UNIVERSITY OF PARIS SUD (XI)

Alternative Approaches to Improve Performance without ILP

A Thesis submitted to the

UNIVERISTY OF PARIS SUD (XI)

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in Computer Science

by

Sami YEHIA

Defended on September 22, 2004

Thesis Jury

André	SEZNEC	IRISA/INRIA	Reviewer
Sanjay	PATEL	University of Illinois at Urbana-Champaign	Reviewer
Marc	DURANTON	Philips Research	Examiner
Olivier	TEMAM	Université Paris XI	Director

UNIVERSITÉ DE PARIS SUD (XI)
U.F.R SCIENTIFIQUE D'ORSAY

THÈSE
présentée
pour obtenir

Le GRADE de DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ PARIS XI ORSAY
SPÉCIALITÉ: INFORMATIQUE

par

Sami YEHIA

SUJET: Approches Alternatives pour Améliorer les Performances en l'Absence de Parallélisme d'Instructions

Soutenue le 22 septembre 2004 devant le jury composé de

André	SEZNEC	IRISA/INRIA	Rapporteur
Sanjay	PATEL	University of Illinois at Urbana-Champaign	Rapporteur
Marc	DURANTON	Philips Research	Examinateur
Olivier	TEMAM	Université Paris XI	Directeur

Face à la roche, le ruisseau l'emporte toujours,
non pas par la force mais par la persévérance.
H. Jackson Brown

A ma chère famille,
Hasnaa et *Noureldine*.

Remerciements

Cette thèse a été préparée au Laboratoire de Recherche en Informatique (LRI) à L'université Paris-Sud (XI) entre septembre 1999 et septembre 2004 sous la direction d'Olivier Temam.

Je voudrais tout d'abord remercier Olivier Temam pour m'avoir soutenu durant mon DEA et mes quatre années de thèse. Je veux surtout le remercier pour sa disponibilité, son écoute et surtout ses directions et conseils sans lesquels ce travail n'aurait pas abouti de la sorte. Au-delà de la thèse, je souhaite surtout le remercier pour nous avoir appris que la recherche, c'est surtout beaucoup de persévérance, d'humilité et de générosité.

Je tiens à exprimer ma gratitude pour l'honneur que m'a fait chaque membre de mon jury. Je souhaite remercier doublement André Seznec pour avoir présidé le jury et pour avoir rapporté la thèse. Je veux surtout remercier Sanjay Patel qui nous fit un grand honneur de rapporter la thèse et d'être membre du jury malgré le long voyage. Merci à Marc Duranton pour avoir participé à mon jury et pour ses conseils et directions pour la continuation de ce travail. Je veux encore souligner combien la lecture attentive et minutieuse de la thèse de la part de chaque membre du jury, ainsi que leurs commentaires, me furent précieux. Je tiens aussi à remercier Jean-Franois Collard à HP Labs pour ses contributions ainsi que son soutien durant la dernière ligne droite de la rédaction.

Merci à tous les membres de l'équipe micro-architecture au LRI et ALCHEMY à l'INRIA-Futurs. Je veux particulièrement remercier Alexandre Farcy pour m'avoir donné beaucoup de son temps et de ses conseils dans de longues discussions pendant ma thèse. Merci à Nathalie Drach-Temam pour avoir toujours été à l'écoute et pour

ses conseils précieux. Merci à David Parello et Gilles Mouchard pour tout le temps, la bonne humeur et parfois les dures périodes de doute que l'on a partagé. Je veux aussi remercier Frederic Gruau et Daniel Gracia Pérez et leur exprimer combien ce fut un plaisir de partager le bureau.

Le plus difficile à faire est de finalement remercier les êtres qui me sont les plus chers, tant ma reconnaissance envers eux est immense, et qui sans eux, aucune réussite n'aurait été possible. Je veux en première place remercier mon épouse, Hasnaa, qui a tant enduré et supporté pendant ces années de thèse et aussi pour tout l'amour et le soutien inconditionnel qu'elle m'a donné dans les moments les plus difficiles. Je veux aussi remercier mon fils Noureldine pour, sans toujours le savoir, avoir fait preuve de patience et d'amour quand je n'étais pas toujours disponible pour lui. Et *last but not least*, je veux remercier du fond de mon cœur mon père, qui fut toujours mon seul exemple, pour tout le soutien, l'éducation, et surtout l'amour qu'il m'a donné pendant toute ma vie, et surtout pour m'avoir appris que l'honnêteté, le travail dur et la persévérance sont les seuls outils pour bâtir une vie solide et heureuse.

Je souhaite à la fin remercier ma maman, qui malheureusement n'aura pas l'occasion de lire ces lignes, et lui dire combien son amour, ses sacrifices et son affection sont toujours resté mes principales sources d'énergie et d'espoir dans ma vie.

Abstract

Current integration technologies and advances in semiconductor manufacturing open the way to an unprecedented number of transistors on a single processor die. Still, few approaches address applications that have complex data structures or irregular data access patterns. Integer applications particularly suffer from such properties.

The main bottlenecks lying in non-numeric applications are the low ILP and irregular data structures that leads to irregular memory accesses having low spatial locality.

In this thesis we propose alternative approaches to exploit on-chip space and reduce the memory wall effect. For codes that have little ILP, we propose a novel approach that collapses dependent instructions to functions that execute independently and in parallel.

Because the collapsing approach is limited by dependent memory accesses, we propose the "load squared", an approach that improves performance of dependent loads that have high miss ratios by adding logic closer to memory. We also investigate a generalization of this concept by presenting a decoupled architecture associated with a language extension that explicitly separates execution from data accesses.

Résumé

La possibilité d'intégrer plus d'un milliard de transistors dans un processeur offre un potentiel inégalé pour la haute performance. L'exploitation de cette immense capacité d'intégration est conditionnée par la quantité de parallélisme que l'on peut extraire. Les processeurs superscalaires actuels exploitent le parallélisme d'instructions afin d'exécuter plusieurs instructions par cycle, ainsi que le parallélisme de données en exécutant la même instruction sur plusieurs données en parallèle. Cependant, pour certain codes, notamment les SPECINT 2000, les performances obtenues sont loin des performances crêtes des processeurs, cet écart est principalement dû au manque de parallélisme ainsi qu'à l'irrégularité de l'application et des structures de données utilisées.

Dans cette thèse nous proposons plusieurs méthodes pour pallier au problème du manque de parallélisme ainsi que le problème des accès mémoires des structures de données irrégulières, notamment les structures de données chaînées.

Dans un premier temps nous proposons un nouveau mécanisme qui traite les séquences d'instructions dépendantes qui contiennent peu de parallélisme, en les écrasant sous forme de fonctions et en les exécutant sur une unité reconfigurable.

L'extraction de fonctions étant limitée par la présence de chaînes d'instructions de chargement dépendantes, nous proposons une méthode qui permet d'accélérer les instructions de chargement indirectes en migrant le calcul d'adresse plus proche de la mémoire (typiquement dans le contrôleur).

Finalement, nous étudions la possibilité d'explicitement migrer de plus grandes parties de l'application vers une mémoire intelligente à l'aide d'une extension du langage C et d'une architecture découplée.

Contents

I Présentation en Français **1**

I.1 Introduction 1

I.1.1 Contributions 1

I.1.2 Organisation de la thèse 2

I.2 Exploitation de l'espace pour plus de calcul 2

I.2.1 Les architectures reconfigurables 2

I.2.2 Les architectures de cellules 3

I.2.3 Conclusion 4

I.3 Extractions de fonctions à partir d'une séquences d'instructions . . . 4

I.3.1 Principes 4

I.3.2 Potentiel et limitations 6

I.3.3 Implémentation 6

I.3.4 Conclusion 7

I.4 Double chargement: une méthode pour réduire la latence des chargements de donnée indirectes 7

I.4.1 Principes et implémentation 8

I.4.2 Evaluation 9

I.4.3 Perspective: Migration des calculs vers la mémoire 10

I.5 Conclusions 13

1 Introduction **15**

1.1 Computer Architecture Challenges 15

1.2 Contributions 16

1.3 Thesis Organization 16

2 Exploiting On-chip Space for Computations 19
2.1 Reconfigurable Architectures 19
2.1.1 PRISC 20
2.1.2 OneChip 21
2.1.3 DISC: A Dynamic Instruction Set Computer 22
2.1.4 PipeRench 23
2.1.5 The Chimaera architecture 24
2.1.6 The Garp architecture 25
2.1.7 Summary of reconfigurable architectures 27
2.2 Cell Architectures 27
2.2.1 RaPiD : Reconfigurable Pipelined Datapaths 28
2.2.2 Raw machines 29
2.2.3 GPA: Grid Processor Architectures 30
2.3 Problems and Limitations 33

3 From Sequences of Dependent Instructions to *Functions*: An Approach for Improving Performance without ILP or Speculation 35
3.1 Introduction 35
3.2 Principles 36
3.3 Experimental Framework 40
3.4 Potential of the Approach 41
3.4.1 Potential performance improvements 41
3.4.2 Analyzing and overcoming the limitations of the approach . . 42
3.5 Implementation 49
3.5.1 Generating DFG and functions 49
3.5.2 Hardware implementation of functions 52
3.5.3 Implementing functions using the rePLay hardware framework 53
3.5.4 Performance analysis of the implementation 57
3.6 Conclusions 61

4 Load Squared: Adding Logic Close to Memory to Reduce the Latency of Indirect Loads with High Miss Ratios 63
4.1 Introduction 63
4.2 Related Work 65
4.2.1 Linked data structures traversal. 65
4.2.2 Memory-side logic. 66
4.2.3 Intelligent memories. 67
4.3 Principles 70
4.3.1 Detecting and issuing Load Squared 73
4.4 Experimental Framework 75

4.5 Performance Evaluation 76
4.5.1 Load Squared potential 76
4.5.2 Efficiency of load predictors 77
4.5.3 Performance results 79
4.6 Perspectives: Explicitly Migrating Memory Computations Closer to Memory. 80
4.6.1 Decoupled architectures 81
4.6.2 The Data Structures Conscious Machine (DSCM) 83

5 Conclusions and Perspectives **87**
5.1 Summary 87
5.2 Perspectives 88

A The DSCM Architecture **89**
A.1 The DSCM using intelligent memory 89
A.2 The single-processor DSCM architecture 92
A.3 DSCM instruction set extension 92
A.4 Methodology and Experimental Results 95

Bibliography **99**

List of Figures

I.1 Example de fontions. 5
I.2 Opérateur matériel. 5
I.3 Architecture. 7
I.4 Double chargement. 8
I.5 Détection de double chargement. 9
I.6 DSCM: List chaînée. 11
I.7 Architecture DSCM. 12

2.1 PRISC architecture. 20
2.2 OneChip architecture. 21
2.3 DISC linear hardware space. 22
2.4 Hardware virtualization in PipeRench. 23
2.5 The overall Chimaera architecture. 24
2.6 Garp compiler's inner process. 26
2.7 RaPiD architecture. 28
2.8 RaPiD-C. 29
2.9 A Raw microprocessor. 30
2.10 GPA. 31
2.11 TRIPS architecture. 32
2.12 D-morph frame management. 32

3.1 An example of instruction collapsing. 37
3.2 Translating function `r3` into a hardware operator. 39
3.3 Phases of the optimization engine. 41
3.4 Different possible DFG shapes. 41
3.5 Theoretical speedup for different trace sizes. 42

3.6 DFG height distribution. 43
3.7 Distributions of cuts. 43
3.8 Cumulative distribution of the number of inputs per function. 44
3.9 Impact of the number of inputs on the theoretical speedup. 45
3.10 Percentage of loads. 45
3.11 Average depth of loads. 47
3.12 Distribution of load depth among all load cuts. 47
3.13 Percentage of non-collapsible instructions. 48
3.14 Cuts because of carries from upper significant bits. 48
3.15 Effect of relaxing the upper significant carries constraints. 48
3.16 Function generation. 50
3.17 Function Generation Engine: generating bit 2 of node n2. 51
3.18 Function Repository Table (FRT). 51
3.19 Implementation of functions. 52
3.20 The core architecture. 54
3.21 Average frame size. 55
3.22 Dynamic instructions coverage. 55
3.23 Local speedup. 56
3.24 Global speedup. 56
3.25 Combining instruction collapsing with perfect address prediction. . . 58
3.26 Combining instruction collapsing with perfect cache. 59
3.27 Combining instruction collapsing with perfect RePLay. 59
3.28 Alpha IPC (1). 60
3.29 Alpha IPC (2). 61

4.1 Lisp processor architecture. 66
4.2 The VIRAM architecture. 67
4.3 Smart memories architecture. 68
4.4 Active Page architecture. 69
4.5 Load squared architecture. 70
4.6 Predicting and issuing load squared. 72
4.7 Load predictors. 73
4.8 Potential loads squared and Miss/Miss occurrences. 77
4.9 Load prediction rate. 78
4.10 Non-speculative load prediction rate. 78
4.11 Percentage of load squared. 79
4.12 Speedup obtained with the load squared mechanism. 80
4.13 A decoupled architecture. 81
4.14 The HiDISC architecture. 82
4.15 List traversal in DSCM. 84

4.16 Tree traversal in DSCM. 85

A.1 The DSCM architecture. 90
A.2 A single-processor DSCM architecture. 91
A.3 ALPHA instruction set extension for the DSCM architecture. 93
A.4 List traversal example. 94
A.5 Matrix multiplication in Decoupled C. 96
A.6 List traversal speedup for different value of overlapped work (W). . . 97
A.7 Matrix multiplication speedup for different sizes (N). 97

Chapter I

Présentation en Français

I.1 Introduction

La possibilité d'intégrer plus d'un milliard de transistors dans un processeur offre un potentiel inégalé pour la haute performance. L'exploitation de cette immense capacité d'intégration est conditionnée par la quantité de parallélisme que l'on peut extraire. Les processeurs superscalaires actuels exploitent le parallélisme d'instructions afin d'exécuter plusieurs instructions par cycle, ainsi que le parallélisme de donnée en exécutant la même instruction sur plusieurs données en parallèle. Cependant, pour certain codes, notamment les SPECINT 2000 [Hen00], les performances obtenues sont loin des performances crêtes des processeurs, cet écart est principalement dû à l'irrégularité de l'application et des structures de données utilisées. Cette irrégularité empêche l'obtention de hautes performances pour deux raisons principales:

- L'existence de longues chaînes d'instructions dépendantes empêchant leur exécution en parallèle.
- L'irrégularité des structures de données empêche le chargement (ou le préchargement) des données en parallèle. Ce problème est surtout aggravé par l'existence de longues chaînes d'instructions de chargement dépendantes très fréquentes dans les structures de données chaînées.

Dans cette thèse nous proposons plusieurs méthodes pour réduire l'effet de ces deux problèmes.

I.1.1 Contributions

Nous proposons dans cette thèse 3 contributions majeures:

- Nous proposons un nouveau mécanisme qui traite les séquences d'instructions dépendantes qui contiennent peu de parallélisme, en les écrasant sous forme de fonctions et en les exécutant sur une unité reconfigurable.
- L'extraction en fonctions étant limitée par la présence de chaînes d'instructions de chargement dépendantes, nous proposons une méthode qui permet d'accélérer les instructions de chargement indirectes en migrant le calcul d'adresse plus proche de la mémoire (typiquement dans le contrôleur).
- Finalement, nous étudions la possibilité d'explicitement migrer de plus grandes parties de l'application vers une mémoire intelligente à l'aide d'une extension du langage C et d'une architecture découplée.

I.1.2 Organisation de la thèse

Le Chapitre 1 introduit les motivations ainsi que les contributions de la thèse. Dans le Chapitre 2, nous présentons un état de l'art des différentes approches et architectures qui exploitent la surface de la puce différemment, en dédiant plus d'espace pour le calcul. Le Chapitre 3 présente une approche alternative pour améliorer les performances des séquences d'instructions dépendantes. Dans le Chapitre 4, nous présentons les *load squared* ou les *doubles chargements*, une méthode qui permet d'accélérer deux instructions de chargements dépendantes ne se trouvant pas dans les caches. Comme perspective nous introduisons aussi dans ce chapitre une approche dans laquelle nous découplons explicitement le calcul des accès de données via une extension du langage C. Nous concluons dans le Chapitre 5. Nous résumons si dessous chaque partie de la thèse.

I.2 Exploitation de l'espace pour plus de calcul

Dans les processeurs actuels, la plus grande partie de la puce est plutôt dédiée aux différents mécanismes de pipeline, spéculation, préchargement et chargement de données. Cependant, de nouvelles architectures émergentes proposent d'utiliser l'espace sur la puce différemment. Principalement, ces architectures cherchent à dédier plus d'espace sur la puce au calcul même, et ceci soit a travers des unités de calcul configurables ou à travers des grilles de processeurs ou de cellules simples sur lesquels est distribuée l'application.

I.2.1 Les architectures reconfigurables

Dans cette partie, nous décrivons un état de l'art des architectures reconfigurables. Les architectures reconfigurables offrent un compromis entre les performances des

Architecture	Sélection des calculs	Compilateur	Granularité	Interface mémoire	Extension du jeu d'instructions
PRISC	Profilage	Oui	Branchements simples	Non	Une instruction
OneChip	Codés à la main	Non	Toute fonction	Direct	Une instruction
CHIMAERA	Analyse des codes	Oui	Branchements simples	Non	RFUOP
PipeRench	Analyse des codes	Oui	Bloques basics	Non	Jeux d'instruction spécifique
DISC	Codés à la main	Non	fonctions fonctions	Direct	Non existant
Garp	Profilage	Oui	Branchements simples et hyper bloques	Direct	Quelques instructions

Table I.1: *Architectures Reconfigurables.*

circuits à applications spécifiques (ASIC) et la flexibilité des architectures générales. L'architecture PRISM [Ath93] est une des premières propositions qui utilisent un co-processeur reconfigurable pour accélérer l'exécution.

L'architecture PRISC [Raz94] est la première à proposer des unités fonctionnelles reconfigurables dans un processeur RISC. Une instruction spéciale pointe vers la configuration appropriée stockée dans la mémoire. Une approche similaire est proposée dans OneChip [Wit96], sauf que l'unité reconfigurable est implémentée hors de la puce, offrant une interface directe avec la mémoire. L'architecture DISC [Wir95] propose une unité reconfigurable séparée supportant une reconfiguration dynamique. PipeRench [Gol00] est une architecture qui utilise un pipeline dimensionnable d'unités reconfigurables. Finalement les architectures CHIMAERA [Hau97a] et Garp [Hau97b] proposent en plus d'une architecture reconfigurable, chacune un compilateur générant les configurations à partir d'un code C. L'approche CHIMAERA en particulier propose des unités fonctionnelles dans un processeur superscalaire qui exécutent les configurations générées par le compilateur. La Table I.1 résume les propriétés des différentes architectures reconfigurable.

I.2.2 Les architectures de cellules

Les architectures de cellules dédient plus d'espace au calcul en exécutant des régions de codes sur une grille ou une structures de processeurs relativement simples (non reconfigurables en général). Ainsi, l'architecture RaPiD est une architecture qui implémente un pipeline de cellules programmables sur lesquels les calculs sont pipelinés. L'architecture Raw est une grille de processeurs simples associée à des éléments de

routage. Un compilateur, Maps, distribue les calculs sur la grille. Finalement, l'architecture GPA (Grid Processeur Architecture) propose une grille d'unités de calculs (ALU). Les instructions sont exécutées sur la grille selon le flot de donnée.

I.2.3 Conclusion

Les architectures présentées proposent de dédier plus d'espace de la puce au calcul plutôt qu'à d'autre mécanismes plus compliqués de spéculation et de contrôles. Cependant, la plupart de ces architectures parient sur la régularité des données et du flot de contrôle. Plusieurs applications, notamment les applications de calcul entier et ceux contenant des structures de données complexes, ont du mal à exploiter efficacement ces architectures. Les principaux obstacles étant l'irrégularité des branchements, les accès mémoires dépendants et les régions de codes contenants des séquences d'instructions dépendantes empêchant un parallélisme d'instruction raisonnable. Le Chapitre 3 de cette thèse traite particulièrement ce dernier point.

I.3 Extractions de fonctions à partir d'une séquences d'instructions dépendantes

Les différentes approches étudiées dans le Chapitre 2 proposent d'exécuter tout ou une partie d'un programme en projetant son flot de données correspondant sur un circuit. Ce circuit peut être reconfigurable ou composé de cellules programmables à grain plus gros. Cependant, les instructions dépendantes sont transformées en éléments matériels qui restent dépendants. Dans notre approche, nous traitons les codes qui contiennent des chaînes d'instructions dépendantes en les écrasant sous forme de fonctions combinatoires tout en évitant l'explosion exponentielle de la taille du circuit.

I.3.1 Principes

La Figure I.1 illustre un code C ainsi que les instructions correspondantes. Les instructions compilées sont toutes dépendantes et ne peuvent exécuter en parallèle.

Notre approche consiste à extraire les fonctions de sorties des graphes de flot de données de l'application (les sorties des nœuds n2, n3, n6 et n7 du graphe de la Figure I.1(c) correspondants aux écritures dans les registres r3, r4, r5 et le résultats de branchement respectivement) et les exécuter en parallèle sur des circuits combinatoires configurables comme le montre la Figure I.1(d). Au prix de calculs redondants, nous avons ainsi transformé une séquence d'instructions dépendantes en 4 fonctions qui exécutent en parallèle. Afin d'éviter le coup exponentiel de la fonction nous

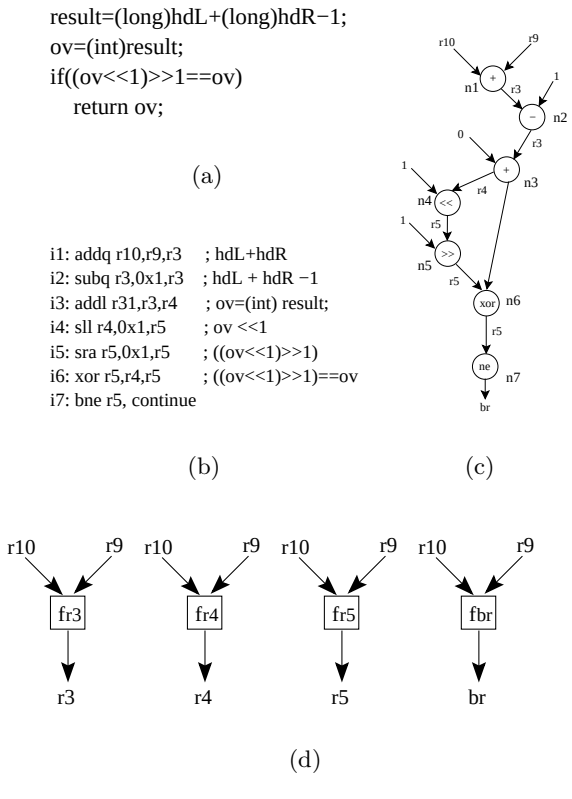


Figure I.1: *(a) Code C, (b) code assembleur, (c) graphe de flot de donnée, et (d) fonctions de sortie.*

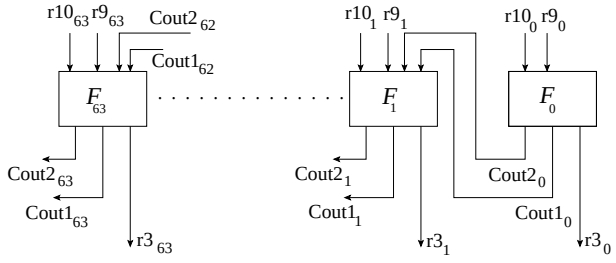


Figure I.2: *Opérateur matériel.*

avons proposé un circuits configurable ou chaque bit de sortie est associée à une fonction plus un réseaux de propagation de retenues comme le montre la Figure I.2.

I.3.2 Potentiel et limitations

Afin d'étudier le potentiel de notre approche nous avons analysé les codes SPECINT 2000 ainsi que certains des codes OLDEN [Rog95] et MIBENCH [Gut01]. Nous avons construit un outil qui extrait les fonctions des traces d'instructions exécutées, et nous en avons déduit la borne supérieure de l'accélération potentielle en mesurant les longueurs des chaînes d'instructions écrasées. Nous avons observé une accélération potentielle de 50% en moyenne et allant jusqu'à 132%. Bien qu'il existe de très longues chaînes d'instructions dépendantes, l'accélération maximale reste relativement limitée à cause des coupures introduites dans le graphe de flots de données. Les coupures de ces graphes sont les nœuds qui ne peuvent être écrasés avec les autres nœuds dépendants. Les cinq principales causes de coupures sont la taille de la trace considérée, les instructions de chargement de la mémoire, la limitation du nombre d'entrées au circuit reconfigurable, les instructions qui ne peuvent être écrasées en fonctions (les appels système par exemple) et finalement une contrainte du circuit configurable qui impose la propagation de la retenue du bit de poids faible au bit de poids fort. Nous analysons dans le Chapitre 3 chacune de ces contraintes en détail.

I.3.3 Implémentation

Dans l'étude du potentiel nous avons montré que, afin d'obtenir le plus de chaînes d'instructions dépendantes, il est important de pouvoir obtenir des traces dynamique d'exécution suffisamment grandes. De plus, il est important que l'extraction des fonctions et la configuration des unités de fonctions correspondantes ne soit pas dans le chemin critique de l'exécution. Afin de réunir ces deux conditions nous avons implémenté notre mécanisme dans l'environnement RePLay présenté par S. Patel et al. [Pat01]. L'environnement RePLay permet d'optimiser de larges traces d'exécution dynamique d'instruction dans une unité d'optimisation qui opère en dehors du chemin critique du processeur. Nous avons implémenté notre engin d'optimisation dans cette unité afin d'extraire les fonctions des traces d'instructions. La Figure I.3 montre l'architecture utilisée.

En optimisant 65% de toutes les instructions exécutées, nous avons obtenu des accélérations qui varient de 3,5% à 19% selon la latence supposée de l'unité de fonctions (nous avons étudié une latence variante de 1 à 6 cycles).

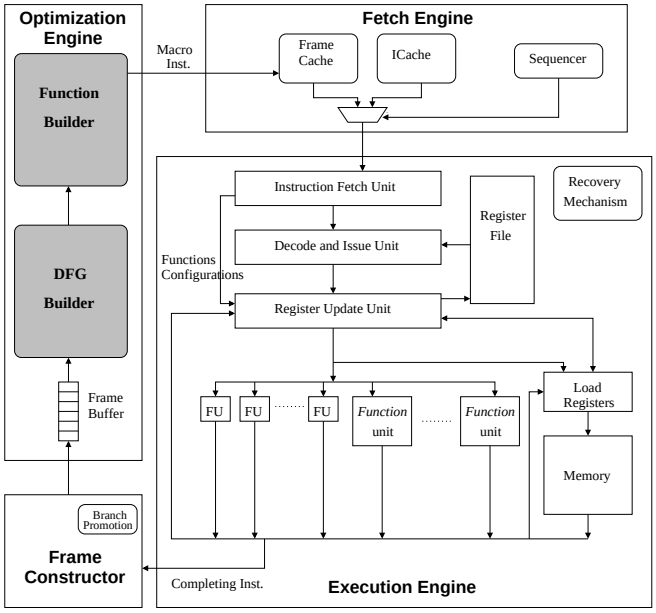


Figure I.3: *Architecture.*

I.3.4 Conclusion

Dans cette partie de la thèse nous avons présenté une approche matérielle qui exploite l'espace de la puce en écrasant des séquences d'instructions dépendantes dans des unités de fonctions configurables. Cette approche ne dépend ni de l'existence de parallélisme d'instructions ni de spéculation. Nous avons aussi remarqué que notre approche est relativement sensible à la capacité d'extraire de longues chaînes d'instructions ainsi que l'existence d'instructions de chargement dépendantes.

I.4 Double chargement: une méthode pour réduire la latence des chargements de donnée indirectes

Une des limitations de l'approche étudiée dans le Chapitre 3 est l'existence de chaînes d'instructions de chargement dépendantes qui empêchent la transformation de plus longues chaînes en fonctions à cause du chargement mémoire. Ce problème est d'autant plus aggravé par la latence croissante du chargement mémoire par rapport au temps de cycle du processeur. Dans ce chapitre nous proposons une approche pour réduire la latence des chargements indirects de la mémoire, notre approche

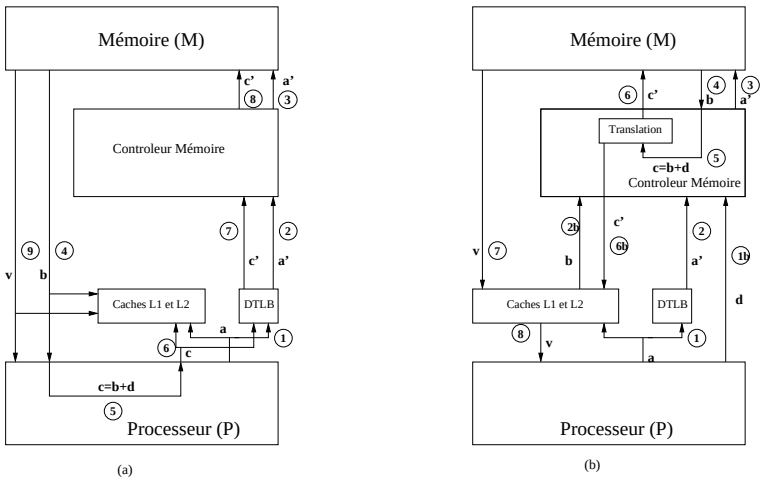


Figure I.4: *Double Chargement: (a) architecture normale, et (b) avec double chargement.*

se base sur (1) une méthode pour identifier les chargements indirects de mémoire susceptible de faire défaut dans les caches et (2) remplacer ces chargement indirects par une opération de double chargement qui permet d'éviter le double trajet vers la mémoire et ceci grâce à une logique spécialisée près de la mémoire (plus précisément dans le contrôleur mémoire).

I.4.1 Principes et implémentation

La squence d'instructions considérée par notre approche est sous la forme:

```
load b = [a]
add c = b + d
load v = [c]
```

Ces instructions correspondent à un accès mémoire indirect. L'idée principale est de remplacer la seconde instruction de chargement mémoire par une instruction double chargement qui peut être lancée sans attendre le résultat du premier chargement. Les instructions exécutent donc d'un manière équivalente à:

```
load b = [a]
load_squared v = [[a]+d]
```

La Figure I.4 décrit l'opération du chargement indirect sur une architecture conventionnelle (Figure I.4(a)) et sur une architecture supportant les opérations de double chargement (Figure I.4(b)). Dans la première architecture, deux allers-retours

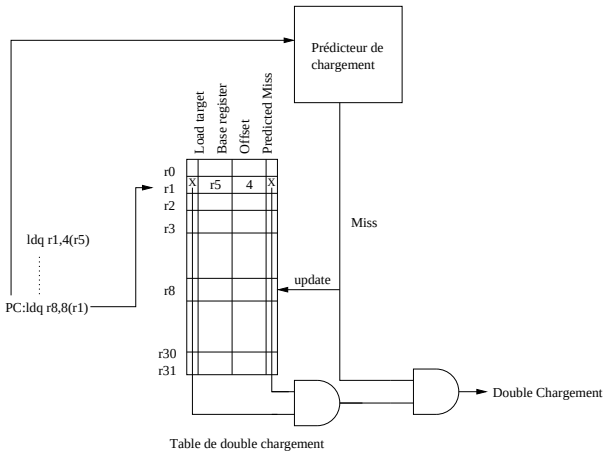


Figure I.5: *Détection de double chargement.*

vers et de la mémoire sont nécessaires pour obtenir la valeur v. Dans la deuxième architecture, nous ajoutons de la logique au niveau du contrôleur mémoire pour effectuer l'addition (b+d) à ce niveau plutôt que de l'effectuer dans le processeur. De cette manière un des deux allers-retours est fait seulement entre le contrôleur et la mémoire.

Afin de bénéficier de cette opération il est important de s'assurer que les deux chargements impliqués dans le double chargement font défaut dans les caches, sinon l'opération de double chargement peut être plus coûteuse en terme de latence que si les deux chargements étaient effectués à partir du processeur. Pour cela nous utilisons une table de double chargement pour détecter les chargements de données dépendants comme le montre la Figure I.5. Afin de pouvoir déterminer le comportement de chacun des chargements par rapport aux caches avant leurs exécution, nous associons à la table de double chargement un prédicteur de chargement qui prédit ce comportement et détermine si le second chargement doit être remplacé par le double chargement. Ce prédicteur de chargement fonctionne de la même manière que les prédicteurs de branchements utilisés dans les processeurs superscalaires.

I.4.2 Evaluation

Les codes analysés ont montré que environs 12% des instructions de chargement sont directement dépendantes d'une autre instruction de chargement. Néanmoins, peu d'entre eux font défaut dans le cache, ce qui rend le rôle du prédicteur très important pour détecter les doubles chargements appropriés. Le taux de bonnes prédictions de

chargements est de 87% ce qui permet d'avoir des accélérations substantielles pour certaines application: 5% pour twolf, 7% pour em3d et 50% pour ammp.

I.4.3 Perspective: Migration des calculs vers la mémoire

Dans cette dernière partie de la thèse nous généralisons l'approche du double chargement en migrant de plus larges parties de codes vers la mémoire. Notre approche cible les accès de structures de donnée irrégulières et se base sur d'une part, une séparation explicite des accès de données et leur traitement via une extension de langage, et, d'une autre part une architecture découplée qui utilise une mémoire intelligente pour exécuter les partie accès mémoire du code.

Etat de l'art

Notre approche étend deux type de travaux présentés dans la littérature: les architectures découplées et les mémoires intelligentes. Les architectures découplées sont des architectures qui séparent les instructions de calculs des instructions d'accès mémoire et exécutent les deux types en parallèle dans un schéma producteur / consommateur. Smith et al. [Smi84] ont proposé une des premières architectures découplées qui consiste en deux processeurs scalaires l'un pour exécuter les calculs, l'autre pour effectuer les accès mémoires. Les deux communiquent entre eux par des files matérielles. Le processeur ZS-1 [Smi87] sépare le flot d'instructions en deux pipelines, l'un pour les calculs flottants, l'autre pour les calculs entiers et accès mémoires. Pleszkun et al. [Ple86] ont présenté une des rares architectures découplées qui traitent directement les structures de donnée complexes, en particulier les listes chaînées. L'architecture HiDISC [Ro03] propose une architecture découplée ainsi qu'un compilateur qui sépare les instructions de calculs et d'accès. Finalement, Roth et al. [Rot00] proposent d'extraire les instructions causant beaucoup de défauts dans les caches dans un flot d'exécution séparé exécutant à l'avance de l'exécution normale afin de fournir aux caches les données nécessaires pour le flot principal d'exécution.

En plus des architectures découplées, l'approche que nous proposons est motivée par l'émergence des mémoires intelligentes. Les mémoires intelligentes, tels que IRAM [Pat97], intègrent de la logique pour effectuer du calcul dans la mémoire même. Cette technique permet d'atteindre environs 100 fois plus de bande passante ainsi que 10 fois moins de latence.

La DSCM: une machine pour les structures de données complexes

La *Data Structure Conscious Machine* (DSCM) est une architecture découplée dans laquelle les opérations relatives aux accès mémoire sont effectuées dans une mémoire

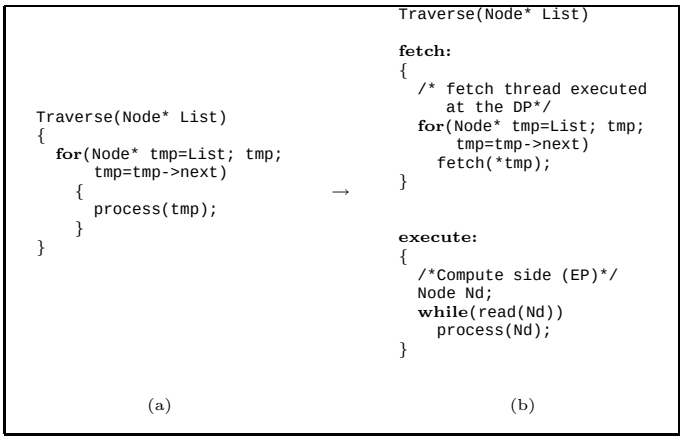


Figure I.6: *Traversée d'une liste chaînée dans l'architecture découplée.*

intelligente pour bénéficier de la large bande passante et la latence réduite. La particularité de notre approche est la séparation explicite des flots d'exécution et d'accès mémoire grâce à un langage d'extension.

Dans l'extension de langage que nous proposons, l'utilisateur défini explicitement les flots d'exécution qui concernent le processeur et ceux qui concernent la mémoire intelligente. La Figure I.6 montre un exemple d'une traversée d'une liste chaînée en utilisant l'extension de langage proposée: la traversé de la liste est scindée en deux flots d'exécution distincts, l'un qui exécute dans le processeur principal, l'autre dans la mémoire. Cette approche explicite garantit un partitionnement approprié, car l'utilisateur détient la sémantique qu'un compilateur ne peut pas toujours déduire. La Figure I.7 montre l'architecture DSCM: nous proposons deux processeurs distincts, l'un d'entre eux, le processeur de données, est une mémoire intelligente tel que l'IRAM. Des files matérielles permettent de communiquer les données entre les deux processeurs. Chaque processeur est multiflot afin de permettre à l'utilisateur de définir plusieurs flots exécutants en parallèle de part et d'autre.

Nous avons effectué des expériences préliminaires sur des boucles élémentaires pour étudier notre approche, et nous avons pu obtenir des accélérations allant jusqu'à 5 fois pour une traversé d'une liste chaînée, et 3 fois pour une multiplication de matrice.

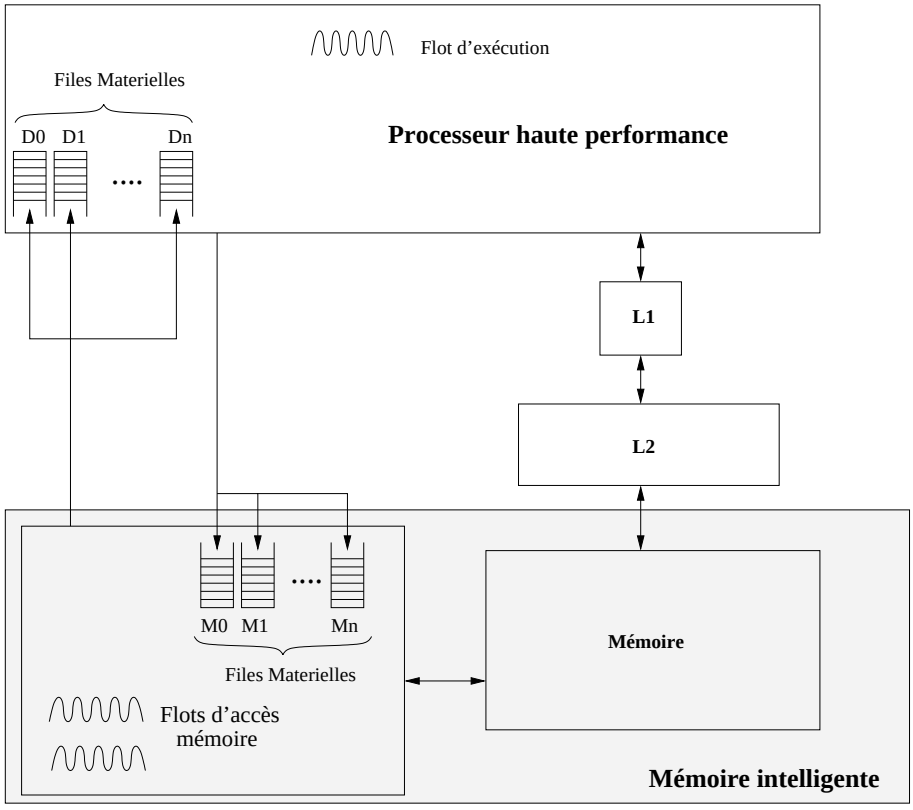


Figure I.7: *Architecture DSCM.*

I.5 Conclusions

Nous avons exploré dans cette thèse plusieurs approches alternatives pour mieux exploiter l'espace disponible sur la puce et réduire l'effet de la latence croissante par rapport au processeur. Dans un premier temps nous avons présenté une approche qui extrait les fonctions de sortie des suites d'instructions dépendantes afin de surmonter le manque de parallélisme d'instructions. Ces fonctions sont exécutées sur des unités de calculs configurables sur la puce.

Dans un deuxième temps nous avons remarqué que l'extraction des fonctions était limitée par les instructions de chargement, notamment les instructions de chargement dépendantes. Nous avons donc proposé une approche qui identifie les pairs d'instructions de chargement dépendantes, et remplace la dernière par une instruction de double chargement qui effectue le calcul d'adresse proche de la mémoire afin d'économiser le chemin aller-retour de et vers la mémoire.

Finalement, nous avons étudié la généralisation de cette approche en proposant une architecture découplée dont la partie accès mémoire est une mémoire intelligente qui bénéficie d'une latence d'accès relativement réduite. Les résultats préliminaires de cette approche sont prometteurs.

Les technologies d'intégration actuelles et futures permettrons sans doute une ou plus de combinaisons des différentes approches discutées. De plus les nouvelles technologies tel que les nanotechnologies ou les mémoire magnétique ouvrirons la voie à plus d'espace de calcul sur la puce de même que plus de défis architecturaux tel que la consommation d'énergie et la tolérance aux fautes.

Publications durant la thèse

- Sami Yehia, Jean-Franois Collard and Olivier Temam, "Load Squared: Adding Logic Close to Memory to Reduce the Latency of Indirect Loads with High Miss Ratios," MEDEA Workshop, held in conjunction with the International Conference of Parallel Architectures and Compilation Techniques (PACT), October 2004.

- Sami Yehia and Olivier Temam, "From Sequences of Dependent Instructions to Functions: An Approach for Improving Performance without ILP or Speculation," 31th Annual International Symposium on Computer Architecture (ISCA), June 2004.

- Sami Yehia and Olivier Temam, "From Sequences of Dependent Instructions to Functions: A Complexity Effective Approach for Improving Performance without ILP or Speculation," 4th Workshop on Complexity-effective Design

(WCED) held in conjunction with the 30th Annual International Symposium on Computer Architecture (ISCA), June 2003.

Chapter 1

Introduction

Current integration technologies and advances in semiconductor manufacturing open the way to an unprecedented number of transistors on a single processor die. Technological improvements have allowed the number of transistors per chip to be doubled every 18 months and performance to double every two years [Pat90]. With this potential technology, the role of computer architecture in achieving best performance and meeting functional and cost goals became predominant.

1.1 Computer Architecture Challenges

While integrated circuit technology is reaching one billion transistors on a single chip [Bur04], the major challenge remains in the best way to exploit this enormous level of integration. This challenge is principally addressed by exploiting all forms of parallelism. The major forms of parallelism are Instruction Level Parallelism (ILP), Data Level Parallelism (DLP) and finally Thread Level Parallelism (TLP).

The advent of superscalar processors addressed ILP by allowing the execution of more than one instruction per cycle. Then several SIMD instruction set extensions such as SSE3 [Hin01], Altivec [Mot99] and 3DNow![AMD00] were added to architectures to achieve DLP, and finally SMT architectures [Tul95] enabled several threads to execute in parallel. On-chip space could be efficiently exploited for applications that have high data level parallelism and regular data access patterns as well as predictable control flow, through aggressive speculation and deeper pipelining.

Still, current architectures can do little with applications having complex data structures or irregular data access patterns. Integer applications particularly suffer from such properties. Agarwal et al. [Aga00] showed that, unless applications are mapped differently on the chip, the achievable performance growth of conventional

microarchitectures will slow substantially due to both diminishing improvements in clock rates and poor wire scaling as semiconductor devices shrink.

The main bottlenecks lying in non-numeric applications are the low ILP and irregular data structures:

- Jouppi et al. [Jou89] showed that aggressive pipelining and superscalar processors have little effect on the performance of many non-numeric applications because they have little ILP. The existence of chains of dependent instructions that cannot execute in parallel is one of the major bottlenecks in non-numeric applications. These region of codes having low ILP cannot benefit from the SIMD optimizations, and agressive ILP support in superscalar processors.

- While integration and processors technology have led to a near-exponential increase in processor speed and memory capacity, memory latencies have not improved as dramatically, and the growing gap between processor speed and memory latencies is increasingly limiting applications performance. This problem is know as the *Memory Wall* problem [Wul95]. While this problem is partially addressed through data prefetching, address and value prediction, a lot of work has yet to be done for linked data structures and pointer chasing problems.

1.2 Contributions

In this thesis we propose alternative approaches to exploit on-chip space and reduce the memory wall effect, our main contributions are:

- We present a novel approach to improve the performance of codes that have little or no ILP, by collapsing sequences of dependent instructions to functions, we also address the problems and limitations of the approach.

- The collapsing approach being limited by memory accesses in sequences of dependent instructions, we propose an approach to improve the performance of dependent loads that have high miss ratios by adding logic closer to memory.

- We investigate a generalization of this concept by presenting a decoupled architecture associated with a language extension to migrate parts of applications that intensively access the memory closer to memory.

1.3 Thesis Organization

In Chapter 2, we present a state of the art of several approaches that exploit on-chip space differently, we split these approaches in two categories: reconfigurable

architectures, and grid architectures. Both directions map applications on the chip in a spatial way, the main difference between them lies in the granularity and the mapping strategy. We also discuss the strength and limitations of the different approaches as well as the motivation to our work.

In Chapter 3, we present an alternative approach to improve the performance of sequences of *dependent* instructions. We observe that many sequences of instructions can be interpreted as *functions*. Unlike sequences of instructions, functions can be translated into very fast but exponentially costly two-level combinational circuits. We present an approach that exploits this principle, speeds up programs thanks to circuit-level parallelism/redundancy, but avoids the exponential costs. We analyze the potential of this approach, and then we propose an implementation that consists of a superscalar processor with a large specific functional unit associated with specific back-end transformations. We also discuss the limitations of the approach as well as its potential performance improvement in an idealistic environment.

Because the collapsing approach is limited by dependent memory accesses, we propose in Chapter 4 an approach to improve the performance of indirect memory accesses having high miss ratios. To reduce the total latency of such accesses, a new operation called *load squared* is introduced. This operation is performed by memory-side logic, typically the memory controller. We present a method to detect indirect loads that frequently miss in caches, and dynamically replace them with load squared.

We conclude this chapter with a perspective that extends the concept of migrating the address calculation by migrating part of the application to an intelligent memory. We present a novel decoupled architecture that decouples memory accesses of complex data structures from computations. We propose an explicit decoupling through an extension to C language, so that the user can explicit which parts of the application to migrate. Then we show that our approach can also be applied to SMT processors without resorting to decoupling, by executing memory access parts of the application in a separate thread in an SMT processor. A general overview of the proposed architecture as well as the instruction set extention is given in Appendix A.

We summarize our research work and propose several directions for future research in Chapter 5.

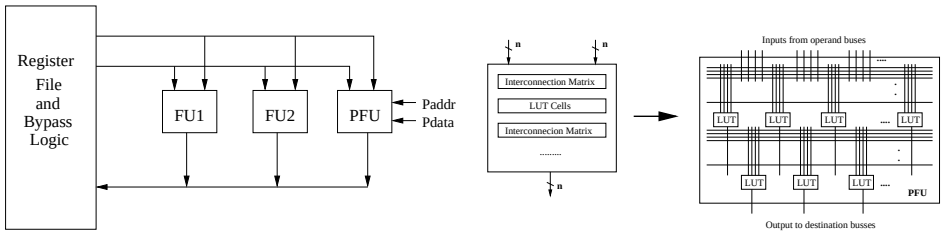
Chapter 2

Exploiting On-chip Space for Computations

Current and upcoming architectures devote available on-chip space to more aggressive execution via pipelining, and exploiting all forms of ILP mainly through speculation. Nevertheless, a lot of emerging architectures attempt to use on-chip space differently. Current processors already devote space to computations through SIMD instructions set extensions such as the Streaming SIMD Extension 3 (SSE3) in the Intel Pentium 4 processor [Hin01], the Altivec in PowerPC [Mot99] and the 3DNow! extension in AMD processors [AMD00]. Nevertheless, such approaches rely heavily on regular access of data in multimedia application and are not always adequate for complex data structures. Two different but complementary approaches exploit more efficiently on-chip space for computation: reconfigurable architectures devote additional space to customizable functional units that may be configured by the user (or the compiler). These architectures can efficiently execute short sequences of operations by mapping them on logic circuits. The second approach, grid architectures, tries to better use space in processor architectures, by mapping computations on arrays of ALUs or simple processors.

2.1 Reconfigurable Architectures

Reconfigurable computing offers the potential to achieve partially the performance of application specific integrated circuits (ASIC) and the flexibility of general purpose processors. By dedicating part of the on-chip space to programmable devices, part of the application may execute more efficiently on this specialized hardware than on a traditional superscalar pipelined processor. The advent of field-programmable gate



(a) PRISC-1 datapath (b) PFU architecture

Figure 2.1: *PRISC architecture.*

array (FPGAs) in the mid-1980s opened the way to applications of reconfigurable computing. FPGAs are arrays of programmable computational elements whose functionality is determined through multiple programmable configuration bits. Reconfigurable computing and FPGAs technologies have been extensively covered and discussed in [Com02; MS97; Tes01; Hau98; Ros93; DeH99; DeH00].

In this section we review the literature on reconfigurable computing from a processor architectural perspective, and especially how it is used in general purpose systems. PRISM [Ath93] is one of the first general purpose systems that compiles and maps critical parts of an application to a reconfigurable co-processor. Below, we present general purpose architectures that partially or completely map the application on programmable devices.

2.1.1 PRISC

The Programmable RISC (PRISC) [Raz94] augments a conventional RISC instruction set with application-specific instructions that are implemented in hardware-programmable functional units (PFUs) as shown in Figure 2.1(a). Physically, the instruction set is extended with a single new instruction containing an index to a programmed PFU. PFUs implement two-input, one-output combinational functions that are defined and generated by the compiler. The compiler uses profiling information to identify potential sequences of instructions to be mapped to PFUs.

The PFU is comprised of alternating layers of two basic components: interconnection matrices and logic evaluation units, See Figure 2.1(b). Those layers are constrained to allow the PFU to execute in one processor cycle. Multi-cycles PFUs are also proposed in [Raz].

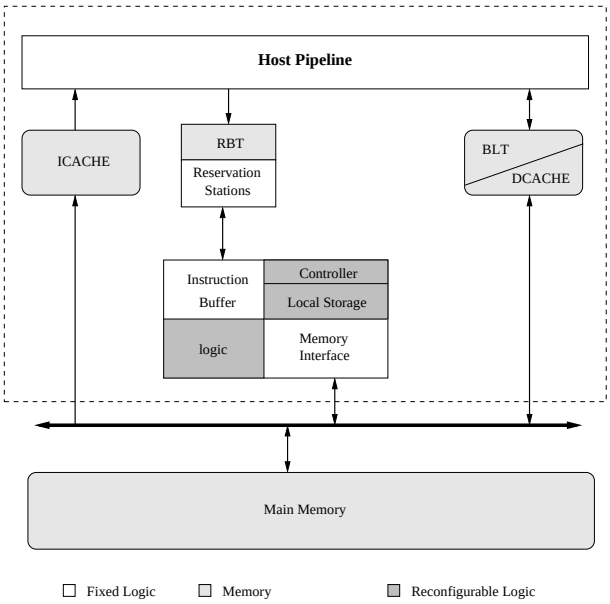


Figure 2.2: *OneChip architecture.*

2.1.2 OneChip

OneChip [Wit96] tightly integrates reconfigurable logic resources and memory into a fixed-logic out of order processor core. OneChip allows the user to transform sequences of complex operations to a single operation that is mapped to the reconfigurable unit. This mapping enables a faster execution of those complex operations. Also, the reconfigurable logic directly interfaces with memory to get its operands, allowing a SIMD-like processing of large blocks of data. Multimedia and streaming applications particularly benefit from this approach. Furthermore, this direct access allows OneChip to execute in parallel with the CPU.

The FPGA reservation stations allow several *macro operations* to operate concurrently on the FPGA [Car01]. Also, OneChip defines a memory consistency scheme in [Jac99] to allow parallel execution between the reconfigurable logic and the CPU. The memory consistency is implemented using a Block Lock Table (BLT) to lock memory blocks. The scheme may also support more than one FPGA.

Figure 2.2 shows the OneChip Architecture. The Reconfiguration Bit Table (RBT) acts as the configuration manager that will keep track of where the FPGA configurations are located. This role is similar to a TLB as configurations may be swapped between the FPGA and the memory. The FPGA may have multiple

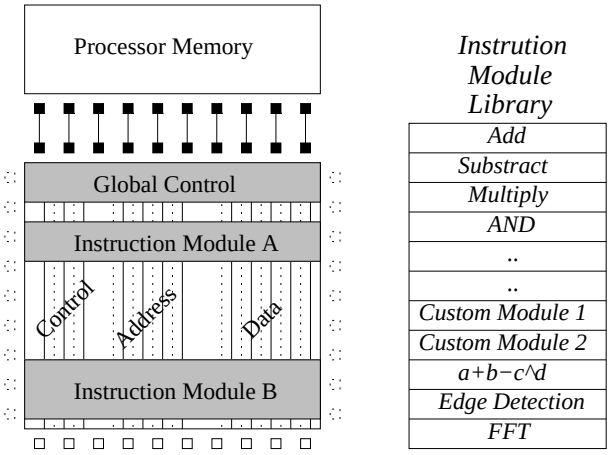


Figure 2.3: *DISC linear hardware space.*

contexts and can *cache* more than one configuration [DeH94].

Unlike other approaches, OneChip does not rely on a compiler. Instead, the user defines the configurations of the FPGA, which executes the operation with a single RISC instruction. This instruction points to the address in memory where the FPGA configuration to be loaded is stored.

While the major asset of allowing the user to define complex operations is the performance gain of such mapping, OneChip still requires to recompile and rewrite the applications in order to take advantage of this approach. Also, due to the complexity in terms of routing and reconfigurability of the FPGA, it is not always easy to configure complex operations so that they execute faster than their equivalent instructions on an out-of-order superscalar processor.

2.1.3 DISC: A Dynamic Instruction Set Computer

DISC [Wir95] is a loosely coupled reconfigurable architecture that treats instructions as removable modules paged in and out. Customizable instructions are fully mapped on the FPGA as shown in Figure 2.3. DISC dedicates the largest portion of the die to customized instructions and may map complex libraries of instructions.

The DISC processor uses run-time reconfiguration to overcome FPGA hardware limitations. The configuration overhead is reduced by partially reconfiguring the die while other configured instructions execute.

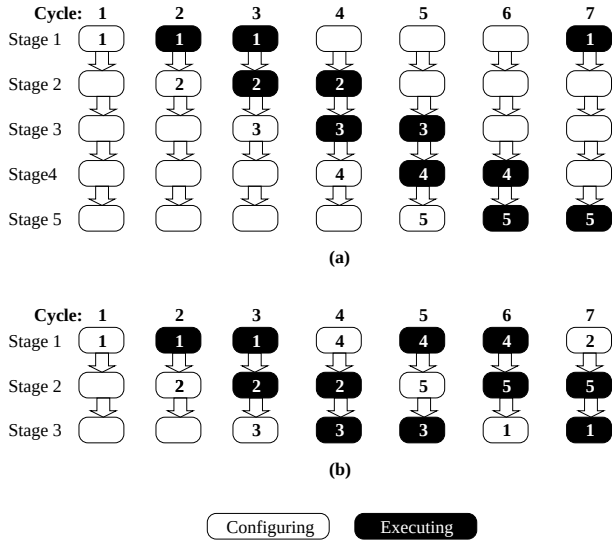


Figure 2.4: *Pipeline reconfiguration showing the virtualization of a five-stage pipeline on a three-stage device:(a) the virtual pipeline stage and (b) the physical pipeline stage (the numbers in the ovals refer to the virtual pipeline stage.)*

2.1.4 PipeRench

Although PipeRench is presented with other reconfigurable systems, PipeRench shares with Grid architectures, discussed in Section 2.2, the relative coarse granularity of its reconfigurable cells. PipeRench [Gol00] [Gol99] works as an attached co-processor and consists of a reconfigurable fabric: an interconnected network of reconfigurable logic and storage elements - not necessarily FPGAs. In fact, PipeRench overcomes the cost issues of implementing a circuit using FPGA cells by implementing a large logic circuit on a small number of FPGA cells through their rapid reconfiguration. This technique is denoted as *hardware virtualization*, see Figure 2.4. Hardware is virtualized by breaking a single static configuration into pieces that correspond to pipeline stages in the application. Pipeline stages are then mapped to a pipeline of reconfigurable fabrics with a number of stages that may be lower than those configured. This technique is scalable because it is a linear function of the device's capability: increasing the number of hardware stages allows a higher throughput. PipeRench is suitable for stream-based media applications and regular fine grain computations.

PipeRench relies on a compiler that creates efficient reconfigurations through a *dataflow intermediate language* (DIL): a single assignment language with C opera-

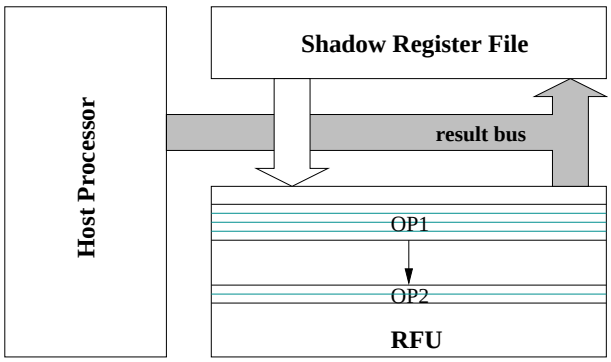


Figure 2.5: *The overall Chimaera architecture.*

tors.

Beside streaming multimedia regular applications, Chou et al. [Cho00] proposed to use PipeRench as an *Instruction Path Co-processor* (I-COP) which is a programmable co-processor that operates on the core processor's instructions to optimize and transform them into a new format that can be more efficiently processed by fast execution cores. The I-COP replaces the *Fill unit* [Fri98] of the trace cache, the unit that collects traces to be stored in the trace cache. Beside efficient trace collection, The I-COP proposes to implement further optimizations on collected traces such as the register move optimization and adding prefetch instructions for strided memory accesses and linked data structures (LDS).

2.1.5 The Chimaera architecture

Chimaera [Ye00a; Hau97a] integrates a reconfigurable logic into the host processor, See Figure 2.5. Sequences of instructions are mapped on an array of reconfigurable cells on a line basis, where each operation takes one or more lines in the array. In other words, sequences of instructions are *collapsed* and mapped to one or more line in the array. Reconfigurable operations (RFUOPs) are extracted using the Chimaera compiler. Chimaera allows multi-operand instructions and support partial run-time reconfiguration to reduce reconfiguration time. The Reconfigurable Functional unit (RFU) is a reconfigurable array which caches RFUOPs, relying on the reusability of reconfigurable operations. A reconfigurable instruction is a 9-input/1-output instruction. When the RFUOP is in the RFU, it does not have to wait for the occurrence of an RFU call in the instruction stream to begin executing, since it already knows which registers it needs to access. Nevertheless, care should be taken in RFU mapping creation and register assignment: for example, if an RFUOP takes

as input register R1, R2 and R3, any subsequent reuse of this RFUOP must use these register inputs.

An interesting aspect is that there is no state-holding elements in the reconfigurable array of Chimaera [Hau97a]. Instead, the host processor register file is used as the only storage element of the system. This aspect is novel, because it allows a *combinational* processing of one or more instructions.

The Chimaera compiler [Ye00b] performs the following optimizations:

1. Instruction combination: Chimaera extracts RFUOPs from a sequence of instructions with no intermediate control flow.

2. Control localization: Chimaera compiler combines branch-containing code sequences as a single unit called *macro-instruction* during assignment and scheduling. This technique is also used in PRISC [Raz94] and the Raw processor [Lee98].

3. SWAR optimization: SIMD Within A Register identifies sub-word operations to be executed in parallel. Chimaera can pack 8-bit operations into a single-word operation.

Because the reconfigurable operations are extracted at compile time, the Chimaera architecture cannot span multiple basic blocks, so potential complex operations that may depend on a certain control flow are not considered. Also the mapped operations are still dependent in the reconfigurable array, so the operations are not really collapsed.

2.1.6 The Garp architecture

Garp [Hau97b] combines a single issue MIPS processor core with a reconfigurable array (similar to DISC) to be used as an accelerator. Garp accelerates loops of general purpose programs and reconfigures the array only at the entrance and exit of loops (unlike DISC that relies on partial reconfiguration of instruction modules). So the Garp array executes independently whole loops. Like OneChip, Garp also has its own direct path to the processor's memory system.

The most important aspect of the Garp project is that it takes standard ANSI C as input, without the need to insert any hints or directives in the source code. The Garp compiler [Cal00] identifies code sections that can be accelerated using the Garp array. The compiler then forms *hyperblocks* by joining all the basic blocks along the frequently executed control paths of the loop body, excluding all uncommon paths and codes that cannot map to the array. When a branch not belonging to the frequently executed control path is taken, an exceptional exit from the array occurs,

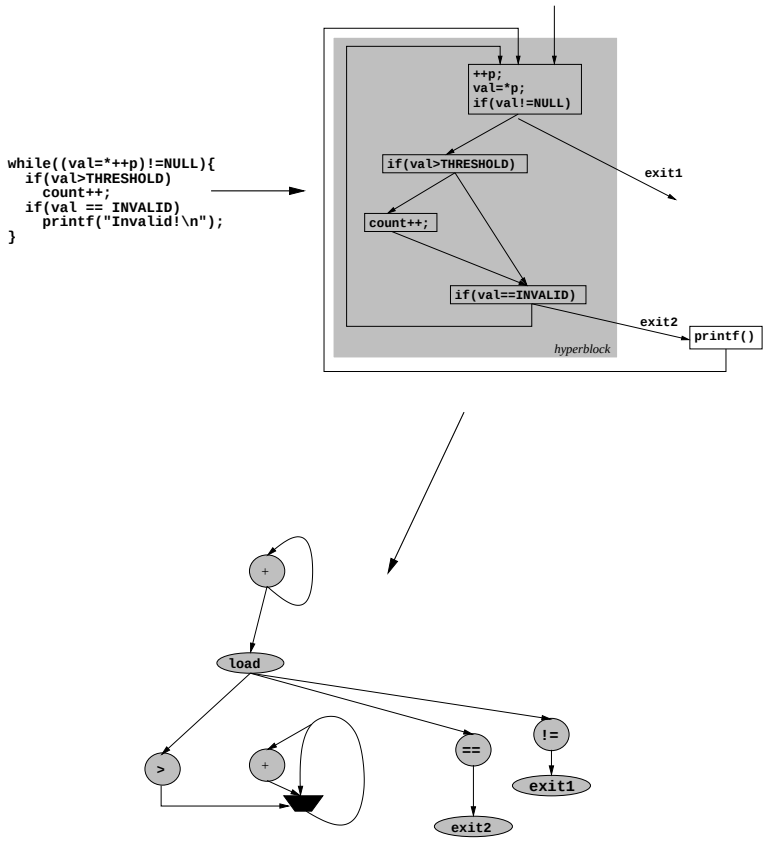


Figure 2.6: *Garp compiler's inner process.*

Architecture	Computation selection	Rely on compiler	Computation granularity	Memory interface	Inst. Set extension
PRISC	Profiling	Yes	Control localization	None	Add an instruction
OneChip	Hand coded	No	Any function	Direct	Add an instruction
CHIMAERA	Code analysis	Yes	Control localization	None	RFUOPs
PipeRench	Code analysis	Yes	Basic Block	None	Own Inst. Set
DISC	Hand coded	No	Any	Direct	None
Garp	Profiling	Yes	Hyperblocks, Control localization	Direct	Few instructions added

Table 2.1: *Summary of reconfigurable architectures.*

see Figure 2.6. Like Chimaera, Garp merges all the included basic blocks using predication, allowing operations from different basic blocks to be brought together into a single large DFG, mapped to the FPGA.

2.1.7 Summary of reconfigurable architectures

Reconfigurable architectures could substantially improve performance of a lot of regular applications. However, the fine granularity of the previously described mechanisms, especially those relying on FPGAs, have important drawbacks, namely the configuration time, hardware constraints and compilation time [Gol99]. For example while Garp, Chimaera, PRISC and PipeRench require a lot of effort to develop an optimizing compiler, DISC and OneChip require that the user explicitly defines the functions to be mapped to the reconfigurable device, which leads most of the time to poorly optimized circuits. Nevertheless, PipeRench offers a good compromise by proposing a language, thus allowing the user to define the functions to be executed so that the compiler can efficiently map it to the reconfigurable device. Table 2.1 summarizes some architectural aspects of reconfigurable architectures.

2.2 Cell Architectures

Because a lot of programs have little or no ILP, and fail to exploit instruction parallelism in superscalar processors, new directions suggest better exploiting on-chip space by building arrays of simple processing elements (cells) but of coarser grain than FPGA. In general, each cell or *tile* includes at least one ALU, one or

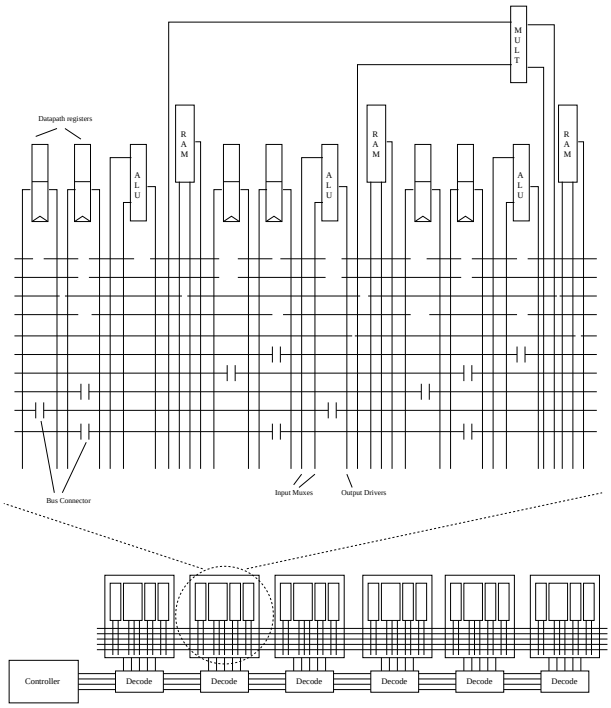


Figure 2.7: *RaPiD architecture.*

more registers and probably a small memory and a sequencer to execute simple instructions on each tile. More complex tiles may include a full simple pipelined processor. Furthermore, cell arrays must contain routing structures. We notice that by attempting to map program DFGs on a network of cells, the processor architecture exploits on-chip space in a manner closer to FPGAs than centralized control.

Taylor et al. [Tay03] called such cell architectures *scalar operand networks*, and identified the major challenges in such architectures, namely delay scalability, bandwidth scalability, deadlock and starvation, efficient operation-operand matching and exception handling.

In this Section we discuss three cell or grid architectures: RaPiD, Raw and GPA.

2.2.1 RaPiD : Reconfigurable Pipelined Datapaths

RaPiD [Ebe96] is one of the first *coarse grain* configurable architectures that may be classified as a cell architecture. RaPiD implements a pipelined architecture of

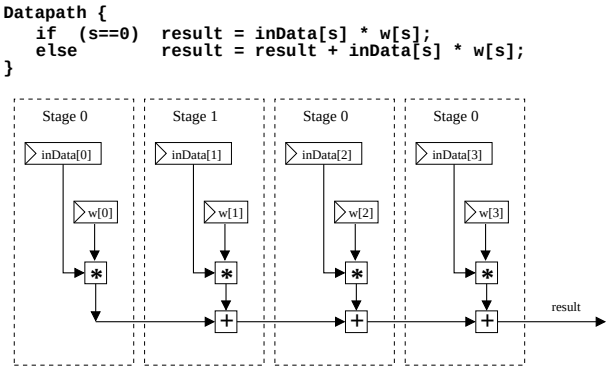


Figure 2.8: *RaPiD-C.*

cells which distribute computations over pipeline stages. RaPiD is a linear array of functional units which is configured to form a mostly linear computational pipeline. The array of functional units is divided into identical cells which are replicated to form a complete one-dimensional array. Figure 2.7 shows RaPiD architecture and a cell which includes an integer multiplier, three integer ALUs, six general-purpose *datapath registers* and three small local memories. A typical single-chip RaPiD array would contain between 8 and 32 of these cells. The functional units are interconnected using a set of segmented buses that run the length of the data path.

Due to the special nature of pipelined computations, Ebeling et al. [Ebe03] proposed RaPiD-C: an extension to C which allows to describe the operation of each pipeline stage for one data element passing through the pipeline. Figure 2.8 shows a vector dot product using RaPiD-C and how it is mapped to the RaPiD architecture. The statements in the `Datapath` blocks are executed for all values of s from `0` to `STAGES-1`.

Proposing an adequate language extension that fits on the proposed architecture is one of the principal merits of RaPiD. Still, the approach deals with regular streaming applications and fails to properly map algorithms with complex data structures manipulations.

2.2.2 Raw machines

Raw [Wai97] is a simple highly parallel replicated architecture directly exposed to the compiler. Figure 2.9 shows the Raw architecture, each tile contains a simple, eight-stage, in-order, single-issue RISC processor and is interconnected with other tiles over a pipelined, point-to-point network. The implemented Raw processor has

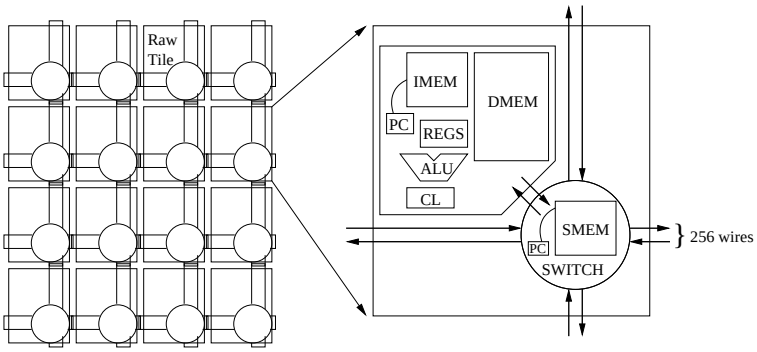


Figure 2.9: *A Raw microprocessor.*

16 tiles [Tay02] [Tay04]. Memory banks are distributed along with the processing elements. Each bank is directly addressable by its local processing element, without going through a layer of arbitration logic. Remote memory accesses are performed through two general inter-tile interconnects: a fast static networks for compilable analyzable accesses and a slower, fail-safe dynamic networks. Each tile includes two sets of control logic: the first set controls the operations of the processing element and the second is dedicated to sequencing routing instructions for the static switch.

Like most parallel computers, Raw relies heavily on the compiler to discover and statically schedule the parallel execution of instructions, but Raw machines can provide multiple instruction streams. Computations are mapped to each processing element statically, therefore the compiler manages data movements. The Raw compiler (Rawcc) comprises two major components. The *Maps* compiler described in [Bar99][Bar01] is the memory front end that performs *Bank disambiguation*, i.e. determines at compile-time which bank a memory reference is accessing. The Maps compiler partitions all memory references into equivalent classes to achieve high data locality on the tiles. The space-time scheduler [Lee98] is the back end of Rawcc. It maps instruction level parallelism to the Raw tile and maps each equivalent class of data objects to the memory bank of a specific Raw tile.

2.2.3 GPA: Grid Processor Architectures

The GPA [Nag01] is an array of ALUs each with limited control to which blocks of statically scheduled instructions are mapped, and executed dynamically in dataflow order. The tiles of the GPA are simpler than those of the Raw architecture, each tile of the *virtual* grid executes a single instruction. Physically, because the number of tiles are limited, each tile contains several *frames*, a frame may be considered as

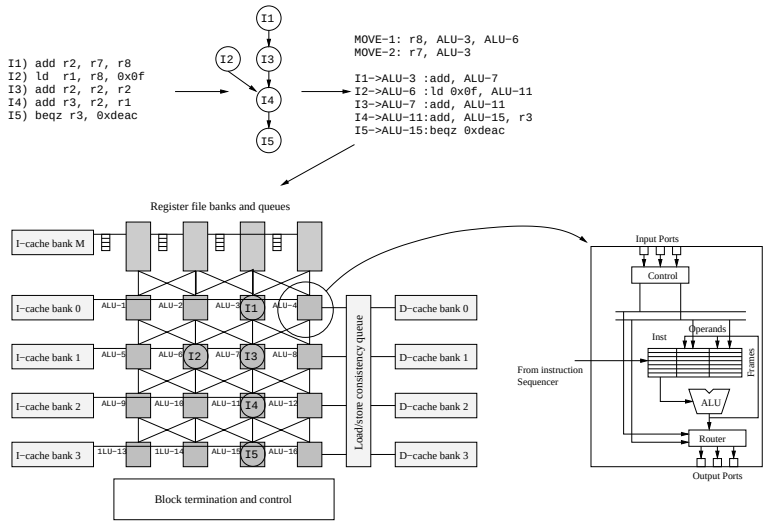


Figure 2.10: *GPA.*

a single virtual cell of the grid.

A compiler is also used to detect parallelism, build hyperblocks [Mah92] which are predicated, single-entry, multiple-exit regions, and to schedule instructions. Figure 2.10 shows a hyperblock mapped to the GPA. GPA allows direct communication between producing and consuming instructions, thus temporary results are not written back to the register file. This way, dependent instructions are processed much more faster because they are directly mapped to the array, unlike superscalar processors where an instruction must wait for the instruction it depends on before it is dispatched on a functional unit.

The TRIPS architecture [San03] elaborates on the GPA and contains four out-of-order, 16-wide issue GPA, as shown in Figure 2.11. TRIPS introduces the concept of *polymorphous architecture* as it can be configured and combined to support different modes of parallelism: instruction (D-morph), thread (T-morph) and data (S-morph) level parallelism:

D-morph. The *Desktop morph* uses the polymorphous capabilities of the processor to run single-threaded codes by exploiting ILP. The compiler schedules hyperblocks into a 3-D region, assigning each instruction to one node in the 3-D space, Figure 2.12 shows two scheduled hyperblocks scheduled, where each frame is limited by the physical size of the ALU array. Each hyperblock is mapped to an *A-frame*.

T-morph. TRIPS proposes two strategies to support TLP, *row processors* in which each thread is allocated one or more rows of the array, and *frame processors*

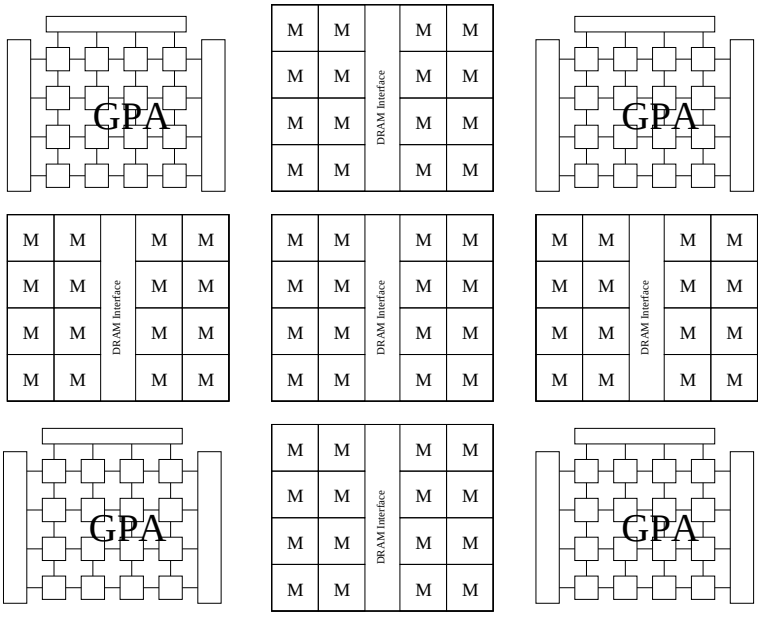


Figure 2.11: *TRIPS architecture.*

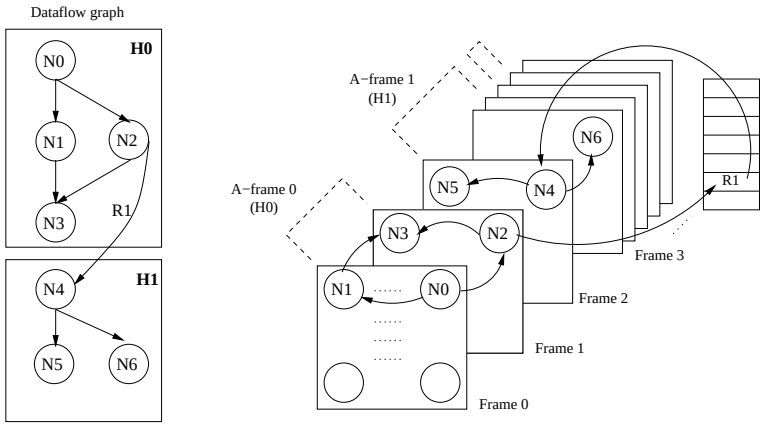


Figure 2.12: *D-morph frame management.*

in which threads are allocated to unique sets of frames.

S-morph. To support high data level parallelism (DLP) for streaming media and scientific applications, the S-morph fuses multiple A-frames to make a *super A-frame*, instead of using separate A-frames for speculation or multithreading. This fusion is motivated by the highly predictable control flow of these applications.

2.3 Problems and Limitations

Reconfigurable and grid architectures are different approaches that exploit on-chip space in computations rather than in more complex speculated-oriented mechanisms as in superscalar processors. Streaming and multimedia applications, more generally regular computations particularly benefit from such approaches. Still many applications, especially integer, pointer-intensive applications fail to benefit from such approaches, the three main obstacles being control flow (or branches), memory accesses and instruction dependencies.

Control Flow / Branches. To exploit on-chip space efficiently, it is important to have large portions of computations that may be directly mapped to the chip. This condition may be fulfilled with regular, highly predictable applications. When the application contains branches that are not predictable, which may be a frequent case when traversing pointer-based data structures (branches are often data-dependent), it is hard to anticipate which computations may be mapped to the die. A lot of the approaches discussed in this chapter such as Chimaera and Garp could partially circumvent or reduce the effect of this obstacle through control localization and hyperblocking. Nevertheless, mapped computations are still limited by control flow.

Memory Access. Memory load instructions account for nearly 25% of all executed instructions in the SpecInt 2000 benchmarks. Memory accesses are particularly harmful when they are chained because it is not possible to fetch a large chunk of instructions and their input data, then execute it on the grid; it is necessary to frequently access memory from the cells/grid.

Instruction dependencies. Reconfigurable computing and grid architecture partially handle this obstacle by executing instructions in a dataflow manner, thus reducing the time instructions wait due to dependencies. For example As stated in Section 2.1.5, all instructions dependencies must be resolved before an RFUOP can execute. In the next chapter we discuss a method that attempts to address these issues.

More generally, the potential performance gains of many of the discussed reconfigurable or cell architectures are only partially exploited, because these approaches only focus on the architecture while one of the core issues is the difficulty to extract information on the parallelism and locality properties of such architectures. We believe that more efforts should be devoted at the programming language level, especially to extract additional program semantic in order to fully exploit the potential of such architectures. We further explore this issue in the Data Structure Conscious Machine discussed in Chapter 4.

Chapter 3

From Sequences of Dependent Instructions to *Functions*: An Approach for Improving Performance without ILP or Speculation

3.1 Introduction

Current and upcoming processors heavily rely on increasing instruction throughput through pipelining and exploiting all forms of ILP. The additional on-chip space that comes with each new processor version is increasingly devoted to these techniques (larger pipelines, larger branch prediction tables, larger caches, larger instruction windows and reservation stations...) rather than to computing resources themselves (functional units). Besides, throughput and ILP techniques increasingly rely on speculative mechanisms (branch prediction, instruction and data prefetching, value prediction...), and the quality of each individual prediction mechanism tends to improve slowly.

Since each mechanism comes at a significant on-chip space cost, it is not obvious that speculation will always remain the most complexity-effective path to performance improvements. Already, two approaches discussed in Chapter 2, the Chimaera architecture [Ye00a] and the Grid Processor Architecture (GPA) [Nag01] used in the TRIPS architecture [San03] propose to use on-chip space differently to improve performance. Both approaches rely on a common principle: directly map part of the program dataflow graph to the architecture, so that instructions be-

come hardware operators and execute much faster; Chimaera maps instructions to reconfigurable circuits, and GPA to grids of ALUs. However, once translated into hardware, a sequence of dependent instructions remains a sequence of dependent (connected) hardware operators. Therefore, both approaches are again limited by intrinsic ILP [Wal91], and even more by the compiler ability to extract ILP [Ye00b], just like current and upcoming processors. And the increasing processor architecture complexity combined with the limitations of static analysis on pointer-based codes, like the SPECInt2000 and the Olden [Rog95] benchmarks, already considerably strain the compiler.

In this work, we propose an approach for exploiting additional on-chip space that is not limited by the lack of ILP and that does not require complex software support. The starting point of our approach is to note that many stateless sequences of instructions can be viewed and expressed as a *function*: it has input data (the function parameters) and has output data (the value of the function). Unlike *algorithms*, *functions* can be mapped very easily to a combinational 2-level sum of products circuit (ORs of ANDs). While this transformation is extreme and its cost is prohibitive, it does show that it is possible to obtain a sequence of dependent instructions as a combinational logic circuit. Implicitly, this transformation trades on-chip space for computing resources and achieves high speed by exploiting *circuit-level parallelism*. We present an approach that exploits this principle while avoiding the exponential cost of the 2-level circuit transformation. The mechanism is implemented in a superscalar processor using a large and scalable functional unit, called the *Function* unit. Even though this unit is reconfigurable, its structure is very different and simpler than traditional FPGAs, especially with respect to its interconnection network. The functions are built offline using the trace builder presented in the rePLay framework [Fah01], and we have implemented the corresponding toolset that automatically converts rePLay frames into mappable functions.

With this mechanism, transformed frames execute up to 49% faster for some codes. This mechanism illustrates a first implementation of an approach that provides a different way to improve the performance of sequences of dependent instructions.

In Section 3.2, we present the principles of the approach, the methodology in Section 3.3, we analyze the potential speedup and limitations of the approach in Section 3.4, we present the implementation and experimental results in Section 3.5.

3.2 Principles

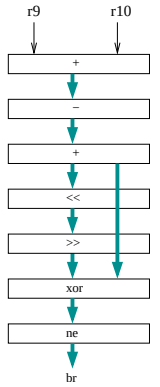
To illustrate our approach and compare it with existing solutions, we will use the example of Figure 3.1(a) extracted from procedure `Sum` of the SPECInt2000 bench-

```
result=(long)hdL+(long)hdR−1;
ov=(int)result;
if((ov<<1)>>1==ov)
   return ov;
```

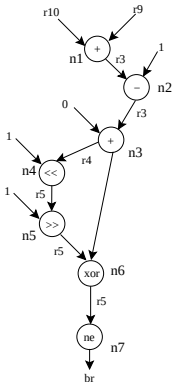
(a)

```
i1: addq r10,r9,r3     ; hdL+hdR
i2: subq r3,0x1,r3     ; hdL + hdR −1
i3: addl r31,r3,r4     ; ov=(int) result;
i4: sll r4,0x1,r5      ; ov <<1
i5: sra r5,0x1,r5      ; ((ov<<1)>>1)
i6: xor r5,r4,r5       ; ((ov<<1)>>1)==ov
i7: bne r5, continue
```

(b)

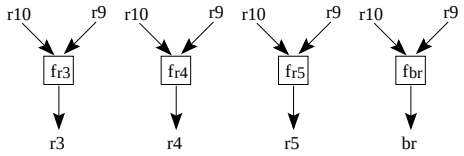


(c)



(d)

$$f_{r3}(r9, r10) = r9 + r10 - 1$$
$$f_{r4}(r9, r10) = sign\_ext(r9 + r10 - 1)_{31:0}$$
$$f_{r5}(r9, r10) = ((r9 + r10 - 1) << 1) >> 1$$
$$f_{br}(r9, r10) = (r9 + r10 - 1) \oplus ((r9 + r10 - 1) << 1) >> 1)$$



(e)

Figure 3.1: *An example of instruction collapsing: (a) C code, (b) assembly code, (c) non-collapsed hardware operators, (d) corresponding DFG, and (e) corresponding functions.*

mark `254.gap`. This code adds two integers and compares the two most significant bits to check for overflow. The resulting assembly code on an Alpha EV6 processor is a fully sequential set of instructions, i.e., no two instructions can execute in parallel, as shown in Figure 3.1(b). Therefore, current and upcoming processors, which heavily rely on the exploitation of all forms of ILP, can do little to improve the performance of such codes. The Chimaera [Ye00a] or GPA [Nag01] approaches would map the corresponding dataflow graph of Figure 3.1(b) respectively to a reconfigurable circuit or a grid of ALUs. The mapped hardware operators would perform faster than a set of instructions, but they would still operate *sequentially*, as shown in Figure 3.1(c).

In our approach, we split the dataflow graph (DFG) of Figure 3.1(d) into a set of independent single-output functions, one for each output of the dataflow graph, as shown in Figure 3.1(e). At the cost of redundant operations, e.g., *r9+r10-1*, and thus hardware resources, all these functions can execute in parallel. Now, each function can be translated into a combinational logic function and *collapsed* into a 2-level logic circuit or a LUT. However, a simple function with two parameters like `fr3`, corresponds to a 2^{128}-bit truth table for each output bit (assuming two 64-bit registers), which is not realistic. One way to alleviate this size problem is to implement a function of n input bits as a set of n 1-bit operators associated with a multiple-carry propagation network, as shown in Figure 3.2(b). Each operator can be implemented as a reconfigurable logic block much like in FPGAs, but the number of 1-bit inputs is higher than in traditional FPGAs, e.g., 4-input lookup-tables (LUTs) in the Virtex-II Xilinx architecture [xil02], versus 6 inputs in our implementation. On the other hand, the placement and routing are much more simple. Further increasing the number of inputs would slightly increase performance (we have evaluated the potential of up to 40-input blocks), but it would also significantly increase operators size, see Section 3.4. The 1-bit logical expressions associated with function `fr3` are shown in Figure 3.2(a). Assuming a collapsed sequence of instructions in a *Function* unit executes as fast as a single instruction, the length of the sequence of dependent instructions is an upper-bound of the speedup, i.e., 7 in the example of Figure 3.1. The impact of the *Function* unit delay on performance is studied in Section 3.5.4.

The notion of collapsing instructions was previously introduced by Phillips et al. who proposed a 3-1 interlock collapsing adder that could collapse two dependent adds into one specific 3-input adder [Phi94]. They later investigated the potential of collapsing up to three dependent instructions [Saz96], but neither the concept nor its implementation were generalized, and the notion of functions was not introduced in these studies. Similarly, an instruction `Scale and Add`, which adds an operand to another multiplied by a factor, is implemented in the Alpha ISA [alp98]. Furthermore, to a certain extent, Chimaera [Ye00a] proposes a limited form of in-

$$(r9+r10)_i = \begin{cases} r9_0 \oplus r10_0 & i=0 \\ r9_i \oplus r10_i \oplus Cout1_{i-1} & 0<i\leq 63 \end{cases} \quad (1)$$

$$Cout1_i = \begin{cases} r9_0 \cdot r10_0 & i=0 \\ r9_i \cdot r10_i + r9_i \cdot Cout1_{i-1} & \\ \quad + r10_i \cdot Cout1_{i-1} & 0<i<63 \end{cases} \quad (2)$$

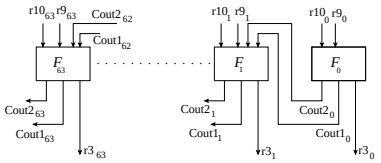
$$(r9+r10-1)_i = \begin{cases} (r9+r10)_0 \oplus 1 & i=0 \\ (r9+r10)_i \oplus 1 \oplus Cout2_{i-1} & 0<i\leq 63 \end{cases}$$

$$= \begin{cases} \overline{r9_0 \oplus r10_0} & i=0 \\ \overline{r9_i \oplus r10_i \oplus Cout1_{i-1}} \oplus Cout2_{i-1} & 0<i\leq 63 \end{cases} \quad (3)$$

$$Cout2_i = \begin{cases} (r9+r10)_0 & i=0 \\ (r9+r10)_i + Cout2_{i-1} & 0<i\leq 63 \end{cases}$$

$$= \begin{cases} r9_0 \oplus r10_0 & i=0 \\ (r9_i \oplus r10_i \oplus Cout1_{i-1}) + Cout2_{i-1} & 0<i\leq 63 \end{cases} \quad (4)$$

(a)



(b)

Figure 3.2: *Translating function* `r3` *into a hardware operator: (a)* `r3` *function, and (b) 64 1-bit operators with multiple-carry propagation.*

struction collapsing by combining arithmetic operations, e.g., ADD, with bit-shifting instructions: in fact, a single arithmetic operation takes place on each row of the reconfigurable unit but the interconnection network between rows is used to implement the shifts, hence the collapsing. The notion of "function" is implicitly widely used in ASIC and ASIP [Fis02], but not in a way that can be applied to general-purpose processors. PRISC [Raz94] proposes to map operations on hardware-programmable functional units (PFUs), and recently, Clark et al. [Cla03] proposed a method to automatically extract candidate functions from programs, but, in both approaches, the operations are not extracted at run time, and cannot span across multiple basic blocks.

Our approach has three major assets: (1) in theory it applies to almost any sequence of dependent instructions, (2) it doesn't rely on ILP exploitation, and (3) translating DFGs into functions requires only straightforward transformations.

3.3 Experimental Framework

We performed two sets of experiments: architecture-independent experiments which aim at determining the potential of the approach, see Section 3.4, and experiments on a superscalar processor coupled with the rePLay framework without optimization (only the frame builder is used), see Section 3.5. We developed a specific toolset to translate Alpha assembly instructions into circuit configuration macros. It can be implemented either as a static compilation tool, a dynamic compilation tool, or in hardware. On purpose, we developed a fully automatic toolset in order to demonstrate that the added compiler and hardware complexity can be harnessed.

Translating dependent instructions into configuration macros. The four phases of our optimization engine are shown in Figure 3.3: first, we dynamically split the program execution trace into large chunks of consecutive instructions that we call a *trace*, and we apply the next steps to each trace. Note that the trace size is fixed for the architecture-independent experiments, and is variable in the superscalar processor experiments, see Section 3.5. Next, data dependencies in the trace are analyzed and the dataflow graph (DFG) for that trace is built as in Figure 3.1(d). Then functions are selected within the DFG modulo the rules described in Section 3.4.2. Finally, the truth table associated with each bit of the function is computed as well as the associated carry chain functions described in Section 3.2 and Figure 3.2. The LUT (Look-Up Table) configurations directly derive from the truth tables. The algorithm and its implementation are detailed in Section 3.5.

Simulation methodology. In all experiments we used the SimpleScalar emulator [Bur96] of the Alpha ISA, the SPECInt2000 benchmarks, as well as 9 of the Olden benchmark suite [Rog95] (bh, em3d, health, mst, perimeter, power, treeadd,

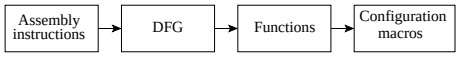


Figure 3.3: *Phases of the optimization engine.*

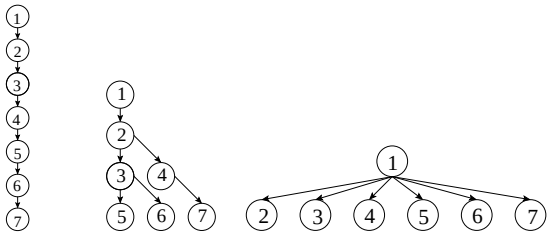


Figure 3.4: *Different possible DFG shapes.*

tsp and bisort) and 3 of the MiBench suite [Gut01] (patricia, tiff2bw and djpeg). We tested 100 million consecutive instruction traces for each benchmark, focused on the most time-consuming procedures (selected using profiling on full benchmarks executions). The benchmarks were compiled using the Compaq Alpha compiler with full optimizations `(-fast)`. For the superscalar processor experiment, we used the `sim-outorder` architecture [Soh90] and applied our transformations to rePLay frames.

3.4 Potential of the Approach

3.4.1 Potential performance improvements

To evaluate the potential of the approach, we want to compute the theoretical speedup over an idealized processor where all instructions that *can* execute in parallel *do* execute in parallel. Thus performance improvements only come from executing sequential instructions as collapsed functions. The idealized processor is defined as having a 1-cycle ideal memory, perfect branch prediction, and infinite instruction window, issue width, and reservation stations.

As explained in Section 3.2, the potential speedup is, in theory, determined by the number of dependent instructions collapsed, i.e., 7 in the example of Figure 3.1. However, consider the DFGs of Figure 3.4. DFG1 in Figure 3.4(a) represents a

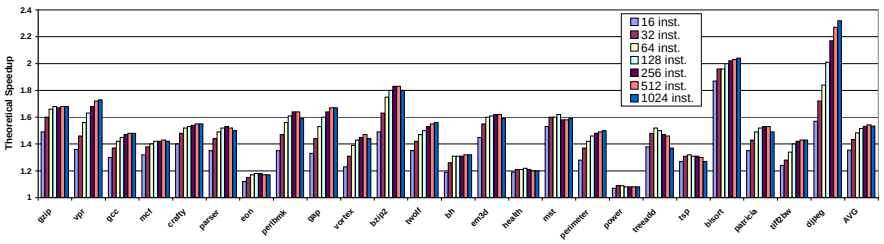


Figure 3.5: *Theoretical speedup for different trace sizes.*

sequence of 7 dependent instructions, like our example of Figure 3.1, resulting in a theoretical speedup of 7. DFG2 in Figure 3.4(b) contains again 7 instructions but it has 3 branches, one per output function, and the largest branch of the DFG contains 4 instructions, thus, the maximum speedup is 4 in this case. Similarly, for DFG3 in Figure 3.4(c), the theoretical speedup is 2, again with 7 instructions. Therefore, to compute the theoretical speedup of a trace of instructions, we need to identify all *disjoint* DFGs in the trace, i.e., DFG_a is disjoint from DFG_b if none of the instructions of DFG_a depends on an instruction in DFG_b and reciprocally. Then, the theoretical speedup of a DFG is equal to the size of its largest branch, or, in other terms, to its critical path or height. Thus, the traces are partitioned into disjoint DFGs, and the theoretical speedup of each DFG is calculated. The theoretical speedup of a program trace is the average height of all DFGs in the trace. An important limiting factor for the speedup is the trace size. The larger the traces, the larger the DFGs and the speedup. Using 1024-instruction traces and up to 40-input operators, the average theoretical speedup for all benchmarks is 1.5 with a maximum speedup of 2.32 for the `djpeg` benchmark, see Figure 3.5. Figure 3.6 shows the distribution of the height of DFGs as a percentage of the total number of instructions executed, also using traces of 1024 instructions, averaged on all benchmarks. While there are very large DFGs, i.e., over 250 instructions, many DFGs are rather small. The factors limiting the size of the DFGs and how these limitations can be overcome are discussed in Section 3.4.2.

3.4.2 Analyzing and overcoming the limitations of the approach

In theory, the whole program can be one huge DFG. In practice, DFGs need to be split due to many factors, which we call DFG *cuts*. A *cut* is an instruction that prevents further collapsing and thus reduces speedup opportunities. It is then transformed into a function output of its parents' DFG and becomes an input for its childs' DFGs. Besides these "true" cuts, the trace size limitation mentioned in

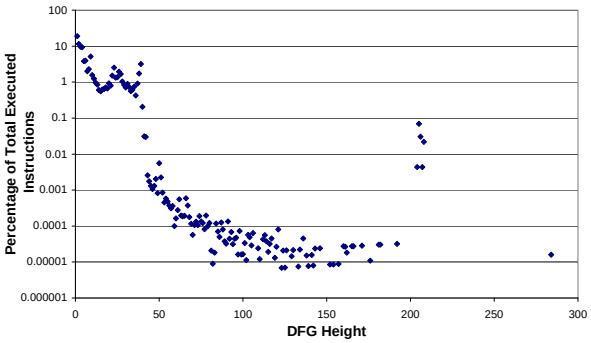


Figure 3.6: *DFG height distribution.*

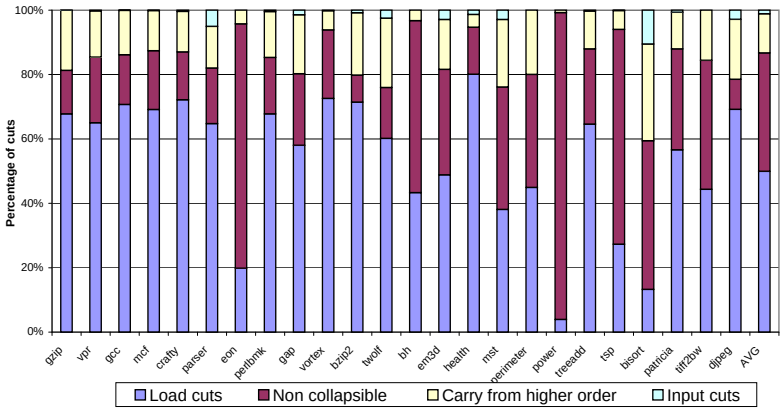


Figure 3.7: *Distributions of cuts.*

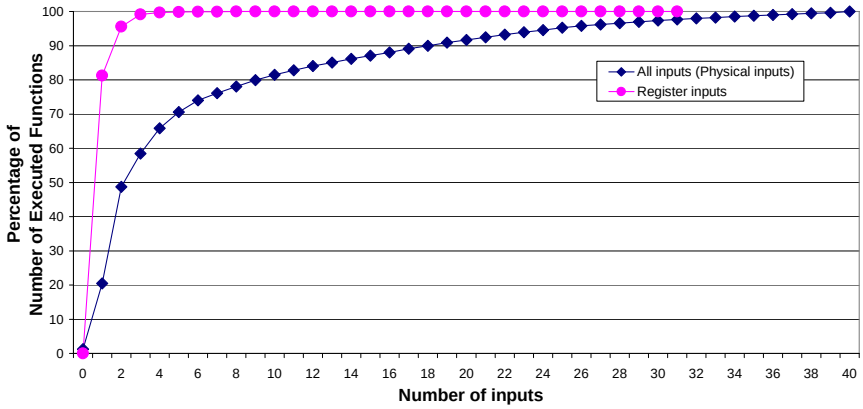


Figure 3.8: *Cumulative distribution of the number of inputs per function.*

Section 3.4.1 introduces additional methodology-related cuts. The different types of "true" cuts are the cuts caused by the number of inputs supported by the hardware operator, the cuts caused by loads, non-collapsible instructions and the carries from upper significant bits. Figure 3.7 shows the contribution of each cut as a percentage of all cuts.

Number of function inputs. We call *physical inputs* both the register inputs and the inputs corresponding to carries, see Figure 3.2(b). The maximum number of physical inputs per function determines the size of the 1-bit hardware operators used to implement functions. Since the hardware operator size is fixed, the maximum number of inputs is fixed as well, and any DFG requiring more than the maximum number of inputs must be cut. Figure 3.8 shows the cumulative distribution of the number of inputs per function, averaged over all benchmarks, using 1024-instruction traces. We observed that more than 80% of the functions require fewer than 10 physical inputs, so that implementing even large functions does not require large 1-bit operators. In our implementation we have used 6-input logic blocks. Figure 3.9 confirms that increasing the number of inputs beyond 10 has a negligible impact on the theoretical speedup.

Load instructions. For the moment, *loads* induce cuts because they cannot be combined with subsequent dependent instructions, though we are currently investigating several ways to alleviate these cuts such as data preloading. Figure 3.10 shows the percentage of loads in each benchmark. While, on average, 24.43% of executed instructions are load instructions, their irregular occurrence in DFGs still enables large DFGs, as shown in Figure 3.6. Store and branch instructions are

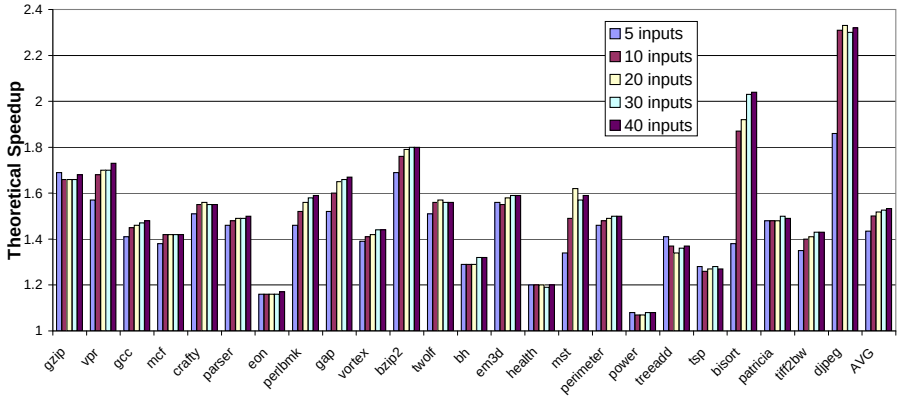


Figure 3.9: *Impact of the number of inputs on the theoretical speedup.*

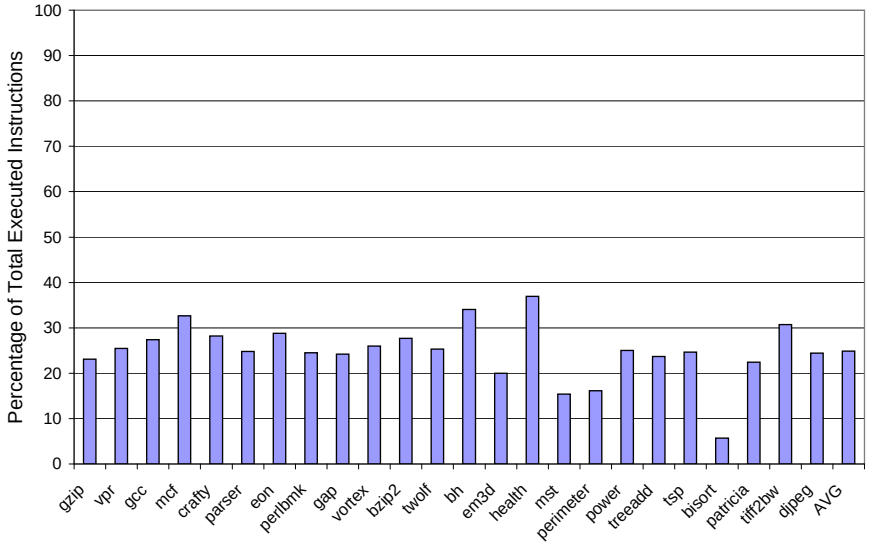


Figure 3.10: *Percentage of loads.*

not *cuts*, they are exit points of DFGs. Load and store instructions are still considered collapsible : address computation instructions can be collapsed with loads and stores, and value computation instructions can be collapsed with stores. Our transformation engine detects pairs of statically dependent store/load and replaces such "load cuts" with true register dependence. By static dependence, we mean a store followed by a load which uses the same register and the same offset for address computation and where the register is not modified between the store and the load. Note that such a store/load pair may not necessarily be in the same basic block, as the generated functions may span across more than one basic block. Dynamic store/load dependencies are not detected.

We note that the main problem in loads is rather their *chaining* then their frequency. Load chains occur frequently in complex data structures because of indirections and pointer chasing. If loads where independent, they could occur in parallel, hiding the effect of the induced cuts. But when a chain of loads occurs, each set of instructions between two loads (typically an address calculation) must wait for the first load to complete before they begin executing, even if they are collapsed. Figure 3.11 shows the average depth of loads for each benchmark. We define the depth of a load as the number of dependent loads in the load chain to which it belongs. The `mcf` and `health` particularly contains long chains of dependent loads. Figure 3.12 shows the contribution of each each value of load depth in the total loads.

Non-collapsible instructions. Like all other RISC codes, the Alpha binary code contains many instructions that cannot be collapsed (e.g., system calls), or which correspond to very costly hardware operators (e.g., floating-point divide). All these instructions are DFG cuts. For the moment, we only collapse integer instructions add/sub, shift by constants, bit operations/manipulations, and conditional branches. Figure 3.13 shows that, on average, benchmark traces contain 15.13% of non-collapsible instructions. The `eon` and `power` benchmarks perform many floating-point operations, hence the large number of non-collapsible instructions.

Carries from upper significant bits. Certain combinations of instructions are treated as cuts due to specific carry propagation issues. Consider the example of Figure 3.14 in which the most significant bit of $(r1 + r2)$ is added to *r6*. Due to the carry propagation in $(r1+r2)$, the addition of *r6* cannot start before $(r1+r2)$ ends. Therefore, the only way to collapse $(X >> 63)+r6$ with $(r1+r2)$ is to add another chain of 64 1-bit operators where the output of the most significant 1-bit operator of $(r1+r2)$ is fed into the least significant 1-bit operator of $(X >> 63)+r6$, see Figure 3.14. In other terms, either the operator chain is larger than the word size and it can accommodate right shifts, or right shifts must be treated as cuts; the same problem occurs whenever a carry comes from upper significant bits. To test the impact of

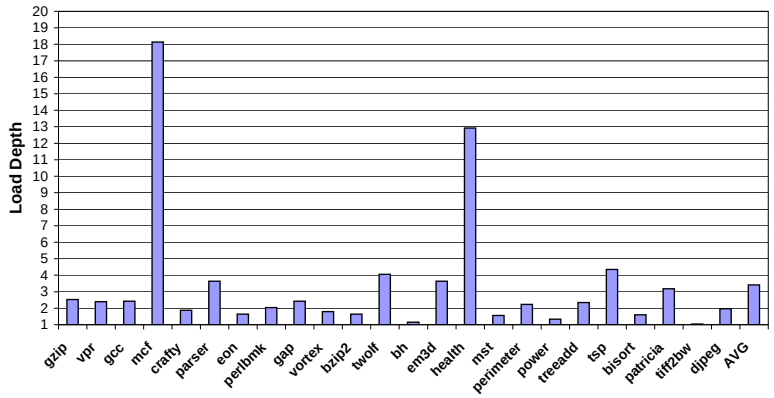


Figure 3.11: *Average depth of loads.*

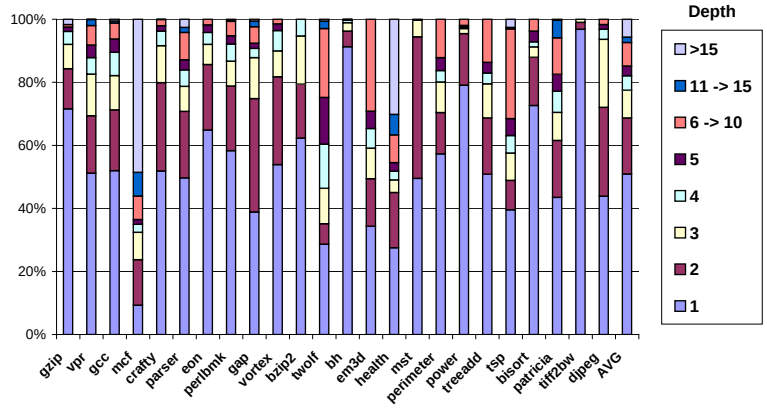


Figure 3.12: *Distribution of load depth among all load cuts.*

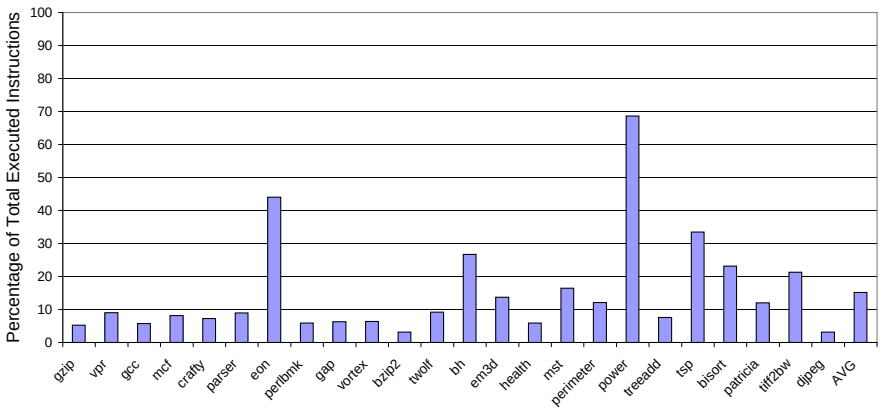


Figure 3.13: *Percentage of non-collapsible instructions.*

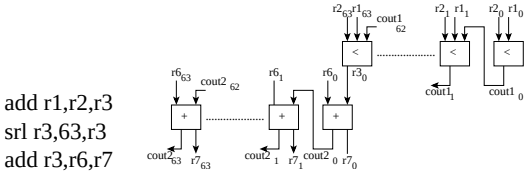


Figure 3.14: *Cuts because of carries from upper significant bits: (a) assembly code, (b) implementation.*

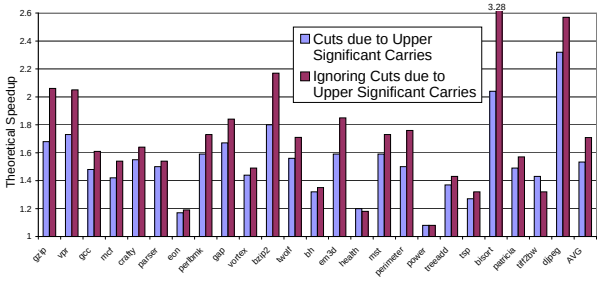


Figure 3.15: *Effect of relaxing the upper significant carries constraints.*

that choice, we have ignored the hardware consequences (cost) of carries from upper significant bits and removed the corresponding cuts. As shown in Figure 3.15, further performance improvements can be achieved by relaxing this constraints. In all other experiments, we chose to treat such cases as cuts for hardware cost reasons, i.e., we assume a "left-only" carry propagation.

3.5 Implementation

In this section, we first explain how functions are built, and then how this mechanism can be implemented within a superscalar processor architecture using the rePLay hardware framework [Pat01].

3.5.1 Generating DFG and functions

The optimization engine described in this section may be implemented either in hardware or software, the rePLay framework is compatible with both types of implementation. We present below the different steps in the process of building functions, and we illustrate this process with the example of Figure 3.1.

Building the DFG. Each instruction in the trace, i.e., the instructions of Figure 3.1(b), are loaded in the DFGT (*DataFlow Graph Table*), with one entry per instruction, see Figure 3.16.[1] Consider instruction `i1:addq r10,r9,r3`: when the instruction is stored in the DFGT, the flag `OutFlag` is set to indicate the instruction is an output of the DFG being progressively built; when instruction `i2:subq r3,0x1,r3` is loaded, this flag will be reset because instruction `i1:addq r10,r9,r3` will no longer be a DFG output.

In the same time, we keep track of the data dependencies between instructions through the *Producing Output Table* (POT). The POT is indexed by the instruction destination register. Each entry contains an index to the DFGT, i.e., to the instruction that produces the corresponding register. For instruction `i1`, since `r9` and `r10` are inputs to the DFG, i.e., they are not generated by another DFG instruction, the corresponding entries in the POT will not point to a DFGT entry. The combined role of the POT and the DFGT is akin to the role of the ROB in a superscalar architecture, except that we cannot use the processor ROB to perform that task since function building is performed *offline* by the rePLay framework.

Generating the function corresponding to an instruction. Once a new DFG instruction is loaded in the DFGT, we build the function corresponding to the

[1] Note that complex instructions, such as the Alpha Scaled Add instruction `s4addq`, are decomposed into several elementary operations, each corresponding to a DFG node, and thus to an entry in the DFGT.

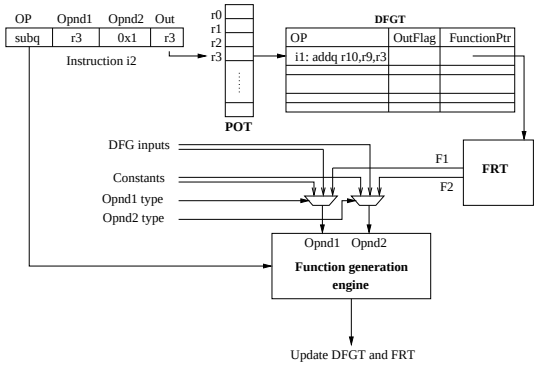


Figure 3.16: *Function generation.*

instruction operation, or more exactly, we compose this operation with the functions producing its source operands, creating a more complex function. For that purpose, we send the instruction source operands to the *Function Generation Engine* (FGE), see Figure 3.17. There are three types of operands, DFG inputs which are operands not produced by other instructions in the DFG, constant inputs, and operands produced by other instructions in the DFG. The first two types are simply sent to the Function Generation Engine. If the operand is produced by another instruction, then we send the function producing this operand as a truth table. Functions are stored in the FRT (*Function Repository Table*) as 64-bit truth tables (for a maximum of 6 inputs for each function), one per word bit, see Figure 3.18. Each DFGT entry (instruction) contains an index to the corresponding function in the FRT. Besides the truth table, the FRT also contains the number of inputs of the truth table. Each bit of the operand is a (*input list, truth table*) pair. When the operand is just a value, as for registers `r9` and `r10` of instruction `i1`, the number of inputs is 1 and the truth table is the identity function (`01`).

To create the truth table of the composed function (the operation of the current instruction composed with the functions creating the operands), the Function Generation Engine performs as follows. For each possible combination of all input variables (the input variables of both operands), the engine looks up the truth tables of the operands, and uses these values to then look up the truth table of the operation itself. The truth table of the operation is stored in a library of operations in the Function Generation Engine. Besides the truth table, the library also indicates if additional function variables must be introduced, such as (and usually) carries. For instance, for instruction `i1`, the operands are `r9` and `r10` with identity truth tables. After looking up the library of operations, the engine sees that the carry

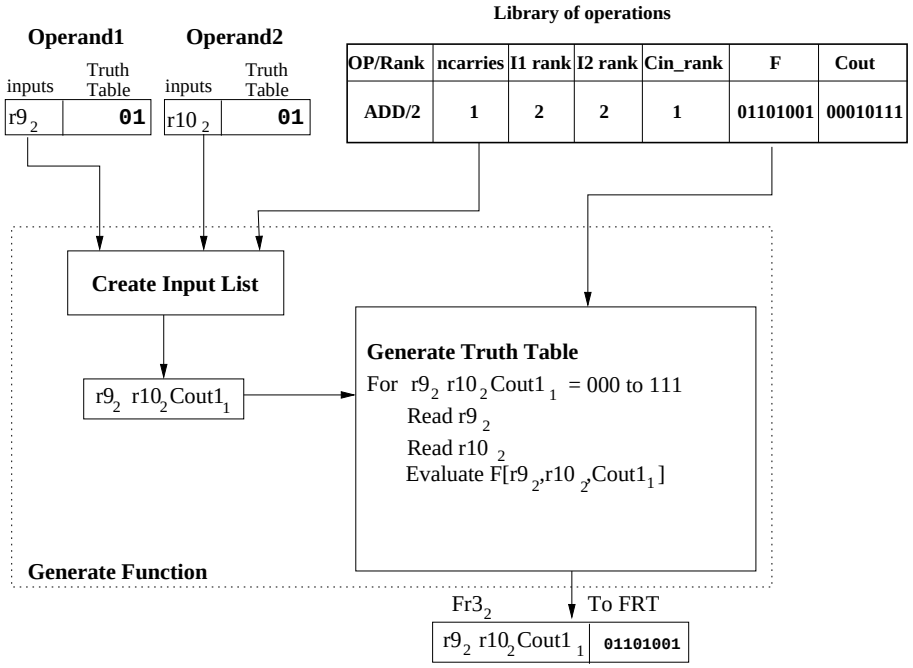


Figure 3.17: *Function Generation Engine: generating bit 2 of node n2.*

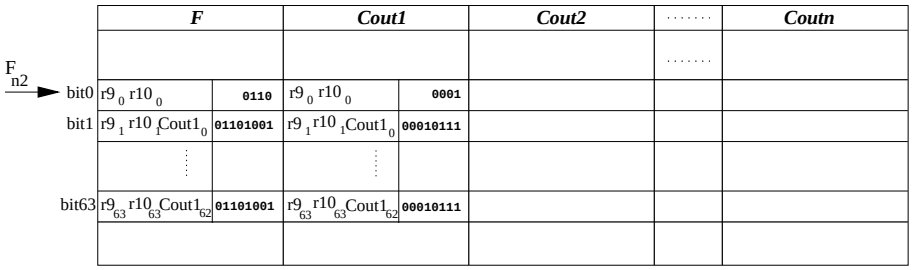


Figure 3.18: *Function Repository Table (FRT).*

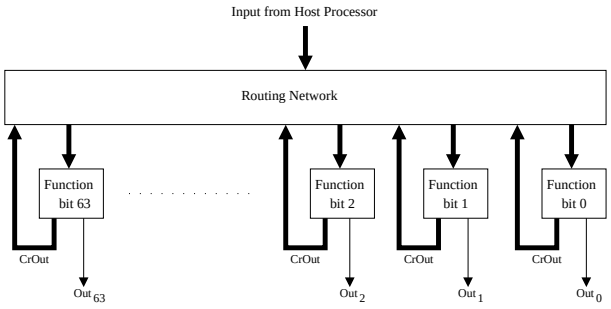


Figure 3.19: *Implementation of functions.*

must be one of the function inputs since the operation is an addition, so there are three input variables: $r9_k r10_k Cout1_{k-1}$ for bit k of the composed function. Then, the truth table of the composed function (in this case, just an addition) is output (in this case, the adder truth table). Function truth tables are then stored in the FRT, and the corresponding index is stored in the DFGT. The whole process for instruction `i2:subq r3,0x1,r3` is similar except that the POT entry corresponding to `r3` will point to the `i1` entry in the DFGT. The function corresponding to this entry will thus be passed to the Function Generation Engine as well as the constant `0x1`.

Cuts. Input operands that are dependent on instructions that were identified as *cuts* (see Section 3.4) are treated as identity functions also.

Filtering functions. Depending on the latency of the *Function* unit, a sequence of collapsed instructions may execute slower in the *Function* unit than in the normal execution units, if the number of collapsed instructions is small. For example, if a function is collapsing three dependent 1-cycle latency instructions, the *Function* unit should execute in less than three cycle to execute faster. Assuming a 1-cycle ALU latency, we *filter* candidate functions by selecting only those disjoint DFGs (see Section 3.4) with height greater than the *Function* unit latency. Instructions that do not belong to these selected DFGs are marked as non-collapsible, and are executed in the normal execution units. This heuristic allows better utilization of all functional units and prevents non-appropriate functions from slowing down the execution.

3.5.2 Hardware implementation of functions

Figure 3.19 shows the implementation of functions as an additional large functional unit. As explained in Section 3.2, we implement functions using a set of 64 1-bit chained operators. These operators represent one of the bits of an n-input function, as explained in Section 3.2. Since the functions vary constantly from one trace to

Fetch width	16
Issue / Decode / Commit width	8
RUU size (Inst. window- ROB)	1024
LSQ size	128
ExeUnits	8 IALU, 4 IMULT, 4 FPALU, 4 FPMULT
Function units	8
Branch	Combined, 4K entries bimodal, and 2 level Gap predictor, 8K 2nd level entries, 14 history wide , 1K meta-table size 7 cycle BR resolution
Memory Latency	70 cycles
L1 DCache	32kB, 1 cycle
L1 ICache	16kB, 1 cycle
L2 Unified Cache	1MB, 6 cycle

Table 3.1: *Baseline configuration of the processor core.*

another, we use reconfigurable logic to implement the 1-bit operators. However, it is important to note that our operators need not bear the same limitations as traditional FPGAs: (1) the chained operators only contain combinational logic, no sequential logic is necessary, and (2) only one row of operators is needed. The operators are linked in an unusual but simple manner: *multiple* carries are propagated from operator i to higher order bits only, therefore avoiding the complex interconnection networks that usually account for more than 90% of on-chip space in FPGA circuits [Com02]. On the other hand, our approach relies on functions with a significant number of inputs, resulting in larger logic blocks. As mentioned in Section 3.4.2, we assumed a maximum of 6 physical inputs for each bit of the *Function* units.

3.5.3 Implementing functions using the rePLay hardware framework

The two major implementation issues of our approach are the overhead of dynamically building DFGs and functions on-the-fly, during execution, and assembling large

traces. The rePLay environment proposed by Patel et al. [Fah01][Pat01][Pat00] can partially address both issues.

The rePLay framework provides a dynamic optimization support for building large traces of instructions (frames) after retirement. Moreover, the frames are transformed offline, i.e., out of the critical path. We implemented the rePLay architecture framework, augmented with our function optimization engine and the associated *Function* units, in the SimpleScalar simulation environment. We assumed a future scaled-up 8-way superscalar processor architecture, see Table 3.1 for the modified parameters. Figure 3.20 shows the core processor together with rePLay and the function mechanism which includes several *Function* units. Our optimiza-

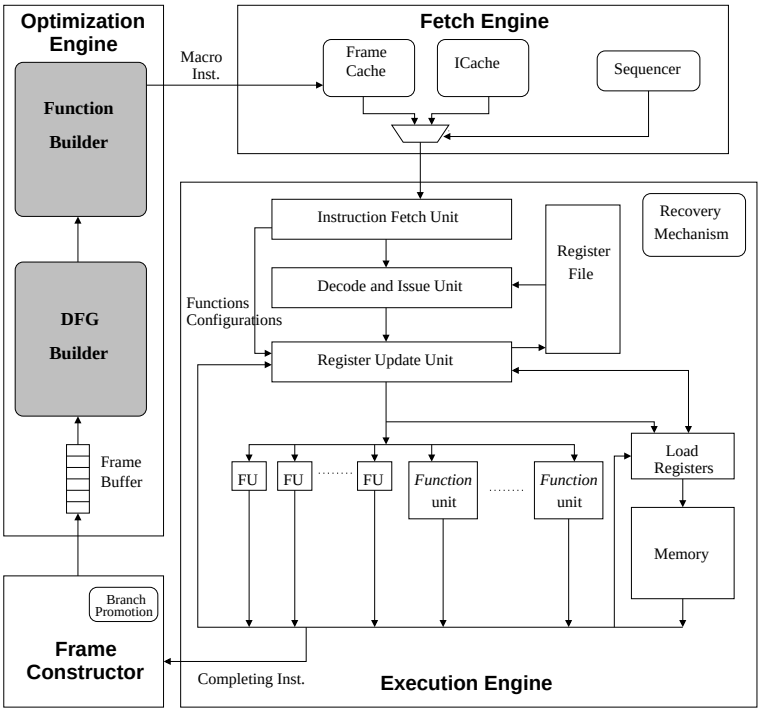


Figure 3.20: *The core architecture.*

tion mechanism can be built on top of the optimizations proposed in [Fah01] which improve ILP while our techniques focus on ILP-deprived code sections. To outline the impact of the *Functions* mechanism, we implemented rePLay without frame optimizations; thus the performance improvements reported in Section 3.5.4 solely correspond to the *Functions* mechanism.

The rePLay framework collects traces of committed instructions to form "frames".

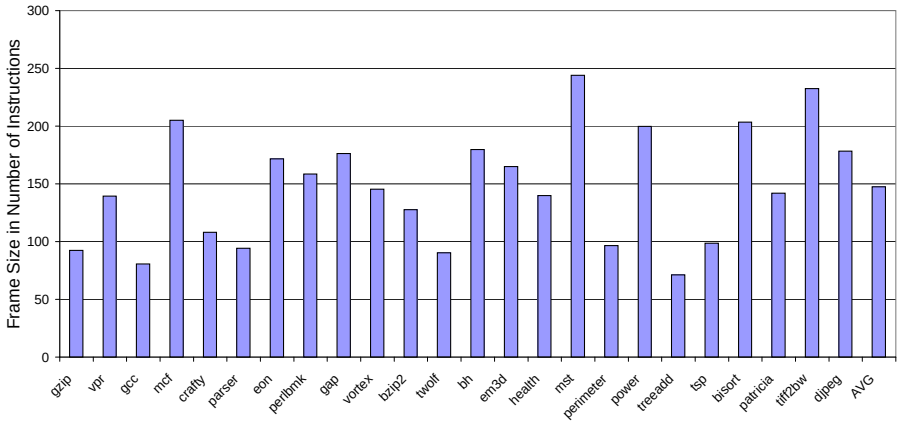


Figure 3.21: *Average frame size.*

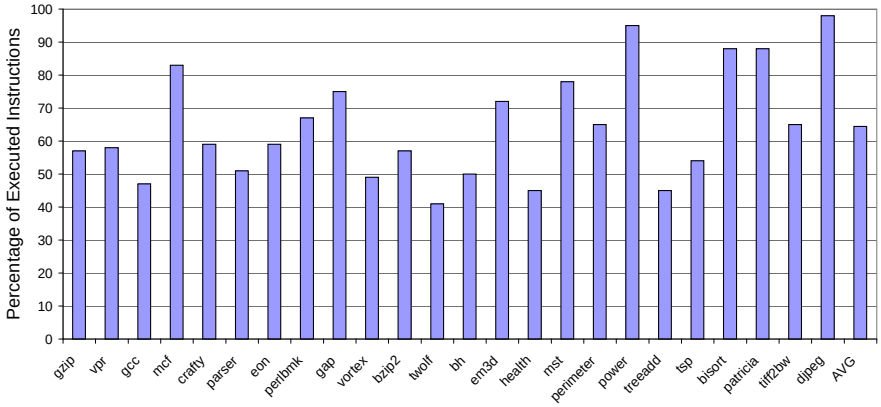


Figure 3.22: *Dynamic instructions coverage.*

The frames are frequently executed sequences of instructions. The *frame constructor* adds each committed instruction into a frame construction buffer, until a branch with a non-highly stable behavior (taken/not taken for conditional branches or constant target addresses for indirect branches) is encountered. Branches with highly stable behavior are called promoted branches in the rePLay framework [Pat98], and are stored in a *branch bias table*. When a non-promoted branch is encountered or when the frame reaches a maximum size of 256 instructions, the frame is passed to the optimization engine to build functions (frames smaller than 32 instructions are

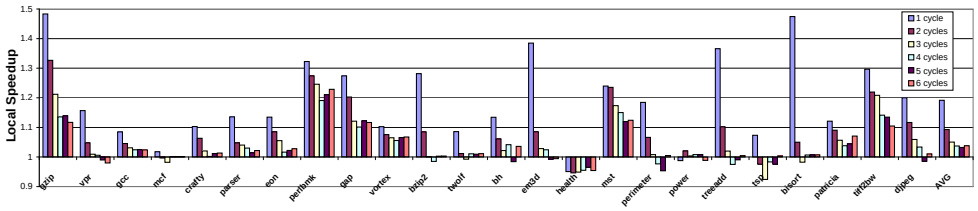


Figure 3.23: *Local speedup.*

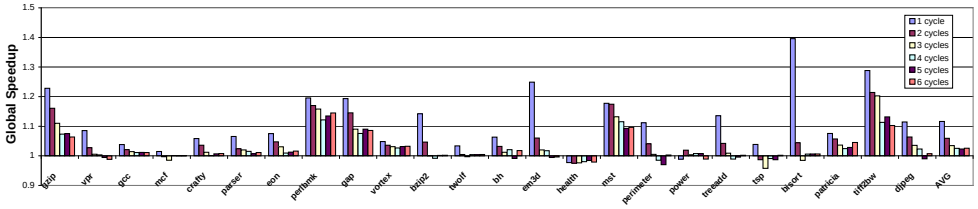


Figure 3.24: *Global speedup.*

discarded to avoid saturating the optimization engine). Our back-end transformation engine forms the DFG for each frame and transforms the trace of instructions into a trace of functions or *macro-instructions*, see Section 3.5.1.

Because it is difficult to estimate a priori the exact delay of the optimization engine (whether implemented in hardware or software), we assumed a 1000-cycle optimization engine delay. Fahs et al. showed in [Fah01] that the optimization delay may have very little impact on performance. We tested a 10000-cycle delay and only observed an average performance slowdown of less than 1%. The generated functions are cached into the frame cache and are directly forwarded to the *Function* units upon a frame cache hit. *Function* units may be configured while they are scheduled provided there is a sufficient number of functional units, as originally proposed in the PipeRench architecture [Cho00]. An important aspect of the rePLay framework is that, once dispatched, the frames should be run to completion. Therefore, branch instructions in frames are replaced with *assertions*. When an assertion is not verified, rePLay provides a mechanism to revert the architectural state to the beginning of the executed frame. We modeled this mechanism using a 10-cycle penalty. The replacement of branches by assertions fits well our approach. Since each conditional branch is transformed into a 1-bit function, as shown in the example of Figure 3.1, collapsing the function can speed up the branch resolution which, in turn, can reduce the frame misprediction penalty. More generally, collapsing functions corresponding

to branch conditions can speed up branch resolution, using similar principles but a different technique than *Anticipation* [Far98].

We parameterized the rePLay environment as follows: a 32K-entry bias table for direct branches, a 4K-entry bias table for indirect branches, a 14-branch history is used as a hashing key to the bias table, a branch is promoted to a highly biased branch after a threshold of 16 consecutive stable behavior, a 128-frame buffer to store frames while they are processed, a 4-way associative frame cache [Fah01] that can store up to 4K frames and 128K macro-instructions, each frame in the frame cache is tagged using a history path of 4 branches. Upon hit, the frame is dispatched, otherwise standard instructions are fetched from the instruction cache and dispatched.

3.5.4 Performance analysis of the implementation

Efficiency of frame building. The main asset of rePLay is its ability to dynamically build large sequences of instructions. We experimentally observed that the average frame size is 147 instructions, see Figure 3.21. On average, 65% of all instructions instances effectively belong to optimized frames, see Figure 3.22. Instruction coverage is limited by non-highly biased branches, too small frames (less than 32 instructions), frame cache misses and mis-speculated frames.

Speedup achieved with the function mechanism. Consequently, we distinguish the speedup achieved on the transformed traces, i.e., the *local speedup* and the global speedup. We use the scaled-up 8-way superscalar architecture (see Table 3.1) coupled with rePLay as the baseline configuration. We have yet to implement a hardware model of the *function* unit at the circuit level to precisely estimate its latency. Even then, there are multiple possible architecture choices: since these chained operators implement multiple but straightforward carry propagation, they can benefit from the the fast but complex carry-propagation schemes that were specifically designed for speeding up FPGA-based carry chains [Hau00], and which are different from the standard high-performance adder carry chains [Zie01]. Therefore, we varied the latency of the *Function* unit from 1 to 6 cycles. Assuming a 1-cycle *Function* unit, Frames transformed into functions execute 19% faster than the baseline configuration on average, with a maximum of 49% for the `gzip` benchmark, as shown in Figure 3.23. Because of the still limited coverage of the rePLay environment, the global performance improvement is only 12% but with strong variations up to 39.6% for the `bisort` benchmark, see Figure 3.24. Codes with large sets of sequential and dependent instructions particularly benefit from the mechanism. The low speedups are mainly due to high percentage of non-collapsible instructions (`eon` and `power`)

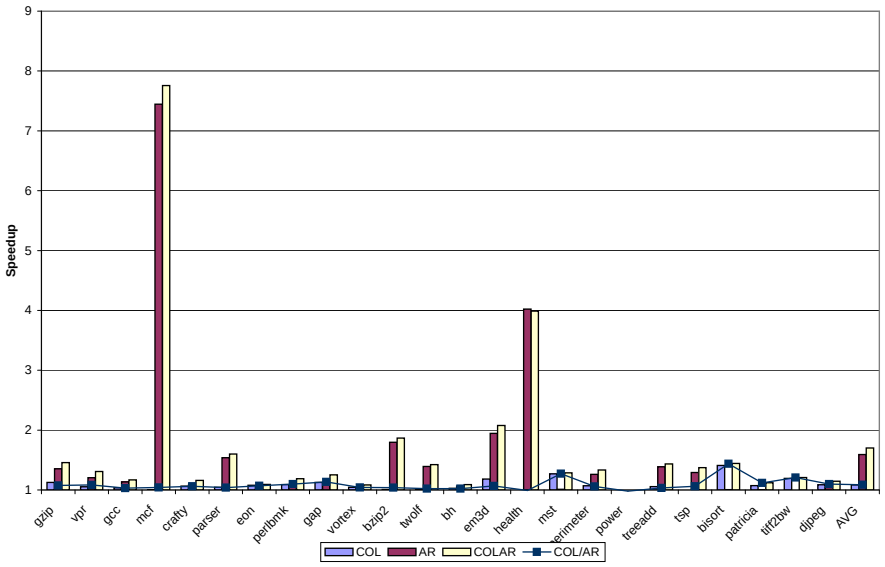


Figure 3.25: *Combining instruction collapsing with perfect address prediction.*

or long chains of dependent loads that limit the height of collapsed instructions (`mcf` and `health`).

Speedup of instruction collapsing under idealistic conditions. In order to study the potential improvement of instruction collapsing in a superscalar processor, we investigated the speedup we obtain under three idealistic conditions:

- **AR**: Address always Ready, in other words we considered a perfect address prediction mechanism to relax the constraint of long chains of loads.
- **PC**: Perfect Cache.
- **PR**: Perfect RePLay, where we assumed that the rePLay framework collects frames that are all useful and not speculative, PR environment is associated with a perfect branch prediction.

We studied each of these environments alone, and combined with our collapsing mechanism (**COL**), and also with the other two ideal environments. We assumed that the hardware operators can support up to 20 physical inputs, and operate in a 1-cycle delay. Also, because the SimpleScalar `sim-outorder` model splits each

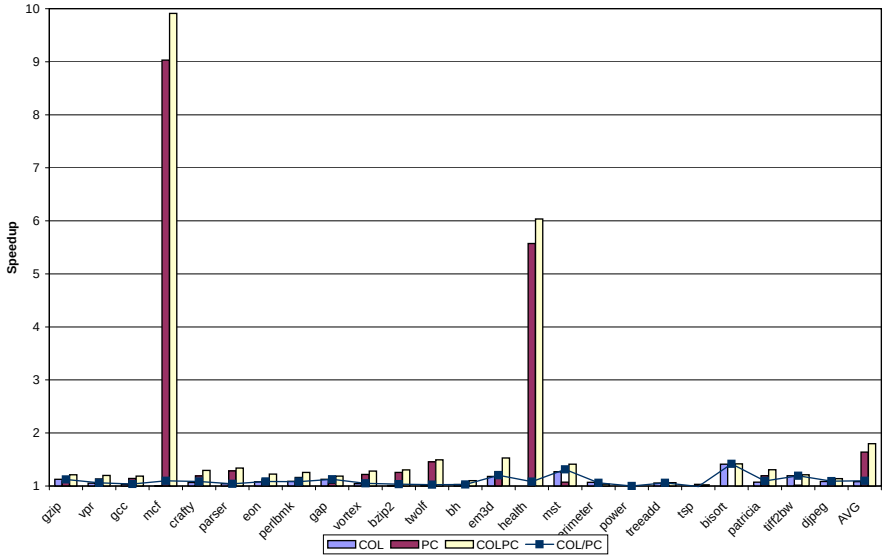


Figure 3.26: *Combining instruction collapsing with perfect cache.*

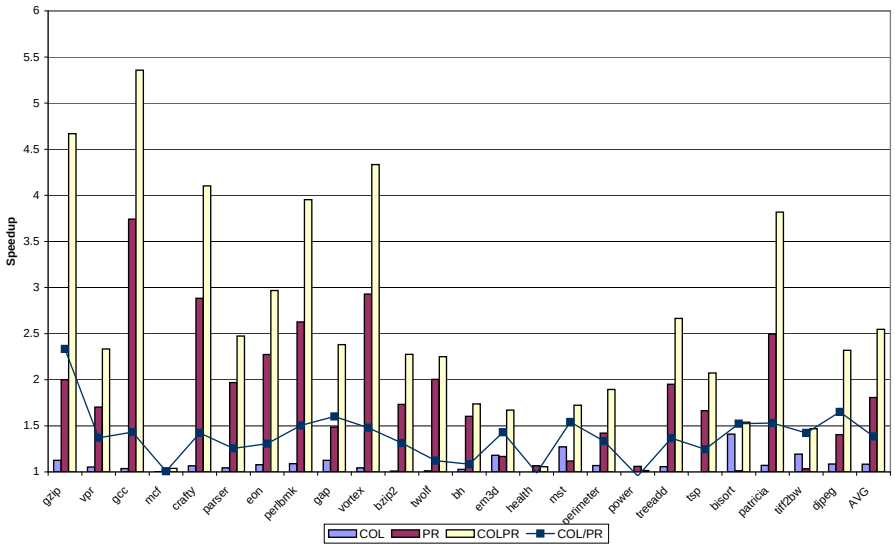


Figure 3.27: *Combining instruction collapsing with perfect RePLay.*

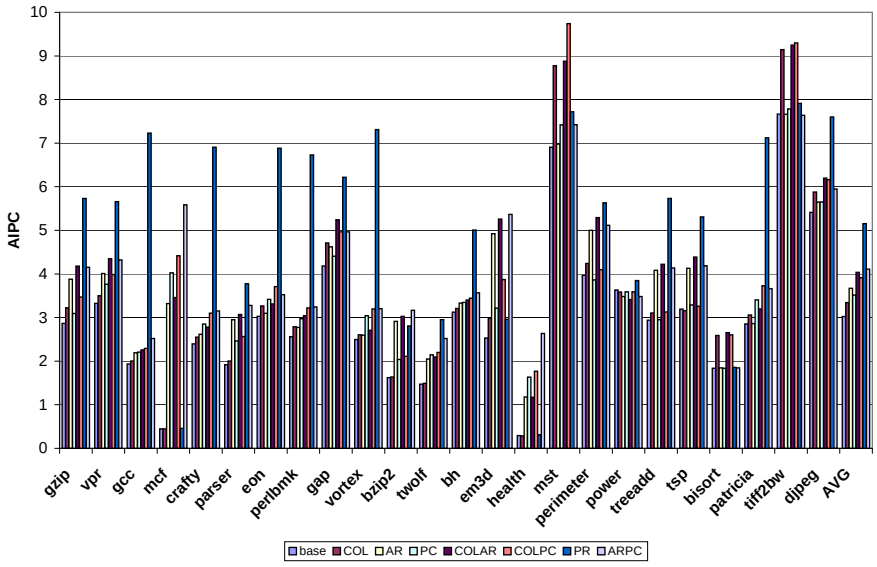


Figure 3.28: *Alpha IPC (1).*

memory operation to two distinct operations (address calculation and memory operation), we assumed an issue width of 16 to allow issuing of 8 memory operations simultaneously. Figure 3.25, 3.26 and 3.27 show the effect of combining the collapsing mechanism (**COL**) with each of the three idealistic environments AR, PC and PR respectively, the speedup of collapsing alone as well as the speedup obtained under the ideal environment *without collapsing* are shown in addition to the speedup we obtain if we combine the two mechanisms, the line shows the speedup of collapsing (COL) applied to the perfect environment. We noticed that applications that particularly suffered from high cache miss rate and long chains of loads, like `mcf` and `health`, could better benefit from instruction collapsing. Still, on the average, the performance improvement of collapsing was just added to the performance we obtain if we apply the ideal environment alone. On the other hand, Figure 3.27 shows that, if we apply a perfect trace collection mechanism, we can substantially benefit from collapsing where on average, we obtain a speedup of 38% over perfect rePLay (PR) compared to 8% initial performance improvement. Furthermore, some benchmarks could have an improvement in performance higher than the theoretical speedup we may obtain with collapsing, for example the `gzip` benchmark has a speedup of 2.32 while the average height of collapsed sequences assuming 256 instruction trace size is 1.67. This is principally because we could benefit from the superscalar approach

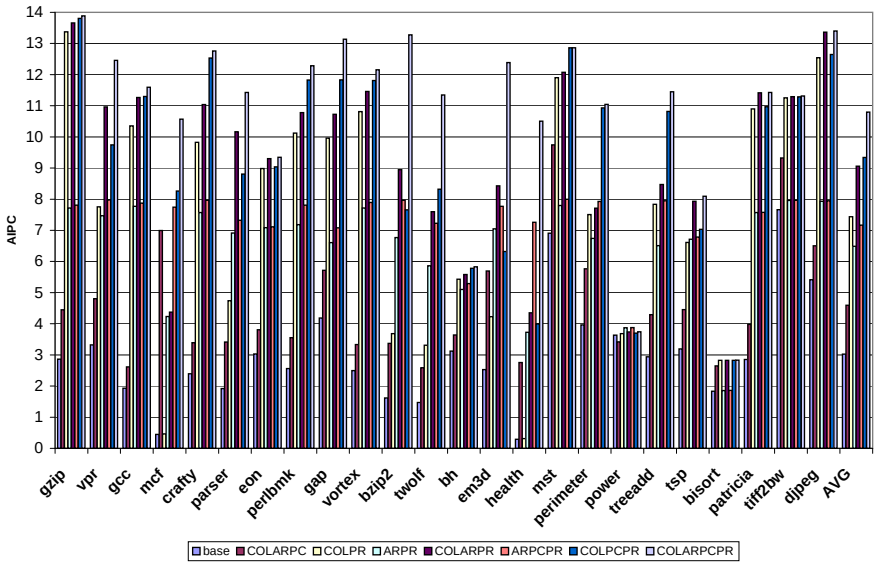


Figure 3.29: *Alpha IPC (2).*

by executing several functions in parallel thus adding Function Level Parallelism (FLP) to the instruction collapsing.

Finally, Figure 3.28 and 3.29 shows the equivalent Alpha Instructions Per Cycle (AIPC) for different combinations of collapsing and idealistic mechanisms. We notice that under some ideal conditions, thanks to instruction collapsing, we obtain IPC higher than 8 which is the maximum IPC the superscalar can achieve.

3.6 Conclusions

In this work, we presented an approach for improving the performance of sequences of dependent instructions by expressing these sequences as functions, collapsing them into hardware operators and taking advantage of circuit-level redundancy. Our approach does not rely on ILP exploitation, and the associated software optimizations are fairly simple.

We tested an implementation of this approach on a scaled-up superscalar processor with the rePLay framework, and we observed an average performance improvement varying from 3.5% to 19% on optimized code sections.

We are currently investigating the coupling of the function mechanism with address prediction mechanisms to remove many of the *cuts* due to load instructions.

Removing some of the cuts will increase the average function size and the overall speedup.

Also further work should be done to improve trace collection, for example using iterative compilation techniques, in order to better benefit from instruction collapsing.

Chapter 4

Load Squared: Adding Logic Close to Memory to Reduce the Latency of Indirect Loads with High Miss Ratios

4.1 Introduction

The different reconfigurable architectures and grid architectures proposed in Chapter 2, as well as our proposed mechanism to collapse instructions to functions exploit on-chip space differently by mapping part of the application directly to hardware. Still, memory accesses, especially load instructions are the major obstacles to map larger parts of the application to hardware. Indirect memory accesses occur in a wide range of data structures (lists, graphs, sparse matrices,...) and programming constructs (e.g., linkage tables a.k.a global offset tables) and can induce severe performance degradations. However, to the best of our knowledge, only prefetching schemes have been proposed to tackle them: for instance, Roth et al. [Rot98] proposed a hardware prefetching scheme for efficiently traversing pointer-based data structures (graphs, lists), and more recently Cooksey et al. [Coo02] proposed a tagged-prefetching-like scheme for indirect accesses that showed significant potential for commercial applications. However, the scope of such schemes may be limited by the difficulty of properly identifying indirect memory accesses.

In this work, we propose a hardware scheme for reducing the latency of many indirect memory accesses that relies on two key features: (1) a way to identify indirect memory accesses using simple hardware to track chains of dependent loads, rather than "guessing" the occurrence of indirect accesses as in prefetching schemes, and (2) logic (and some tables, see below) to perform indirect loads as close as

possible to main memory (in the memory controller, as evaluated in our experimental set-up, or possibly in memory) in order to reduce the total indirect load memory latency. Moreover, our scheme applies not only to complex data structures traversals, but also to very irregular and hard to predict indirect accesses such as accesses to linkage tables and sparse matrices accesses. Note that access through the linkage table is extremely common due to two factors. One is preemption: A symbol can be preempted if some time after linkage, the object it refers to, or the address of that object, may change. In Linux, all symbols but the internal ones (local data of procedures, static procedures and variables) can be preempted. The second is position-independent code (PIC). For instance, ELF linkers support PIC code with a linkage table in each shared library that contains pointers to static data referenced in the main program. The dynamic linker resolves and relocates all of the pointers in the linkage table. The sad thing is, only one read was required by the algorithm; the indirection here is only due to software convention.

The general code pattern tackled by our scheme is:

```
load b = [a]
add c = b+d
load v = [c]
```

which corresponds to most indirect accesses. (We use the Alpha ISA, whose `ldq` instruction uses a base register and an offset. Thus, code patterns we target may not have an intervening `add`.) In a nutshell, our method operates as follows: the dataflow dependence between two nearby load instructions is identified and recorded, the behavior of both loads (hit/miss) is recorded and if both are found to miss frequently, then the second load is dynamically replaced by a new load instruction, called *load squared*, which takes two arguments, the address `a` of the first load and displacement `d`. The load squared instruction returns the value stored at address `[a]+d`, where `[a]` is the content of the pointer-sized location pointed to by `a`. Note that two loads use address `a`: the first load instruction, and the inserted load squared instruction.

We applied this scheme to all SpecInt and SpecFP benchmarks [Hen00]. Providing results for all SPEC benchmarks is important for two reasons: first, it prevents us from showing only favorable results. Second, it shows that, even though the performance speedup we get are not always remarkable, we slow none of them down. We also show results on 9 of the 10 Olden benchmarks [Rog95]. (We could not make `voronoi` execute correctly.)

We present a state of the art of related work in Section 4.2. In Section 4.3, we present the principles and implementation of our scheme, in Section 4.4, we present the experimental framework and the performance evaluation in Section 4.5.

4.2 Related Work

4.2.1 Linked data structures traversal.

Several software-based prefetching techniques [Lip95; Luk96; Kar00] proposed compiler optimizations for inserting instructions to early-prefetch linked data structures. Roth et al. [Rot98] proposed a hardware mechanism called *dependence-based prefetching* that dynamically identifies loads that access linked data structures, collects them and executes them speculatively in a prefetch engine to slip ahead of the execution. They also classified loads into recurrent load, traversal loads and data loads. *Recurrent* loads are loads which produce addresses consumed by future instances of themselves (e.g. p=p->next), *traversal* loads are class of loads that produces addresses for pointer loads other then themselves, and *data* loads are load data other than addresses. Our mechanism addresses loads that depend on either recurrent or traversal loads.

Beside the speculative nature of dependence-based prefetching, these mechanisms assume that there are enough non-load instructions for the prefetching engine to run ahead of the execution; moreover, dependence-based prefetching has the same caveats as all prefetching schemes, it does not reduce the overall latency of indirect accesses but only hides part of it. Also, the same authors later proposed [Rot99] to add *jump pointers* that are used to prefetch future nodes in a linked list; the approach relies on making pointers explicit in data structures.

Cooksey et al. proposed [Coo02] a content-directed data prefetching mechanism that searches for virtual addresses in data fetched from memory. By then prefetching these addresses, the mechanism implements a form of pointer chasing. However, because the mechanism relies on guessing which addresses are pointer addresses, the number of useless prefetches or the number of missed prefetching opportunities can be significant. For example, we observed in `ammp` that, when the first `next` address is read in a linked list of relatively "large" nodes (larger then the prefetched line size), the prefetcher will only partially prefetch the node, and since the `next` pointer is defined at the end of the node, the prefetcher fails to get the `next` address. A similar approach [Col02] proposes to store pointer addresses in a cache rather than guessing them on-the-fly.

Bekerman et al. [Bek99] proposed to enhance a stride address prefetcher with a correlated load-address predictor to predict linked data structures addresses. Again, this mechanism does not efficiently handle long load dependencies, if the amount of work to overlap is not sufficient. Besides, correlated load address predictor needs prior traversal of a data structure in order to properly learn the correlation. Hence, the first traversal of a data structure, which is more likely to miss in the cache hierarchy, may not benefit from the approach.

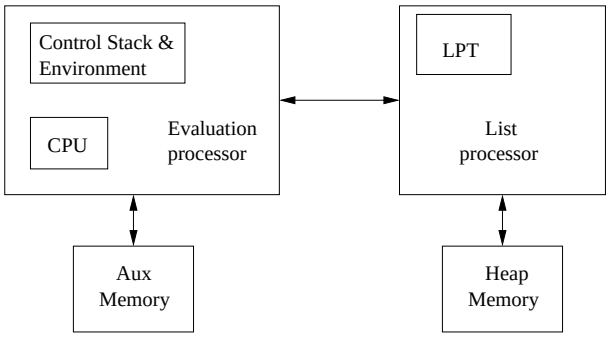


Figure 4.1: *Lisp processor architecture.*

Solihin et al. [Sol02] proposed a User-Level Memory Thread (ULMT) that can be located either in a memory-side logic or integrated in the DRAM. The ULMT is a correlated prefetcher that sends prefetched data to the L2 cache of the main processor. Also, Yang et al. [Yan00] proposed to attach a prefetcher to each level of the memory hierarchy to push data to the processor rather than pulling data from the memory.

Pleszkun et al. presented in [Ple86] a decoupled architecture for Lisp list access. This is one of the few research works that specializes hardware for a certain data structure (lists). Figure 4.1 shows the proposed architecture, which is organized as two processing units: the List Processor (LP) and the Evaluation Processor (EP). the LP allows efficient memory accesses and management through a translation table (LPT) that *virtualizes* lists to the EP. That is, the EP accesses elements using *list primitives* independently from their storage in memory. Further decoupled architectures are discussed in Section 4.6.

4.2.2 Memory-side logic.

FlexRAM [Kan99] is a distributed architecture where compute nodes are attached to local memories. It is more powerful and more complex than what we suggest. It also introduces a new programming model, which we don't.

Impulse [Car99] is a memory controller equipped with address translation hardware. Its architecture is therefore close to our work, but its goal vastly different: Impulse allows the controller to access shadow addresses, i.e., legitimate addresses that are not backed by DRAM. This feature in turn allows applications and/or the compiler to deploy optimizations like mapping noncontiguous addresses to contiguous shadow addresses, thereby improving spatial locality.

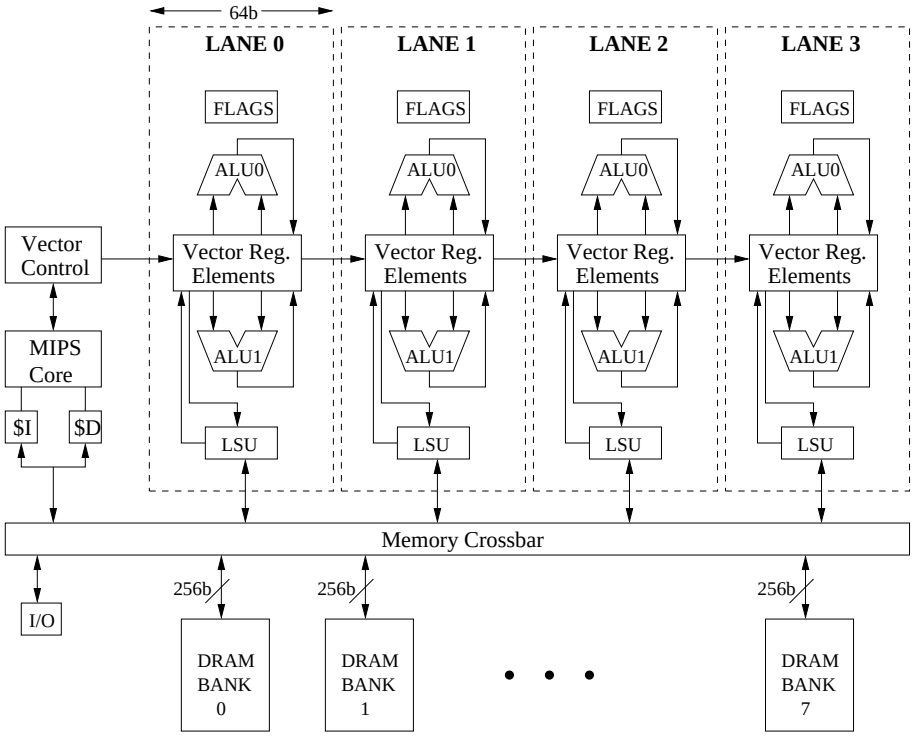


Figure 4.2: *The VIRAM architecture.*

4.2.3 Intelligent memories.

The increasing processor/memory gap motivated a lot of research to physically migrate computations *in* the memory in order to benefit from a minimum access latencies.

Intelligent RAM (IRAM) unifies logic and DRAM on a single chip [Pat97]. IRAM offers about a 100x increase in internal bandwidth and almost a 10x latency reduction because the processor logic is physically *in* the memory.

Because of the low latency of IRAM and its high interleaved nature, IRAM naturally matches the need for a vector processor (VIRAM). Figure 4.2 shows the VIRAM architecture. The VIRAM includes a single issue 64-bit MIPS pipelined core and a vector coprocessor. The register and datapath resources in the vector coprocessor are partitioned vertically into four identical vector lanes to allow higher scalability and lower complexity. The four lanes receive identical control signals on each clock cycle.

But IRAM is slower (by a factor of 1.3 to 2.0) because of DRAM technology,

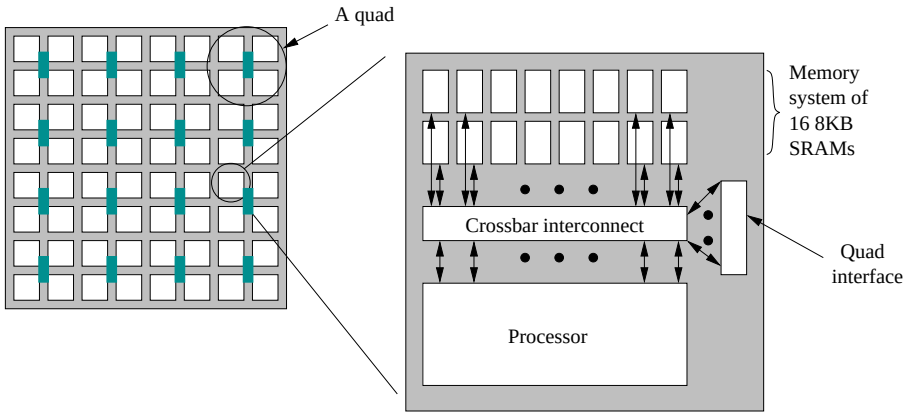


Figure 4.3: *Smart memories architecture.*

and assume that since DRAM chips on PCs has been shrinking since 1986 [Pat97], many PCs may require only a single DRAM chip, which does not favor scalability for high performances even if this argument is valid for portable personal devices. Therefore VIRAM particularly suits embedded multimedia benchmarks [Koz02] and regular vector processing applications.

Smart memories [Mai00] is a tile reconfigurable architecture. It contains an array of processor tiles and on-die DRAM memories connected by a packet-based, dynamically routed network. A tile can contain a 64-bit, 2-issue, in-order processor with 64KB of on-die cache. Each four processor tiles are clustered into a *quad* to support large scale computations as shown in Figure 4.3. Alternatively some tiles may be replaced by embedded DRAM (2-4 MB per tile). The memory system in each tile is highly configurable to match the application demands. The reconfigurable nature of the memory system allows the processor to support different execution modes to exploit different types of parallelism in the target application (ILP, DLP, TLP) like the TRIPS architecture [San03]. Still, the proposed architecture does not fully exploit the proximity of the processor units from the DRAM tiles by proposing an efficient way to program such architecture.

Active Pages [Osk98] shift data-intensive computations to the memory system. Figure 4.4 shows the basic architecture of Active Pages. Active Pages consist of a page of data in DRAM and a set of associated functions that operate on that data.

The authors propose a model of computation which partitions applications between a processor and an intelligent memory systems. They also discuss coordination, computation scaling and data manipulation issues. However, their model lacks an intuitive programming language extension.

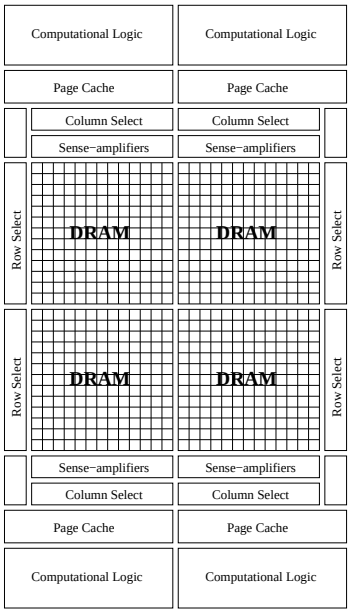


Figure 4.4: *Active Page architecture.*

Initially, Active Pages where proposed on RADram (Reconfigurable Architecture DRAM), a memory system based upon the integration of DRAM and reconfigurable logic. The authors later investigated in [Osk99] to replace the reconfigurable logic of the Active Pages by a scalar processor or a VLIW processor in the computational logic. They demonstrated that the VLIW Active Pages perform as well as their reconfigurable counterparts but with substantial additional benefits in power, area, and programmability.

In general, intelligent-memory approaches allow efficient data accesses, and presents an important opportunity for applications that require high bandwidth and low latency. Nevertheless, aggressive superscalar processors are still dominating in terms of performances and provide higher computation throughput. Furthermore, a lot of the proposed architectures does not provide efficient way to program them or deal with complex data structures. The Load Squared approach provides a way to migrate critical address calculation parts of the application closer to memory without having to pay the high cost and architectures changes of intelligent memories.

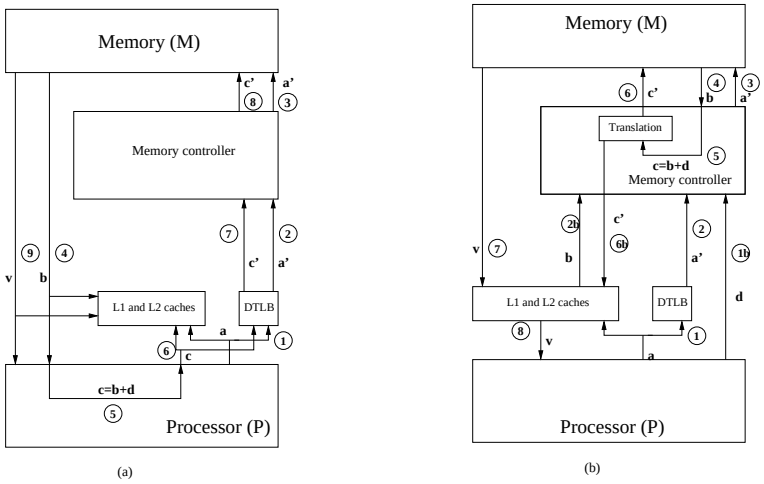


Figure 4.5: *Load squared architecture (a) Normal Operation (b) Load Squared Operation.*

4.3 Principles

The general principle of our approach is to reduce the overall latency of two dependent loads, and more precisely, the time necessary to perform two round-trips to memory, in case both loads miss. As noted in the introduction, the value of the first load is brought back to the processor for the sole purpose of computing the address of the dependent load, and in many cases, this computation simply consists in adding an integer offset to the data fetched by the first load. If memory-side logic is added to perform such computations, it is no longer necessary to go back to the processor to compute the address of the second load, so that the overall latency of the two dependent loads can be significantly reduced. Observe again that this logic may be implemented in memory, even though we assume in our simulations that it is located in the memory controller.

Figure 4.5 describes both the behavior of two dependent loads (a) on a simplified standard memory hierarchy, and (b) on a memory hierarchy augmented with the *load squared* technique. (Part (c) is described in Section 4.4.) In the two cases, we assume two dependent load instructions, where the first one has issued a virtual address `a` corresponding to data `b`, which is used to compute a new virtual address `c` for the second load instruction. On a standard memory hierarchy, the two requests are processed as follows. In Step 1, the processor sends virtual address `a` to the first level cache and the TLB. Assuming the first load misses, physical address `a'` is sent to the memory-side logic in Step 2. The memory-side logic accesses main memory M

in Step 3 and value b (along with its corresponding cache line) is sent to the cache and the processor in Step 4. The processor computes virtual address `c=b+d` in Step 5 and in Step 6 makes a second request to the memory hierarchy through Steps 7 to 9.

Figure 4.5(b) shows the *load squared* operation. In this example, we assume both loads miss, and we will see in the next section that the mechanism includes a predictor for deciding when both loads are most likely to miss and for applying the *load squared* technique only in this case. In Step 1, virtual address `a` is sent to the cache (miss) then to the memory; the mechanism identifies the pairs of dependent loads, and when the second load is ready to issue, it is replaced by a *load squared* which sends the address `a` of the *first* load along with the offset `d` to memory. In Step 3, the memory-side logic sends address `a` to memory, gets data `b` in Step 4, computes address `b+d` in Step 5, then immediately sends the new virtual address `c` to memory after translating it in its own TLB in Step 6. Following the idea of the hardware page walker on Itanium 2 [Int], address translation is done by the memory-side logic entirely in hardware. This assumes the page table follows a predefined format, which the OS must follow. In Step 6, a fetch of `v` at address `c'` is sent to memory; that request returns to the processor in Steps 7 and 8.

Area Cost. The main area cost of the mechanism lays in the additional TLB and the adder located in the memory-side logic, and the prediction tables used to detect dependent loads which both miss (see next section). Several modifications are also necessary in the cache controller and the miss address file.

Coherence. The value of `v` is speculative since the content of memory may not be up-to-date at this step: if some of the cache levels are write-back, they may contain a dirty line that contains the most up-to-date content of address `c'`. (Even if all cache levels are write-through, the request from the memory-side logic may arrive to memory just before the copy back is complete.) To ensure memory coherence, the memory-side logic sends a read request for address `c'` to all cache levels, at Step 6b, simultaneously with the memory fetch in Step 6. If the cache read is a hit, the value in the cache is at least as recent as the value read from memory, and this value is sent by the cache to the processor (not to the memory-side logic) as the final result of the load squared. Eventually, value `v` read in Step 7 from memory is sent by the memory-side logic to the main processor. This value is propagated by the cache hierarchy and is simply ignored by the miss address file. Alternatively if the read is a miss, no further action is taken by the cache at this point (except perhaps for initiating a fill request) and the memory read initiated by the memory-side logic at Step 7 provides `v`.

Note also that a coherence issue also arises when a load squared is followed by a store. In this case, the processor cannot disambiguate the load squared from the

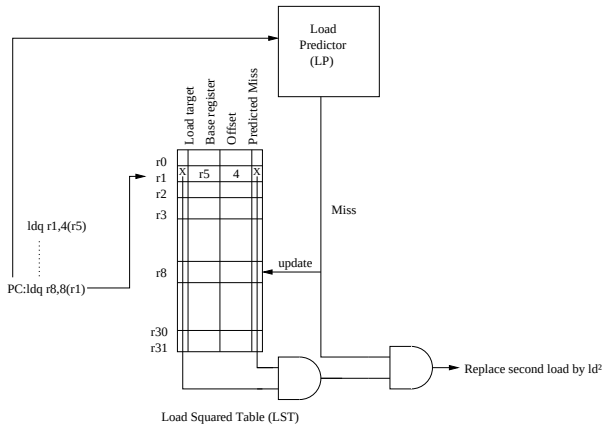
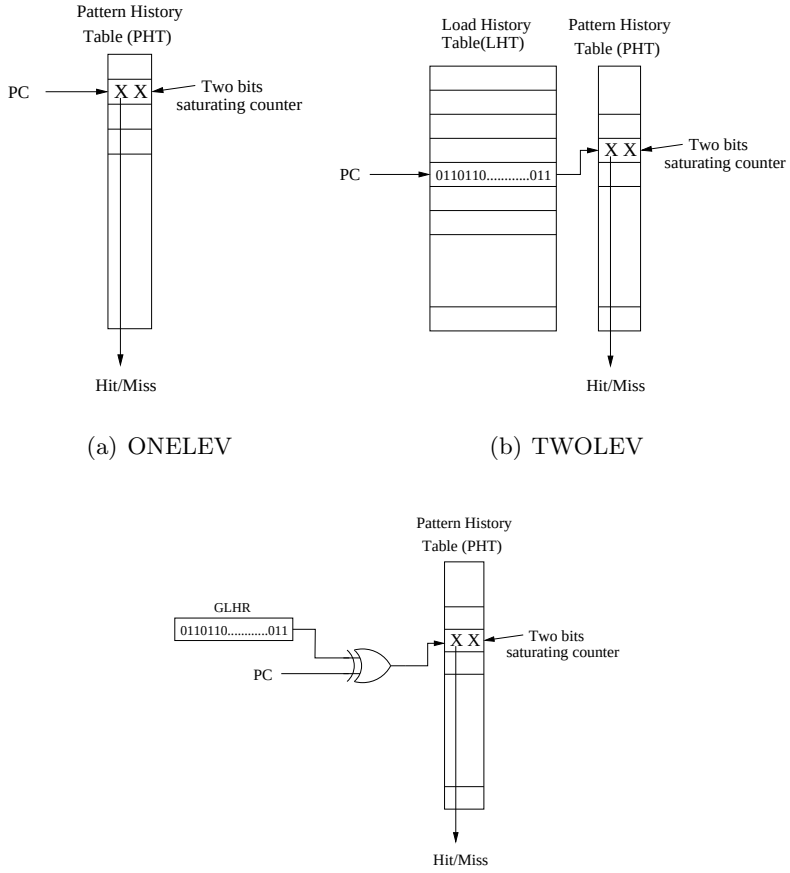


Figure 4.6: *Predicting and issuing load squared.*

store, even dynamically (it can only disambiguate the first load and the store), and must conservatively assume a WAR dependence. We solve this issue by enforcing the following restriction: a store following a load squared cannot be presented to the memory hierarchy before the load squared completes. Since stores are seldom on the critical path, delaying them that way is not a significant performance issue. Other solutions to this problem, which we will consider in future work, include ways to let the compiler assert the store and preceding loads squared do not alias.

Detecting a dual miss. Replacing the second load by a load squared is only worthwhile if both loads miss in all cache levels, otherwise the time spent accessing the DRAM twice may be higher than a full memory access plus a cache access in case of one hit and one miss (let alone two cache accesses in case of two hits). Therefore, we complement the *load squared* approach with a predictor for detecting when two dependent loads are likely to miss; the second load is replaced by a *load squared* only in this case.

Optimization when the first load is a miss. A load squared request is sent to memory via the existing memory hierarchy. Therefore, it only makes sense to check on the fly whether address `a` misses or hits the different caches. If the address hits any cache level, value `b` is read and sent to the memory-side logic, which directly computes `c` and fetches the corresponding value. This can be seen as an on-the-fly change of opcode, from that of a load squared instruction with arguments `a` and `d` to that of a "load squared with hit" instruction with arguments `b` and `d`.



(a) ONELEV (b) TWOLEV

(c) GSHARE

Figure 4.7: *Load predictors.*

4.3.1 Detecting and issuing Load Squared

In addition to the memory-side logic, the mechanism relies on a table, the Load Squared Table (LST), to dynamically identify chains of dependent loads, and on a predictor to guess which Load/Load chains probably result in a double miss. Note that because the Alpha instruction set provides a load instruction with offset, we consider only pairs of directly dependent loads.

The LST is shown in Figure 4.6, and its operation is illustrated with instructions `ldq r1,0(r5)` and `ldq r8,8(r1)`. The LST keeps track of "producer" loads:

when a register is the target of a load (`r1` after the first load), it is marked as such in the LST. The base register of the load, and the value of its offset, are also recorded in the second and third column of the LST, respectively. When a candidate "consumer" load is decoded, the entry in the LST corresponding to its base register is read. If the base register is marked as the target of a load, then a Load/Load chain is detected. Of course, any intervening instruction (other than a load) that modifies a load target register causes the load target bit of the LST to be cleared. Note also that this mechanism spots Load/Load chains in distinct, and possibly distant, basic blocks.

Detecting Load/Load chains is the first condition for replacement by a load squared; the second is to predict a double miss with high probability. Each time a load instruction is decoded, it is predicted as a hit or a miss (see predictor below), and the predicted miss bit of the LST entry is updated accordingly. This way, when the second load is decoded (`ldq r8,8(r1)` in the example), the hit/miss prediction for the producing load is available. Separately, the consumer load is itself predicted as a hit or a miss; if it is predicted missing, and it is fed by a load, and that load was also predicted missing, then the second load is replaced by a load squared. The base address (`a` in Figure 4.5) of the load squared is computed from the second and third columns of the LST. The offset of the load squared is that of the second load. For instance, `ldq r8,8(r1)` is replaced by a load squared whose target register is `r8` and input operands are immediate 8 and the target address of the first load, i.e., `4(r5)`.

The Load Predictor. We have found that it is possible to relatively accurately predict whether pairs of dependent loads will miss in all cache levels before the second load is issued, and we have come up with a hardware mechanism to implement a prediction strategy. The strategy is similar to history-based branch predictors [Smi81; Yeh91; McF93]: prediction is based on both the hit/miss behavior of previous loads and on the target load address. We studied three types of load predictors: the one-level load predictor (ONELEV), the two-level load predictor (TWOLEV) and the gshare (GSHARE) load predictor, see Fig. 4.7. A predictor is read when the load is decoded and updated when it is committed.

The one-level load predictor (ONELEV) shown in 4.7(a) consists in one table of 2-bit saturating counters, the Pattern History Table (PHT). The PHT is indexed by the PC of the load. When the load misses, the counter is incremented, and decremented otherwise. Therefore, the predictor predicts the load will miss if it missed at least on the last two occurrences. The one-level load predictor behaves particularly well with loops that access chained data structures not yet present in cache (cold misses).

The two-level load predictor (TWOLEV) further exploits the historical behavior

Fetch width	8
Issue / Decode / Commit width	8
RUU size (Inst. window- ROB)	128
LSQ size	32
ExeUnits	4 IALU, 1 IMULT, 4 FPALU, 1 FPMULT
Branch	Combined, 4K entries bimodal, and 2 level Gap predictor, 8K 2nd level entries, 14 history wide, 1K meta-table size 7 cycle BR resolution
L1 DCache	16kB, 4 ways, 1 cycle
L1 ICache	16kB, 1 way, 1 cycle
L2 Unified Cache	256kB, 4 ways, 6 cycles

Table 4.1: *Baseline configuration of the processor.*

of each load. It includes a Load History Table (LHT) that records the hit/miss behavior history of each load, see Figure 4.7(b). The LHT is indexed by the PC of each load, the corresponding history of the load indexes the PHT.

The gshare load predictor (GSHARE) exploits the correlation between several previous loads by maintaining a global behavior history in the Global Load History Register (GLHR). The GLHR is XORed with the PC to index the Pattern History Table as shown in 4.7(c). We further study the performance of each predictor in Section 4.5.

4.4 Experimental Framework

We simulated an 8-way superscalar processor using the SimpleScalar 3.0 Alpha toolset [Bur96]; Table 4.1 shows our baseline simulation environment.

We modified the system bus to operate at 1/5 the processor frequency. Also, to properly measure the effect of load squared, we subdivided the processor-to-memory round-trip latency into: 60 cycles for the processor-to-memory-side logic latency, 10 cycles for the memory-side logic latency, 20 cycles for logic-to-memory latency, 80

cycles for a DRAM access, 20 cycles from the memory back to the memory-side logic, and 60 cycles from the logic back to the processor. The TLBs in the memory-side logic and in the CPU have 32 entries. The cost of a TLB miss in the memory-side logic is 80 cycles, and 30 cycles for the CPU's TLB (because a CPU TLB miss can be serviced by the cache). We also assume a 60-cycle latency for direct transfer from the memory to the processor, so that a traditional read to memory (without TLB miss) has a total latency of 240 cycles.

The latency of a load squared depends on whether address `a'` was a cache hit (Steps 2 or 2b of Fig. 4.5), and on whether address `c'` is a hit (at Step 6b). Assuming the memory-side TLB hits, the latency can therefore be one of the following: 60+10+60 = 130 cycles (two hits), 60+10+20+80+60 = 230 (`a'` hits), 60+10+20+80+20+60 = 250 (`c'` hits) and 60+10+20+80+20+20+80+60 = 350 (no hits). As said earlier, any miss in the memory-side TLB adds 80 cycles.

Predictor configuration. We used a Pattern History Table (PHT) of 4K entries for the ONELEV load predictor, a 4K-entry Load History Table (LHT) for the TWOLEV load predictor. The PHT for both the TWOLEV and the GSHARE load predictors are indexed using a 14-bit history length (XORed with the PC for the GSHARE predictor)

Benchmarks. We simulated all SpecInt and SpecFP benchmarks [Hen00] as well as 9 of the Olden [Rog95] benchmark suite. We simulated 100 Million instructions for each benchmark, and we fast-forwarded 2 billion instructions for the SpecInt and the SpecFP benchmarks.

4.5 Performance Evaluation

4.5.1 Load Squared potential

Figure 4.8 shows the percentage of loads that are directly dependent on other loads (Load/Load). On average, 12.5% and up to 47% (`ammp`) of all executed loads are directly dependent on another load. The Miss/Miss columns of Figure 4.8 show the percentage loads that are fed by another load, and such that both miss at all cache levels. Those Miss/Miss loads are those that should be replaced by loads squared. While the percentage of Miss/Miss occurrences is relatively low, their effect on the performance is significant because of the high latency of memory accesses.

Note also that we did not observe significant differences in TLB miss ratios with and without loads squared.

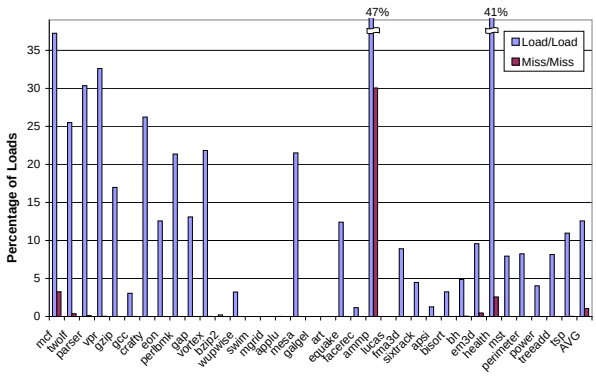


Figure 4.8: *Potential loads squared and Miss/Miss occurrences.*

4.5.2 Efficiency of load predictors

Because the *load squared* mechanism targets pairs of loads that both miss in the cache hierarchy, replacing a load by a load squared when one of the two loads (or both) hits may negatively affect performance.

On the other hand, failing to catch appropriate Miss/Miss pairs naturally means achieving only a fraction of the potential performance improvement. Therefore, the load squared mechanism is fairly sensitive to the efficiency of the load miss prediction. Figure 4.9 shows the load prediction rates of the three types of predictors discussed in Section 4.3.1, i.e., how often a dynamic instance of a load is correctly predicted as hitting or missing (the average load prediction rate is 87%). We notice that the TWOLEV and the ONELEV load predictors exhibit a better prediction behavior than GSHARE for many benchmarks. The relative low prediction rate of the GSHARE load predictor suggests that loads behavior have little correlation.

Interestingly, control speculation has a strong impact on the load prediction rates. While Figure 4.9 gives the rate for all loads, Figure 4.10 only considers non-speculated loads, or loads that were on a correctly predicted path. (Both figures are on the same scale.) Clearly, the accuracy of our Miss/Miss predictors is then much better, which indicates a possible correlation between branch prediction and Miss/Miss prediction. How to leverage this observation to improve our Miss/Miss predictors is left for future work.

Figure 4.11 shows the percentage of loads replaced by loads squared, for each predictor. This figure should be compared with Figure 4.8. Miss/Miss sequences are numerous in `ammp`, `mcf` and `health`; a few occurrences also occur in `twolf` and `em3d`.

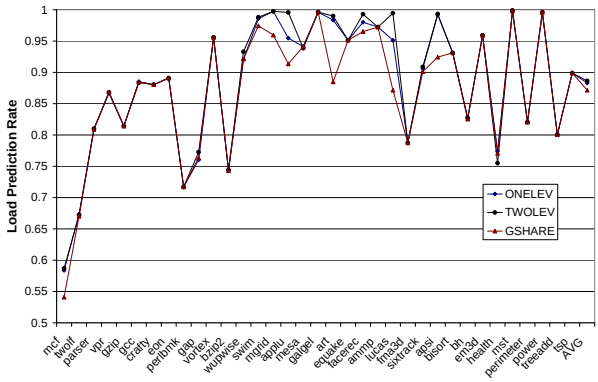


Figure 4.9: *Load prediction rate.*

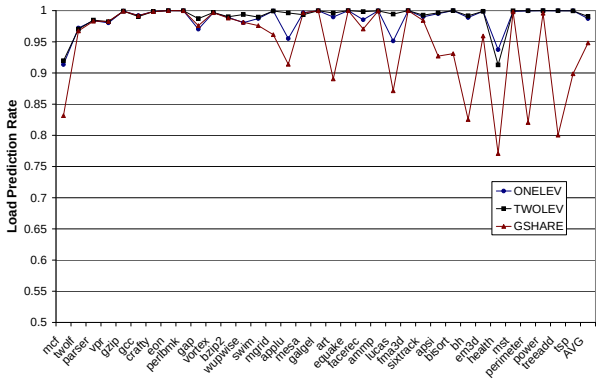


Figure 4.10: *Non-speculative load prediction rate.*

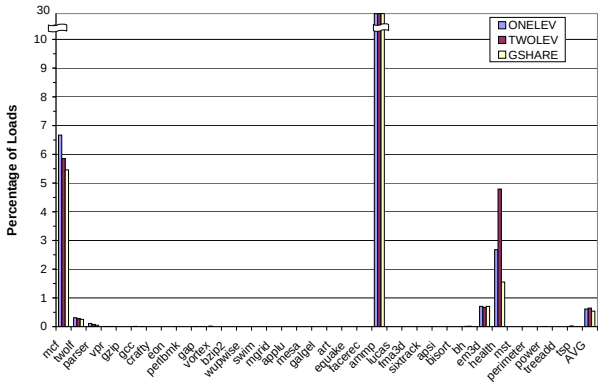


Figure 4.11: *Percentage of load squared.*

4.5.3 Performance results

Figure 4.12 shows the speedups achieved with the load squared mechanism. Our approach significantly improves the performance of several benchmarks (5% for `twolf`, more than 7% for `em3d` and about 50% for `ammp`). Importantly, performance is never degraded.

We also notice that the speed-up for `mcf` is smaller than we could have hoped for from looking at Figure 4.11. Comparing Fig. 4.8 with Fig. 4.11, a possible explanation is that our predictors may be too aggressive on this benchmark. However, a deeper investigation shows that the root cause may be more complex. Consider Table 4.2, where each row provides data for a specific static load in the main loop of procedure `refresh_potential`, which accounts of 35% of the total execution time. The loads considered here are the most frequent loads that are fed at least once by another load (control flow may imply that not all dynamic instances of a load are fed by the same instruction). Note that there are four back-to-back loads at addresses 7980 to 798c. Column 1 gives the PC of each load, and Column 2 their dynamic counts. Column 3 says how many instances were fed by a load. Columns 4 and 5 indicate how many instances missed all cache levels, and how many both missed and were fed by a load that missed, respectively. Clearly, Column 5 shows our sweet spot. Column 6 gives the average latency, in cycles, elapsed between the issuing of the producer load and the write-back of the current load, when load squared is turned off. When it is turned on, Column 7 indicates how many instances were converted to loads squared using the ONELEV predictor, and Column 8 gives the resulting average latency, in cycles, of the load squared.

Table 4.2 first confirms that caches do their work well on these loads: most

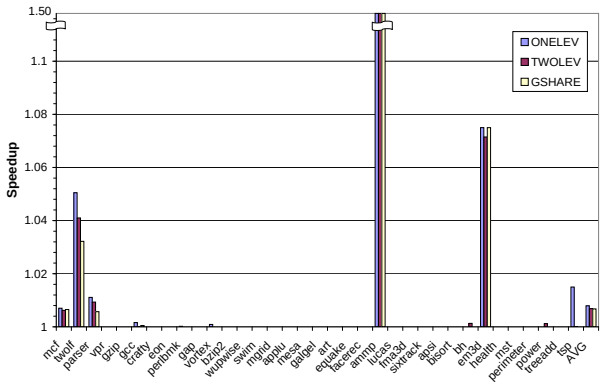


Figure 4.12: *Speedup obtained with the load squared mechanism.*

PC	Occur. count	Fed by a LD	Miss count	Both loads missed	Avg Lat (w/o LD2)	LD2 Count	Avg Lat (w/ LD2)
0x120007974	2626041	740306	314269	62	553.163	0	549.284
0x120007980	2614229	734581	2231030	4830	555.701	51	554.132
0x120007984	2610730	733853	1418065	769	551.326	16	548.146
0x120007988	2598325	2595990	2597224	2154484	517.025	2467031	371.110
0x12000798c	2597455	2589596	48916	392	283.307	0	262.348
0x1200079c0	2621919	743407	567721	1350	551.763	24	550.839

Table 4.2: *Stats for 181.mcf.*

Load/Load's are not Miss/Miss. The ONELEV predictor sees this correctly and, where double misses are rare, it seldom converts the 2nd load into a load squared. This is not true, however, for the load at address ending in 7988: most of its instances miss and are fed by a load that also misses. Also, most of them are converted into loads squared – again, a bit too many. Nevertheless, the result on the latency of this specific load is dramatic.

4.6 Perspectives: Explicitly Migrating Memory Computations Closer to Memory.

We have shown that it is possible to tackle the long latency of even very irregular indirect memory accesses using a simple memory-side logic, and processor-based hardware add-ons. Still, a lot of potential resides in migrating larger parts of the application closer to memory or in the memory rather than address calculations.

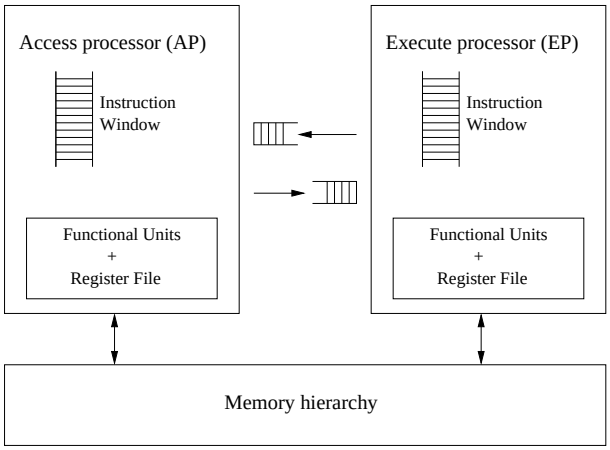


Figure 4.13: *A decoupled architecture.*

We conclude this chapter by presenting a more general decoupled architecture associated with a language extension to C that allows the user to explicitly specify which part of the application to migrate to memory. This proposed decoupled architecture speedups computations that involve irregular or non-contiguous data structures. This work is still on-going, and only few microbenchmarks were studied. We present an overview of existing decoupled architectures then our proposed approach. Further architectural details as well as some preliminary results are given in Appendix A.

4.6.1 Decoupled architectures

The idea of decoupling computation instructions from memory access instructions was initially proposed in order to parallelize the two producer/consumer processes and hide memory communication delays. One of the first decoupled architectures was proposed in [Smi82]. Smith proposed an operand Access/Execute decoupled architecture via two separate scalar processors, each with its own instruction stream (two distinct programs), and communicating via architectural queues as shown in Figure 4.13. This architecture is more assimilated to an early two-instruction issue *constrained* out of order processor [Smi84]. A similar pipelined architecture was also presented in [Goo85].

The ZS-1 processor [Smi87] splits the instruction stream to two separate pipelines, an X pipeline to execute floating point operations and an A pipeline to execute memory access operations as well as integer operations. Note that integer operations are

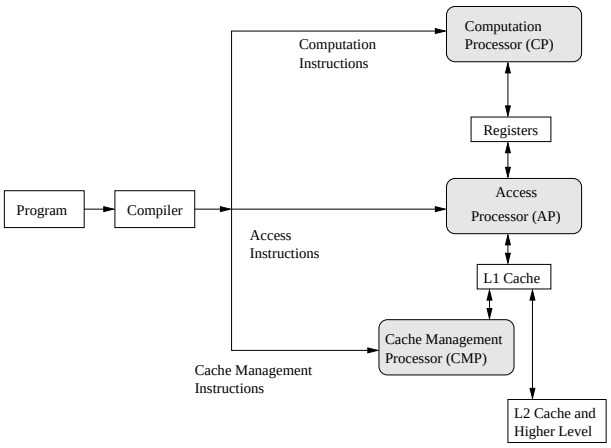


Figure 4.14: *The HiDISC architecture.*

treated like memory access operations because the ZS-1 was targeting scientific and engineering applications. This *decoupled* architecture is somewhat similar to the Alpha 21264 architecture [Kes99] where the memory/integer instructions are issued on a different path than the floating point instructions. The WM architecture [Wul92] proposed a similar decoupled machine using FIFO communicating buffers and 3-operand instructions.

Jones et al. showed in [Jon97] that the principal advantage of decoupled architecture is that a decoupled machine allows to have an effective instruction window size greater than the sum of the individual windows sizes of each of the Access processor (AP) and Execute Processor (EP): because the access instructions slip ahead of the execution instruction, they feed the execution with the data it needs on time, so that the processor behaves as if it has a larger instruction windows where older instructions are the execution instructions. The potential of decoupling as well as its effectiveness are further studied in [Bir93] and [Zha98]. Furthermore, a decoupled SMT architecture was proposed in [Par01] to improve the memory latency tolerance of simultaneous multithreading. A partitioning compiler that automates Access/Execution separation was proposed in [Ric01].

HiDISC [Ro03] is one of the few decoupled architectures that targets pointer and data intensive applications. In addition to the access (AP) and computation (CP) processors, HiDISC proposes a cache management processor (CMP) at the L1 cache level to keep the cache supplied with data which will be used by the AP. Figure 4.14 shows the HiDISC architecture. Also, HiDISC proposes a compiler that performs program slicing to separate computation, access and prefetch instruction streams.

Finally Roth et al. proposed in [Rot00] to decouple accesses that are likely to miss in the cache. This access stream is executed as a speculative thread. But this thread also suffers from the same high memory latencies of the initial thread.

To summarize, the reviewed decoupled architectures allow memory accesses to slip ahead of the execution to continuously supply it with data. However, decoupled approaches suffer from three major disadvantages: because they still operate at the processor level they suffer from the same latency problems of traditional approaches. Also, these architectures can do little with chained data structures, especially when the work to overlap is small. Finally, recent out-of-order superscalar architectures having large instruction windows virtually decouple access from execution, so explicitly decoupling access from execution becomes useless unless access instructions can provide data faster than superscalar processors. These issues are further explored in the Data Structures Conscious Machine (DSCM).

4.6.2 The Data Structures Conscious Machine (DSCM)

The proposed decoupled architectures require considerable compiler effort or complex architectural mechanisms to properly separate access code from execute code. Particularly, the proposed mechanisms sometimes fail to improve the performance of codes that involve complex data structures traversals. Furthermore, many of those proposed decoupled architectures execute the memory access instructions in the processor, and does not propose to migrate these instructions closer to memory in order to benefit from lower latency memory accesses.

Also, migrating part of the application to an intelligent memory requires that either the memory processor executes as fast as the main processor, or that care should be taken that the appropriate part of the application be migrated to memory. For example, consider the procedure `Traverse` that traverses a linked-list in Figure 4.15(a), if the `process` procedure has a lot of floating-point computations for example, migrating the whole function to the memory may worsen the performance if the processor in the memory is not as fast as the main processor. Besides, existing Linked Data Structures (LDS) prefetching schemes and decoupled architectures can do well in such cases because the amount of work between two consecutive memory accesses is high.

More generally, we notice that a lot of semantic is lost when programs are compiled and executed on processors: consider for example the simple case where a user wants to perform an operation on a list of elements. Assume that the user decides to implement a linked list to do this, so he writes a procedure similar to the one shown in Figure 4.15(a). Then the compiler compiles this code into a sequence of instructions that access the memory to load the nodes of the linked list and process them. At this point, semantic was lost twice, the first time when the user decided

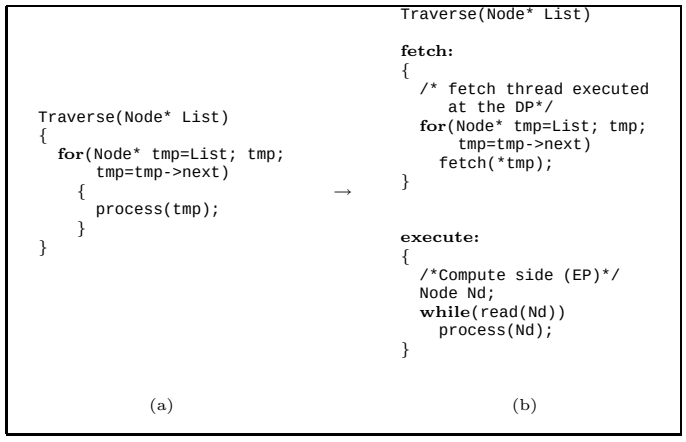


Figure 4.15: *List traversal in DSCM: (a) C (b) Decoupled C.*

to implement its list as a linked list while the nodes might be processed in parallel, the second time, when the compiler compiled a sequential sequence of instructions that traverse the linked list (while nodes could be processed in parallel). On the other hand, a lot of effort is done to restore this semantic, so the compiler does some transformations to parallelize this loop (by unrolling it for example). Then micro-architectural optimizations can be introduced to further dig up this semantic, such as prefetching to dissociate memory accesses from data processing. The architecture also pipelines the execution, uses branch prediction to maximize the throughput of execution, and uses register renaming to break false data dependencies that where introduced by the compiler. In other words, a lot of effort is done to restore the semantic that was lost when the algorithm was programmed, while the user could sometimes easily transmit this semantic to the architecture, provided he had the appropriate language extensions to do so.

In the approach we propose we want to

- Present language extensions that allows the user to explicitly pass semantic information on memory accesses to the architecture.
- Design a decoupled architecture that exploits the language extensions as well as intelligent memory approaches (low latency and high bandwidth).

The basic idea is to explicitly separate code for accessing data structures and code for processing them in separate threads that communicate between each other.

Figure 4.15(a) shows a traversal of a linked list in which some processing is done for each node (procedure `process`). And Figure 4.15(b) shows an implementation

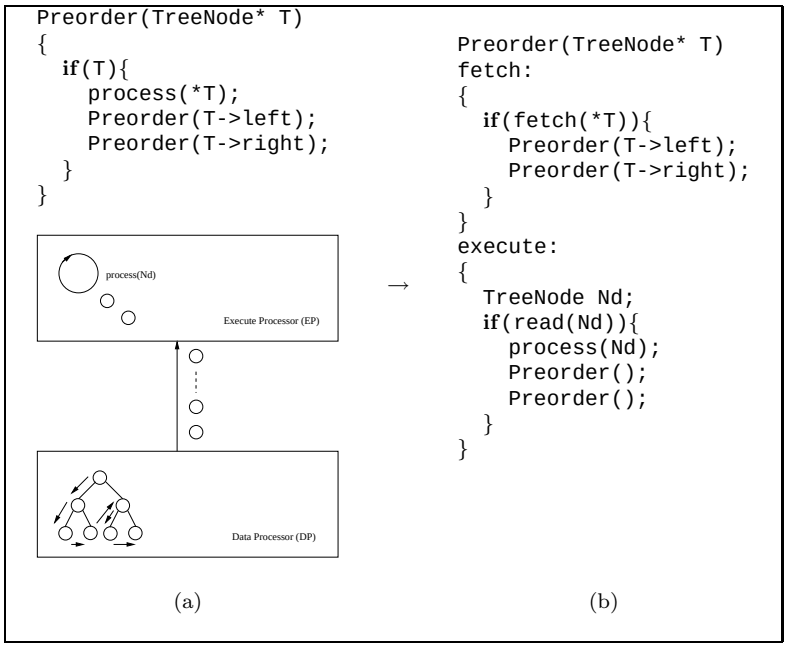


Figure 4.16: *Tree traversal in DSCM.*

of the same code using our language extension. In this example, the user defines two separate threads, the `fetch` thread that fetches data and the `execute` thread that processes these data. The two threads communicate with each other trough the two functions `fetch()` and `read()`, this later function collects data fetched by the `fetch()` function to process them. The main advantage of the explicit separation of fetch and execute codes in a language is to allow the user to explicit the parallelism between fetching and executing. The `fetch` thread may execute in two different ways: either as a thread running on a simultaneous multithreaded processor [Tul95], in a way similar to the miss/execute decoupled approach proposed by Roth et al. [Rot00], or in an intelligent memory to benefit from a low latency memory access. We discuss both architectures in Appendix A.

An important measure in decoupling memory access from execution is the amount of work that is overlapped with a memory access, for example, the amount of computations in the `process` function of Figure 4.15. If the fetch thread is executed in an intelligent memory, and if the amount of computations is very low (for example a simple addition), the user may decide to migrate the whole procedure in the fetch thread (i.e., in the intelligent memory) to benefit from low latency accesses. So the explicit decoupling of access/execute at the language level allows the user to define an appropriate partitioning of the code according to the semantic and the complexity of the data structures he uses.

Figure 4.16 shows an example of tree traversal using the proposed language extension; an important benefit is the *push* aspect where nodes are pushed serially to the execution for processing, rather than pulled on demand. That is, the processor does not have to wait for each node to calculate the address of the next. Instead, the memory supplies the processor with the data in a dataflow manner

The preliminary performance results shown in Appendix A showed that the Data Structure Conscious Machine (DSCM) can achieve substantial performance for applications that have irregular accesses corresponding to complex data structures. A speedup of up to 5X can be achieved on a simple list traversal. Furthermore, this approach does not require a detailed understanding of complex processor architectures. Future works involve further detailing the decoupled architecture and validating this approach with a larger spectrum of benchmarks.

Chapter 5

Conclusions and Perspectives

5.1 Summary

The high clock rates and the poor wire scaling of upcoming processors, as well as their increasing complexity, may lead to a lot of difficulties to scale performance with upcoming integration technologies. In this thesis we explored several alternative approaches to better exploit on-chip space and reduce the effect of the memory wall, especially for non-numeric applications that have little or no ILP and involve accesses to complex data structures.

In a first approach, we significantly improved the performance of code regions with little or no ILP by dynamically extracting sequences of dependent instructions and collapsing them into *Functions*. We proposed an implementation of hardware operators that map these functions, and we also implemented the approach on a scaled-up superscalar processor with the rePLay framework. Furthermore, we studied the potential performance of our approach in idealistic rePLay and cache environments.

Because dependent loads are special forms of instruction dependencies, and because they cannot be collapsed in combinational functions, we explored a complementary approach that improves indirect loads having high miss ratios. The load squared approach transforms loads that depend on other loads and have high miss ratios into load squared instructions. The load squared operation migrates the address calculation closer to memory, typically to the memory controller, to speedup the address calculation process in chains of dependent loads. Our approach improved the performance of codes that have linked data structures with high miss ratios.

Finally, we extended the notion of migrating address calculations closer to memory by migrating all or part of the application to an intelligent memory. The novel

aspect of this decoupled architecture lies in the ability to explicitly specify what part of the application should be migrated to the memory through a proposed language extension to C.

5.2 Perspectives

This research opens several avenues for further exploration and future work.

In Chapter 3 we showed that integer applications have a great potential of instruction collapsing if we consider the dynamic execution of instructions. Still, the implementation in a superscalar processor could not extract a lot of potentially large sequences of dependent instructions. Further work should address the issue of extracting those sequences of dependent instructions. Techniques such as iterative compilation may improve significantly the performance of our approach.

With respect to the load squared approach presented in Chapter 4, further work should be done in considering other patterns than the `load-add-load` patterns studied. Also, other improvements are possible in the load predictor we presented, specifically by correlating it to the branch predictor.

Finally, The DSCM approach presented in the perspectives of Chapter 4 is still an on-going work and needs to be further explored and analyzed. Because we hand-coded the micro-benchmarks we studied, a compiler may allow larger benchmarks to be implemented and studied to validate this approach.

More generally, the billion-transistor era will open the way to combine a lot of the approaches and ideas we explored in this thesis. Also, The advent of new emerging technologies such as nanotechnologies [Gol01], and non-volatile high performance memories that can be embedded in the processor, like MRAMs [Des02], will probably shift computer architecture research to more innovations in spacial computing and also to new challenges such as fault-tolerant architectures and new computing models and languages.

Appendix A

The DSCM Architecture

In this part of the thesis, we discuss a proposal for a DSCM architecture. As stated in Section 4.6.2, we implemented two different architectures: the first one is physically decoupled and the access threads are executed in an intelligent memory, and the other is the single-processor DSCM architecture, in which all access and execute threads are executed in the main processor.

A.1 The DSCM using intelligent memory

The architecture we propose is a decoupled architecture and consists of two separate processors; the Execution Processor (EP) is the main processor, and the Data Processor (DP) is physically located in the DRAM like IRAM [Pat97] to benefit from lower latency and higher bandwidth. In this first study, we assumed both processors are 4-way superscalar. The EP executes the main program thread and dispatches access threads to the DP (through the added instruction `ldth` and `ldth`). The DP is 2-threaded to support executing both `fetch` and `store` threads. Figure A.1 shows the DSCM decoupled architecture.

The EP and the DP communicate with each other using FIFO Buffers. Data are read from memory and are sent to the *D-buffers*. The `read()` function shown in Figure A.4(b) reads data from the D-buffer instead of normal registers. On the DP side, the *M-buffers* are used to store data from the EP (using the `store` function, see the example of Figure A.5) to the DP (using the `read` function in the `store` thread). In this study, we assume that the memory consistency is handled by the user, that is we suppose that the user does not write to cache hierarchy at the addresses accessed through the D-buffers, or vice-versa.

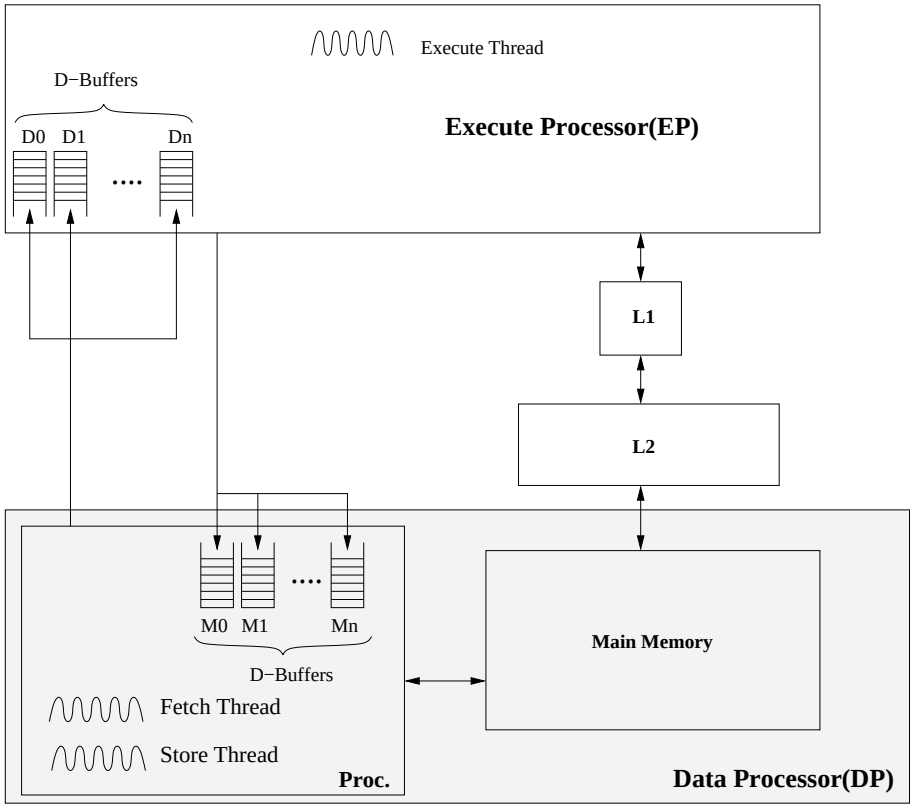


Figure A.1: *The DSCM architecture.*

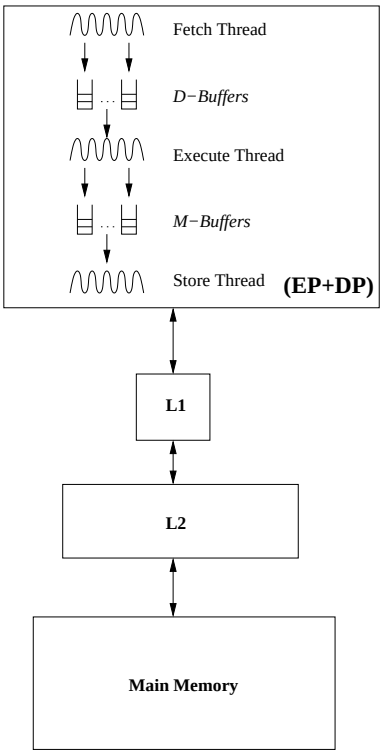


Figure A.2: *A single-processor DSCM architecture.*

A.2 The single-processor DSCM architecture

Our mechanism can also be implemented on a single SMT processor where all execute, fetch and store threads are executed on a single processor as shown in Figure A.2. The fetch thread supplies the execute thread with data, so it can slip ahead of the execution. The idea is also similar to the SMT architecture presented by Roth et al. [Rot01], but the prefetch is not speculative. Like the intelligent-memory DSCM architecture, threads communicate with each other using FIFO buffers as shown in Figure A.2.

A.3 DSCM instruction set extension

We extended the Alpha instruction set to allow buffer communication between the Execution Processor (EP) and the Data Processor (DP). Figure A.4(c) shows the corresponding assembly code of the list traversal of Figure A.4(b). We summarize the DSCM instruction set extension in Figure A.3, the ISA extension defines four different types of instructions.

Load/Store buffer instructions. `ldBm` and `stBm` load or store data from or in memory at the address pointer stored in the register operand. To allow efficient streaming instructions and also to handle complex data structures (C `struct` for example), a 9-bit immediate operand specifies the size of the data to load or store. For example, in Figure A.4(c), the instruction `ldBm D0,(r16),16` is executed in the DP and corresponds to the `fetch` command in Figure A.4(b). This instruction loads 16 bytes (the whole node of the list) from the memory at the address pointed to by `r16` and sends the data to the buffer D0 in the EP. Similarly, the `stBm` reads data from the M-buffers of the DP (or the store thread if a single processor is used) and stores them in the address specified by the register operand instruction.

In addition to normal data, the DP can send a special EOD (End Of Data) token to the processor. Such tokens are used to end data traversal loops in a dataflow manner. The two special instructions `ldEOD` and `stEOD` are used to push special EOD (End Of Data) tokens in the buffers of the processor and the memory respectively. For example the `ldEOD D0` in the fetch thread of Figure A.4(c) sends an EOD at the end of the list traversal loop, a special branch instruction in the EP ends the loop in the execute thread.

DSCM branch instructions. The two instructions `bod` (branch on data) and `boed` (branch on end of data) are used to implement *dataflow* based loops like the `while` loop of the `execute` thread of Figure A.4(b). In this example the `bod D0,L5`

```
1. D-Buffer transfer instruction format
=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=
comment: Similar to immediate operate instruction format
         Memory /processor communication instructions
 31              26 25         21 20             12 11         5 4       0
-----------------------------------------------------------------------------
|    OPCODE (6)     |   Buffer (5)|      Size (9)    |  Function(7)| @ reg(5)|
-----------------------------------------------------------------------------
opcode.function
    02.01    LD THREAD    ldBm Di,ri,size       Load from mem "size" bytes addressed
                                                at ri in buffer Di
    02.02    LD THREAD    ldEOD Di              Send and "End Of Data" to buffer Di of processor

    02.11    ST THREAD    stBm Mi,ri,size       Store in mem "size" bytes addressed
                                                at ri from buffer Mi
    02.12    EX THREAD    stEOD Mi              Send an "End Of Data" to buffer Mi of memory
                                                 =*=*=*=*=*=*=*=
Thread instruction format
=*=*=*=*=*=*=*=*=*=*=*=*=
comment : Similar to Alpha Branch instruction
          Manage memory threads on processors
 31              26 25         21 20                                      0
-----------------------------------------------------------------------------
|    OPCODE (6)     |  Function(5)|    Branch displacement (mem thread code) |
-----------------------------------------------------------------------------
opcode.function
    03.01    EX THREAD    ldth label       Fork a "load thread"
    03.02    LD THREAD    ldthe            End of a "load thread"
    03.11    EX THREAD    stth label       Fork a "store thread"
    03.12    ST THREAD    stthe            End of a "store thread"
                                                 =*=*=*=*=*=*=*=
Branch instructions
=*=*=*=*=*=*=*=*=*=
comment : use branch instruction format
 31              26 25         21 20                                     0
-----------------------------------------------------------------------------
|    OPCODE (6)     |   Buffer (5)|                                          |
-----------------------------------------------------------------------------
opcode
   06        EX/ST THREAD    boed [D|M]i            Branch on End of data
   07        EX/ST THREAD    bod  [D|M]i            Branch on data
                                                 =*=*=*=*=*=*=*=
Buffer Operate instructions
=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*
comments: *  ALPHA operate instructions having OPCODE 10 and 11 respectively are mapped
             to opcode 04 and 05, the first destination is a data buffer
           * For each instruction of function x, function x+1 is added for  the same instruction
         but the data are not removed from the buffer.
       ex : 10.20   addq r0,r2,r3   ==>
         EX THREAD           04.20      dpaddq  D0,r2,r3   //pointer point to next data
                             04.21      daddq D0,r2,r3     //pointer unchanged

         EX THREAD           04.23      addqD r1,r2,Mi       Send to M-buffers of memory

Floating point operate format
=*=*=*=*=*=*=*=*=*=*=*=*=*=*=
    14.028    EX THREAD    Dtoft Di,fc        move from buffer to FP reg (T FP)
    14.008    EX THREAD    Dtofs Di,fc             move from buffer to FP reg (S FP)
    1C.80     EX THREAD    ftoDt fi,Mi             move from FP reg to buffer in mem
    1C.88     EX THREAD    ftoDs fi,Mi             move from FP reg to buffer in mem
```

Figure A.3: *ALPHA instruction set extension for the DSCM architecture.*

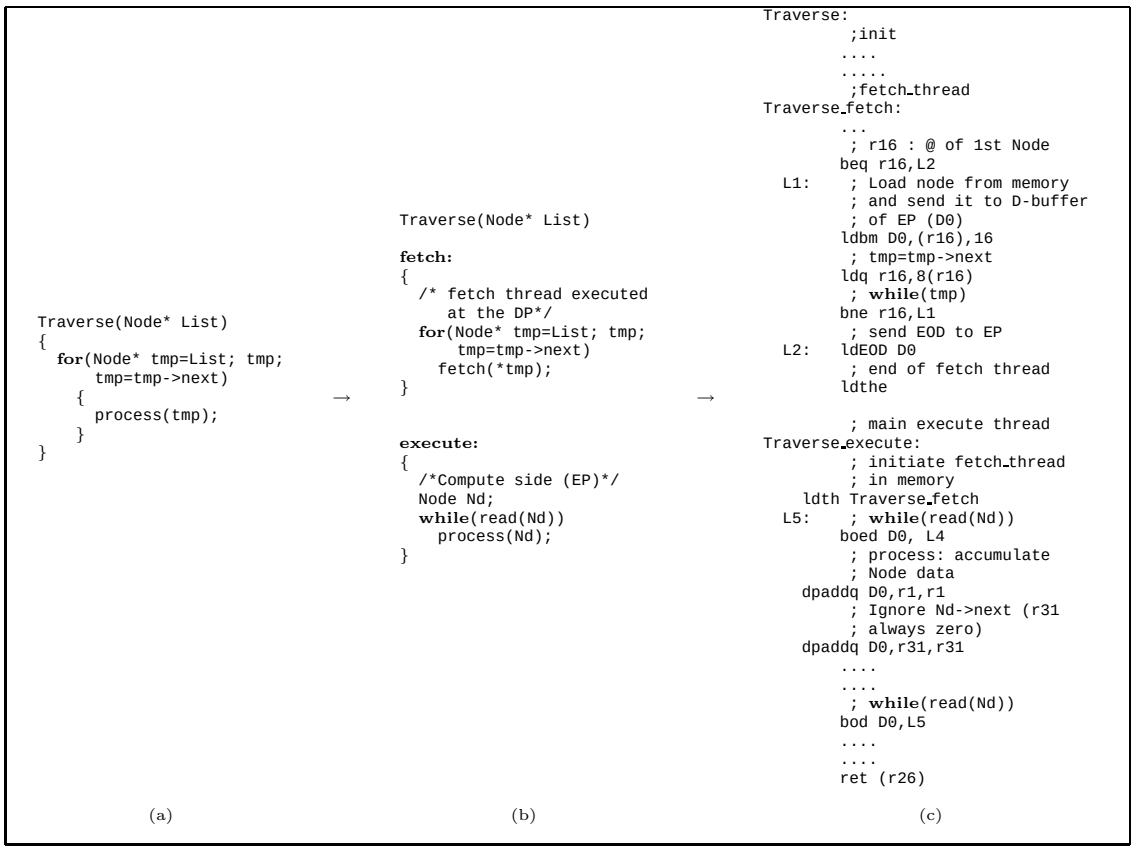


Figure A.4: *List traversal:(a) C (b) Decoupled C (c) Assembly code.*

instruction branches to the beginning of the `while` loop until an EOD is sent in the D0 buffer. On the other hand, the `boed` instruction branches when an EOD is read in the D-buffer.

D-Operate instructions. D-operate instructions are instructions that directly operate on data in the D-buffers or push results in the M-buffers. Again, in the example of Figure A.4, assuming that the `process()` function sums the data in the nodes `(S+=Nd->data)`, the instruction `dpaddq D0,r1,r1` adds the data in the D-buffer `D0` to the register `r1` and removes the data from the buffer.

Thread instructions. The `ldth` and `stth` initiate respectively load and store threads in the DP, while `ldthe` and `stthe` terminate them. In Figure A.4(c) the `ldth Traverse_fetch` instruction signals the fetch thread to execute the thread located at address `Traverse_fetch`. The `ldthe` instruction in the fetch thread terminates the thread.

A.4 Methodology and Experimental Results

We modified the out-of-order processor simulator (`sim-outorder`) of the SimpleScalar toolset [Bur96] to allow multithreaded processing and we added a processor at the memory level to evaluate the DSCM architecture. The simulator operates in two modes, the `DSCM onchip` mode, which simulates the single-processor DSCM architecture, and the `DSCM IRAM` mode which simulates the intelligent-memory architecture. Both the EP and DP are four-way superscalar processors with 96-instruction windows in each. The EP is multithreaded in the `DSCM onchip`, the DP also is multithreaded to allow execution of both `fetch` and `store` threads. We used a 16K 4 way L1 cache with 1-cycle latency, and a 256K 4 way L2 cache with 6-cycle latency. The processor (EP) to memory latency is fixed to 200 cycles.

DP configuration. As suggested in [Pat97], the processor to memory latency when the processor is physically located in the DRAM logic may be from 5X to 10X lower. Therefore, we evaluated the performance assuming EP to memory latency 20 cycles (`lat20`) and 40 cycles (`lat40`). Also, because the DRAM logic is slower than the EP logic, we evaluated the performance assuming no slowdown (`sd1`) and a 2X factor slowdown (`sd2`).

Evaluated microbenchs. We hand-coded the list-traversal example shown in Figure A.4 as well as the matrix multiplication loop shown in Figure A.5. This later example illustrates the use of a third thread, the *store* thread which is responsible

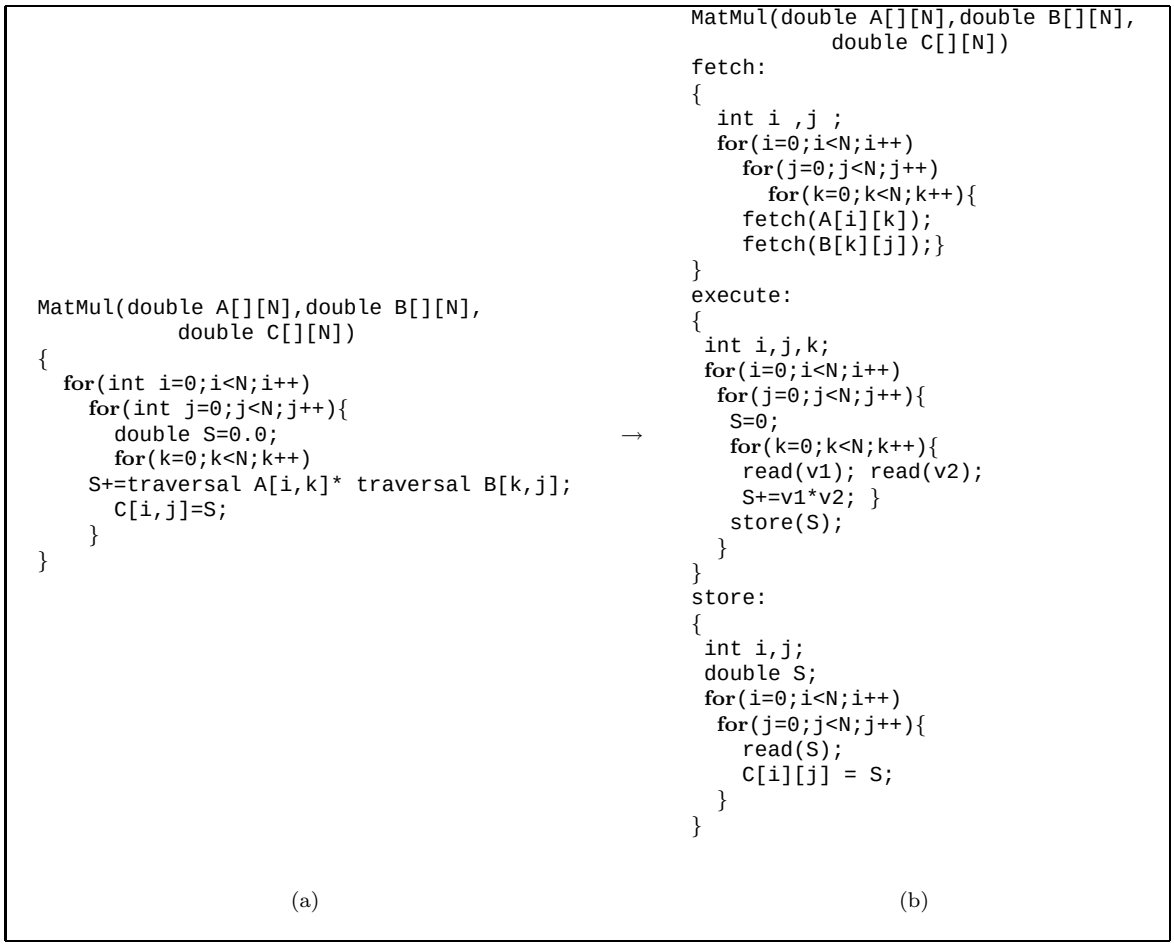


Figure A.5: *Matrix multiplication in Decoupled C.*

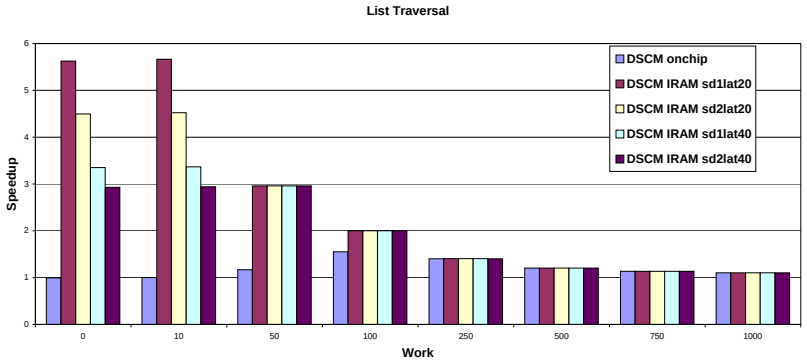


Figure A.6: *List traversal speedup for different value of overlapped work (W).*

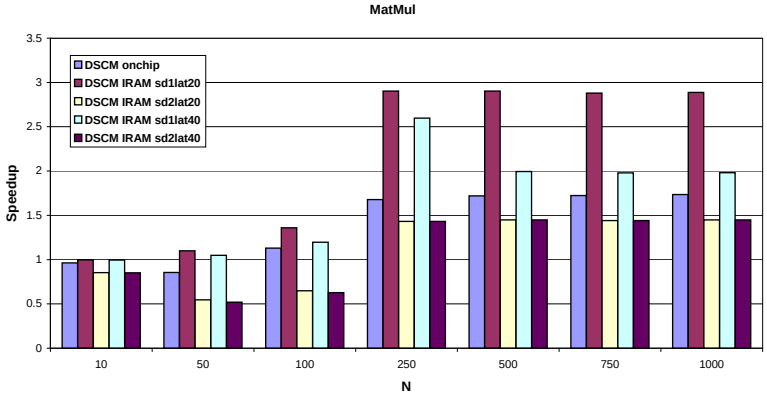


Figure A.7: *Matrix multiplication speedup for different sizes (N).*

for storing results in memory through the `store()` function. This thread allows the user to explicit more parallelism by streaming results back to the memory.

To study the effect of overlapping memory accesses with calculation, we replaced the `process` function in the list traversal example of Figure A.4 with a dummy loop that sums the values from 1 to W, where W is the amount of work to overlap. We evaluated the matrix multiplication loop for matrices of different sizes. Figure A.6 shows the speedup achieved on the list traversal for different values of overlapping work (W). The column `onchip` shows the speedups we obtain when all the threads are executed on the EP (no DP). The speedup results show that we can achieve substantial speedups (more than 5X) when the amount of work to overlap is small. Figure A.7 shows the speedup achieved on a matrix multiplication kernel, we notice a slow down for small matrices because the communication between the data processor and the execution Processor is longer than cache accesses. However for large matrices that miss in the cache, more than a 2X speedup can be acchieved

Still an effort is required from the user to explicit the partitioning between memory accesses and computations. However, unlike program optimizations, which require a detailed understanding of complex processor architectures, our approach only requires that the user be aware of the data structures he uses. We are investigating higher-level language extensions to render this task even more effortless.

Bibliography

[Aga00] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. In *Proceedings of the 27th annual international symposium on Computer architecture*, pp. 248–259. ACM Press, 2000.

[alp98] Alpha architecture handbook, October 1998. Compaq Computer Corporation.

[AMD00] AMD. *3DNow! Technology Manual*. Advanced Micro Devices, Inc, 2000.

[Ath93] P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *Computer*, vol. 26(3):pp. 11–18, 1993.

[Bar99] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: a compiler-managed memory system for raw machines. In *Proceedings of the 26th annual international symposium on Computer architecture*, pp. 4–15. IEEE Computer Society, 1999.

[Bar01] R. Barua, W. Lee, S. Amarasinghe, and A. Agarawal. Compiler support for scalable and efficient memory systems. *IEEE Trans Comput*, vol. 50(11):pp. 1234–1247, 2001.

[Bek99] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser. Correlated load-address predictors. In *Proceedings of the 26th annual international symposium on Computer architecture*, pp. 54–63. IEEE Computer Society, 1999.

[Bir93] P. L. Bird, A. Rawsthorne, and N. P. Topham. The effectiveness of decoupling. In *Proceedings of the 7th international conference on Supercomputing*, pp. 47–56. ACM Press, 1993.

[Bur96] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar tool set. Tech. Rep. CS-TR-1996-1308, 1996.

[Bur04] D. Burger and J. R. Goodman. Billion-transistor architectures: There and back again. *Computer*, vol. 37(3):pp. 22–28, 2004.

[Cal00] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The garp architecture and c compiler. *Computer*, vol. 33(4):pp. 62–69, 2000.

[Car99] J. B. Carter, W. C. Hsieh, L. Stoller, M. R. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *HPCA*, pp. 70–79. 1999.

[Car01] J. E. Carrillo and P. Chow. The effect of reconfigurable units in superscalar processors. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pp. 141–150. ACM Press, 2001.

[Cho00] Y. Chou, P. Pillai, H. Schmit, and J. P. Shen. PipeRench implementation of the instruction path coprocessor. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pp. 147–158. ACM Press, 2000.

[Cla03] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture*. 2003.

[Col02] J. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pp. 62–73. IEEE Computer Society Press, 2002.

[Com02] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (CSUR)*, vol. 34(2):pp. 171–210, 2002.

[Coo02] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pp. 279–290. ACM Press, 2002.

[DeH94] A. DeHon. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 31–39. Napa, CA, Apr 1994.

[DeH99] A. DeHon and J. Wawrzynek. Reconfigurable computing: what, why, and implications for design automation. In *Proceedings of the 36th ACM/IEEE conference on Design automation conference*, pp. 610–615. ACM Press, 1999.

[DeH00] A. DeHon. The density advantage of configurable computing. *Computer*, vol. 33(4):pp. 41–49, 2000.

[Des02] R. Desikan, C. R. Lefurgy, S. W. Keckler, and D. Burger. On-chip MRAM as a high-bandwidth, low-latency replacement for DRAM physical memories. Tech. Rep. TR-02-47, Department of Computer Sciences, The University of Texas at Austin, September 2002.

[Ebe96] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD - reconfigurable pipelined datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pp. 126–135. Springer-Verlag, 1996.

[Ebe03] C. Ebeling. RaPiD-C manual. Tech. rep., Department of Computer Science and Engineering, University of Washington, 2003.

[Fah01] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta. Performance characterization of a hardware framework for dynamic optimization. In *Proceedings of the 34th annual IEEE/ACM international symposium on Microarchitecture*. ACM Press, December 2001.

[Far98] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pp. 59–68. IEEE Computer Society Press, 1998.

[Fis02] D. Fischer, J. Teich, M. Thies, and R. Weper. Efficient architecture/compiler co-exploration for ASIPs. In *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems*, pp. 27–34. ACM Press, 2002.

[Fri98] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: dynamic optimizations for trace cache microprocessors. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pp. 173–181. IEEE Computer Society Press, 1998.

[Gol99] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. PipeRench: a coprocessor for streaming multimedia acceleration. In *Published in proceedings of the 26th International Symposium on Computer Architecture ISCA 99*, pp. 28–39. Atlanta, GA, 1999.

[Gol00] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor. PipeRench: A reconfigurable architecture and compiler. *IEEE Computer*, vol. 33(4):pp. 70–77, April 2000.

[Gol01] S. C. Goldstein and M. Budiu. Nanofabrics: Spatial computing using molecular electronics. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. june 2001.

[Goo85] J. R. Goodman, J. tu Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young. Pipe: a vlsi decoupled architecture. In *Proceedings of the 12th annual international symposium on Computer architecture*, pp. 20–27. IEEE Computer Society Press, 1985.

[Gut01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*. December 2001.

[Hau97a] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. In K. L. Pocek and J. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 87–96. IEEE Computer Society Press, 1997.

[Hau97b] J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings FCCM*, pp. 24–33. April 1997.

[Hau98] S. Hauck. The roles of FPGAs in reprogrammable systems. *Proceedings of the IEEE*, vol. 86(4):pp. 615–638, April 1998.

[Hau00] S. Hauck, M. Hosler, and T. Fry. High performance carry chains for FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8(2), April 2000.

[Hen00] J. Henning. Spec cpu2000: measuring cpu performance in the new millennium. *IEEE Computer*, vol. 33(7):pp. 28–35, 2000.

[Hin01] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, (Q1), 2001.

[Int] Intel Corp. Intel Itanium 2 Processor Reference Manual.

[Jac99] J. A. Jacob and P. Chow. Memory interfacing and instruction specification for reconfigurable processors. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pp. 145–154. ACM Press, 1999.

[Jon97] G. P. Jones and N. P. Topham. A comparison of data prefetching on an access decoupled and superscalar machine. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pp. 65–70. IEEE Computer Society, 1997.

[Jou89] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pp. 272–282. ACM Press, 1989.

[Kan99] Y. Kang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, and J. Torellas. Flexram: Toward an advanced intelligent memory system. In *Proceedings of the 1999 IEEE International Conference on Computer Design*, p. 192. IEEE Computer Society, 1999.

[Kar00] M. Karlsson, F. Dahlgren, and P. Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of the The Sixth International Symposium on High-Performance Computer Architecture (HPCA'6)*, pp. 206–217. IEEE Computer Society, 2000.

[Kes99] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, vol. 19(2):pp. 24–36, 1999.

[Koz02] C. Kozyrakis and D. Patterson. Vector vs. superscalar and vliw architectures for embedded multimedia benchmarks. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pp. 283–293. IEEE Computer Society Press, 2002.

[Lee98] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pp. 46–57. ACM Press, 1998.

[Lip95] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. Spaid: software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pp. 231–236. IEEE Computer Society Press, 1995.

[Luk96] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pp. 222–233. ACM Press, 1996.

[Mah92] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th annual international symposium on Microarchitecture*, pp. 45–54. IEEE Computer Society Press, 1992.

[Mai00] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: a modular reconfigurable architecture. In *Proceedings of the 27th annual international symposium on Computer architecture*, pp. 161–171. ACM Press, 2000.

[McF93] S. McFarling. Combining branch predictors. Technical Note TN-36, Digital WRL, june 1993.

[Mot99] Motorola, Inc. *AltiVec Technology Programming Interface Manual*. Motorola, Inc, 1999.

[MS97] W. H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. K. Prasanna, and H. A. E. Spaanenburg. Seeking solutions in configurable computing. *Computer*, vol. 30(12):pp. 38–43, 1997.

[Nag01] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pp. 40–51. IEEE Computer Society, 2001.

[Osk98] M. Oskin, F. T. Chong, and T. Sherwood. Active pages: a computation model for intelligent memory. In *Proceedings of the 25th annual international symposium on Computer architecture*, pp. 192–203. IEEE Computer Society, 1998.

[Osk99] M. Oskin, J. Hensley, D. Keen, F. T. Chong, M. Farrens, and A. Chopra. Exploiting ilp in page-based intelligent memory. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pp. 208–218. IEEE Computer Society, 1999.

[Par01] J.-M. Parcerisa and A. Gonzalez. Improving latency tolerance of multithreading through decoupling. *IEEE Trans Comput*, vol. 50(10):pp. 1084–1094, 2001.

[Pat90] D. A. Patterson and J. L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., 1990.

[Pat97] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent ram. *IEEE Micro*, vol. 17(2):pp. 34–44, 1997.

[Pat98] S. J. Patel, M. Evers, and Y. N. Patt. Improving trace cache effectiveness with branch promotion and trace packing. In *Proceedings of the 25th annual international symposium on Computer architecture*, pp. 262–271. IEEE Press, 1998.

[Pat00] S. J. Patel, T. Tung, S. Bose, and M. M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. In *Proceedings of the 33rd annual IEEE/ACM international symposium on Microarchitecture*, pp. 303–313. ACM Press, 2000.

[Pat01] S. J. Patel and S. S. Lumetta. rePLay: A hardware framework for dynamic optimization. *IEEE Transactions on Computers*, vol. 50(6), June 2001.

[Phi94] J. Phillips and S. Vassiliadis. High-performance 3-1 interlock collapsing ALU's. *IEEE Transactions on Computers*, vol. 43(3), March 1994.

[Ple86] A. R. Pleszkun and M. J. Thazhuthaveetil. An architecture for efficient lisp list access. In *Proceedings of the 13th annual international symposium on Computer architecture*, pp. 191–198. IEEE Computer Society Press, 1986.

[Raz] R. Razdan. *PRISC: Programmable Reduced Instruction Set Computers*. Ph.d. thesis, Harvard University, Division of Applied Sciences.

[Raz94] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pp. 172–180. ACM Press, 1994.

[Ric01] K. D. Rich and M. K. Farrens. Code partitioning in decoupled compilers. *Lecture Notes in Computer Science*, vol. 1900, 2001.

[Ro03] W. W. Ro, J.-L. Gaudiot, S. P. Crago, and A. M. Despain. Hidisc: A decoupled architecture for data-intensive applications. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, p. 3.2. IEEE Computer Society, 2003.

[Rog95] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 17(2):pp. 233–263, 1995.

[Ros93] J. Rose, A. E. Gamal, and A. Sangiovanni-Vincentelli. Architecture of field-programmable gate arrays. *Proceedings of the IEEE*, vol. 81(7):pp. 1013–1029, July 1993.

[Rot98] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pp. 115–126. ACM Press, 1998.

[Rot99] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th annual international symposium on Computer architecture*, pp. 111–121. IEEE Computer Society, 1999.

[Rot00] A. Roth, C. B. Zilles, and G. S. Sohi. Micro-architectural miss/execute decoupling. In *MEDEA Workshop*. october 2000.

[Rot01] A. Roth and G. S. Sohi. Speculative data-driven multithreading. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01)*, p. 37. IEEE Computer Society, 2001.

[San03] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th annual international symposium on Computer architecture*, pp. 422–433. ACM Press, 2003.

[Saz96] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The performance potential of data dependence speculation & collapsing. In *Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture*, pp. 238–247. IEEE Computer Society Press, 1996.

[Smi81] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pp. 135–148. IEEE Computer Society Press, 1981.

[Smi82] J. E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th annual symposium on Computer Architecture*, pp. 112–119. IEEE Computer Society Press, 1982.

[Smi84] J. E. Smith. Decoupled access/execute computer architectures. *ACM Trans Comput Syst*, vol. 2(4):pp. 289–308, 1984.

[Smi87] J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, and C. M. Rozewski. The zs-1 central processor. In *Proceedings of the second international conference on Architectual support for programming languages and operating systems*, pp. 199–204. IEEE Computer Society Press, 1987.

[Soh90] G. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, vol. 39(3), March 1990.

[Sol02] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th annual international symposium on Computer architecture*, pp. 171–182. IEEE Computer Society, 2002.

[Tay02] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, vol. 22(2):pp. 25–35, 2002.

[Tay03] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar operand networks: On-chip interconnect for ilp in partitioned architectures. In *Proceedings of the The Ninth International Symposium on High-Performance Computer Architecture (HPCA'03)*, p. 341. IEEE Computer Society, 2003.

[Tay04] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman,

V. Strumpen, M. Frank, S. Amarasinghe, , and A. Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *Proceedings of the 31st annual international symposium on Computer architecture*. IEEE Computer Society, 2004.

[Tes01] R. Tessier and W. Burleson. Reconfigurable computing for digital signal processing: A survey. *J VLSI Signal Process Syst*, vol. 28(1-2):pp. 7–27, 2001.

[Tul95] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pp. 392–403. ACM Press, 1995.

[Wai97] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *Computer*, vol. 30(9):pp. 86–93, 1997.

[Wal91] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pp. 176–188. ACM Press, 1991.

[Wir95] M. J. Wirthlin. A dynamic instruction set computer. In *Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*, p. 99. IEEE Computer Society, 1995.

[Wit96] R. Wittig and P. Chow. OneChip: An FPGA processor with reconfigurable logic. In K. L. Pocek and J. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 126–135. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[Wul92] W. A. Wulf. Evaluation of the wm architecture. In *Proceedings of the 19th annual international symposium on Computer architecture*, pp. 382–390. ACM Press, 1992.

[Wul95] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput Archit News*, vol. 23(1):pp. 20–24, 1995.

[xil02] Virtex-II pro platform FPGAs: Functional description. Tech. Rep. DS083-2, January 2002. Xilinx Corporation.

[Yan00] C.-L. Yang and A. R. Lebeck. Push vs. pull: data movement for linked data structures. In *Proceedings of the 14th international conference on Supercomputing*, pp. 176–186. ACM Press, 2000.

[Ye00a] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 225–235. IEEE Computer Society and ACM SIGARCH, Vancouver, British Columbia, Jun 12–14, 2000.

[Ye00b] Z. A. Ye, N. Shenoy, and P. Baneijee. A C compiler for a processor with a reconfigurable functional unit. In *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pp. 95–100. ACM Press, 2000.

[Yeh91] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pp. 51–61. ACM Press, 1991.

[Zha98] Y. Zhang and G. B. Adams, III. Performance modeling and code partitioning for the ds architecture. In *Proceedings of the 25th annual international symposium on Computer architecture*, pp. 293–304. IEEE Computer Society, 1998.

[Zie01] M. Ziegler and M. Stan. Optimal logarithmic adder structures with a fanout of two for minimizing the area-delay product. In *Proceedings of the the International Symposium on Circuits and Systems*. May 2001.