting narrow-width regions

Having analyzed the distribution of the narrow-width operands and the relative bit-width convergence of the regions, this section discusses the formation of the narrow-width regions in the back-end compiler.

5.5.1 Selecting candidates regions

In selecting the candidates regions, we may be forced to leverage the availability of the narrow-width operands against the probability that a bit-width misprediction occurs at runtime. This might be primarily due to the fact that only a few regions would be able to exhibit narrow-width operands exclusively. This phenomenon can be indeed observed in Figure 5.3, where no perfect candidates regions can be found. Therefore, these regions may be chosen according to an arbitrary narrow-width operands availability, first ignoring the constraints due to the wide data. We assume for the rest of this study a threshold at 80%, which corresponds, on average, to one instruction out of five that executes with at least one 32-bit operand. Under this consideration, Figure 5.8 illustrates the average bit-width profile of a narrow-width operands region. The figure reveals that some applications have perfect narrow-width operands regions, e.g. *bcnt* and *bilv*. Some others, however, include instructions with larger operand's bit-width. These are labeled in the figure with *32-bit other* and *32-bit address*. The former indicates the fraction of instructions, not counting the memory instructions, having one of their operands with 32-bit. The latter represents the fraction of memory instructions. We may then attempt to build 32-bit-free operands regions out of these regions.

5.5.2 Regions transformation

It is explicit from the previous section that building a narrow-width operands region implies to deal with the 32-bit operands instructions. This section discusses a technique to efficiently overcome this problem.

Graph partitioning

By assuming that we have a means to deal with address instructions separately, e.g. accumulator registers, the problem to which we are confronted at this stage may be viewed as a graph partitioning problem. Let the graph G denotes the data dependence graph confined to a basic block. A node N of G represents a basic block operation. Two nodes, N and M, of G are connected via an edge e if there exists a ***def-use*** relationship among them. The graph partitioning problem consists in selecting the set of load/store nodes having one of their operand with 32-bit, in order to replace them with equivalent accumulator-based instructions, while minimizing the cut-size. The cut-size may be viewed as the number of additional instructions needed to move the data between the accumulators and the register file. This latter must be kept small enough in order not to impair the performance and the energy. We use a simple branch-and-bound heuristic to achieve this goal, deciding at each processing step whether or not the cut-size is within an acceptable range. Otherwise, the transformations are simply undone and the region is left unchanged.

Code restructuring

The problem that is pointed out in this section arises as soon as we have a narrow-width operands availability of less than 100% within a region. Let us assume that we are dealing with such a candidate region (a basic block). The problem to which we are confronted is to reorder the instructions in that region such that instructions having at least one operand with 32-bit (determined during profiling) are moved around it. The solution to this problem may be better illustrated in Figure 5.9.

A first operation consists in renaming all destination operands of instructions having one of their source operand with 32-bit, that may be used ahead of its computation. In this way, we augment the opportunities of finding more instructions that can be moved around. A second operation consists in computing the sets $MoveUp$ and $MoveDown$ corresponding to the 32-bit instructions that can be moved towards the beginning or the end of the basic block, respectively. Finally, a last operation consists in scheduling the instructions contained in each one of these sets upwards or downwards the underlying basic block, depending on the set to which they belong. Note that a side-effect of this algorithm may eventually cause some instructions to be duplicated if they are scheduled across a control flow graph join point. In addition, this might also lead to augment the pressure on the register file. This latter point can however be avoided if we consider a large register file, e.g. 64 general-purpose registers like that featured in our processor

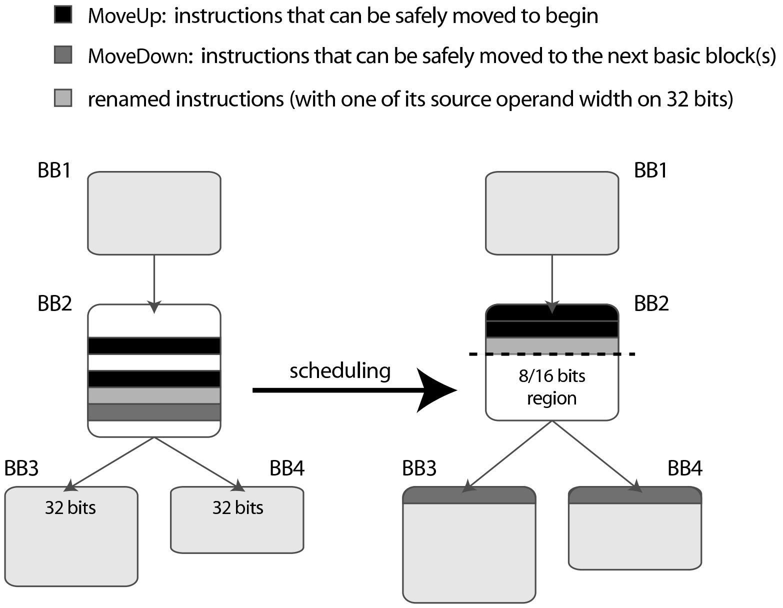

MoveUp: instructions that can be safely moved to begin
MoveDown: instructions that can be safely moved to the next basic block(s)
renamed instructions (with one of its source operand width on 32 bits)
BB1
BB2
BB3
BB4
32 bits
32 bits
scheduling
BB1
BB2
8/16 bits region
BB3
BB4

Figure 5.9: Bit-Width sensitive scheduling example.

model.

5.6 Solution overview

A synoptic view of our approach can be depicted in Figure 5.10. It consists of two main phases, a profiling phase and a narrow-width regions formation phase. In the first phase, the program is instrumented and stressed with different input data sets. At each time, statistics about the operand's width are gathered and stored for further utilization. The instrumentation is done by means of SALTO [16], which is a general, compiler-independent tool that makes the manipulation of the assembly code at the CFG level easier. In the second phase, the profiled data collected during the first phase are merged to create a converged profile for each application. From this profile, narrow-width regions candidates are initially identified by SALTO and then processed to create more refined regions. The reconfiguration of the processor datapath as well as the width of the register file is performed at runtime, every time the execution proceeds through a narrow-width region. For this latter to take place, we assume that the widths of the execution datapath and the register file are exposed to the compiler via explicit reconfiguration instructions (see Section 5.4.1).

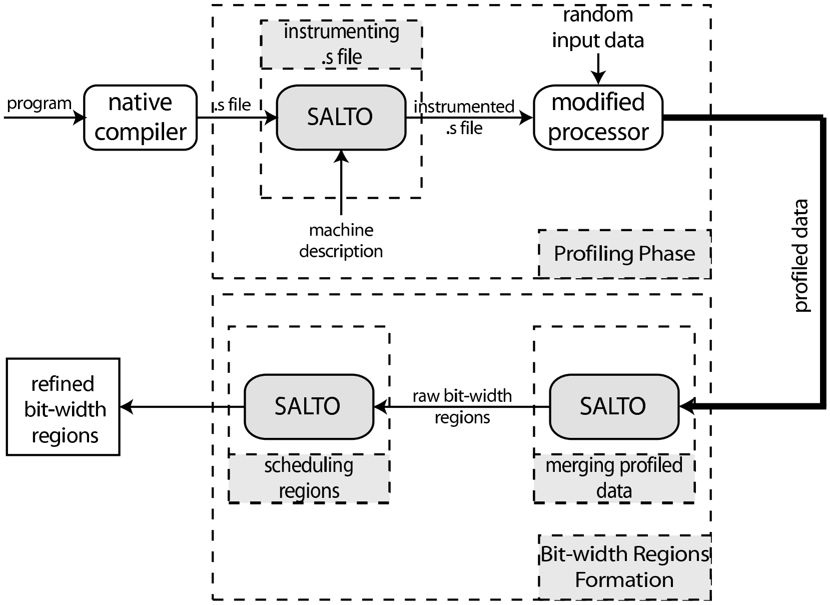


Figure 5.10: Optimization flow-graph.

5.7 Methodology

5.7.1 Platform and Simulation

Our experiments were conducted on a RISC-like, 32-bit embedded processor belonging to the *Lx* family of customizable, multi-cluster VLIW architectures [35]. The processor's implementation used in this study features a six-stages pipeline, 4-issue width processor composed of 4 ALUs, 2 Multipliers, and 1 Load/Store unit with in-order execution, on each cluster. The different pipeline stages model the instruction fetch (IF), the instruction decode (ID), the register read (RR), the first stage execution (EX1), the second stage execution (EX2), and the write-back (WB). There are 3 forwarding paths which are EX1-EX1, EX2-EX1 and EX2-RR. Each cluster provides a set of 64 32-bit general purpose registers organized in a monolithic conventional register file. A set of 8 1-bit registers are used as branch registers.

The Lx platform is provided with a software tool-chain, where no visible changes are exposed to the programmer. The tool-chain comprises, among other things, an aggressive ILP compiler, called the Lx compiler, from which we generate an input assembly. The extracted assembly code is processed by SALTO [16] as described in Section 5.6, to instrument the code and construct the narrow-width regions. The instrumented code is used to gather runtime statistics about register file access frequency, instruction's types, and operands bit-width.

Benchmark	**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

Table 5.2: Benchmarks.

5.7.2 Benchmarks

We evaluated our scheme with applications collected from the Powerstone [95] suite of benchmarks. All the chosen applications were compiled with the Lx native compiler, with the optimization level 3, and then run until completion. Table 5.2 provides an overview of each benchmark used.

5.7.3 Energy model

In order to have a rough estimate of the energy savings that one may expect to gain with our scheme, we must quantify the energy consumption that is due to the register file on one side, and to the various pipeline stages on the other side. Let us first consider the register file. We model the dynamic energy consumption of a register file, $E_{RF}^{(dyn)}$, as follows:

$$E_{RF}^{(dyn)} = N_{rw} * E_{access} \tag{5.1}$$

where E_{access} is the average energy consumption on a read/write access, and N_{rw} the number of read/write accesses to the register file. We used a modified version of CACTI [98] for estimating the values of E_{access}, for both a conventional register file, as well as for our *byte-slice* register file architecture.

We employ the expression shown in (5.2) to quantify the static energy consumption due to the register file.

components	**datapath energy**	**saving 16-bit**	**saving 8-bit**
latches	36%	18%	9%
ALU	27%	13%	7%

Table 5.3: Maximal energy savings.

Parameter	**Value**
Clock	1 GHz
nb of read/write ports	8/4
E_{access} (monolithic RF)	0.36 nJ
E_{access} (8-bit *byte-slice*)	0.11 nJ
E_{access} (32-bit *byte-slice*)	0.40 nJ
normal leakage power/cell	9.47 pW
drowsy leakage power/cell	2.34 pW

Table 5.4: Simulation parameters.

$$E_{RF}^{(stat)} = N_{cyc} * N_{cell} * P_{Leak} * \frac{1}{f} \tag{5.2}$$

In the above expression, N_{cyc} is the number of cycles needed to execute the program, N_{cell} the number of cells contained in the register file, P_{Leak} the leakage power consumption per cell and f the processor's clock speed. The term P_{Leak} is strongly dependent on the technology and may vary with transistor size, width and temperature. Assuming current process technology parameter of 0.18μm, we estimate P_{Leak} by means of Hotleakage [117], for both the normal and the drowsy modes.

Estimating the energy consumed by the other pipeline stages is a more difficult task, since very few processors vendors communicate detailed results about it. Nevertheless, we rely on power consumption estimates found in some research articles to derive realistic trends that govern the energy consumption of the involved processor's components. For our purpose, we are primarily interested on the energy consumption of the integer ALU and pipeline latches. In [112], the authors published energy results for a generic embedded processor, with a pipeline model very similar to ours. They noted that the obtained energy values were independent of the code being executed. One could deduce from this study that, on average, the register file and the pipeline latches account for ~64% of the datapath power consumption, with the former representing 28% of the power and the latter 36%. The remaining 36% is due to the datapath multiplexers and the ALU, with the latter contributing for more than 27%.

Since on a narrower bit-width mode, the clock-gating circuitry prevents the high-order bytes of a pipeline to be latched, we can expect that the corresponding energy savings will be proportional to the bit-width mode of the latch. Similarly, we save energy in the ALU structure by preventing its input latches from changing; thus restricting the computation to the low-order bytes of the input latches, yielding a linear reduction in

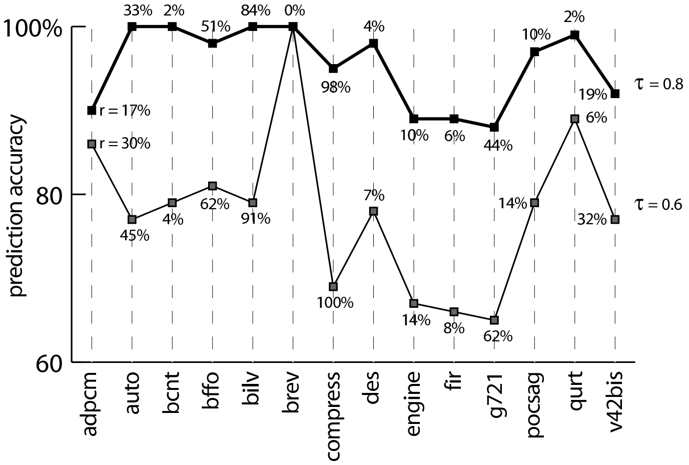


Figure 5.11: Accuracy.

the energy[1]. We summarize all the simulation parameters and the obtained ratios in Table 5.4 and Table 5.3.

5.8 Results

This section presents the evaluation results of the proposed narrow-width regions formation scheme. We center our discussion around four different aspects: the impact of the recovery mechanism, the code size growth, the dynamic energy reduction, and the leakage energy reduction.

5.8.1 Recovery mechanism

Let us consider a per-region narrow-width prediction rate of r for a total of nbb executed basic blocks. Then, assuming a misprediction frequency of m, and an associated miss penalty of p, we may express the diminishing returns, $Cost$, of the recovery mechanism as follows:

$$Cost = nbb * r * m * p \tag{5.3}$$

In (5.3), the misprediction penalty p may be viewed as the cost of flushing the pipeline plus the additive cost to recover the correct bit-width size. Since the misprediction takes place at the execute stage, we may assume a 3 cycles penalty for flushing the pipeline. On the other hand, the cost to recover the correct bit-width mode may vary with the implementation complexity and the processor design. We assume a 5

[1] We assume that the carry signal can be prevented from propagating along the higher-bit carries of the ALU. In such case, the energy savings can even be more important than what we have presumed.

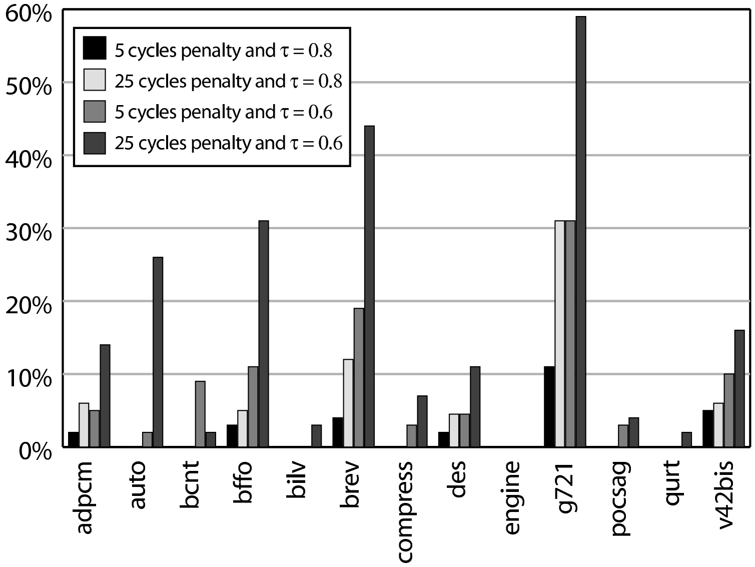


Figure 5.12: IPC degradation for different values of τ and p.

cycles recovery penalty for the best case and 25 cycles for the worst case. In Figure 5.11, we illustrated the impact of varying the narrow-width operand availability τ on the performance. As we increase τ, the speculation rate decreases because few regions may have high narrow-width operand availability. As a consequence, the misprediction frequency is also expected to decrease because the accuracy is sharpened. In contrast, lowering τ increases both r and m. Considering this fact, we plotted in Figure 5.12 the IPC degradation observed by varying the values of the narrow-width operands availability τ and the misprediction penalty p. On average, most applications experience IPC degradations without consequences on the performances when considering a best case misprediction penalty $p = 5$. In contrast, a worst case misprediction penalty of $p = 25$ can affect the performances by up to 31% for $\tau = 0.8$ and 60% for $\tau = 0.6$. An efficient scheme may therefore strive to keep p as low as possible.

5.8.2 Code size growth

Code size growth is mainly due to the re-encoding of the 32-bit address instructions with equivalent accumulator-based instructions and to the scheduling of the 32-bit instructions around the underlying basic block. Practically, a good cross-block scheduling algorithm may benefit from this code motion to improve the IPC and thereby alleviating the impact of the code size growth. We have not implemented such a tricky cross-block scheduling algorithm. Still, the impact of the code size growth is marginal. Considering a per-region narrow-width operands availability of 80%, we experienced less than 3,1% code size growth, on average, for our benchmarks set.

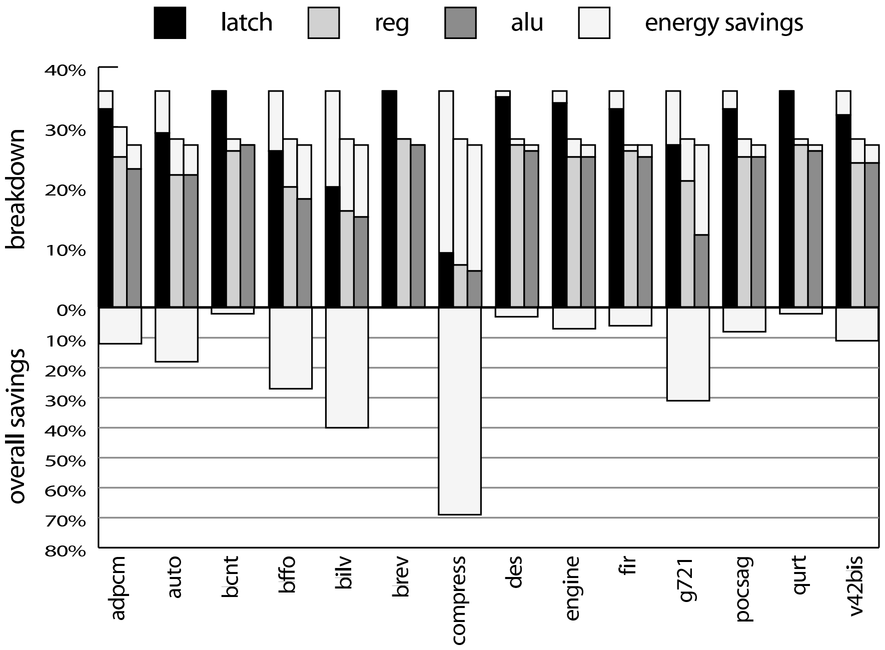


Figure 5.13: Breakdown of the datapath dynamic energy savings and overall gain.

5.8.3 Dynamic energy reduction

Figure 5.13 shows the breakdown of the dynamic energy savings obtained for each component of the processor's datapath. Some applications such as *bcnt*, *brev* and *qurt* show no benefit from using our scheme. This is mainly because not enough static narrow-width regions have been uncovered at code generation time. We may probably improve the detection of these regions by combining our scheme with a software-based technique such as the one proposed in [25]. On the other applications, however, we can observe an average datapath energy savings of up to 17%. On some modern embedded processors such as the *M.Core* [95], for instance, the datapath dynamic energy contributes to as much as 42% of the total processor's power consumption. Achieving a 17% energy reduction can therefore provide a substantial energy gain.

5.8.4 Leakage energy reduction

The *byte-slice* register file architecture we proposed also permits to tackle the static energy consumption. This is mainly due to the fact that when executing on a narrower datapath-width, the upper byte-slices of the register file are put into in low-power mode. Figure 5.14 illustrates the static energy savings observed in the register file when using our scheme. For the vast majority of the applications, an average of 22% reduction of the static energy is realized, with a peak energy savings of roughly 80% for the *compress* benchmark.

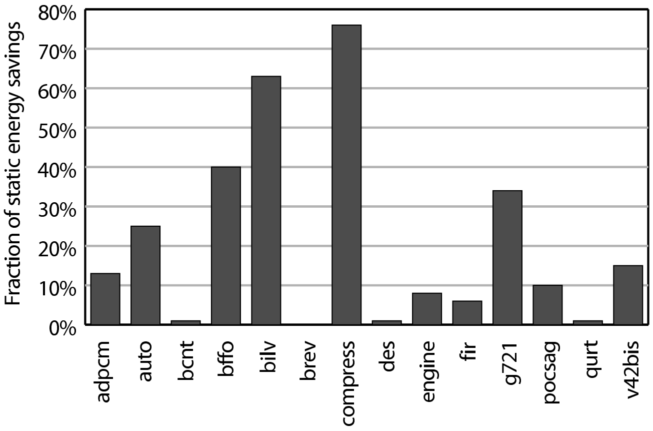


Figure 5.14: Register file static energy savings.

5.9 Summary

Operand-gating has been recently proposed as a means to dynamically exploiting the availability of narrow-width data elements in programs. Implementations of operand-gating have principally relied on the hardware to drive the gating decision. From a software point of view, some solutions have also emerged that take benefit of narrow-width operands to save energy. However, software-only solutions suffer a lot from not considering runtime informations. Hence, they must often be very conservative about the ranges of bit-width values that a data may take during its execution.

In this chapter, we have proposed a speculative software management scheme to overcome the difficulties encountered by software-only solutions. Central to our approach is the ability to expose dynamic narrow-width operands to the compiler; this in order to allow it to speculatively accommodate the execution of static narrow-width regions on a narrower datapath-width. For this purpose, we have introduced a novel register file organization, the ***byte-slice*** register file, that permits the width of the register file to be dynamically reconfigured; and a simple and efficient exception management mechanism to handle width mispredictions. Our evaluation results have indeed demonstrated the efficacy of our approach in managing the energy consumption at the software level. We showed that up to 17% of the dynamic energy and 22% of the static energy can be saved on the datapath, while only a negligible IPC degradation is observed for most applications.

Conclusions and perspectives

VLIW architectures have established themselves as one of the most successful approaches to deliver high performance at lower costs. They offer a good compromise between processing power and energy dissipation; as such they are becoming very popular in the embedded system domain. The key point with embedded VLIW architectures is that most of the circuit complexity found in general purpose processors to achieve higher performance is withdrawn from the processor's critical path and placed into the compiler, which is responsible to deliver the maximum of the performance. However, with power consumption becoming a key design constraint, and due to the lack of resource usage anticipation in the compiler, many power-related optimizations are actually obfuscated at the compiler level. The central idea of this thesis is that some improvements in the reduction of the energy consumption can be achieved at the compiler level if VLIW processors had some degree of adaptability to the application they are running. To defend this position, we proposed synergistic hardware-software solutions that can potentially improve the compiler effectiveness to manage power consumption, while still guaranteeing acceptable performance levels. Our investigations have focussed on four main areas of research:

1. ILP compilation analysis
2. study of program behaviors
3. adaptability of the cache subsystem
4. adaptability of the processor data-path

In this chapter, we quickly review some of the key research contributions presented in this thesis. We also suggest some future directions to improve this research.

Summary of research contributions

In Chapter 2, we have investigated the energy consumption issues involved when compiling for performance. The classic approach to this problem is based on the observation that energy and power consumption are roughly proportional to the total program execution time. In order to challenge this classical view of power management at the software level, we have developed a prototype compiler which includes classical ILP scheduling techniques such as the hyperblock. We have proposed an heuristic for studying the energy requirement of ILP techniques which basically relies on the observation that monitoring the variations in program performance could be achieved through some form of prediction mechanism or profiling at the software level. The rationale behind this approach is that using performance monitoring to drive energy or power consumption makes it possible to identify conditions leading to energy increase. Our analysis have revealed that there exists an ILP threshold above which an increase in performance may turn into diminishing energy reduction returns.

The results obtained in Chapter 2 confirmed our hypothesis about architecture-specific ILP limitations that could put a burden on power management opportunities. In order to account for these architecture peculiarities, we have started looking at compile time techniques that can be coupled with hardware management mechanisms to improve power management. From a compiler standpoint, this is equivalent to identifying the regions of the program that can benefit most from runtime optimizations and then adapting the underlying hardware to meet the architecture requirement of each of these regions.

In this sense, the first step toward achieving the above goal has been to lay emphasis on program characterization in order to understand the dynamic program behavior. Knowing the most frequently executed paths of a program can be of a crucial importance for the compiler since specific optimizations can be associated to them according to the characteristics enclosed in a program path. In Chapter 3, we have introduced such a program path analysis technique to reveal and characterize hot paths. The originality of the proposed approach is to rely on suffix arrays to devise a fast and efficient algorithm for searching hot program paths in a trace. This approach is unique in itself as no prior attempt to analyze a whole-program paths with suffix arrays has existed.

Scrutinizing inside a hot program path can reveal interesting characteristics that can benefit an optimizer. For instance, one can compute the optimal cache size required to host the data referenced within a hot region. This makes it possible to adapt the size of a cache to that of a hot region in order to save energy. In the embedded computing domain, however, current techniques to make a cache re-configurable addressed this at the application level. In Chapter 4, we introduced a new model of a cache to make it re-configurable on a per-phase basis in order to better take advantage of hot program regions. The model uses an intelligent combination of two techniques called *way-concatenation* and *selective cache ways* to propose an hybrid cache model with enhanced cache resizing granularities, and provides the possibility to adapt the size of the cache on a per-phase basis. A cache reconfiguration instruction is exposed to the compiler to take advantage of this at compile time.

Finally, in Chapter 5, we also looked at the possibility to adapt the width of a processor data-path to that of a program region. The principal motivation to lay emphasis on the processor data-path has come from the fact that its bit-width size keeps growing, especially because the pipeline is widened to process multiple instructions in

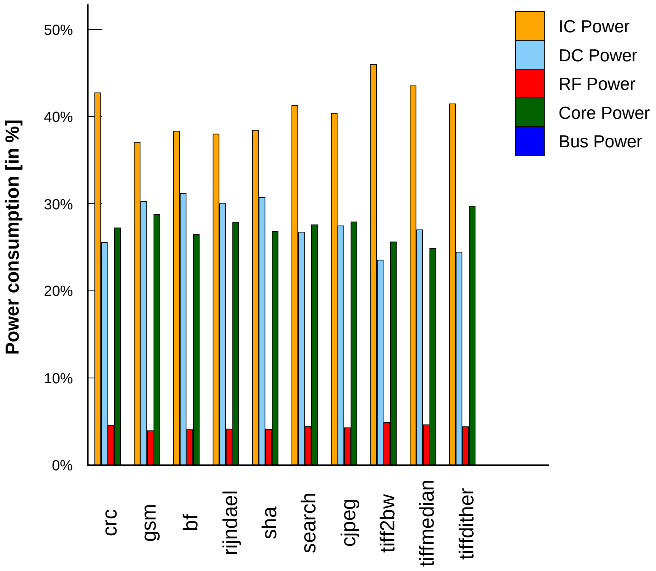


Figure 5.15: Distribution of power consumption.

parallel on one hand, and that the demand for more memory bandwidth increases on the other hand. Hence, on some embedded systems, the processor data-path accounts for a large fraction of the total dynamic power consumption (e.g. 42% on the M.Core). Up to now, the energy consumption of the processor data-path has been successfully addressed with hardware techniques, in particular for general purpose processors. At the compiler level, the few attempts that tried to do so bumped into the limitations of static analysis which are too restrictive in nature. We introduced a speculative software approach to better exploiting dynamic narrow-width operands at the compiler level, bursting most of the restrictions due to a static analysis. An ISA instruction allows the compiler to switch from a normal execution mode to a narrower bit-width mode, and vice-versa. Then, at hardware level, a simple exception management mechanism allows to recover to the correct execution mode if a misprediction is encountered.

Future work and perspectives

Time considerations prevented us from fully considering some of the key research contributions we investigated, but which would have warranted to be studied in more details. In this section, we elaborate on some of these which need to be looked into more precisely. Then, we draw some future research directions in the continuation of this thesis.

Future work

There are a few short terms studies that can bring added values to the quality of this research. At first, recall that most of the experiments conducted in this thesis targeted the data cache (*d-cache*). Although, such a choice can be easily warranted since the *d-cache* consumes an important fraction of the total power consumption, there is another important fraction of the energy consumption that is dissipated in the instruction cache (*i-cache*). Figure 5.15 shows the power consumption distribution for a few applications collected from the MiBench suite of benchmarks. The energy consumption model used to obtain these values is that described in Chapter 2, Section 2.2.1. It is clear from the figure that the most energy consuming resource is the *i-cache*, with more than 35% of the dynamic energy consumption being dissipated in this structure.

The question that directly comes in mind is to which extent the research contributions proposed in this thesis apply to the *i-cache* as well ? There is an avenue to exploit the cache resizing scheme proposed in Chapter 4 whenever SIMD operations can give rise to substantial performance improvements. This might be the case when embedded VLIW processors are used to implement DSP algorithms and graphics applications, which show high data parallelism potential. The fact is that when SIMD operations are employed, the number of executed instructions can be potentially reduced; hence, there may exist some opportunity to resize the *i-cache* as well. This, however, must be traded off with the amount of available data parallelism which sometimes requires that some transformations such as loop unrolling be applied to uncover even more opportunities; thus increasing the total instruction count.

The impact of using SIMD operations on the overall power consumption is more questionable, and we see here some opportunity to provide models that can help better understanding these issues. At first, there is a need to extend the instruction-based energy model described in Chapter 2, Section 2.2.1, to include the energy cost of SIMD operations. Then, the study we conducted to investigate the energy issues involved by ILP compilation can also be done to analyse the energy issues due to data parallelism. Of principal concern here are not only the energy cost of SIMD operations, but also that of the overhead operations used to uncover data parallelism, e.g. packing and unpacking operations.

An interesting study in the continuation of the work presented in Chapter 5 will be to consider using SIMD operations to exploit narrow-width operands as well. Intuitively, using SIMD operations can give rise to more opportunities to apply clock gating at the functional unit level. It might be interesting to see how effective this technique compares to the narrow-width scheme we proposed in Chapter 5.

There is a very interesting research paper of Dhodapkar et al. [32] about managing multi-configuration hardware via dynamic working set analysis. It would be of same interest to see in which extent such a technique can be used conjointly with the program path analysis scheme of Chapter 3 to catch different program path signatures into a single one. This will provide an effective basis to analyse the interplay of using different re-configuration techniques to save energy, e.g. cache size adaptation and data-path reconfiguration.

Perspectives

This thesis has provided the groundwork for a number of future key research contributions. May be the most noticeable one we see in the direct continuation of this thesis is on improving embedded VLIW processors to allow them take runtime decisions about a program dynamic execution. This may be at the advantage of both the performance and the power consumption. There are several clue which make we believe that this would be a plausible scenario in a near future. First, even if the compiler can take advantage of runtime data to reconfigure the underlying architecture, this is done at static time and is still highly reliant on profiles information accuracy. Hence, there is often a large optimization potential made possible by the knowledge of runtime information that is lost. Second, with the advances in processor architecture and the denser silicon manufacturing processes, it is now possible to envision integrating some architecture features into embedded processors to tolerate some degree of dynamic control mechanisms. This might open several research directions. One of them must be concerned with determining to which extent the integration of these architectural features in embedded VLIW processors can be energy counter-efficient with respect to the added performance value. There are also some opportunities for new compilation techniques that can take advantage of such features. In this respect, we see some opened doors in the direction of dynamic compilation to provide some added values to the optimization process.

Bibliography

[1] Albonesi, D.H. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the 32nd International Symposium on Microarchitecture*, November 1999.

[2] Allen, J.R., Kennedy, K., Porterfield, C., and Warren, J. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages, (POPL 83)*, January 1983.

[3] Amicel, R., and Bodin, F. A new system for high-performance cycle-accurate compiled simulation. In *Proceedings of the 5th International Workshop on Software and Compilers for Embedded Systems*, 2001.

[4] 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 Conference on Application-Specific Systems, Architectures and Processors*, June 2003.

[5] Bahar, R.I., and Manne, S. Power and energy reduction via pipeline balancing. In *Proceedings of the 28th International Symposium on Computer Architecture*, June 2001.

[6] Bala, V. Low overhead path profiling. Technical Report HPL-96-87, Hewlett Packard Labs, 1996.

[7] Balasubramonian, R., Albonesi, D.H., Buyuktosunoglu, A., and Dwarkadas, S. Memory hierarchy reconfiguration for energy and performance in general purpose processor architectures. In *Proceedings of the 33th International Conference on Microarchitecture*, pages 245–257, December 2000.

[8] 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.

[9] Ball, T., and Larus, J.R. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[10] Ball, T., and Larus, J.R. Efficient path profiling. In *Procedings of the 29th Annual International Symposium on Microarchitecture*, December 1996.

[11] Ball, Thomas. What's in a region? -or- computing control dependence regions in linear time and space. Technical Report 1108, University of Wisconsin – Madison, Computer Sciences Department, September 1992.

[12] Bellas, N., Hajj, I.N., Polychronopoulos, C.D., and Stamoulis, G. Architectural and compiler support for energy reduction in the memory hierarchy of high-performance microprocessors. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 70–75, August 1998.

[13] Benini, L., and De Micheli, G. System-level power optimization: techniques and tools. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(2):115–192, April 2000.

[14] Benini, L., Bruni, D., Chinosi, M., Silvano, C., Zaccaria, V., and Zafalon, R. A power modeling and estimation framework for vliw-based embedded systems. In *Proceedings of the IEEE Eleventh International Workshop on Power and Timing Modeling, Optimization and Simulation*, September 2001.

[15] Beszedes, A., Ferenc, R., Gyimothy T., Dolenc, A., and Karsisto, K. Survey of code-size reduction methods. *ACM Computing Survey*, 35(3):223–267, September 2003.

[16] 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 December.

[17] Bohr, M. Silicon trends and limits for advanced microprocessors. *Communications of the ACM*, 41(3):80–87, March 1998.

[18] Brooks, D., and Martonosi, M. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, January 1999.

[19] Brooks, D., Martonosi, M., and Bose, P. Modeling and analyzing cpu power and performance: Metrics, methods and abstractions. In *Tutorial presentation at the 7th IEEE Symposium on High Performance Computer Architecture (HPCA)*, January 2001.

[20] Brooks, D., Tiwari, V., and Martonosi, M. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proc. of the 27th Int'l Symp. on Computer Architecture*, June 2000.

[21] 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)*, August 2000.

[22] Burger, D., and Austin, T. The SimpleScalar Tool Set, Version 2.0. *Computer Architecture News*, pages 13–25, 1997.

[23] Butts, J.A., and Sohi, G.S. A static power model for architects. In *Proceedings of the 33th Annual International Symposium on Microarchitecture*, December 2000.

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

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

[26] Cao, Y., and Yasuura, H. System-level energy minimization approach using datapath width optimization. In *Proceedings of the International Symposium on Low Power Electronics and Design*, August 2001.

[27] Cao, Y., and Yasuura, H. Low-energy design using datapath width optimization for embedded processor-based systems. *IPSJ Journal*, 43(5):1348–1356, May 2002.

[28] Chang, P.P., Mahlhe, S.A., and Hwu,W.W. Using profile information to assist classic code optimizations. *Software-Practice and Experience*, 21:1301–1321, December 1991.

[29] 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.

[30] Cytron, R., Ferrante, J., Rosen, B., Wegman, M., and Zadeck, K. Efficiently computing static single assignement form and the control dependence graph. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 25–35, January 1989.

[31] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., and Zadeck, K. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[32] Dhodapkar, A., Smith, J.E. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[33] Drach, N., and Sebot, J. SIMD ISA Extensions: Tradeoff between Power Consumption and Performance on a Superscalar Processor. In *Proceedings of the Kool Chips Workshop*, December 2000.

[34] Philips Electronics. *TM1000 Preliminary Data Book*, 1997.

[35] 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.

[36] Fisher, J.A. Trace scheduling: A technique for global microcode compaction. *IEEE Transaction on Computers*, 30(7):478–490, 1981.

[37] Fisher, Joseph A. Very long instruction word architectures and the eli-512. In *Proceedings of the 10th annual international symposium on Computer architecture*, pages 140–150, June 1983.

[38] Flautner, K., Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[39] Gandhi, K.R., and Mahapatra, N.R. A detailed study of hardware techniques that dynamically exploit frequent operands to reduce power consumption in integer function units. In *Proceedings of the 2nd Workshop on Duplicating, Deconstructing, and Debunking*, June 2003.

[40] Gonzalez, R. and Horowitz, M. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1283, September 1996.

[41] Grossi, R., and Vitter, J.S. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the ACM Symposium on the Theory of Computing*, 2000.

[42] Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., and Brown, R.B. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th IEEE International Workshop on Workload Characterization*, pages 3–14, December 2001.

[43] Hasegawa, A., Kawasaki, I., Yamada, K., Yoshioka, S., Kawasaki, S., and Biswas, P. Sh3: High code density, low power. *IEEE Micro*, pages 11–19, 1995.

[44] Hayakawa, F., Okano, H., and Suga, A. An eight-way vliw embedded multimedia processor with advanced cache mechanism. In *Proceedings of the Third IEEE Asia-Pacific Conference on ASICs*, August 2002.

[45] Hennessy, J. The future of systems research. *IEEE Computer*, pages 27–33, August 1999.

[46] Hill, M.D., Smith, A.J. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38:1612–1630, December 1989.

[47] Horowitz, M., Indermaur, T., and Gonzalez, R. Low-Power Digital Design. In *Proceedings of the IEEE Symposium on Low Power Electronics*, pages 8–11, October 1994.

[48] Hsu, C-H., Kremer, U., and Hsiao, M. Compiler-directed dynamic frequency and voltage scheduling. In *Proceedings of the Workshop on Power-Aware Computer Systems*, November 2000.

[49] Hwu, W.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Ouellette, R.G, Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., and Lavery, D.M. The superblock: An effective technique for vliw and superscalar compilation. *Journal of Supercomputing*, 7(1-2):229–248, 1993.

[50] Inoue, K., Ishihara, T., and Murakami, K. Way-predicting set-associative cache for high performance and low energy consumption. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 273–275, August 1999.

[51] Texas Instruments. *TMS320C62xx CPU and Instruction Set: Reference Guide*, January 1997.

[52] Irwin, M.J., Kandemir, M., and Vijaykrishnan, N. *Low Power Design Methodologies: Hardware and Software Issues.* Low Power Design Tutorial at PACT, 2000.

[53] Jacobson, Q., Rotenberg, E., and Smith, J. Path-based next trace prediction. In *Proceedings of the 30th International Symposium on Microarchitecture*, November 1997.

[54] Karp, R.M., Miller, R.E., and Rosenberg, A.L. Rapid identification of repeated patterns in strings, arrays and trees. In *Proceedings of the 4th ACM Symposium on Theory of Computing*, 1972.

[55] Kaxiras, S., Hu, Z., and Martonosi, M. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[56] Kim, N.S, Todd, A., Blaauw, D., Mudge, T., Flautner, K., Hu, J.S., Irwin, M.J., Kandemir, M., and Vijaykrishnan, N. Leakage current : Moore's law meets static power. *IEEE Computer*, pages 68–75, December 2003.

[57] Kin, J., Gupta, M., and Magione-Smith, W.H. The filter cache: An energy efficient memory structure. In *Proceedings of the 30th International Conference on Microarchitecture*, December 1997.

[58] Langdale, G., and Gross, T. Evaluating the relationship between the usefulness and accuracy of profiles. In *Proceedings of the 2nd Workshop on Duplicating, Deconstructing, and Debunking*, June 2003.

[59] 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.

[60] Larus, J.R. Whole program paths. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1999.

[61] Lee, C., Lee, J.K., and Hwang, T. Compiler optimization on instruction scheduling for low power. In *Proceedings of 13th International Symposium on System Synthesis*, September 2000.

[62] Lee, C., Potkonjak, M., and Mangione-Smith, W.H. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[63] Lee, S., Ermedahl, A., and Min, L. An accurate instruction-level energy consumption model for embedded RISC processors. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'01)*, June 2001.

[64] C. Lefurgy. *Efficient Execution of Compressed Programs.* PhD thesis, University of Michigan, 2000.

[65] Lekatsas. *Code Compression for Embedded Systems Princeton University.* PhD thesis, Princeton University, 2000.

[66] Loh, G. Exploiting data-width locality to increase superscalar execution bandwidth. In *Proceedings of the 35th International Symposium on Microarchitecture*, November 2002.

[67] Lowney, P.G., Freudenberger, S.G., Karzes, T.J., Lightenstein, W.D., Nix, R.P., O'Donnell, J.S., and Ruttenberger, J.C. The multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[68] Mahlke, S. A., Lin, D. C., Chen, W. Y., Hank, R. E., and Bringmann, R. A. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992.

[69] 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), November 2001.

[70] Malik, A., Moyer, B., and Cermak, D. A low power unified cache architecture providing power and performance flexibility. In *Proceedings of International Symposium on Low Power Electronics and Design*, pages 241–243, 2000.

[71] Manber, U., and Myers, G. Suffix arrays: A new method for on-line string searches. In *Proceedings of 1st ACM-SIAM SODA*, pages 319–327, 1990.

[72] 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.

[73] Marculescu, D. Profile-driven code execution for low power dissipation. In *Proceedings of ACM International Symposium on Low Power Electronics and Design (ISLPED)*, July 2000.

[74] Moreno, J.H., et al. An innovative low-power high-performance programmable signal processor for digital communications. *IBM Journal of Research & Development*, 47(2-3):299–326, March/May 2003.

[75] Muchnick, Steven S. *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers, 1997.

[76] Mudge, T. Power: A first class design constraint. *Computer*, 34(4):52–57, April 2001.

[77] 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.

[78] Nevill-Manning, C.G. and Witten, I.H. Identifying hierarchical structure in sequences. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[79] Nicolau, A., and Fisher, J. Measuring the parallelism available for very long instruction word architectures. *IEEE Transaction on Computers*, 33(11):968–976, November 1984.

[80] Palacharla, S., Jouppi, N., and Smith, J.E. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[81] Parikh, A., Kandemir, M., Vijaykrishman, N., and Irwin, M.J. Instruction scheduling based on energy and performance constraints. In *Proceedings of the IEEE Computer Society Annual Workshop on VLSI*, April 2000.

[82] Parthasarathy Ranganathan, N.P.J., Adve, S., and Jouppi, N.P. Reconfigurable caches and their application to media processing. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[83] Plezkun, A.R. Techniques for compressing program address traces. In *Proceedings of the 27th International Conference on Microarchitecture*, pages 32–40, 1994.

[84] Pokam, G., and Bodin, F. An Off-line Approach for Whole-Program Paths Analysis using Suffix Arrays. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, 2004.

[85] Pokam, G., and Bodin, F. Energy-efficiency potential of a phase-based cache resizing scheme for embedded systems. In *Proceedings of the 8th IEEE International Worskhop on Interaction between Compilers and Computer Architectures (INTERACT-8)*, February 2004.

[86] Pokam, G., and Bodin, F. Understanding the energy-delay tradeoff of ilp-based compilation techniques on a vliw architecture. In *Proceedings of the 11th Workshop on Compilers for Parallel Computers*, July 2004.

[87] 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.

[88] Pokam, G., Rochecouste, O., Seznec, A., and Bodin, F. Speculative software management of datapath-width for energy optimization. In *Proceedings of the ACM SIGPLAN/SIGBED International Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'04)*, June 2004.

[89] Pokam, G., Simonnet, J., and Bodin, F. A retargetable preprocessor for multimedia instructions. In *Proceedings of the 11th Workshop on Compilers for Parallel Computers*, June 2001.

[90] Powell, M.D., Agarwal, A., Vijaykumar, T., Falsafi, B., and Roy, K. Reducing set-associative cache energy via way prediction and selective direct-mapping. In *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001.

[91] Powell, M.D., Yang, S-H., Falsafi, B., Roy, K., and Vijaykumar, T. Gated-vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 2000.

[92] Ruby, Lee. Subword parallelism. *IEEE Micro*, 16(4):51–59, 1996.

[93] Russell, J.T., and Jacome, M.F. Software power estimation and optimization for high performance, 32-bit embedded processors. In *Proceedings of the International Conference on Computer Design*, 1998.

[94] Saputra, H., Kandemir, M., Vijaykrishnan, N., Irwin, M.J., Hu, J.S., Hsu, C-H., and Kremer, U. Energy-conscious compilation based on voltage scaling. In *Proceedings of the ACM SIGPLAN Joint Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'02) and Software and Compilers for Embedded Systems (SCOPES'02)*, June 2002.

[95] Scott, J., Lea Hwang Lee, John Arends and William Moyer. Designing the low-power m.core architecture. In *Proceedings of Power Driven Microarchitecture*, June 1998.

[96] Sharangpani, H. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, September 2000.

[97] Sherwood, T., Calder, B. Time varying behavior of programs. Technical Report CS99-630, University of California, San Diego, August 1999.

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

[99] SIA. *International Technology Roadmap for Semiconductors*, 2001.

[100] Sinha, A., and Chandrakasan, A. Energy aware software. In *Proceedings of the 13th International Conference on VLSI Design*, January 2000.

[101] Slingerland, N., Smith, A.J. Cache performance for multimedia applications. In *Proceedings of International Conference on Supercomputing*, pages 204–217, June 2001.

[102] Smith, I.E., et al. The zs-i central processor. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–204, October 1987.

[103] Smith, J.E, and Sohi, G.S. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, December 1995.

[104] Sprangle, E., and Carmean, D. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 25–34, May 2002.

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

[106] Thompson, S., Packan, P., and Bohr, M. MOS scaling: Transistor challenges for the 21st century. *Intel Technology Journal*, 1998.

[107] Tiwari, V., Malik, S., and Wolfe, A. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on VLSI Systems*, 2(4):437–445, December 1994.

[108] Tiwari, V., Malik, S., Wolfe, A., and Lee, M. Instruction level power analysis and optiomization of software. *Journal of VLSI Signal Processing Systems*, (1):1–18, 1996.

[109] Tiwari, V., Singh, D., Rajgopal, S., Mehta, G., Patel, R., and Baez, F. Reducing power in high-performance microprocessors. In *Proceedings of the 35th Design Automation Conference*, June 1998.

[110] Tseng, J.H., and Asanovic, K. Banked multiported register files for high-frequency superscalar microprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[111] Unsal, O.S., Koren, I., Krishna, C.M., and Moritz, C.A. Cool-fetch: Compiler-enabled power-aware fetch throttling. *ACM Computer Architecture Letters*, 1, 2002.

[112] Vijaykrishnan, N., Kandemir, M., Irwin, M.J., Kim, H.S., and Ye, W. Energy-driven integrated hardware-software optimizations using simplepower. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[113] 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 (ISLPED)*, pages 163 – 168, 1999.

[114] Yang, S-H., Powell, M.D., Falsafi, B., and Vijaykumar, T. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Proceedings of International Symposium on High Performance Computer Architecture*, February 2002.

[115] Yang, S-H., Powell, M.D., Falsafi, B., Roy, K., and Vijaykumar, T. An integrated circuit/architecture approach to reducing leakage in deep-submicron high performance i-caches. In *Proceedings of International Symposium on High Performance Computer Architecture*, January 2001.

[116] Young, Cliff., and Smith, Michael. Better global scheduling using path profiles. In *Proceedings of the 31th International Symposium on Microarchitecture*, December 1998.

[117] Zhang, C., Vahid, F., and Najjar, W. A highly configurable cache architecture for embedded systems. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[118] 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, March 2003.

[119] Zhou, H., Toburen, M.C., Rotenberg, E., and Conte, T.M. Adaptive mode-control: A static-power-efficient cache design. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[120] Zyuban, V., and Kogge, P. Split register file architectures for inherently lower power microprocessors. In *Proceedings of Power-Driven Microarchitecture Workshop*, pages 32–37, 1998.

Résumé

Cette thèse propose de réduire la consommation d'énergie des architectures VLIW tout en essayant de préserver le maximum de performance possible. Contrairement à certaines approches logicielles tendant à favoriser l'optimisation de code pour obtenir des gains en énergie, nous présentons des arguments en faveur d'une approche synergique intégrant matériel et logiciel à la fois. L'idée principale défendue tout au long de cette thèse repose sur le fait que seule une compréhension avancée du comportement dynamique d'un programme au niveau du compilateur est susceptible de produire un meilleur contrôle de la gestion de la consommation d'énergie. Pour cela, nous introduisons une technique d'analyse statique du comportement dynamique d'un programme afin de parvenir à identifier et à caractériser les chemins les plus fréquemment exécutés d'un programme. L'objectif visé étant la réduction de la consommation d'énergie, nous montrons par la suite que sur directive du compilateur, l'architecture de la machine peut être modifiée pour s'adapter à un état dynamique particulier du programme. Nous présentons les conditions d'une telle reconfiguration ainsi que les éventuelles modifications à apporter à l'architecture, à la fois pour le système des caches que pour le chemin de données d'un processeur VLIW.

Mots clefs : compilation, puissance dissipée, analyse de programmes, architecture reconfigurable

Abstract

This thesis concerns low power compilation techniques for VLIW architectures. In contrast to many software approaches that rely on code optimization techniques to save energy, we present arguments in favour of a synergistic approach that integrates both the hardware and the software. The principal idea defended throughout this thesis rests on the fact that only a comprehensive understanding of a program dynamic behavior at the compiler level is likely to provide a better power consumption control mechanism. For this purpose, a static approach for analyzing a program dynamic behavior is introduced that can identify and characterize hot program paths. Since the focus of this thesis is on energy reduction, we also present how a compiler might take advantage of the knowledge of a program dynamic behavior to adapt the underlying architecture to a specific dynamic program instance. We present the conditions to undertake such a reconfiguration, as well as the possible modifications brought to the architecture, both for the cache components and the processor data-path.

Keywords : compilation techniques, low power, program analysis, reconfigurable computing