Revisiting Wide Superscalar Microarchitecture

Andrea Mondelli

▶ To cite this version:

Andrea Mondelli. Revisiting Wide Superscalar Microarchitecture. Hardware Architecture [cs.AR]. Université de Rennes 1, 2017. English. NNT : . tel-01597752v1

HAL Id: tel-01597752
https://inria.hal.science/tel-01597752v1
Submitted on 28 Sep 2017 (v1), last revised 18 Jan 2018 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ANNÉE 2017

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

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

Andrea Mondelli

préparée à l'unité de recherche INRIA
Institut National de Recherche en Informatique et Automatique
Université de Rennes 1

Revisiting Wide Superscalar Microarchitecture

Thèse soutenue à Rennes
le 12 Septembre 2017
devant le jury composé de :

Steven Derrien
Professeur à l'Université de Rennes 1 / Président

Bernard Goossens
Professeur à l'Université de Perpignan / rapporteur

Smail Niar
Professeur à l'Université de Valenciennes/rapporteur

Karine Heydemann
Maître de conference / examinateur

André Seznec
Directeur de recherche, INRIA Rennes / directeur de thèse

Pierre Michaud
Chargé de recherche, INRIA Rennes / co-directeur de thèse

Dear Mary,
do you [know who] will be on the boats? I'm still in Gaza, waiting for you. I will be at the boat to greet you. **Stay human**. Vik.

Vittorio Arrigoni

Remerciements

Contents

Résumé en Français **5**

1 Introduction **9**

1.1 Purpose of this work . . . 9
1.2 Contributions . . . 10
1.3 Organization . . . 12

2 State of the Art **13**

2.1 From Pipeline to Superscalar . . . 13
2.2 Performance Technique of Superscalar Processors . . . 16
  2.2.1 Instruction Level Parallelism . . . 16
  2.2.2 The Branch Predictor . . . 18
  2.2.3 The Register Renaming . . . 18
  2.2.4 Out-of-Order Execution . . . 20
2.3 Wide-issue complexity . . . 22
  2.3.1 Limits of performance scaling . . . 22
  2.3.2 Impact of critical components . . . 24
  2.3.3 Front-end bandwidth . . . 25
  2.3.4 Level-one data cache . . . 25
  2.3.5 Bypass network paths . . . 26
  2.3.6 Load/Store queues . . . 27
  2.3.7 Reduce the complexity by limiting pipeline activity . . . 29
2.4 Clustering . . . 29
  2.4.1 Clustered microarchitectures . . . 30
  2.4.2 Issue Buffer . . . 31
  2.4.3 Steering . . . 32
  2.4.4 Write Specialization . . . 35
  2.4.5 Bypass network and intercluster delay . . . 37
  2.4.6 Clustering in Commercial Superscalar Processors . . . 37
  2.4.7 Clustered VLIW/DSP architectures . . . 39
2.5 Energy saving exploiting loops . . . 40

2.5.1 Saving loop energy in the front-end 40
2.5.2 Using a dedicated cache for loops 41
2.5.3 Saving loop energy in the back-end 44
2.5.4 Loop accelerators 45
2.5.5 Industrial adoption of loop cache solutions 46

3 Wide Issue Clustered Microarchitecture 47
3.1 A case of increasing single-thread IPC 47
3.2 Experimental Framework 49
3.2.1 Simulation Setup 49
3.2.2 Benchmarks 50
3.2.3 Baseline Microarchitecture 51
3.3 Potential IPC gains from a more complex superscalar microarchitecture 54
3.4 Dual-Clustered Configurations 55
3.4.1 Dual-Cluster with baseline instruction window size 57
3.4.2 Dual-cluster with double instruction window 57
3.4.3 Analysis 60
3.4.4 Possible steps toward the proposed dual-cluster configuration 62
3.5 Energy Considerations 63
3.5.1 Static EPI 63
3.5.2 Gating intercluster communications for reduced dynamic EPI 64
3.6 Summary 65

4 Exploiting loops for reducing energy in a superscalar out-of-order core 67
4.1 Baseline and Experimental Setup 68
4.2 Loop Buffer and Loop Detector 69
4.2.1 Description 70
4.2.2 Loop buffer size 73
4.2.3 Tuning MinIter 74
4.3 Redundant Micro-Op Removal 76
4.3.1 Proposed mechanism 76
4.3.2 Identification of redundant micro-ops 77
4.3.3 Modification of register renaming 78
4.3.4 Loads and stores 79
4.3.5 Simulation Results 80
4.3.6 Compiler Optimization impact: a case study 82
4.4 Reducing the energy of loads 86
4.4.1 Speculative load execution 86
4.4.2 DL1/STQ gating? 88

4.4.3 Store Queue and DL1 gating in Loop Mode 90
4.4.4 STQ gating alone 91
4.4.5 DL1 gating alone 91
4.4.6 STQ gating and DL1 gating combined 92
4.4.7 Gating the memory dependence predictor table 94
4.5 Summary 95

5 Conclusion **97**
5.1 The superscalar architecture of the future 97
5.2 Exploiting loops for power consumption 99
5.3 Perspectives 99

Bibliography **103**

Author's Publications **118**

List of Acronyms **123**

List of Figures **125**

Résumé en Français

Depuis plusieurs décennies, la fréquence des processeurs à usage général n'a cessé d'augmenter grâce aux transistors de plus en plus rapides et aux micro-architectures avec des pipelines plus profonds. Cependant il y a environ 10 ans, à cause des courants de fuite et de la température, la finesse de gravure des processeurs a atteint sa limite physique. Depuis lors, la fréquence des processeurs n'a pas augmenté. Au lieu d'augmenter la fréquence du processeur, les fabricants ont intégré plus de cœurs sur une seule puce, agrandi la hiérarchie de caches et amélioré l'efficacité énergétique.

Mettre plus de cœurs sur une seule puce a augmenté le rendement de la puce et bénéficie aux applications parallèles. Cependant, avoir plus de cœurs ne suffit pas. Il est également important d'accélérer les processeurs individuellement.

En outre, la réduction de la consommation énergétique est devenue un objectif majeur lors de la conception d'une micro-architecture pour la haute performance. Certaines fonctionnalités ont été introduites dans les unités superscalaires principalement pour réduire la consommation énergétique. Un exemple de fonctionnalité est le tampon de boucles ("loop buffer"), qui est maintenant mis en œuvre dans plusieurs micro-architectures superscalaires. Le but d'un tampon de boucle est d'économiser l'énergie dans le bloc avant du microprocesseur (cache d'instructions, prédicteur de branchements, décodeur, etc.) lors de l'exécution d'une boucle avec un corps assez petit pour tenir dans cette mémoire tampon spécifique.

Pendant l'exécution d'une boucle, les instructions décodées sont fournies au bloc arrière directement depuis le tampon de boucle: le cache d'instructions, les tables de prédiction de branchement, et la plupart de la logique du bloc avant peuvent être en mode horloge fermée (clock gated), ce qui réduit la consommation énergétique de la partie avant.

Si la fréquence du processeur reste constante, la seule possibilité laissée libre pour l'amélioration des performances des applications séquentielles dans les futurs processeurs est d'augmenter l'exploitation du parallélisme d'instructions (ILP). Certaines améliorations des micro-architectures (e.g., une meilleure prédiction de branchement) améliorent simultanément la performance et l'efficacité énergétique. Cependant, améliorer l'exploitation du parallélisme d'instructions a généralement un coût: augmentation de la surface de silicium, de la consommation d'énergie, des efforts de conception, etc. Par conséquent, la micro-architecture est modifiée lentement, incrément par incrément.

En effet, les fabricants de processeurs ont fait des efforts continus afin d'exploiter davantage l' ILP avec de meilleurs prédicteurs de branchements, de meilleurs préchargeurs de données, de plus grandes fenêtres d'instructions, ajout de registres physiques, et ainsi de suite. Par exemple, la micro-architecture Intel Nehalem

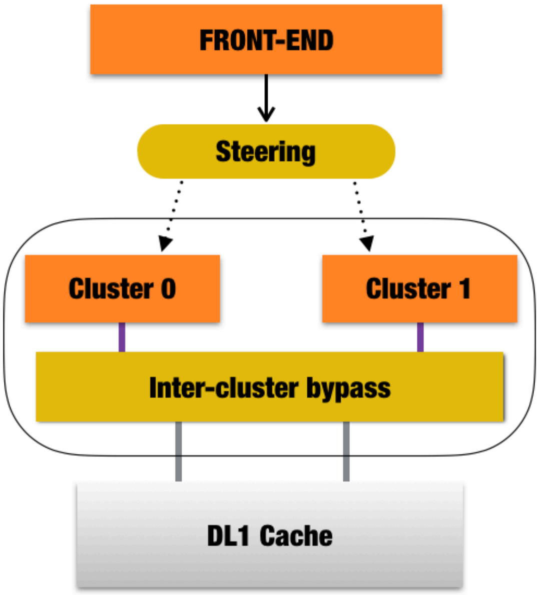


Figure 1: Exemple d'architecture en grappe.

peut émettre 6 micro-opérations par cycle à partir d'un tampon d'exécution (issue buffer) de 36 entrées, tandis que les plus récentes micro-architectures Intel Haswell peuvent émettre 8 micro-opérations par cycle à partir d'un tampon de 60 entrées.

Cette thèse décrit ce que devraient être les unités superscalaires dans les 10 ans à venir et explore la possibilité d'exploiter le comportement des boucles afin de réduire la consommation énergétique au-delà du bloc avant. Certaines propositions ont été publiées notamment sur les accélérateurs de boucles et sur les unités superscalaires à bloc arrière non conventionnel. Il est soutenu que la taille de la fenêtre d'instructions peut être augmentée en combinant le regroupement (clustering) et la spécialisation des registres d'écriture (register write specialization).

Une différence majeure avec les précédentes études sur les micro-architectures en grappe (Figure 1) est l'utilisation de grappes larges (wide issue clusters), contrairement aux études passées qui étaient principalement axées sur des petites grappes (narrow issue cluster). Le passage de petites grappes à des grappes larges n'est pas qu'un changement quantitatif, mais a aussi un impact qualitatif sur le problème de regroupement, et en particulier sur la politique de pilotage (steering policy). De précédentes études sur les politiques de pilotage ont montré que, tout en réalisant un bon équilibrage de charge, la réduction des communications inter-groupe est un problème difficile. L'une des conclusions d'une décennie de travaux sur les politiques de pilotage est que les plus simples tel que Mod-3 génèrent une perte significative du nombre d'instructions par cycle (IPC), alors que les politiques de pilotage qui minimisent la perte IPC sont trop complexes pour les implémentations matérielles.

De facto, cette étude montre que considérer des grappes larges a un impact dramatique sur les performances de Mod-N, l'une des politiques de pilotage les plus simples. Mod-N envoie N instructions consécutives à une grappe, puis les N prochaines à une autre grappe, et ainsi de suite de façon circulaire. Baniasadi et Moshovos ont constaté que, pour les petites grappes, la valeur optimale de N est en général très faible et préconisent une politique Mod-3. Il semblerait qu'après l'article de Baniasadi et Moshovos, personne n'ait considéré la politique Mod-N autre que Mod-3.

Force est de constater qu'avec des grappes larges, si la fenêtre d'instructions est assez grande et compte tenu d'un délai inter-groupe réaliste, la valeur optimale de N est beaucoup plus grande que trois, généralement plusieurs dizaines. Grâce à la localité des données dépendantes, une politique Mod-64 conduit à beaucoup moins de communications inter-grappe qu'une politique Mod-3. En conséquence, Mod-64 tolère plus les retards inter-grappe que Mod-3. En outre, environ 40% des valeurs produites par une grappe n'ont pas besoin d'être transmises à une autre grappe, ce qui permet de réduire l'énergie dépensée dans les communications inter-grappe.

La seconde contribution propose deux optimisations indépendantes et orthogonales concernant la consommation énergétique et exploitant les boucles. La première optimisation détecte les micro-opérations redondantes produisant le même résultat à chaque itération puis supprime définitivement ces micro-opérations. La seconde optimisation se concentre sur la diminution de l'énergie consommée des micro-opérations de chargement (*load*), en détectant les situations où un chargement n'a pas besoin d'accéder à la file d'attente des enregistrements (store queue) ou n'a pas besoin d'accéder au cache de données de niveau 1 (DL1).

Les optimisations proposées ont un impact négligeable sur la performance de calcul, elles concernent essentiellement la consommation énergétique.

Chapter 1

Introduction

1.1 Purpose of this work

For several decades, the clock frequency of general purpose processors was growing thanks to faster transistors and microarchitectures with deeper pipelines. However, about ten years ago, technology hit leakage power and temperature walls. Since then, the clock frequency of high-end processors did not increase. Instead of increasing the clock frequency, processor makers integrated more cores on a single chip, enlarged the cache hierarchy and improved energy efficiency.

Putting more cores on a single chip has increased the total chip throughput and benefits some applications with thread-level parallelism. However, many applications have low thread-level parallelism [BDMF10]. So having more cores is not sufficient. It is important also to accelerate individual threads.

Moreover, limiting the energy consumption has become a major challenge when designing a high-performance microarchitecture. Some microarchitecture features have been introduced in superscalar cores mainly for reducing energy. An example of such feature is the *loop buffer* today implemented in several superscalar microarchitectures [Int16a, Lan11, Rup12]. The purpose of a loop buffer is to save energy in the core's front-end (instruction cache, branch predictor, decoder, etc.) when executing a loop with a body small enough to fit in the loop buffer. During the loop execution, decoded instructions are provided to the back-end directly from the loop buffer: the instruction cache, branch prediction tables, and most of the front-end logic can be clock gated, which reduces front-end power consumption[1].

If the clock frequency remains constant, the only possibility left for higher single-thread performance in future processors is to exploit more Instruction-level Parallelism (ILP). Certain microarchitecture improvements (e.g., better branch predictor) can simultaneously improve performance and energy efficiency. However, in general, exploiting more ILP has a cost in silicon area, energy consumption, design effort, etc. Therefore, nowadays the microarchitecture evolves slowly, incrementally, taking advantage of technology scaling. Processor makers have made continuous efforts to exploit more ILP, with better branch predictors, better data prefetchers, larger instruction windows, more physical registers, and so forth. For example, the Intel Nehalem microarchitecture (November 2008) can issue 6 micro-ops per cycle from a 36-entry issue buffer, while the more recent Intel Haswell microarchitecture (June 2013) can issue 8 micro-ops per cycle from a 60-entry issue buffer [Int16a].

---

[1] A recent white paper by NVIDIA mentions a 50% reduction of the front-end power in the ARM Cortex-A15, thanks to the loop buffer [NVI13]

1.2 Contributions

The first contribution of this thesis is an exploration of the possible design of a very wide issue superscalar processor (16-issue processor). Such a processor could be a core of the high-end multicores ten years from now. A major difference with past research on clustered microarchitecture is that we assume wide issue clusters ($\leq$ 8-issue), whereas previous research mostly focused on narrow issue clusters ($\leq$ 4-issue). Going from narrow issue to wide issue clusters is not just a quantitative change, it has a qualitative impact on the clustering problem, in particular on the steering policy. Past research on steering policies showed that minimizing intercluster communications while achieving good cluster load balancing is a difficult challenge.

One of the conclusions of a decade of research on steering policies was that simple steering policies generate significant performance opportunity loss while steering policies minimizing performance loss are too complex for hardware-only implementations [BM00, SZ05, CCGG08].

This study shows that considering wide issue instead of narrow issue clusters has a dramatic impact on the performance of *Mod-N*, one of the simplest steering policy. Mod-N sends N consecutive instructions to a cluster, the next N instructions to another cluster, and so forth in round-robin fashion [BM00]. Baniasadi and Moshovos found that, on narrow issue clusters, the optimal value of N is generally very small, advocating a Mod-3 policy. To the best of our knowledge, after Baniasadi and Moshovos's paper, nobody has considered Mod-N policies other than Mod-3.

We find that, with wide issue clusters, if the instruction window is large enough and considering a realistic intercluster delay, the optimal value of N is much larger than three, typically several tens. Owing to data-dependence locality, a Mod-64 policy leads to much fewer intercluster communications than a Mod-3 policy. As a result, Mod-64 tolerates greater intercluster delays than Mod-3. Moreover, about

40% of the values produced by a cluster do not need to be forwarded to the other cluster, which permits reducing the energy spent in intercluster communications.

We argue that the instruction window and the issue width can be augmented by combining clustering [LFK+93, Kes99, PJS97] and register write specialization [CPG00, ZK01, STR02].

Many programs spend a significant part of the execution in loops [Kob84].The second contribution is an exploration of the possibility of exploiting loop behaviors to reduce energy consumption beyond the front-end. Some propositions have been published for loop accelerators [CHM08, SIT+14, NGS15] or for unconventional superscalar core back-ends [HNL14]. Sodani and Sohi, in [SS97], pointed out that redundant execution exists in most programs. In this thesis, we propose two independent and orthogonal energy optimizations exploiting loops.

The first optimization we propose is a mechanism to detect redundant micro-ops producing the same result on every iteration and to remove these micro-ops from the execution core. The simplest case of redundancy is when a micro-op reads all its operands from registers not modified in the loop. Registers remain unchanged when values are set before entering the loop. A micro-op that only reads these constant values can be considered redundant, and the produced value is constant through the different iterations. In this way, the *redundancy status* can spread to other micro-ops, and the loop buffer can be manipulated to remove these micro-ops.

We propose the modification of register renaming to keep alive the physical register that holds the result of a redundant micro-op, and we describe a solution to recover the register file in case of trap or loop exit. The proposed modification uses a small fully associative table called Redundant Load Table to keep track of conflicts between stores and removed loads. We evaluate the impact of removing redundant micro-ops and the cost of hardware modifications.

The second optimization focuses on saving energy consumed by load micro-ops. We detect situations where a load does not need to access the store queue or does not need to access the Level-1 data cache (DL1). In the conventional case, every load checks the store queue (STQ) and the DL1 simultaneously, but one of these two accesses is useless. We predict in which queue the load will hit the data, to avoid the unnecessary accesses.

The proposed methods allow the gating of STQ or DL1. These methods can be used together (they are orthogonal) or independently of each other. We describe the effects of wrong DL1/STQ gating prediction and the opportunities that tested benchmarks offer for each type of gating.

We also test this mechanism with a configuration which emulates what happens for very large instruction footprint applications. We first analyze each type of gating individually, then combined, focusing on the performance impact of this energy saving solution. The optimizations proposed have a negligible impact on performance but they mostly generate energy consumption saving.

1.3 Organization

This thesis is organized as follows. Chapter 2 presents a summary of the state of the art related to the two main contributions of this thesis. It describes what was done in the last two decades for increasing back-end performances with clustered micro-architecture and decreasing the power consumption using loops.

The first contribution is described in Chapter 3. We argue for using some of the benefits of technology scaling for increasing single-thread Instruction per cycle (IPC). We explore the performance impact of various microarchitecture parameters, and we show that, by doubling all the parameters of a modern high-end core simultaneously, the IPC of `SPEC INT` benchmarks could be increased by 25% on average, that of `SPEC FP` by 40%. We focus on the performance impact of clustering the out-of-order engine, finding that significant performance gains are still possible, despite the intercluster delay, provided the instruction window parameters are doubled. We show that a Mod-64 steering policy tolerates intercluster delays of a few cycles. We explain how the energy consumption of intercluster communications can be reduced by detecting micro-ops whose results are not needed by the other cluster.

The second contribution is described in Chapter 4. In this section, some statistics about the loop behavior of benchmarks are provided. Then we describe and evaluate the two energy optimizations proposed: the removal of redundant micro-ops, and the DL1/STQ gating. We offer an overview of how the two gating technique can be combined.

Chapter 5 concludes this thesis, with considerations about the future of superscalar architecture using the clustered back-end and about future uses of the dynamic instruction reuse technique for power consumption reduction.

Chapter 2

State of the Art

Microprocessors are the hearts of most of the electronic systems we use, today. Not only computers, smartphones, tablets, but also cars, dishwasher, TV and so on. In Computer Science, the architecture of microprocessors has been evolving from the first microprocessor[1] to the recent Intel/AMD microprocessors. Today the technology allows the creation of tiny microprocessors capable of executing several times more instructions than supercomputers 20 years ago. A common (and most of the time, commercial) way to evaluate this capability is the clock speed. This value, expressed in Hertz, was a kind of keyword in past commercials. The reality is quite different: clock speed and processor's performance are different, and they express different processor capabilities.

The large use of microprocessors in smart and portable devices like laptops or smartphones opened the eyes on another important feature of modern processors as well as performance: the energy consumption.

2.1 From Pipeline to Superscalar

Design and work with microarchitectures means conceiving new solutions for the *datapath*. The datapath can be defined as the path that the instructions must fulfill within the processor to finish their execution. Different instructions may need different datapath, and an instruction generally requires more execution time if it is to fulfill a longer path within the datapath, and then passes through a larger number of logic gates.

A fundamental concept in the study of the microarchitecture is the Instruction Set Architecture (ISA). The ISA describes those aspects of the architecture design that are visible to the programmer. It is the set of basic instructions that the microprocessor can perform. Two microprocessors that use the same ISA can have two completely different microarchitectures, and consequently different ways to execute instructions and different execution times. In the rest of the thesis, the words microarchitecture and architecture will be used interchangeably.

---

[1] The term "microprocessor" is attributed to Viatron Computer Systems describing the custom integrated circuit used in their System 21 small computer system announced in 1968

The scalar pipeline architecture is an implementation that assumes one instruction is executed at each clock cycle and each component can be used only by one instruction per cycle. This is one of the simplest architecture to design, but its speed is limited by the longest path, by the long delay of memory compared to the registers and by the slower component. The evolution from pipeline to superscalar allowed to increase performances increasing the hardware complexity.

The pipelining allows the partial overlap of instructions, thus trying to exploit their potential inherent parallelism, also called Instruction-level Parallelism (ILP). The actual degree of instruction parallelism that it is possible to exploit, however, depends on several factors, like the limited number of available functional units, the data dependencies between instructions and the presence of branch instructions.

In an ideal situation, we have two solutions to increase performance of the pipeline: increase the clock frequency and execute more instructions in parallel.

If we increase the clock frequency (which means using a shorter clock cycle), the clock must be sized to allow the instruction to pass through the portion of datapath content in a single pipeline stage. So that the pipeline can continue to operate, it is necessary to redesign the architecture, dividing the work in a greater number of stages, in each of which it is possible to perform the least amount of work permitted by the new shorter clock cycle.

However, if the number of pipeline stages (also called *depth of the pipeline*) is increased, there will be more instructions running in parallel, and therefore the parallelism inherent in the running program is potentially fully exploited. In other words, the number of pipeline stages and the clock frequency (and the length of the clock cycle) are closely related to each other impacting the actual performance.

The increase of the depth of pipeline was heavily exploited by the Pentium IV, in which the architects were able to reach frequencies that touched the 4 GHz with a pipeline of nearly 30 stages [HSU$^+$01]. Unfortunately, it was not possible to exploit this technique indefinitely, because of architectural and technological problems. The increased complexity of the pipeline and thus of its control unit, the thickness of the link between transistors, the interferences, energy consumption and heat dissipation are all typical problems of a deep pipeline [AHKB00].

Executing more instructions in parallel, called *multiple issue* execution, means executing two or more independent instructions, i.e., they do not need the result produced by any other instruction executed in the same clock cycle. The multiple issue execution requires sophisticated functional units, and a sufficient number of

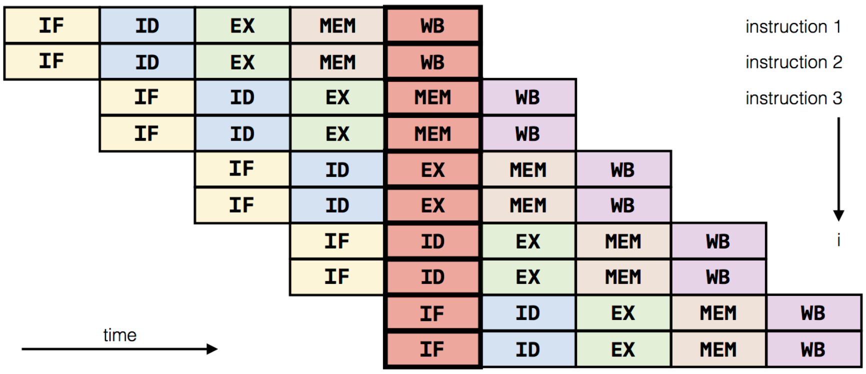


Figure 2.1: The basic outline of a superscalar execution for instruction from 1 to *i* and more. In the real implementation, the pipeline stages are more than 5, but the execution of the instructions can still be conceptually divided into few basic phases. In this example two instructions can use the same pipeline phases under ideal conditions, and we have CPI=0.5

functional units must be available to execute multiple instructions in parallel. It must be possible to fetch multiple instructions from the instruction memory and multiple operands from the data memory at each clock cycle. It must be possible to address more CPU registers in parallel, and it must be possible to read and write registers used by instructions executed in the same clock cycle.

In a pipelined architecture without multiple issue execution, in an ideal case of absence of dependencies, one instruction is completed at every clock cycle: the Cycle per Instruction (CPI) is equal to 1. In a multi-issue pipelined architecture, instead, it is virtually possible to complete the execution of more than one instruction per clock cycle, having a CPI less than 1 (Figure 2.1).

In order to implement the multiple issues, the logic unit must be able to determine which and how many instructions it is possible to execute in a given clock cycle. Once identified these instructions, they are put in an *issue packet* and issued in the same clock cycle. The multi-issue processors can be conceptually divided into two categories, depending on how (and especially when) this identification is carried out.

In static multi-issue processors, the selection of which instructions can be issued in parallel is made by the compiler, that is in software. When this job is entirely done by the compiler, we call it Static Instruction-level Parallelism.

In dynamic multi-issue processors, the processor resolves at run-time the dependencies and decides which instructions execute in parallel. Obviously, there is a limit to the number of instructions that the logic unit can examine to identify as an independent group of instructions. This limit depends on the architecture implementation and on the fact that the search is done at run-time and the processor has a short time window (the clock cycle) to detect them. The number of identified independent instructions may change at each clock cycle (up to the maximum allowed by the architecture). This latter solution is called Dynamic Instruction-level Parallelism.

2.2 Performance Technique of Superscalar Processors

2.2.1 Instruction Level Parallelism

Determining the instructions dependencies is essential to quantifying how much parallelism exists in a program and how it can be exploited. If two instructions are independent, they can be executed simultaneously and in any order in the pipeline, so long as there are sufficient resources (i.e., functional units). If two instructions are dependent, they must be executed in order, and they can overlap only partially. Two dependent instructions can stay in the pipeline at the same time but in “sufficiently distant” phases. In order to exploit the parallelism offered by programs, it is essential to determine the dependencies between instructions.

If we consider two instructions, i and j, with i appearing before j, we can have tree kinds of data hazard:

Read-After-Write (RAW) Instruction j tries to read a register or memory location before it is written by i. The instruction j would read the old value that is incorrect. In this case, j cannot continue the execution because the data j needs is not yet available. The processor must notice the dependence of j from i and suspend its execution until the missing data is available. This is also called true dependency.

Write-After-Write (WAW) Instruction j tries to write a register or memory location before it is written by i. The two writing end in the wrong order, leaving the register with a value written by i instead of j. This is an output dependency.

Write-After-Read (WAR) Instruction j tries to write a register or memory location before i reads it. The instruction i reads a wrong value. This is an anti-dependency, which rarely occurs because in most of the pipeline the reading of the operands usually takes place “very before” (the Instruction Decode (ID) phase) the actual write (in the Write-Back (WB) phase).

The dynamic pipeline scheduling consists of a set of run-time techniques to reduce the frequency and the duration of the hazards, like changing the order in which the instructions are executed or renaming the registers.

The Out-of-Order execution makes it possible to maximize the use of computing resources by executing instructions according to their availability. For example, stalling the instruction fetch on the occurrence of any single branch would result in very poor performance, therefore, the processor uses prediction to predict the outcome of branches. This technique is called speculation, and can be applied at compile time or can be hardware based; we will focus on the latter. Speculation techniques require specific recovery mechanisms in case of wrong predictions.

Several speculation techniques [CR00, SVS96, Smi81, SMR05, LWS96] have been proposed in the literature and implemented in commercial architectures. Branch prediction consists in predicting the execution flow direction in the presence of branch instructions. A high percentage of instructions are branches, and a “good” branch predictor, trying to predict both the direction of the branch and the target address of a jump, can give important benefits in terms of performance and power saving.

Speculation can also be applied to the memory accesses (*Data Prefetching*), trying to predict the data that will be needed soon, loading it into caches before it is actually required. Data Prefetching is useful to improve tolerance to the high memory latency, and it can be implemented either via software or hardware. Software prefetching is typically implemented using the compiler [MLG92, AS79, McI98].

The hardware prefetcher tries to exploit spatial locality making available, in the cache, the block or blocks containing the data that will be needed soon. This prefetching can be done through identification of specific memory access patterns or through use of special threads usually called *helper threads* [SKS14, KLW$^+$04, LDH$^+$05, KST11]. The downsides of an aggressive data prefetching are the risk of polluting the cache with useless data and wasting bandwidth.

Another type of speculation is the *Load-Hit* speculation. It consists in predicting that the load instruction will hit the first level cache, issuing instructions following the load early. In the case of a misprediction, however, the load and the instructions scheduled after the load need to be rescheduled. Load-Hit speculation allows instructions that depend on a load to benefit from the possibility of issuing early, but it requires a mechanism for re-issuing these instructions in case of cache miss predicted as hit.

Load and store queues allow executing memory instructions keeping the correct order of memory accesses. Each load must wait until all prior store addresses are calculated. However, it is possible to predict the dependency between load and store and anticipate the load issue. Load speculation allows the early execution of loads, and it postpones the check of correctness to subsequent pipeline stages. In the case of a mispeculation, it is necessary to perform a roll-back[2], discarding the load and all instructions issued after the load [CR00].

2.2.2 The Branch Predictor

The branch prediction was introduced in order to eliminate control hazards. Programs are not linear sequences of instructions, but they are full of branch-like operations.

A branch predictor requires hardware structures that keep track of program execution and branch instructions history like previously taken direction and target addresses of previous jumps. The branch predictor tables are read during the fetch phase, and the Program Counter (PC) of the branch instruction is used to query these tables in order to choose the next instruction to fetch. There are numerous implementations of the branch predictor. Recent branch predictors, like TAGE [SM06], achieve a high prediction accuracy.

2.2.3 The Register Renaming

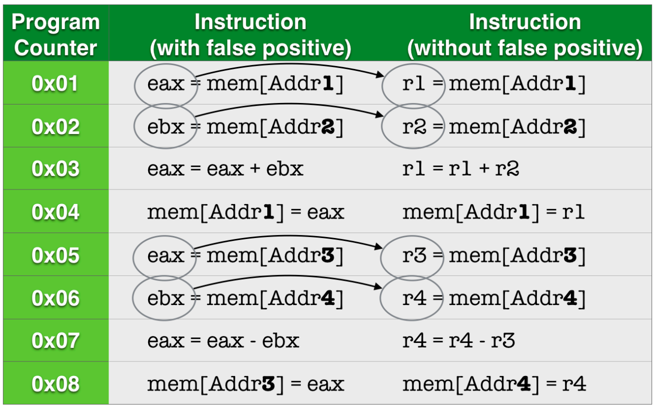
An essential component of modern Out-of-Order architectures is the register renaming. The architecture features a limited number of logical registers. This limit implies the existence of false register dependencies, as illustrated in the example in Figure 2.2. The ability of the system to exploit the instruction level parallelism is limited by false dependencies between instructions which use the same logical registers but work on independent data. It is possible, however, to have a higher number of physical registers in the system. During the register renaming stage, each logical register is renamed and assigned to an available physical register. The association between logical and physical registers is kept in a specific table. This solution tries to eliminate false register dependencies while maintaining true data dependencies between instructions.

The first example of register renaming, proposed in 1967, was Tomasulo's algorithm [Tom67]. This algorithm is based on the use of a Register Allocation Table (RAT) to keep track of renamed registers. In modern architectures, the information about which physical register to release is kept in the Reorder Buffer (ROB) and used when the instruction is committed. The renaming uses a table that contains a list of the actual free physical register, and if there are no free registers, the front-end stalls.

The major key of the Tomasulo's scheme is the Reservation Station (RS) associated with the functional units, where the instructions, after the instruction fetch and decode, wait to be executed.

---

[2] The roll-back mechanism could be different on different architectures



Program Counter	Instruction (with false positive)	Instruction (without false positive)
0x01	eax = mem[Addr**1**]	r1 = mem[Addr**1**]
0x02	ebx = mem[Addr**2**]	r2 = mem[Addr**2**]
0x03	eax = eax + ebx	r1 = r1 + r2
0x04	mem[Addr**1**] = eax	mem[Addr**1**] = r1
0x05	eax = mem[Addr**3**]	r3 = mem[Addr**3**]
0x06	ebx = mem[Addr**4**]	r4 = mem[Addr**4**]
0x07	eax = eax - ebx	r4 = r4 - r3
0x08	mem[Addr**3**] = eax	mem[Addr**4**] = r4

Figure 2.2: Instructions from `0x05` to `0x08` are independent of instructions from `0x01` to `0x04`, but the processor cannot finish the load at `0x05` and `0x06` until the register `eax` is written, otherwise the value written at `Addr1` will be wrong. With the rename of architectural registers to physical registers, the false dependence disappears.

The Reservation Stations and the load/store buffer consist of multiple fields, implemented by internal registers (invisible to the ISA) that allow storing the information required to handle the entire mechanism.

The Tomasulo scheme has two essential characteristics:

1. The access to the operands is performed in a distributed manner
2. The WAW and WAR hazards are eliminated

If the CPU fetches a new instruction at each clock cycle, the use of Reservation Station allows to have multiple consecutive instructions running simultaneously, and the RS themselves act as additional registers.

Several variants of Tomasulo's original algorithm were proposed, like adding the Register Update Unit [SV87] or additional structures [Smi98], in order to maintain and restore the precise microarchitectural state. In the solution proposed by Smith and Pleszkun [Smi98] the instructions are kept in the ROB until the in-order commit. The Tomasulo's scheme with an in-order commit is shown in Figure 2.3.

The Tomasulo algorithm and its variant with the Reorder Buffer have been used in commercial architecture like Intel P6 [Col05, Gwe95]. The Intel P6 introduced, in the Intel Family, the Speculative Execution and the Out-of-Order (OoO) mechanism, called *dynamic execution* by Intel.

An overview of register renaming techniques was described by Sima in his survey [Sim00].

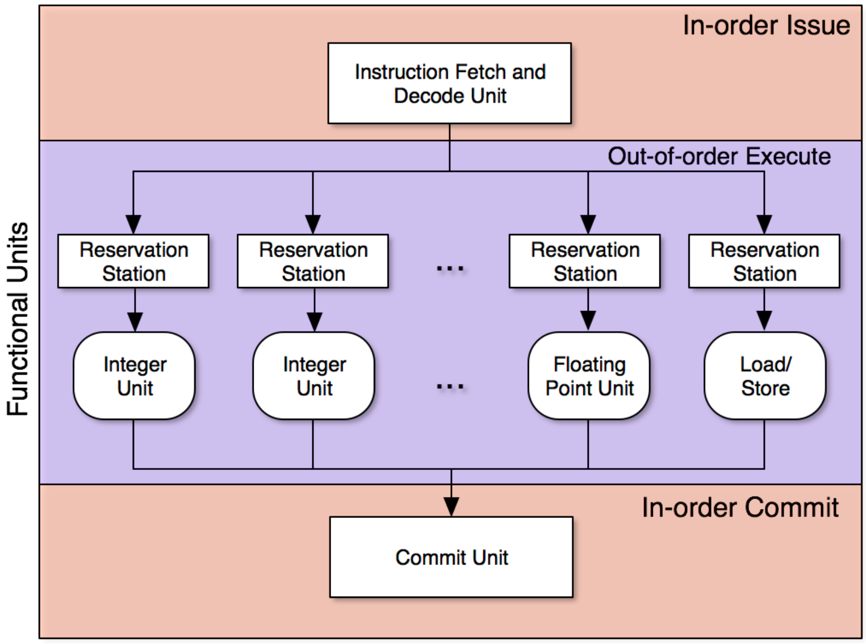

In-order Issue
Instruction Fetch and Decode Unit
Out-of-order Execute
Functional Units
Reservation Station
Reservation Station
...
Reservation Station
Reservation Station
Integer Unit
Integer Unit
...
Floating Point Unit
Load/ Store
In-order Commit
Commit Unit

Figure 2.3: A simplifier overview of Tomasulo's Reservation Stations associated with Functional Units. Instructions are fetched and decoded in order, and sent to RS according to their type, thus the type of functional unit required. Using a Reorder Buffer, it is possible to have an in-order commit that helps the handling of precise interrupts.

In this thesis, we propose a register renaming modification in order to support the loop mechanism proposed in Chapter 4.

2.2.4 Out-of-Order Execution

The evolution of single core processors has allowed exploiting the instruction level parallelism through the introduction of Out-of-Order (OoO) execution. The OoO execution allows to schedule instructions which are ready to be executed without order restrictions, allowing to reduce the impact of chip connection latency and the underlying memory latency. The use of OoO implies the introduction of new hardware components, like the ROB, the issue queues, the load/store queues, and so on. These new components lead to an increase in complexity, compared to in-order processors.

In order to understand the ideas proposed in this thesis, it is necessary to give a brief summary of the main OoO structures present in modern microarchitectures, briefly describing how they work and in which way they can be modified.

All the instructions in the Out-of-Order back-end are handled by the ROB. The ROB is the First-In-First-Out (FIFO) queue we described in previous sections. Instructions are read from the instruction cache, then decoded, and inserted in the ROB in an ordered sequence. At the end of the execution, each instruction is removed from the ROB in the same order in which it was inserted. In this way, the order of execution is guaranteed for all committed instructions. The instructions can be executed out of order, depending on source data and hardware resource availability.

The instruction dependencies are guaranteed by the use of an *Issue Queue*. This queue contains the instructions ready to be executed, and it is used by the scheduler to decide the issue sequence. The total number of instructions in the issue queue is a subset of the instructions in the ROB. When an instruction is executed, it may produce a result that is used by another instruction in the issue queue.

Each time an instruction is executed, all the ready instructions in the issue queue are informed about the availability of the result and, eventually, can become ready. The *ready* status depends on the availability of required data input and the availability of the Functional Unit (FU) required to execute the instruction. The issue queue can be implemented as a single queue or divided into multiple queues. The Alpha 21264 [Kes99], for example, uses an integer issue queue and a floating-point issue queue. In a clustered architecture, described later, it is possible to have separated issue queues for each cluster.

The dispatch of instructions from the issue queue to Functional Units, in the OoO engine, is based on two steps: the *Wake-Up* and *Select*, as mentioned in previous section. In a given clock cycle, there are instructions in the Issue Queue waiting to be executed. These instructions wait for the availability of their input operands. The input operand of a given instruction is ready when the producer of this operand is executed, and the operand become available. When operands are available, the instruction in the Issue Queue is flagged as *ready* and could be chosen in the next Select step for the execution.

Since the hardware resources are limited (i.e., there are not enough available functions units), a dynamic selection of instructions to execute is made. The dynamic selection happens during the *Select step.* The selection of the ready instructions to dispatch is based on a specific heuristic; the common one is to choose the oldest ready instruction in the Instruction Window.

When the ready instruction is selected, its tag is broadcasted to the Issue Queue to inform other instructions about the future availability of the result. This information allows, in the next Wake-Up step, to flag new instructions as ready to execute. From now on, these instruction become *ready* and they can be selected in the Select step in the next cycle.

Speculation is possible also in the Wake-Up phase. For load instructions, for example, the scheduler cannot determine in advance how many cycles will be necessary to complete the memory access. The load can miss in memory, and the time needed to complete the execution can grow from a few cycles to hundred of cycles. When a load is selected and executed, it is possible to speculate about its execution latency and execute dependency instruction according to this speculation. If the load misses in Level-1 Data Cache (DL1), we have a *misspeculation*, and a recovery mechanism is necessary.

The *Load Speculation* was proposed to resolve this delay issue for memory operations [FS96, MBVS97, YERJ99] and different solutions were proposed in the literature, i.e., the Store Set [CE98].

The memory instructions (load and store) use two specific structures called Load Queue (LDQ) and Store Queue (STQ), a Content-Addressable-Memory (CAM) structures used to identify dependencies between load and store instructions. Their purpose is to maintain the correct memory access order and verify dependencies between memory addresses. Load/Store units are generally maintained as two separate queues.

2.3 Wide-issue complexity

The introduction of the out-of-order execution in superscalar architectures has introduced a further degree of complexity. Indeed it is not sufficient to increase the total number of Functional Units in order to have a multiple-issue execution. The Instruction Window size limits the number of instruction that can be potentially selected by the scheduler at each cycle. The superscalar processor implementation requires changes at all stages of the pipeline. It is necessary to carry out, each cycle, the fetch, the decode and rename of multiple instructions. Also, the register file and the cache memory require modifications to allow the access of multiple instructions per cycle.

2.3.1 Limits of performance scaling

Exploiting Instruction-level Parallelism is an efficient way to increase the performance of microarchitectures. A way to evaluate the performance is counting the number of instructions executed per cycle (IPC). The IPC is equivalent to performance when the clock frequency is fixed, for a given binary.

Performance can also be evaluated in terms of raw execution speed. In the past years, a common way to increase the processor performance (and the commercial attractiveness) was the clock frequency. Increasing the frequency requires taking into account physical limits, like internal wire delay and the delay of individual hardware components. In the last ten years, the semiconductor fabrication node of companies like Intel moved from the 65 nm of Intel Conroe/Merom to 14 nm of Skylake. Transistors are getting smaller, but not faster, and the wire connections between them are becoming a major problem due to delay [AHKB00]. It is possible to reduce the clock period by increasing the number of pipeline stages, but it may increase the overhead of the latches between pipeline stages, as well as phenomena such as clock skew and jitter.

There is a technological problem for the scaling of the wire delay: the wire delay does not scale at the same pace as the gate delay. This problem is greater for global wires, as we need connections through an ever-increasing number of gates along technology. Along with the wire problem, energy consumption is another factor to consider. For example, the energy consumption increases with the size of the register file [PJS97, San06]. The area occupied by the register file increases linearly with the number of registers, but quadratically with the number of ports. The power consumption of the register file, then, increases quadratically compared to the processor width, since the number of ports required is proportional to the number of instructions which need to access the register file.

Palacharla studied the design complexity, defining it as “the delay through the critical path of a piece of logic, and the longest path through any of the pipeline stages determines the clock speed” [Pal98]. It is necessary to find new methods to increase performances and reduce the power consumption without increasing the complexity. The increase of computational capability usually results in an increase of the size of structures. This increase is reflected by the growth of design complexity and energy consumption. This trend can not go on any longer, since problems on wire delay, clock period size, etc., lead to a slowdown of performance improvements.

Many research works have focused the attention on the limits inherent to physical scalability [Bor99, Mat97]. These limits, for example, were put in evidence by Agarwal *et al.* [AHKB00]. They examined the effects of technology scaling on clock speed and wire delay on a hypothetical aggressive microprocessor, concluding that the scaling rate of performance cannot remain constant. In the deep sub-micron fabrication processes there is no future for large monolithic cores and the limits of technology constraints will become bigger in future designs. They proposed that future microprocessors should be partitioned into independent physical regions, with a focus on the latency for communicating among partitions.

In a superscalar architecture, performance scaling is not limited only by wire delay. As described by Palacharla *et al.* [PJS97], many of the elements in the pipeline present a complexity that requires further considerations. Palacharla *et al.* analyzed components such as the register rename logic, the wake-up logic, selection logic and data bypass logic, comparing three different technologies (0.8 μm, 0.35 μm and 0.18 μm). They proposed a new architectural paradigm that addresses the above problems: partitioning part of the architecture into smaller and simple units, called clusters, running at a high clock rate.

2.3.2 Impact of critical components

All these modifications have increased the architecture complexity. The trend, starting after the Pentium 4, is to keep the same clock frequency but increase the ability to exploiting the ILP, also increasing the wide-issue capabilities. As a consequence, we have an increase in hardware complexity, opening the door to new challenges for the research.

One of the most performance-critical components of superscalar architectures is the issue logic. Optimizing the issue logic makes it possible to maximize the amount of ILP that it is possible to exploit during the program executions. Increase the complexity permits to have wider microarchitectures, with instruction window wide enough to allow more in-fly instructions in the out-of-order engine, maximizing the resources usage. Having a big issue queue allows the issue logic to select the next instructions among a larger number of waked up instructions, allowing to minimize the number of functional units that, in a given clock cycle, are idle.

Increase the wide-issue requires more systems buses. For example, with N functional units, it is possible to generate N different results from N instructions being executed in the same clock cycle. These results will be compared with each entry of the Issue Queue, in order to select which instruction is possible to wake-up. To do this comparison, for each entry, we need $2 \times N$ comparators because of the two operators.

If the Issue Queue is W entries, a total of $2 \times N \times W$ comparison is required. Increasing N and W means increasing the total number of comparators, buses, and wires needed. A small increase of wide-issue involves adding much hardware. Also, increasing the complexity also increases the power consumption. In modern microarchitectures, power consumption has become a greater problem than in the past. In this thesis, we will try to address this issue proposing solutions to reduce the number of instructions executed in the OoO engine exploiting loops. The use of loop buffer will allow us to identify redundant instructions and even redundant memory access. We will see how exploiting loops helps to reduce power consumption.

Wide-issue architectures require a significant number of interconnection wires between components like FUs, buffers and so on. Due to the miniaturization, the wire-delay has become one of the dominant components of the total delay, also limiting the increase of clock cycle. In order to design a microarchitecture, it is necessary to take into account the trade-off between performance, clock cycle, and complexity [PJS97].

Different solutions have been proposed in the literature to try to reduce the complexity of one or more part of microarchitectures, such as for register rename [San06], register file [STR02], load/store queue [POV03], issue logic [CG01, TP08], dispatch queue [RG09] and so on.

2.3.3 Front-end bandwidth

Executing more instructions per cycle requires fetching more instructions per cycle. A possible way to increase the front-end bandwidth is to predict and fetch two basic blocks per cycles instead of one, as in the Alpha EV8 [SFKS02], and scale instruction decode accordingly. However, decode itself may be a microarchitecture bottleneck for Complex Instruction Set Computer (CISC) instructions sets such as Intel x86. A trace cache (aka decoded I-cache or micro-op cache in the Intel Sandy Bridge and Haswell microarchitectures [Int16a]) addresses this issue. A trace cache stores in traces micro-ops that are likely to be executed consecutively in sequential order [RBS96] (Figure 2.4).

It is possible to transform traces dynamically to increase performance and/or decrease energy consumption [FPP98]. In particular, Jacobson *et al.* proposed to perform constant propagation before storing a trace for future reuse [JS99].

A trace cache is also a solution to the register renaming bandwidth problem. When creating a trace, intra-trace dependencies can be determined, and this information can be stored along with the trace. The number of read and write ports of the rename table can be reduced: each read-after-write or read-after-read occurrence within a trace saves one read port on the rename table, and each write-after-write occurrence saves one write port [VM97]. The number of read and write ports of the rename table is part of the trace format definition.

2.3.4 Level-one data cache

Increasing the issue width also means increasing the L1 data cache load/store bandwidth. The need for sustainable cache bandwidth requires an exploration of new low-cost techniques in order to meet the requirement of modern microprocessors, paying attention also to the power consumption. Several solutions are possible for increasing the load/store bandwidth. The multi-porting solution allows N cache accesses per cycle using an N-ported cache in which each port is independent. This solution is expensive, and not used in commercial processors. The Alpha 21264 [Kes99] implements a time-divided multiplexer solution with a virtual multi-porting in which the cache run at twice the clock speed of the processor. This solution, unfortunately, does not scale well with the increase of issue width. Unlike the Alpha 21264, the Alpha 21164 [Cor94] implemented a

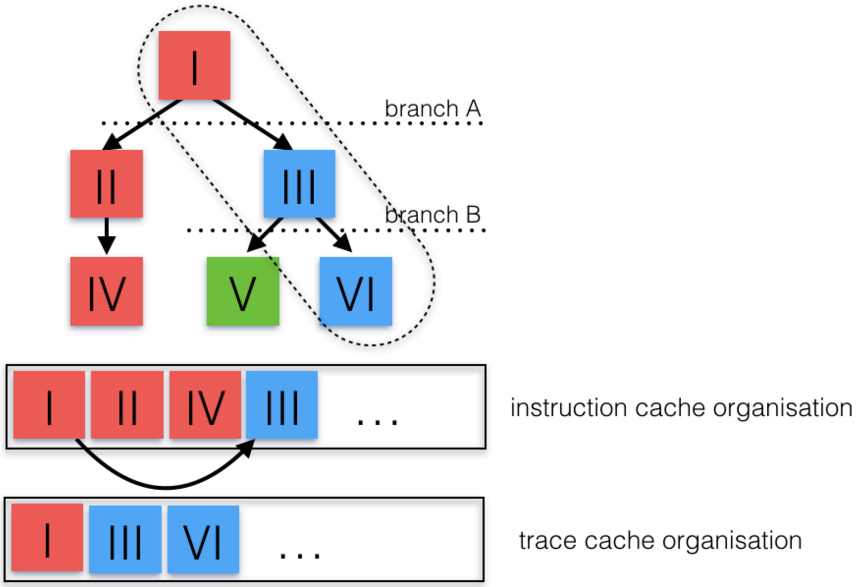


Figure 2.4: Difference between conventional cache and trace cache for the dynamic path: I - III - VI. In trace cache any instruction may appear multiple times because if can be part of different traces. In conventional cache the frequency of taken branches limits the maximum bandwidth per memory port.

dual-copy replication solution in which two copies of the cache are kept coherent. The consistency is permitted by writing the data on both cache copies on each store. The replication requires more die area and has limitations in scalability.

Recent high-end processors use banking to provide the adequate bandwidth. The first example of 2-bank interleaving data cache was proposed in the MIPS R10000 [Yea96], and extended in subsequent architectures. Moreover, alternative solutions based on interleaved caches [RTDA97], replication caches [AZ05], power-efficient caches [DB07] or balanced caches [RFD13], were proposed to resolve different problems in modern processors. Banking leads to the possibility of conflicts when several loads/stores issued simultaneously access the same bank.

2.3.5 Bypass network paths

When an instruction completes the execution, the result is written in the register file. A subsequent instruction that uses that value as input should wait to read the data from the register (Figure 2.5a). The bypass network in necessary to execute dependent instructions in consecutive cycles. In modern architecture, a broadcast-based bypass network is used to bypass result values produced by an instruction to subsequent consumers (Figure 2.5b). This bypass logic requires introducing more hardware in the datapath. The complexity introduced by the bypasses is proportional to the issue width, and to the number of pipeline stages (Figure 2.5c), i.e., a significant amount of wiring area is used on the chip [ACR95], and the wire delay is an increasing problem in technology era [fS13]. Moreover, the use of wide multiplexers and the number of long wires when we increase bypass width have a substantial impact on power consumption [TSR+98].

In a fully bypassed design, the number of bypass paths grows quadratically with the issue width. For example, considering an issue width W and a pipeline with N stages after the first stage in which a 2-input FU results become available, we need $2 \times N \times W^2$ bypass paths. The scaling of bypass delay is also affected by the reduction of feature size [PJS97], increasing the difficulty of implementing a single cycle bypass network in wide superscalar architectures.

2.3.6 Load/Store queues

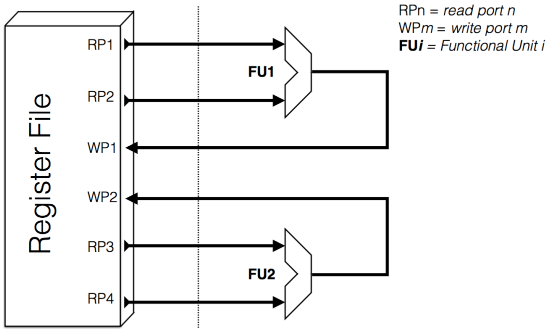
The load queue is for guaranteeing a correct execution while allowing loads to execute speculatively before older independent stores. The load queue is searched when a store retires. When there is a load queue, the store queue is mostly for preventing memory dependencies to hurt performance. The store queue is searched when a load executes.

Enlarging the instruction window to exploit more ILP may require the load/store queues to be enlarged too. However, conventional load/store queues are fully associative structures which cannot be enlarged straightforwardly. Clustering the OoO engine does not solve the load/store queues scalability problem. Specific solutions may be needed, e.g., [BZ06, CL04, SMR05, SL06]. For this study, we ignore the problem, and we assume that it is possible to enlarge load/store queues without impacting the clock cycle or the load latency.

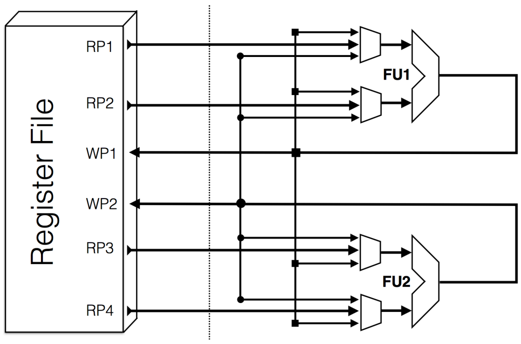
The reduction of structure accesses for energy saving is not a novel idea. A large number of loads miss in the STQ [SDB+03, POV03], and some researchers have tried to reduce the number of STQ accesses, for lower energy consumption. Sethumadhavan *et al.* consider filtering accesses to the load/store queues with Bloom filters [SDB+03].

With the Store Set technique, it is possible to create *sets* of conflicting memory instructions [CE98]. Park *et al.* [POV03] filter the STQ searches by modifying the Store Set memory dependence predictor [MBVS97] so that loads that are predicted to be independent of all the stores in the instruction window only access the L1 data cache, not the STQ.

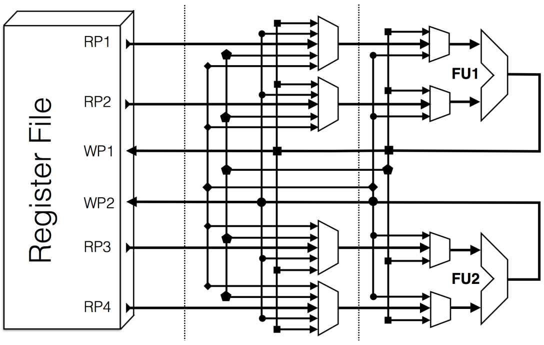
Observing that accessing the L1 data cache generally costs much more energy than accessing load/store queues, Nicolaescu *et al.* proposed the *cached LSQ*: they modify a unified load/store queue (LSQ) so that it behaves like an L0 data cache, with the L1 cache accessed only upon an LSQ miss [NVN03]. They assumed that


RPn = *read port n*
WP*m* = *write port m*
FU*i* = *Functional Unit i*
Register File
RP1
RP2
WP1
WP2
RP3
RP4
FU1
FU2

(a) Simplified scheme for a no-bypass design of execution engine


Register File
RP1
RP2
WP1
WP2
RP3
RP4
FU1
FU2

(b) Value-bypass implementation using multiplexers


Register File
RP1
RP2
WP1
WP2
RP3
RP4
FU1
FU2

(c) Value-bypass implementation for deep pipeline

Figure 2.5: Each functional unit is connected to read and write ports through latches. Increasing the pipeline depth, the data could be forwarded from different stages of the producer to different stages of the consumer. It requires more multiplexers and more wires.

the LSQ latency is three times shorter than the L1 cache latency. In order to avoid the performance penalty of serializing accesses to the cached LSQ and L1 cache, Carazo *et al.* proposed to predict L0 hits with a bimodal predictor and a Bloom filter: when a L0 hit is predicted, the LSQ and L1 cache are accessed in series, otherwise they are accessed in parallel [CAC$^+$10]. Like Nicolaescu *et al.*, Carazo *et al.* assume a 1-cycle latency for the LSQ.

2.3.7 Reduce the complexity by limiting pipeline activity

Due to the end of Dennard scaling [DGR$^+$74], the energy benefits obtained from new generations of transistor densities is reducing. To address this problem, it is necessary to explore solutions in which the transistor switching should be limited, along limiting the clock cycle, and optimize the existing design. Furthermore, the demand for computational efficiency is continuing to increase. These problems force architects to take into account new energy constraints and eliminate, whenever possible, unnecessary pipeline activity. For example, the front-end in modern architecture like ARM Cortex-A15, represents a predominant source of power consumption [Lan11], forcing to find new architectural solutions independent of technology improvement.

The implementation of dedicated memory for loops helps to save energy in front-end, reducing the impact of fetch, decode and rename. However, despite potential energy saving, no commercial out-of-order processor has attempted to exploit loops behavior to save energy in the back-end. As we will see, in the back-end and specifically in the out-of-order engine, a source of power consumption that we will try to eliminate is the access in parallel to the data cache and store buffer by load instructions. We will see that we can reduce these accesses introducing small hardware modification to support a mechanism of prediction for memory instructions.

2.4 Clustering

We can consider the baseline superscalar model shown in Figure 2.6. During the fetch phase, instructions are read from the instruction cache memory and decoded. In this phase, multiple instructions are delivered to the OoO execution engine. A predictor is used to fetch a continuous stream of good-path instructions. The main goal is to keep functional units busy as much as possible. Every instruction is converted by one or multiple decoders in RISC-like instructions (called *micro-ops* in Intel architecture) and register operands are renamed.

Significant IPC gains can be obtained by increasing the issue width, the front-end width, and the instruction window size.

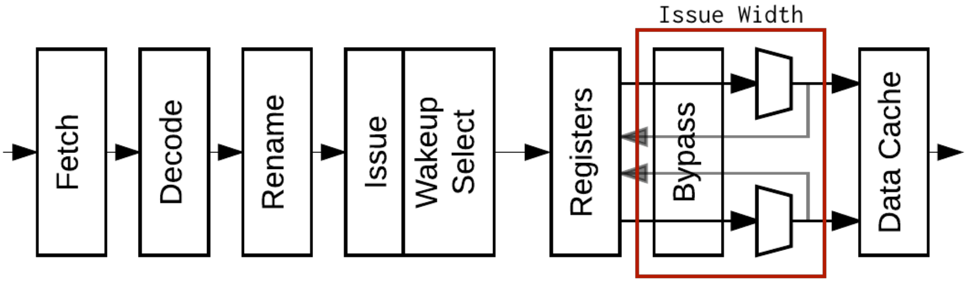


Figure 2.6: Simplified Superscalar pipeline. The Clustering allow to increase the issue width of the back-end.

In this section, we emphasize on clustering as a solution to increase the issue width without increasing the complexity of the microarchitecture.

2.4.1 Clustered microarchitectures

Clustering was proposed in the 1990s as a solution for reducing the clock cycle of superscalar processors [PJS97, FCJ97].

The basic idea is to partition the Execution Units (EUs) into clusters so that the output of one EU can be used as input by any other EU in the cluster in the next clock cycle through a local bypass network [GLM10].

In [PJS97] an architecture with the *dependence-based* steering policy was proposed. In this architecture, the issue window is replaced by simpler FIFO queue structures, in order to reduce the clock period without reducing the IPC. Based on the fact that dependent instructions execute sequentially, the idea is to steer[3] dependent instructions to the same queue, and to issue only the instructions at the head of each queue. The dependencies between instructions are evaluated using specific hardware tables that keep track of the assigned cluster for each register producer. The steering logic uses this table to understand the instruction dependency and, eventually, uses this information for deciding to which cluster to assign the instruction, following a specific heuristic. In their work, Palacharla *et al.* have evaluated the performance of the dependence-based architecture with 2x4-way clustered system[4]. They compared it with a conventional 8-way microarchitecture, founding the clustered solution allows a faster clock cycle than the conventional one, but with the cost of a small performance degradation.

Clustering can also be applied to the issue buffer (one issue buffer partition per cluster). Hence clustering is a solution to two of the most significant frequency bottlenecks in the OoO engine: the bypass network and the issue buffer. The price to pay is that communications between clusters require an extra delay, which may impact the performances.

---

[3] The steering is performed at run-time during the rename phase

[4] Two clusters, each of which contains four FIFOs and four functional units

A natural form of clustering, which has been used in several superscalar processors, is to have an integer cluster and a floating-point cluster. This form of clustering does not generate intercluster communications, and the term “clustered microarchitecture” is mostly applied to cases where intercluster communications are frequent.

Clustering introduces a degree of freedom for instructions that can execute on several clusters. Choosing on which cluster to execute an instruction is called steering. Microarchitectures such as the DEC Alpha 21264 [FF98, Kes99] that cluster the EUs but not the scheduler do the steering at instruction issue (execution-driven steering [PJS97]). Microarchitectures such as the IBM POWER7 [SKS$^+$11] that cluster both the EUs and the scheduler do the steering before inserting the instruction into the issue buffer (dispatch-driven steering [PJS97]). An example of dispatch-driven clustered scheme is shown in Figure 2.7.

Execution-driven steering does a good job at mitigating the impact of the intercluster delay, as the scheduler can send an instruction to the cluster where it can execute sooner [PJS97]. However, execution-driven steering limits the size of the issue buffer, not only because the issue buffer does not benefit from clustering (unlike the bypass network), but also because postponing the steering until issue makes it part of the scheduling loop, which impacts the clock frequency.

Zyuban and Kogge proposed clustering as a solution for decreasing the Energy per instruction (EPI) for a given IPC [ZK01]. Like us, they considered wide-issue clusters. However, they did not quantify the IPC improvements that clustering can provide under a fixed clock cycle.

The Integer-Decoupled architecture proposed by Palacharla [Pal98] is based on the necessity of using the floating-point subsystem resources during integer intensive code. The idea is to avoid the resource idling augmenting the floating-point units to perform simple integer operations. The benefits obtained during the execution of integer intensive workload compensates the little hardware complexity introduced in the Floating Point (FP) subsystem.

2.4.2 Issue Buffer

Clustering the issue buffer like the EUs (as in the IBM POWER7) permits increasing the total issue buffer capacity without impacting the clock cycle: the select operation is done independently on each issue buffer partition, and the wake-up operation is pipelined between the partitions, taking advantage of the

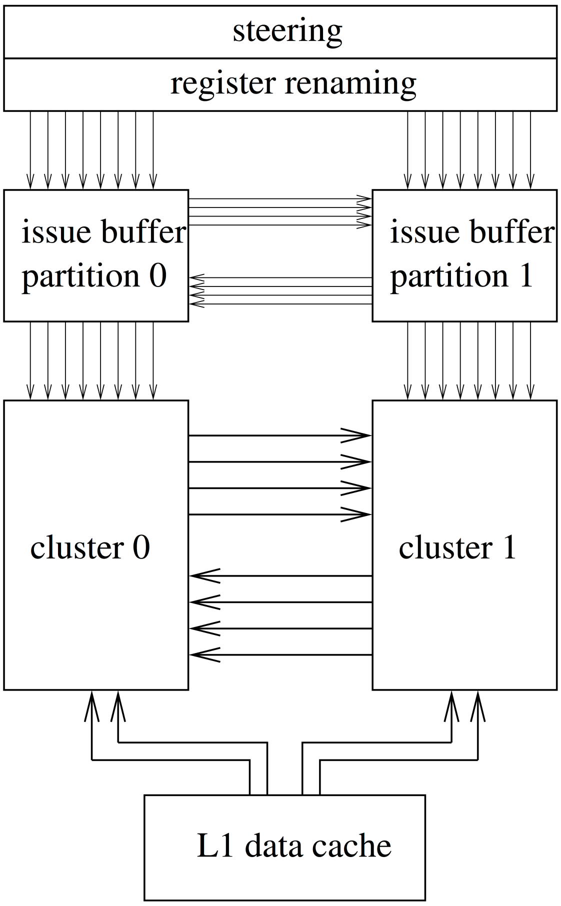

steering
register renaming
issue buffer partition 0
issue buffer partition 1
cluster 0
cluster 1
L1 data cache

Figure 2.7: Example of 2-cluster Out-of-Order engine, assuming dispatch-driven steering and register write specialization (which puts steering before register renaming).

intercluster delay [GNK+01].

2.4.3 Steering

Ideally, one wants both a good cluster load balancing and limited intercluster communications. However, these two goals contradict each other. The main problem is to achieve a good trade-off between load balancing and intercluster communications while still being able to steer several instructions simultaneously.

Several papers have studied steering policies for clustered microarchitectures. Some papers proposed different variants of dependence-based steering policies trying to steer dependent instructions to the same cluster to minimize intercluster communications, while trying to maintain sufficient load balancing be-

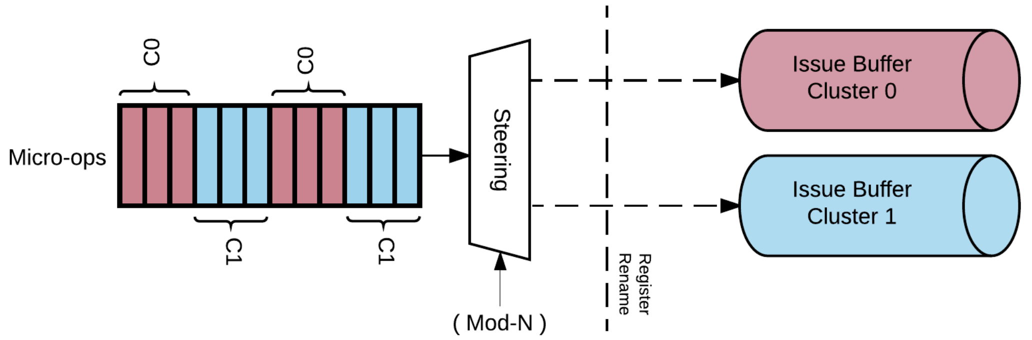


Figure 2.8: In Mod-N steering algorithm the destination cluster alternates every N micro-ops. C0 is Cluster 0, C1 is Cluster 1.

tween clusters [PJS97, CPG99, CPG00, FRB01, GLG04, SZ05]. However, these dependence-based policies are complex, and steering multiple instructions per cycle is an implementation challenge [SZ05, CCGG08].

Some authors proposed steering policies taking into account register dependencies [PJS97, CPG99, CPG00, BM00, GLG04]. However, these dependency-based steering policies are complex and make steering a clock frequency bottleneck, as the steering bandwidth must match the register renaming bandwidth.

A brief summary of various steering mechanism proposed in the literature [BM00, PG00, CPG00] is as follows:

First-Fit Assignment of instructions to the same cluster until it fills up completely. Each time a cluster is full, the next instructions are assigned to a new cluster, chosen using different possible heuristics: the cluster with the fewest assigned instructions, the next cluster in a specific sequence (in case of more than two clusters), the cluster close to became full, and so on

Load-cut (LC) Assignment of instructions to the same cluster until a load is encountered, except for sequences of consecutive loads. The intuition behind this heuristic is that a chain of dependent instructions could be introduced by a load.

Branch-Cut (BC) Assignment to the same cluster until a branch[5], based on the idea that a branch instruction is used to divide different blocks of instructions, and instructions within basic-block should be mostly dependent.

---

[5] According to Baniasardi and Moshovos [BM00], using only *backward* branches does not give significant performance improvements. Moreover, both BC at LC provide, on average, similar performance improvements

Slice-based mechanisms Based on the Slice (SLC) concept, described by Sastry *et al.* [SPS98] (e.g., LdSt Slice, Br Slice, Non-Slice), in which the steering is done for groups of correlated instructions. The idea is to reduce the number of intercluster communications by assigning all parents instructions and their consuming child to the same cluster.

Register Mapping Based mechanisms They use a distributed register file and a generated copy instruction for reading operands on a different cluster. These steering mechanisms (e.g., Simple RMB, Balanced RMB, Advanced RMB) try to minimize the number of intercluster communications by assigning each instruction to the cluster in which the highest number of its source operands are mapped [CPG01].

Steering solutions can be categorized as adaptive or unadaptive, instruction-based or group-based, complex or simple concerning of hardware implementation.

In our work, we consider a very simple steering policy: Mod-N. Mod-N steers N instructions to a cluster, the next N instructions to the next cluster, and so on in round robin fashion [BM00] as shown in Figure 2.8. Mod-N is simple enough to be implemented in hardware.

Baniasadi and Moshovos compared several different steering policies and found that a simple Mod-3 steering policy performs relatively well on their microarchitecture configuration (four 2-issue clusters, 1-cycle intercluster delay) [BM00].

In contrast to solutions like Mod-N, simple and with little need of hardware modifications, algorithms that perform the steering by using a predictive heuristic have been studied. Tune *et al.* [TLTC01] proposed a steering policy based on the run-time predicted critical-path. In their architecture, each cluster has its issue queue and a copy of the register file. The critical-path predictor works with two simultaneous phases: identify instructions of critical paths and predict future critical instructions using those already identified. The instruction is identified as critical through a mechanism of threshold that uses a counter in a dedicated *Critical Path Buffer*. They found a better performance improvement for the simplest solution (the *Blind* one) when applied to the dual-cluster configuration, but for 4-cluster configuration the situation looks different and a more accurate (and complex) assignment is required.

More complex mechanisms of critical path detection were proposed, like the idea of Field, Rubin and Bodik [FRB01] based on the observation that the compiler ability to identify the critical path is limited to the data dependencies.They indicate that the critical path varies depending on the architecture's resource constraints. A *dependence-graph-based* model is presented, and its fine-grain optimization opportunity is evaluated. The predictor can improve[6] the performance by up to 10% on average, with a spike of 21%.

Trace processors distribute chunks of consecutive instructions (traces) to the same Processing Element (PE), like Mod-*N* steering [VM97, RJSS97]. Rotenberg observed that the optimal trace size depends on the PE issue width and that, with 4-issue PEs, 32-instruction traces generally yield higher IPC than 16-instruction traces [Rot99]. A good place to store information about data dependencies and cluster assignment information could be the trace cache [RBS96, FPP98, PFP97]. For instance, the cluster assignment algorithm used by the register dependence pre-processing unit in [RF98] uses a register data flow graph (RDFG) to analyze the trace. The information obtained are used to divide the trace in data-dependent chains of instructions, and the assignment is done according to the dependency of the instruction at the head of each chain with the previous traces.

In the Integer-Decoupled architecture proposed by Palacharla, the steering was limited by constraints like the dependence between memory instructions. The load/store instructions were distributed among clusters so to avoid cross-cluster register dependencies. Then, in the work of Sastry *et al.* [SPS98], many of the constraints were relaxed like adding support for the copy of registers among clusters. A step forward was the mechanism of load/store slice partitioning, with the opportunity to duplicate specific instructions on both clusters.

We will not focus, in this thesis, on further details about critical-path predictors [SL98, JLW01, TLTC01] or trace cache solutions [FPP98, BJ03] that could be used in cluster architectures, leaving them for future analysis.

2.4.4 Write Specialization

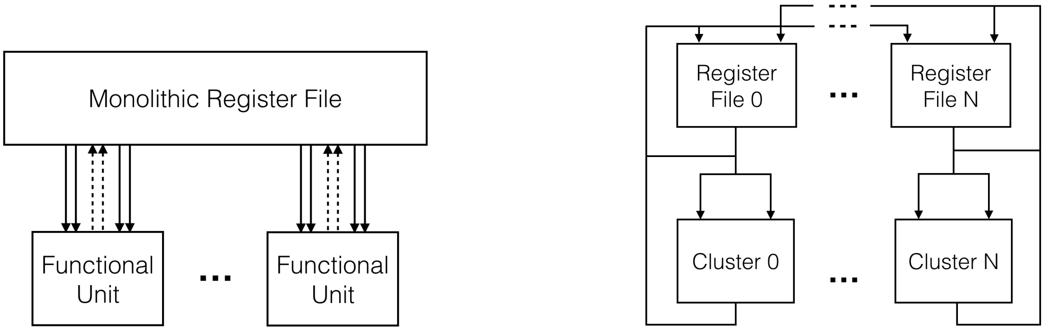
Executing more instructions per cycle requires to increase the number of read and write ports on the register file. However, the area and access energy of a Static Random Access Memory (SRAM) array increases quickly with the number of ports, especially with write ports [STR02].

A simple solution to increase the number of read ports is to duplicate the physical register file, as in the Alpha 21264 [Kes99]. However, this does not solve the problem for write ports. Register Write Specialization has been proposed to solve this problem [STR02]. For a clustered superscalar OoO engine, register write specialization means that each cluster writes in a subset of the physical registers [CPG00, ZK01, STR02] (Figure 2.9).

As an example, let us consider a single-cluster OoO engine using 128 physical registers with 4 write ports and 8 read ports. A dual-cluster OoO engine has

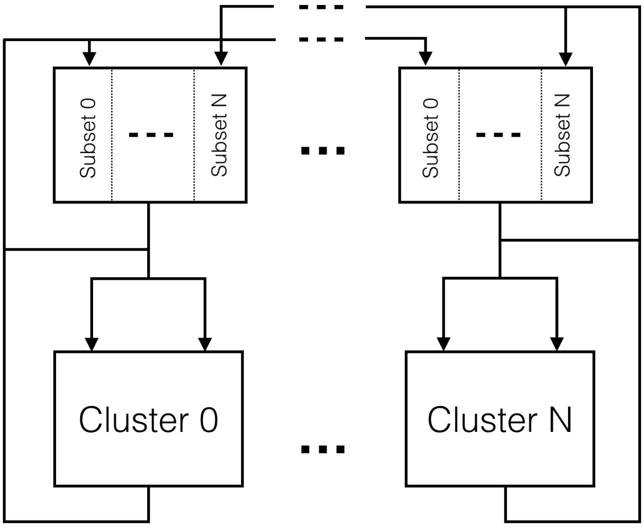
---

[6] The performance gain is evaluated over the configuration without the critical path predictor, using 12 SPEC2000 benchmarks


Monolithic Register File
Functional Unit
...
Functional Unit
Register File 0
...
Register File N
Cluster 0
...
Cluster N

(a) Monolithic Register File

(b) Clustered Register File


Subset 0
Subset N
Subset 0
Subset N
Cluster 0
...
Cluster N

(c) Subset for Write Specialization

Figure 2.9: Differences between different register file organizations

the double total issue width. We apply register write specialization, keeping the same total number of physical registers: partition 0 contains physical registers 0 to 63 and can be written only by cluster 0, partition 1 contains physical registers 64 to 127 and can be written only by cluster 1.

In order to allow the reading of any register on either cluster, each cluster has a mirror copy of the register partition of the other cluster. That is, each cluster has 2 banks, each bank holding half of the registers and having 4 write ports and 8 read ports. If we keep the total number of physical registers constant, the area of the per-cluster register file is roughly the same in the single-cluster and dual-cluster configurations.

Write specialization implies that steering should be finished before register renaming. It means that a complex steering policy cannot be completely overlapped with register renaming and would probably require extra pipeline stages for steering. However, Mod-N steering is very simple and can be done while the rename table is being read.

2.4.5 Bypass network and intercluster delay

Increasing the issue width increases the bypass network complexity, which may impact the clock cycle. Clustering solves this problem by making the bypass network hierarchical. It is illustrated in Figure 2.10 on an example of bypass network implementation for a dual-cluster OoO engine, assuming register write specialization. In this example, the first (i.e., most critical) bypass level (MUX 1) is not impacted by clustering. However, this costs an extra cycle of intercluster delay.

Moreover, the physical distance between clusters requires pipelining the result buses to decrease RC delays. Hence a bypass network such as the one depicted in Figure 2.10 entails at least 2 cycles of intercluster delay, one (or more) from pipelining the result buses and one from isolating the first bypass level. Note that there are several possible implementations for a bypass network[7]. In our simulations, we consider intercluster delays of up to 3 cycles.

Communication between clusters is an important key to reduce the overhead of clustering. The intercluster communication is subject to wire delay, and the number of these communications could be reduced through appropriate steering heuristics.

A survey of hierarchical interconnects for on-chip clustering is offered by Aggarwal and Franklin [AF02]. In brief, interconnections can be of different type (bar, crossbar, ring). The efficacy of each topology is related to the number of clusters, the architecture type, the executed program and other elements. The scalability problem could be addressed with hierarchical interconnections layout (as Single and Multiple rings of crossbars) associated with appropriate instruction distribution techniques. Moreover, Aggarwal and Franklin pointed out the improvement from distributing data caches among clusters connected with a hierarchical interconnect over the increased access latency of a centralized cache.

2.4.6 Clustering in Commercial Superscalar Processors

To the best of our knowledge, three commercial superscalar processors[8] have used clustering: the DEC Alpha 21264 [Kes99] and, recently, the IBM POWER7

---

[7] Data movements in a pipelined bypass network consume much power. This can be avoided with a CAM-like implementation of the bypass network [CMO99, PBB$^+$02]

[8] Some commercial VLIW processors also used clustering, e.g., Multiflow [LFK$^+$93]

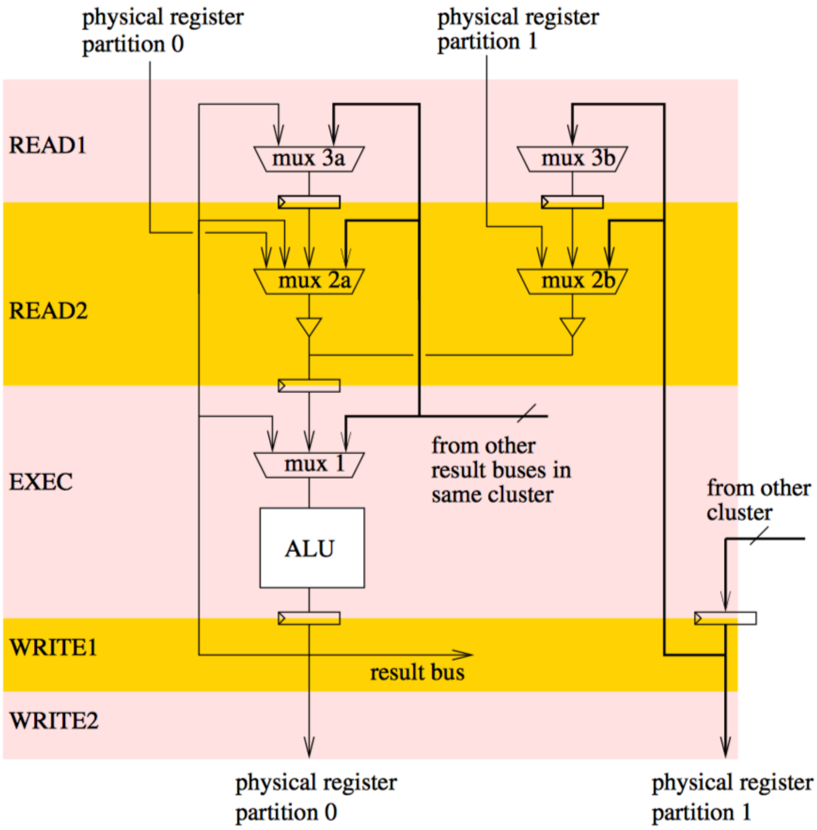

physical register
partition 0
physical register
partition 1
READ1
mux 3a
mux 3b
READ2
mux 2a
mux 2b
EXEC
mux 1
ALU
from other
result buses in
same cluster
from other
cluster
WRITE1
result bus
WRITE2
physical register
partition 0
physical register
partition 1

Figure 2.10: Example of bypass network implementation for a dual-cluster OoO engine, assuming 2 register read/write cycles and register write specialization (a single execution port and single source operand are shown, comparators are not shown). The first bypass level (MUX 1) is not impacted by clustering, but this costs an extra cycle of intercluster delay.

and POWER8 [SKS$^+$11, SVNE$^+$15]. These processors implement narrow-issue clusters with an intercluster delay of one cycle.

In IBM POWER7 several optimizations to reduce power consumption and chip area were proposed. It supports three modes: ST (Single Thread), SMT (Two-way thread mode) and SMT4 (four-way thread mode). The SMT4 uses a partitioned approach with two separate physical general-purpose registers, one for each couple of threads. The unified issue queue (UQ) is implemented as two 24-entry queues, and the load balance is maintained by dispatching instructions to each half of the UQ alternatively. The way in which operation types (i.e., integer, vector, floating-point, memory) are assigned to each queue changes according to the active thread-mode.

An example of dual-cluster commercial architecture is the Alpha 21264 [Kes99]. This architecture has two clusters for integer instructions, which communicate with each other through intercluster connections. The communication between clusters takes place when necessary to make the result of an instruction executed in a cluster available to the other cluster.

A goal of the Alpha 21264 was to reduce the complexity in order to increase the clock frequency. The complexity of some components like the local bypass network is reduced, and the registers require half of the number of register ports. Each cluster contains a copy of the register file, and a broadcasting system is in charge of keeping the state of both register file copies consistent. The instruction steering among clusters is based on the availability of data and functional units. The Alpha architecture provides two separate issue queues for integer and floating instructions. A problem of a dual-cluster system like the one implemented in the Alpha 21264 is the workload balancing between the two clusters. Each of them is able to process most integer instructions.

2.4.7 Clustered VLIW/DSP architectures

Although in this thesis we focus on the x86_64 architectures, various cluster solutions were proposed and introduced in other types of processors, such as Very long Instruction Word (VLIW) or Digital signal processor (DSP) architectures. In this section, we will not provide details about these implementations, but we provide some references to highlight the importance of this paradigm for different architectures.

Ozer and Benerjia [OBC98] proposed a scheduling algorithm named unified-assign-and-schedule (UAS) for clustered, statically-scheduled architectures. They examine instruction scheduling for clustered processors, focusing on the problem of assignment in VLIW architectures.

In [SG00], Codina *et al.* proposed to partition all resources among clusters, using the compiler for generating code in order to balance the workload. They use a scheduling technique in their architecture (called *multiVLIWprocessor*) that minimizes the intercluster communication using the data dependence graph.

Overall, the clustered paradigm was also used in DSP compilers [Des98] and embedded domain for commercial solutions. In TigerSHARC [FG00] has two computation blocks (*CompBlockX* and *CompBlockY*) connected to the three internal buses. Each block consists of a general-purpose register file, a shifter and execution units for different operations. Instructions could be issued to one of the two CompBlock or both. The TS320C67x [Inc06] presents two separate data paths, each of which contains a register file (16 registers) and four functional units. Each FU reads from and writes to the register file of its data path, and data paths are connected through 1X and 2X cross paths.

2.5 Energy saving exploiting loops

Due to the temporal locality of instructions during the loop execution, it can be convenient to make loop retrieval fast and exploit these capabilities. This potential has been addressed using both hardware or software solutions. To understand the impact of loops and why focusing on this specific component, we need to recall that the greatest part of the execution time is spent in loops.

The hardware solution we discuss in this thesis uses a dedicated logic that detects loops at run-time and a small cache (or *loop buffer*) to keep decoded micro-ops that belong to the detected loop body. The impact of this implementation depends on the ability (and speed) of the loop detector and on the size of the loop buffer.

Exploiting loops helps to reduce power consumption by switching off or reducing the usage of hardware components during the dynamic execution. The existence of blocks of instructions that are executed multiple times permits their reuse without further fetch (and eventually decode and rename). The existence of loops allows techniques of energy saving in both front-end and back-end, as described in this section.

Our focus is on exploiting a dynamic hardware mechanism that uses the presence of loops not only to reduce front-end energy during "loopy" workloads but to reduce the back-end energy consumption through the loop buffer manipulation, by eliminating useless memory accesses and redundant micro-op executions in the OoO engine.

2.5.1 Saving loop energy in the front-end

Ghose and Kamble [GK98] have attempted to resolve the problem of energy efficient organization of caches in superscalar processors, proposing the use of multiple block buffers with line bit isolation and sub-banking.

In Superscalar architectures, we exploit the ILP by executing multiple instructions in the same clock cycle. To make it happen, the front-end needs to fetch more instructions in the same clock cycle. The high instruction fetch bandwidth is a major source of energy consumption, and it must be supported by a decoding phase that also supports multiple instructions.

Bunda *et al.* [BFA95] and Su and Despain [SD95] explore the energy cost of fetching and delivering instructions. A simple small cache can achieve a good hit rate because of the locality principle during the fetch phase, reducing the total number of caches accesses. They observed that having a block buffer in the I-cache permits decreasing the energy consumption of the I-cache substantially, and is most effective for loops fitting entirely in a single instruction block.

The problem of energy consumption has become even more important with the spread of portable devices and embedded systems. This problem has been dealt with at the technology level, through the implementation of smaller and smaller transistors, more advanced production techniques for miniaturization and increasingly efficient frequency scaling mechanism. Nevertheless, the power consumption has become dominant also in non-embedded systems. The complexity introduced by the modern superscalar design imposes the research for new solutions to limit this problem. The front-end is a major limiting factor in the ability of a system to exploit instruction level parallelism because narrowing the number of fetched and decoded instructions represent an upper bound limit to the parallelism we can exploit with the wide-issue back-end.

An aggressive fetch unit can become a significant source of energy consumption [TM05]. The intuition for saving energy in the front-end comes from the idea of reuse instructions already fetched and store them in a specific memory buffer to avoid additional useless fetch. Instructions belonging to loops are perfect candidates to this reuse mechanism because during the loop each static instruction needs to be fetched and decoded only one time. The need for fetching multiple instructions each cycle may result in a significant power inefficiency; instructions are fetched from memory in blocks, and not individually. That causes unnecessary effort in the presence of branches, partially resolved by branch prediction mechanisms described in Section 2.2.2.

Another power inefficiency occurs even in the presence of loops. The instructions belonging to a loop are fetched, decoded and renamed at each loop iteration. The energy spent in front-end, if any buffer is used, is wasted energy. Yang *et al.* [YO06] assumed that all required information for the instructions reuses is provided by the front-end (with fetch and decode). They decided to use the ROB as a buffer for re-issue loop instructions, forwarding instructions to the rename stage directly from there through a *rob look-up step*. The renamed instruction is put back in the ROB as normal. The execution path changes when the instruction is reused from the ROB.They introduced an identification code for the logical blocks in which the ROB is divided. It allows the search of reusable blocks of instructions inside the queue.

2.5.2 Using a dedicated cache for loops

As accessing a small memory costs less energy than a large one, Kin *et al.* proposed to trade performance for reduced energy consumption in embedded processors by introducing a tiny level-0 (L0) I-cache [KGMS97]. They evaluate the energy dissipation in the cache of an embedded processor and propose to focus on simple direct mapped cache for future design in embedded processors, in order to improve the Energy-Delay performance. Subsequently, some researchers sought to recover the performance loss due to the L0 cache while keeping most of its energy benefit.

Bellas *et al.* proposed some heuristics to direct instruction fetch to the L0 or directly to the L1 I-cache depending on the occurrence of branch mispredictions and/or low confidence branch predictions [BHP99]. Based on the assumption that the program spends most of the time on specific portions of code, the idea was to identify these portions through a dynamic analysis of program access behavior and store them in the L0.

The approach to tackling the energy consumption in front-end optimizing the use of the I-Cache with the introduction of a new level of memory in the cache hierarchy could be not enough. Have misses in the additional L0 increases the total power usage. Besides a new level of cache, i.e., the filter cache, it is helpful to introduce a new mechanism to discriminate between the L1 and L0 accesses during the I-Fetch.The loops may have a predictable behavior, and if we use a special memory for loop instructions, we can reduce the number of miss in that memory during the entire execution. It is necessary to introduce a mechanism to predict if the instruction will hit in the L0 and allow the CPU to access the L0 cache directly. The L0 cache should be used only when the probability of a hit is high.

Tang *et al.* proposed to introduce a tiny predictor that predicts whether the *next* instruction fetch is likely to hit in the L0 cache, and to access the L1 cache directly in case an L0 miss is predicted [TGN01]. Due to the temporal reuse of instructions in small loops, a filter cache placed between the I-Cache and the Central Processing Unit (CPU) can save energy when requested instructions hit in this small cache. There is a loss of performance when the filter cache miss because of a misprediction. The trade-off is between the energy saved in case of filter cache hit and performance loss due to a miss.

Instead of an L0 cache, Lee *et al.* proposed to have a loop buffer associated with a mechanism detecting dynamic loops by monitoring short backward jumps [LMA99a]. This loop cache has only a data array for storing instructions, without the address tag. The technique is based on the concept of *branch distance* and the *short backward branch instruction* (sbb). The former is the distance, in number of instructions, between the branch and the target, assuming a Reduced Instruction Set Computer (RISC) ISA. The latter is a PC relative branch instruction, detected by the proposed hardware mechanism. The loop cache is used during the ACTIVE mode, enabled when the sbb is detected. During the ACTIVE mode, the main cache is turned off. The advantage of having a hardware loop detector is that it can be determined with certainty whether the next instructions can be fetched from the loop buffer.

When the dynamic hardware loop caching scheme is designed with compatibility in mind, like the one proposed by Lee *et al.*, the risk is to lose the ability to capture more complex or nested loops. However exploiting the potential benefits of dynamic loop buffering became more interesting for researchers. Rivers *et al.*

proposed a loop buffer able to detect and hold complex loops containing several branch instructions [RAWM03]. Besides the loop buffer, they add two specific registers that contain the address of start and end of detected loop, and a finite state machine (the DLB controller). The latter is used to control the state of the loop buffer memory. The controller is also capable of managing portions of a loop which are too large to fit within the dedicated memory. Their work shows a decrease of energy consumption over a traditional i-cache by a factor of three, using signal processing functions written for the *eLite* [MZS+03] as benchmarks.

When we identify a loop, we can use a special cache to store instructions reducing the energy consumption of the I-cache. Because the loop instructions are reused at each iteration, a step further can be to store decoded instructions. The decoded instructions can be fetched from the loop cache, allowing to gate the decoder during the program execution, sending the decoded instructions to the next step of the pipeline right after fetch. Anderson and Agarwala proposed a *decoded* loop buffer sitting between the instruction decoder and the execution core [AA00]. With their solution, they achieved activity reduction for instruction fetch by up to 83%.

Bajwa *et al.* proposed a decoded loop buffer with loop detection performed by the compiler, assuming a special Instruction Set [BHK+97]. They proposed two approaches, the first based on the use of a decoded instruction buffer (DIB) and the second based on a decoded instruction cache (loop-cache). The DIB is a SRAM used as a loop buffer that contains decoded instructions and exploits the simple innermost loops. The loop-cache (comparable in area) is a more complex solution and can exploit any locality of small loops, with the cost of a slight increase of area.

Identify and capture the loop allows using a smaller memory (i.e., loop buffer) with a lower energy consumption compared to the traditional instruction cache. Fetching from loop buffer allows storing decoded instructions avoiding the power consumption of decoders. Unfortunately, loops do not have all the same size. The number of loops in a program can vary widely: there can be many loops with a slight number of instructions in the body or loops with few iterations of tens or hundreds instructions. In the first case, we can maximize the loop buffer performing a partial loop unroll and storing in the buffer more than one iteration of the same loop. In the second case, the buffer loop size could represent a limit for the loop detector. It is essential to find a solution for storing at least a portion of the loop, in order to exploit the buffer even in the presence of loops with a large body size.

Another solution in which architecture components can be turned on according to the program requirement was proposed by Gordon-Ross *et al.* [GRCV02]. Their idea is based on a program pre-analysis used to specify a set of loop bodies to be pre-loaded into the loop cache, avoiding the dynamic detection overhead.

Recent studies have shown that most applications execute 10% of static instructions during 90% of their time [dAK01], mainly due to the presence of loops. Solutions such as the loop buffer allow the processor to identify, during the execution, instructions belonging to a loop. Introducing semantic information in the loop structures of the processor may allow exploiting ILP in a more complexity-effective way. With this semantic information, the execution of loops can be optimized. The energy consumption can also be reduced by decreasing the number of accesses to register rename structures or reducing the impact of branch prediction within the loop. In is necessary to make a further modification to the architecture in order to making it more “loop compliant”.

García *et al.* proposed to save energy further with a virtual register renaming scheme so that the dependency analysis required for register renaming can be completely done inside the loop buffer [GSF$^+$08]. The approach is to store renamed instructions in a buffer called *loop windows* that can directly feed the back-end. The intuition is that during each loop iteration the same group of instructions, executed repeatedly, are renamed once and again until the end of the loop execution. By observing the execution behavior in conventional processors, they notice that during the “loopy” workload there are repetitive operations that consume energy accessing hardware structures and executing the same operation repeatedly. Their architecture uses a Logical-To-Virtual Map table (LVM) in order to keep the association between virtual registers and tag. In their results, this solution not only has reduced the front-end activity but has slightly increased the performance.

Recent Intel processors feature a Loop Stream Detector (LSD) associated with the micro-op queue [Int16a]. When a dynamic loop that fits in the queue is detected, micro-ops constituting the loop body are kept there until the loop exits. Micro-ops are streamed to the register renaming stage directly from the micro-op queue.

2.5.3 Saving loop energy in the back-end

Some authors have tried to exploit loops in the back-end to decrease the energy consumed in the issue buffer or in the reorder buffer.

In Hu’s work [HVK$^+$04], the issue queue is capable of detecting reusable instructions during the execution. The detection is not limited to the instructions belonging to tight loop, and a loop is identified only when the static distance between the branch address and the target instruction address is less than the issue queue size. The issue queue is augmented with new components for supporting the instruction reuse and a mechanism for recovery to the normal execution state. Hu *et al.* also point out the impact of compiler optimizations on the detection of large loops with a relatively small issue queue.

Pratas *et al.* [PGB+08] proposed a finite state machine for the loop detection, and hardware modifications to use the reorder buffer as the source of instruction loop reuse, like a small buffer used to store register renaming data information. Hayenga *et al.* proposed novel instruction scheduler, register renaming, and memory disambiguation for allowing in-place execution of loops [HNL14].

The use of a dedicated cache memory for loops in the front-end allows implementing new solutions for exploiting the ILP and saving energy in the front-end. There are, however, not so many studies in the literature that aim to exploit the loop behavior to save energy in the back-end. The loops show intrinsic properties that we have tried to exploit in this thesis. With these properties in mind, we can make changes to the back end to reduce the power consumption in a novel way. As we will see in Chapter 4, we can identify loop instructions that do not require to be issued at each iteration. These loop instructions need to be executed only once.

When a loop has, for example, a high number of iterations, the execution of these instructions only once allows to reduce the activity of the back-end and consequently the consumption of energy. Concerning the instruction loop that accesses the memory, it is possible to identify the access pattern and to implement a prediction mechanism able to predict the structures required. This prediction allows discriminating between access to the cache and access to the store queue, reducing the number of useless accesses. The redundant access to cache or store buffer without taking into account where the instruction will find the data required involves a high energy consumption. In this thesis, we propose a way to reduce this redundant access, thereby reducing the power consumption without increasing the complexity of the processor.

2.5.4 Loop accelerators

In addition to the approach of modifying the conventional superscalar core to save energy in loop execution, another solution to exploit loops is to implement a hardware loop accelerator for increasing performance and/or decreasing energy consumption.

Clark *et al.* proposed a co-designed virtual machine, with a programmable loop accelerator that can execute modulo-schedulable loops and a dynamic translator that maps loops dynamically onto the accelerator [CHM08]. This *Virtualized Execution Accelerator for Loops* (VEAL) tries to eliminate the costs associated with designing loop-accelerators from scratch and requires a dynamic compilation system to create binaries that are also compatible with general-purpose systems without any hardware accelerator.

Explicit-dataflow architectures [BKM+04, MCC+06] offer a different approach compared to the traditional “control flow” paradigm, and recent studies [MHS+15, ME15, PA12, HMS$^+$15] try to use the dataflow idea to accomplish performance improvements or scalable architecture solutions. Nowatzki *et al.* proposed to implement a loop accelerator as an explicit-dataflow engine [NGS15].

Srinath *et al.* proposed a small extension to the conventional instruction set allowing, with the help of programmer annotations, to compile loops in such a way that they can be executed either on a loop accelerator or on a (conventional) general-purpose core, without recompilation [SIT$^+$14].

2.5.5 Industrial adoption of loop cache solutions

Some loop buffers were already implemented in mainframe computers in the 1960's, but only for performance [Tho65, AST67]. A "loop cache"-like solution to profit from branches and tight loops was implemented in the AMD-29000 [Eve90] with a branch target cache on-chip that can provide the first four instructions after the target branch.

Recent Intel processors have a decoded L0 I-cache, aka *micro-op cache*, sitting between the L1 I-cache and the micro-op queue [Int16a]. A micro-op cache is a particular implementation of the more general concept of *trace cache* [RBS96, PFP97, SMR$^+$03].

Various recent microarchitectures feature a loop buffer, such as ARM Cortex-A15 [Lan11], AMD Jaguar [Rup12] and recent Intel processors. The AMD64 architecture of the AMD Family 16h processor [AMD13] includes a loop buffer of 128 bytes which can reduce power consumption when hot loops fit entirely within in. The loop is logically divided in chunks and integrated with the instruction cache line. The Cortex-A15 has a 32 entry loop cache which can contain up to two forward branches and one backward branch. When this buffer is used, the fetch phase and most of the decode stages of the pipeline are completely disabled.

In the Intel Silvermont architecture [Int16a], loops are detected by the *Loop Stream Detector*, that helps to prevent some decode restrictions on the instruction length. The LSD was introduced with the Sandy Bridge, in which the buffer can contain up to 28 micro-ops with a limitation on the number (up to 8) and type (no `call` or `return`) of taken branches.

A difference between Sandy Bridge and subsequent implementation of the LSD in Ivy Bridge and Haswell concerns the multi-thread use of the loop cache. In Sandy Bridge, the 28 entry micro-op queue was replicated for each thread. In recent Intel implementations, it is combined into a single structure that is statically partitioned when two threads are active. In this way, the single thread execution can better use the available resources. The trend is to increase the loop cache size. For instance, The LSD in the Intel Skylake can detect loop bodies of up to 64 micro-ops [Int16a].

Chapter 3

Wide Issue Clustered Microarchitecture

During the last ten years, the clock frequency of high-end superscalar processors did not increase. Performance keeps growing mainly by integrating more cores on the same chip and by introducing new instruction set extensions. However, this benefits only some applications and requires rewriting and/or recompiling these applications. A more general way to accelerate applications is to increase the Instruction per cycle (IPC), the number of instructions executed per cycle. Though the focus of academic microarchitecture research moved away from IPC techniques, the IPC of commercial processors was continuously improved during these years. We argue that some of the benefits of technology scaling should be used to raise the IPC of future superscalar cores further.

3.1 A case of increasing single-thread IPC

If the clock frequency remains constant, the only possibility left for increasing the single-thread performance of future processors is to exploit more ILP.

Table 3.1 shows three possible scenarios for exploiting one step of technology scaling under constant clock frequency and constant voltage[1], assuming identical cores. The core power, core IPC, Energy per instruction (EPI) and clock frequency are related as follows:

$$\textit{core power} = \text{EPI} \times \text{IPC} \times frequency$$

If voltage remains constant, technology scaling decreases the switching energy

---

[1] High leakage currents make difficult to scale down the voltage without severely hurting transistor speed. This is one of the reasons for voltage scaling down only very slowly [fS13]. We conservatively assume a constant voltage.

Table 3.1: Three possible scenarios for exploiting one step of technology scaling, assuming constant frequency, constant voltage and homogeneous cores

	same core micro-arch		**more complex core micro-arch**
dimensions x, y, z	$\times\frac{1}{\sqrt{2}}$	$\times\frac{1}{\sqrt{2}}$	$\times\frac{1}{\sqrt{2}}$
frequency	$\times 1$	$\times 1$	$\times 1$
voltage	$\times 1$	$\times 1$	$\times 1$
core area	$\times\frac{1}{2}$	$\times\frac{1}{2}$	$\times\frac{1}{2}\gamma$
number of cores	$\times 2$	$\times\sqrt{2}$	$\times 1$
core IPC	$\times 1$	$\times 1$	$\times\alpha$
dynamic EPI	$\times\frac{1}{\sqrt{2}}$	$\times\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}} \times \beta$
total power (cores)	$\times\sqrt{2}$	$\times 1$	$\times\frac{1}{\sqrt{2}}\alpha\beta$
power density	$\times\sqrt{2}$	$\times\sqrt{2}$	$\times\sqrt{2}\frac{\alpha\beta}{\gamma}$

$\frac{1}{2}CV^2$ only by reducing capacitance C, i.e., roughly as $\frac{1}{\sqrt{2}}$ [DGR+74]. Hence if we consider a fixed core microarchitecture, the silicon footprint reduction from technology scaling means roughly a $\frac{1}{\sqrt{2}}$ reduction of the dynamic EPI.

The second and third column in Table 3.1 assume a fixed core microarchitecture. The first scenario (second column) corresponds to doubling the number of cores on each new technology generation, which technology scaling makes possible in a constant silicon area. However, this scenario leads to an increase of the total power (this is the dark silicon problem [EBSA+11]).

The second scenario (third column) corresponds to increasing the number of cores but more slowly than what technology scaling permits, so as to keep the total power constant. Power density increases but remains inversely proportional to the circuit dimensions, hence hot spots temperature remains constant.

The third scenario (fourth column of Table 3.1) shows a situation where the number of cores is kept constant, but single-thread performance is increased with a more complex core microarchitecture[2].

For example, a 10% increase of core IPC ($\alpha = 1.10$) which is obtained at the cost of 28% more EPI ($\beta = 1.28$)[3] results in a constant total power and 10%

---

[2] The second and third scenario of Table 3.1 do not exclude each other. They can be used at different technology generations. Moreover, keeping the number of cores constant does not preclude introducing more simultaneous multi-threading (SMT) contexts.

[3] Making the core more complex does not necessarily increase the EPI (factor β). A large

overall EPI reduction after scaling down the feature size.

The main point is that the core microarchitecture complexity can be increased progressively, over several technology generations, with the EPI globally decreasing thanks to technology scaling.

3.2 Experimental Framework

We decided to use Haswell as a base scheme for the baseline we will describe below. Most of the information about this architecture are present in [Int16a]. Intel, with 32 nm Sandy Bridge, introduced its first System-on-Chip (SoC) in the market, improving most of the fundamental aspect of architecture pipeline and proposing, among other introductions, new solutions for energy consumption.

Haswell is a CISC architecture capable to decode five instructions per cycle and dispatch up to eight micro-ops each cycle (Figure 3.1) to the eight ports present, divided by memory and arithmetic operations. Two separate buffer are used for memory micro-ops: a load queue (72 entries) and a store queue (42 entries). Moreover, there is a dedicated address generator unit for store operations. It has three simple decoders and one complex decoder. The complex one is capable to emit up to four micro-ops per cycle. A micro-op cache buffer is added to avoid to decode instructions decoded recently. The decoders consume a relevant amount of energy, and can be disabled when micro-ops are reused during loops. This cache helps to save energy at the cost of adding a new memory zone for containing decoded micro-ops.

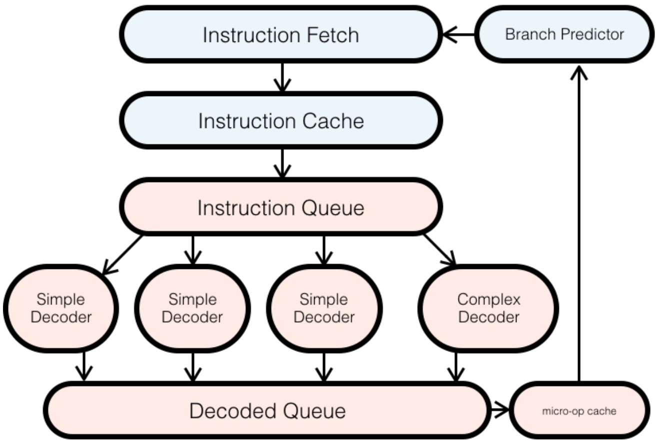
Haswell has unified integer and vector renaming, scheduling and execution resources. The pipeline is shared between two threads (when needed). The reorder buffer is capable to contain up to 192 micro-ops, allowing the scheduler to keep the execution units busy most of the cycles. Each reorder buffer entry can contain a single fused micro-ops. The register file has 168 integer registers and 168 vector registers. Both micro-op and macro-op fusion techniques are supported. The first is used to save pipeline bandwidth, the second allows the decoder to fuse arithmetic or logic instructions with subsequent conditional jump into a single *compute-and-branch* micro-op.

3.2.1 Simulation Setup

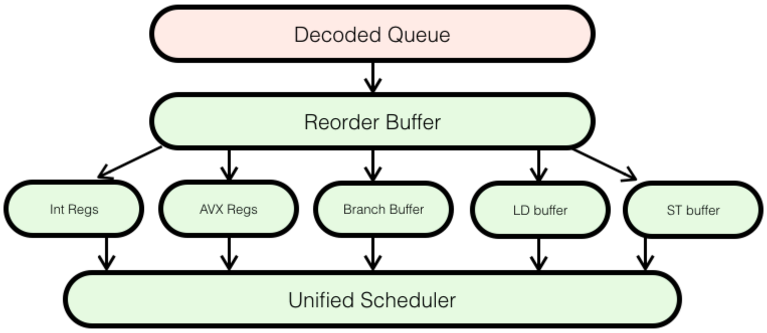
The microarchitecture simulator used for this study is an in-house simulator based on Pin [LCM+05]. Operating system activity is not simulated. The simulator is

---

fraction of the total CPU power is static power from leakage currents. And a large fraction of this static power is gateable on modern processors, i.e., can be turned off once a task is finished. Increasing the IPC makes the execution time shorter, and may reduce static energy consumption [CLG+14]


Instruction Fetch
Branch Predictor
Instruction Cache
Instruction Queue
Simple Decoder
Simple Decoder
Simple Decoder
Complex Decoder
Decoded Queue
micro-op cache

(a) Front-end


Decoded Queue
Reorder Buffer
Int Regs
AVX Regs
Branch Buffer
LD buffer
ST buffer
Unified Scheduler

(b) Back-end

Figure 3.1: A simplified scheme of front-end and back-end in recent Intel microarchitecture.

trace driven and does not simulate the effects of wrong-path instructions[4]. We simulate the 64-bit x86 instruction set. Instructions are split by the microarchitecture into micro-ops. For simplicity, the simulator defines 18 micro-op categories.

3.2.2 Benchmarks

The benchmarks used for this study are the `SPEC CPU 2006` [Hen06]. They were all compiled with `gcc -O2` and executed with the reference inputs. A trace is generated for each benchmark. Each trace consists of about 20 samples stitched

---

[4] We believe that our general conclusions are largely independent from wrong path effects. Modern OoO schedulers prioritize older instructions (in program order) when selecting among ready ones [PBB+02, SKS+11, GAV11], and correct-path instructions are generally not delayed by wrong-path ones.

Table 3.2: Baseline Microarchitecture

clock frequency	3.5 GHz
branch predictor	31 kB TAGE, 6 kB ITTAGE
I-fetch	1 cache line, 1 taken branch per cycle
decode	8 instructions / cycle
rename	12 micro-ops / cycle
issue (micro-ops)	4 INT, 2 FP, 3 address, 2 store data
load issue	2 loads / cycle
retire	12 micro-ops / cycle
reorder buffer (ROB)	256 micro-ops
physical registers	128 INT, 128 FP
issue buffers	60 INT micro-ops, 60 FP micro-ops
load queue	72 loads
store queue	42 stores
branch misp. penalty	12 cycles (min), redirect I-fetch at branch exec
cache line	64 bytes
MSHRs	32 block requests
DL1 cache	32 kB, 8-way assoc.,latency 3 cycles, 8 banks, 2 reads & 1 write/cycle
IL1 cache	32 kB, 8-way assoc.
L2 cache	512 kB, 8-way assoc., latency 11 cycles
L3 cache	8 MB, 16-way assoc.,latency 21 cycles
memory	latency (fixed) 245 cycles, bandwidth 16 bytes/cycle
prefetcher	stride prefetcher (L1), stream prefetchers (L2/L3)
page size	4 MB
store sets	SSIT 2k (4-way skewed-assoc.), LFST 42 stores

together. Each sample represents 50 million executed instructions. The samples are taken every fixed number of instructions and represent the whole benchmark execution. In total, 1 billion instructions are simulated per benchmark.

3.2.3 Baseline Microarchitecture

Our baseline superscalar microarchitecture is representative of current high-end microarchitectures. The main parameters are given in Table 3.2. Several parameters are identical to those of the Intel Haswell microarchitecture, in particular the issue buffer and load/store queues [Int16a].

Macro-fusion is applied to conditional branches when the previous instruction

Table 3.3: IPC for the baseline configuration - `SPEC2006` Benchmark Suite

SPEC INT	**IPC**
400.perlbench	2.97
401.bzip2	1.70
403.gcc	1.80
429.mcf	0.50
445.gobmk	1.97
456.hmmer	3.64
458.sjeng	2.40
462.libquantum	3.28
464.h264ref	2.99
471.omnetpp	1.05
473.astar	0.75
483.xalancbmk	1.44

SPEC FP	**IPC**
410.bwaves	2.09
416.gamess	2.94
433.milc	2.46
434.zeusmp	1.46
435.gromacs	1.76
436.cactusADM	1.97
437.leslie3d	1.57
444.namd	1.95
447.dealII	2.70
450.soplex	0.94
453.povray	2.56
454.calculix	2.36
459.GemsFDTD	1.55
465.tonto	2.73
470.lbm	1.38
481.wrf	1.97
482.sphinx3	1.95

modifies the flags and is of type ALU [Int16a]. That is, the two instructions give a single micro-op. Recent Intel processors feature a micro-op cache and a loop buffer [Int16a], which are not simulated here (this study focuses on the OoO engine). Instead, we simulate an aggressive front end delivering up to 8 decoded instructions per cycle.

For every memory access, we generate an address micro-op for the address calculation and a store micro-op or a load micro-op for writing or reading the data. The load queue can issue 2 loads per cycle.

The micro-ops have two inputs and one output. Some partial register writes require to read the old register value. When necessary, we introduce an extra micro-op to merge the old value and the new value. Each physical register is extended to hold the flags, which are renamed like architectural registers. A micro-op that only reads the flags, such as a non-fused conditional branch, accesses a read port of the physical register file. Micro-ops that do not write a register or do not update the flags do not reserve a physical register. These include branch, store, and address micro-ops (address micro-ops write in the load/store queues).

Loads are executed speculatively. Load mispeculations are repaired when a mis-executed instruction is to be retired from the reorder buffer, by flushing the instruction window and re-fetching from the mis-executed instruction. The

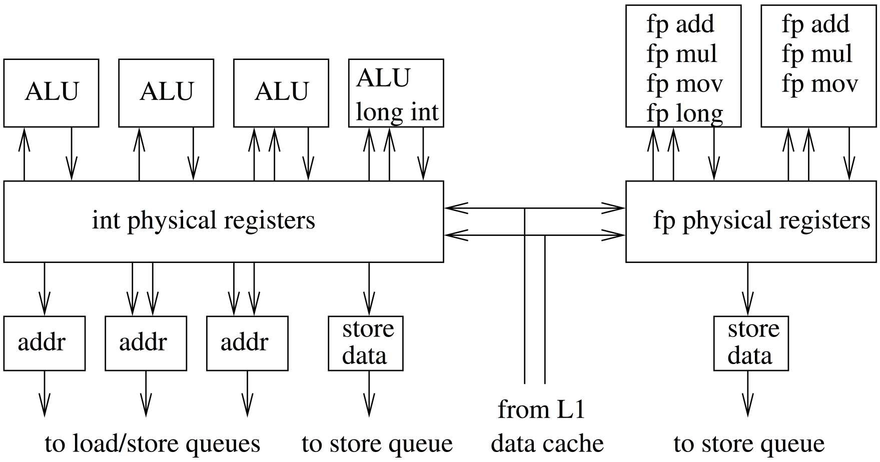

ALU
ALU
ALU
ALU
long int
fp add
fp mul
fp mov
fp long
fp add
fp mul
fp mov
int physical registers
fp physical registers
addr
addr
addr
store
data
store
data
from L1
data cache
to load/store queues
to store queue
to store queue

Figure 3.2: Baseline Out-of-Order engine with 8 INT execution ports and 3 FP execution ports. The INT register file has 12 read ports and 6 write ports. The FP register file has 5 read ports and 4 write ports. The issue buffers are not shown

memory independence predictor is the Store Sets [CE98]. A load can execute speculatively if the most recent store in its store set (the "suspect" store) has been retired from the store queue. A load can also execute speculatively if a matching store queue entry can provide the data to the load and that store is not older than the "suspect" store. The Store Set ID Table (SSIT) is indexed by hashing load/store PCs, using different hashing functions for loads and stores (an x86 instruction may contain both a load and a store).

Figure 3.2 depicts the baseline OoO engine. The integer and floating point clusters each have their own issue buffer and scheduling logic. There are 8 Integer (INT) execution ports and 3 FP execution ports. The execution ports for address micro-ops and store micro-ops do not need a register write port. Instead, they write in the load/store queues. We assume 128 INT and 128 FP physical registers, which is sufficient for our single-threaded baseline core. Three INT execution ports are specialized with only one register read port. This saves 3 read ports[5].

In total, the INT register file has 12 read ports and 6 write ports. The FP register file has 5 read ports and 4 write ports. The ALU execution ports can execute most INT micro-ops. An execution port is selected for a micro-op before it enters the issue buffer. For load balancing, if several execution ports can execute a given micro-op, the micro-op is put on the execution port that received a micro-op least recently.

---

[5] We checked that this has very little impact on the IPC

3.3 Potential IPC gains from a more complex superscalar microarchitecture

This section studies the impact on IPC of enlarging certain critical parts of the microarchitecture, ignoring implementation issues. The configurations considered in this section are non-clustered microarchitectures. We focus on first-order back-end parameters that may have an important impact on IPC. The parameters are listed in Table 3.4. Some of the parameters are lumped to limit the configuration space. We include front- end width in the list of parameters as a wider back-end generally requires a wider front-end. The 7 parameters of Table 3.4 define 128 non-clustered configurations (including the baseline), each parameter being either as in the baseline core, or doubled compared to the baseline.

We simulated all 128 configurations and obtained the IPC of each of them. The IPC for the baseline are displayed in Table 3.3. For all the other configurations, the IPC of each benchmark is normalized to its baseline IPC, that is, only speedups are given

To present our simulation results in a concise way, we use the following method for naming the 128 configurations. Each of the 7 parameters is represented by a unique symbol (see Table 3.4). The presence of that symbol in a configuration's name means that the corresponding parameter is twice as large as in the baseline core, otherwise it is dimensioned as the baseline. For instance, configuration *Bicw* features INT and FP issue buffers of 120 micro-ops, 16 INT execution ports (the ones shown of Figure 3.2, duplicated), can issue 4 loads and do 2 writes in the L1 data cache per cycle, can predict 2 taken branches and fetch 2 instruction cache blocks per cycle cycle. Otherwise, it is identical to the baseline.

For summarizing the simulation results, we define configuration classes based on the configuration's name length. For example, the baseline configuration is in class 0 and configuration *Bicw* is in class 4. Then, for each configuration class from 0 to 7, we find the worst (lowest mean speedup) and best (highest mean speedup) configurations in that class.

Figure 3.3 shows speedups over the baseline for the worst and best configurations in each configuration class, with the configurations' names indicated. Notice that there is no single bottleneck in the baseline for the `SPEC INT`, as the best configuration in class 1 yields only +5% sequential performance. For the `SPEC INT`, the most effective single parameter is the number of INT execution ports. Indeed, the best configuration in class 1 doubles the number of INT execution ports (i), and the worst configuration in class 6 is the one without i (the speedup drops from 1.25 to 1.12).

For the `SPEC FP`, the most effective single parameter is the number of FP execution ports. Just doubling the number of FP execution ports (f) yields +11% sequential performance on average, and the worst configuration in class 6 is the

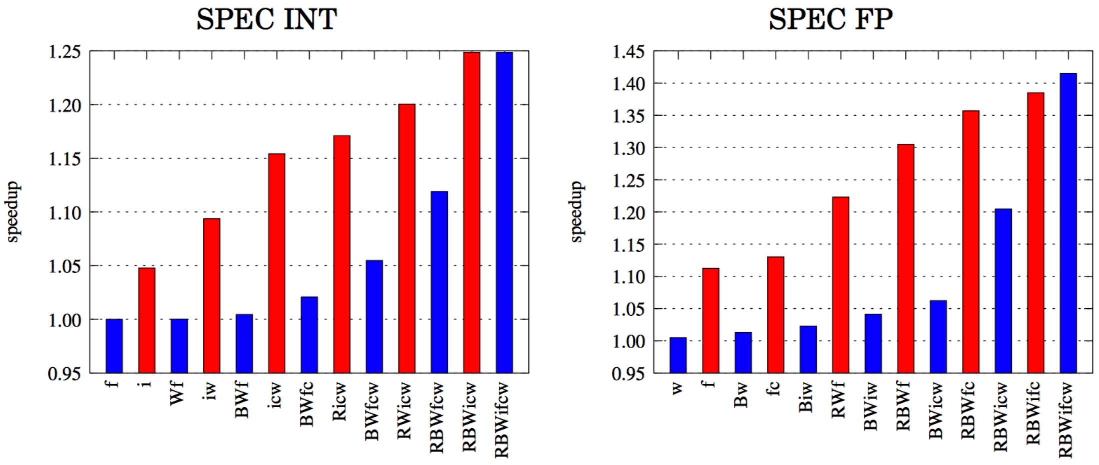


Figure 3.3: Speedup over the baseline for the worst and best configuration in each configuration class. The left graph is the geometric mean speedup for the 12 `SPEC INT` benchmarks, the right graph for the 17 `SPEC FP` benchmarks. The presence of a letter in a configuration's name indicates that the corresponding parameter is doubled compared to the baseline

one without f (the speedup drops from 1.41 to 1.20). The other width parameters are not important bottlenecks for the `SPEC FP`, which are more sensitive to window parameters (physical registers, ROB, etc.).

The extra microarchitecture complexity can be introduced incrementally over several technology generations. For instance, a first step could be to increase the number of FP execution ports, as this brings significant performance gains on scientific workloads. Increasing simultaneously the number of INT execution ports, DL1 bandwidth and front-end width could be a second step, which would allow more Simultaneous Multi-Threading (SMT) contexts.

However, the hardware complexity of the scheduler and bypass network increases quadratically with the issue width [PJS97], and clustering must be introduced at some point in order not to impact the clock frequency. In the next section, we quantify the impact of clustering on IPC.

3.4 Dual-Clustered Configurations

Section 3.3 did not take into account the potential impact on the clock cycle. Increasing the number of execution ports while keeping the same clock frequency will require to use clustering at some point. We consider only dispatch-driven steering, i.e., the issue buffer is clustered just like the EUs and the bypass network, and the steering is done before instruction enters the issue buffer. We assume

symbol	**parameters**
R	total number of physical registers
B	total issue buffer size
W	ROB, load & store queues, LFST, MSHR
I	INT execution ports
f	FP execution ports
c	load issue width & DL1 write ports
w	front-end (I-fetch, decode, rename, retire)

Table 3.4: Non-clustered microarchitecture parameters. Upper case letters are for instruction window parameters, lower case letters for width parameters. Parameters W, w and c are lumped parameters.

symmetric clusters, i.e., the two clusters are identical and can execute all micro-ops. An advantage of symmetric clustering is that in SMT mode, the clusters can execute distinct threads and the intercluster bypass network can be clock-gated.

In this section, we study the impact of the intercluster delay on IPC, assuming symmetric clustering and register write specialization (Section 2.4).

The INT and FP clusters depicted in Figure 3.2 are duplicated[6], which means a total a 8 ALUs, 6 address generators, and 4 floating-point operators. The INT and FP registers are split in 2 partitions (write specialization). Cluster 0 writes in partition 0 and cluster 1 writes in partition 1. Each partition is implemented with 2 mirror banks, one bank in each cluster. Each cluster has a bank for partition 0 and a bank for partition 1, but writes only in its own partition. Each of the 4 banks has the same number of read and write ports as the register file in the single-cluster baseline (Section 2.4.4).

The L1 data cache is banked, with 8 banks interleaved per 8-byte word, like the Intel Sandy Bridge [Int16a]. The load queue can issue 4 loads per cycle, instead of 2 in the baseline. Before being entered in the load queue, loads are “steered” to one of the register file partition (a bit in each load queue entry indicates to which partition the load has been steered). There are 2 write ports dedicated to loads in each physical register partition (Figure 2.7). Each cycle, the load scheduling logic selects the 2 oldest ready loads in each partition.

For the clustered configurations, we assume a DL1 latency of 4 cycles, instead of 3 cycles for the baseline, to take into account the extra complexity of the DL1 cache.

We consider 2 clustered configurations, named using a method similar to the one used in Section 3.3, except that notation *iiffcc* means that the baseline cluster of Figure 3.2 is duplicated and the issue buffer and physical registers are partitioned:

---

[6] The reason for clustering the FP execution ports is that floating-point operators have a big silicon footprint, and result buses have to span a long physical distance.

- ***iiffccw***. Same total instruction window capacity as the baseline (same ROB, same Miss Status Holding Register (MSHR), etc.), but double front-end bandwidth and dual-cluster back-end. Each cluster has an issue buffer partition of 30 micro-ops and a physical register partition of 64 registers.
- ***RBWiiffccw***. Dual cluster back-end, but with the total instruction window capacity doubled: twice bigger ROB, MSHR, load/store queues and Last Fetched Store Table (LFST), each cluster having an issue buffer partition of 60 micro-ops and a physical register partition of 128 registers.

3.4.1 Dual-Cluster with baseline instruction window size

Figure 3.4 shows the IPC gain over the baseline for the *iiffccw* clustered configuration, assuming intercluster delays (ICD) of 0,1,2 and 3 cycles[7] under a Mod-N steering policy with N ranging from 1 to 256. Recall that Mod-N steers N x86 instructions (i.e., more than N micro-ops) to a cluster, the next N instructions to the other cluster, and so forth. The leftmost bar of each group of bars in Figure 3.4 shows the IPC gain when steering all micro-ops to cluster 0.

While the non-clustered configurations showed that it is beneficial to increase the issue width first, and then the instruction window, Figure 3.4 shows that for the clustered configurations the situation is quite different. Indeed, the main conclusion from Figure 3.4 is that clustering without enlarging the total instruction window does not bring any IPC gain when the intercluster delay is 3 cycles.

Several factors contribute to this. The increased DL1 latency (from 3 to 4 cycles), the partitioning of the issue buffer and physical registers, the clustering of execution units and load issue ports contribute to decreasing the potential IPC gain, even with a null intercluster delay and a Mod-1 steering. But the biggest impact comes from the intercluster delay. As the intercluster delay increases, the IPC gain drops quickly and eventually becomes an IPC loss, even with the best steering policy (Mod-32 here).

3.4.2 Dual-cluster with double instruction window

Figure 3.5 gives the IPC gain over the baseline for the *RBWiiffccw* clustered configuration. Here, the total instruction window capacity is doubled. In particular, the issue buffer partition and physical register partition of each cluster have the same size as the baseline.

---

[7] intercluster delays of 0 and 1 cycles are not realistic, they are provided only for analysis.

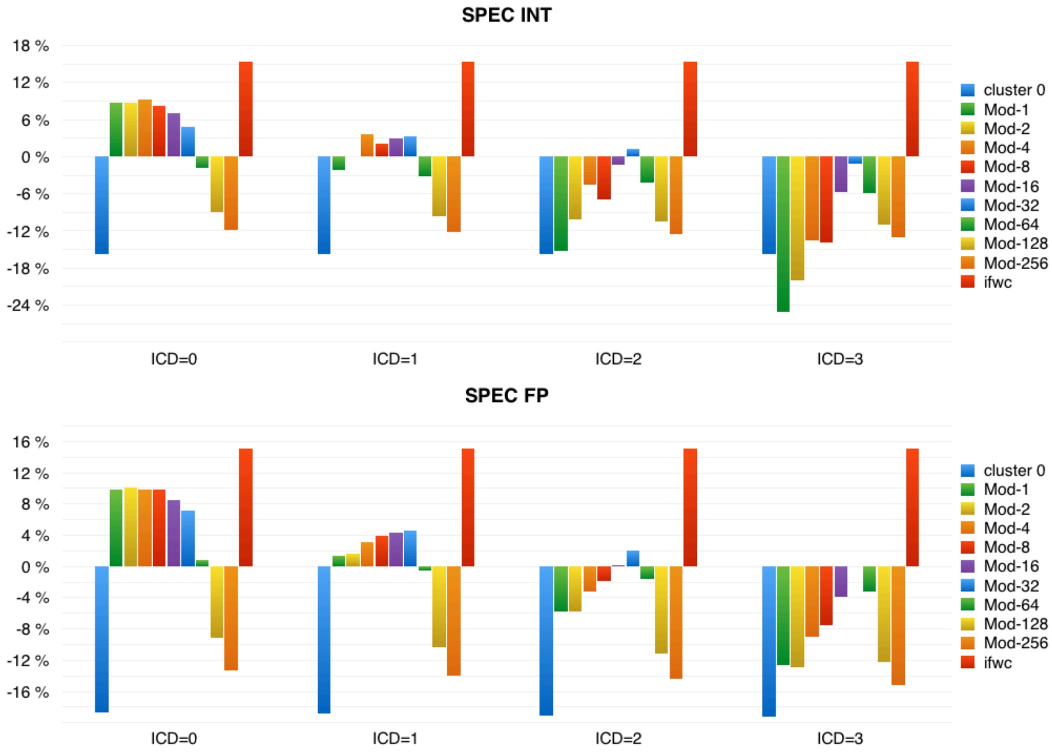


Figure 3.4: IPC gain over the baseline for clustered *iiffccw* configurations, for intercluster delays (ICD) of 0,1,2 and 3 cycles, under a Mod-N steering policy with N ranging from 1 to 256. The leftmost bar of each group of bars is the IPC gain when steering all the micro-ops to cluster 0. The rightmost bar is the IPC gain of the non-clustered *ifcw* configuration. The top graph is the average for SPEC INT, the bottom one for SPEC FP.

Now, the dual-cluster can outperform the baseline for some Mod-N steering. When steering all micro-ops to cluster 0, the IPC is very close to the baseline, with the impact of the increased DL1 latency more or less compensated by the increased reorder buffer and MSHRs. When the intercluster delay is null, Mod-N steering with $N \leq 32$ achieves a good ILP balancing. For $N > 32$ ILP imbalance decreases the IPC.

As the intercluster delay increases, Mod-N steering with small values of N generates significant IPC drop. With a large N, there are fewer intercluster communications and better tolerance to the intercluster delay. With an intercluster delay of 2 cycles, Mod-64 gives an average IPC gain of +14.3% on the `SPEC INT` and +29.6% on the `SPEC FP`. With an intercluster delay of 3 cycles, the IPC gain with Mod-64 is still +12.9% and +28.0% on the `SPEC INT` and `SPEC FP`

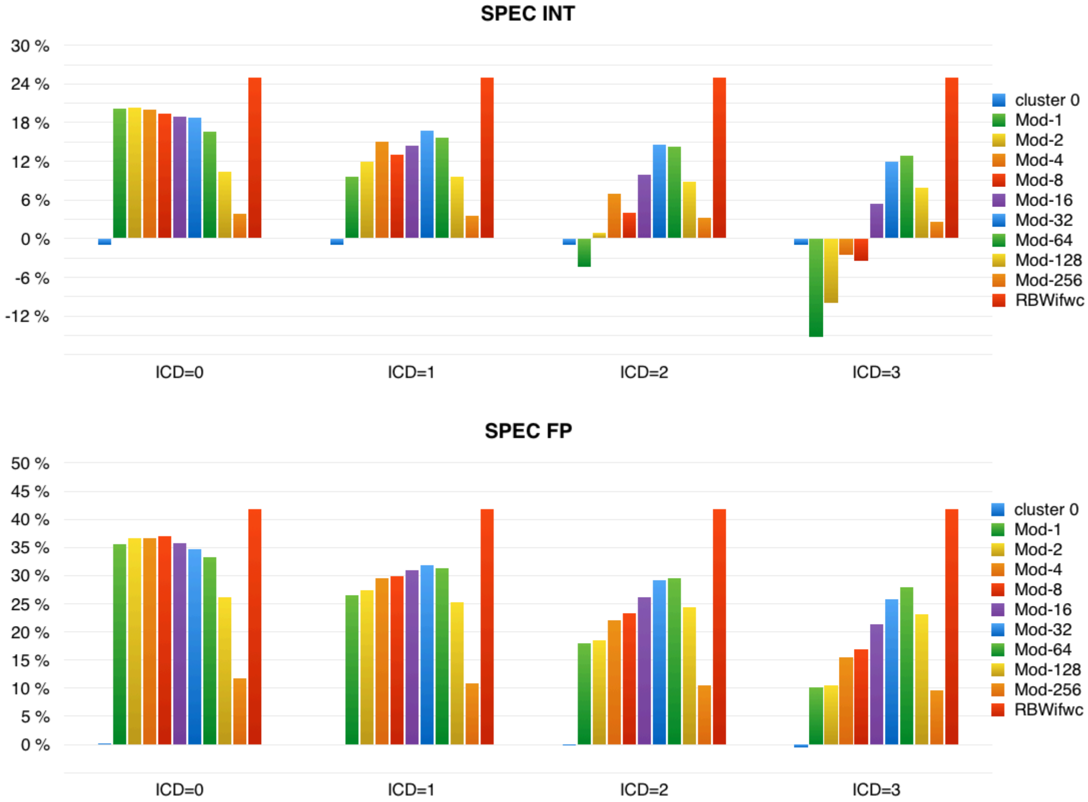


Figure 3.5: IPC gain over the baseline for clustered *RBWiiffccw* configurations (Figure 3.4).

respectively.

It can be observed in Figure 3.5 that when the intercluster delay is 2 cycles or more, the IPC is very sensitive to Mod-N steering. Figure 3.6 shows the IPC gain over the baseline, benchmark per benchmark, for the *RBWiiffccw* clustered scheme, assuming a 3-cycle intercluster delay, comparing Mod-32 and Mod-64 steering. For some benchmarks, Mod-32 outperforms Mod-64. For some other benchmarks it is the other way around.

Baniasadi and Moshovos proposed that, instead of having a fixed Mod-N, we could have an adaptive Mod-N trying to find the best N dynamically [BM00]. We tried an adaptive method similar to Baniasadi's, trying to identify dynamically the best Mod-N with N in $\{32, 48, 64, 80, 96\}$. After having executed 500k micro-ops under Mod-N steering, we try successively Mod-max$(32, N-16)$[8], Mod-N and Mod-min(A, B)[9] for 50k retired micro-ops each, counting the number of

---

[8] Mod-max(A, B) = Mod-N with N the largest of A and B

[9] Mod-min(A, B) = Mod-N with N the smallest of A and B

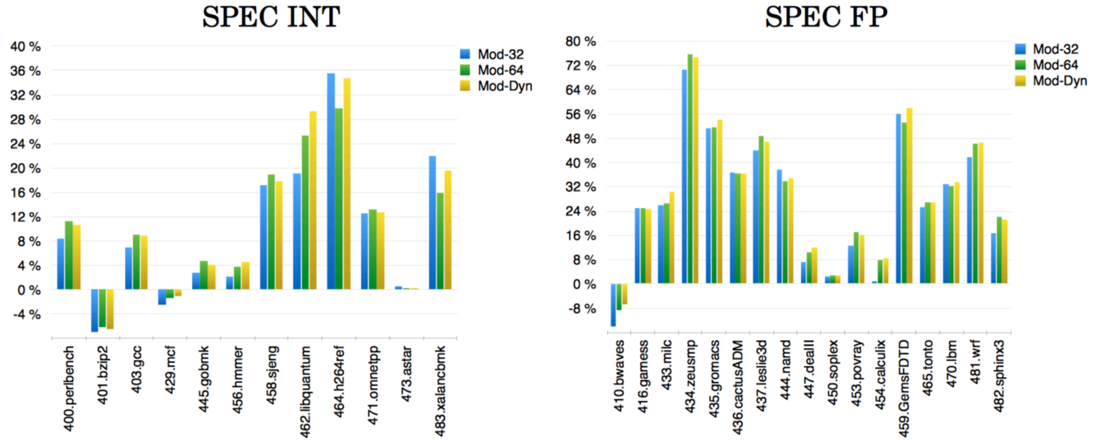


Figure 3.6: IPC gain over the baseline for a clustered *RBWiiffccw* configuration with an intercluster delay of 3 cycles: benchmark per benchmark comparison of Mod-32 steering, Mod-64 and adaptive Mod-N.

cycles. We choose the one with the best local performance, Mod-$N*$, and run under Mod-$N*$ for the next 500k micro-ops. We repeat this process periodically, every 500k micro-ops. Results for the adaptive method are shown in Figure 3.6.

On average, assuming an intercluster delay of 3 cycles, the adaptive steering slightly outperforms Mod-64 and Mod-32 and achieves an IPC gain of +14.1% over the baseline for the SPEC INT and +28.8% for the SPEC FP.

3.4.3 Analysis

Our findings should be contrasted with those of Baniasadi and Moshovos. They found that, among all Mod-N steering policies, Mod-3 was the best policy on average [BM00]. However they were considering 2-issue clusters.

We first explain with a simple analytical model why Mod-N with a large N is better for wide-issue clusters. Our analytical model is based on the empirical observation that the average ILP is roughly the square-root of the instruction window size [RF72, MSJ01, KS04]. We further assume a null intercluster delay. The square-root law models the fact that instructions that are ready for execution at a given instant are more likely to be found among the oldest instructions in the instruction window than among the youngest ones.

In the example of Figure 3.7, at the instant considered, the average ILP on cluster 0 is greater than on cluster 1. The ILP imbalance between the 2 clusters increases with the value of N. Because the instantaneous IPC is the minimum of the issue width and the ILP, if the per-cluster issue width is 2 instructions, the

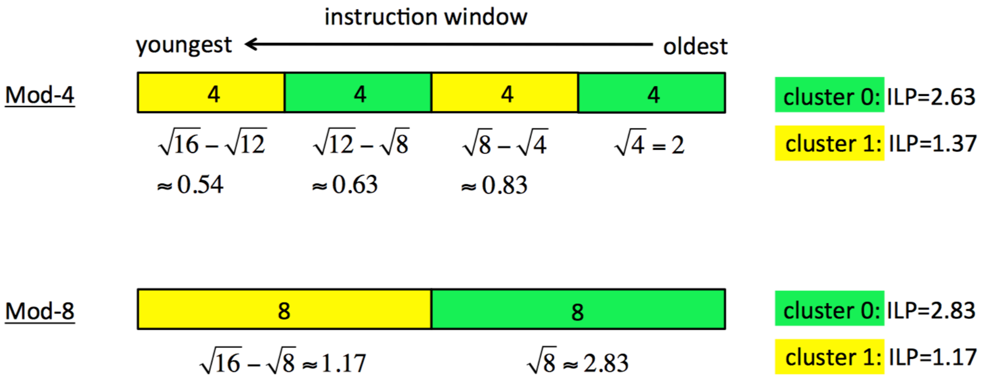

instruction window
youngest
oldest
Mod-4
4
4
4
4
$\sqrt{16}-\sqrt{12} \approx 0.54$
$\sqrt{12}-\sqrt{8} \approx 0.63$
$\sqrt{8}-\sqrt{4} \approx 0.83$
$\sqrt{4}=2$
cluster 0: ILP=2.63
cluster 1: ILP=1.37
Mod-8
8
8
$\sqrt{16}-\sqrt{8} \approx 1.17$
$\sqrt{8} \approx 2.83$
cluster 0: ILP=2.83
cluster 1: ILP=1.17

Figure 3.7: Illustration of ILP imbalance on 2 clusters, assuming a null intercluster delay and assuming the average ILP is the square root of the instruction window size. The upper example is for a Mod-4 steering policy, the lower example for Mod-8. In both examples, the instruction window holds 16 instructions.

IPC on cluster 0 is limited by the issue width, and Mod-4 outperforms Mod-8 (the total IPC is $2+1.37 = 3.37$ with Mod-4 and $2+1.17 = 3.17$ with Mod-8). If we increase the per-cluster issue width to 3 instructions instead of 2, the IPC on cluster 0 is not limited by the issue width, and both Mod-8 and Mod-4 yield the same IPC (hence Mod-8 outperforms Mod-4 if the intercluster delay is non null). There is roughly a square relation between the per-cluster issue width and the value of N beyond which ILP imbalance impacts performance significantly. This explains why the optimal N is much greater for wide-issue than for narrow-issue clusters.

To confirm this analysis, we simulated a quad-cluster back-end, each cluster being able to issue and execute 2 ALU or address micro-ops per cycle. The load queue can issue 2 loads per cycle. We assume an issue buffer partition of 15 micro-ops per cluster, so that the total issue buffer capacity is equivalent to the baseline. We assume 128 physical registers, as the baseline, but we removed register write specialization[10]. The other microarchitecture parameters are identical to the baseline.

Figure 3.8 shows the results of this experiment, for the SPEC INT only. When the intercluster delay is (unrealistically) null, Mod-N steering with $N \leq 4$ achieves a good ILP balancing. The gap between Mod-16 and Mod-32 is probably related to the size of the issue queue. The quad-cluster slightly outperforms the baseline thanks to the 8 ALUs (instead of 4 in the baseline). However, for N above 8, ILP imbalance impacts performance significantly because of the small

---

[10] Having more architectural registers than physical registers in a cluster partition may lead to deadlocks [STR02].

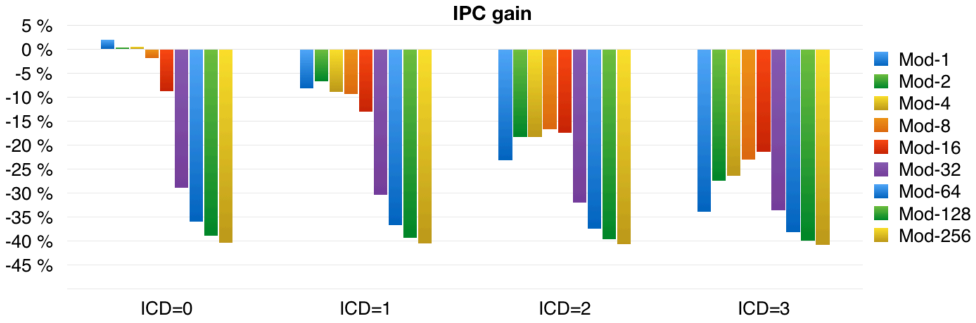


Figure 3.8: IPC gain over the baseline for a 4-cluster back-end. Each cluster can issue and execute 2 micro-ops per cycle. Only SPEC INT averages are shown.

per-cluster issue width.

With an intercluster delay of one cycle, as Baniasadi and Moshovos assumed in their study, the trade-off between ILP balancing and intercluster communications is at work, and the best steering policy is Mod-2. This is consistent with Baniasadi and Moshovos' finding that Mod-3 is the best Mod-N policy[11]. As the intercluster delay increases to 2 and 3 cycles, though, the best Mod-N steering becomes Mod-8 and Mod-16 respectively, but the IPC loss is important.

By contrast, with wide-issue clusters, ILP imbalance remains bearable for values of N up to 32~64, as shown in the previous section. As a result, there are few intercluster communications, which allows to tolerate longer intercluster delays.

3.4.4 Possible steps toward the proposed dual-cluster configuration

The dual-cluster microarchitecture we described is supposed to be the result of incremental microarchitecture modifications over several technology generations. However, going from a non-clustered microarchitecture to a clustered one introduces a performance discontinuity. Our results show that, to absorb the performance impact of the intercluster delay, the instruction window must be large enough.

However, clustering is what permits enlarging one of the components of the instruction window: the issue buffer. Yet, we would like an increase of microarchitecture complexity to be rewarded by a performance gain, at least on some applications.

---

[11] For us, the “2” in Mod-2 means two x86 CISC instructions, while Baniasadi and Moshovos considered RISC instructions.

The results in Figure 3.3 suggest a possible path toward the proposed wide-issue dual-cluster configuration:

1. Introduce clustering for the FP execution units only[12].
2. Enlarge the instruction window, except the issue buffer (reorder buffer, load/store queues, physical registers, MSHRs). This can be done progressively.
3. When the instruction window is large enough, introduce clustering for the INT execution units and increase the total issue buffer capacity.

The first two steps should benefit applications with characteristics similar to the SPEC FP benchmarks (Figure 3.3, configuration *RWf*).

3.5 Energy Considerations

The EPI is likely to be higher in the dual-cluster microarchitecture than in the single-cluster baseline. As explained in Section 3.1, if the microarchitecture is modified incrementally, the microarchitectural EPI increase can be hidden by the energy reductions coming from technology. Nevertheless, we provide in this section a few research directions for tackling the microarchitectural EPI increase.

3.5.1 Static EPI

The second cluster and larger instruction window substantially increase the back-end static power. The effect on the static EPI, however, is mitigated by the speedup brought by the dual-cluster and the larger instruction window. If the core (including the L2 cache) is powered off during long idle periods, the static EPI depends on the speedup: the higher the speedup, the lower the static EPI. However, not all applications have the same speedup (Figure 3.6).

For example, let us assume that the L2 cache represents 40% of the baseline core static power, the back-end 40% and the front-end 20%. Moreover, let us assume that the back-end static power of the dual-cluster is twice that of the baseline core, and the front-end static power 30% higher. For example, if the IPC is unchanged, the static EPI is multiplied by $0.4 \times 1 + 0.4 \times 2 + 0.2 \times 1.3 = 1.46$. But with a speedup of +20%, the static EPI is multiplied by only $1.46/1.20 = 1.22$.

A topic for future research is to find a way to turn the second cluster on and off dynamically depending on the expected speedup.

---

[12] Our Mod-64 policy steers to the other FP cluster every 64 *general* instructions. We did not evaluate the variant of Mod-N that steers to the other FP cluster every N floating-point instructions.

trace format	**avg micro-ops/trace**		**in-cluster values**	
	INT	**FP**	**INT**	**FP**
8 micro-ops 1 branch	5.6	6.6	25%	32%
12 micro-ops 2 branches	10.2	11.1	39%	42%
16 micro-ops 3 branches	14.6	15.2	48%	49%

Table 3.5: Percentage of values (i.e., not counting addresses) produced by a cluster that do not need to be sent to the other cluster, depending on the trace format.

3.5.2 Gating intercluster communications for reduced dynamic EPI

The dual-cluster microarchitecture has a higher dynamic EPI than the baseline single-cluster. Some of the extra dynamic EPI comes from larger shared structures with a higher bandwidth (DL1 cache, DL1 TLB, load/store queues,etc.). Intercluster communications also contribute to the increased dynamic EPI, in the issue buffer, in the register file and in the bypass network.

If we could identify micro-ops that do not need to forward their result to the other cluster, which we call in-cluster micro-ops, then the result bus segment going out of the cluster could be gated[13] when these micro-ops executes. Such gating would reduce the energy spent in the bypass network (charging and discharging the long result buses connecting the clusters) and writing the distant register file bank.

In particular, if the steering policy steers all the micro-ops from the same instruction to the same cluster, a micro-ops that writes a physical register not mapped to an architectural register produces a value which lives only within the instruction and does not need to be sent to the other cluster.

Roughly, 55% of all the micro-ops executed by the SPEC INT and 65% of all the micro-ops executed by the SPEC FP (compiled with `gcc -O2`) are micro-ops that produce a value, not counting address computations. On average, about 12% of these micro-ops do not write an architectural register. That is, 12% of the intercluster communications can be gated, on average, just by considering these micro-ops.

To identify more in-cluster micro-ops, a possible solution is to add some information in the trace cache (Section 2.3.3). We assume that all the micro-ops from the same instructions are put in the same trace, and that all the micro-ops in a trace are steered to the same cluster[14]. If two micro-ops in the same trace write the same architectural register (write-after-write dependency), the first micro-op is an in-cluster one. So when building the trace, in-cluster micro-ops are identified and this information is stored along with the trace in the micro-op cache (one bit per micro-op).

---

[13] By putting some tri-state buffers in high impedance state, or by clock-gating some latches.

Table 3.5 shows the percentage of in-cluster values for various trace formats. As expected, the longer the trace, the more in-cluster values can be identified. For instance, with traces containing about 10 micro-ops on average, about 40% of the values (i.e., not counting addresses) produced by a cluster do not need to be sent to the other cluster, meaning that the corresponding intercluster communications can be gated.

Gating some intercluster communications requires to keep the physical register file content consistent, as each physical register partition has two copies (one in each cluster). When an in-cluster micro-op executes, it writes in its local register bank and updates its local scoreboard, but it does not update the distant bank and scoreboard. A branch misprediction can result in an inconsistent state.

A possible solution is to retire traces from the reorder buffer only when all the micro-ops in the trace have been executed successfully. When a branch misprediction is detected, the in-cluster micro-ops that are before the branch in the same trace are still in the reorder buffer. The branch misprediction recovery logic must find these micro-ops, get their destination physical registers Pi, and inject directly in the local issue buffer some special micro-ops `mov Pi,Pi` that send the value of Pi to the distant cluster. These `mov` micro-ops execute while the correct-path instructions go through the front-end pipeline stages.

An interesting direction for future research is the possibility to dedicate some execution ports and/or some physical registers to in-cluster micro-ops, which would decrease the hardware complexity of the bypass network and register file [VM97, RJSS97].

3.6 Summary

In this chapter we explored the potential of the Out-of-Order clustered microarchitecture. We described the limits of actual microarchitecture and we tried to analyze costs and benefits of clustering. Future microarchitectures will require more functional units, bigger registers file, more hardware. The importance of accelerating the single core is still one of the main goal of the research community, also because applications show an instruction level parallelism that the researcher are trying to exploit. In this chapters we analyzed what is proposed in previous research about clustered microarchitecture and we offered an overview of how the cluster concept can be applied to future microarchitecture. The proposed solution tries to solve some of actual problems in modern superscalar back-end. We describe our idea of how to use concepts like clustering, register write specialization, and steering policy in order to solve the technology limits of single cores and understanding how future processors may look like in the future.

---

[14] If a trace contains on average 10 instructions, a policy equivalent to Mod-60 steers 6 traces to a cluster, the next 6 traces to the other cluster, and so forth.

Chapter 4

Exploiting loops for reducing energy in a superscalar out-of-order core

The technological evolution in the last years has revolutionized the way to design computer's microarchitectures. The market requirements changed over time: if some years ago priority was mainly to increase performance, today new challenges move the research. One of the main concerns is the need of new power saving solutions, in order to take advantage of the miniaturization and the creation of new classes of devices (i.e. smartphones, tablets, smartwatches) that did not exist 20 years ago and that require high performance but small energy consumption.

In addition, power efficiency has represented an important obstacle to traditional methods of performance improvement, limiting the benefits that can be obtained with more aggressive designs. Solutions like multi-thread and multi-core architectures allow to circumvent some problems, but not all applications are suitable for a multi-core processor. For this reason it is important to search new power-saving solutions for general-purpose architectures.

We have mentioned the exploitation of instruction loops as a useful solution for reducing power consumption in both front-end and back-end (Section 2.5.1 and 2.5.3). During the program execution, the number of instructions dynamically executed is (considerably) greater than the number of static instructions. Program execution presents instructions or blocks of instructions which are executed in loops. The detection of these loops allows to turn off (if possible) some hardware components, saving energy.

In this chapter we will describe how it could be possible to reduce the energy consumption in a superscalar Out-of-Order (OoO) core by exploiting the "loopy" behavior of applications, using a special cache memory called *loop buffer*. Our approach is to take advantage of this special cache, reducing the total number of instructions executed in the OoO core (and the pressure on OoO core components) and reducing the total number of memory accesses in STQ or L1-cache for load micro-ops during program execution.

The loop cache helps gating the front-end during the loop execution. As we described in Chapter 3, in the coming years it could be possible (and desirable) to increase the instruction window and further exploit the ILP offered by applications. This scenario makes room for new solutions of energy saving in the Out-of-Order core. In this chapter, we will describe our solutions and the impact on two configurations: one close to an actual commercial processor, and one more aggressive.

4.1 Baseline and Experimental Setup

For the study of loops we used the same in-house microarchitecture simulator based on Pin [LCM$^+$05] described in Section 3.2.

The method described in this section is based on the exploitation of instruction loops. Only a subset of **SPEC2006 Suite** [Hen06] contains a sufficient number of loops. For this reason, and in order to demonstrate the impact of the proposed technique, we also used the **CORTEX Suite** [TGT$^+$14] and the **The San Diego Vision Benchmark Suite** (SD-VBS) [VAJ$^+$09], compiled with `gcc -O2` and executed with the reference inputs.

The baseline microarchitecture is loosely modeled after the Intel Haswell [Int16a]. Details of the baseline are reported in Table 3.2: we used the same baseline already described. Two parameters are added:

- the loop buffer size, equal to 64 for the realistic configuration that emulates the sizing of Intel Haswell and equal to 128 for an aggressive configuration for future implementations

- the value of MinIter (minimum number of loop iterations), equal to 30 and calculated trying to replicate the real loop buffer behavior in recent commercial microarchitecture

Our baseline features the Store-Set memory dependence predictor [CE98] and two queues for memory disambiguation a *Store Queue* (STQ) and a *Load Queue* (LDQ).

We simulate back-to-back execution with two pipeline stages between the issue stage and the first execution stage. These pipeline stages correspond to reading the registers. The hit/miss prediction for loads is obtained with a 4-bit global counter predictor, as in the Alpha 21264 [Kes99]. In case of a DL1 miss that was predicted to hit, a *DL1-miss replay* occurs, and all the micro-op issued is the shadow of the load (i.e., in the three cycles after and including the load issue cycle) are rescheduled, including those independent of the load.

Like recent Intel processors, our baseline also features a loop buffer, which we describe in the next section.

4.2 Loop Buffer and Loop Detector

In modern processors, the loop buffer is a small memory containing instructions decoded into micro-ops[1]. The main purpose of the loop buffer is to save energy while executing loops, bypassing large parts of the front-end consuming substantial energy (branch predictor, instruction cache, decoder).

Though the loop buffer is not the focus of this study, it is included in our baseline microarchitecture and we leverage it throughout this chapter. As the implementation details of loop buffers in recent commercial processors are not publicly documented, this section provides a description of our loop buffer and loop detector implementation.

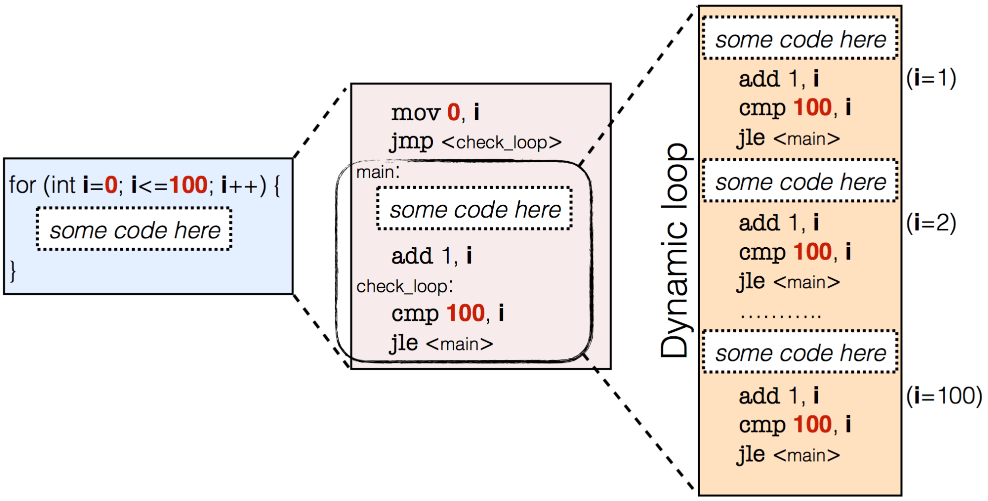

for (int i=0; i<=100; i++) {
some code here
}
mov 0, i
jmp <check_loop>
main:
some code here
add 1, i
check_loop:
cmp 100, i
jle <main>
Dynamic loop
some code here
add 1, i
cmp 100, i
jle <main>
(i=1)
some code here
add 1, i
cmp 100, i
jle <main>
(i=2)
....
some code here
add 1, i
cmp 100, i
jle <main>
(i=100)

Figure 4.1: Left: inside the while statement we have code that iterates 100 times (It may contains other loops). Center: the assembly version in which the branch is clearer. The loop circled, during the dynamic execution, will be detected and stored in the loop buffer. Right: the dynamic loop.

---

[1] Following Intel nomenclature

4.2.1 Description

We define a *dynamic loop* as a periodic sequence of dynamic instructions. More precisely, we detect periodicity by looking for the shortest repeating sequence of instruction PCs (PC = program counter = address of the instruction), called a *period*. The instructions corresponding to the PCs in a period constitute the *loop body*. It should be noted that dynamic loops generally come from program loops, but all program loops do not generate dynamic loops (as we define them). For example, a program loop may contain one or several weakly-biased branches preventing it from generating a dynamic loop. In the rest of the chapter, *loop* is used for *dynamic loop*.

An example of dynamic loop detected follows. The Figure 4.1 is a simplified view of the example, showing the code that produces the dynamic loop described above. The code in Listing 4.1 has an outer loop that iterates many times and an inner loop with 2 iterations. The corresponding dynamic loop is in Listing 4.2: this is the sequence of dynamic instructions sent to the back-end by the front-end. We want to show how a *dynamic loop* looks like and how the loop detector works. In the example, each instruction is numbered with an incremental number N.

Listing 4.1: Section of code that contains a loop

```
4004fa:  movl      $0x0,-0x4(%rbp)
400501:  jmp       40051a
400503:  movl      $0x2,-0x8(%rbp)
40050a:  jmp       400510
40050c:  addl      $0x1,-0x8(%rbp)
400510:  cmpl      $0x0,-0x8(%rbp)
400514:  jg        40050c
400516:  addl      $0x1,-0x4(%rbp)
40051a:  cmpl      $0x63,-0x4(%rbp)
40051e:  jle       400503
400520:  pop       %rbp
```

In this example, the detected loop starts at $N = 1$, it ends at $N = m$ and iterates 100 times. The loop body starts with the instruction at address `0x400503` and it ends with the instruction at address `0x40051e` (Listing 4.1).

Listing 4.2: Dynamic sequence of executed micro-ops

```
N:     instruction
 0:     movl     $0x0,-0x4(%rbp)
 1:     jmp      40051a            *outer start*
 2:     movl     $0x2,-0x8(%rbp)
 3:     jmp      400510            |inner start |
 4:     addl     $0x1,-0x8(%rbp)
 5:     cmpl     $0x0,-0x8(%rbp)
 6:     jg       40050c            |inner iteration (iter 1)|
 7:     addl    $0x1,-0x8(%rbp)
 8:     cmpl    $0x0,-0x8(%rbp)
 9:     jg      40050c            |inner exit (iter 2)|
A:      addl    $0x1,-0x4(%rbp)
B:      cmpl    $0x63,-0x4(%rbp)
C:      jle     400503            *outer iteration*
D:      movl    $0x2,-0x8(%rbp)
E:      jmp     400510            |inner start |
F:      addl    $0x1,-0x8(%rbp)
 10:    cmpl    $0x0,-0x8(%rbp)
 11:    jg      40050c            |inner iteration (iter 1)|
 12:    addl    $0x1,-0x8(%rbp)
 13:    cmpl    $0x0,-0x8(%rbp)
 14:    jg      40050c            |inner exit (iter 2)|
 15:    addl    $0x1,-0x4(%rbp)
 16:    cmpl    $0x63,-0x4(%rbp)
 17:    jle     400503            *outer iterations*
.............. (more iterations)  .................
m-1:    cmpl    $0x63,-0x4(%rbp)
m:      jle     400503            *outer exit (iter 100)*
m+1:    pop     %rbp
```

Our Loop Detector (LD) has two main parameters: the maximum loop body size, *MaxBody*, and the minimum number *MinIter* of iterations for being considered a loop. In the example, if we set $MinIter = 3$, the loop body presents an inner loop that is not detected because it does not iterate at least for $MinIter = 3$ times. The loop detector, therefore, can detect the dynamic loop regardless of the presence of the inner-loop. The latter is considered part of the periodic sequence of instructions that iterates, in this example, 100 times. If the value of MinIter is less that 100, the outer loop is detected. If the MaxBody value is less than the loop body size[2], all the decoded instructions in the loop body will be stored in the loop buffer.

There is a Loop Detector Table (LD-Table) for monitoring recent backward jumps (as a loop must contain at least one backward jump). Our LD-Table has 8 entries and is fully associative. Each LD-Table entry contains:

- an End-of-Loop Program Counter (EOLPC) which is the search key, a valid bit
- a body size B
- a signature S
- a current body size B'

---

[2] The loop cache stores micro-ops. The body size corresponds to the number of micro-ops in the loop body

- a current signature S'
- an iteration count I
- a First-Iteration Bit (FIB)

For each decoded instruction, the current body size B' of each valid LD-Table entry is incremented by the number of micro-ops of that instruction, and the current signature S' is updated.

The signature is a hash of the instructions PCs in the loop body[3]. When the current body size B' of any valid LD-Table entry exceeds MaxBody, we close that entry by resetting its valid bit.

At decode, when a branch instruction jumps backward, it is considered a potential end-of-loop, and we search the LD-Table with the PC of the branch. If no matching EOLPC is found, we create an entry (e.g., by reusing a closed entry): the EOLPC is set to the branch PC, the valid bit and first-iteration bit are set, and all the other fields are reset. Otherwise, if there exists a valid entry whose EOLPC matches the branch PC, there are two cases:

1. if the FIB is set, or if $S' = S$ and $B' = B$, we increment the iteration count I
2. otherwise we reset I, we copy S' into S and we copy B' into B

Then, in both cases, we reset S', B' and the FIB.

When the iteration count I equals MinIter, we have detected a loop: we close all the other LD-Table entries, and the *loop capture mode* is entered. The next B micro-ops generated by the decoder constitute the loop body and are stored in the loop buffer, along with the branch predictions for all the branches inside the loop, so that we can detect the loop exit.

When the loop buffer provides the instructions, then branches are statically predicted. All the dynamic instances from the same static branch are predicted identically on each loop iteration. Once the loop body is loaded into the loop buffer, the front-end pipeline stages are flushed and instruction fetching is gated. From now on and until loop exit, the execution is in *steady loop mode*, micro-ops being delivered to the register rename stage directly from the loop buffer.

It should be noted that, in the case of nested loops, if the inner loop iterates a small (fixed) number of times (less than MinIter), the inner loop is unrolled by our loop detection algorithm and can be part of the body of the outer loop. However, once a loop has been detected, even if the loop body is much smaller

---

[3] The longer the signature, the less likely for two distinct loop bodies to have the same signature. We assume 64-bit signatures. We update the current signature by rotating it 1 bit to the left and applying an XOR with the new instruction PC.

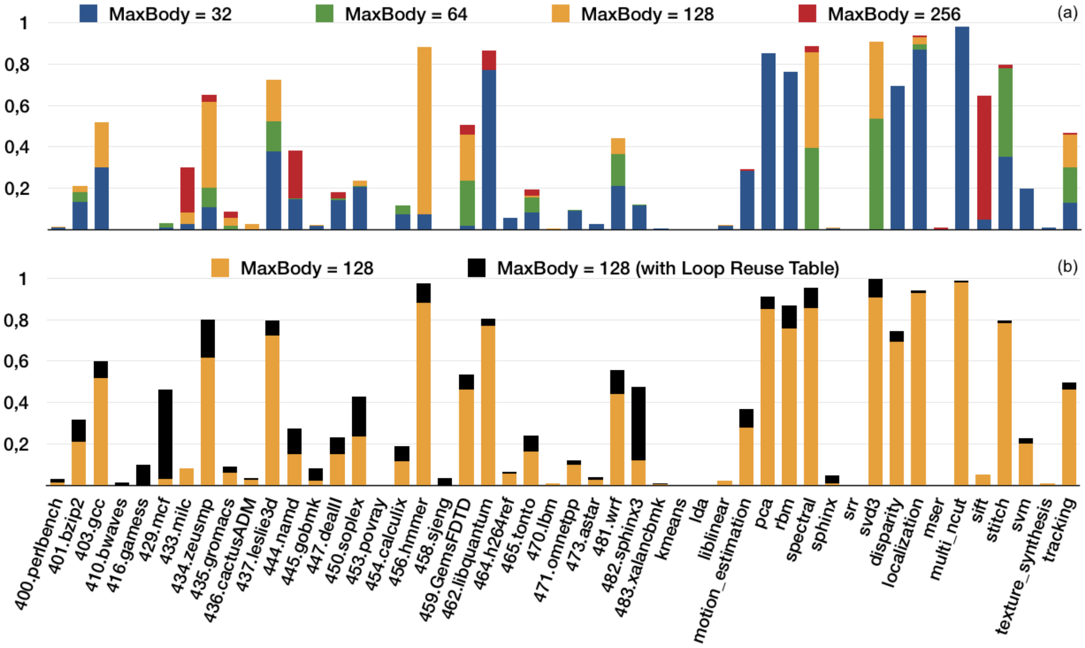


Figure 4.2: Loop coverage, assuming $MinIter = 30$. Top graph (a): impact of the loop buffer size. Bottom graph (b): impact of using a Loop Reuse Table, with $MaxBody = 128$.

than the loop buffer, we do not unroll the loop. That is, the loop buffer contains exactly one loop body.

4.2.2 Loop buffer size

In practice, the MaxBody parameter is the loop buffer size. A large loop buffer allows to capture large loop bodies, but it consumes more energy.

We define *loop coverage* as the fraction of micro-ops dispatched in steady loop mode. For instance, a coverage of 60% means that 60% of the micro-ops executed and retired came from the loop buffer, and the remaining 40% came from the instruction cache through the decode stage.

Figure 4.2 shows loop coverage, assuming $MinIter = 30$. The top graph shows how coverage varies with MaxBody (the bars are cumulative, e.g., the yellow part shows the contribution to coverage brought by a loop buffer of 128 micro-ops that is not captured with a loop buffer of 64 micro-ops). Different benchmarks have very different behaviors.

With a loop buffer size of 64 micro-ops (the loop buffer size in recent Intel processors), 7 of our benchmarks have a loop coverage greater than 50%: `437.leslie3d`, `462.libquantum`, `pca`, `rbm`, `disparity`, `multi_ncut`, and `stitch`.

Coverage increases quite substantially with a loop buffer size of 128 micro-ops for 8 of our benchmarks, namely, `403.gcc`, `434.zeusmp`, `437.leslie3d`, `456.hmmer`, `459.GemsFDTD`, `spectral`, `svd3` and `tracking`.

It is possible to increase loop coverage further with a Loop Reuse Table (LRT). The loop detection algorithm previously described waits until MinIter iterations have been seen before entering the loop mode. But if a loop had already been encountered in the past and iterated more than MinIter times, it is likely to iterate more than MinIter times again. By recording loop information (EOLPC and body size) in the LRT for loops recently detected, and searching the LRT on backward jumps, we can enter the loop mode immediately in case of LRT-hit.

The impact of the LRT is shown in the bottom graph of Figure 4.2, for $MaxBody = 128$. The LRT increases loop coverage substantially for some benchmarks, particularly `429.mcf`, `450.soplex` and `482.sphinx3`.

In this study, we consider two baselines: a *conservative* baseline with a 64-micro-op loop buffer, like the Intel Skylake [Int16a], and an *aggressive* baseline with a 128-micro-op loop buffer associated with a 16-entry LRT. In the rest of the chapter, to save space, we will sometimes show results only for “loopy” benchmarks, which we define as the benchmarks having a loop coverage at least 10% with the aggressive configuration.

4.2.3 Tuning MinIter

Unlike the loop buffer size, the details of the loop detection algorithms implemented in commercial processors are generally undisclosed. We believe our algorithm is a reasonable one. However, we must determine a realistic value for parameter MinIter.

On the one hand, a high value for MinIter may hurt loop coverage, hence the energy savings expected from a loop buffer. On the other hand, if MinIter is low, the LD may trigger the loop mode for loops iterating a small number of times. However, every time the loop mode is entered, we eventually pay the performance and energy cost of a loop exit, which is that of a branch misprediction.

High-performance branch predictors, such as TAGE [SM06], are generally able to predict correctly the loop exit for loops with few iterations. However, once the steady loop mode is entered, the main branch predictor is not being used, and the loop exit is necessarily mispredicted. Moreover, in loop mode, the global branch history used by the TAGE predictor is not updated. Upon the loop exit, the global branch history is repaired from the last valid checkpoint we have for the global history, which does not contain branches executed in loop mode. This may generate extra branch mispredictions after the loop exit.

For this study, we found experimentally that $MinIter = 30$ provides good loop coverage with negligible impact on branch mispredictions compared to other

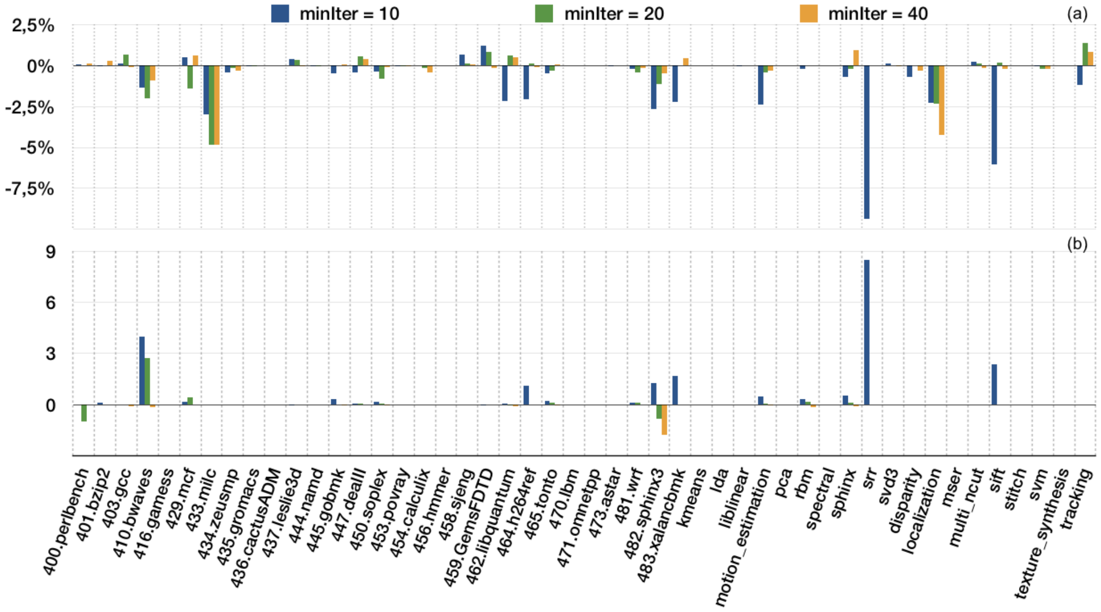


Figure 4.3: Varying the size of MinIter on the conservative configuration ($MaxBody = 64$). On top (a) the speedup compared with a conservative baseline with $MinIter = 30$. On bottom (b) the different of MPKI between various MinIter and the conservative baseline with $MinIter = 30$. Benchmarks vertically aligned.

MinIter values. The bottom graph of Figure 4.3 shows extra Mispredictions per 1000 Instructions (MPKI) for $MinIter = 10, 20, 40$. More precisely, the graph gives the MPKI of a given configuration minus the MPKI of the conservative baseline, which assumes $MinIter = 30$. The top graph shows speedups with respect to the conservative baseline.

Clearly, $MinIter = 10$ increases the MPKI substantially and negatively impacts performance (e.g., `srr`, `sift`, `410.bwaves`, `464.h264ref`, `482.sphinx3`, `483.xalancbmk`). $MinIter = 20$ is close to be a good tradeoff, except for benchmark `410.bwaves`. With $MinIter = 30$ and above, the impact on the MPKI is negligible.

It can be observed that, for benchmarks `433.milc` and `localization`, there are speedup variations despite no MPKI change. These performance variations come from the varying loop coverage. Although the loop buffer is mostly intended for energy, it also increases the micro-ops dispatch throughput[4].

---

[4] Our instruction fetch mechanism fetches a single instruction cache line per cycle. Loops with a small loop body crossing a cache line boundary may have their instruction throughput limited by this constraint. The loop buffer is not impacted by cache line boundaries and can always deliver 8 micro-ops per cycle.

4.3 Redundant Micro-Op Removal

The loop buffer is primarily intended for reducing the energy consumption in the front end. We propose to exploit loop behaviors further to reduce energy consumption in the back end. In this section, we take advantage of the fact that some loop bodies contain *redundant* micro-ops producing the same result on every loop iteration. If, instead of executing a redundant micro-op again and again, we could just reuse its result, this would allow to remove the micro-op completely and save some energy in the OoO engine.

In this section we describe the hardware mechanism used to detect and remove redundant micro-ops, and we evaluate its effectiveness.

4.3.1 Proposed mechanism

A good compiler generally tries to hoist loop-invariant code outside the loop. However, the compiler cannot always detect and move the loop-invariant work, for various reasons (branches directions unknown at compile time, ambiguous pointers, etc.). Hence the dynamic loops identified at execution time may contain some redundant micro-ops.

An obvious case of redundancy is when the registers read by a micro-op are not modified by the loop, i.e., they are constant throughout the loop execution. We call this kind of micro-op *primary* redundant micro-ops. Another case of redundancy is when a micro-op depends on a primary redundant micro-op: this is a *secondary* redundant micro-op. We may have *tertiary* redundant micro-ops depending on secondary redundant micro-ops, and so forth. This leads to the following recursive definition: a micro-op is redundant if its source registers are not modified by the loop or are produced by redundant micro-ops.

It should be noted that primary redundant micro-ops always produce the same value on every iteration in steady loop mode, but it may take several loop iterations before a non-primary redundant micro-op produces a constant value[5].

The mechanism we propose detect redundant micro-ops during the first few iterations of the steady loop mode removing them one by one, progressively: first the primary micro-ops, then the secondary ones, and so on. We found experimentally that, for most loops, two loop iterations are sufficient to remove all the redundant micro-ops (for a few loops with a large body, up to three iterations are necessary). When no new redundant micro-op is detected in an iteration, redundant micro-op detection is turned off for the rest of the steady loop mode.

---

[5] Consider for instance the case of a secondary redundant micro-op Y depending on a primary redundant micro-op X, such that Y occurs before X in the loop body. On the first iteration, Y reads a value that was produced before the loop. On subsequent iterations, Y reads the (constant) value produced by X.

Once a redundant micro-op has been removed from the loop body, it no longer generates any dynamic micro-op. Detecting and removing redundant micro-ops entails a little extra energy consumption during the first two or three iterations. However if the loop iterates more than a few times, this initial energy overhead is more than compensated by the energy saved from no longer sending the micro-op to register renaming, to the instruction scheduler and to the execution units.

Load/store micro-ops require special care and we explain later in this section how to deal with them.

4.3.2 Identification of redundant micro-ops

Redundant micro-op detection and removal can be implemented by modifying register renaming. When a redundant micro-op is removed, the physical register holding the value produced by the micro-op is kept alive for the rest of the steady loop mode, and micro-ops needing the value obtain it by reading that register.

The source operand of a micro-op is considered constant if the associated architectural register is not modified by any previous instruction belonging to the loop, or if it is defined by a micro-op already identified as redundant (Figure 4.4).

Each loop buffer entry is extended with extra information. A bit (Cd) indicating whether the micro-op is redundant, two bits (Cs), one per source operand, to indicate whether the associated operand is constant, the destination physical register identifier (Pd) of the last executed instance of that micro-op if the micro-op is redundant, and one physical register identifier (Ps) per source operand, to locate constant values.

Both Cd and Cs are reset during the loop capture mode. The identification of redundant micro-ops is carried out progressively and in parallel with register renaming. When a micro-op is identified as redundant, it keeps this state until the loop exit.

We introduce a *Lock-Bit vector* in the register renamer. There is one lock bit per architectural register. The lock bit for an architectural register is set if and only if the register is known to contain a constant value. All the lock bits are set before entering the loop capture mode. During loop capture, for each micro-op inserted into the loop buffer that writes an architectural register, the lock bit corresponding to that register is reset. When the capture mode ends, the Lock-Bit vector has a bit reset for every architectural register updated inside the loop.

In steady loop mode, for each micro-op source operand, if Cs=0 and lock bit=1, we set the Cs bit, and the source physical register identifier obtained from the rename table is copied into the Ps field. The Cd bit is updated by ANDing the two Cs bits. If Cd becomes equal to one (i.e., all source operands are constant,

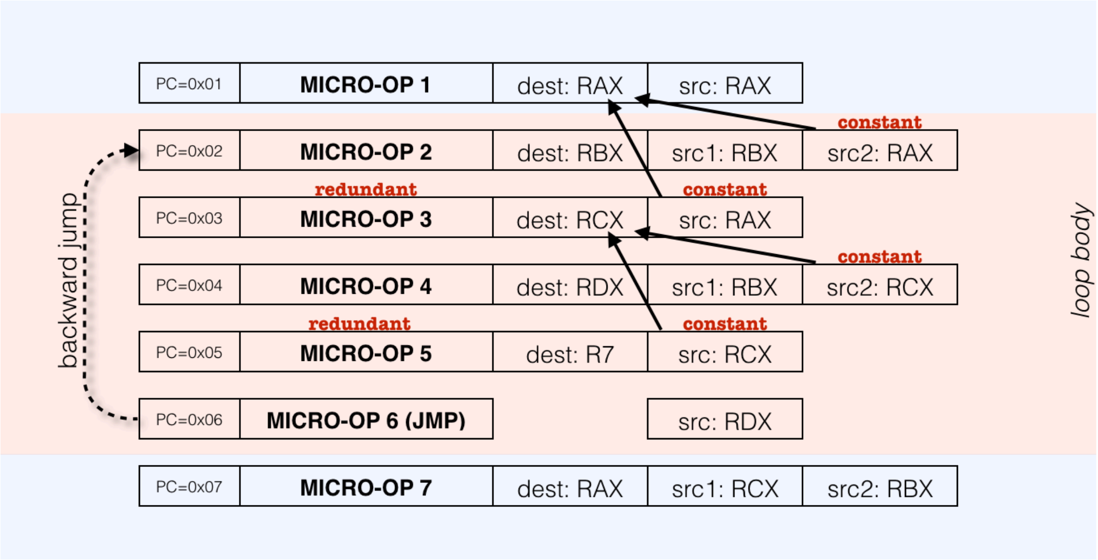

PC=0x01
MICRO-OP 1
dest: RAX
src: RAX
constant
PC=0x02
MICRO-OP 2
dest: RBX
src1: RBX
src2: RAX
redundant
constant
PC=0x03
MICRO-OP 3
dest: RCX
src: RAX
constant
PC=0x04
MICRO-OP 4
dest: RDX
src1: RBX
src2: RCX
redundant
constant
PC=0x05
MICRO-OP 5
dest: R7
src: RCX
PC=0x06
MICRO-OP 6 (JMP)
src: RDX
PC=0x07
MICRO-OP 7
dest: RAX
src1: RCX
src2: RBX
backward jump
loop body

Figure 4.4: Identification of constant source operands and redundant micro-ops. Red background for loop micro-ops, blue for the rest. The register `rax` is modified only outside the loop, and it is considered as a constant source for micro-ops at address `0x02` and `0x03`. As `rax` is the only source operand for micro-op 3, this micro-op is redundant. The register `rcx` is considered a constant source operand until the next modification (not present in this example) or the loop exit. Micro-ops at address `0x04` and `0x05` consider `rcx` a constant source operand. The latter (micro-op 5) is redundant because the only source operand (`rcx`) is modified by a redundant micro-op (at address `0x03`). The physical registers holding the value produced by micro-ops 3 and 5 are kept alive until the loop exit. For simplicity, we illustrate one micro-op for each decoded instruction (one micro-op = one PC).

the micro-op is redundant), the Pd field is updated with the identifier of the obtained from the register rename, and the destination lock bit is set, otherwise the lock bit is reset. Once the Cd is set, subsequent dynamic instances of the micro-op are completely removed from execution.

When no new redundant micro-op has been detected during a loop iteration, the detection of redundant micro-ops is turned off.

4.3.3 Modification of register renaming

The physical register holding the result of a redundant micro-op must be kept alive during the steady loop mode.

Register renaming is modified as follows. For each source operand, if the Cs bit is set, the Ps identifier is used instead of reading the rename table. Before overwriting an entry of the rename table, the old physical register is read (as usual), along with the lock bit. If the lock bit is 0, the old physical register is put into the ROB, as usual. If the lock bit is 1, this physical register holds the value produced by a redundant micro-op and must be kept alive, therefore it is not put into the ROB and will be released after the loop exit.

The trickiest part is repairing the state of the rename table upon loop exit (most of the time, on a branch changing its direction). First, the rename table and the free list are repaired using the check-pointed state, as for a normal branch misprediction. If no redundant micro-ops had been removed from the loop, we are done.

If some redundant micro-ops had been removed, the rename table and the free list must be repaired further. We use a *Last-Bit vector*, identical to the Lock-Bit vector, with one bit per architectural register. The *last* bit is set by the last valid micro-op in the loop writing this architectural register. The micro-ops in the loop buffer are numbered from 0 to $B-1$, with B the loop body size. Let us assume the loop exit is triggered by micro-op E, i.e., it is the last valid micro-op executed in loop mode. Initially, all the *last* bits are reset. We scan the B micro-ops in *reverse* order, starting from micro-op E, that is, $E, E-1, E-2, \ldots, 0, B-1, \ldots, E+2, E+1$. The rename table and the Lock-Bit vector are accessed using the destination architectural register identifier of the micro-op. There are 4 cases:

- **last=0 and micro-op non redundant**: set $last = 1$
- **last=0 and micro-op redundant**: set $last = 1$; if lock=0 and if the physical register in the rename table entry is different from Pd, put that register in the free list and write Pd in the rename table
- **last=1 and micro-op non redundant**: do nothing
- **last=1 and micro-op redundant**: put Pd in the free list

To minimize the restoration time, the reverse scan processes several micro-ops in parallel, e.g., 8 micro-ops per cycle. Instruction fetching can resume just after the loop exit is triggered, and the restoration of the register rename table and free list begins while instructions progress through the frond-end pipeline stages. However, instructions cannot pass through register renaming until the restoration is finished, and the front-end may have to stall if the loop body is large.

4.3.4 Loads and stores

In principle, one may consider removing redundant loads and stores. However there are some difficulties here. In this section, we describe a solution for removing redundant loads, but we do not try to remove redundant stores: **all the stores**

in a loop are executed normally. The main reason for not removing stores is the memory consistency model: even if a store always write the same value at the same address on every loop iteration, another core may access that address, and removing the store would change the program semantics.

A redundant load accesses the same address on every iteration. Removing redundant loads is possible, but we must have a mechanism for detecting memory conflicts (within the loop or with other cores).

To detect conflicts, we introduce a small 8-entry fully-associative Redundant Load Table (RLT). The RLT is initially empty and is used only during the loop mode. When a redundant load is removed from the loop body, an RLT entry is allocated for that load, and the last dynamic instance of the load writes the load's physical address in the RLT entry. If the RLT is full, preventing to allocate an entry, the load is not removed and executes normally.

The function of the RLT is the same as the load queue. In steady loop mode, when a store retires from the ROB, the RLT is searched with the store address (in parallel with the load queue, for usual memory disambiguation). If the address is not in the RLT, nothing happens (except if the load queue signals a memory order violation). If the address is in the RLT, this is an *RLT-trap*: a loop exit is triggered, with the retiring store as the loop exit point.

RLT-traps are costly, and we must minimize their occurrences. Indeed, just after an RLT-trap, we generally re-enter the same loop immediately, and we want to prevent the RLT-trap to occur again. Each RLT entry contains a loop-buffer pointer to the associated load, from where the load's PC can be obtained to train the store-set memory dependence predictor. A load can be removed only if it is predicted to be independent of any store in the store queue.

4.3.5 Simulation Results

The top graph of Figure 4.5 gives (for "loopy" benchmarks) the fractions of non-redundant loop micro-ops, redundant loop loads, and other redundant loop micro-ops, for the baseline loop buffer and the aggressive loop buffer. On our benchmarks, the fraction of loop micro-ops that are redundant is variable and ranges from 0 to about 50% (`svd3`). For 9 benchmarks, the fraction of loop micro-ops that are redundant is at least 10%: `401.bzip2`, `403.gcc`, `444.namd`, `450.soplex`, `456.hmmer`, `471.omnetpp`, `spectral`, `svd3` and `localization`. Loads represent a significant fraction of redundant micro-ops for some benchmarks, particularly `456.hmmer` (aggressive loop buffer), `471.omnetpp`, `spectral`, `svd3` and `localization`.

Of course, having some redundant loop micro-ops is interesting only if loop coverage is significant. The bottom graph of Figure 4.5 gives the fractions of non-loop micro-ops, non-redundant loop micro-ops and redundant loop micro-ops.

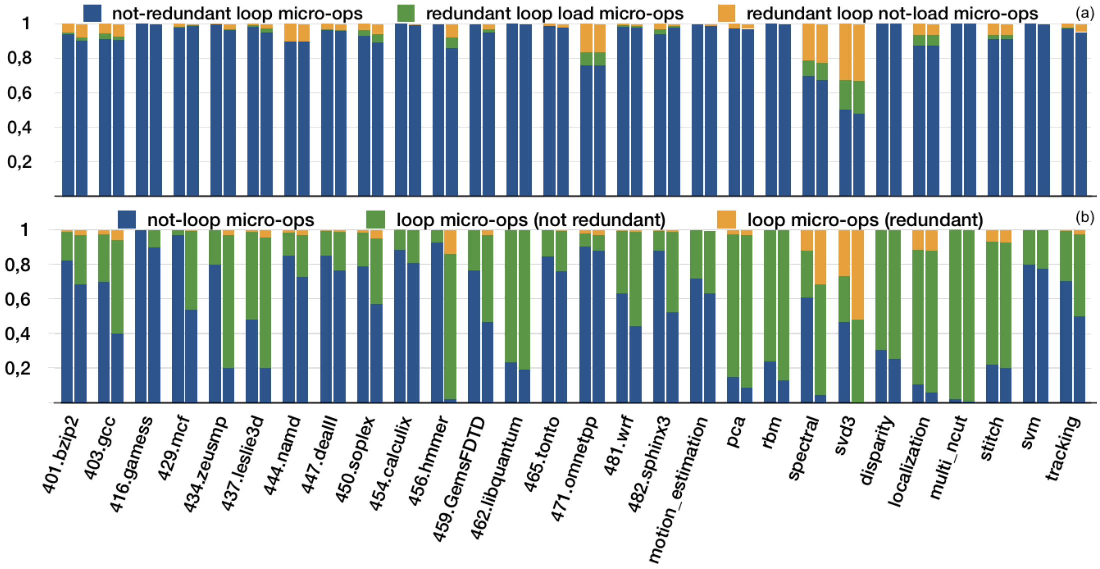


Figure 4.5: Top graph: fractions of non-redundant loop micro-ops, redundant loop loads, and other redundant loop micro-ops. Two bars per benchmark: conservative (left bar) and aggressive loop buffer (right bar). Bottom graph: fractions of non-loop micro-ops, non-redundant loop micro-ops and redundant loop micro-ops.

For the four benchmarks `456.hmmer` (aggressive loop buffer), `spectral`, `svd3` and `localization`, redundant loop micro-ops represent a substantial fraction of the total micro-ops. For other benchmarks, the fraction of redundant loop micro-ops is less than 10%. On a few benchmarks (`456.hmmer`, `spectral`, `svd3`), the aggressive loop buffer increases the fraction of redundant loop micro-ops quite significantly.

We can conclude from these experiments that redundant micro-op removal will benefit only the most “loopy” applications. But for these “loopy” applications, this will reduce the energy consumption substantially. Loopy applications already exploit the loop buffer to cut energy consumption in the front-end, which means that most of the core energy is spent in the back end: if we can remove 50% of micro-ops, as on benchmark `svd3` (128-entry loop buffer), the core’s dynamic energy consumption should be roughly halved.

Figure 4.6 shows the speedups for the conservative and aggressive baselines. The benchmark `482.sphinx3` presents a slowdown for the conservative baseline we do not fully understand, but seems to come from our prefetcher. Though the main purpose of redundant micro-op removal is to save energy, it actually brings some speedup on a few benchmarks (up to 8%, on benchmark `localization`). This (modest) speedup comes from the fact that the micro-ops removed do not

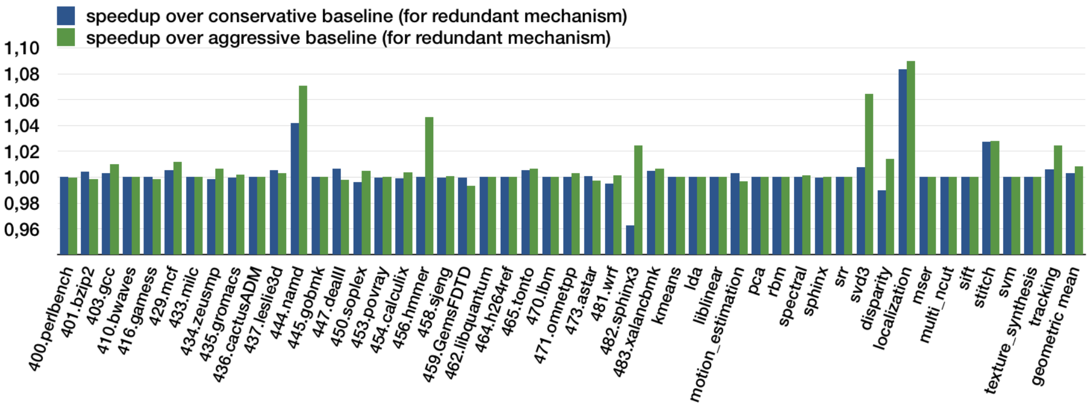


Figure 4.6: Speedups enabling redundant micro-ops removal.

use execution resources (ROB, physical registers, execution units, etc.), which can be used by other, non-redundant micro-ops.

4.3.6 Compiler Optimization impact: a case study

In Section 4.3.5, we saw that for "loopy" benchmarks, redundant loop micro-ops represent a substantial fraction of the total micro-ops. One of the benchmarks in which we have a large part of redundant micro-ops is `svd3`, in the cortex suite.

The `svd3` benchmark is an implementation in C language of the Singular Value Decomposition (SVD) algorithm. In our test we found four loops that are the main cause of our high "loopy" execution. Only one of them, as an example, is reported. The input set used in our benchmarks is *med.txt*.

The svd.c source code is about 250 lines. Most of them are loops with operations between matrix elements.

Listing 4.3: Snippet of `svd3` benchmark source code

```
for (j = l; j < n; j++)
    {
        for (s = 0.0, k = l; k < m; k++)
            s += ((double)a(k,i) * (double)a(k,j));
        f = (s / (double)a(i,i)) * g;
        for (k = i; k < m; k++)
            a(k,j) += (float)(f * (double)a(k,i));
    }
}
```

The example source in 4.3 executes operations for each element of the given matrix. The macro $a(k, j)$ is the short way used in the source for the matrix access. The index k is the row and the index j is the column of the initial matrix. The input set is a 500×500 matrix. One of the inner loop of the code iterates over all the row elements of the given matrix. The assembly version of the detected loop is shown in the source 4.4. Note that the loop buffer stores micro-ops, not instructions.

Listing 4.4: Assembly code of detected loop, `svd3` benchmark, compiled with -O2

```
405dcd:  mov       -0x30(%rbp),%eax             R
405dd0:  cmp       -0x94(%rbp),%eax             R
405dd6:  jge        405e3b <svd+0x3e5>
405dd8:  mov       -0x8(%rbp),%rsi              R
405ddc:  mov       -0x8(%rbp),%rax              R
405de0:  mov        (%rax),%eax                 R
405de2:  imul      -0x30(%rbp),%eax             R
405de6:  add       -0x28(%rbp),%eax             R
405de9:  movslq     %eax,%rdi
405dec:  mov       -0x8(%rbp),%r8               R
405df0:  mov       -0x8(%rbp),%rax              R
405df4:  mov        (%rax),%eax                 R
405df6:  imul      -0x30(%rbp),%eax             R
405dfa:  add       -0x28(%rbp),%eax             R
405dfd:  movslq     %eax,%rdx
405e00:  mov       -0x8(%rbp),%rcx              R
405e04:  mov       -0x8(%rbp),%rax              R
405e08:  mov        (%rax),%eax
405e0a:  imul      -0x30(%rbp),%eax             R
405e0e:  add       -0x20(%rbp),%eax             R
405e11:  cltq
405e13:  cvtss2sd   0x8(%rcx,%rax,4),%xmm0
405e19:  mulsd     -0x48(%rbp),%xmm0            R
405e1e:  cvtsd2ss   %xmm0,%xmm1
405e22:  movss      0x8(%r8,%rdx,4),%xmm0
405e29:  addss      %xmm1,%xmm0
405e2d:  movss      %xmm0,0x8(%rsi,%rdi,4)
405e33:  lea       -0x30(%rbp),%rax
405e37:  incl       (%rax)
405e39:  jmp        405dcd <svd+0x377>          R
```

This tiny loop is a common example of "loop sensitive" execution. In the source 4.4 we use symbol "R" to indicate instructions that contain redundant micro-ops. All move instructions are decoded as `addr` and `load`, some instructions are completely redundant and removable[6] (like `0x405dd8` or `0x405dec`) and others contains redundant `addr` (`0x405dcd`, `0x405dd0`, `0x405de2`, `0x405e0e`, basically stack accesses in the form "*basepointer+offset*" where the values are unchanged among iterations). Moreover, instructions like `0x405dd0` or `0x405de6` contain redundant loads, but they are not removable due to the ALU micro-op.

The last instruction, the `jmp`, is also redundant: the loop buffer is used as source for the fetching phase, and after the last micro-op fetched (the store micro-op belonging to the `incl` instruction at address `0x405e37`), the fetch begins all over again. The jump at address `0x405dd6` will compare the incremented value in `-0x30(%rbp)` with the one stored in `-0x94(%rbp)`, and it acts as exit point.

Listing 4.5: Assembly code of detected loop, `svd3` benchmark, compiled with -O3

```
404746:  lea       (%rcx,%r9,1),%eax
40474a:  lea       (%rcx,%rdi,1),%edx
40474d:  inc       %esi
40474f:  add       %r8d,%ecx
404752:  cmp       %r12d,%esi
404755:  cltq
404757:  movslq    %edx,%r11
40475a:  cvtss2sd  0x8(%rbx,%rax,4),%xmm14
404761:  mulsd     %xmm1,%xmm14
404766:  cvtsd2ss  %xmm14,%xmm13
40476b:  addss     0x8(%rbx,%r11,4),%xmm13
404772:  movss     %xmm13,0x8(%rbx,%r11,4)
404779:  jl        404746 <svd+0x276>
```

For this benchmark, the compiler plays an important role in the code optimization. Source in 4.5 shows the same loop compiled with higher optimization (`gcc -O3`). The code doesn't present the same impact as the previous one in terms of redundant micro-ops.

In order to better understand the impact of compiler optimizations on redundant micro-ops, we analyzed other benchmarks. The assembly code in 4.6 is a loop of `456.hmmer` (SPEC2006) with a high number of redundant micro-ops. Again, symbol "R" is for instructions that contain redundant micro-ops.

Listing 4.6: Assembly code for a loop of `456.hmmer`, compiled with -O2

```
4032f8:  add   $0x1,%r15
4032fc:  add   $0x4,%rbx
403300:  cmp   %r12,%r15
403303:  je    403360
403305:  mov   0x8(%rdi),%rsi        R
```

---

[6] An instruction is defined removable when all its micro-ops are redundant

```
403309:    mov      %r14d,%r11d              R
40330c:    sub      %r15d,%r11d
40330f:    add      %r13,%rsi
403312:    movb     $0x2,(%rsi,%r15,1)
403317:    mov      0x68(%rsp),%rdi          R
40331c:    mov      0x10(%rdi),%rsi          R
403320:    mov      %r11d,(%rsi,%rbx,1)
403324:    mov      0x18(%rdi),%rsi          R
403328:    movl     $0x0,(%rsi,%rbx,1)
40332f:    lea      (%r15,%rdx,1),%esi
403333:    cmp      %esi,%ebp
403335:    jne      4032f8
```

In the code 4.7 the optimization doesn't change the assembly as in the previous example, and the number of instructions that contain redundant micro-ops remains the same.

Listing 4.7: Assembly code for a loop of `456.hmmer`, compiled with -O3

```
4044c0:    add      $0x1,%r12
4044c4:    add      $0x4,%r14
4044c8:    cmp      %r15,%r12
4044cb:    je       404530
4044cd:    mov      %rbp,%rsi                R
4044d0:    add      0x8(%rdi),%rsi           R
4044d4:    mov      %r13d,%r10d
4044d7:    sub      %r12d,%r10d
4044da:    movb     $0x2,(%rsi,%r12,1)
4044df:    mov      0x68(%rsp),%rdi          R
4044e4:    mov      0x10(%rdi),%rsi          R
4044e8:    mov      %r10d,(%rsi,%r14,1)
4044ec:    mov      0x18(%rdi),%rsi          R
4044f0:    movl     $0x0,(%rsi,%r14,1)
4044f7:
4044f8:    lea      (%rcx,%r12,1),%esi
4044fc:    cmp      %esi,%ebx
4044fe:    jne      4044c0
```

With compiler optimizations turn on, the compiler is able to remove potentially redundant instructions. Thus, the redundancy detector can not find instructions to remove. However, the compiler is not always able to perform this optimization.

For the `svd` benchmark, the redundancy detector may become less effective, despite the ability to target the same loops. In fact, in our example, the same loop (source code 4.3) is compiled, via `gcc`, into binaries containing two different sets of assembly instructions (code 4.4 with default optimization and code 4.5 with `-O3` optimization). The latter is more compact and presents less redundant micro-ops. For `456.hmmer`, i.e., the `-O3` optimization doesn't impact too much the assembly code produced for the loop, therefore the total number of redundant micro-ops detected remains the same.

4.4 Reducing the energy of loads

In this section, we explore the possibility of reducing the energy of loads by predicting whether they will obtain their data from the DL1 cache or from the STQ. The goal is to save energy by gating whichever of the two accesses, DL1 or STQ, is useless. In principle, DL1/STQ gating is a stand-alone idea, independent of loops.

We show however that DL1/STQ gating must be done very carefully and that its viability depends on the memory dependence predictor behaving well, which is not necessarily always the case. We argue that obtaining energy savings from DL1/STQ gating is easier when it is restricted to loops. We propose an implementation for DL1/STQ gating and we evaluate its potential benefits.

4.4.1 Speculative load execution

First we explain how speculative load execution works in our baseline microarchitecture. Although this part of the microarchitecture is generally not documented for recent commercial processors, we believe that the problems with DL1/STQ gating are not tied to a particular implementation, provided there is an STQ.

In high-performance superscalar microarchitectures, loads can execute speculatively before older stores have written into the DL1 cache. When a load is predicted to be independent of any store in the instruction window, it can execute as soon as its address can be computed. When a load is predicted to depend on a store not yet retired, the load execution may be delayed. Thanks to the Store Queue, the load does not need to wait until the store retires from the ROB and can obtain the data from the STQ. Because the load execution is speculative, a mechanism is necessary to maintain correctness: this is the Load Queue, which maintains loads in the instruction window in program order. When a store retires from the ROB, an associative search is done in the LDQ with the store address, and loads that were misspeculated are marked (those that were correctly speculated are unmarked if they were marked by an older store).

When a marked load retires from the ROB, the instruction window is flushed, the processor state is repaired, and instruction fetching resumes at the instruction containing the load: this is a *memory trap*. The performance cost of a memory trap is higher than that of a branch misprediction because a memory trap is triggered at retirement.

Stores execute as follows. Each store is allocated an STQ entry. The STQ maintains stores in program order. A store executes as two independent micro-ops: an STA micro-op for computing the address, and an STD micro-op for moving the data from the register file to the STQ. When a store retires from the ROB, its STQ entries, which contains the store address and store data, becomes non speculative, and the store is now allowed to write in the DL1 cache.

Loads execute as follows. First the memory dependence predictor is accessed. If the load is predicted to wait on a particular STQ entry, the pointer for that STQ entry is written into the load's LDQ entry so that the load can be woken up when both the store address and data (whichever is computed last) are available. When a load is issued (and whether or not it was predicted dependent), it accesses *simultaneously* the DL1 cache and the STQ. The STQ is fully associative. If several stores older than load conflict with the load's address, the conflicting STQ entry is the youngest of these stores. Three situations can occur:

1. **STQ-miss** if no STQ entry conflicts with the load
2. **STQ-hit** if the conflicting STQ entry can provide the data
3. **STQ-conflict** if the conflicting STQ entry cannot provide the data to the load (typically, because the data is not yet available)

Upon an STQ-miss, the load gets the data from the DL1. Upon an STQ-hit, the data read from the STQ supersedes the data read from the DL1. Upon an STQ-conflict, the load execution is aborted, that of micro-ops issued after the load too, this is an *STQ-conflict replay*. The STQ pointer in the LDQ entry is updated, and the load will be reissued later. The performance and energy cost of an STQ-conflict replay is similar to that of a DL1-miss replay.

The memory dependence predictor in our baseline is the Store Sets [CE98]. The Store Sets is trained to prevent memory traps and STQ-conflict replays. The classical Store Sets predictor has two components: a Store Set ID Table (SSIT), indexed by hashing load/store PCs[7], and a LFST, searched with the Store Set Identifier (SSID). The baseline SSIT has 2048 entries (Table 3.2). Our LFST is isomorphic to the STQ. The Store Sets predictor is very effective at preventing memory traps. The price of this effectiveness is that false dependences are sometimes predicted, which delay the execution of some loads more than necessary[8].

---

[7] We use different hashing functions for loads and stores, as some instructions contain both a load and a store

4.4.2 DL1/STQ gating?

Loads access the DL1 and the STQ simultaneously, so that the DL1 latency is not serialized with the STQ latency. However, eventually, one of these two accesses is unnecessary. If we could predict before issuing the load which access is unnecessary, the STQ or the DL1, we could save energy by gating that access, without impacting performance:

- **STQ gating**: the load accesses the DL1 cache only. Correctness is guaranteed by the LDQ (the STQ is here only for performance).
- **DL1 gating**: the load accesses the STQ only. In case of an STQ-miss, an *STQ-miss replay* occurs.

STQ gating and DL1 gating are orthogonal, they can be used together, or one can be used and not the other. STQ gating and DL1 gating have different performance costs in case of wrong prediction, and different energy benefits in case of correct prediction. On the one hand, a DL1 cache access generally consumes more energy than an STQ access. Hence the potential benefit of DL1 gating is greater than that of STQ gating.

On the other hand, an incorrect STQ gating prediction generates a memory trap, which has a high performance and energy cost, while an incorrect DL1 gating prediction generates an STQ-miss replay, whose performance and energy cost is less than that of a memory trap. Opportunities for STQ gating are more frequent than those for DL1 gating. However, DL1 gating is a safer and more rewarding “game” than STQ gating. Still, as the gain is only energy, both STQ gating and DL1 gating must be done very carefully.

The first idea that comes to mind is to use the prediction from the memory dependence predictor:

- If a load is predicted to be independent of any store in the instruction window, do STQ gating.
- If a load is predicted to depend on a store in the instruction window, do DL1 gating.

The top graph of Figure 4.7 shows the number of memory traps per 1000 instructions for 4 configurations:

---

[8] Compared to an oracle predictor, the average performance loss from Store Set false dependences is about 2%.

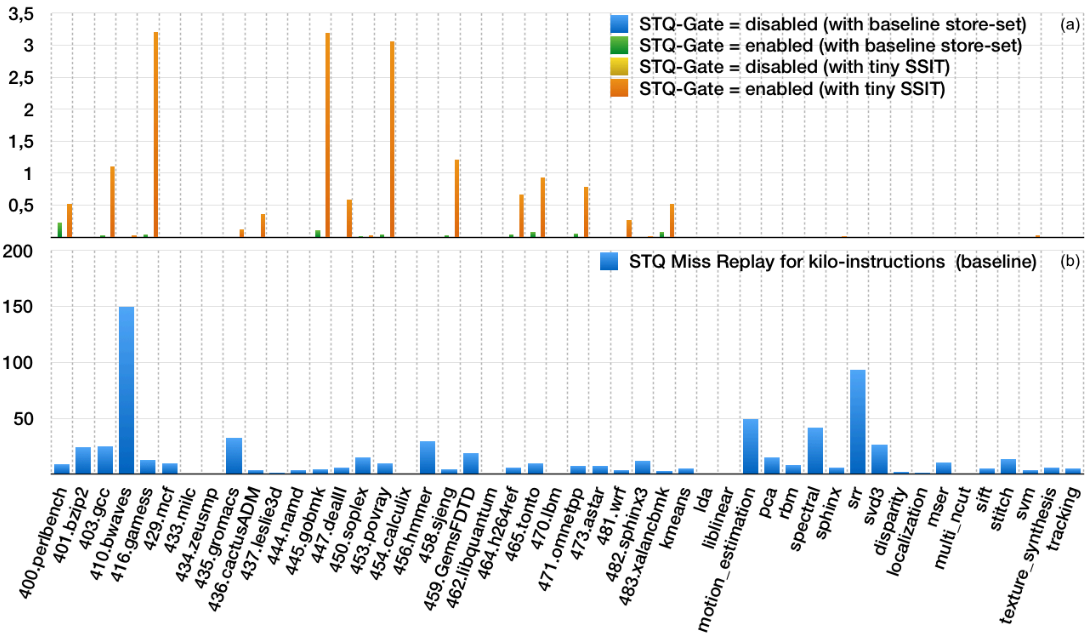


Figure 4.7: Top graph: memory traps per 1000 instructions, for 4 configurations: SSIT=2048 or SSIT=256, with and without global STQ gating. Bottom graph: STQ-miss replays per 1000 instructions when enabling global DL1 gating.

1. conservative baseline (no gating, 2048-entry SSIT)
2. STQ gating and 2048-entry SSIT
3. no gating and 256-entry SSIT
4. STQ gating and 256-entry SSIT

The 256-entry SSIT is for observing what happens for applications with a very large instruction footprint, as our set of benchmarks does not include any. Applications with a very large instruction footprint do exist, and microarchitects in the industry have to take them into account.

We see that, in the baseline microarchitecture, the Store Set predictor is very effective at minimizing the number of memory traps. Surprisingly, the number of memory traps does not increase significantly with the 256-entry SSIT. What happens here is that the degraded behavior of the Store Set predictor is hidden by the STQ, which acts as a safety net for keeping memory traps low. Even with OoO execution, address computations tend to occur in program order, which is why the STQ can compensate for the degraded Store Set predictions[9].

---

[9] Hence the importance of splitting stores into 2 micro-ops.

However, with STQ gating, the picture changes. Using Store Set predictions to drive STQ gating looks OK with the 2048-entry SSIT. But with the 256-entry SSIT, the number of memory traps increases dramatically because (1) Store Sets predictions are no longer reliable and (2) we lose the safety-net effect of the STQ.

The bottom graph of Figure 4.7 shows the number of STQ-miss replays per 1000 instructions when DL1 gating is enabled. Without DL1 gating, STQ-miss replays are inexistent (we can only have STQ-conflict replays, cf. Section 4.4.1). With DL1 gating enabled, STQ-miss replays can occur, and there are a lot, as can be seen in Figure 4.7 . The reason for the numerous STQ-miss replays is that the Store Set predictor predicts many false dependencies. We conclude that the Store Set predictor is not the right tool for DL1 gating.

4.4.3 Store Queue and DL1 gating in Loop Mode

As explained above, DL1/STQ gating must be done very carefully to obtain net energy savings. A memory dependence predictor such as the Store Sets, whose goal is to keep memory traps at a minimum, is definitely not the right tool for DL1 gating, and probably not for STQ gating[10]. DL1/STQ gating might be viable with a hypothetical memory dependence predictors that would have the following characteristics:

1. it would keep memory traps infrequent without introducing too many false dependencies (DL1 gating) and
2. it would provide reliable confidence estimations for its predictions, without being oversized (STQ gating)

We do not dismiss the possibility that such memory dependence predictor might exist. Nevertheless, we argue that DL1/STQ gating is straightforward in loop mode as we can take advantage of the small instruction footprint and of the regularity of memory dependencies (when a load depends on a store in one loop iteration, the same load is likely to depend on the same store in all loop iterations).

The following describes minimal modifications to the loop buffer, LDQ and STQ to allow STQ gating and DL1 gating in loop mode. We also propose to gate the Store Sets SSIT like the rest of the front-end. We first describe STQ gating and DL1 gating independently from each other, and then we explain how they can be combined. We assume that each LDQ entry contains a *loop bit* allowing to distinguish loop loads from pre-loop and post-loop ones.

---

[10] The SSIT could be made very large. However, for applications with a small or average instruction footprint, the SSIT would consume more energy, and it is not obvious that the energy saved by not accessing the STQ would compensate the extra energy spent in an oversized predictor.

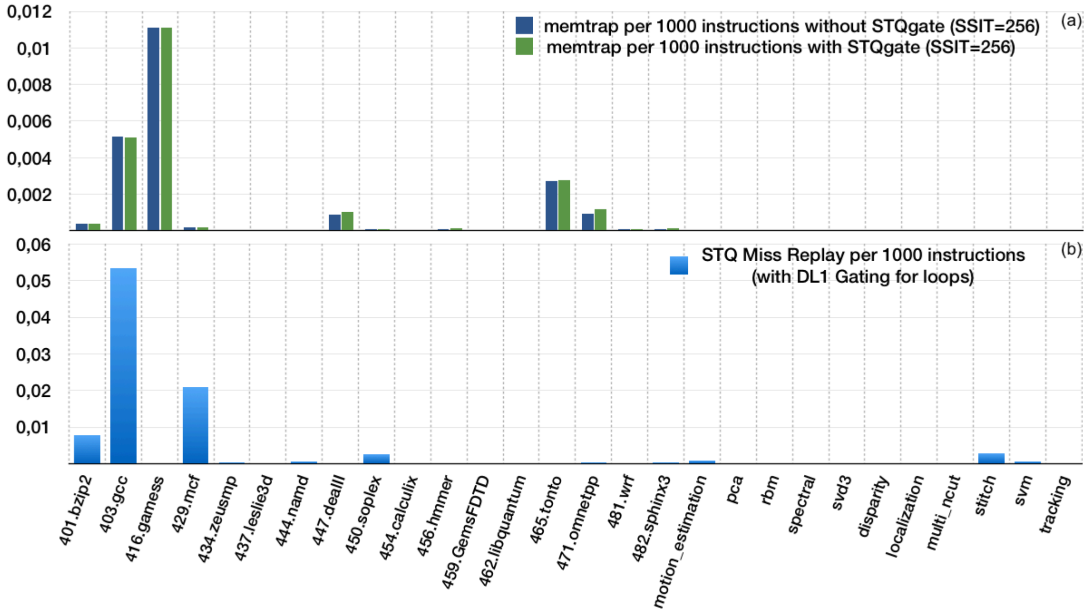


Figure 4.8: Top graph: memory traps per 1000 instructions, with and without STQ gating enabled for loops. Bottom graph: STQ-miss replays per 1000 instructions when DL1 gating is enabled for loops. Only the conservative loop buffer and “loopy” benchmarks are shown.

4.4.4 STQ gating alone

As discussed previously, the problem with using Store Sets predictions for STQ gating is when the prediction quality is degraded because the application’s instruction footprint is large. In loop mode however, we can assume, with high confidence, that the instruction footprint is small. We apply STQ gating to every loop load that is predicted by the Store Sets independent from any store in the STQ. STQ gating does not require any modification in the hardware other than loop bits in the LDQ.

Figure 4.8 (a) gives the number of memory traps per 1000 instructions, assuming a 256-entry SSIT, with and without STQ gating. Unlike what we observed when we tried to apply STQ gating independently of loop detection, the number of memory traps does not increase significantly.

4.4.5 DL1 gating alone

For DL1 gating, the main problem is the many STQ-miss replays coming from the false dependences predicted by the Store Sets. The Store Sets predictor cannot be used for DL1 gating.

To solve this problem, we introduce a *STQ-hit bit* in every loop buffer entry (though it is used only by loads). The purpose of the STQ-hit bit is to identify loads that hit in the STQ. We apply DL1 gating to every load that has its STQ-hit bit set. Initially, upon entering the steady loop mode, all the STQ-hit bits are unset. When a load hits in the STQ, its STQ-hit bit is set. When a STQ-miss replay happens, the STQ-hit bit of the load generating that replay is reset.

Opportunities for DL1 gating are not as frequent as opportunities for STQ gating. To increase the opportunities for DL1 gating, we keep the STQ always full by doing as follows: whenever a store writes into the DL1 cache, instead of invalidating the STQ entry, we keep it valid and we just increase the head pointer of the STQ to indicate that this STQ entry is now reclaimable. Valid reclaimable STQ entries do not generate any STQ-conflict replay. Keeping the STQ always full increases the chances to have STQ hits.

Only slight hardware modifications are required in the LDQ[11]. We extend each LDQ entry with a copy of the STQ-hit bit and with a loop buffer pointer (6 or 7 bits) indicating the location of the load in the loop buffer. These extra bits are gated off when execution is not in loop mode and, in loop mode, they remain gated until an STQ hit occurs (so that no extra dynamic and static energy is spent unless DL1 gating opportunities exist).

The STQ-hit bit in the LDQ is for enabling/disabling DL1 gating for the corresponding load. The loop buffer pointer is for updating the STQ-hit bit in the loop buffer whenever necessary. There is a narrow bus (6 or 7 bits) going from the STQ back to the loop buffer for switching the STQ-hit bit.

Most of the time, because of the stability of memory dependences in loops, the STQ-hit bit is not switched at all, or is set only once (during the first loop iteration). If the state of the STQ-hit bit does not change (which we know from the STQ-hit bit copy in the LDQ), the switching of the STQ-hit bit can be gated and no dynamic energy is spent in the narrow bus.

As for maintaining an STQ entry valid after it has written into the DL1, this does not require any change other than not resetting the valid bit and preventing this entry from generating STQ-conflict replays. We believe that this change in STQ management has negligible impact on energy consumption.

Figure 4.8 (b) gives the number of STQ-miss replays per 1000 instructions when DL1 gating is enabled. As can be seen, STQ-miss replays are very infrequent.

4.4.6 STQ gating and DL1 gating combined

STQ gating and DL1 gating were previously described separately. Now we combine them.

---

[11] The LDQ is not only for verifying that loads execute correctly, it is also the load scheduler.

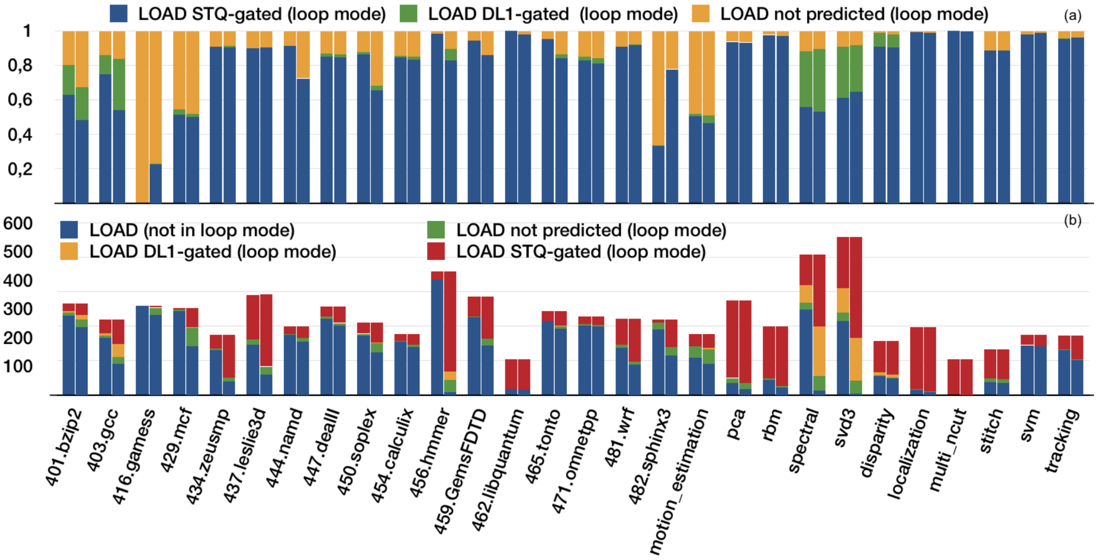


Figure 4.9: Top graph: fraction of loop load micro-ops that are DL1-gated, STQ-gated and not predicted. Bottom graph: per 1000 instructions, number of non-loop loads, not-gated loop loads, STQ-gated loop loads and DL1-gated loop loads. Only "loopy" benchmarks are shown.

A potential issue is with loads hitting on valid reclaimable STQ entries. For these loads, both the DL1 and the STQ can provide the data. We can do either STQ gating and DL1 gating (not both). When both STQ gating and DL1 gating are possible, DL1 gating is preferable because accessing the STQ costs less energy than accessing the DL1.

The difficulty comes from the fact that some loads that are predicted independent by the Store Sets may hit in the STQ, because the Store Sets predictor is trained to prevent memory traps and STQ-conflict replays but is not trained to detect all STQ hits. As described previously, we apply STQ gating to any predicted-independent load. Once a load is STQ-gated, it no longer accesses the STQ, and if there is an opportunity for an STQ hit, this opportunity is not detected.

To maximize opportunities for DL1 gating, we disable STQ gating during the first few iterations in steady loop mode. Once STQ gating is enabled, if the STQ-hit bit is set for a predicted-independent load, DL1 gating prevails over STQ gating for that load.

The top graph of Figure 4.9 shows the fraction of loop loads that are non-gated, STQ-gated, and DL1-gated. STQ-gated loads are generally the majority of loop loads. DL1-gated loads are much less frequent. Still, DL1-gated loads constitute a non negligible fraction of loop loads for 6 benchmarks: `401.bzip2`,

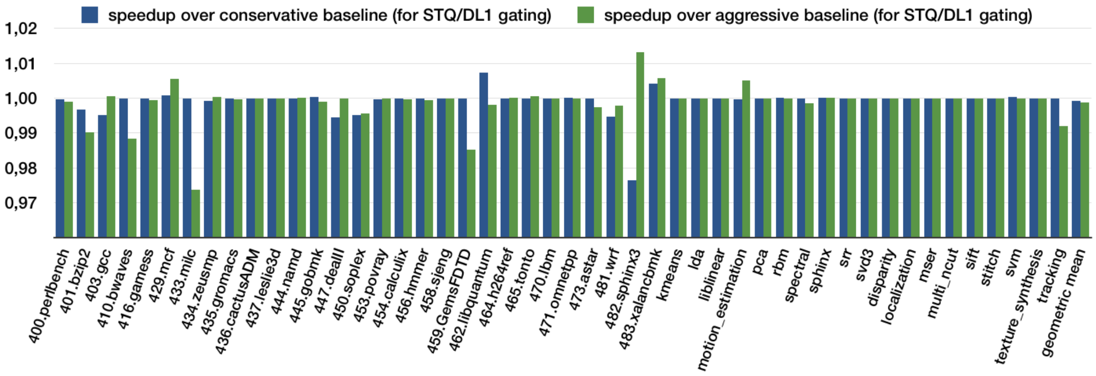


Figure 4.10: Speedups when STQ gating and DL1 gating are both enabled for loops.

`403.gcc`, `456.hmmer` (128-micro-ops loop buffer), `spectral`, `svd3` and `disparity`.

The bottom graph of Figure 4.9 takes into account loop coverage and shows, per 1000 instructions, the number of non-loop loads, non-gated loop loads, STQ-gated loop loads and DL1-gated loop loads. As expected, for the most "loopy" benchmarks, a majority of loads are STQ-gated. For instance, on `456.hmmer` with the 128-micro-ops loop buffer, there are about 350 STQ-gated loads per 1000 instructions. A majority of benchmarks have very few DL1-gated loads. Still, for a few benchmarks, DL1-gated loads are non negligible. For instance, benchmark `spectral` with the 128-micro-ops loop buffer has about 180 DL1-gated loads per 1000 instructions.

Figure 4.10 shows the speedups for the conservative and aggressive baselines when STQ gating and DL1 gating are combined. Memory traps and STQ-miss replays are very infrequent and have negligible impact on performance. There is a tiny slowdown for the aggressive baseline, which we do not fully understand at the time of this thesis was submitted. It does not come from memory traps or STQ-miss replays, but seems to come from our prefetcher (we are still investigating). Nevertheless, on average (geometric mean, rightmost bar), the slowdown is insignificant.

4.4.7 Gating the memory dependence predictor table

In our description so far we have assumed that the SSIT continues to be accessed by loads and stores in loop mode. However the SSIT is a large table, and accessing it consumes some energy. Yet, the content of the SSIT changes only infrequently. In particular, in loop mode, a given load or store gets the same SSID on every loop iteration. The idea is to store the SSID in the loop buffer so that the SSIT needs not be accessed. We could augment each loop buffer entry so that it holds the SSID. On the first iteration, SSIT accesses are enabled: the SSID for a load/store is obtained from the SSIT and it is copied into the loop buffer entry. From the second iteration, SSIT accesses are disabled, and SSIDs are obtained directly from the loop buffer. The SSIT must be updated if any of the following two events happens: (1) memory trap or (2) STQ-conflict replay. A memory trap triggers a loop exit. However an STQ-conflict replay does not trigger a loop exit. When an STQ conflict replay occurs, the SSIT is updated and SSIT accesses are enabled for one loop iteration to update the SSIDs in the loop buffer (actually a single SSID changes, that of the conflicting load or that of the conflicting store). This solution may be explored and evaluated in future works.

4.5 Summary

Energy efficiency is an important aspect of modern microarchitecture. In this chapter we tried to approach this problem proposing solutions that help to reduce the microarchitecture power consumption, exploiting the behavior of loops in two different ways.

In the first we want to reduce the computational load of the backend. We used an existing loop buffer, we manipulate it in order to detect the so called redundant micro-ops, and we introduced a mechanism for avoiding the execution of these micro-ops. In this way the total number of micro-ops executed in the backend is reduced. Using a loop buffer is also useful for gating entirely the front-end and further energy saving.

The second solution proposes to identify load micro-ops, inside the loop buffer, that don't require the access to the store-queue or the DL1-cache memory. Fewer accesses means less energy consumption. For both solutions we analyzed the impact on performance, keeping in mind that our goal is to reduce the energy consumption without reducing the overall performance and without introducing complex hardware.

Chapter 5

Conclusion

In this chapter, the main conclusions of this thesis are presented, as well as some prospectives on future works and how the ideas presented in this thesis can be expanded.

5.1 The superscalar architecture of the future

As the number of cores grows, fewer applications benefit from this growth. Hence sequential performance is still very important. If the clock frequency remains fixed, as in the last 10 years, the only way to increase sequential performance without recompiling is to increase performance. Increasing performance significantly likely requires more hardware complexity, in particular a wider issue and a larger instruction window.

At some point, issuing more micro-ops per cycle requires to use clustering. Clustering was introduced and studied at a time when microarchitects were trying to push the clock frequency as high as possible. However, if the clock frequency remains constant, clustering becomes a means to increase performance, and our understanding of clustering must be updated.

Starting from microarchitecture parameters similar to recent commercial high-end cores, we showed that an effective way to increase the IPC is to allow the out-of-order engine to issue more micro-ops per cycle (Figure 5.1). But this must be done without impacting the clock cycle. We proposed to combine two techniques: clustering and register write specialization. Past research on clustered microarchitectures focused on narrow issue clusters, as the emphasis at that time was on allowing high clock frequencies.

Instead, in this study, we considered wide issue clusters, with the goal of increasing the IPC under a constant clock frequency. We showed that, on a wide-issue dual cluster, a very simple steering policy that sends 64 consecutive instructions to the same cluster, the next 64 instructions to the other cluster, and

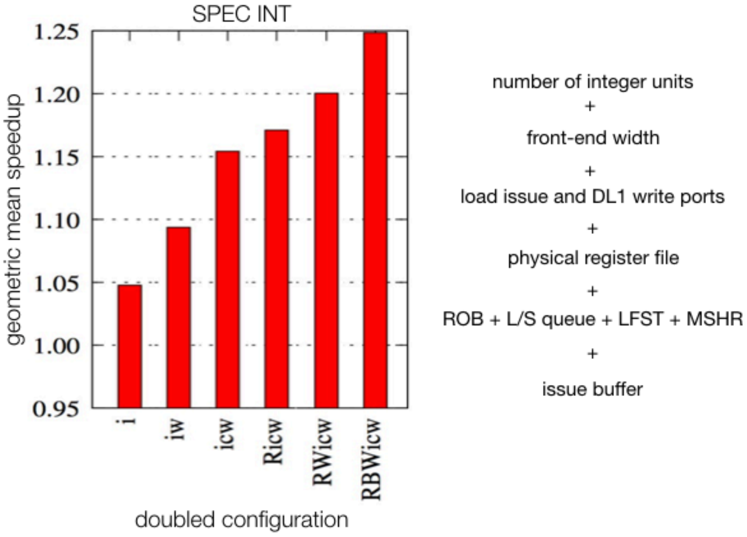


Figure 5.1: The extra complexity can be introduced incrementally: introducing more integer execution units, then increasing the front-end width, load issue and number of DL1 write ports, and continuing until the issue buffer. SPEC INT results are reported.

so forth, permits tolerating an intercluster delay of 3 cycles. We also proposed a method for decreasing the energy cost of sending results from one cluster to the other cluster.

We have also shown that, in single-thread execution, a significant fraction of the values produced by a cluster do not need to be forwarded to the other cluster. This can be exploited to gate some intercluster communications and decrease energy consumption.

The wide-issue dual-cluster configuration that we studied is supposed to be the result of an incremental complexification of the whole microarchitecture over several technology generations. Though clustering solves some key complexity issues in the OoO core (e.g., the complexity of more FUs, the large issue queue, etc.), other parts of the microarchitecture which were not the focus of this study, such as front-end bandwidth and load/store queues, will have to be scaled up too. Some of the solutions that will be needed for these other parts are already known. Some new solutions will probably be needed too.

5.2 Exploiting loops for power consumption

Recent superscalar processors feature a loop buffer for exploiting loop behaviors to reduce energy consumption in the front-end. For applications with a very loopy behavior, thanks to the loop buffer, most of the core energy is spent in the back-end. We argue in this thesis that loop behaviors can also be exploited in the back-end to reduce energy further.

The first optimization we propose is to detect and remove redundant micro-ops producing the same result in every loop iteration.

The micro-ops removal will benefit only benchmarks that show a high presence of loop instructions, as expected. However the hardware modification required for the redundant detection is transparent to normal operation. This allows to introduce the proposed hardware without changing the normal Out-of-Order behavior when non-loop instructions are executed. The advantage is that, for applications with a lot of loops, this mechanism is capable to reduce the total number of micro-ops sent to the back-end: these micro-ops can be also loads. In this way we have two consequences: less micro-ops executed in the back-end, and less memory micro-ops sent to the memory system. The benefits, in both cases, is mainly a reduction of power consumption.

The second optimization we propose focuses on loop loads, specifically on recognition of load micro-ops that need accessing only the DL1 cache or only the store queue. These two optimizations are independent and orthogonal. With STQ gating applied only for loops, the number of memory traps does not increase significantly but the total number of STQ accesses is reduced (Figure 4.9(a)). The number of DL1 memory accesses that we can avoid thanks to this mechanism is lower, due to the nature of benchmarks, but still present and detected. Introducing DL1 gating for loop instructions, however, does not introduce a number of STQ-miss replays that would reduce the global performance. Both STQ gating and DL1 gating can be used at the same time, however the impact in terms of energy saving for DL1 gating is bigger and is usually preferable.

The two proposed architectural modifications require very little hardware, used only in loop mode, and they both exploit the regularity of loops.

5.3 Perspectives

In this thesis two aspects of modern microarchitectures are being addressed: the need for increasing single-thread performance and the need for reducing the energy consumption without reducing performance. Despite the common interest for multi-threaded architectures, it is interesting to understand the direction that processor makers will follow to achieve these results.

We studied the impact of clustering on modern architectures, and how it differs from previous studies. In this thesis we focused on dual-clustered configurations, balancing the issue of micro-ops between clusters and avoiding the loss of performance due to intercluster communication.

The conclusions we gave, however, can evolve along with technology improvement: an example is given by the natural evolution of intercluster connections, for instance in direction of optical connections. Faster connections could allow to increase the number of clusters in the back-end without the constraint of the communication delay, or allow the implementation of different steering algorithms for the two cluster configuration proposed here, pushing the priority on balancing over communication (discussed in 3.4.3).

The steering algorithm offers several possible ways to exploit the dual-cluster architecture: the distribution of micro-ops may be bound to the nature of each micro-op, to its type, or to the memory load it eventually requires. The steering algorithm can be kept as simple as possible, or can be more complex and can use different approaches to the balancing problem (like the adaptive method seen in Section 3.4.2).

These solutions allow to extend the instruction window. Current technological limits suggest to follow the philosophy of “Divide et Impera”[1] for the back-end, that can be applied also to the register file (i.e., using the register write specialization) or with separate memory caches.

In terms of energy consumption, the approach used is to reduce the total number of operations that the architecture must perform and the number of accesses in cache or in store queue. The use of a loop buffer reduces the energy consumed by the front-end, but we demonstrated that energy reduction in the back-end is also possible.

We may increase the changes and manipulations of the loop buffer using, i.e., more complex dedicated hardware, or accelerate the execution of loop instructions avoiding (or optimizing) the use of conventional back-end elements. The loops have interesting behavior that can be exploited, like the STQ/DL1 gating proposed in 4.4, or the implementation of a custom Store Set predictor, identical to that implemented in modern architectures, but dedicated only to the loops. This predictor could be used to predict static loads that depend on static stores with the same data size.

The content of the loop buffer may be manipulated not only for removing micro-ops, but also for changing the order of micro-ops or inserting new ones, for example introducing a `mov` micro-ops when a store-to-load dependency is detected, or new micro-ops for the address checking during the optimized execution of this “store-to-load bypass”.

---

[1] Divide and Conquer

Other optimizations may take advantage of read-after-write and write-after-write register dependencies for increasing the register renaming bandwidth or for directly feeding the input of a `mov` micro-op to the consumers of that micro-op.

Clustering techniques and loop buffer manipulations could work together. The steering could be modified in order to force the issue to a specific cluster only for micro-ops that belong to the same iteration, or may perform steering according to the dependency of micro-ops among different iterations.

Another idea can be to identify, within large loops, groups of micro-ops mutually independent but dependent only from the result of the previous iteration. Once identified, we can issue these groups to different cluster despite the fact they belong to the same loop body. Finally, the steering can consider the memory demand of the loop, statistically evaluated during the loop capture mode or during the first iterations of the steady loop mode.

Although the aspects of microarchitecture addressed in this thesis are orthogonal, they could be tackled together, maximizing the mutual benefits.

Bibliography

[AA00] T Anderson and S Agarwala. Effective hardware-based two-way loop cache for high performance low power processors. In *Proceedings 2000 International Conference on Computer Design*, pages 403–407. IEEE Comput. Soc, 2000.

[ACR95] P. S. Ahuja, D. W. Clark, and A. Rogers. The performance impact of incomplete bypassing in processor pipelines. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 36–45, Nov 1995.

[AF02] Aneesh Aggarwal and Manoj Franklin. Hierarchical interconnects for on-chip clustering. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 173–, Washington, DC, USA, 2002. IEEE Computer Society.

[AHKB00] 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 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pages 248–259, June 2000.

[AMD13] AMD. Software optimization guide for amd family 16h processors. `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/SOG_16h_52128_PUB_Rev1_1.pdf`, 2013.

[AS79] W. Abu-Sufah. Automatic program transformations for virtual memory computers. *Proc. Nat. Computer Conf.*, 48:969–975, 1979.

[AST67] D W Anderson, F J Sparacio, and R M Tomasulo. The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling. *IBM Journal of Research and Development*, 11(1):8–24, 1967.

[AZ05] Bramha Allu and Wei Zhang. Exploiting the replication cache to improve performance for multiple-issue microprocessors. *SIGARCH Comput. Archit. News*, 33(3):63–71, June 2005.

[BDMF10] Geoffrey Blake, Ronald G Dreslinski, Trevor Mudge, and Krisztián Flautner. Evolution of thread-level parallelism in desktop applications. *ACM SIGARCH Computer Architecture News*, 38(3):302–313, June 2010.

[BFA95] J Bunda, D Fussell, and W C Athas. Energy-efficient instruction set architecture for CMOS microprocessors. In *Twenty-Eighth Annual*

Hawaii International Conference on System Sciences, pages 298–305. IEEE Comput. Soc. Press, 1995.

[BHK$^+$97] R S Bajwa, M Hiraki, H Kojima, D J Gorny, K Nitta, A Shridhar, K Seki, and K Sasaki. Instruction buffering to reduce power in processors for signal processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(4):417–424, 1997.

[BHP99] Nikolaos Bellas, Ibrahim Hajj, and Constantine Polychronopoulos. Using dynamic cache management techniques to reduce energy in a high-performance processor. In *Proceedings of the International Symposium on Low Power Electronics and Design*, ISLPED '99, pages 64–69, New York, NY, USA, 1999. ACM.

[BJ03] Ravi Bhargava and Lizy K. John. Improving dynamic cluster assignment for clustered trace cache processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 264–274, New York, NY, USA, 2003. ACM.

[BKM$^+$04] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, July 2004.

[BM00] A Baniasadi and A Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 337–347, 2000.

[Bor99] Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, July 1999.

[BZ06] L Baugh and C Zilles. Decomposing the load-store queue by function for power reduction and scalability. *IBM Journal of Research and Development*, 50(2.3):287–297, 2006.

[CAC$^+$10] P Carazo, R Apolloni, F Castro, D Chaver, L Pinuel, and F Tirado. L1 Data Cache Power Reduction Using a Forwarding Predictor. In *Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation*, pages 116–125. Springer Berlin Heidelberg, Berlin, Heidelberg, September 2010.

[CCGG08] Qiong Cai, J. M. Codina, J. Gonzalez, and A. Gonzalez. A software-hardware hybrid steering mechanism for clustered microarchitectures. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, April 2008.

[CE98] George Z. Chrysos and Joel S. Emer. Memory Dependence Prediction Using Store Sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, pages 142–153, Washington, DC, USA, 1998. IEEE Computer Society.

[CG01] Ramon Canal and Antonio González. Reducing the complexity of the issue logic. In *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, pages 312–320, New York, NY, USA, 2001. ACM.

[CHM08] Nathan Clark, Amir Hormati, and Scott Mahlke. VEAL: Virtualized Execution Accelerator for Loops. In *35th International Symposium on Computer Architecture (ISCA)*, pages 389–400. IEEE, 2008.

[CL04] Harold W. Cain and Mikko H. Lipasti. Memory ordering: A value-based approach. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 90–, Washington, DC, USA, 2004. IEEE Computer Society.

[CLG$^+$14] Kenneth Czechowski, Victor W. Lee, Ed Grochowski, Ronny Ronen, Ronak Singhal, Richard Vuduc, and Pradeep Dubey. Improving the energy efficiency of big cores. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 493–504, Piscataway, NJ, USA, 2014. IEEE Press.

[CMO99] S Curtis, R J Murray, and H Opie. Multiported bypass cache in a bypass network, 1999. Patent, US 6000016.

[Col05] Robert P. Colwell. *The Pentium Chronicles: The People, Passion, and Politics Behind Intel's Landmark Chips (Software Engineering "Best Practices")*. Wiley-IEEE Computer Society Pr, 2005.

[Cor94] Digital Equipment Corporation. *Alpha Architecture Handbook*, 1994.

[CPG99] Ramon Canal, Joan-Manuel Parcerisa, and Antonio Gonzalez. A cost-effective clustered architecture. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, pages 160–, Washington, DC, USA, 1999. IEEE Computer Society.

[CPG00] R. Canal, J. M. Parcerisa, and A. Gonzalez. Dynamic cluster assignment mechanisms. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*, pages 133–142, 2000.

[CPG01] Ramon Canal, Joan-Manuel Parcerisa, and Antonio González. Dynamic code partitioning for clustered architectures. *Int. J. Parallel Program.*, 29(1):59–79, February 2001.

[CR00] Brad Calder and Glenn Reinman. A comparative survey of load speculation architectures. *Journal of Instruction-Level Parallelism*, 2000.

[dAK01] M. R. de Alba and D. R. Kaeli. Runtime predictability of loops. In *Proceedings of the Workload Characterization*, WWC '01, pages 91–98, Washington, DC, USA, 2001. IEEE Computer Society.

[DB07] Kaveh Jokar Deris and Amirali Baniasadi. Investigating cache energy and latency break-even points in high performance processors. *SIGARCH Comput. Archit. News*, 35(4):13–20, September 2007.

[Des98] G. Desoli. Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach. Technical Report HPL-98-13, HP Labs, 1998.

[DGR$^+$74] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, October 1974.

[EBSA$^+$11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[Eve90] Dave J. Everitt. Inexpensive performance using the Am29000. *Microprocessors and Microsystems*, 14(6):397–406, July 1990.

[FCJ97] K I Farkas, P Chow, and N P Jouppi. The multicluster architecture: Reducing cycle time through partitioning. *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 149–159, 1997.

[FF98] J A Farrell and T C Fischer. Issue logic for a 600-MHz out-of-order execution microprocessor. *IEEE Journal of Solid-State Circuits*, 33(5):707–712, May 1998.

[FG00] J. Fridman and Z. Greenfield. The TigerSHARC DSP architecture. *IEEE Micro*, 20(1):66–76, Jan 2000.

[FPP98] Daniel Holmes Friendly, Sanjay Jeram Patel, and Yale 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*, MICRO 31, pages 173–181, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[FRB01] Brian Fields, Shai Rubin, and Rastislav Bodík. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 74–85, New York, NY, USA, 2001. ACM.

[FS96] Manoj Franklin and Gurindar S. Sohi. Arb: A hardware mechanism for dynamic reordering of memory references. *IEEE Trans. Comput.*, 45(5):552–571, May 1996.

[fS13] International Technology Roadmap for Semiconductors. Process integration, devices, and structures. `http://www.semiconductors.org/clientuploads/Research_Technology/ITRS/2013/2013PIDS.pdf`, 2013.

[GAV11] M. Golden, S. Arekapudi, and J. Vinh. 40-Entry unified out-of-order scheduler and integer execution unit for the AMD Bulldozer x86-64 core. In *2011 IEEE International Solid-State Circuits Conference*, pages 80–82, Feb 2011.

[GK98] Kanad Ghose and Milind B Kamble. Energy efficient cache organizations for superscalar processors. In *Power-Driven Microarchitecture Workshop In Conjunction With ISCA98 in Barcelona*, 1998.

[GLG04] José González, Fernando Latorre, and Antonio González. Cache organizations for clustered microarchitectures. In *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*, WMPI '04, pages 46–55, New York, NY, USA, 2004. ACM.

[GLM10] Antonio Gonzalez, Fernando Latorre, and Grigorios Magklis. *Processor Microarchitecture:An Implementation Perspective*, volume 5. Morgan & Claypool, 2010.

[GNK$^+$01] Masahiro Goshima, Kengo Nishino, Toshiaki Kitamura, Yasuhiko Nakashima, Shinji Tomita, and Shin-ichiro Mori. A high-speed dynamic instruction scheduling scheme for superscalar processors. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 225–236, Washington, DC, USA, 2001. IEEE Computer Society.

[GRCV02] A. Gordon-Ross, S. Cotterell, and F. Vahid. Exploiting fixed programs in embedded systems: A loop cache example. *IEEE Computer Architecture Letters*, 1(1):2–2, January 2002.

[GSF$^+$08] Alejandro Garcia, Oliverio J Santana, Enrique Fernandez, Pedro Medina, and Mateo Valero. LPA: A First Approach to the Loop Processor Architecture. In *High Performance Embedded Architectures and Compilers*, pages 273–287. Springer Berlin Heidelberg, Berlin, Heidelberg, January 2008.

[Gwe95] Linley Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2):9–15, 1995.

[Hen06] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[HNL14] Mitchell Hayenga, Vignyan Reddy Kothinti Naresh, and Mikko H Lipasti. Revolver: Processor architecture for power efficient loop execution. In *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 591–602. IEEE, 2014.

[HSU$^+$01] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean Alan Kyker, and Patrice Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 2001.

[HVK$^+$04] J S Hu, N Vijaykrishnan, S Kim, M Kandemir, and M J Irwin. Scheduling reusable instructions for power reduction. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 148–153. IEEE Comput. Soc, 2004.

[Inc06] Texas Instrument Inc. TMS320C67x/C67x+ DSP CPU and Instruction Set reference guide. `http://www.ti.com/lit/ug/spru733a/spru733a.pdf`, 2006.

[Int16a] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, January 2016.

[JLW01] Roy Dz-ching Ju, Alvin R. Lebeck, and Chris Wilkerson. Locality vs. Criticality. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 132–143, New York, NY, USA, 2001. ACM.

[JS99] Q Jacobson and J E Smith. Instruction pre-processing in trace processors. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 125–129. IEEE, 1999.

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

[KGMS97] J Kin, Munish Gupta, and W H Mangione-Smith. The filter cache: an energy efficient memory structure. In *30th Annual International Symposium on Microarchitecture*, pages 184–193. IEEE Comput. Soc, 1997.

[KLW$^+$04] D. Kim, S. S. W. Liao, P. H. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen. Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 27–38, March 2004.

[Kob84] M Kobayashi. Dynamic characteristics of loops. *IEEE Transactions on Computers*, C-33(2):125–132, 1984.

[KS04] T S Karkhanis and J E Smith. A first-order superscalar processor model. In *Proceedings. 31st Annual International Symposium on Computer Architecture*, pages 338–349. IEEE, 2004.

[KST11] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 393–404, New York, NY, USA, 2011. ACM.

[Lan11] T Lanier. Exploring the Design of the Cortex A15 Processor. `https://www.arm.com/files/pdf/AT-Exploring_the_Design_of_the_Cortex-A15.pdf`, 2011.

[LCM$^+$05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[LDH+05] Jiwei Lu, Abhinav Das, Wei-Chung Hsu, Khoa Nguyen, and Santosh G. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 93–104, Washington, DC, USA, 2005. IEEE Computer Society.

[LFK+93] P Geoffrey Lowney, Stefan M Freudenberger, Thomas J Karzes, W D Lichtenstein, Robert P Nix, John S O'Donnell, and John C Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[LMA99a] Lea Hwang Lee, Bill Moyer, and John Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design*, ISLPED '99, pages 267–269, New York, NY, USA, 1999. ACM.

[LWS96] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 138–147, New York, NY, USA, 1996. ACM.

[Mat97] D. Matzke. Will physical scalability sabotage performance gains? *Computer*, 30(9):37–39, Sep 1997.

[MBVS97] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 181–193, New York, NY, USA, 1997. ACM.

[MCC+06] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. Tartan: Evaluating spatial computation for whole program execution. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 163–174, New York, NY, USA, 2006. ACM.

[McI98] Nathaniel McIntosh. *Compiler Support for Software Prefetching.* PhD thesis, Rice University, 1998.

[ME15] George Matheou and Paraskevas Evripidou. Architectural support for data-driven execution. *ACM Trans. Archit. Code Optim.*, 11(4):52:1–52:25, January 2015.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[MSJ01] Pierre Michaud, André Seznec, and Stéphan Jourdan. An exploration of instruction fetch requirement in out-of-order superscalar processors. *Int. J. Parallel Program.*, 29(1):35–58, February 2001.

[MZS$^+$03] J. H. Moreno, V. Zyuban, U. Shvadron, F. D. Neeser, J. H. Derby, M. S. Ware, K. Kailas, A. Zaks, A. Geva, S. Ben-David, S. W. Asaad, T. W. Fox, D. Littrell, M. Biberstein, D. Naishlos, and H. Hunter. An innovative low-power high-performance programmable signal processor for digital communications. *IBM Journal of Research and Development*, 47(2.3):299–326, March 2003.

[NGS15] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. Exploring the potential of heterogeneous Von Neumann/Dataflow execution models. In *the 42nd Annual International Symposium*, pages 298–310, New York, New York, USA, 2015. ACM Press.

[NVI13] NVIDIA. NVIDIA Tegra 4 family CPU architecture. `http://www.nvidia.com/docs/IO/116757/NVIDIA_Quad_a15_whitepaper_FINALv2.pdf`, 2013.

[NVN03] D Nicolaescu, A Veidenbaum, and a Nicolau. Reducing data cache energy consumption via cached load/store queue. In *ISLPED International Symposium on Low-Power Electronics and Design*, pages 252–257. ACM, August 2003.

[OBC98] Emre Özer, Sanjeev Banerjia, and Thomas M. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, pages 308–315, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[PA12] O. Pell and V. Averbukh. Maximum performance computing with dataflow engines. *Computing in Science Engineering*, 14(4):98–103, July 2012.

[Pal98] S Palacharla. *Complexity-Effective Superscalar Processors.* PhD thesis, University of Wisconsin - Madison, 1998.

[PBB$^+$02] R P Preston, R W Badeau, D W Bailey, S L Bell, L L Biro, W J Bowhill, D E Dever, S Felix, R Gammack, V Germini, M K Gowan, P Gronowski, D B Jackson, S Mehta, S V Morton, J D Pickholtz, M H Reilly, and M J Smith. Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading. In *IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pages 334–472 vol.1. IEEE, 2002.

[PFP97] S. Patel, D. Friendly, and Y. Patt. Critical issues regarding the trace cache fetch mechanism. Technical Report CSE-TR-335-97, University of Michigan, 1997.

[PG00] Joan-Manuel Parcerisa and Antonio González. Reducing wire delay penalty through value prediction. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 317–326. ACM, 2000.

[PGB$^+$08] Frederico Pratas, Georgi Gaydadjiev, Mladen Berekovic, Leonel Sousa, and Stefanos Kaxiras. Low power microarchitecture with instruction reuse. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, pages 149–158, New York, NY, USA, 2008. ACM.

[PJS97] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 206–218, New York, NY, USA, 1997. ACM.

[POV03] Il Park, Chong Liang Ooi, and TN Vijaykumar. Reducing design complexity of the load/store queue. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 411. IEEE Computer Society, 2003.

[RAWM03] J a Rivers, S Asaad, J D Wellman, and J H Moreno. Reducing instruction fetch energy with backwards branch control information and buffering. In *ISLPED International Symposium on Low-Power Electronics and Design*, pages 322–325. ACM, 2003.

[RBS96] E Rotenberg, S Bennett, and J E Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 24–34. IEEE Comput. Soc. Press, 1996.

[RF72] E M Riseman and C C Foster. The Inhibition of Potential Parallelism by Conditional Jumps. *IEEE Transactions on Computers*, C-21(12):1405–1411, 1972.

[RF98] Narayan Ranganathan and Manoj Franklin. An empirical study of decentralized ilp execution models. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 272–281, New York, NY, USA, 1998. ACM.

[RFD13] Dyer Rolán, Basilio B. Fraguela, and Ramón Doallo. Virtually split cache: An efficient mechanism to distribute instructions and data. *ACM Trans. Archit. Code Optim.*, 10(4):27:1–27:24, December 2013.

[RG09] Won W. Ro and Jean-Luc Gaudiot. A Complexity-effective microprocessor design with decoupled dispatch queues and prefetching. *Parallel Comput.*, 35(5):255–268, May 2009.

[RJSS97] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 138–148, Dec 1997.

[Rot99] E Rotenberg. *Trace processors: Exploiting hierarchy and speculation.* PhD thesis, University of Wisconsin - Madison, 1999. Section 5.3.2.

[RTDA97] Jude A. Rivers, Gary S. Tyson, Edward S. Davidson, and Todd M. Austin. On high-bandwidth data cache design for multi-issue processors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 46–56, Washington, DC, USA, 1997. IEEE Computer Society.

[Rup12] Jeff Rupley. Jaguar, AMD's next generation low power x86 core. In *IEEE Hot Chips 24 Symposium (HCS)*, pages 1–20. IEEE, 2012.

[San06] Rama Sangireddy. Reducing rename logic complexity for high-speed and low-power front-end architectures. *IEEE Trans. Comput.*, 55(6):672–685, June 2006.

[SD95] Ching-Long Su and A M Despain. Cache designs for energy efficiency. In *Twenty-Eighth Annual Hawaii International Conference on System Sciences*, pages 306–315. IEEE Comput. Soc. Press, 1995.

[SDB+03] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable hardware memory disambiguation for high ILP processors. In *International Symposium on Microarchitecture (MICRO)*, 2003.

[SFKS02] A Seznec, S Felix, V Krishnan, and Y Sazeides. Design tradeoffs for the Alpha EV8 conditional branch predictor. In *29th Annual International Symposium on Computer Architecture*, pages 295–306. IEEE Comput. Soc, 2002.

[SG00] J. Sanchez and A. Gonzalez. Modulo scheduling for a fully-distributed clustered VLIW architecture. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, pages 124–133, 2000.

[Sim00] Dezsö Sima. The design space of register renaming techniques. *IEEE Micro*, 20(5):70–83, September 2000.

[SIT+14] Shreesha Srinath, Berkin Ilbeyi, Mingxing Tan, Gai Liu, Zhiru Zhang, and Christopher Batten. Architectural Specialization for Inter-Iteration Loop Dependence Patterns. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 583–595. IEEE, 2014.

[SKS+11] B Sinharoy, R Kalla, W J Starke, H Q Le, R Cargnoni, J A Van Norstrand, B J Ronchetti, J Stuecheli, J Leenstra, G L Guthrie, D Q Nguyen, B Blaner, C F Marino, E Retter, and P Williams. IBM POWER7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1:1–1:29, 2011.

[SKS14] B. N. Swamy, A. Ketterlin, and A. Seznec. Hardware/software helper thread prefetching on heterogeneous many cores. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, pages 214–221, Oct 2014.

[SL98] Srikanth T. Srinivasan and Alvin R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, pages 148–159, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[SL06] Samantika Subramaniam and Gabriel Loh. Fire-and-Forget: Load-/Store Scheduling with No Store Queue at All. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 273–284. IEEE, 2006.

[SM06] A Seznec and P Michaud. A case for (partially) TAgged GEometric history length branch prediction. *Journal of Instruction Level Parallelism*, 2006.

[Smi81] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[Smi98] James E. Smith. Retrospective: Implementing precise interrupts in pipelined processors. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, pages 42–, New York, NY, USA, 1998. ACM.

[SMR$^+$03] B Solomon, A Mendelson, R Ronen, D Orenstien, and Y Almog. Micro-operation cache: a power aware frontend for variable instruction length ISA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(5):801–811, 2003.

[SMR05] Tingting Sha, M M K Martin, and A Roth. Scalable Store-Load Forwarding via Store Queue Index Prediction. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pages 159–170. IEEE, 2005.

[SPS98] S. Subramanya Sastry, Subbarao Palacharla, and James E. Smith. Exploiting idle floating-point resources for integer execution. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 118–129, New York, NY, USA, 1998. ACM.

[SS97] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In *The 24th Annual International Symposium on Computer Architecture*, pages 194–205, June 1997.

[STR02] A Seznec, E Toullec, and O Rochecouste. Register write specialization register read specialization: a path to complexity-effective wide-issue superscalar processors. In *35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, pages 383–394. IEEE Comput. Soc, 2002.

[SV87] G. S. Sohi and S. Vajapeyam. Instruction issue logic for high-performance, interruptable pipelined processors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, ISCA '87, pages 27–34, New York, NY, USA, 1987. ACM.

[SVNE+15] B Sinharoy, J A Van Norstrand, R J Eickemeyer, H Q Le, J Leenstra, D Q Nguyen, B Konigsburg, K Ward, M D Brown, J E Moreira, D Levitan, S Tung, D Hrusecky, J W Bishop, M Gschwind, M Boersma, M Kroener, M Kaltenbach, T Karkhanis, and K M Fernsler. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development*, 59(1):2:1–2:21, 2015.

[SVS96] Yiannakis Sazeides, Stamatis Vassiliadis, and James E. Smith. The Performance Potential of Data Dependence Speculation & Collapsing. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 238–247, Washington, DC, USA, 1996. IEEE Computer Society.

[SZ05] P Salverda and C Zilles. A Criticality Analysis of Clustering in Superscalar Processors. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pages 55–66. IEEE, 2005.

[TGN01] Weiyu Tang, R Gupta, and a Nicolau. Design of a predictive filter cache for energy savings in high performance processor architectures. In *ICCD International Conference on Computer Design*, pages 68–73. IEEE Comput. Soc, 2001.

[TGT+14] Shelby Thomas, Chetan Gohkale, Enrico Tanuwidjaja, Tony Chong, David Lau, Saturnino Garcia, and Michael Bedford Taylor. CortexSuite: A Synthetic Brain Benchmark Suite. *IISWC*, October 2014.

[Tho65] James E Thornton. Parallel operation in the control data 6600. In *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*, AFIPS '64 (Fall, part II), pages 33–40, New York, USA, Oct 1965. ACM Press.

[TLTC01] E. Tune, Dongning Liang, D. M. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 185–195, 2001.

[TM05] E. Talpes and D. Marculescu. Execution cache-based microarchitecture power-efficient superscalar processors. *IEEE Transactions* *on Very Large Scale Integration (VLSI) Systems*, 13(1):14–26, Jan 2005.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, 11(1):25–33, January 1967.

[TP08] Francis Tseng and Yale N. Patt. Achieving Out-of-Order performance with almost In-Order complexity. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 3–12, Washington, DC, USA, 2008. IEEE Computer Society.

[TSR$^+$98] Vivek Tiwari, Deo Singh, Suresh Rajgopal, Gaurav Mehta, Rakesh Patel, and Franklin Baez. Reducing power in high-performance microprocessors. In *Proceedings of the 35th Annual Design Automation Conference*, DAC '98, pages 732–737, New York, NY, USA, 1998. ACM.

[VAJ$^+$09] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64, Washington, DC, USA, 2009. IEEE Computer Society.

[VM97] Sriram Vajapeyam and Tulika Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 1–12, New York, NY, USA, 1997. ACM.

[Yea96] K. C. Yeager. The Mips R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–41, Apr 1996.

[YERJ99] Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, pages 42–53, Washington, DC, USA, 1999. IEEE Computer Society.

[YO06] Chengmo Yang and Alex Orailoglu. Power-efficient instruction delivery through trace reuse. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 192–201, New York, NY, USA, 2006. ACM.

[ZK01] V V Zyuban and P M Kogge. Inherently lower-power high-performance superscalar architectures. *IEEE Transactions on Computers*, 50(3):268–285, March 2001.

Author's Publications

[HMS$^+$15] Nam Ho, Andrea Mondelli, Alberto Scionti, Marco Solinas, Antoni Portero, and Roberto Giorgi. Enhancing an x86_64 multi-core architecture with data-flow execution support. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 41:1–41:2, New York, NY, USA, 2015. ACM.

[HPS$^+$14] N. Ho, A. Portero, M. Solinas, A. Scionti, A. Mondelli, P. Faraboschi, and R. Giorgi. Simulating a multi-core x86_64 architecture with hardware isa extension supporting a data-flow execution model. In *2014 2nd International Conference on Artificial Intelligence, Modelling and Simulation*, pages 264–269, Nov 2014.

[MHS$^+$15] A. Mondelli, N. Ho, A. Scionti, M. Solinas, A. Portero, and R. Giorgi. Dataflow support in x86_64 multicore architectures through small hardware extensions. In *Digital System Design (DSD), 2015 Euromicro Conference on Digital System Design*, pages 526–529, Aug 2015.

[MMS15] Pierre Michaud, Andrea Mondelli, and André Seznec. Revisiting clustered microarchitecture for future superscalar cores: A case for wide issue clusters. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(3):28:1–28:22, August 2015.

[Mon14] Andrea Mondelli. *Analisi e Valutazione di schemi di Replicazione per Memorie Cache*. OmniScriptum GmbH & Co. KG, 2014.

List of Acronyms

CAM	Content-Addressable-Memory
CISC	Complex Instruction Set Computer
CPI	Cycle per Instruction
CPU	Central Processing Unit
DL1	Level-1 Data Cache
DSP	Digital signal processor
EOLPC	End-of-Loop Program Counter
EPI	Energy per instruction
EU	Execution Unit
FIB	First-Iteration Bit
FIFO	First-In-First-Out
FP	Floating Point
FU	Functional Unit
ID	Instruction Decode
ILP	Instruction-level Parallelism
INT	Integer
IPC	Instruction per cycle
ISA	Instruction Set Architecture
LD	Loop Detector
LDQ	Load Queue
LFST	Last Fetched Store Table
LRT	Loop Reuse Table
LSD	Loop Stream Detector
MPKI	Mispredictions per 1000 Instructions
MSHR	Miss Status Holding Register
OoO	Out-of-Order
PC	Program Counter
PE	Processing Element
RAT	Register Allocation Table

RAW	Read-After-Write
RISC	Reduced Instruction Set Computer
RLT	Redundant Load Table
ROB	Reorder Buffer
RS	Reservation Station
SMT	Simultaneous Multi-Threading
SoC	System-on-Chip
SRAM	Static Random Access Memory
SSID	Store Set Identifier
SSIT	Store Set ID Table
STQ	Store Queue
VLIW	Very long Instruction Word
WAR	Write-After-Read
WAW	Write-After-Write
WB	Write-Back

List of Figures

1 Exemple d'architecture en grappe. 6

2.1 The basic outline of a superscalar execution for instruction from 1 to i and more. In the real implementation, the pipeline stages are more than 5, but the execution of the instructions can still be conceptually divided into few basic phases. In this example two instructions can use the same pipeline phases under ideal conditions, and we have CPI=0.5 15

2.2 Instructions from `0x05` to `0x08` are independent of instructions from `0x01` to `0x04`, but the processor cannot finish the load at `0x05` and `0x06` until the register `eax` is written, otherwise the value written at `Addr1` will be wrong. With the rename of architectural registers to physical registers, the false dependence disappears. . . 19

2.3 A simplifier overview of Tomasulo's Reservation Stations associated with Functional Units. Instructions are fetched and decoded in order, and sent to RS according to their type, thus the type of functional unit required. Using a Reorder Buffer, it is possible to have an in-order commit that helps the handling of precise interrupts. 20

2.4 Difference between conventional cache and trace cache for the dynamic path: I - III - VI. In trace cache any instruction may appear multiple times because if can be part of different traces. In conventional cache the frequency of taken branches limits the maximum bandwidth per memory port. 26

2.5 Each functional unit is connected to read and write ports through latches. Increasing the pipeline depth, the data could be forwarded from different stages of the producer to different stages of the consumer. It requires more multiplexers and more wires. 28

2.6 Simplified Superscalar pipeline. The Clustering allow to increase the issue width of the back-end. 30

2.7 Example of 2-cluster Out-of-Order engine, assuming dispatch-driven steering and register write specialization (which puts steering before register renaming). 32

2.8 In Mod-N steering algorithm the destination cluster alternates every N micro-ops. C0 is Cluster 0, C1 is Cluster 1. 33

2.9 Differences between different register file organizations 36

2.10 Example of bypass network implementation for a dual-cluster OoO engine, assuming 2 register read/write cycles and register write specialization (a single execution port and single source operand are shown, comparators are not shown). The first bypass level (MUX 1) is not impacted by clustering, but this costs an extra cycle of intercluster delay. 38

3.1 A simplified scheme of front-end and back-end in recent Intel microarchitecture. 50

3.2 Baseline Out-of-Order engine with 8 INT execution ports and 3 FP execution ports. The INT register file has 12 read ports and 6 write ports. The FP register file has 5 read ports and 4 write ports. The issue buffers are not shown 53

3.3 Speedup over the baseline for the worst and best configuration in each configuration class. The left graph is the geometric mean speedup for the 12 `SPEC INT` benchmarks, the right graph for the 17 `SPEC FP` benchmarks. The presence of a letter in a configuration's name indicates that the corresponding parameter is doubled compared to the baseline 55

3.4 IPC gain over the baseline for clustered *iiffccw* configurations, for intercluster delays (ICD) of 0,1,2 and 3 cycles, under a Mod-N steering policy with N ranging from 1 to 256. The leftmost bar of each group of bars is the IPC gain when steering all the micro-ops to cluster 0. The rightmost bar is the IPC gain of the non-clustered *ifcw* configuration. The top graph is the average for SPEC INT, the bottom one for SPEC FP. 58

3.5 IPC gain over the baseline for clustered *RBWiiffccw* configurations (Figure 3.4). 59

3.6 IPC gain over the baseline for a clustered *RBWiiffccw* configuration with an intercluster delay of 3 cycles: benchmark per benchmark comparison of Mod-32 steering, Mod-64 and adaptive Mod-N. 60

3.7 Illustration of ILP imbalance on 2 clusters, assuming a null intercluster delay and assuming the average ILP is the square root of the instruction window size. The upper example is for a Mod-4 steering policy, the lower example for Mod-8. In both examples, the instruction window holds 16 instructions. 61

3.8 IPC gain over the baseline for a 4-cluster back-end. Each cluster can issue and execute 2 micro-ops per cycle. Only SPEC INT averages are shown. 62

4.1 Left: inside the while statement we have code that iterates 100 times (It may contains other loops). Center: the assembly version in which the branch is clearer. The loop circled, during the dynamic execution, will be detected and stored in the loop buffer. Right: the dynamic loop. 69

4.2 Loop coverage, assuming $MinIter = 30$. Top graph (a): impact of the loop buffer size. Bottom graph (b): impact of using a Loop Reuse Table, with $MaxBody = 128$. 73

4.3 Varying the size of MinIter on the conservative configuration ($MaxBody = 64$). On top (a) the speedup compared with a conservative baseline with $MinIter = 30$. On bottom (b) the different of MPKI between various MinIter and the conservative baseline with $MinIter = 30$. Benchmarks vertically aligned. 75

4.4 Identification of constant source operands and redundant micro-ops. Red background for loop micro-ops, blue for the rest. The register `rax` is modified only outside the loop, and it is considered as a constant source for micro-ops at address `0x02` and `0x03`. As `rax` is the only source operand for micro-op 3, this micro-op is redundant. The register `rcx` is considered a constant source operand until the next modification (not present in this example) or the loop exit. Micro-ops at address `0x04` and `0x05` consider `rcx` a constant source operand. The latter (micro-op 5) is redundant because the only source operand (`rcx`) is modified by a redundant micro-op (at address `0x03`). The physical registers holding the value produced by micro-ops 3 and 5 are kept alive until the loop exit. For simplicity, we illustrate one micro-op for each decoded instruction (one micro-op = one PC). 78

4.5 Top graph: fractions of non-redundant loop micro-ops, redundant loop loads, and other redundant loop micro-ops. Two bars per benchmark: conservative (left bar) and aggressive loop buffer (right bar). Bottom graph: fractions of non-loop micro-ops, non-redundant loop micro-ops and redundant loop micro-ops. 81

4.6 Speedups enabling redundant micro-ops removal. 82

4.7 Top graph: memory traps per 1000 instructions, for 4 configurations: SSIT=2048 or SSIT=256, with and without global STQ gating. Bottom graph: STQ-miss replays per 1000 instructions when enabling global DL1 gating. 89

4.8 Top graph: memory traps per 1000 instructions, with and without STQ gating enabled for loops. Bottom graph: STQ-miss replays per 1000 instructions when DL1 gating is enabled for loops. Only the conservative loop buffer and "loopy" benchmarks are shown. . 91
4.9 Top graph: fraction of loop load micro-ops that are DL1-gated, STQ-gated and not predicted. Bottom graph: per 1000 instructions, number of non-loop loads, not-gated loop loads, STQ-gated loop loads and DL1-gated loop loads. Only "loopy" benchmarks are shown. 93
4.10 Speedups when STQ gating and DL1 gating are both enabled for loops. 94

5.1 The extra complexity can be introduced incrementally: introducing more integer execution units, then increasing the front-end width, load issue and number of DL1 write ports, and continuing until the issue buffer. SPEC INT results are reported. 98