SPECULATIVE VECTORIZATION FOR SUPERSCALAR PROCESSORS

A Dissertation Presented

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

Doctor per la Universitat Politècnica de Catalunya

by

Alex Pajuelo González

July 2005

SPECULATIVE VECTORIZATION FOR SUPERSCALAR PROCESSORS

Alex Pajuelo González

Thesis advisors:

Antonio González

Mateo Valero

Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya, 2005

A mi familia.

A Laura.

ABSTRACT

Traditional vector architectures have been shown to be very effective in executing regular codes in which the compiler can detect data-level parallelism, i.e. repeating the same computation over different elements in the same code-level data structure.

A skilled programmer can easily create efficient vector code from regular applications. Unfortunately, this vectorization can be difficult if applications are not regular or if the programmer does not have an exact knowledge of the underlying architecture.

The compiler has a partial knowledge of the program (i.e. it has a limited knowledge of the values of the variables). Because of this, it generates code that is safe for any possible scenario according to its knowledge, and thus, it may lose significant opportunities to exploit SIMD parallelism. In addition to this, we have the problem of legacy codes that have been compiled for former versions of the ISA with no SIMD extensions, which are therefore not able to exploit new SIMD extensions incorporated into newer ISA versions.

In this dissertation, we will describe a mechanism that is able to detect and exploit DLP at runtime by speculatively creating vector instructions for prefetching and precomputing data for future instances of their scalar counterparts. This process will be called *Speculative Dynamic Vectorization*.

A more in-depth study of this technique reveals a very positive characteristic: the mechanism can easily be tailored to alleviate the main drawbacks of current superscalar processors, particularly branch mispredictions and the memory gap. In this dissertation, we will describe how to rearrange the basic Speculative Dynamic Vectorization mechanism to alleviate the branch misprediction penalty based on reusing control-flow independent instructions. The memory gap problem will be addressed with a set of mechanisms that exploit the stall cycles due to L2 misses in order to virtually enlarge the instruction window.

Finally, more refinements of the basic Speculative Dynamic Vectorization mechanism will be presented to improve its performance at a reasonable cost.

ACKNOWLEDGEMENTS

First of all, I would like to thank to my thesis advisors Antonio González and Mateo Valero all these years of hard work and unconditional support. In particular, to show me two different points of view of the same thing.

My family has been a great support. This document is the answer to your repetitive question. Specially, I thank Laura for her infinite patience during all these years. Finally, I have finished it!

The PhD students of the Computer Architecture Department have proved to be very good friends. In any order: Jaume Abella, Javier Verdu, Francisco Cazorla, Oliver Santana, Jordi Guitart, Josep Maria Codina, Tanausu Ramírez, Carmelo Acosta, Ayose Falcon, Germán Rodríguez, Marco Galluzi, Enric Gibert, Josep Aguilar, Victor Mora, Jaume Vila, José Lorenzo Cruz, Ramon Canal, Joan Manel Parcerisa, Daniel Jiménez and Alex Settle. Thanks for all those coffee breaks and for those launch parties.

David López, thank you for show me that a researcher is always a person. Thank you for teach me how to teach. I hope I recover my peace within as you did!

Of course, I would like to thank all those anonymous reviewers who accept/reject my papers. All their comments are valuables and have been had into account to improve the work.

Finally, I would like to thank you, anonymous reader, for your time reading this dissertation. I hope you enjoy it as much as I enjoyed write it.

This work has been partially supported by the Ministry of Education and Science under grants TIN2004-07739-C02-01 and TIN2004-03072, the CICYT project TIC2001-0995-C02-01, the Feder Funds, Intel Corporation and the European Network of Excellence on High Performance Embedded Architecture and Compilation

(HiPEAC).

TABLE OF CONTENTS

Chapter 1

Introduction

1.1 MOTIVATION

Nowadays, the software-hardware cooperation present in current computers seeks to maximize the throughput of applications. On the one hand, a software compiler translates applications from a high-level to a low-level language that the processor can comprehend. Furthermore, during the translation process, the compiler interprets the code in order to apply, if possible, optimizations to improve the performance of the application. However, due to the fact that the compiler has a limited knowledge of the application, the low-level code that is generated is conservative which reduces the opportunities to exploit the parallelism that is available.

On the other hand, hardware processors include mechanisms for efficiently executing the low-level coding of applications. These mechanisms study the code at runtime to detect and exploit several levels of parallelism, even if they are not clearly exposed by the compiler.

This software-hardware cooperation is more effective when the compiler has an extensive knowledge of the applications and/or the target processors. In these cases the compiler can easily detect and pass on semantic information on the application to the processor through a specialized ISA. This information helps the processor to be more effective in executing applications.

Scientific scenarios can be good examples of this cooperation. The main characteristic of scientific applications is that large amounts of *Data Level Parallelism* (DLP) are exposed by the high level language. This parallelism consists in repeating the same computation on different elements in large data structures such as vectors or matrices. This DLP can easily be detected by a vectorizing compiler that is able to pass this semantic information through a *Single Instruction Multiple Data* (SIMD) ISA to a vector processor. This processor is very effective in executing this kind of code since it provides wide resources in which several elements of a vector can be held and computed in parallel.

Superscalar processors are a good alternative for executing applications with a moderate amount of DLP. These processors rely on compilers to expose *Instruction Level Parallelism* (ILP). This fine-grain parallelism seeks to execute small sections of the code, usually instructions, in parallel. Furthermore, these processors include mechanisms for speculatively executing portions of the code to expose more ILP. Recently, vector-like resources have been included in these processors to efficiently exploit the DLP exposed by the compiler in multimedia applications.

Finally, multithreading, multicore or multiprocessor approaches all benefit from DLP and ILP and, furthermore, they exploit *Thread Level Parallelism* (TLP) in which large sections of the code can be executed in parallel.

This dissertation describes a hardware enhancement of a superscalar processor to dynamically expose and exploit the DLP of codes in which the compiler failed to detect this parallelism. This mechanism searches for vector patterns in the code to create speculative vector instructions that will prefetch and precompute data for non-speculative instructions. This preexecution exploits the newly discovered ILP and DLP and thus improves the performance of applications.

1.1.1 Speculative execution

One way of optimizing execution in scenarios with limited parallelism is to include speculative execution mechanisms in the processor. Tasks that are likely to be executed in the future can be speculatively performed in advance to reduce the latency of non-speculative instructions.

This way of executing instructions can produce a net increase in the processor's performance when speculation is correct since all the speculative work can be reused by non-speculative instructions.

Unfortunately, when a misspeculation is detected, a mechanism for rolling back the speculative work must be provided to ensure the code has been executed correctly.

This useless speculative execution is not for free since squashed instructions have allocated resources and consumed energy. Therefore, it is necessary to find a trade-off between speculation, efficiency, performance and resource usage. An excessive amount of speculative instructions can overload the execution stage of the processor delaying non-speculative instructions and reducing performance. However, if speculative work is negligible or non-existent and hardly increases the throughput of the system, it may be pointless to complicate the design of the processor by including speculative mechanisms.

Branch prediction and data prefetch are the most common speculative mechanisms in current processors.

1.1.2 Vector execution

Vector processors [ea95] [Asa98] [Esp97] [LD97] [Rus78] are the best choice for executing scientific applications in which a large amount of DLP is present, this is performing the same computations over different elements in a large storage structure such as vectors or matrices. Vector compilers [ZC90] [AK87] detect this parallelism and pass the semantic information through a *SIMD ISA* to the processor. The main characteristic of SIMD instructions is that a single instruction deals with a set of elements in a given structure, which improves the ratio between the fetching and execution of instructions. Therefore, vector processors rely on the vector compiler to create a code that executes efficiently. Speculative mechanisms, other than branch prediction, are rarely used in vector scenarios.

Furthermore, vector architectures overcome many of the problems present in superscalar processors. Vector ISAs are the best and easiest way to manage large amounts of hardware. Vector memory accesses are capable of hiding the memory latency by efficiently using the available memory bandwidth. Finally, since conditional branches can be replaced by vector mask registers and predicated execution, branch mispredictions are reduced.

Nowadays, scientific applications are used in particular environments. The general purpose processors that are used in personal computers are not usually used to execute this kind of application. At this point, one may legitimately think that it would be pointless to make the effort to detect and exploit DLP. However, this is not so. A few years ago due to the evolution of the technology, a new kind of application appeared on the scene: *Multimedia Applications*. These applications transform streamed data into video, music and/or still images. In these transformation processes, as in scientific applications, the storage structures that are typically used are vectors and matrices.

Processor manufacturers noticed this and decided to include vector-like functional units, called *Multimedia Extensions* in general purpose processors. These extensions are aimed at increasing the performance of multimedia applications by exploiting the inherent DLP. MMX [Int99], SSE and SSE2 [Int02] by Intel Corp. and 3DNow! [AMD99] by AMD are examples of this.

However, when the compiler fails to detect DLP multimedia extensions become useless. From Chapter 3 onwards, a mechanism that dynamically exposes DLP will be presented. This mechanism is able to create speculative vector instructions that are executed in multimedia extensions, which boosts the performance of any application.

1.1.3 Thesis objectives

Are integer applications really non-vectorizable? The fact that a specialized compiler is not able to detect vector patterns does not necessarily mean that these patterns are not present. Usually, a compiler does not have enough knowledge of a given application to detect the underlying DLP.

In this dissertation we claim firstly that all applications, even if they are not vectorizable, present vector patterns. Secondly, we show that these patterns can easily be detectable at runtime using speculative techniques. We prove that mixing the best of the two worlds (speculation and vector execution) results in an explosion of ILP and DLP that improves the usage of the memory bandwidth and therefore the performance of the processor.

To accomplish this we describe a novel mechanism that is able to find vector patterns dynamically in codes in which the compiler failed to detect DLP. This mechanism takes advantage of vector functional units to speculatively execute the vector instructions created.

We will then show that vector instructions can be dispensable. A scalar implementation of the presented mechanism, i.e. without vector resources, simplifies the processor without penalizing the performance of the original vector mechanism.

Later on, we show that the mechanism presented can easily be tailored to deal with critical performance degrading sources in current processors. In particular, branch misprediction and the memory gap will be alleviated by rearranging of the basic mechanism.

Finally, in order to show that the mechanism presented can easily be implemented in current processors, we describe successive power and resource aware refinements of the mechanism.

1.2 THESIS OVERVIEW

This dissertation describes all the work done in this thesis: firstly, designing and improving the *Speculative Dynamic Vectorization* (SDV) mechanism, and secondly, rearranging the mechanism to alleviate the main sources of performance degradation in current superscalar processors.

For the sake of readability, we base the explanation of the following sections on the code in Figure 1.1.

1.2.1 Speculative dynamic vectorization

We motivate this thesis providing statistics that show that, on average, in codes compiled for a superscalar processor, independently of the nature of the application, 30% of the scalar instructions can be translated into vector code. The first [PGV02] and second Speculative Dynamic Vectorization mechanisms developed in this thesis are described. These mechanisms will speculatively create vector instructions,

```
for (i=0; i<N; i++)
{
  a+=elem->n;
  if (a%2==0)
  {
    a++;
  }
  a+=b[i];
  elem=elem->next;
}
```

(a) Original code

```
I0:   MOV   R4, 0
I1:   LD    R1, R0[n]
I2:   ADD   R2, R2, R1
I3:   AND   R3, R2, 0x1
I4:   CMP   R3, 0
I5:   BNE   I7
I6:   INC   R3
I7:   LD    R5, b[R4]
I8:   ADD   R3, R3, R5
I9:   LD    R0, R0[next]
I10:  INC   R4
I11:  CMP   R4, N
I12:  BL    I1
```

(b) Assembler transformation

Figure 1.1: Example code

```
I0:     MOV      R4, 0
I1:     LDV      V1, R0[n]
I2:     ADDVS    V2, R2, V1
I3:     ANDV     V3, V2, 0x1
I4:     CMPV     V3, 0
I5:     BNE      I7
I6:     INCV     V3
I7:     LDV      V5, b[R4]
I8:     ADDV     V3, V3, V5
I9:     LD       R0, R0[next]
I10:    INC      R4
I11:    CMP      R4, N
I12:    BL       I1
```

(a) Dynamic Vectorization result code

```
I0:     MOV      R4, 0
I1:     LD       R1, R0[n]
I2:     ADD      R2, R2, R1
I3:     AND      R3, V2, 0x1
I4:     CMP      R3, 0
I5:     BNE      I7
I6:     INC      R3
I7:     LDV      V5, b[R4]
I8:     ADDV     R3, R3, V5
I9:     LD       R0, R0[next]
I10:    INC      R4
I11:    CMP      R4, N
I12:    BL       I1
```

(b) CI-selection code

```
I0:     MOV      R4, 0
I1:     LDV      V1, R0[n]
I2:     ADDVS    V2, R2, V1
I3:     ANDV     V3, V2, 0x1
I4:     CMPV     V3, 0
I5:     BNE      I7
I6:     INCV     V3
I7:     LD       R5, b[R4]
I8:     ADDV     R3, V3, R5
I9:     LD       R0, R0[next]
I10:    INC      R4
I11:    CMP      R4, N
I12:    BL       I1
```

(c) Memory gap selection code

Figure 1.2: Possible transformations of the DV mechanism

whenever a vector pattern is detected. For the mechanisms, a vector pattern begins with a strided load.

Furthermore, we evaluate the mechanisms for a wide range of configurations enumerating the sources of benefit: prefetch, preexecution, DLP efficiency through wide resources and virtual enlargement of the instruction window.

In the example code in Figure 1.1 a vector compiler does not create vector code since it does not know *a priori*, at compile time, the memory position for every *elem* of the structure. However, this detection can easily be performed at runtime since the memory address of every *elem* is known. The procedure is simple. The Speculative Dynamic Vectorization mechanism studies the effective addresses of every executed load and as soon as a strided pattern is detected, for example the strided load in instruction *I1*, a speculative vector instruction is created. Dependent instructions (*I2*, *I3*, *I4*, *I6*, *I7* and *I8*) are also vectorized by propagating the characteristic of "vectorization" down the dependence graph. The resulting code is shown in Figure 1.2(a). Further details are available in Chapter 3.

1.2.2 Reducing the penalty of branch mispredictions

We show that the SDV mechanism can easily be re-targeted to alleviate the effects of some performance degrading sources in current processors.

The mechanism is tailored to select *control-flow and data independent* instructions for vectorization [PGV05a]. We introduce the concept of *Selective Speculative Dynamic Vectorization*. In this case, only Control-Flow independent instructions are selected for vectorization.

This selection decreases the number of speculative computations the mechanism performs and thus reduces the resource requirements.

Moreover, we justify why control-flow and data independent instructions are the best choice for applying the SDV mechanism.

The Control-Flow independent scheme will only vectorize those strided loads whose dependent instructions are control-independent. In the example code in Figure 1.1, Instruction *I6* and all its dependent instructions are control-dependent. Therefore, the selection mechanism will prevent these instructions from being vectorized.

To accomplish this, only instructions after the *Reconvergence Point* (instruction *I7*) of a hard-to-predict branch (instruction *I5*) are considered vectorization. Of these instructions, only those that are data-independent (*I7* and *I8*) will be vectorized. To vectorize these instructions, the mechanism will vectorize the strided load that creates the source operands of these instructions. The resulting code is shown in Figure 1.2(b). Further details can be found in Chapter 4.

1.2.3 Overcoming the memory gap

The memory gap [WM95] is one of the most critical performance degrading sources in current processors.

We show how to take advantage of the stall cycles of the processor that result from a long latency load.

We describe the third implementation of the *Speculative Dynamic Vectorization* mechanism. In this case, a small separated core that implements the mechanism is presented.

To alleviate the memory penalty we provide two different mechanisms [PGV05c] [PGV04]. The first mechanisn is based on the control-flow independence mechanism but selection and vectorization are performed as soon as the delinquent load is detected. The second mechanism waits until an L2 miss load becomes the oldest instruction in the instruction window to fire the vectorization. Both mechanisms only vectorize the independent instructions of the L2 miss load that fired the mechanism.

We show in the evaluation of this part that even if the performance obtained for every mechanism is nearly equal, the second mechanism is the best choice since it is resource-aware.

Supposing that in Figure 1.1 instruction *I7* is the L2 miss load, the mechanism will only select those instructions *I1*, *I2*, *I3*, *I4*, *I6* and *I8* that are independent of that delinquent load. As in the other mechanisms, only instructions dependent of a strided load can be vectorized. Figure 1.2(c) shows the final transformation of the code in Figure 1.1 when the selection scheme is applied to alleviate the memory gap. Further details can be found in Chapter 5.

1.2.4 Cost-effective dynamic vectorization

Speculation mechanisms are not perfect. Misspeculations can degrade performance if they are not correctly and efficiently rolled back.

In this chapter the main sources of misspeculations of the Speculative Dynamic Vectorization mechanism are enumerated. Heuristics for reducing these sources are presented and evaluated to enhance the effects of the dynamic vectorization mechanisms[PGV05b].

1.2.5 Mechanism comparison

In order to conclude this dissertation, since many implementations are described, we provide a brief comparison to evaluate which is the best mechanism in terms of performance, hardware requirements and energy consumption.

Since the motivation for every proposal is different, a global ranking of the mechanisms will not be provided.

1.3 STRUCTURE OF THIS WORK

The structure of this dissertation is as follows:

- Chapter 2 presents our simulation environment. The reference platform for this work is presented, as well as the benchmarks and tools used in this thesis.
- Chapter 3 describes the basic *Speculative Dynamic Vectorization* mechanism. The second part of the chapter justifies and presents an evolved scalar version of the basic mechanism. A completed evaluation of the scalar version is provided in the third part of the chapter.
- Chapter 4 introduces the concept of *Selective Dynamic Vectorization.* This chapter proposes some modifications to the mechanism to alleviate branch penalties by detecting and preexecuting control-flow independent instructions.
- Chapter 5 focuses on the memory gap. To alleviate this problem we provide a set of mechanisms that exploit the processor stall cycles caused by an L2 miss load to virtually enlarge the instruction window.
- Chapter 6 analyzes the sources of misspeculation in the mechanism. Based on this study, a set of heuristics for reducing the register requirement of the basic DV mechanism will be provided. This reduction will be reflected in terms of extra instructions.

- Chapter 7 summarizes this dissertation by comparing all the mechanisms presented in this dissertation in terms of hardware, performance and energy consumption.
- Chapter 8 concludes this dissertation by remarking on the most important points of this thesis and providing a brief future work.

Chapter 2

Platform and Benchmarks

2.1 INTRODUCTION

This chapter describes the baseline processors in our study. Firstly, we consider that the baseline processor is a superscalar processor with vector capabilities. It is assumed that these extensions are used to execute SIMD instructions belonging to the processor's ISA. It is out of the scope of this dissertation to simulate these instructions. Instead, we assume that these instructions are available but that the compiler has not been able to find vector patterns in the code to insert them.

The second proposal of the *Speculative Dynamic Vectorization* mechanism is based on a superscalar processor without vector capabilities. We will show in subsequent chapters that vector resources are indeed not needed to maintain the performance benefits of the first mechanism.

Later on, we will discuss the reason behind the choice of benchmarks. The main characteristics of each set of benchmarks will be explained to justify the following evaluations. We will see why SpecINT2K and SpecFP2K are radically different and how their characteristics fit into the scope of this thesis.

2.2 REFERENCE PLATFORM

As mentioned in the introduction, this dissertation presents a mechanism that is able to create speculative vector instructions for prefetching and precomputing data for non-speculative instructions. The first design of the mechanism is built on top of an out-of-order superscalar processor with vector capabilities. A scheme of this processor is depicted in Figure 2.3.

2.2.1 Scalar pipeline

The scalar pipeline of the processor is divided into 6 stages: fetch, decode, issue, execution, writeback and commit. All the stages, except execution, take 1 cycle to complete. Execution latency depends on the instruction.

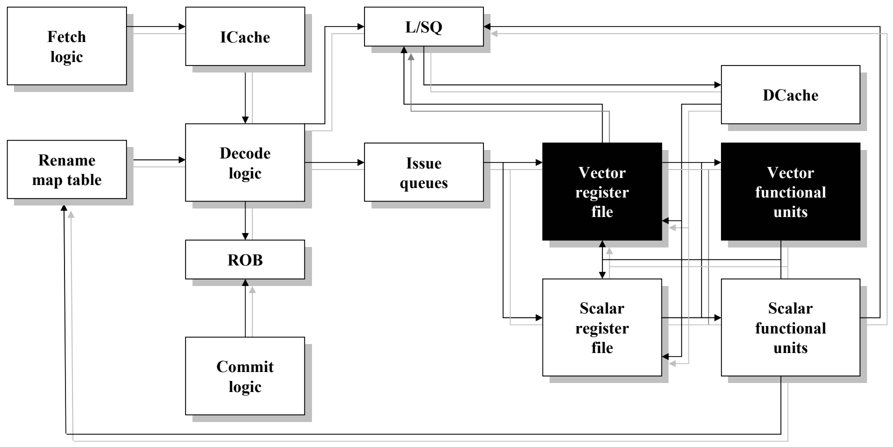

Figure 2.3: Baseline processor with vector capabilities

As we will show in subsequent chapters, the Speculative Dynamic Vectorization mechanism can easily be accommodated in this pipeline. The main structures (stride predictor and SRSMT) of the SDV mechanism are out of the critical path, which allows long latencies when they are being accessed. Furthermore, since they are addressed with the PC of an instruction, and the data provided must be supplied to the decode logic, the access to these structures can be overlapped with the fetch and decode of the instructions. Therefore, the access to the DV structures can be overlapped with the fetch and decode of the instructions.

Nowadays technology allows latencies of 2 cycles when structures of 8Kbytes in a 3Ghz processor are accessed. In the DV mechanism, the largest structure measures 10Kbytes and it is accessed from the fetch to the decode stage, which means that there is a minimum latency of 2 cycles. Superpipelining of these stages allows longer latencies for these structures, which leads us to believe that our design is easily affordable for current and future processors. In fact, figures obtained with Cacti3.0 [SJ01] show that these structures can be accommodated in a 5Ghz processor implemented in 0.13 nanometers (IBM-Sony [ea05] plan to release the The Cell processor with a 4.5Ghz clock frequency). To standardize the baseline, we will suppose that the processor's clock raises 2.4Ghz.

The configuration of the scalar pipeline is summarized in Table 2.1. Parameters

Parameter	**Value**
Fetch Width	8 instructions (up to 1 taken branch)
I-Cache	64KB, 2-way set associative, 64 byte lines, 1 cycle hit, 6 cycle miss time
Branch Predictor	Gshare with 64K entries
Inst. Window Size	256 entries
Load/store queue	64 entries with store-load forwarding
Issue mechanism	8-way out of order issue; loads may execute when prior store addresses are known
Scalar functional units (latency in brackets)	6 simple int(1); 3 int mult/div (2 for mult and 12 for div); 4 simple FP(2); 2 FP mult/div (4 for mult and 14 for div); 1 load/store
D-cache	64KB, 2-way set associative, 32 byte lines, 2 cycle hit time, write-back, 10 cycle miss time, up to 16 outstanding misses
L2 cache	256KB, 4-way set associative, 64 byte lines, 10 cycle hit time, 1000 cycle miss time
Commit Width	8 instructions

Table 2.1: Configuration parameters

remain unchanged for all evaluations except when explicitly stated otherwise.

2.2.2 Vector capabilities

It may seem striking that the baseline processor presents real vector resources and not just multimedia extensions. One of the main drawbacks of current multimedia extensions is the limited range of computable values that they allow (only up to 32 bits can be used per value). The lack of 64-bit support should not be blamed on the processors' designers: the main target for multimedia extensions is mainstream multimedia software, especially 3D games, in which the precision difference between 32-bit and 64-bit FP computations would hardly be noticeable. However, for scientific applications, in which precision is important, more powerful floating point functional units are required. For this reason, we really believe that future processors will enhance their multimedia extensions to provide more developed vector support.

A detailed scheme of the processor's execution engine is provided in Figure 2.4.

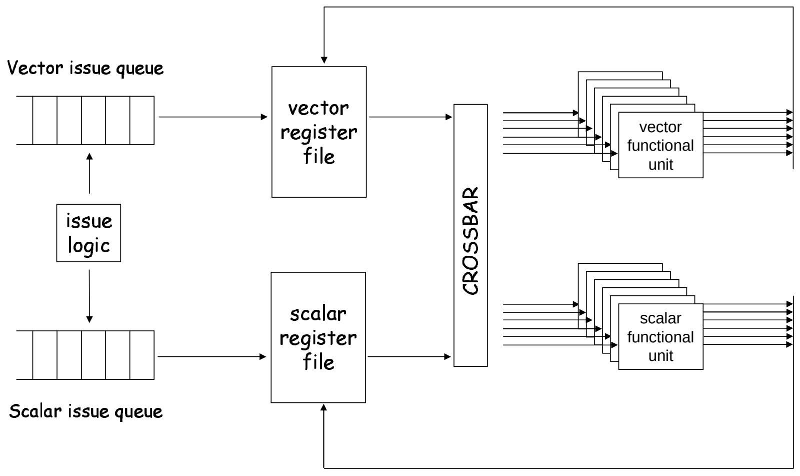

Figure 2.4: Detail of the execution engine

To completely separate the scalar and vector engine, two issue queues are implemented, one for scalar instructions and the other for vector instructions. These queues are not connected and therefore bypasses among instructions of different kinds are disallowed. Scalar instructions only get vector results from the vector register file. Vector instructions that need scalar values, obtain them from the scalar register file.

The baseline processor has a complete vector register file with 64-bit 4 element vector registers connected to the vector and scalar functional units through an issue-bound crossbar. Vector functional units are designed as scalar functional units, i.e. only one element per register can be computed at once. Neither vector nor scalar functional units can write results in the other register file.

From this design one can see that the execution engine has become extremely complex since two kinds of register files and functional units are interconnected in order to extend the processor with vector capabilities. This will be explained in further detail in Chapter 3 and will justify the complete removal of these vector extensions.

2.3 BENCHMARKS

Benchmark behavior definition is an important aspect that must be defined to understand how performance is achieved. The base *Dynamic Vectorization* mechanism described in this thesis is not tailored to fulfill the characteristics of a particular kind of application but rather it is designed to work in a wide range of different codes.

The following rearrangements of the technique exploit relevant characteristics that must be taken into account.

Our performance figures are obtained by simulating the whole Spec2K benchmark suite. This set of applications includes a wide range of codes with different behaviors, although they can be grouped into two subsets: irregular and regular codes. A summary of their main characteristics is provided.

2.3.1 Irregular codes

This group includes the most typical applications used in desktops. Since this is a large group it must be taken into account when our mechanism is being tested. It would not be advisable that for *Dynamic Vectorization* to lose performance when it executes these applications since this would decrease the chances of incorporating the mechanism into general purpose processors, such as those that are provided in desktop computers.

Due to their behavior, this group of codes is the main challenge and the most important justification for the mechanisms proposed. The most important characteristics of these codes are as follows:

- Loops: Small with an irregular number of iterations (6,5 iterations on average).

Thanks to compilation optimization techniques such as inlining, loop bodies can be large.

- Branches: Unpredictable behavior for a considerable percentage of branches: about 4% of the total.
- DLP: Memory structures are, in most cases, small and easily allocatable in the available on-chip memory. L2 misses are not frequent.
- Vectorization: Memory instructions do not present vector patterns at compile time. Vectorization is not possible.
- Benchmarks: SpecINT2000 [Spe00].

2.3.2 Regular codes

Scientific and multimedia applications are the most representative sets that constitute this group. Scientific applications are employed in very restricted environments but multimedia is present in the day-to-day use of a desktop computer. Audio and video stream, 3D games and photo editors are typical applications in this group. Since these applications are easily vectorizable, one may argue that using *Dynamic Vectorization* in this context is tricky. Note that these applications are basically compiled with a scalar compiler that is not good at detecting vector patterns. However, it is possible that the source code of vectorizable applications will not be available for recompiling it to allow vector support. For these reasons, *Dynamic Vectorization* could help by creating vector code at runtime in order to use those idle multimedia extensions.

The main characteristics of this group of applications are as follows:

- Loops: Large and usually with a constant number of iterations. The loop of the body consists of hundreds of instructions.

- Branches: Most of the branches are easily predictable since they are branches that finish the loop body.
- DLP: Large vector and/or matrices are the most typical structures found. DLP is clearly present since loops traverse these structures element by element.
- Vectorization: Since loops are regular and DLP is exposed in high level languages, vector compilers are very effective in creating optimized vector code.
- Benchmarks: SpecFP2000 [Spe00].

2.4 SIMULATION FRAMEWORK

To evaluate our proposal we use SimpleScalar v3.0d [BA97] as a microarchitectural simulator. SimpleScalar simulates an out-of-order superscalar processor with a pipeline of 6 stages: fetch, decode, issue, execution, writeback and commit. Each stage takes 1 cycle, except for execution whose latency depends on the instruction. This simulator has been extended to implement the mechanisms described.

SimpleScalar is able to execute Alpha applications by emulating the ISA, which is a highly representative *instruction set architecture* of a current *RISC* processor. In order to generate code for it, we used gcc 2.8.1 for Digital Unix. All Spec2K benchmarks were compiled with all the optimizations enabled, using the following flags: *-non_shared -ifo -O3.*

For simulations, we chose the following procedure: for every benchmark we skipped the initialization part and simulated the next 100M of instructions. No Spec was executed until completion.

Chapter 3

Speculative Dynamic Vectorization

3.1 INTRODUCTION

This chapter motivates the basis of the work developed in this thesis. Figures will show that both irregular and regular codes present at runtime enough vector patterns, represented as strided loads, to make dynamic vectorization worth. As said in Chapter 2 all the benchmarks are compiled with a superscalar compiler. This means that we are simulating that the compiler has not been able to find vector patterns to create vector instructions even if the processor provides vector resources.

The two main implementations of the proposal are provided. The first architecture is based on the baseline with vector resources. The second overcomes the hardware drawback of having two different register files. Moreover, vector functional units are also removed from the baseline. This new layout oblige to change how the mechanism works, although the philosophy underneath remains unmodified. In this case, the mechanism will not *vectorize* but *replicate*.

3.2 MOTIVATION

Strided memory loads [Gon00] are the instructions that fire the proposed speculative dynamic vectorization mechanism. To identify a strided load, at least three dynamic instances of the static load are needed. The first dynamic instance sets the first memory address that is accessed. The second dynamic instance computes the initial stride, subtracting the memory address of the first dynamic instance from the current address. The third dynamic instance checks if the stride is repeated computing the current stride and comparing it with the first computed stride.

Figure 3.5 shows the stride distribution for SpecINT2K and SpecFP2K (for this figure, the stride is computed dividing the difference of memory addresses by the size of the accessed data).

As shown in Figure 3.5, the most frequent stride for SpecINT2K and SpecFP2K is 0. This means that dynamic instances of the same static load access the same

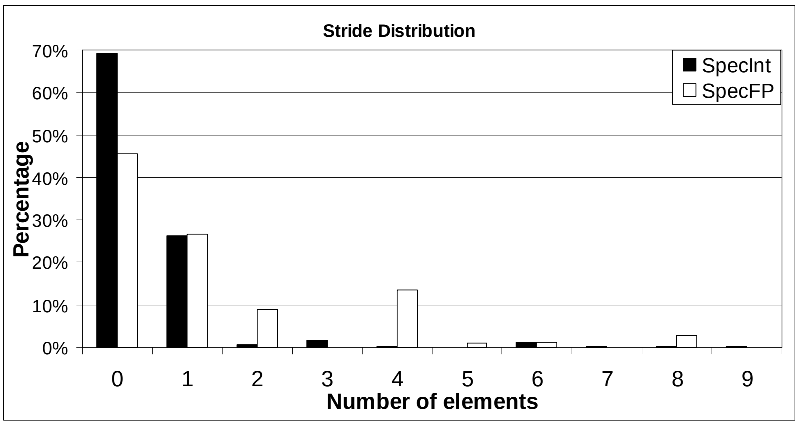

Figure 3.5: Stride distribution for Spec2K

memory address. For SpecINT this stride is due, mainly, to the accesses of local variables and memory addresses referenced through pointers. For SpecFP the stride 0 is mainly due to spill code.

Usually, for SpecFP, the most frequent stride is stride 1 because these applications execute the same operations over every element of some array structures. However, due to the code optimizations [BGS93] [Ken78] included by the scalar compiler, such as loop unrolling, some stride 1 accesses become stride 2, 4 or 8. The bottom line of this statistics is that strided accesses are quite common both in integer and FP applications.

The results in Figure 3.5 also suggest that a wide bus to the L1 data cache can be very effective at reducing the number of memory requests. For instance, if the cache line size is 4 elements, multiple accesses with stride lower than 4 can be served with a single request if the bus width is equal to the line size. These types of strides represent 97,9% and 81,3% of the total strided loads for SpecInt2K and SpecFP2K respectively.

3.3 RELATED WORK

Dynamic vectorization is not a very extended topic at literature. In fact, only two proposals can be referred as related work on this subject: dynamic vectorization in trace processors and the CONDEL architecture.

Vajapeyam [VJM99] presents a dynamic vectorization mechanism based on trace processors. The mechanism executes in parallel some iterations of a loop. This mechanism tries to enlarge the instruction window capturing in vector form the body of the loops. The whole loop body is vectorized provided that all iterations of the loop follow the same control flow. The mechanism proposed in this paper is more flexible/general in the sense that it can vectorize just parts of the loop body and may allow different control flows in some parts of the loop.

The CONDEL architecture [Uht92] proposed by Uht captures a single copy of complex loops in a static instruction window. It uses state bits per iteration to determine the control paths taken by different loop iterations and to correctly enforce dependences.

The use of wide buses has been previously considered to improve the efficiency of the memory system for different microarchitectures [RTDA97][WD94b][WO01].

Rotenberg et al. present a mechanism to exploit control flow independence in superscalar [RJS99] and trace [RS99] processors. Their approach is based on identifying control independent points dynamically, and a hardware organization of the instruction window that allows the processor to insert the instructions after a branch misprediction between instructions previously dispatched, i.e., after the mispredicted branch and before the control independent point.

Lopez et al. [LLVA98] propose and evaluate aggressive wide VLIW architectures oriented to numerical applications. The main idea is to take advantage on the existence of stride one in numerical and multimedia loops. The compiler detects load instructions to consecutive addresses and combines them into a single wide load instruction that can be efficiently executed in VLIW architectures with wide buses.

The same concept is applied to groups of instructions that make computations. In some cases, these wide architectures achieve similar performance, compared to architectures where the buses and functional units are replicated, but at reduced cost.

3.4 OVERVIEW

Speculative dynamic vectorization begins when a strided load is detected. When this happens, a vectorized instance of the instruction is created and it is executed in a vector functional unit storing the results in a vector register. Next instances of the same static instruction are not executed but they just validate if the corresponding speculatively loaded element is valid. This basically consists in checking that the predicted address is correct and the loaded element has not been invalidated by a succeeding store. Every new instance of the scalar load instruction validates one element of the corresponding destination vector register.

Arithmetic instructions are vectorized when any of the source operands is a vector register. Succeeding dynamic instances of this instruction just check that the corresponding source operands are still valid vector elements (details on how the state of each element is kept is later explained).

When a validation fails, the current and following instances of the corresponding instruction are executed in scalar mode, until the vectorizing engine detects again a new vectorizable pattern. With this dynamic vectorization mechanism, as shown in Figure 3.6, with unbounded resources, 29% of the SpecINT2K instructions and 30% of the SpecFP2K instructions, on average, can be vectorized. It is important to note that the percentage of vectorizable instructions varies significantly from one benchmark to another.

The following example shows how the mechanism works. Imagine we have a code like that in Figure 3.7.

Figure 3.7(a) shows the typical code a superscalar compiler will create. In this

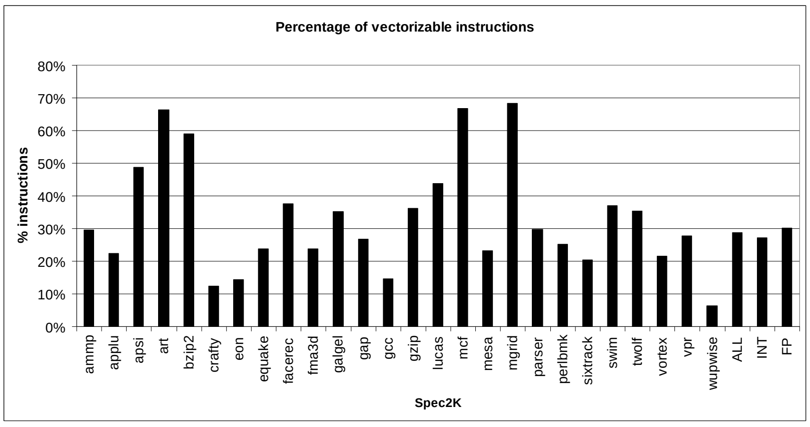

Figure 3.6: Percentage of vectorizable instructions

```
I1:  LD  R1, a[R0]
I2:  LD  R2, b[R1]
I3:  ADD R3, 3, R1
I4:  ADD R4, R1, R2
I5:  ST  c[R0], R3
I6:  ADD R0, 4
I7:  CMP R0, 1000
I8:  JNE I1
```

(a) Original code

```
I1:  LD  V1, a[R0]
I2:  LD  V2, b[V1]
I3:  ADD V3, 3, V1
I4:  ADD V4, V1, V2
I5:  ST  c[R0], V3
I6:  ADD R0, 4
I7:  CMP R0, 1000
I8:  JNE I1
```

(b) Transformed code

Figure 3.7: Dynamic vectorization transformation

code, the stride predictor will detect instruction I_1 as a strided load, which offset is 4 bytes, but not I_2 since it presents an irregular access pattern. As soon as the DV mechanism detects that I_1 is a strided load, a vector register (V_1 in the example) is allocated. Dependent instructions of this load (I_2, I_3 and I_4) will be also vectorized since the "vectorization" characteristic is propagated down the dependence graph. A vector register will be also allocated for these instructions.

Stores are not vectorized since it could be difficult to rollback a memory change. But the mechanism must be able to supply the corresponding position inside a vector register to these instructions. For this reason, the renaming logic of the decode stage is modified to deal with both vector and scalar registers.

Branches are neither vectorized to simplify the design of the mechanism.

After applying our mechanism to code in Figure 3.7(a) the new code looks like the one shown in Figure 3.7(b) where vector instructions are created to precompute speculative data for being used later by their scalar counterpart.

3.5 FIRST APPROACH: DYNAMIC VECTORIZATION WITH VECTOR RESOURCES

As previously said, the first implementation of the mechanism is built on a superscalar processor with vector capabilities to execute the speculatively created vector instructions.

3.5.1 Instruction vectorization

The first step to create vector instances of an instruction is detecting a strided load. To do this, it is necessary to know the recent history of memory accesses for every load instruction. To store this history the processor includes a stride predictor where, for every load, the PC, the current address, the stride and a confidence counter are stored as shown in Figure 3.8.

PC	Last Address	Stride	Confidence Counter

Figure 3.8: Entry of the stride predictor

When a load instruction is decoded, it looks for its PC in the stride predictor. If the PC is not in this table, the last address field is initialized with the current address of the load and the stride and confidence counter fields are set to 0.

Next dynamic instances compute the new stride and compare the result with the stride stored in the table, increasing the confidence counter when both strides are equal or resetting it to 0 otherwise. When the confidence counter is 2 or higher, a new vectorized instance of the instruction is generated. The last address field is always modified with the current memory address of the dynamic instance.

When a vectorized instance of an instruction is generated, the processor allocates a vector register to store its result. The processor maintains the associations of vector registers and vector instructions in the *Vector Register Map Table*. This table contains, for every vector register the PC of the associated instruction, the vector element (*offset*) corresponding to the last fetched scalar instruction that will validate (or has validated) an element of this vector, the source operands of the associated instruction, and, if the instruction is vectorized with one scalar operand and one vector operand, the value of the scalar register is also stored, as shown in Figure 3.9.

PC	Offset	Source Operand 1	Source Operand 2	Value

Figure 3.9: Entry of the Vector Register Map Table

Every time a scalar instruction is fetched, this table is checked and if its PC is found the instruction is turned into a validation operation. In this case, the *offset* field determines which vector element must be validated and then, the *offset* is incremented. In the case that the *offset* is equal to the vector register length, another vectorized version of the instruction is generated and a free vector register is allocated to it. The VRMT table entry corresponding to this new vector register is initialized with the content of the entry corresponding to the previous instance, excepting the field *offset*, which is set to 0.

The register rename table is also modified (see Figure 3.10) to reflect the two kind of physical registers (scalar and vector). Every logical register is mapped to either a physical scalar register or a physical vector register, depending on whether the last instruction that used this register as destination was vectorized or not. Every entry of the table contains a V/S flag to mark if the physical register is a scalar or a vector register and the field offset indicates the latest element for which a validation has entered in the pipeline.

PC	Offset	Source Operand 1	Source Operand 2	Value

Figure 3.10: Entry of the modified rename map table

When every instruction is decoded the V/S flags (vector/scalar) of their source operands are read and if any of the two is set to V, the instruction is vectorized. In parallel, the VRMT table is accessed to check if the instruction was already vectorized in a previous dynamic instance. If so, the instruction is turned into a validation operation. Validation is performed by checking if the source operands in the VRMT table and those in the rename table are the same. If they differ, a new vectorized version of the instruction is generated. Otherwise, the current element pointed by *offset* is validated and this validation is dispatched to the reorder buffer in order to be later committed (see next section for further explanation). Besides, if the validated element is the last one of the vector, a new instance of the vectorized instruction is dispatched to the vector data-path.

Arithmetic instructions that have been vectorized with one vector source operand and one scalar register operand, wait in the decode stage, blocking the next instructions, until the value of the physical register associated to the scalar source operand is available. Then, it checks if the value of the register matches the value found in the VRMT and if so, a validation is dispatched to the reorder buffer. Otherwise, a new vectorized instance of the instruction is created. This stalls do not impact much performance since the number of vectorized instructions with one scalar operand that is not ready at decode is low.

Note that the cost of a context switch is not increased since only the scalar state of the processor needs to be saved. The additional structures for vectorization are just invalidated on a context switch. When the process restarts again the vectorization of the code starts from scratch at the point where the process was interrupted.

3.5.2 Vector registers

Vector register is one of the most critical resources in the processor because they determine the number of scalar instructions that can be vectorized. Vector registers can be regarded as a set of scalar registers grouped with the same name.

A vector register is assigned to an instruction in the decode stage when this instruction is vectorized. If no free vector register is available, the instruction is not vectorized, and continues executing in a scalar mode.

To manage the allocation/deallocation of vector registers, each register contains a global tag and each element includes a set of flags of bits as shown in Figure 3.11.

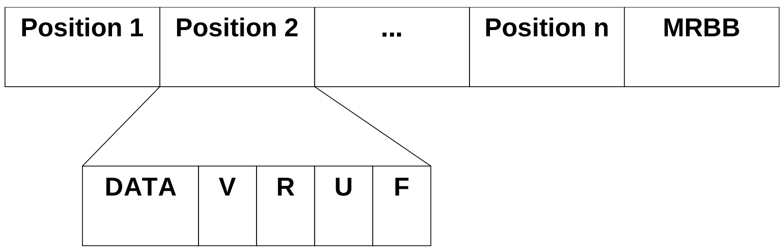

Figure 3.11: Modified vector register

The V (Valid) flag indicates whether the element holds committed data. This bit is set to 1 when the validation associated to the corresponding scalar instruction commits.

The R (Ready) flag indicates whether the element has been computed. Depending on the kind of instruction the data will be ready when is brought from memory or computed by a vector functional unit.

When a validation of an element has been dispatched but not committed yet, the U (Used) flag is set to 1. This prevents the freeing of the physical register until the validation is committed (details on the conditions to free a vector register are described below).

The F (Free) flag indicates whether the element is not longer needed. This flag is set to 1 when the next scalar instruction having the same logical destination register or its corresponding validation commits.

A vector register will be release when all its computed elements have been freed (i.e. are not needed any more). Besides, a register is also released if all validated elements are freed and no more elements need to be produced. In order to estimate when no more elements will be computed, we assume that this will happen when the current loop is terminated. For this purpose, the processor includes a register that is referred to as GMRBB (Global Most Recent Backward Branch) that holds the PC of the last backward branch that has been committed [TG98]. Each vector register stores in the MRBB (Most Recent Backward Branch) tag the PC of the most recently committed backward branch when the vector register was allocated. This backward branch, usually, coincides with the last branch of a loop, associating a vector register to an instruction during some iterations.

A vector register is freed when one of the following two conditions holds:

1. All vector elements have the flags R and F set to 1. This means that all elements have been computed and freed by scalar instructions.

2. Every element with the flags V set, has the flag F set, and all the elements have the flag R set and flag U cleared, and the content of the tag MRBB is different of the register GMRBB. This means that all the validated elements have been freed. Furthermore, all elements have been computed and no element is in use by a validation instruction. It is very likely that the loop where the vector operation that allocated the register was, has been terminated.

3.5.3 Vector data path

Vector instructions wait in the vector instruction queues until their operands are ready and a vector functional unit is available (i.e. instruction are issued out-of-order). Vector functional units are pipelined and hence can begin the computation of a new vector element every cycle. Every time an element is calculated, the vector functional unit sets to 1 the flag R associated to that position, allowing others functional units to use it.

Vector functional units can compute operations having one vector operand and one scalar operand. To do this, the functional units must have access to the scalar register file and the vector register file. In the case of the scalar register, the element is read just once.

Note that some vector instruction can be executed having a different initial offset for their source vector operands. This can happen, for example, when two load instructions begin vectorization in different iterations and their destination vector registers are source operands of an arithmetic instruction. To deal with these cases, vector functional units compare these offsets to obtain the greatest. The difference between this offset and the vector register length determines the number of elements to compute. Fortunately, the percentage of the vector instructions whose source operands' offsets are different from 0 is very low, about 4,5%.

3.5.4 Branch mispredictions and control-flow independence

When a branch misprediction is detected, a superscalar processor recovers the state of the machine by restoring the register map table and squashing the instructions after the branch.

In the proposed microarchitecture, the scalar core works in the same way as a conventional processor, i.e. a precise state is recovered, but vector resources are not modified: vector registers are not freed, and no vector functional unit aborts the execution because they can be computing data that can be used in the future. The objective is to exploit control-flow independence. When the new path enters again in the scalar pipeline, the source operands of each instruction will be checked again, and if it happens that the vector operands are still valid, the instruction does not need to be executed. Our studies show that the percentage of instructions in the 100 instructions (100 is a size arbitrarily chosen) that follow a mispredicted branch that do not need to be executed since they were executed in vector mode and continue to have the same source operands after the misprediction, are 20% for SpecInt2K and 6% for SpecFP2K. SpecFP percentage is lower since branches are easily predictable so, control-flow independence is hardly present.

Note that when a scalar instruction in a wrongly predicted speculative path is vectorized, the vector register may remain allocated until the end of the loop to which the instruction belongs. This wastes vector registers but fortunately only happens for less than 1% of the vector instructions in our benchmarks.

3.5.5 Memory disambiguation

To ensure memory disambiguation, stores are critical instructions because these instructions make changes in memory that the processor cannot recover. For this reason, a store instruction modifies the memory hierarchy only when it commits.

A vectorized load copies memory values into a register. However, there may be intervening stores before the scalar load operation that would have made the access in a non-vectorized implementation. Thus, stores must check the data in vector registers to maintain coherence.

For this purpose, every vector register has two fields, the first and the last address of the corresponding memory elements (used only if the associated vector instruction is a load) to indicate the range of memory positions accessed. Stores check whether the addresses that are going to modify are inside the range of addresses of any vector register. If so, the VRMT entry associated to this vector register is invalidated. Then, when the corresponding scalar instruction is decoded, it will not find its PC in the VRMT and another vector instance of this instruction will be created. Besides, all the instructions following the store are squashed.

Fortunately, the percentage of the stores whose memory address is inside the range of addresses of any vector register is low (4,5% for SpecInt and 2,5% for SpecFP).

Due to the complexity of the logic associated to store instructions, only two store instructions can commit in the same cycle.

3.5.6 Wide bus

To exploit spatial locality a wide bus of 4 words has been implemented. This bus is able to bring a whole cache line every time the cache is accessed. In parallel, the range of addresses held in this cache line are compared with the addresses of pending loads, and all loads that access to the same line are served from the single access (in our approach, only 4 pending loads can be served at the same cycle). This organization has been previously proposed elsewhere [LLVA98] [RTDA97] [WD94a] [WO01].

Wide buses are especially attractive in the presence of vectorized loads, since multiple elements can be retrieved by a single access if the stride is small, as it is in most cases.

3.6 SECOND APPROACH: DYNAMIC VECTORIZATION WITH SCALAR RESOURCES

3.6.1 Motivation

Up to this point, the basic dynamic vectorization mechanism has been presented. Following mechanisms maintain the philosophy of the technique but are modified to simplify the processors's design.

One may think that with this mechanism the processor has been excessively complicated. But it is not. The new structures added present a regular layout and the access time fits comfortably in the cycle time of the reference processor. In fact, the critical point of this design is located in the baseline, in particular, the crossbar among register files and functional units [PJS97]. Since the vector replicas create values for scalar instructions, a movement between banks is needed to supply that data. So, a communication between both the scalar and vector register file and the scalar and vector functional units is implemented through a crossbar.

For our simulations we suppose that this crossbar is easily traversable in a negligible amount of time. But a deeper study of this structure reveals that this supposition is wrong. In multigigahertz scenarios where some stages of the pipeline are exclusively used to send and receive signals, a crossbar with these characteristics cannot be implemented efficiently.

For this reason, we modify the original design to remove this crossbar. To accomplish this, we keep the same philosophy (speculative precomputation of data through vector patterns) but now, instead of creating vector instructions, the mechanism will replicate instructions. This means that, whenever an instruction is vectorized, scalar copies of this instruction will be created. Furthermore, instead of allocating one vector register per vectorizable instruction, a set of scalar registers will be used. So, from this point onwards, *vectorization* and *replication* will be used for the same concept: creation of scalar copies of one vectorizable instruction.

Note that this implementation can overload the issue logic. Since many instructions are created per vectorizable instruction, the issue queue can fill with speculative instructions delaying the non-speculative ones. But it is more preferable this effect than to impact the cycle time and the complexity of the processor with a hard-to-implement crossbar. Later on we will show that some modifications of the issue queue are needed to prioritize the non-speculative instructions.

Since vector register management is different from set of scalar register management, some structures of the original mechanism must be adapted to track the state of every scalar register allocated for replication.

3.6.2 Overview

Figure 3.12 shows the main difference between vectorizing and replicating an instruction.

The load instruction (`Load R1, a(R0)`) has been as a strided load. In the original mechanism, a vector register is allocated and a vector instruction is speculatively created to precompute data for its counterpart instruction.

In the new implementation of the mechanism, a set of registers is allocated.

Furthermore, as many copies as allocated registers are created to precompute data. Every copy has its own destination registers to hold. In the example in Figure 3.12, four copies of the load instruction are created. Notice that every copy is unique: all them have a different destination register and a different effective address (computed with the stride predictor).

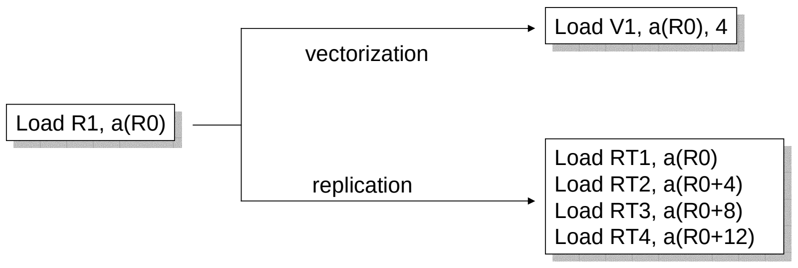

Figure 3.12: *Vectorization* and *Replication*

3.6.3 Adapting the Dynamic Vectorization mechanism

Following subsection describe the structure adaptation to the new version of the mechanism where no vector capabilities are available.

The Scalar Register Set Map Table

Once a strided load is detected, multiple instances of it are speculatively dispatched to the issue queue. Each dynamic instance will read a different memory address, which is computed by adding to the last effective address the stride multiplied by the order rank of the dynamic instance (current address+(stride*n)). Depending on where to store the precomputed data, replicas will use, as destination operand, a different scalar register. When a replicated instruction is refetched, the first speculative instance is validated and if the validation is correct, the instruction is just marked as completed. Following replicas will be validated in the same way. When the last replica is validated, another set of multiple speculative instances of the instruction are dispatched again.

These multiple instances are managed by means of an additional table, the SRSMT table (Scalar Register Set Map Table). This table centralizes both the association of sets of scalar registers and vectorized instructions (provided in the previous design by the *VRMT* table) and the status of every register in the set (equivalent to the bits added to the vector registers). See Figure 3.13 for details on how the VRMT and the vector registers fields have been mapped in the SRSMT.

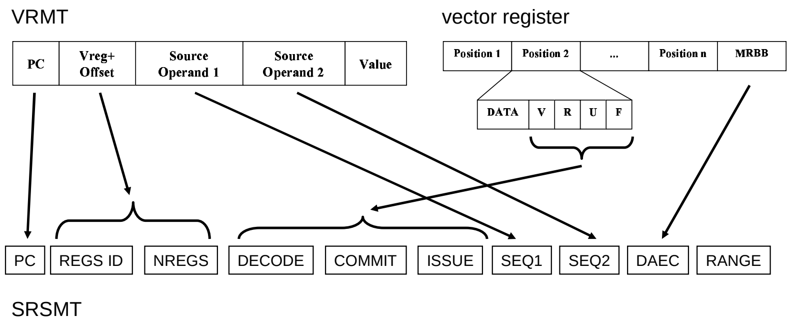

Figure 3.13: Scalar Register Set Map Table details

The *Set of registers* field holds the identifiers of the physical registers that will be used as destination registers for the replica instructions and the field *Nregs* stores the number of registers that have been allocated (which equals to the number of replicas). Note that, in the case that not enough free registers are available for the desired number of replicas, a lower number of replicas or none at all are created.

The next two fields, the *decode* and the *commit* fields, track the state of the set of replicas and are equivalent to the V, R, F and U bits of every position of the vector register. The *decode* indicates which is the next replica to be validated. This field is incremented when a new dynamic instance of the instruction enters into the decode stage. The *commit* field indicates the last replica that has been committed. This field is incremented when a dynamic instance of an instruction is successfully validated and commits. When a recovery action is needed, (e.g. in case of a branch misprediction) the state of the table can be easily recovered by copying the content of the commit field into the decode field for every entry of the table. When the *decode* and *commit* fields are equal, the entry in the SRSMT is deallocated. Note that this does not imply the deallocation of physical registers.

The *issue* field holds the number of replicas that are being executed (i.e., have been issued but their execution has not finished). The purpose of this field is later discussed in this section.

The next two fields, *seq1* and *seq2*, identify the instructions that compute the source operands if they have been vectorized, or the value of the scalar operand otherwise (not all source operands must be vector operands). The identifier of a vectorized instruction is its PC (also called Seq or Sequence).

The rename table, as in the first design, is extended to include for each logical register whether the latest instruction that writes to it has been vectorized and if this is the case, it contains the PC of that instruction.

When an instruction is vectorized, an entry is allocated in the SRSMT table. A free entry is chosen but if none is available, an entry is tried to be deallocated. An entry can be deallocated when the fields *decode* and *commit* have the same value, and the field *issue* is set to 0. If several entries are candidates to be deallocated (depending on the indexing function) the LRU is chosen. When an entry is deallocated, the resources allocated by it are released. If no entry can be deallocated, the instruction is not vectorized.

The identifiers of the source operands for a newly vectorized instruction are obtained from the rename table. If an operand is scalar, its value is read from the register file. If the value is not ready, the instruction and following ones are stalled.

Validation of speculative precomputed data, remains unchanged except for the accesses to the new adapted structures. Every time an instruction is fetched, its PC is looked up in the SRSMT and if found, it means that the instruction has been vectorized and must be validated. Validation of arithmetic instructions consists in checking whether the producer's identifiers currently found in the rename table for its source operands are equal to those of the SRSMT validates. For a load, the stride must keep on being the same. If these checks are successful, the instruction is not executed and it is sent to the commit stage where it will finalize its validation. In the commit stage it will wait until the fields *decode* and *commit* of its source operands in the SRSMT table are equal. When it commits, the *commit* field of its entry in the SRSMT is increased. Notice that every dynamic instance of a replicated instruction sets the bit V/S to 1 and the field *sequence* is set to the "sequence" of the instruction in the rename map table.

If the speculation is not correct, the corresponding entry in the SRSMT and the scalar registers associated to the replicas of this instruction are deallocated, and new replicas are created with the new operands.

Issue logic

In the previous design two issue queues were available to differentiate the scalar data path from its vector homologous sharing the same issue logic. With the new disposition of the mechanism, the vector data path disappears since speculative (instructions created by the mechanism) and non-speculative instructions are executed in the same functional units holding their results in the same register file.

But this new layout has one main problem. If the speculative work is created carelessly the execution stages of the processor can overload. This makes that non-speculative instructions delay since neither entries in the issue queues nor scalar registers are available for these instructions. To overcome this problem, we follow a conservative approach. First, replicas are not created if no entry in the issue queue is available. Furthermore speculative replicated instructions are given less priority than the rest. This prevents from delaying the non-speculative instructions present in the issue queues. The register availability will be discussed later in section 3.6.3.

On the other hand, in case of a branch misprediction replicas are not squashed since, as we will show in Chapter 4 most of them are control independent.

Branch mispredictions

When a branch misprediction is detected, the SRSMT table for the mechanisms can be in an inconsistent state (the decode and commit fields are not equal, meaning that there are values that are going to be validated). To solve this easily, on branch mispredictions, the *commit* field of the SRSMT is copied in the *decode* field squashing the validation instructions after the mispredicted branch. Furthermore, the field *DAEC* is increased, and when equal to MAX_DAEC, the mechanism deallocates the resources (scalar registers) and the entry in the SRSMT of the instruction. This field *DAEC* is used to detect dead associations (i.e. replicated instructions that will not reenter the pipeline) among replicated instructions and allocated set of registers. High values of MAX_DAEC avoid the early release of sets of registers, but force the mechanism to use more registers to allocate data for replicas because registers and replicated instructions are associated for too long. Low values of MAX_DAEC are useful for early detection of dead associations but provoke that most of the speculative data will be discarded before it is used increasing the number of speculative instructions. Our studies show that 2 is the optimal value for MAX_DAEC.

Memory disambiguation

As the objective of replicas is to precompute values for instructions beyond the instruction window, scalar registers of strided load replicas hold memory values that can be modified by stores, provoking memory inconsistency. A simple approach to solve this problem is to deallocate entries (and associated registers) of the SRSMT of the replicated loads, whose field RANGE includes the memory address of a committing store. Even if the SRSMT table is small, this may cause the deallocation of several entries of the SRSMT. To simplify the hardware, up to 2 stores can be committed per cycle, and the commit latency for stores has been increased by 1 cycle.

Wide bus

Instruction replication exploits the spatial locality of data since most of the strided loads have a unit stride. For this reason, buses of the data cache are assumed to be wide. A wide bus can read a whole cache line for every access and multiple outstanding loads can use these data. For complexity considerations (reducing the number of ports to the register file) up to four outstanding loads can be served during a wide access.

Replica register file

As we will show later in the Performance Evaluation section, Dynamic Vectorization is a register-hungry mechanism. In the previous design, speculative vector instructions store their data in a separated register file. This made that normal instructions were not be prejudiced by the speculative work due to a lack of registers. But in the new design, replicas and normal instructions share a scalar register file.

Even if we have adopted the conservative approach, an instructions is not replicated if registers are not available, the mechanism can degrade the performance of the processor since non-speculative instructions cannot allocate registers, because they are being used by replicas. Early-release, late-allocate techniques can be applied to the mechanism to improve the register allocation algorithm of the processor. But this would complicate the design.

We adopt a simpler approach: a two-level hierarchical register file [CGVT00] [BDA01b] has been considered as shown in Figure 3.14. The low level (the fastest one) has been implemented similar to a monolithic register file. The upper level (the slowest one) is used to store the precomputed values of replicas and it is implemented as a slow and cheap RAM memory (for simplification, we will refer as a register every position in this memory). With this organization, the mechanism exploits the non-criticality of replicas since values created by replicas will not be used until some time after they are created. Furthermore, since replicas enhance the temporal locality of data, the management of the high level of the register file can be easily performed by the mechanism: once replica *I* has been accessed, the probability of accessing replica *I+1* is high. Therefore, a prefetch access for replica *I+1* can begin.

When a validation instruction enters the decode stage, a register is allocated in the low level and a copy instruction is inserted in the issue queue. When executed, this instruction performs a movement of the value from the upper level to the

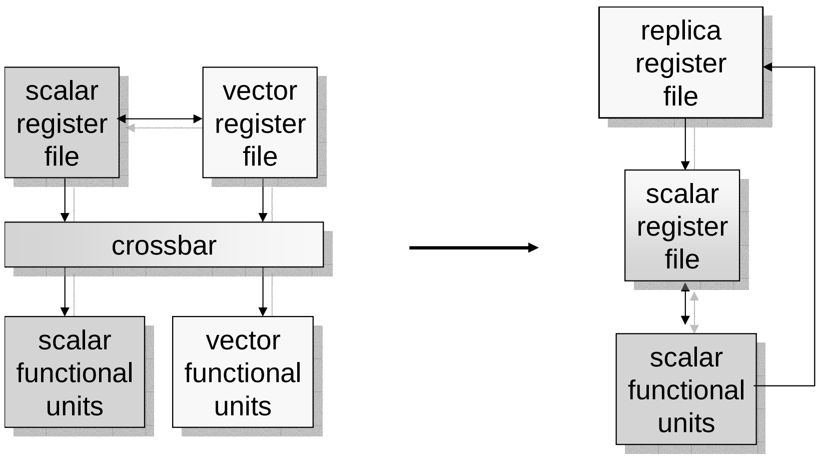

Figure 3.14: Hierarchical register file

lower one. Dependent instructions will use the register of the lower level as source operand. Validation instructions pass from decode to the commit stage to validate the speculative data without waiting for the execution of the copy instruction. Both registers, (from the lower and the upper levels) are deallocated when the following instruction with the same logical destination register commits. For our simulations, the copy instructions have a latency of 2 cycles, and up to 2 movements between banks can be performed per cycle.

This layout alleviates the pressure in the scalar register file and allows the elimination of the cycle-impacting crossbar among registers and functional units.

3.7 PERFORMANCE EVALUATION

This section only evaluates the last design of the mechanism. As said before, the main motivation to build the scalar design is the complexity reduction in the execution engine of the processor. This fact makes that the scalar design is used in following refinements. Performance is nearly the same for both designs although the scalar version improves about 3% the vector design due to the finest grain management of the registers. Apart from that, the other sources of performance improvement remains the same.

The parameters for the Dynamic Vectorization mechanism are shown in Table

Parameter	Value
Stride predictor	4-way set associative with 512 sets
SRSMT	4-way set associative with 64 sets
Hierarchical register file	768 positions, 2 read/write ports, 2 cycle access time

Table 3.2: Configuration parameters for the Dynamic Vectorization mechanism

3.2.

3.7.1 Dynamic Vectorization performance

Figure 3.15 shows the performance achieved by a wide range of memory configurations with a superscalar processor (xscalar), a superscalar processor with a wide bus (x-WB) and a superscalar processor with a wide bus and dynamic vectorization (x-WB-DV) for a different number (x) of L1 data cache ports.

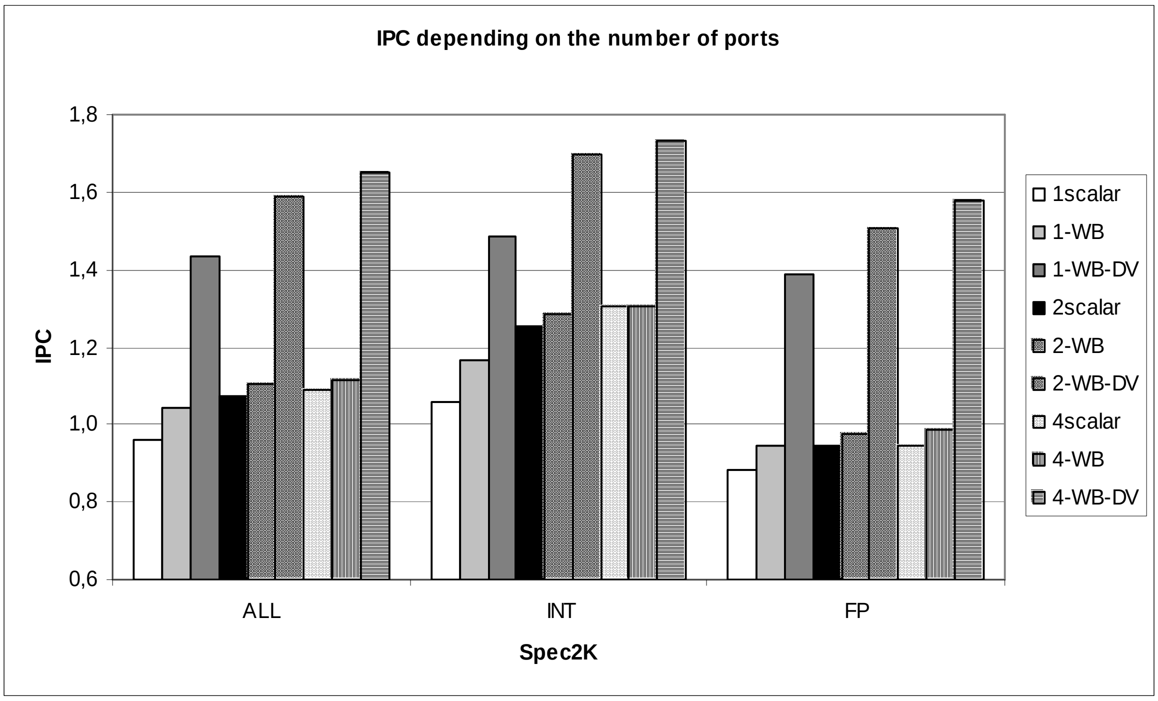

Figure 3.15: Dynamic vectorization performance

As shown in Figure 3.15, in most cases the configurations with wide buses increase clearly the performance of the configurations with scalar buses. The main reason is the bottlenecks due to the memory system in configurations like a superscalar processor with 1 scalar bus. For this configuration the average IPC increased from 0,95 to 1,04, on average (from 1,05 to 1,16 for SpecINT2K and from 0,88 to 0,94 for SpecFP2K) when a wide bus substitutes the scalar bus. The benefits for the configurations with 2 or 4 scalar buses are smaller since they already have a significant memory bandwidth.

Figure 3.15 also shows that dynamic vectorization boosts the performance considerably for every configuration, raising from 0,95 to 1,43 (39%) the IPC on configurations with one wide bus. When more memory bandwidth is available, the obtained speedup ranges from 49% to 51%. This performance improvement comes basically, from three sources.

Management of the memory system

The first one is the **better management of the memory system**. The wide bus to the L1 data cache enhances the access to this structure since several pending loads that coincide in the same cache line can be served at once. As shown in Figure 3.16, when a wide bus to the L1 data cache is used, nearly 10M of accesses, on average, are reduced.

The explosion of DLP exposed by replicas conjugated with the wide bus, makes more efficient the access to the L1 data cache. Dynamic vectorization reduces even more the number of accesses, about 4M (Figure 3.16), related to the wide bus. This means that nearly 50% of the scalar loads present either spatial or temporal locality that are not exploited by current designs of data caches. The distributions of loads served per access are shown in Figure 3.17 for configurations with 1 wide bus.

In addition to this management of the memory system, dynamic vectorization offers prefetch for long latency loads. Since the stride predictor provides the next effective addresses for strided loads, data can be brought in advance to the lower

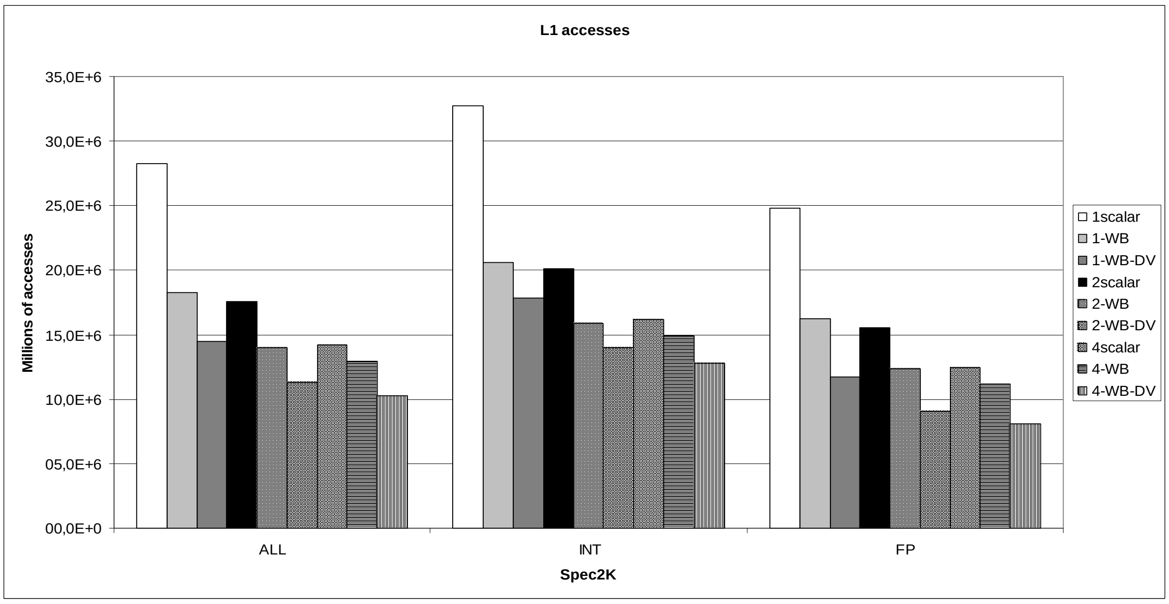

Figure 3.16: L1 access reduction of Dynamic Vectorization

levels of the memory hierarchy overlapping these accesses with the execution of the normal stream of instructions. On the other side, the ability of preexecuting non-strided loads allows successful prefetching in code schemes, i.e. linked lists, where current prefetchers fail. Our studies show that this prefetch effect is, on average, half of the total performance improvement achieved with the mechanism. The topics of prefetch and the memory gap will be later discussed in Chapter 5.

Reuse of precomputed data

As previously said, replicas precompute data for future instances of instructions currently present in the mainstream. This data precomputation effectively reduces the latency of instructions entering the pipeline. An instruction with precomputed data becomes a validation instruction, therefore it does not execute, but passes from the decode to the commit stage of the pipeline. Furthermore, functional units occupancy is reduced since the ratio of fetched instructions that must be executed goes down due to validation instructions. So, the second source of performance improvement of the mechanism is the **successful precomputation of speculative data**.

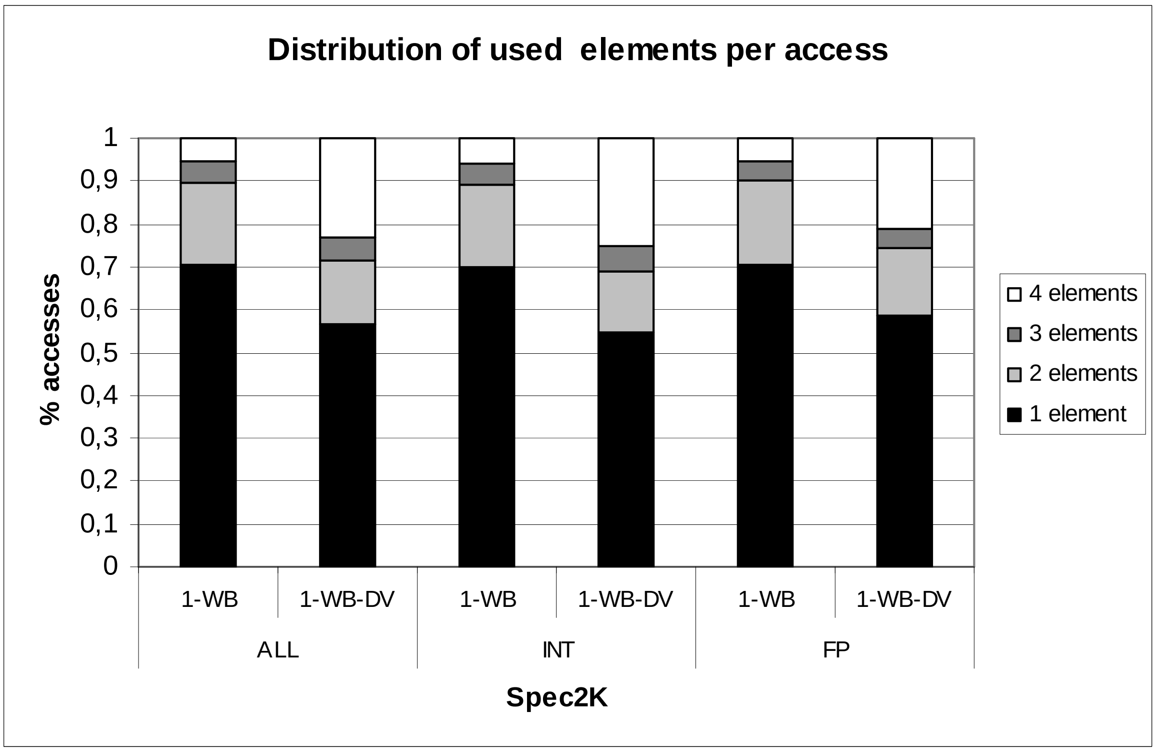

Figure 3.17: Distribution of loads served per cycle

There is an important aspect that must have had into account: replicas use resources, such as functional units or scalar registers, to execute. Even if replicas have lower priority than normal instructions, an excess of speculative work can overload the execution engine of the processor degrading the performance. As we will see later in Chapters 4 and 6, it is necessary to survey the amount of speculative work allowable by the processor.

Figure 3.18 shows the distribution of instructions executed by the processor. In this Figure, we differentiate four kinds of instructions: committed instructions that have not precomputed data (no reuse), successful validation instructions (reuse), speculative instructions executed under incorrect path predictions(specBP) and mis-speculated replicas (specDV). It is interesting see from this figure, that most of the vectorizable instructions have been effectively vectorized, nearly 84,5%. The rest of vectorizable instructions have not been replicated due to a lack of resources, such as registers or entries in the SRSMT table.

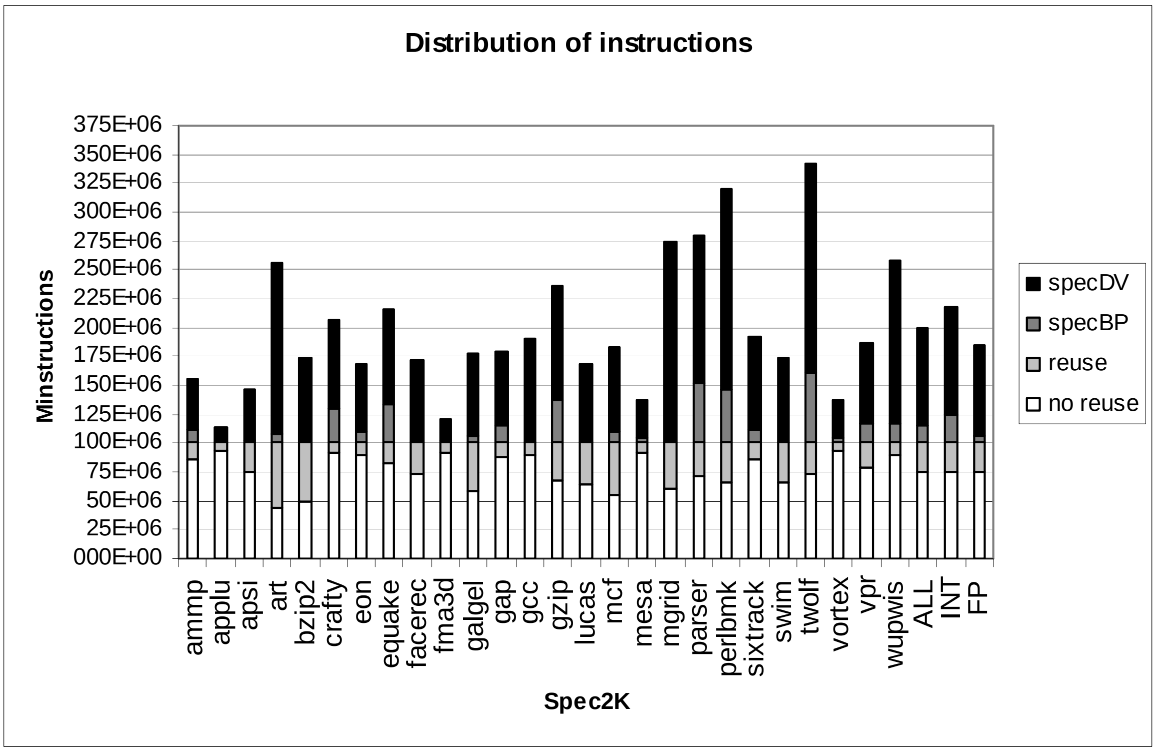

Figure 3.18: Distribution of instructions

There are two important aspects in Figure 3.18. First, reuse ranges from 6,7% (applu) to 56% (ammp), and, on average, is 24,5%. Second, the amount of speculative replicas executed to obtain these percentages of reuse is quite high (84M of instructions on average).

Regarding reuse, these numbers show that, in fact, there exists a considerable amount of underlaying DLP only detectable at runtime and Dynamic Vectorization is able to expose it. Even for SpecINT, some benchmarks, like bzip2, present a clearly code regularity inappropriate in these applications.

On the other hand, SpecFP benchmarks present low ratios of reuse. Basically, this is due to the fact that compiler optimizations, such as loop unrolling, reduce the chances of stride pattern detection. The basis of loop unrolling is to replicate the loop body to execute, in parallel, several iterations. This optimization makes the number of strided load increase, but the number of times that every load is executed becomes lower reducing the chances to detect the stride pattern. In addition, since the number of instructions in the loop body is augmented, more registers are needed for vectorization.

Even if good reuse numbers are obtained, replication must be done carefully. Figure 3.18 shows that, on average, 84M of extra instructions are needed to obtain 24,5% of successful reuse. As previously said in the *Replica register file* section, Dynamic Vectorization is register-hungry. The more replicas are created, the more registers in the upper level of the hierarchical register file are needed and the more power and energy are employed in the processor for misspeculated work. Since the register file is a critical resource in the processor, due to power, energy and area constrains, we cannot allow a larger register file for replicas. On the other hand, register allocation for mispredicted data prevents the mechanism to increase the performance improvement since replicated work are limited to the actual available resources.

As we will discuss later in Chapter 4, a selection mechanism to decide whether an instruction can be vectorized must be provided to reduce the number of wrongly replicated instructions.

Virtual enlargement of the instruction window

The third important source of performance improvement of the Dynamic Vectorization mechanism is the virtual enlargement of the instruction window.

Replicas' work consists in creating data for future instances of instructions. These instances, usually, have not entered in the pipeline, what means that the mechanism is able to create values for instructions that will be fetched sooner or later.

Chapter 5 will discuss in more detail this source of performance improvement. As a brief, our numbers show that the Dynamic Vectorization mechanism is able to create instruction windows of more than 1200 instructions with reorder buffers of only 256 entries. This is basically due to the fact that replicas don't need any entry in the reorder buffer to execute thanks to their speculative nature.

Control-flow independence reuse

As described in section 3.6.3, the recovery mechanism does not squash vector instructions after branch mispredictions. In fact, speculative work is only squashed is misspeculations are detected by validation instructions. Due to this, the control-flow independence instructions can reuse the precomputed data. This source of performance improvement will be explained deeper in Chapter 4 since it motivates the next optimization of the mechanism.

3.8 SUMMARY

In this chapter we have presented the basis of the Dynamic Vectorization mechanism. Two versions of the mechanism are described. The first implementation relies on the existence of vector resources to execute the speculative created data. Speculative vector instructions are created as soon as a vector pattern, represented as a strided load, is detected at runtime. These speculative vector instructions prefetch and precompute data for their scalar counterparts. After that, instructions with precomputed data become validation instructions that check if that precomputation is correctly performed.

An evolved version of the mechanism is derived from the vectorial design due to the complexity of the execution engine that deals with both scalar and vector instructions. For this design, the vector resources have been removed from the processor.

A detailed evaluation of the mechanism is provided. Four main sources of performance improvement have been detected: efficient management of the memory system, successful reuse of precomputed data, virtual enlargement of the instruction window and control-flow independence reuse of data. Some of these sources (virtual enlargement of the instruction window and control-flow independence reuse) will be explained in following chapters. Furthermore, we have shown that regular patterns are presented in irregular codes and are easily detectable at runtime by the Dynamic

Vectorization mechanism.

Finally, we have expose the problem of the Dynamic Vectorization mechanism: an excess of speculative work can overload the execution engine of the processor, degrading severely the performance. Following chapters will show how to include selection mechanisms to reduce the number of mispredicted replicas.

Chapter 4

Control-Flow Independence Reuse

4.1 INTRODUCTION

Current processors' potential to exploit instruction level parallelism depends on their ability to build a large instruction window. Branch instructions are the main problem to build such large instruction windows for non-numeric applications. Every time a branch prediction is wrong, the pipeline is flushed, and the instruction window is built again through the correct path. However, control independent instructions (from now onwards CI instructions), i.e., instructions that are encountered in every branch path computing the same values, could theoretically remain in the instruction window and their re-execution could be avoided.

In this chapter, we present a specialization of the Dynamic Vectorization mechanism to detect and replicate CI instructions. From the point of view of DV, CI instructions are the best instructions to vectorize since they present a high ratio of correctly precomputed data. This is due to the fact that CI instructions present the same source operands, thus computing the same result, whatever the path is taken by branches. This will result in a better resource management at a relative low performance degradation (3%).

On the other hand, Dynamic Vectorization proves to be a very effective mechanism to improve performance when focused on CI instructions. Since the probability of a CI instruction will reenter later in the pipeline is high, replication of these instructions will effectively enlarge the instruction window.

4.2 MOTIVATION

Figure 4.19 is the main motivation of the current Chapter. This Figure shows the percentage of successful validation instructions that are control independent between two consecutive mispredicted branches.

It is easy to see from Figure 4.19 that for many SpecINT2K, the ratio of validation instructions that are CI is quite high (nearly 84% on average).

This chapter presents a selection mechanism that will only replicate these in-

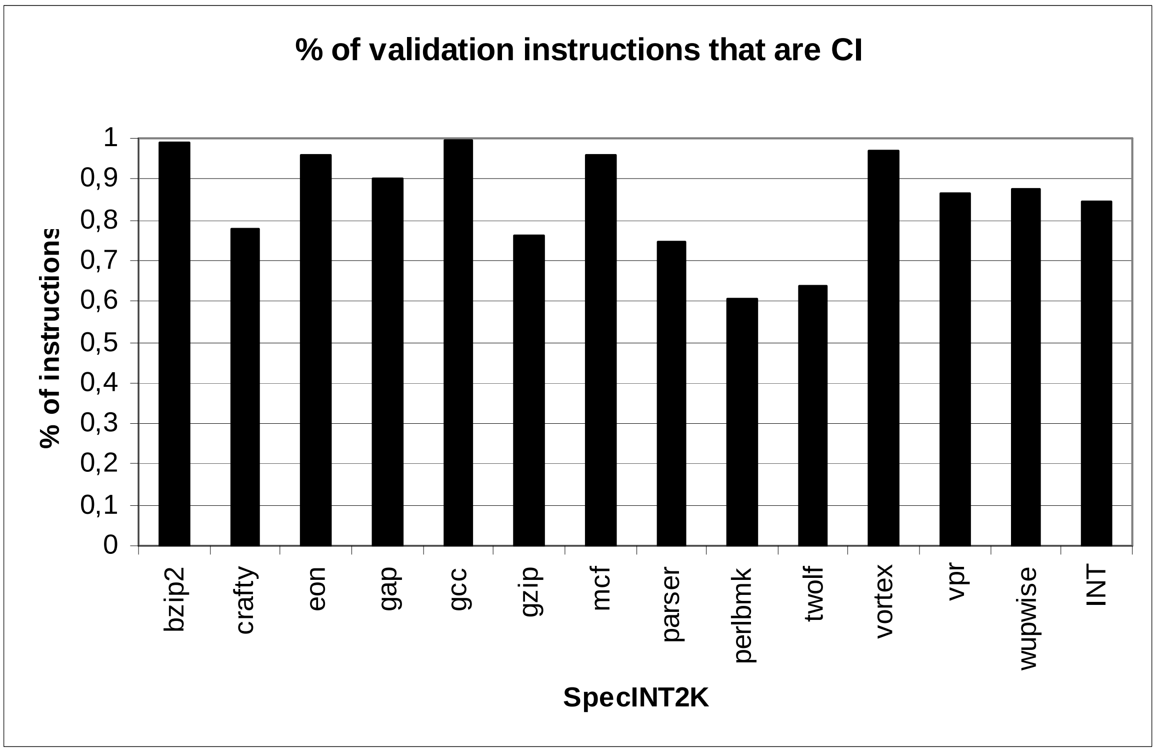

Figure 4.19: Percentage of validation instructions that are control independent

structions to reduce the speculative work to increase the effectiveness of the basic dynamic vectorization mechanism.

4.3 RELATED WORK

Chou et al. [CFS99] present a mechanism for exploiting control independence that is based on a structure called DCI that stores copies of decoded instructions. After a recovery from a branch misprediction, new fetched instructions are looked up in the DCI to locate the beginning of a control independent region. Furthermore, an out-of-order fetch mechanism is presented to start fetching from the control independent region after a branch prediction is performed.

Rotenberg et al. present a mechanism to exploit control flow independence in superscalar [RJS99] and trace processors [RS99]. Their approach is based on identifying re-convergent points dynamically, and a hardware organization of the instruction window that allows the processor to insert the instructions after a branch misprediction in between instructions previously dispatched, i.e., after the mispredicted branch and before the control independent point.

Sodani et al. [SS97] present a mechanism for global reuse based on keeping history of previous executions of instructions. When an instruction is decoded a structure, called Reuse Buffer, is checked to see if there is a previous execution of this instruction and if the result for that execution can be reused. In that paper, several implementations of the Reuse Buffer that differ in the way the reuse test is performed are proposed and evaluated.

Cher et al. present Skipper [CV01], a mechanism to overlap the latency of hard-to-predict branches with the execution of control-flow independent instructions following the re-convergent point of those branches. To achieve this, when a repeatedly mispredicted branch is detected, the fetch is redirected to the re-convergent point, creating a gap in the reorder buffer (large enough to hold the skipped instructions), and the processor proceeds to execute the instructions after the re-convergent point. When the branch is resolved, the skipped instructions are executed. Dependences among skipped instructions and the instructions after the re-convergent point are checked to ensure the correctness of the execution, performing recovery actions when needed.

4.4 THE APPROACH

4.4.1 Control flow independent instructions

An instruction is control-flow independent with respect to a given branch instruction if its result is the same regardless of the branch outcome. Control-flow independent instructions are common in hammock control flow structures resulting from if-then-else constructs. An example is shown in Figure 4.20.

The code in Figure 4.20 counts how many elements of vector a are equal to zero (stored in register R3) and how many are not (stored in register R2). Furthermore,

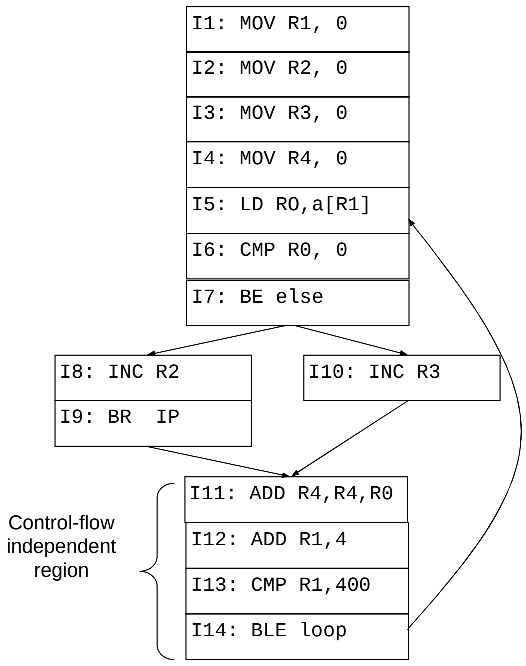

Figure 4.20: Sample code with a hammock

the code accumulates the sum of all elements of vector a in register R4. The branch at instruction I_7 may be hard to predict (e.g., the data of vector a does not follow any regular pattern). However, instructions $I_1 1$-$I_1 4$ are executed and produce the same results regardless of the branch outcome. The first sequential instruction that is common to both taken and not taken paths of a branch will be referred to as the re-convergent point. In Figure 4.20, instruction $I_1 1$ is the re-convergent point of branch I_7. Control-flow independent instructions can be located starting from the re-convergent point onwards.

4.4.2 Overview of the mechanism

The proposed mechanism works in four steps. The first two steps select the control-independent instructions when a branch misprediction is detected. The last two steps, effectively vectorize the selected instructions. The selection part and the vectorization part work separately and are communicated through just one bit in the stride predictor. These steps are explained now following the example in Figure

4.20.

- First step: when a hard-to-predict conditional branch is detected (see details later in Section 2.3.1) and mispredicted, the mechanism tries to find the re-convergent point of that branch. Supposing that I_7 in Figure 4.20 is the mispredicted branch, the first step has to find $I_1$1 as the re-convergent point.

- Second step: identification of the instructions ($I_1$1, $I_1$2 and $I_1$3) after the re-convergent point (included) that are likely to produce the same outcome after the branch misprediction recovery and can be effectively vectorized (only $I_1$1 can be vectorized). For this purpose, every instruction after the re-convergent point is analyzed and it is checked whether its source operands have been changed by an instruction after the branch and before the re-convergent point. If the source operands have not changed, the set of nearest strided loads above the branch on which the instruction depends are selected for speculative vectorization. In the example of Figure 4.20, the first instruction whose operands have not changed after the branch is the re-convergent point itself (instruction $I_1$1). The loads above the branch on which instruction $I_1$1 depends is just instruction I_5.

- Third step: speculative vectorization of the selected instructions next time they are encountered. Vectorization is performed by generating multiple speculative replicas of the vectorized instruction. These speculative instructions are dispatched to the issue queue and executed but not committed until they are verified. Moreover, every time an instruction is fetched, it is checked whether any of its source operands is the outcome of a previously vectorized instruction, and if this is the case, it is also speculatively vectorized. In the example of Figure 4.20, instruction I_5 is the selected strided load that will be vectorized. Instructions I_6 and $I_1$1 will also be vectorized because they are dependent on instruction I_5. Notice that $I_1$1 is a control-flow independent instruction.

- Fourth step: every time an instruction is fetched, it is checked whether it was previously vectorized. If so, it is checked whether the vectorization was correct, and in this case, the instruction is just marked as completed and sent to the commit stage. Otherwise, the instruction is normally executed.

These steps are further detailed below.

4.4.3 First step: hard-to-predict branches and re-convergent point detection

First of all, in order to apply the control independence scheme to branches that are responsible for a significant number of mispredictions [GKMP98] [JRS96], a table that we refer to as the MBS table (Mispredicted Branch Status) is used. This table is indexed by the PC of branches and has a 4-bit saturated up-down counter per entry. The counter is increased by taken branches and decreased by not taken branches, if the direction is the same as the previous outcome. Otherwise, the counter is set to the value that is in the middle of its range. If the value of this counter is the maximum or minimum value, the branch is considered to be highly biased and thus assumed to be easy to predict. Otherwise, the control independence scheme is activated.

The scheme to identify re-convergent points for mispredicted branches is an extension of previous work in [CFS99] and involves basically two hardware structures. The first one is a queue, called NRBQ (Not Retired Branch Queue), where the estimated re-convergent points of the in-flight conditional branches are stored. The second is the CRP (Current Re-convergent Point). Figure 4.21 shows these structures and how they interact. More details follow.

Identification of re-convergent points does not need to be correct. Wrongly estimated re-convergent points will affect the performance of the processor but not the correctness of the execution. Re-convergent points are estimated with the following heuristics:

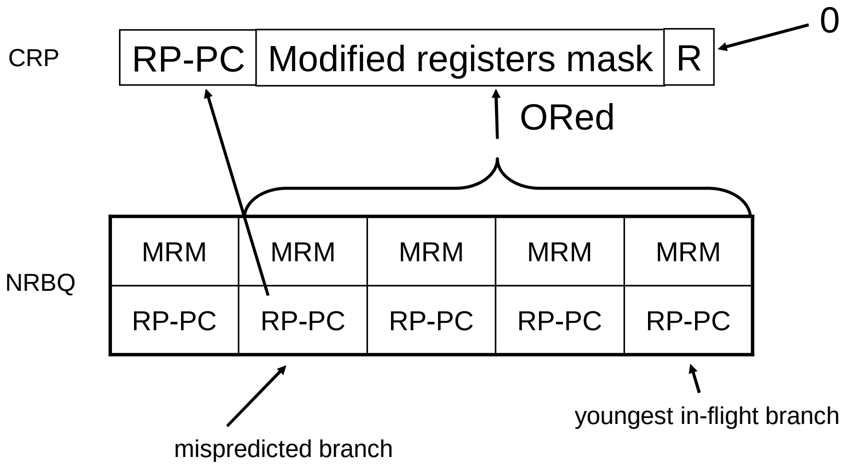

Figure 4.21: NRBQ and CRP interaction

- If the branch is a backward branch [SC00], the re-convergent point is assumed to be the next instruction, in program order, that follows the branch (a backward branch usually corresponds to the closing branch of a loop as shown in Figure 4.22-a).

- If the branch is a forward branch, the instruction situated one location above the target address [CHP97] is fetched and analyzed. If the branch is predicted as taken this instruction is fetched in the next cycle, possibly together with the target instructions and succeeding ones.

- If the branch is predicted as not taken, this instruction is fetched just after the recovery of the misprediction. If this instruction is an unconditional forward branch (which is the common case for an if-then-else structure as shown in Figure 4.22-c), the re-convergent point is assumed to be the address pointed by this branch. Otherwise, the re-convergent point is assumed to be the destination address of the conditional branch (which is the common case for and if-then structure as shown in Figure 4.22-b).

When a branch is executed its prediction is checked. In case of a misprediction, younger instructions are squashed and if the static branch is supposed to cause

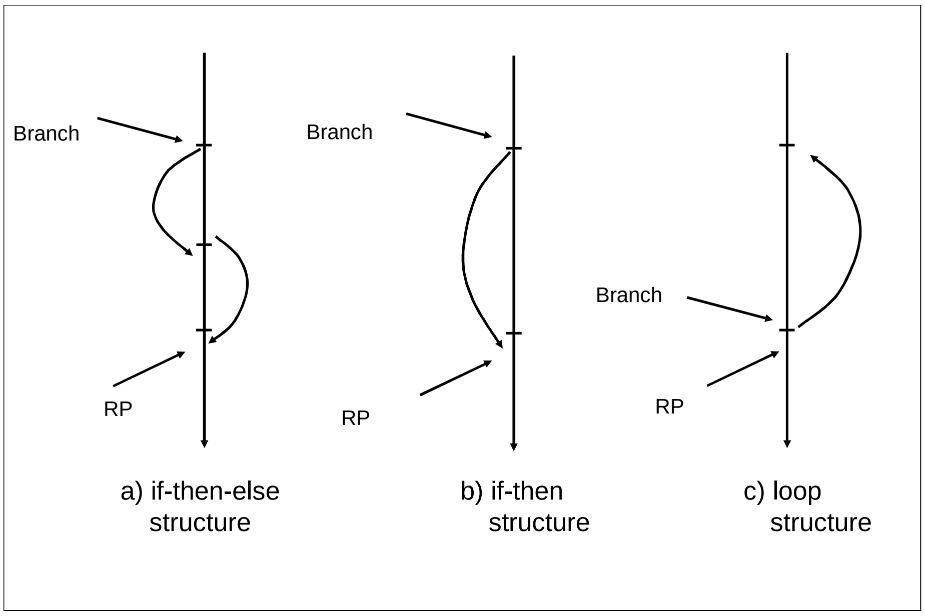

Figure 4.22: Common program constructs

many dynamic branch mispredictions, the information regarding this static branch is introduced into the CRP register (Current Re-convergent Point). This register contains the PC of the re-convergent point and the R (Reached) flag that indicates whether the re-convergent point has been reached.

4.4.4 Second step: control-flow independent instruction detection and filtering

Every time an instruction is fetched, its PC is checked with the PC stored in CRP. If they match, the R flag is set, which indicates that the re-convergent point has been reached. To identify whether an instruction after the re-convergent point does not depend on the instructions between the branch and the re-convergent point, every entry of the NRBQ is extended with a mask of bits. Each bit is associated to a logical register and indicates whether this logical register has been written after this branch and before the next branch. When a branch is found, the corresponding mask is cleared. For each new instruction, the bit corresponding to the destination register is set to one for the entry at the tail of the NRBQ. After a branch misprediction, the information of the mispredicted branch is copied into the CRP as described above. The CRP has also a mask of bits that in this case, indicates whether or not the corresponding logical register has been written since the branch was fetched and before the re-convergent point is reached (in either the wrong or the correct path). In a branch misprediction, the CRP mask is initialized by ORing all the masks in NRBQ starting from the mispredicted branch to the branch at the tail of the queue (i.e. the youngest one). Afterwards, for every new decoded instruction before the re-convergent point is found, the bit corresponding to its destination logical register is set to 1.

An instruction is considered to be control independent if it is fetched after the re-convergent point, and its source operands have their corresponding bits cleared in the mask of the CRP. These instructions will be the target of the speculative vectorization scheme. In addition, all instructions that belong to any dependence chain of the backward slice (i.e. all its predecessors) of a selected instruction are also vectorized if the chain starts with a strided load. For example, in the code of Figure 4.20, I_11 and I_5 are vectorized if I_5 has been observed to follow a strided pattern. But instructions I_12 and I_13, even if they are control independent, will not be considered for our mechanism given that they are not dependent on a strided load. In the worst case, if I_5 is not an strided load, no instruction will be vectorized in the example of Figure 4.20.

To identify these backward chains that start with a strided load, every time a load is fetched the stride predictor is checked and if the load is considered to follow a strided pattern, its PC is associated to the logical destination register. To implement CI selection, every entry of the stride predictor is extended with 1 bit called S. This bit is set when the load corresponding to that entry is selected for vectorization, as described later. Note that this bit is the bridge between the selection logic and the dynamic vectorization mechanism, since only those loads with the S bit set will be vectorized.

To propagate the PC of a strided load down the dependence graph, every entry in the rename map table is extended with a new field called stridedPC., where the PC of the strided load is stored. Arithmetic instructions propagate the stridedPC of their source operands to their destination. In theory, one instruction may have many strided loads as in its backward slice. However, we have experimentally evaluated that SpecINT2000 needs on average 1,7 PCs per entry.

When an instruction after the re-convergent point is selected for vectorization, the strided loads on which it depends are also selected for vectorization, by setting to 1 the flag S in the stride predictor. When the selected load reenters the pipeline, it checks whether the stride keeps on being the same, and in this case, this load is vectorized. Every time an instruction is fetched, if any of its source operands is vectorized, the instruction is also vectorized.

4.4.5 Third step: instruction replication

Once a load with the corresponding bit S set to 1 is fetched, begins the replication of instructions. This third step and the next one, implements the basis scalar Dynamic Vectorization mechanism.

It is important remark that the selection logic implemented in steps first and second and the underlying Dynamic Vectorization mechanism works separately. The only thing that both mechanisms have in common is the bit S of the stride predictor. So, in one side the selection logic analyzes the mainstream looking for chains of control-independent instructions and discriminates the backward strided loads that begin those chains. On the other side, the Dynamic Vectorization mechanism studies which loads present a vector pattern and, from those, replicates only the selected loads.

4.4.6 Fourth step: data validation

As in the previous mechanism, data created speculatively by replicas must be validated. For validation purposes, every entry in the rename map table is extended with the field PPC and the bit R. Stride checking for strided loads and producer's PC checking for arithmetic instructions is performed to validate data. Instructions with correctly speculated data pass to the commit stage for validation. Other previously wrong replicated instructions deallocate resources (scalar registers and the entry of the SRSMT) of replicas and execute normally and perform a recovery action to execute normally the instruction.

4.5 PERFORMANCE EVALUATION

In this Chapter, only SpecINT2K are used to evaluate the mechanism since the branch misprediction ratio of SpecFP2K is negligible. Furthermore, to reduce the memory gap, and to emphasize the goodness of CI reuse, we have changed the latency of main memory from 1000 to 400 cycles for all configurations, to reduce, as much as possible, the prefetch effect of the Dynamic Vectorization mechanism.

Note that for this selection scheme, all the sources of performance improvement of the Dynamic Vectorization mechanism remain the same. This scheme follows two main goals:

- Reduce the misspeculated work. So, more correctly speculated work can be performed, thus improving IPC. Since less instructions are replicated, resource management can be improved. Less register are needed for nearly the same performance.
- Improve control independence reuse taking advantage of the virtual enlargement of the instruction window by the Dynamic Vectorization mechanism.

Control independent is only applicable to SpecINT2K benchmarks since they present a considerable amount of mispredicted branches. SpecFP2K are not considered because nearly 99% of branches are correctly predicted. Table 4.3 shows the percentage of branch mispredictions with the branch predictor stated in Chapter 2.

Benchmark	**Total branches**	**Mispredictions**	**Percentage**
bzip2	12328368	53283	4,32%
crafty	15029217	973640	6,48%
eon	13052907	534598	4,09%
gap	8664311	502559	5,8%
gcc	2683468	10967	0,41%
gzip	13356850	916341	6,68%
mcf	33115933	414022	1,25%
parser	23687815	1555852	6,57%
perlbmk	16985159	2226150	13,11%
twolf	25796746	1674938	6,49%
vortex	15419475	266628	1,73%
vpr	15663865	425314	2,72%
Average	**185775114**	**9554292**	**4,88%**

Table 4.3: Branch statistics for the SpecINT2K (100M instructions per program)

Control independence scope

As shown in Table 4.3 for some benchmarks the percentage of mispredicted branches is high. It would be desirable that our scheme can take advantage of all of them. But this is not possible. There are three constrains related to branches that must be accomplished to replicate instructions:

- A mispredicted branch must be a hard-to-predict branch. This means that the probability that a given branch will be mispredicted must be high. This constrains restricts the scope of the selection mechanism since few of the mispredictions are due to the fact that not enough history is available for some branches, i.e. they are only mispredicted the first time they are found.
- A re-convergent point must be detected for a mispredicted branch. Indirect jumps are out of the scope of the selection scheme since they present several outcomes, complicating the re-convergent point detection.
- At least one control independent instruction must be found.

Figure 4.23 shows that, in fact, not for all mispredicted branches the selection scheme is effective finding control independent instruction for replication.

Every bar in this figure (4.23) shows the percentage of mispredicted branches for which the mechanism does not find any control independent instruction (white portion), selects at least one control independent instruction (gray portion), and selects control independent instructions and successfully reuse precomputed instances (through speculative vectorization) of them (black portion) for every benchmark of SpecINT2000. Control independent instructions are selected for about 70% of the mispredicted branches (black and gray portions). For 49% of the mispredicted branches (black portion), at least one control-independent instruction is correctly vectorized. The remaining 21% of the mispredicted branches (gray portion) where vectorization is not successful are basically due to the fact that they do not depend on strided loads.

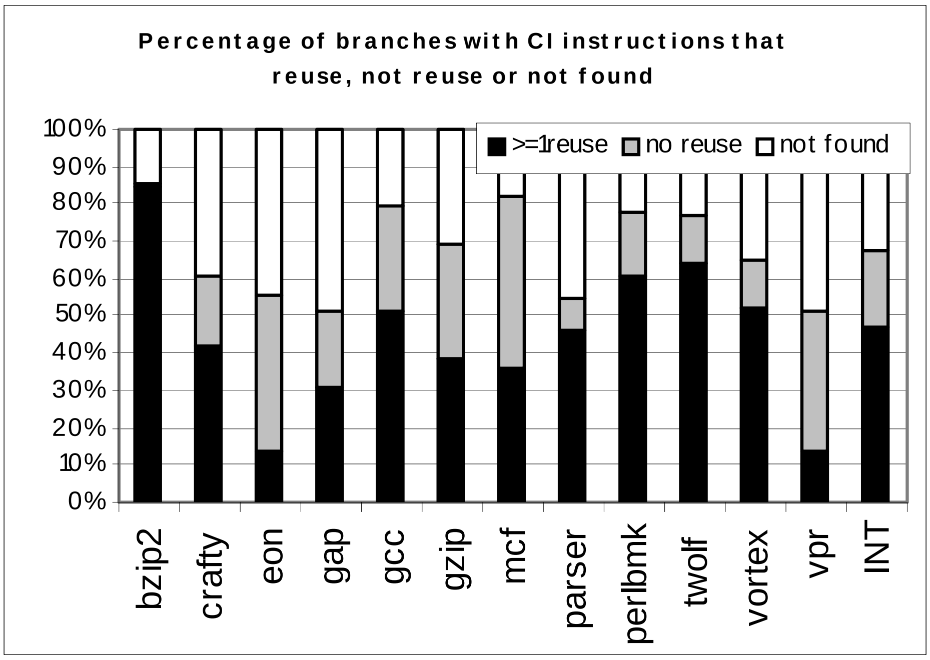

Figure 4.23: Selection scope

Register file pressure

Figure 4.24 shows the IPC obtained with the proposed mechanism (cixp) compared with a superscalar processor (scalxp) and a superscalar processor with wide buses (wbxp), for a varying number of L1 data cache ports (1 port x=1, 2 ports x=2) and a varying number of physical registers (128, 256, 512, 768 and infinite). In this first set of experiments, a single-level register file is considered. Harmonic means are used to average IPC across the whole benchmark suite.

Vectorization creates up to 4 replicas per vectorized instruction. To study the register pressure of the dynamic vectorization mechanism, this first results are obtained with a monolithic register file.

Several conclusions can be drawn from Figure 4.24. First, we can see that wide buses provide a significant benefit for a superscalar processor. This is due to the fact that a wide bus exploits the spatial of memory accesses. As the number of ports increases, this performance benefit decreases since multiple ports can also exploit spatial locality although with a much higher implementation cost.

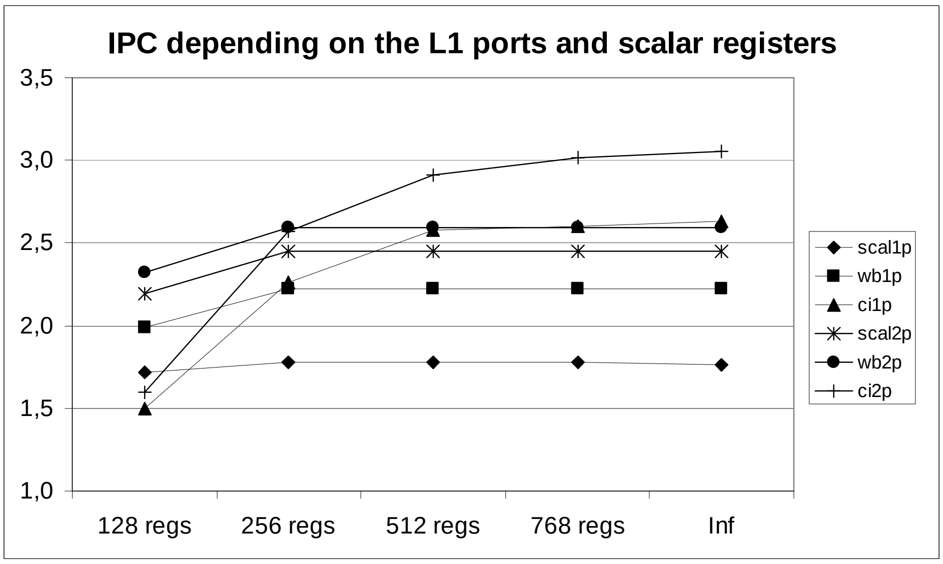

Figure 4.24: Performance of the CI scheme with a monolithic register file

For the baseline configuration with and without wide buses, performance is significantly improved when the number of registers increases from 128 to 256, except for 1 non-wide port, which is mainly limited by memory bandwidth. For configurations with more than 256 registers the reorder buffer has been increased to the size of the number of registers; otherwise, many registers would be useless due to the lack of instructions in-flight. However, this hardly improves performance due to the branch mispredictions and the limited ILP of these applications.

When the mechanism for control independence is included, and enough registers are available, the performance increases more than 17,8% for both configurations (1 port and 2 ports) over the superscalar processor with wide buses. This is basically due to the exploitation of control independence, which allows the processor to execute instructions ahead of the resolution of the branches on which they depend, regardless of the correctness of the branch prediction, and even if they are far away of the current instruction window. The vectorization scheme also favors the exploitation of spatial locality, since the speculative instances of a vectorized load instruction are unit strided most of the times.

The control independence scheme increases the pressure in the register file due to longer lifetimes and wrongly speculated instructions. Because of that, when the number of registers is too low (128 registers for a 256-entry reorder buffer), the control independence scheme results in some performance degradation. For configurations with 256 registers, the control independence mechanism hardly affects performance when compared with the superscalar configuration with wide buses. This is due to the fact that a large number of scalar registers are used to store the values created by the speculative instructions, slowing down the execution of the code that has not been vectorized because less registers are available for it. However, when the number of physical registers keeps on increasing, the superscalar processor performance flattens out whereas the control independence scheme provides significant performance gains.

An important parameter of the proposed scheme is the number of speculative instances that are generated for every vectorized instruction. A higher number of speculative instances implies a higher potential to exploit control independence but also a higher pressure on the register file (i.e., more mispeculations and longer lifetimes). Figure 4.25 shows the effect when this parameter is varied from 1 to 8 instructions. From this experiment we can conclude that either 2 or 4 replicas per vectorized instruction seem the most convenient approach. Generating only 1 speculative version looses a lot of opportunities to exploit control independence. On the other hand, generating 8 replicas only improves performance by very little when the number of registers is very high.

Figure 4.26 shows the number of: a) committed instructions that do not reuse a precomputed value (dark portion), b) committed instructions that reuse a precomputed value (dark gray), c) fetched instructions that do not commit due to a branch misprediction (light gray), and d) speculative instructions generated by the control independence scheme (white) for 2 (left bars per spec) and 4 (right bars) replicas per vectorized instruction.

Figure 4.26 shows that increasing the number of replicas from 2 to 4 increases the percentage of committed instructions that benefit from reuse increases from 12,3% to

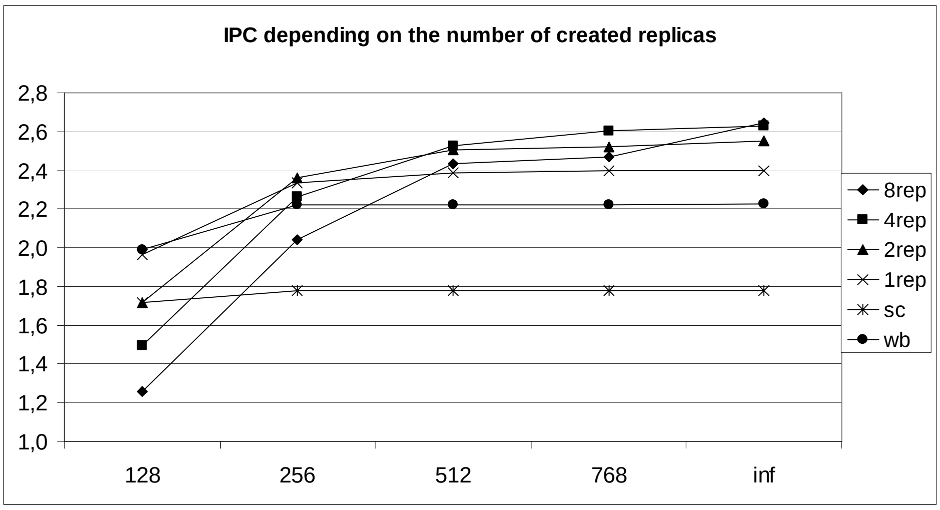

Figure 4.25: IPC varying the number of replicas and available registers

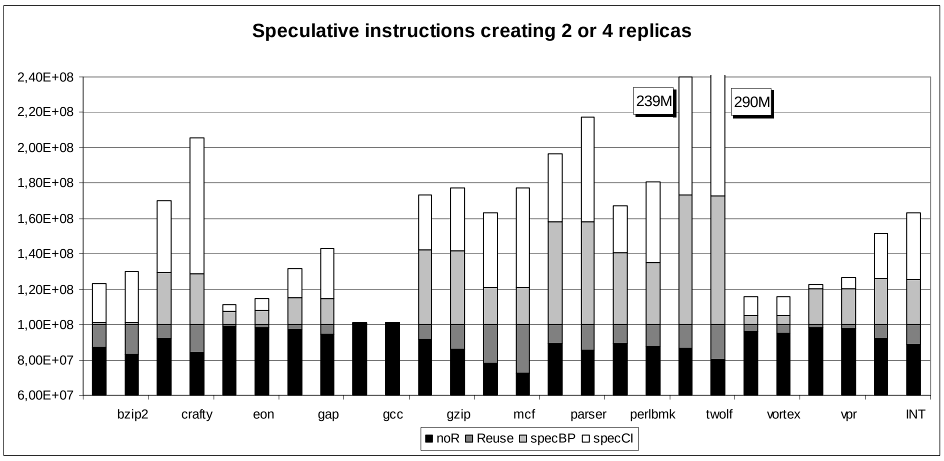

Figure 4.26: Distribution of instructions creating 2 and 4 replicas

14%. On the other hand, this extra reuse comes at the expense of a non-negligible increase in number of speculative instructions generated by the control independence scheme. We can also observe in Figure 4.26 that the amount of speculative activity generated by the control independence scheme is comparable to the activity generated by wrongly speculated instructions due to branch misspredictions.

Up to this point, we have shown that the CI selection schemes effectively reduces the number of speculative instructions at a low register cost. Anyway, even if our scalar version of the Dynamic Vectorization mechanism is focused to reduce the hardware complexity, we can take advantage on the hierarchical layout of the register file for both alleviating the pressure in scalar registers and not to increasing the cycle time due to a large register file.

Figure 4.27 shows the performance of the control-flow independence mechanism with a memory able to hold 128 (ci-h-128), 256 (ci-h-256), 512 (ci-h-512) and 768 speculative values (ci-h-768). The number of registers in the register file is given by the X-axis. The graph also shows a superscalar processor (scal), a superscalar processor with a wide bus (wb) and a superscalar processor with a wide bus and the control-flow independence mechanism with a monolithic register file (ci), for a varying number of registers.

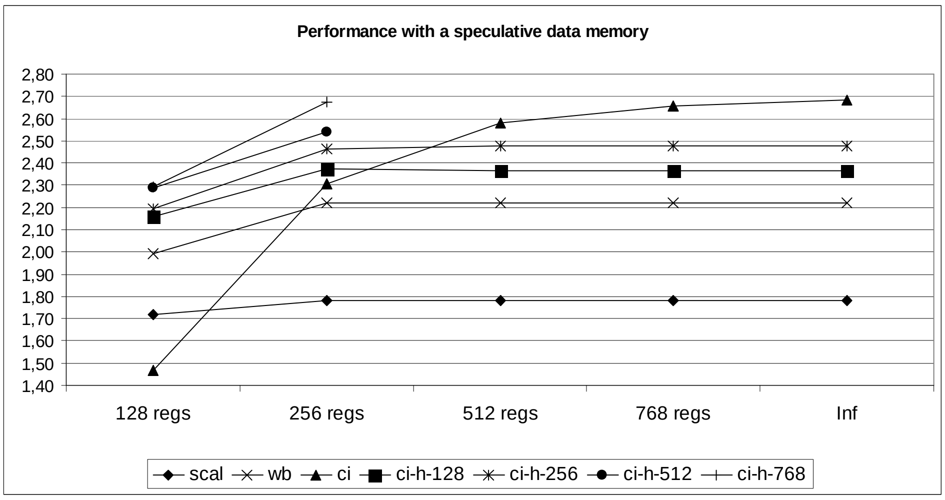

Figure 4.27: IPC with a hierarchical register file

Since this memory is out of the critical path and movements of values from this memory to the register file are not critical, longer latencies are allowed without degrading significantly the performance (a latency of 5 cycles only slowdowns about 3% in configurations with 256 registers in the register file and 768 positions in the proposed small memory). Several configurations of this memory have been simulated as shown in Figure 4.27.

Finally, Figure 4.27 shows that a register file of 256 registers and a memory holding 768 speculative values has about the same performance as a monolithic, single-latency register file with an unbounded number or registers. Following chapters will show a more refined selection mechanism that reduce considerably these register availability.

Control independent reuse out of the instruction window

Since the presented mechanism is the first scheme able to reuse data for control independent instructions out of the instruction window, it is interesting to see how it performs compared to other schemes limited to reorder buffer boundaries. For this comparison, we have chosen two state of the art mechanisms with different philosophy. For those mechanisms, to obtain the best case, the number of scalar register is unbounded, whereas for our scheme we keep on using a hierarchical register file with 256 registers in the lower level and 768 positions in the upper level.

The first design [CFS99] (Chou in Figures) bases its performance improvement in squash reuse. Data computed for instructions down the mispredicted branch is stored in a table for lately reuse it by control independent instructions after the branch recovery. So, instructions must have been executed before the branch misprediction detection.

The second mechanisms [CV01] (Skipper in Figures) uses heuristics to predict re-convergence points of hard-to-predict branches. As soon as one of those branches enters the pipeline, the fetch is redirected to the re-convergence point to prevent the execution of control dependent instructions. Skipper is limited to the reorder buffer boundaries. If this structure completely fills up, no new instructions after the re-convergence point can be executed.

For this study, we will show numbers varying the memory latency (to emphasize the memory gap), the number of entries in the reorder buffer (to overcome the memory wall problem) and the number of stages in the pipeline (to increase the latency of branches).

Figure 4.28 shows the performance obtained for a superscalar processor, the mechanisms Chou and Skipper and our CI scheme for a wide range of main memory latency. Caches latency has been also modified for realistic configurations as shown

Main memory latency	100	500	1000	2000
L1 data cache	1	2	2	4
L2 data cache	3	6	10	15

Table 4.4: Memory latencies for CI schemes

in Table 4.4.

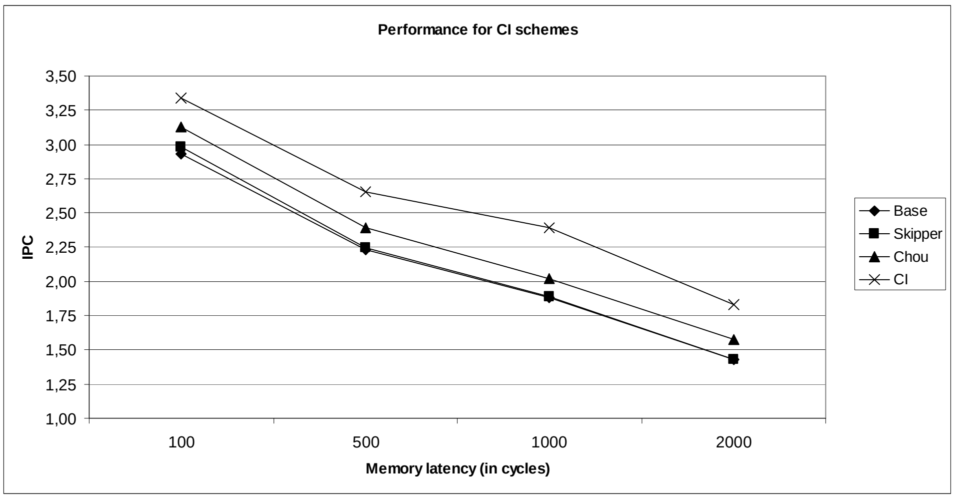

Figure 4.28: Performance of CI schemes varying the memory configuration

Several conclusions can be drawn from Figure 4.28. First of all, Skipper is a mechanism that is not able to outperform considerably the baseline (speedup ranges from 0,06% to -0,7%). Our simulations (obtained with the original simulator, kindly ceded by the authors of the paper) show that Skipper does not obtain any performance improvement when tested in current and future configurations. This mechanism obtains its performance improvement in Spec95 because branches and reconvergence points in those benchmarks are easy to predict. But when tested with SpecINT2K, since branch behavior is completely different, Skipper converts correctly predicted branches into mispredictions. This causes that the number of recovery actions increases counteracting the benefits of the mechanism. For this reason, even if we include Skipper in Figures, it will not be used as a comparator.

From Figure 4.28 can be also derived that reusing out of the instruction window is beneficial. Chou's mechanism performance improvement ranges from 6% to 10%, whereas the presented scheme speedups nearly 29% the baseline configuration. The more latency L2 misses have, the more speedup by our scheme.

Finally, Figure 4.28 also shows that when the memory gap is the predominant problem, all mechanism trend to the performance obtained by the baseline. Even if CI virtual enlarges the instruction window, the lack of enough in-flight instructions under a L2 miss prejudices dramatically the performance due to the abundance of processor's stall cycles waiting for data to be brought from main memory. This problem will be boarded in next chapter. On the other side, if memory is perfect, the CI scheme performance equals the baseline. This is due to the fact that branch latency is narrowed reducing the penalty of mispredictions.

Reorder buffer size also must have had into account. Figure 4.29 shows the performance for the CI schemes depending on the number of entries available in the reorder buffer with a latency to main memory of 1000 cycles. Load/store and issue queues are also scaled.

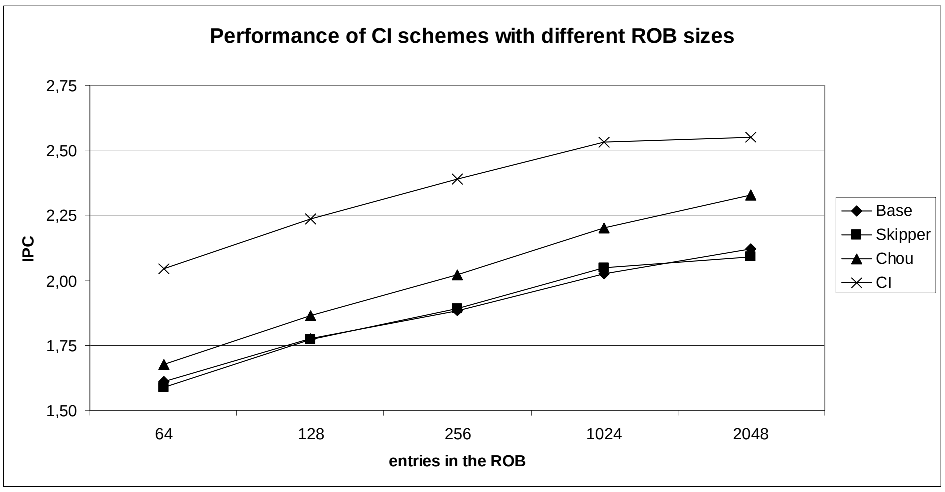

Figure 4.29: Performance of CI schemes varying the ROB size

As before, Figure 4.29 our CI scheme keeps on being the one who obtains better performance for all configurations, being Chou's scheme the second best.

Two details are interesting in this Figure (4.29. First, the CI scheme saturates when 1024 ROB entries are available whereas Chou´s performance improvement rests nearly linear. This is due to the fact that more registers are needed for the CI scheme with such large ROBs to replicate instructions. If registers are unbounded, as Chou's scheme, CI performance improvement becomes nearly linear.

The second fact is that despite of reducing the proportional speedup related to the baseline when enlarging the ROB, both schemes maintain the same linear improvement. Since more in-flight instructions are allowable with larger ROBs, both schemes are able to execute more instructions for later reusing data (Chou's mechanism) or to create more replicas for precomputing data (our mechanism). Anyway, Figure 4.29 shows that prefetch and precomputation of data keep being effective even for unimplementable larger windows.

Finally, Figure 4.30 shows the performance obtained for every CI scheme when the pipeline becomes deeper, i.e. the latency until branch resolution is augmented, thus increasing the penalty of branch mispredictions. Other parameters of the configurations remain unchanged.

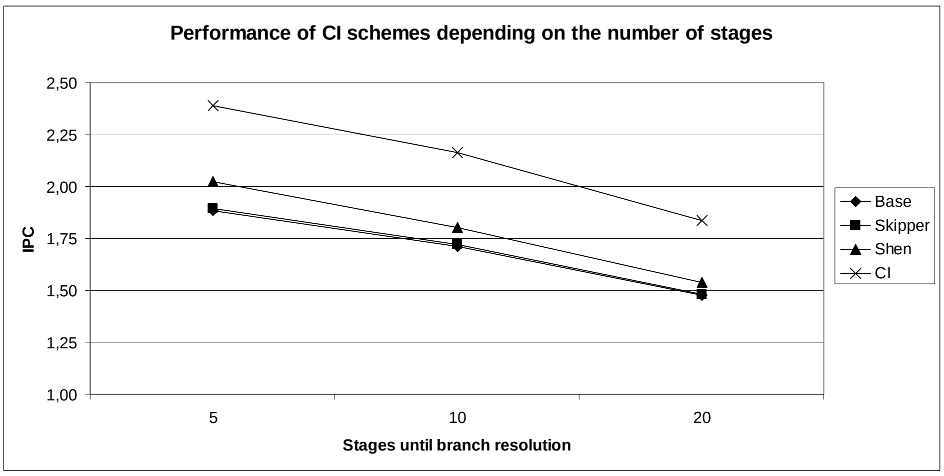

Figure 4.30: Performance of CI schemes varying latency of branches

This branch latency increment allows more in-flight instructions in processor after branch speculation. CI schemes benefit from this fact to increase the performance. But as a counterpart, branch mispredictions become even more critical since more cycles are needed to refill the pipeline after a recovery action.

4.6 SUMMARY

In this chapter a selection mechanism built on top of the scalar Dynamic Vectorization mechanism has been presented. This selection scheme is based on the fact that vectorizing only control independent instructions improves the effectiveness of the underlying replication mechanism since those instructions present a high ratio of reusing precomputed data.

We have also shown that this scheme improves the Dynamic Vectorization mechanism alleviating the pressure on the register file since less speculative work is created. Furthermore, we have shown that the inclusion of the hierarchical register file is not only worth to reduce the design complexity but to alleviate even more the pressure on the scalar registers since replicas hold their results in the upper level of this register file, lightening the work of the critical lower level.

Furthermore, we have also shown, that this is the first mechanism able to perform control independence reuse out of the instruction window and that this reuse is very beneficial. Compared to other state of the art schemes limited to ROB boundaries, in all tested configurations, our mechanism obtains performance improvements of nearly 14%.

Chapter 5

Overcoming the Memory Gap

5.1 INTRODUCTION

The memory gap is a very well-known problem in current processors. This makes memory instructions to have very long latencies when going out of the processor for data. These long latencies reduce drastically the opportunities of exploiting DLP and ILP in applications, not only because dependent instructions of a L2-miss load cannot execute until the data is brought from main memory, but also because independent instructions cannot commit due to the in-order nature of the commit process. This problem becomes worse when the instruction window completely fills up stalling the processor due to the lack of entries in the ROB, preventing the fetch, decode and execution of new instructions.

To reduce this problem, one solution could be enlarging the on-chip caches. With this solution the percentage of L2 misses decreases because more memory is available in the chip. But enlarging caches is expensive and it can impact seriously the cycle time of the processor or increase the latency of memory instructions.

Prefetch mechanisms have been also studied to reduce the penalty of L2 misses. These mechanisms try to predict memory addresses of loads to bring in advance the data that will be needed from main memory to the lower (and faster) levels of the memory hierarchy. However, prefetch is difficult when the memory access patterns of L2-miss loads are hard to predict (i.e., pointer-based memory accesses). Wrong prefetches overload the memory bus and pollute the on-chip caches with useless data.

From another point of view, mechanisms to enlarge virtually the instruction window have been proposed. These mechanisms try to solve the stall problem of the processors under L2 misses with out-of-order commit. This allows keeping executing independent instructions of the L2-miss loads before the data of this memory instruction is available. The main problem of these mechanisms is that it can be difficult to recover the state of the processor, e.g. under branch mispredictions.

In this chapter we describe two mechanisms that board directly this problem by selecting and replicating instructions following a L2 miss load. The first mechanism is based on the control-independent scheme to select in-flight instructions to replicate. The second approach analyzes the reorder buffer under a L2 miss load to replicate the current stream of instructions. This second scheme implements the third version of the Dynamic Vectorization mechanism: a separate engine that feeds the processor with replicas through the issue queue.

As we will show the virtual enlargement of the instruction window is performed by prefetch and precomputation of data by replicas. To isolate the effects of enlarging the instruction window, we will compare our mechanisms with a baseline where a state of art aggressive stride-based prefetcher is provided.

5.2 MOTIVATION

As memory latency increases, caches play a very important role to exploit the memory parallelism. However, when data does not exhibit spatial nor temporal locality, caches are nearly useless.

L2-miss loads and the in-order nature of the commit process provoke stalls in the processor because the instruction window completely fills (this happens about 61% of applications' execution time). Dependent instructions of these L2-miss loads cannot execute because their data is not available. However, independent instructions could be executed if entries in the instruction window were available. Figure 5.31 depicts the problem.

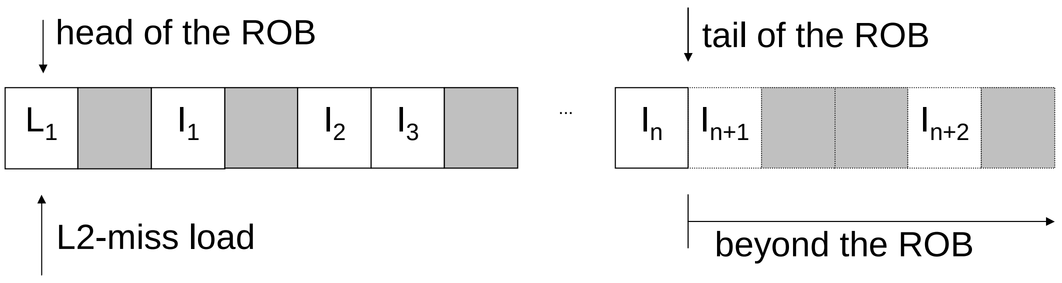

Figure 5.31: Instruction window filling due to a L2 miss load

Figure 5.31 shows a possible state of the ROB in a L2-miss load. L_1 is a L2-miss load. White boxes are independent instructions of *L1*. Grey boxes are dependent instructions of L_1. From L_1 to In are instructions in the ROB. From I_{n+1} onwards are instructions that are going to enter the pipeline next. In this scenario, the processor can execute I_1, I_2... because they are independent of L_1. However, instructions I_{n+1}, I_{n+2} ... cannot be executed because the fetch and decode stages of the processor are stalled due to the full filling of the ROB. In this paper we propose the speculative execution of these instructions (I_{n+1} onwards), thus enlarging the instruction window, so that, when these instructions are encountered, they will have their data computed and they will reuse it, avoiding their execution.

5.3 RELATED WORK

Prefetch mechanisms have been widely studied in literature. Software approaches [CKP91][LM96][MLG92] rely on the compiler to detect and evict cache misses at compile time. To do this, prefetch instructions are inserted in the code.

Hardware prefetching schemes [BC91][GG97][JG97][Jou90] study the access patterns of memory instructions to predict the next memory address to prefetch data. Low prediction ratio causes cache pollution and an unnecessary bandwidth consumption degrading performance.

Prefetch in multithread scenarios [BDA01a][CSK+99][CTWS01][ZS01] uses secondary threads to prefetch data for a primary thread. The problem of these mechanisms is that they need free thread contexts that may not be available if the multithread processor is well used.

Dundas in [DM97] proposed run-ahead execution. This mechanism creates special values for L2-miss loads that are propagated to dependent instructions. Independent instructions continue execution until the instruction window is full. When the data of the L2-miss load is available, a recovery action is performed to re-execute correctly the instructions following that load. In [MSWP03], run-ahead is enhanced with speculative retirement. The processor is extended with a complex checkpointing mechanism to allow out-of-order retirement, enlarging virtually the instruction window.

Several mechanisms [COM+04][DM97][MRH+02] to enlarge virtually the instruction window have been proposed at literature. These mechanisms perform out-or-order retirement of instructions to avoid stall cycles due to the lack of entries in the ROB. Complex advanced checkpointing mechanisms are used to recover the state of the processor e.g. in branch mispredictions. To alleviate the pressure on register files for these virtually large instruction windows, mechanisms of physical register late-allocate and early-release are used.

Lebeck in [LKL+02], proposed a scheme where instructions dependent on a long-latency operation are moved from a small (and faster) scheduling window to a large (and slower) waiting buffer until the operation is completed. At this point, instructions are moved again to the small scheduling window. This scheme allows large instruction windows regarding the cycle time of the processor with a small scheduling window. This approach requires a large register file that impacts the cycle time.

Our proposals differs in two main aspects. First, those mechanisms only execute instructions that enter in the pipeline. In our proposals, instructions that have not entered the processor can be speculatively executed. Second, no advanced checkpointing mechanism is needed to recover the state of the processor in branch mispredictions. In fact, the default recovery mechanism of the processor is not modified.

5.4 FIRST APPROACH: *L2MISS*

To overcome the memory gap we propose two different schemes based on the dynamic vectorization mechanism. This first approach reuses the control independent scheme to select instructions following a L2 miss load. As we will discuss, this is an aggressive mechanism that effectively improves performance. As a drawback, this mechanism creates large amounts of speculative instructions. The second scheme, *L2stall*, exploits the stall cycles due to a L2 miss load to replicate parts of the mainstream currently present in the instruction window.

5.4.1 Overview

This mechanism, *L2miss*, emphasizes the fact that futures values for independent instructions of a L2-miss load can be precomputed as soon as the instructions are encountered. In this case, the mechanism replicates sets of strided loads above the L2-miss load and dependent instructions of the stride loads after this L2-miss load. Replication is only possible if the L2-miss load is in a loop body. Figure 5.32 depicts how the mechanism works.

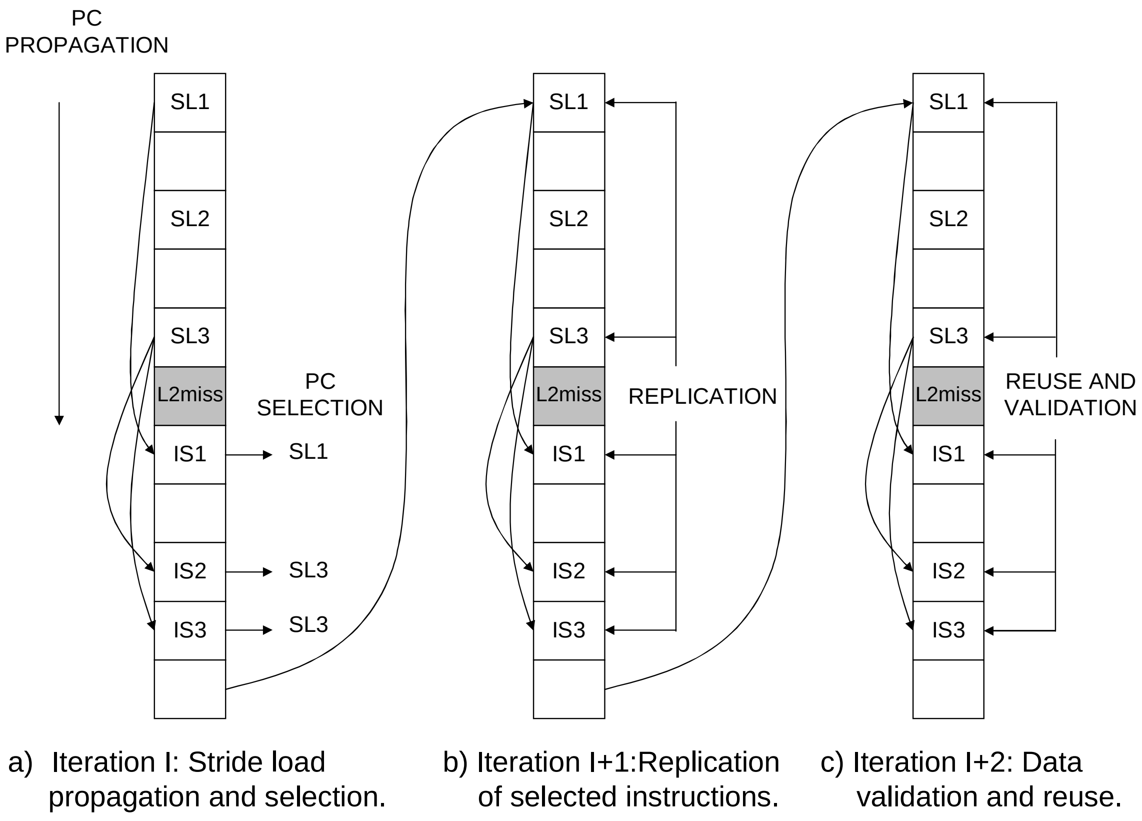
PC PROPAGATION
SL1
SL2
SL3
L2miss
IS1
IS2
IS3
PC SELECTION
SL1
SL3
SL3
REPLICATION
REUSE AND VALIDATION
a) Iteration I: Stride load propagation and selection.
b) Iteration I+1:Replication of selected instructions.
c) Iteration I+2: Data validation and reuse.

Figure 5.32: Steps of the *L2 miss* mechanism

In figure 5.32a, SL_1, SL_2 and SL_3 are strided loads before the L2-miss load that begin the dependence graphs for the independent instructions IS_1, IS_2 and IS_3 after the L2-miss load. Our first mechanism replicates SL_1 and SL_3 for later replicate

IS_1, IS_2 and IS_3 (Figure 5.32b). SL_2 will not be replicated because there are no instructions dependent on this strided load after the L2-miss load.

Strided loads after the L2-miss load are also considered as independent instructions because no memory address check is performed and will be also replicated.

This mechanism, as the control-independent scheme, works in four steps. First of all, it detects the strided loads and propagates their PCs down the dependence graph. In Figure 5.32a, the PCs of SL_1 and SL_3 are propagated to instructions IS_1, IS_2 and IS_3. Since no L2-miss prediction hardware has been included in the processor the stride detection and the PC propagation is performed continuously, to know, when a L2 miss occurs, the PCs of those strided loads that independent instructions depend on.

When a L2-miss load is committed, the second step of the mechanism is fired to identify the instructions independent of the L2-miss load. For this purpose, it analyzes the source operands of every instruction after the load to check whether they are independent of the destination register of this L2-miss load. If the instruction is independent, the set of propagated PCs of strided loads above the L2-miss load, on which the instruction depends, are selected for replication. In Figure 5.32a, as soon as instructions IS_1, IS_2 and IS_3 commit, the loads they depend on (SL_1 and SL_3) are selected for replication.

The third step is the replication of the selected instructions (SL_1 and SL_3) the next time they are fetched and decoded (Figure 5.32b). Dependent instructions (IS_1, IS_2 and IS_3) of a replicated instruction are also replicated by propagating the "replicated" characteristic down the dependence graph.

Finally, fourth step (Figure 5.32c), every time an instruction is fetched, it is checked whether it was previously replicated to validate the speculative precomputed data.

This mechanism focuses on the fact that only replication will be performed for independent instructions of loads with high probability of missing the L2 cache. Note that the first time a L2-miss load is detected, independent instructions may not reuse data because no data has been precomputed.

In Figure 5.32 it is supposed that the L2-miss load has a high ratio of L2 misses. Independent instructions will not have precomputed data the first time the L2-miss load is detected. But for next iterations (in Figure 5.32c, Iteration *I+2* onwards), as independent instructions have been replicated, precomputed data would be available for reusing.

5.4.2 First step: strided load propagation

Figure 5.33 shows the hardware modifications in the decode stage to implement this first step of the mechanism.

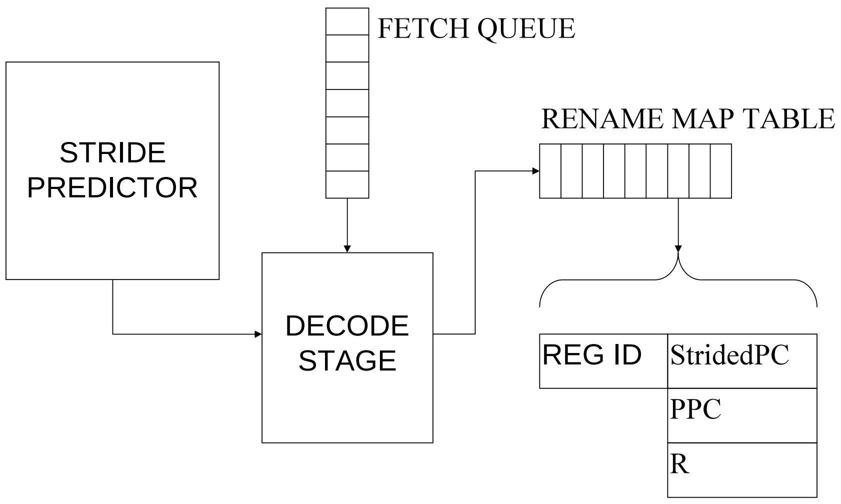

Figure 5.33: Hardware modifications in the decode stage

Every time an instruction is fetched, its PC is checked on the stride predictor to see whether it is a strided load or not. If the instruction is a strided load, its PC is put in the new field StridedPC of the entry corresponding to its destination logical operand in the extended rename map table.

If the fetched instruction is a load but not a strided load, the StridedPC field is set to 0. Arithmetic instructions copy the content of the field StridedPC of their source operands to the ones of their destination operands. In parallel, a copy of the fields StridedPC of the source operands is stored in the ROB. As the number of

StridedPCs is limited (2 per entry), only the first StridedPC of the source operands is propagated to the destination operand.

This strided load detection and propagation is performed continuously during instruction execution to ensure that independent instructions after the L2-miss load know these PCs when the selection of instructions is fired.

5.4.3 Second step: strided load selection

As soon as a L2-miss load is committed this second step is fired. In this step, the target strided loads for replication are selected. As said before, only independent instructions of the L2-miss load will be replicated. The independence check for instructions following a L2-miss load is performed when the instructions commit to reduce the amount of extra hardware. To implement this step, few modifications in the commit stage are needed (Figure 5.34).

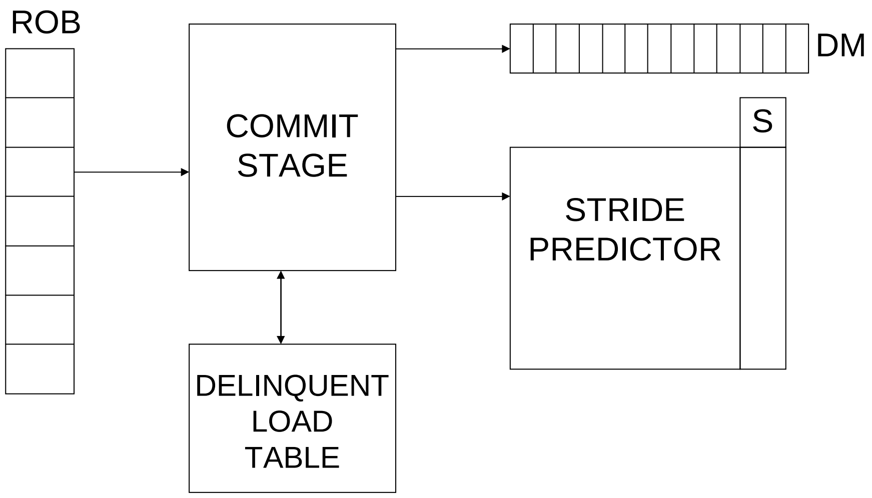

Figure 5.34: Hardware modifications in the commit stage

Stride load selection involves two main structures: a table called delinquent load table (DLT) where the PCs of the L2-miss load are stored and a mask of bits (one per logical register) called DM (Dependence Mask) used to know whether an instruction is dependent on the L2-miss load. An extension of the stride predictor (every entry is extended with a bit called S) is also needed.

When this step is fired, the mechanism clears every position of the DM except the bit corresponding to the destination operand of the L2-miss load. Following instructions set the bit corresponding to their destination operand with the logical OR of the bits corresponding to their source operands. Independent instructions are those instructions with all the bits of their source operands cleared.

Once independent instructions are detected, they select the strided loads from which they are dependent. To perform this, the bit S of the stride predictor is set for every strided load an independent instruction depends on and which PC is stored in the entry of the ROB for this instruction.

The last structure, the DLT, holds up to 8 PCs of the L2-miss loads for which the mechanism has previously selected instructions. If no entry is available in this table, the LRU load is replaced. Every committed non-L2-miss load checks its PC in this table. If the PC is found, following independent instructions of this load clear the bit S for the strided loads they depend on. This prevents previously selected strided loads from keeping being selected, and replicated, when a load, that has previously missed the L2, hits the L1 or the L2.

5.4.4 Third step: instruction replication

Once a strided load with the corresponding S flag set is fetched and decoded, multiple replicas of it are created and dispatched to the issue queue.

As in the control independence scheme, this and the next one steps implement the underlying scalar Dynamic Vectorization mechanism. The next mechanism introduces the third and last implementation of the scalar DV mechanism: a separate engine to select and replicate instructions.

5.4.5 Fourth step: data validation

Like in the control independence mechanism, data created speculatively by replicas must be validated. For validation purposes, every entry in the rename map table is extended with the field PPC and the bit R. Stride checking for strided loads and producer's PC checking for arithmetic instructions is performed to validate data. Instructions with correctly speculated data pass to the commit stage for validation. Other previously wrong replicated instructions deallocate resources (scalar registers and the entry of the SRSMT) of replicas and execute normally and perform a recovery action to execute normally the instruction, and if possible, new replicas are created.

5.5 SECOND APPROACH: *L2STALL*

This second approach exploits the stall cycles of the processor under L2 miss load to enlarge speculatively the instruction window. We will provide a separate engine that will feed the processor with replicas obtained from analyzing the mainstream of instructions currently available in the reorder buffer.

To implement this mechanism we present the third version of the Dynamic Vectorization mechanism. Based on the scalar design, this implementation is thought as a separate engine connected to the main core of the processor through the issue queue and the SRSMT table.

5.5.1 Overview

One of the main goals of the mechanism is to use the idle functional units during L2 miss loads without delaying the non-speculative instructions. This makes the decision of when to fire the mechanism become important. For this case, two choices are possible. First, wait to the reorder buffer to completely fill. In this case, we ensure that functional units will be available since most of the instructions have finished and remain in the reorder buffer waiting to be retired. But this is not the best choice. From the L2 miss detection to the start of the precomputation, some cycles are wasted. In the worst case, the mechanism is not fired because the instruction window never fills up under L2 miss load due to the low ratio of fetch and decode, losing opportunities of starting the mechanism.

We have adopted the second choice. The mechanism will be fired as soon as a L2 miss load is the oldest instruction in the ROB since the probability to reach a stall status of the processor is high. With this method a continuous flow of speculative instructions will feed the processor without disturbing the non-speculative instructions currently present.

The mechanism works in four stages. When the mechanism is fired, the first stage explores sequentially the ROB looking for independent instructions of the L2-miss load. When all the ROB is analyzed, the fetch and decode stages of the processor are started up to explore instructions that are going to enter into the processor next.

The second stage allocates resources to replicate the selected independent instructions. In particular, destination physical registers are allocated for replicas.

The third stage replicates the selected instructions and dispatches the replicas to the issue queue for execution.

Finally, fourth stage, after the L2 miss load retirement, new fetched and decoded instructions check whether they have been speculatively preexecuted. If they are, they must validate the precomputed data to use it.

Figure 5.35 depicts the hardware added to implement this mechanism.

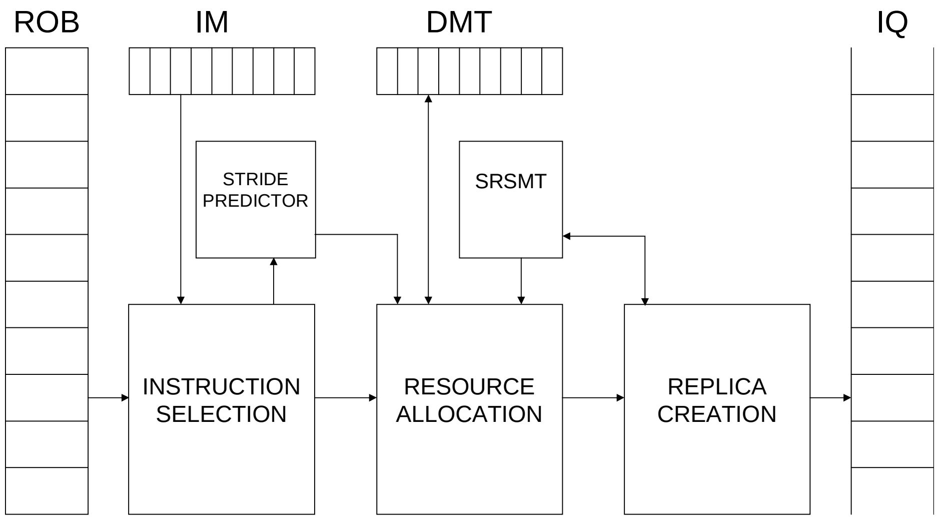

Figure 5.35: Stages of the *L2stall* mechanism

More details of these four stages follow.

5.5.2 First stage: instruction selection

As soon as a L2 miss load becomes the oldest instruction in the ROB, a signal is triggered to the engine of the mechanism, starting up the analysis of the instruction window looking for independent instructions of that load. To check whether an instruction is independent or not of the L2 miss load, we have adopted a simple approach. A mask of bits, called Independence Mask (IM) is provided to propagate the dependences among instructions. This mask has one bit per logical register.

When the mechanism is fired, every position of this mask is cleared except the one of the destination operand register of the L2-miss load. Following instructions propagate the values of their positions corresponding to their source operands in the IM to the position of the destination operand with a logical OR. Only instructions with all the bits of their source operands cleared are selected and passed to the next stage. Loads are always supposed to be independent of the L2 miss load.

To virtually enlarge even more the instruction window, when the exploration reaches the end of the ROB (90% of the times), the processor fetches and decodes speculatively new instructions. These instructions are supposed to come next from the last instruction stored in the ROB. Fetched instructions under a processor stall are predecoded by this first stage of the mechanism. Since we considerate these instructions as always speculative, they don't need to occupy an entry in the reorder buffer. Therefore, like the instructions created by the mechanism, these fetched instructions disappear after writing back their results.

5.5.3 Second stage: resource allocation

Selected independent instructions are passed to this stage to allocate resources for their replicas. Two main structures are involved: the DMT (Dependence Map Table) structure and the RM (Replication Maps) table.

To avoid misspeculations as much as possible, we rely on the vector patterns present in the code. So, every chain of replicable instructions must begin with a strided load. In order to detect these loads, a stride predictor like the one in is included in the processor. This predictor studies every load to detect a stride pattern. For replication we have considered only chains of instructions beginning with a strided load, since it is easy to compute the next effective addresses through the stride predictor. Althought it is possible that the effective addresses are mispredicted. In these cases, the validation stage of the mechanism will ensure that only replicas with a correctly computed address can reuse speculative data. So, a second selection in this stage is performed. Only instructions are replicable if they are strided loads or they depend on replicable instructions (arithmetic instructions).

Arithmetic instructions use the DMT table to know if they are replicable. The DMT has as many entries as architectural registers. Every replicable instruction stores its PC on the entry corresponding to its destination operand. Arithmetic instructions check the entries corresponding to their source operands to know whether they are dependent on a replicable instruction, i.e. there exists a PC in any of the entries corresponding to its source operands. Non-replicable instructions clear the position corresponding to their destination operand. Arithmetic instructions are only replicable if all their source operands are produced by replicated instructions.

After these checks have been performed, resource allocation, such as scalar registers and an entry in the SRSMT table, begins as in the normal scalar version.

5.5.4 Third stage: instruction replication

Instructions reaching this stage have all needed resources to be replicated. Replication consists in creating as many copies of an instruction as the number (up to four) of registers allocated for that instruction. Replication presents a different behavior, depending on the instruction. For speculative loads, the mechanism changes the memory addresses with those speculatively created by the stride predictor. Arithmetic instructions use as source operands, the destination registers of the instructions they depend on (available in the RM table). For both kinds of instructions, the allocated registers are used as source operands, one per replica.

In the current scenario, a processor nearly stalled due to a L2 miss load, the engine can benefit on the dispatch logic to enqueue instructions in the issue queue. Every cycle, this third stage uses the available slots of the dispatch logic to feed the execution engine of the processor with the recently created replicas (up to four due to replica creation logic simplicity).

5.5.5 Fourth stage: data validation

After the blocking L2 miss load is retired, speculative data can be used in a non-speculative fashion. In order to ensure the correctness of the data, a process of validation must be performed, similar to the normal scalar version of the Dynamic Vectorization mechanism. In this case, to avoid an excess of extra speculative instructions, mispredicted instructions are not replicated again.

5.6 PERFORMANCE EVALUATION

To evaluate and compare the mechanism we will assume a baseline configuration with a wide bus to the L1 data cache. Furthermore, the baseline configuration includes an aggressive prefetch (wb+pref in Figures). This is done to emphasize the benefits of the proposed mechanism beyond those coming from just prefetching. To extend our study, regarding the choice of when to start the L2stall mechanism, we will reference it as L2stall if the mechanism is not fired until the processor is stalled or L2nostall if the scheme is fired as soon as the L2 miss load is the oldest instruction in the instruction window.

Statistics of the proposed mechanisms with the aggressive prefetch will be also provided (miss+pref, sta+pref and nosta+pref for, respectively L2miss, L2stall and L2nostall extended with an aggressive prefetch).

5.6.1 Performance improvement

Figure 5.36 shows the IPC of the proposed mechanisms compared with the baseline (wb) and the baseline with aggressive prefetch (wb+pref).

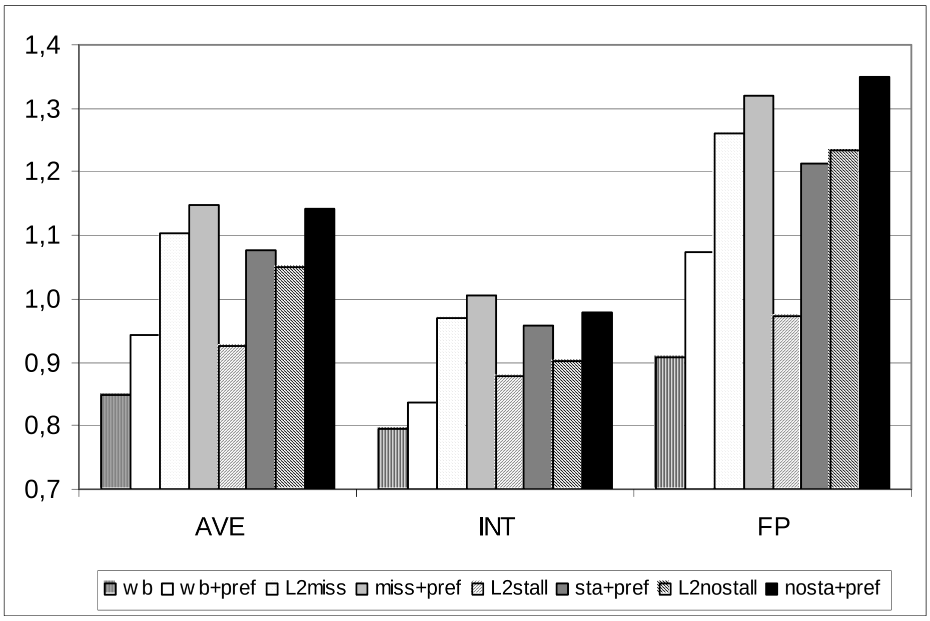

Figure 5.36: Performance comparison between the L2 mechanisms

As shown in Figure 5.36, the aggressive prefetch mechanism improves the baseline about by 11,2% on average, by successfully reducing the number of L2 load misses by about 40%.

Our mechanisms outperform the baseline by about 29%, 8% and 23,5% respectively for L2miss, L2stall and L2nostall. However, when compared against the baseline with aggressive prefetch, these speedups are reduced to 16%, -3% (slowdown) and 11% for the L2miss, L2stall and L2nostall mechanisms respectively. Figure 5.36 also shows that there is room for improvement. When implementing the proposed mechanisms on top of the configurations that already include a prefetcher, there is a noticeable performance improvement (21%, 13% and 20% for miss+pref, sta+pref and nosta+pref compared with wb+pref). The reason is that the prefetch schemes increase the potential of the proposed mechanisms. Prefetch increases the performance by removing strided-based L2 misses. Our proposals are basically fired on

L2-miss loads. When these mechanisms are combined, prefetch prevents our mechanisms to be fired on easy-to-predict L2 misses, focusing in those L2-miss loads without a clear access pattern. So, there is a net performance increment due to prefetch easy-to-predict L2-miss loads, and preexecution of independent instructions of not prefetcheable loads.

There are three main sources of performance improvement. The first one is the memory subsystem. The ability of going beyond the instruction window allows our mechanisms to avoid future L2 miss even before the prefetch detects them. Furthermore, the ability of preexecuting instructions allows prefetching loads without a clear access pattern. This L2 miss reduction is show in Figure 5.37.

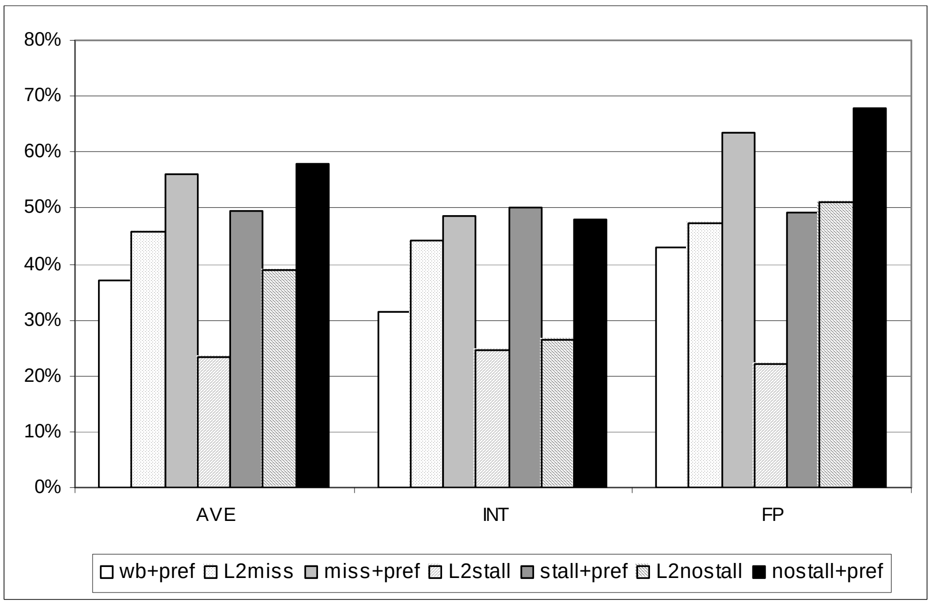

Figure 5.37: L2 miss reduction for the L2 mechanisms

Furthermore, our mechanisms are able to manage more efficiently the wide bus due to the high percentage of unit strides and the exploitation of data locality explicitly denoted by replicas, reducing the total amount of accesses. Figure 5.38 shows the percentage of accesses in which 1, 2, 3 or 4 elements can be served from the same L1 access, for the baseline configuration compared with the proposed mechanisms. Although this exploitation is not important in the baseline configuration, compared with a superscalar processor with 2 scalar ports, the wide bus suffers a slowdown of less that 2%. Note that the control logic of the wide bus is simpler than that of 2 scalar ports.

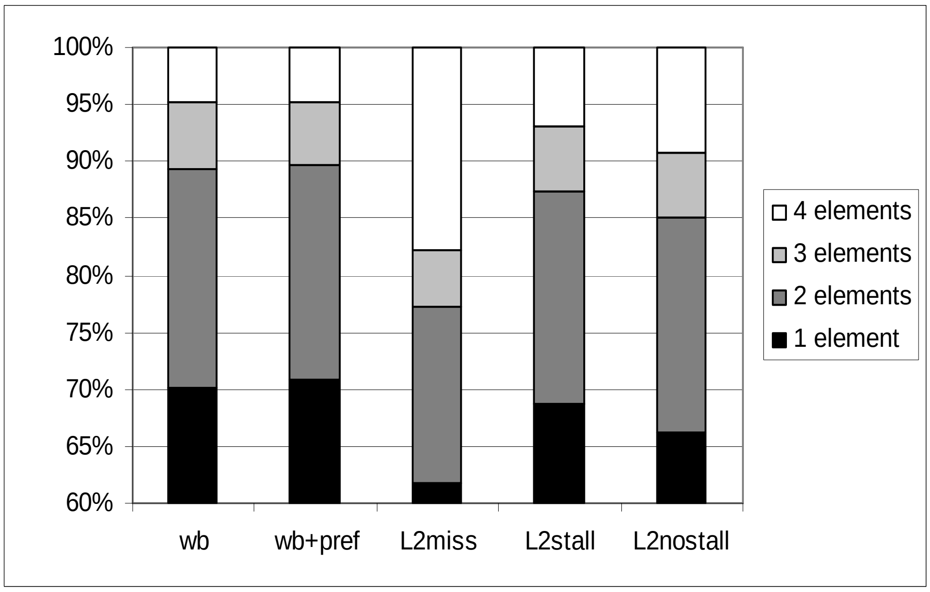

Figure 5.38: Percentage of number of elements bypassed per wide bus access

The second source of performance improvement is the reuse of speculative precomputed data. Figure 5.39 shows the number of commit instructions that cannot reuse data (black portion, commit), the commit instructions that reuse data (dark grey, reuse), the speculative instructions created by branch mispredictions (light grey, specbp) and the number of speculative instructions created by our mechanisms (white, specL2).

The first algorithm, L2miss, is the one that creates more speculative instructions due to its aggressiveness (white portion, specL2), but is the one that is able to reuse more precomputed data (dark gray portion, reuse), nearly 16% of the commited instructions are able to reuse data precomputed by the mechanism. The other two mechanisms, L2stall and L2nostall create 4,5 and 13 millions of extra instructions respectively, reusing 2,7 and 10 millions of instructions. As seen in this figure, the two conservative mechanisms create less speculative instructions but are able to reuse, proportionally, more data. Furthermore, this reduction of the number of extra speculative instructions impacts, directly, in the amount of extra resources

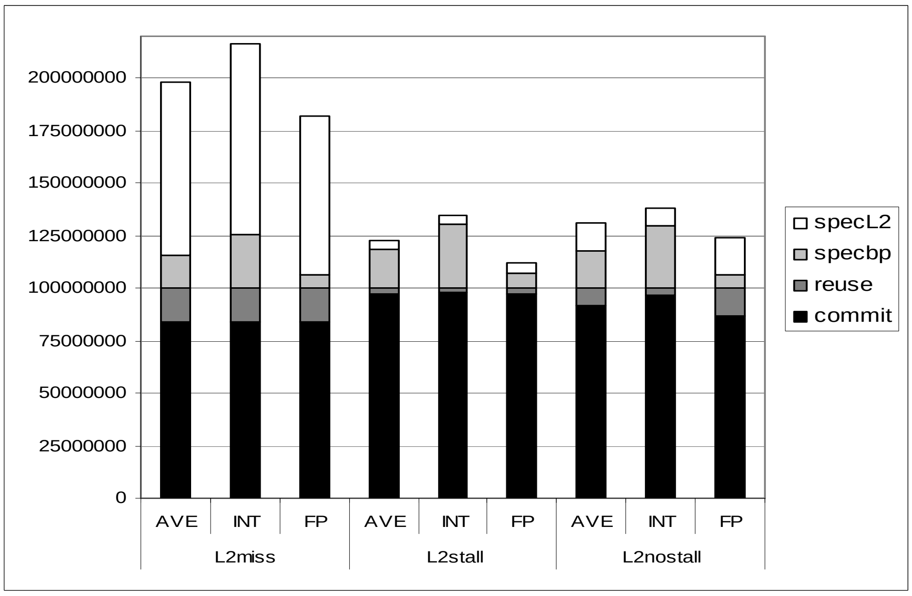

Figure 5.39: Distribution of instructions for the proposed L2 mechanisms

needed to achieve good speedups.

Finally, the third source of improvement is the virtual enlargement of the instruction window, due to the ability to create data for instructions that have not enter into the pipeline. Figure 5.40 summarizes this study, showing the performance obtained with a 64-entry ROB and the proposed mechanisms (64-L2miss, 64-L2stall and 64-L2nostall for the L2miss, L2stall and L2nostall respectively) compared to the baseline with a ROB of 256 entry. Furthermore, the proposed mechanisms are evaluated with a reorder buffer of 256 entries (256-L2miss, 256-L2stall and 256-L2nostall) and are compared to a large processor (2048-base) with 2048 entries in the ROB and 256 in the L/S and issue queues.

This virtual enlargement benefit can be viewed from two points of view. First, with a bounded hardware, we can achieve larger instruction windows (about 1180 and 1154 instructions, on average, with the L2stall and the L2nostall mechanisms respectively). A large processor with 2048 entries in the ROB, and 256 in the LSQ and issue queues and infinite physical registers (2048-base in Figure 8) only improves about 10% the L2smiss mechanism (256-L2stall in Figure 5.40. Note that the complexity of the 2048-entry processor is higher than the implementation of the proposed mechanism due to the complexity of managing such large structures without impacting the cycle time.

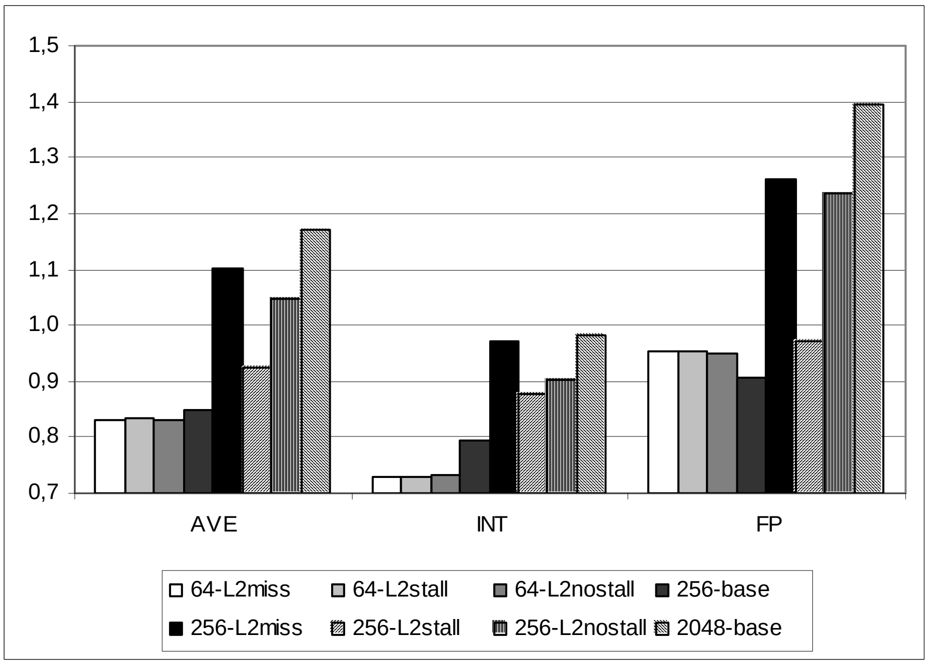

Figure 5.40: Importance of the ROB size for the L2 mechanisms

Following with Figure 5.40, the second point of view is that with smaller instruction windows, thus, reducing processor's complexity, we can achieve nearly the same IPC. In configurations with 64 entries in the ROB and an issue queue of 32 instructions the mechanisms only loses, on average, 1% of IPC compared to configurations with 256 entries in the ROB and an issue queue of 128 instructions (256-base, Figure 5.40).

Note in Figure 5.40, that the proposed mechanisms outperform the 256-base configuration for SpecFP thanks to the fact that in regular codes the number of L2 miss is higher, increasing the chances of firing the presented mechanism, and it is easier to build large instruction windows since most of branches are easy to predict.

Second, with smaller instruction windows, thus, reducing processor's complexity, we can achieve nearly the same IPCs (in configurations with 64 entries in the ROB and an issue queue of 32 instructions the L2miss mechanisms only loses 0,5% of IPC compared to configurations with 256 entries in the ROB and an issue queue of 128 instructions).

5.6.2 ***L2miss* versus *L2stall/L2nostall***

One of the main motivations to implement the L2stall/L2nostall mechanisms is to reduce the complexity and the amount of resources required by the L2miss mechanism. Previous Figure 5.39 shows that the L2miss mechanism creates nearly 82,7 millions of extra speculative instructions to precompute data. These extra instructions use functional units to execute and scalar registers to hold the precomputed data. But an excess of extra instructions can produce two negative effects. The first one is functional unit overloading, what may cause a delay for non-speculative instructions and reduces the global performance of the processor. Our studies show that the L2miss mechanism increases the average functional unit utilization more than 12%, while the L2stall and the L2nostall mechanisms only increase this average about 1

The second effect is the allocation of scalar registers of the upper level of the hierarchical register file for misspeculated data. This reduces the chances to create correctly speculated data because no registers are available to hold it. The average of free positions in the upper level of the register file for the L2miss mechanism is 83. The averages for the L2stall and L2nostall mechanisms are 592 and 453 respectively. This gives an idea of the size of the upper level of the register file that can be decreased for the L2stall and the L2nostall mechanisms without decreasing dramatically the speedup, and, in fact, our studies show this. Reducing the number of positions from 768 to 384 in the upper level only decreases 1,7% and 2% the performance improvement for the L2stall and the L2nostall mechanisms respectively. The L2miss mechanism improvement is reduced about 10%.

These results shows that the L2stall and L2nostall mechanisms need fewer resources to achieve high speedups because they create less extra speculative instructions. In addition, the L2stall and L2nostall mechanisms have a simpler implementation than the L2miss mechanism.

5.6.3 Virtual enlargement of the instruction window

Run-ahead [DM97] [MSWP03] is a state-of-the-art mechanism to alleviate the memory gap. It is based on executing speculatively instructions following a L2 miss load. As soon a L2 miss load is detected, the processor enters into Run-Ahead mode and creates a checkpoint for later recovering. An INV value is associated to the L2 miss load as a result, and propagated to the depending instructions. Since all the work performed after the L2 miss load is speculative, instructions perform a pseudo-commit to deallocate their associated entry in the ROB and scalar registers allowing to keep on fetching and decoding speculative instructions, thus, enlarging the instruction window. Once the L2 miss load is retired, the mechanism performs a recovery action to recover the state of the processor in that load. Run-ahead benefits on the prefetch of loads following the L2 miss load.

The presented mechanism differs from Run-ahead in one main aspect: precomputed data can be reused by instructions out of the instruction window. Since Run-ahead discards all the speculative data precomputed by instructions following the L2 miss load, the only source of performance improvement is prefetch of L1 and L2 miss loads.

Our results show that, opposite to what authors claim in [MKSP05], reuse of precomputed data is possible and beneficial. The fact that, for the presented mechanism, more that 77% of instructions reuse data is what raises the IPC about 9% compared to Run-ahead.

Figure 5.41 compares the baseline (wb), the baseline with aggressive prefetch (wb+pref), Run-ahead and the *L2nostall* mechanism (L2nostall). The processor with Run-ahead has the same configuration parameters as the baseline, and it also includes the wide bus.

Another important factor is the number of extra speculative instructions created by the Run-ahead scheme compared against the proposed mechanism. Our simulations show that Run-Ahead executes 22% of extra instructions. All this speculative

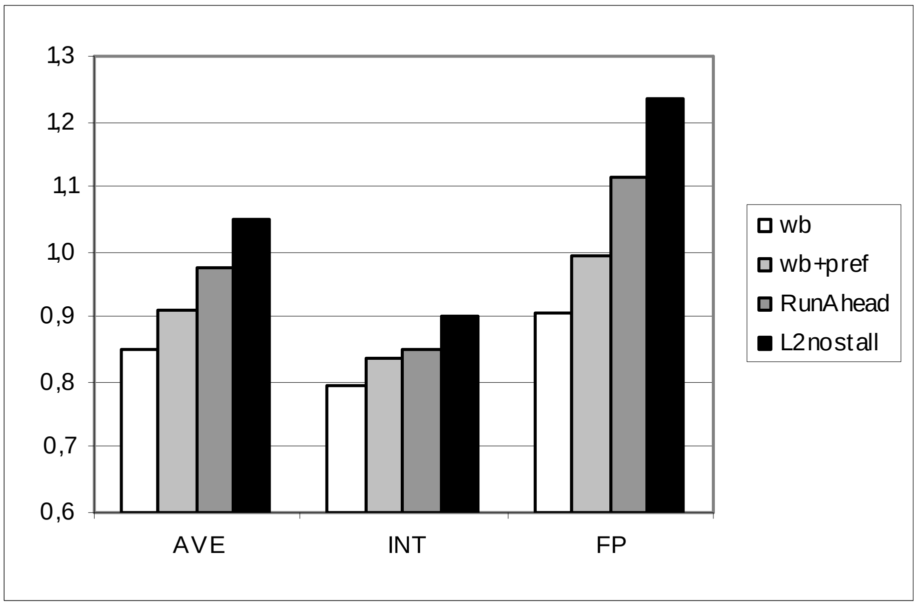

Figure 5.41: Performance effect of the virtual enlargement of the instruction window

work is discarded, since the performance improvement of Run-ahead comes directly from prefetch.

On the other hand, the *L2nostall* mechanism only executes 13% of extra instructions (59% less than Run-ahead). And, from these instructions, nearly 77% of precomputed data is successfully reused by validation instructions, giving only 3% of useless work, much less than the created due to branch mispredictions.

5.7 SUMMARY

In this chapter we have presented two mechanisms able to overcome the memory gap by virtually enlarging the instruction window.

The first mechanism, *L2miss* is based on the control-independence scheme presented in Chapter 5 to select replicable instructions following a L2 miss load. The second mechanism, *L2stall* uses the third version of the Dynamic Vectorization mechanism. In this case, the basis mechanism is implemented as a separate engine that communicates to the main core through the issue queue and the SRSMT table. This second mechanism analyzes the status of the reorder buffer under a L2 miss to replicate the mainstream.

We have shown that both mechanisms outperform the baseline processor taking advantage on the main sources of performance improvement of the basic Dynamic Vectorization mechanism.

Furthermore, we have compared the proposed mechanisms each other to show that a net reduction of resources is possible by keeping the performance. In particular, the *L2stall* mechanism is more cost-effective than the *L2miss* since creates a negligible amount of extra instructions (13M and 82,7M of replicas for the L2stall and the L2miss mechanisms respectively).

Finally, we have demonstrated that the virtual enlargement of the instruction window is positive. First, with smaller reorder buffers, the L2stall mechanism achieves nearly the same performance of processors with large instruction windows. Second, the reuse of data precomputed for instructions out of the instruction window is beneficial. For this case, we have used as baseline a processor with Run-ahead. We have shown that this effective reuse raises about 9% the performance.

Chapter 6

Cost-Effective Dynamic Vectorization

6.1 INTRODUCTION

As shown in previous chapters of this dissertation, dynamic vectorization is a mechanism to effectively boost the performance of a processor. This performance improvement comes from different sources such as prefetch, preexecution and the virtual enlargement of the instruction window.

But this improvement is not for free. Additional extra instructions are created to speculatively precompute values in advance for instructions. These extra instructions need resources, such as registers and functional units, for execution. An excess of resource allocation for speculative instructions can be prejudicial for the execution of non-speculative instructions. In a processor with the dynamic vectorization mechanism and a moderated number of registers, the execution of non-speculative instructions can be delayed if most of the scalar registers are allocated by the speculative instructions. Even if we apply the two level register file, it is necessary to reduce the amount of extra hardware to simplify the design of the processor.

In other scenarios where functional units become critical, the extra instructions can overload the back-end stages of the processor delaying the non-speculative instructions.

But things go worse when these speculative instructions are misspeculated. Not only resources are allocated for these misspeculated instructions, but also all the performed speculative work will be discarded.

From another point of view, this misspeculated extra instructions also affect the optimal performance achievable by the dynamic vectorization mechanism. Since resources are allocated to execute misspeculated instructions, chances to create correctly speculated vector instructions are reduced.

In this chapter we present several heuristic-based schemes to select instructions that are worth to vectorized. The decision of vectorizing an instruction cannot be indiscriminate. Wrong selection mechanisms can reduce the performance improvement of the dynamic vectorization mechanism if performance-impacting instructions are not vectorized. Thus, the main goal of these selection schemes is to reduce as much as possible the extra misspeculated instructions without penalizing the performance improvement of the dynamic vectorization mechanism.

Since we are presenting several heuristics, figures of performance and reduction of the number of extra instructions are provided having into account that these metrics are necessary but not enough to select the best scheme. Chapter 8 will solve this lack by introducing Figures of Energy-Delay^2.

To quantify the benefit of the proposed schemes, we will use as a baseline for our simulations a superscalar processor with a scalar version of the DV mechanism. On top of this processor we will build the proposed heuristics.

The structure of this chapter is as follows. Next section motivates the creation of the heuristics by showing the optimal performance of the DV mechanism. Section 6.3 studies what are the most important sources of replica misspeculation. Next section describes the employed heuristics. The evaluation of the proposed schemes and a summary close the chapter.

6.2 MOTIVATION

Figure 6.42 shows the performance achieved by the basic scalar vectorization mechanism (blind) compared with a superscalar processor (scal), and a superscalar processor with a wide bus to the L1 data cache (wb).

Figure 6.42 shows the maximum performance achievable by the dynamic vectorization mechanism (optimal in Figure) with a processor where the extra speculative instructions do not delay the execution of non-speculative instructions, i.e. unbounded resources for vectorization are available and replicas are executed in a separate set of functional units. This Figure (6.42 motivates this study, showing that a correct management of resources can improve up to 66% the basic DV mechanism.

Figure 6.43 shows the amount of extra misspeculated instructions generated by the dynamic vectorization mechanism, divided into speculative loads (white portion)

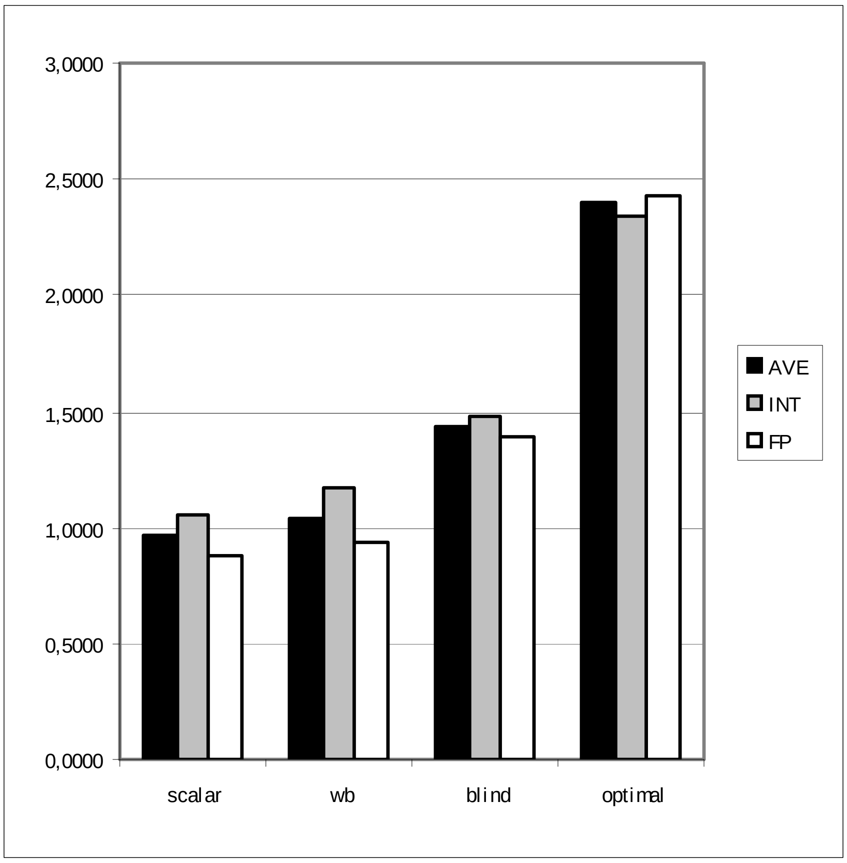

Figure 6.42: Optimal performance of the dynamic vectorization mechanism

and arithmetic instructions (black portion of each bar). The dynamic vectorization mechanism creates, on average, 60 million of replicas which data is not validated.

The goals of this chapter are to find out the main sources of these misspeculations and to create heuristics to reduce them without penalizing the performance improvement of the dynamic vectorization scheme.

6.3 DYNAMIC VECTORIZATION MISPREDICTIONS SOURCES

Where do all these 60 millions of misspeculations come from? We will explain it with the code in Figure 6.44

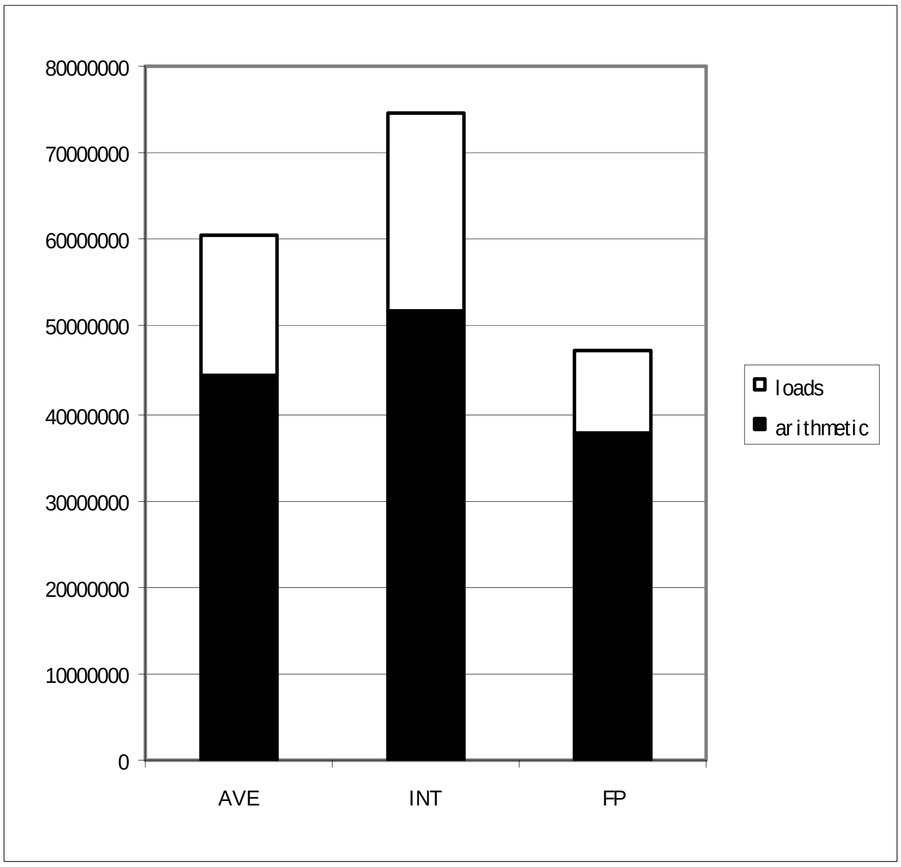

Figure 6.43: Extra instructions generated by the dynamic vectorization mechanism

Stride mispredictions

Strided loads are detected at runtime through a stride predictor. Since this is a speculative detection, mispredictions are possible. Look at instructions I_2, I_5 and I_6 in Figure 6.44. I_2 is a strided load which stride changes every 100 loop iterations. This is due to instruction I_5, which computes the modulus 100 of R0 to obtain the stride of the next element to be processed. In this case, since the stride predictor only tracks the last two stride computations, it is not able to detect this pattern (the change of the stride every 100 iterations). Due to this fact, instances 100*N+1 (being N every integer greater or equal to 0 and less or equal to 10) cause a stride misprediction.

In the dynamic vectorization, stride mispredictions are an important source of misspeculations as we will shown later in Figure 6.45 not only because replicas are

```
I1:   LD  R2, a[R0]
I2:   LD  R3, b[R1]
I3:   ADD R4, R2, R3
I4:   ADD R5, R2, R6
I5:   MOD R7, R0, 100
I6:   ADD R1, R1, R7
I7:   ADD R6, R6, R7
I8:   ADD R0, 4
I9:   CMP R0, 1000
I10:  JNE I1
```

Figure 6.44: Misspeculation example code

wrongly created for strided loads but replicas of dependent instructions must also be squashed since they are wrongly precomputated.

Wrong construction mispredictions

This misprediction source occurs when either, one scalar operand becomes replicated or vice versa, one replicated operand becomes scalar.

In the code of Figure 6.45, having into account that there are stride mispredictions, when the load I_2 changes the stride, if dependent instructions are replicated, one of their source operands change from replicated to scalar. This is due because that load is not replicated since another stride is detected.

On the other hand, when I_2 presents again a stride, thus being replicated, dependent instruction I_3 causes a misprediction since R3 becomes replicated.

DAEC mispredictions

To avoid long associations among sets of scalar registers and vectorized instructions, a counter in every entry of the SRSMT is provided (see Section 3.6.3). This counter is incremented in every mispredicted branch if a vectorized instruction has not validated any speculative data since the last branch misprediction. When this counter reaches a threshold value (in our case 2), resources of a vectorized instruction are deallocated. This is had into account as misspeculation because more data than the strictly necessary has been created.

To simplify our studies we will not considered DAEC as a possible target for heuristics given that is hard to know precisely when values discarded by this counter are really misspeculated. Anyway, we will count replica removal due to DAEC as mispredictions.

Scalar source operand mispredictions

This misspeculation source only appears in instructions vectorized with one scalar operand. In this case, misspeculations are caused to the change of value of this scalar source operand.

In Figure 6.45, I_7 changes the value of R6 every 100 loop iterations (as for the strided load in instruction I_2). This change in the value of the operand makes that instruction I_4 causes mispredictions, provoking the squash of possible replicas created for this instruction since it is replicated thanks to R2 (created by I_1).

Replicated source operand mispredictions

One of the replicated source operands of a vectorized instruction has changed. This means that either the instruction that created the replicated source operand of an instruction has changed or it has newly replicated.

Figure 6.45 shows the contribution of every source of mispredictions to the total amount of extra misspeculated instructions for every benchmark. *Stride*, *construct*, *daec*, *construct1*, *sourceop* and *vect* stand for, respectively, stride mispredictions, scalar to replicated operand mispredictions, DAEC mispredictions, replicated to scalar mispredictions, scalar operand value changed mispredictions and vector operand changed mispredictions.

From Figure 6.45 it is easy to note that *construct* is the most important source

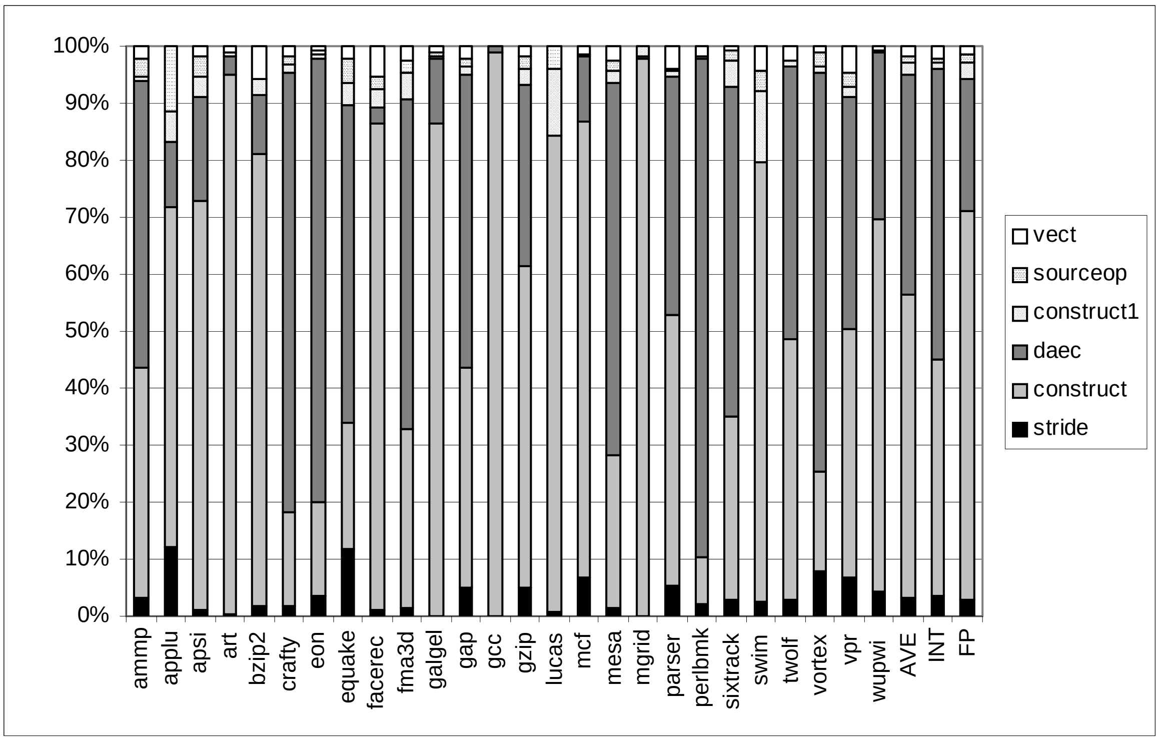

Figure 6.45: Sources of misspeculations of the DV mechanism

of mispredictions, weighing nearly 53%, on average of the total. This is due to the fact that it is the collateral effect of the other sources of mispredictions, i.e. nearly always one misprediction of any kind, except Construct, causes, by propagation, a Construct misprediction.

The second most important source is DAEC, counting nearly 38%. It is important to see what this number expresses. A considerable amount of squashed replicas are due to the fact that they are not validated during a long period of time. Allowing the association of instructions and replicas during an infinite period of time effectively increases the performance about 10% but it is not implementable since registers are bounded.

Surprisingly, stride mispredictions only causes, directly, 4% of mispredictions. This means that a simple stride predictor, as the one we use, is effective for our scheme. Since it is not trivial, indirect mispredictions caused by stride mispredictions are not measured.

The other sources of mispredictions form the other 5% of total mispredictions. A negligible percentage worthless to improve.

6.4 HEURISTICS

This section describes all the heuristics we have ideated to reduce the number of mispredictions. Even if they all have been implemented and evaluated, only the ones that show a better trade-off between performance and misprediction reduction will be used for comparison purposes.

6.4.1 Fine grain heuristics

Fine grain heuristics are those that are applied individually to an instruction. The following heuristics try to reduce the number of mispredictions by modifying dynamically the number of replicas that are going to be created per instruction using some previous history of replica reuse of that instruction. They are intended to study the instruction as an individual without knowing the execution scope.

Incremental number of replicas

The first heuristic is created from a conservative point of view. Studying the ratio of mispredictions and reused instructions, it is legitimate that it is more probable that a replicated instruction reuses less data than the created by replicas. For this reason, what this heuristic pretends is to assign a low number of registers the first time an instruction is replicated. If the instructions demonstrates to reuse all the precomputed data, the number of replicas is incremented in following replications.

Notice that this heuristic reduces the chances of data reuse for very regular patterns of code since less replicas per instruction are created at the beginning even if the instruction could reuse the speculative data.

To implement this heuristic (and the next one), a small associative table (32 entries) is provided to keep the number of replicas reused the last time an instruction was replicated. Note that this prediction does not need to be correct, preventing for storing all the bits of the PCs of all replicable instructions in this table.

This heuristic proves to be effective reducing the mispredictions, but it also penalizes the performance about 10% since in many cases no replica per instruction is created due to code irregularity.

Decremental number of replicas

This heuristic engages the misprediction problem from a point of view opposite to the one of the last heuristic. Since replicas expose ILP and DLP, it would be beneficial to create the maximum number of them the first time an instruction is replicated. Future reuse will decide either if that number of replicas is correct or it has to be reduced in one unit since there is data that has not been validated.

Even if this approach is more aggressive than the previous one, it does not reduce the chances of replica creation and data reuse for regular codes. Unfortunately, for irregular codes, mispredictions are not reduced as much as desirable (about 5% on average per SpecINT2K).

Assignation by halves

The main problem of the two previous heuristics if their impossibility of allowing patterns of both incremental and decremental number of iterations. Such patterns, like 4-4-2-4-4-2, make previous heuristics to create always the maximum, in the case of the incremental heuristics, or the minimum number, in the case of the decremental heuristic, of replicas allowed.

To solve this, we propose a third heuristic where the number of replicas per instruction can be either increased, in the case that all precomputed data is reused, or decreased if some misprediction is detected.

In this case, the first time an instruction is replicated, the mechanism creates half the maximum number of replicas allowed. Every time all data is reused, this number is doubled until the maximum is reached. If mispredictions are detected, the number of replicas is halved until the instruction is not replicated anymore.

As in the previous heuristic, even if the amount of mispredicted extra instructions is reduced, the performance is severely impacted, being reduced about 9,5%.

Heuristic based on last reuse

Most of the current speculative execution mechanisms are based on the idea that the execution of the code follows certain inertia. For example, branch predictor mechanisms present a high ratio of hits because conditional branch outcome, in most cases, is the same that the last time they were found. This inertia makes that some characteristics of certain portions of the code become highly predictable. Applied to the vectorization mechanism this means that the probability that a vectorized instruction validates the same amount of data that the last time it was vectorized is quite high.

Following this idea, we provide the processor with a table where the number of replicas an instruction validated the last time it was vectorized is hold. Since it is not necessary that this prediction have to be correct, only a tag instead of the whole PC is stored per instruction.

This heuristic must be extended to prevent a constant non-vectorization situation. Forcing that the mechanism creates a maximum number of replicas that can only decrease, it is possible a situation where all instructions have this number to zero, i.e., no replicas are created. To avoid this kind of situations, the heuristic is extended: when the number of replicas validated is the same that the predicted number, the next predicted number will be the current one with some increment. The higher this number the more misspeculated instructions are created, and the worse this heuristic is.

Figures of this heuristic are shown in Section 6.5.

6.4.2 Coarse grain heuristics

All the previous heuristics treat the instruction as an individual without a scope. In order to solve this, we present a family of heuristics to solve the misprediction problem at loop level.

Number of iterations

Dynamic vectorization can only be applied in loops. In sequential codes, the mechanism is not able to vectorize any instruction because no strided load is detected. Fortunately, most of the current codes expend a high percentage of the time executing instructions that belong to a loop body.

This characteristic of codes makes current processors, e.g. Intel Centrino, to include a loop predictor to improve the branch prediction ratio. These loop predictors try to predict the number of times a backward branch will be taken, to predict the number of iterations of a loop. This is performed this way because backward branches are the typical branches at the end of a loop body.

Applying this idea to the DV mechanism means that a replicated instruction will validate as many replicas as the number of iterations of the loop it belongs to. In other words, it exists a direct association between the number of replicas and the number of times the nearest backward conditional branch is taken.

Taking advantage of this idea, we extend the branch predictor of the processor with a counter per entry where the number of consecutive "taken" is recorded for backward branches. In addition, a global register, called MRBB (most recent backward branch) with that number of the last committed backward branch is provided in the processor. So, when an instruction is vectorized, the number of replicas created is computed with the minimal between the maximum number of replicas that can be created per instruction and the value hold in the MRBB.

Dependent load blockade

Since replication is performed down the dependence graph, a mispredicted instruction provokes a cascade of squashes of all replicas created for dependent instructions. This work squash is necessary because the correct execution of the code must be ensured. This is the typical effect of snowball.

Following this directive, a simple block-based heuristic is to avoid this cascade of mispredictions by preventing the replication of instructions dependent on a high mispredictable replicable instruction. To accomplish this, since our mechanism begins the creation of replicas from a strided load, and after that, all the dependent instructions, the only thing that must be avoided is the replication of that loads.

So, when an instruction is detected to be problematic, all dependent loads even if strided, are not replicated. Directly dependent arithmetic instructions are also marked for not being replicated the next time, even if they have a replicated source operand.

With this procedure, we ensure that replication for blocks of code that present a high probability of causing mispredictions is deactivate.

Figures of this heuristic are shown in Section 6.5.

Block information

Even if this method is not a pure heuristic, we have decided to include it in this chapter due to its nature since it is a mechanism to reduce the amount of mispredictions without penalizing the performance. In fact, as shown later in Figure 6.5 we will see that this method raises the basic mechanism performance about 14,7% on average.

For this method, we have extended the basic mechanism with a table, called LI (Loop Information) where runtime information of loops is stored. For every loop (up to 32) with replicated instructions, the mechanism maintains:

- The address of the first instruction of the loop and also the index of the table.

- The number of iterations the loop performed the last time it was executed.
- A vector where every position holds the number of the last replica an instruction validated the last iteration of the loop. Since in our scheme only allows up to 4 replicas per instruction, every position has 2 bits. Furthermore, to limit the size of every entry of this table, only information for the first 16 instructions are maintained.

With this information the mechanism is able to know where a loop begins, the number of iterations the loop performs, which instructions have not to be vectorized and how many replicas an instruction validates. This last number is obtained with the formula: replicas=(iterations/4)+replicas validated.

Every time an instruction is fetched, its PC is looked up in this table to see if a loop body begins. This PC is obtained studying the outcome of backward branches.

If the PC is found, the mechanism notices that a loop is going to begin, and studies the entry of the LI table to know whether a replicable instruction must be replicated, and in this case, how many replicas must be created.

In the case that the PC, as outcome for a backward branch, is not in the table, an entry in the LI table is reserved and information for this loop is tracked. Replication in this case works as in the basis mechanism.

As we will see later in Section 6.5 this is our best refinement of the dynamic vectorization mechanism.

6.4.3 Criticality

But heuristics not only must be applied to decide the number of replicas created per instruction, but to the necessity of power up the performance of the dynamic vectorization mechanism at a low cost.

To accomplish this, we introduce the definition of criticality. Several studies at literature [JLW01] [FRB01] try to conclude with an universal definition of criticality of an instruction but they have not a completed success. In fact, our feeling is that the definition of criticality must be tailored to the scope where it is applied.

For this reason, we define that an instruction is critical, in the scope of dynamic vectorization, when an instruction writebacks its results sooner than older instructions in the instruction window. This means that, by any reason, an instruction has finished later than other younger instructions in sequential order.

This definition works well for the dynamic vectorization mechanism since prefetch and preexecution can alleviate the problem of out-of-order finalization of instructions.

So, the criticality method will detect which instructions writeback their results out-of-order, and in the case that younger instructions are not finished, this scheme will mark them for replication.

In fact, this scheme is easily implementable. As soon as an instruction writebacks their results, the PCs of unfinished instructions, up to 32, younger than the one that has finished are stored in a table.

After that, when a replicable instruction reenters the pipeline, its PC is checked against this table, and if found, this instruction is effectively replicated.

Prefetch and preexecution by the dynamic vectorization mechanism provides data even before the instruction has entered the pipeline, thus reducing its latency. This makes that an instruction writebacks its results earlier since validation instructions do not execute, by pass directly to the commit stage.

Finally, this scheme is effective reducing the number of replica mispredictions. Since replicas are not created until instructions are marked, no speculative extra computation is performed. If no replica is available, no misprediction will occur.

Figures of this technique are shown in the next Section (6.5).

Scheme	Description
DV	Basic dynamic vectorization mechanism
DAEC100	Replicas are not squashed until DAEC reaches a value of 100
DepEsc	Instruction with one scalar operand are not replicated
Last reuse	Fine grain heuristic *Last Reuse*
Loop reuse	Coarse grain heuristic *Loop Reuse*
L-last reuse	Fine grain heuristic *Last Reuse* only applied to loads
L-loop reuse	Coarse grain heuristic *Loop Reuse* only applied to loads
Criticality	Heuristic *Criticality*
Load blockade	Coarse grain heuristic *Dependent load blockade*
Blocks	Heuristic *Block information*

Table 6.5: Evaluated heuristics

6.5 PERFORMANCE EVALUATION

This section shows the benefit of applying the previously defined heuristics in top of a superscalar mechanism with the dynamic vectorization mechanism. Table 6.5 resumes the evaluated heuristics, related names will be used in Figures:

To provide a fair comparison, the evaluation is extended with two schemes: *DAEC100* and *DepEsc*. The first one, DAEC100, modifies the threshold of the counter DAEC to avoid replica squashes until this counter reaches the value of 100. This makes that associations of registers and replicated instructions last more time, increasing the probability of reusing data. Unfortunately, this reduces the chances of replicating more instructions since less positions in the upper level of the register file are available to replicate new instructions.

The second scheme, *DepEsc*, is the basic Dynamic Vectorization mechanism with one limitation: instructions with one scalar and one replicated operand are not vectorized.

These schemes do not need a modification of the Dynamic Vectorization mechanism to be implemented. They are just modifications of the configuration parameters in simulations, and the fast way to remove the *DAEC* and the *wrong construction mispredictions* by the DAEC100 and DepEsc schemes respectively.

Figure 6.46 shows the effectiveness of the heuristics reducing the number of mispredicted extra instructions.

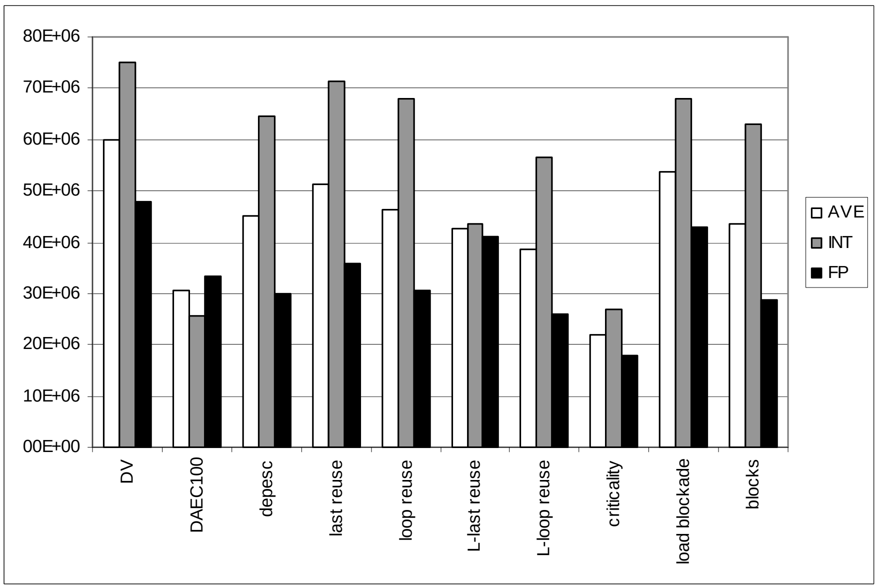

Figure 6.46: Reduction of misspeculated instructions of the presented schemes

Figure 6.46 shows that every heuristic reduces notably the amount of mispredicted extra instructions, being the *Criticality* heuristic the one that reduces the mispredictions the most, about 36%. The modifications of the parameters, schemes *DAEC100* and *DepEsc*, are also useful for this intention, reducing the number of speculative instructions about 50% and 74% respectively.

It is also noticeable, that this reduction in the number of misspeculated instructions is more effective in SpecFP2K than is SpecINT2K. This is due to the fact that these heuristics force that replication only will be performed in the most regular pieces of the code.

But even if the reduction is advisable but itself, the main goal is to reduce the number extra instructions without any performance penalty. Figure 6.47 shows that the reduction of mispredicted instructions effectively improves the performance for every heuristic except for *DAEC100* and *L-loop reuse*, being the most competitive the *Block Information* heuristic.

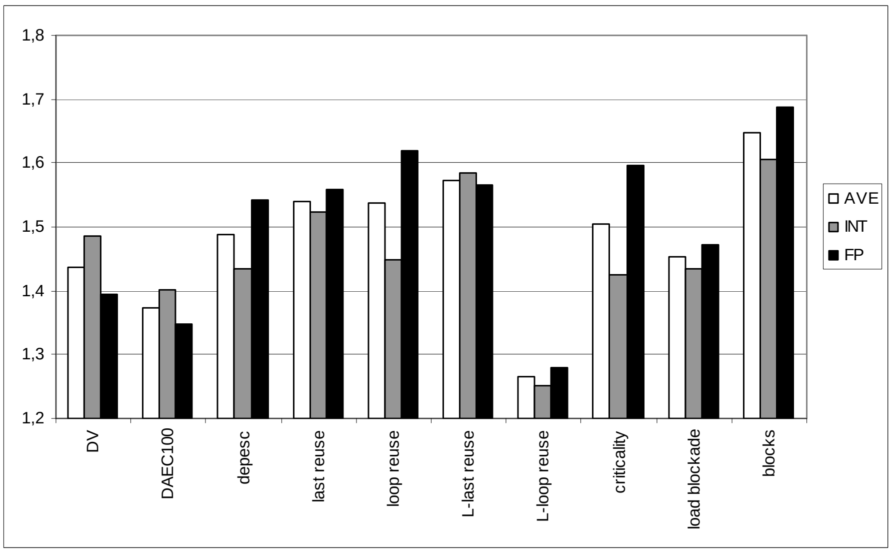

Figure 6.47: Performance improvement of the presented heuristics

Even if DAEC100 is one of the most competitive heuristics in removing extra instructions, is the second one which performs the worst, losing 4,5% of speedup. This is due to the fact that registers are associated to replicated instructions for too long, reducing the chances of replicating more instructions.

On the other hand, DepEsc demonstrates to be very competitive in performance for SpecFP2K but not for SpecINT2K. This means that replication must be aggressive for irregular codes to obtain some performance benefit.

The other heuristics prove to be effective improving the performance, with speed ups ranging from 1,2% (load blockade) to 14% (Block Information) even if they do not keep the number of instructions that validate data, as shown in Figure 6.48.

Some heuristics also reduce the number of validation instructions, but maintain or improve the performance. The main reason is that there are instructions that impact more positively in the performance than others. There is where the Criticality definition plays. The heuristic Criticality is the fourth heuristic that performs the better and it is the one that reduces the number of mispredicted instructions the

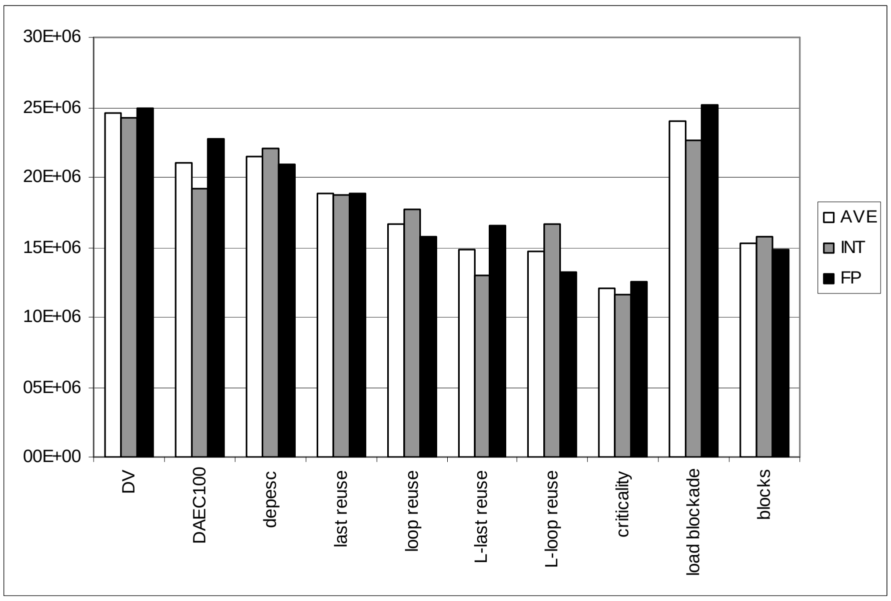

Figure 6.48: Validation instructions of the presented heuristics

most. This means that, in fact, there are some instructions that are more critical than others. Validating only 12 millions of instructions is able to perform 5% better than the basic Dynamic Vectorization mechanism creating 64% less speculative instructions.

As shown, most heuristics effectively reduce the number of extra speculative instructions without penalizing or even increasing the performance of the basic DV mechanism. This gives an idea that blind vectorization is not recommendable at all and a selection of replicable instructions must be performed to take as much advantage as possible of the Dynamic Vectorization mechanism. This concept will be described deeper in Chapter 7 where the concept of Energy-Delay^2 is introduced.

6.6 SUMMARY

In this Chapter we have described a set of heuristics applied directly to the dynamic vectorization mechanism. From the motivation we have shown the necessity to reduce the number of extra mispredicted instructions without penalizing the performance improvement of the mechanism.

We have presented several families of heuristics divided in accordance with their scope in *Fine grain* and *Coarse grain* heuristics depending if they affect just one instruction or a whole loop body.

Furthermore, we have added a third heuristic based on *Criticality*. We have defined the concept of a critical instruction tailored to the Dynamic Vectorization mechanism.

We have evaluated all these heuristics in terms of reduction of extra instructions, performance improvement and data validation. Two more baselines are added to the figures: one increasing the threshold value to the DAEC counter and other where one instruction is not replicated if it has one scalar and one vector source operand.

Figures have shown that our heuristics effectively reduce the mispredicted extra instructions and that this reduction, in most cases, increases the speed up of the basic Dynamic Vectorization scheme.

Finally, we have shown that the concept of criticality is positive since the Criticality heuristic improves considerably the performance, creating 64% less speculative instructions and validating only 12 millions of instructions (50% less than the Dynamic Vectorization mechanism).

Chapter 7

Mechanism Comparison

7.1 INTRODUCTION

At first, the dynamic vectorization mechanism was ideated to be implemented in a general purpose desktop processor with or without vector extensions. But nowadays tendencies and constrains make us reopen this decision.

The variety of scenarios where a processor can be coupled, enforces a set of constrains that must be had into account when stating the initial requirements of design. It may not have sense to use a power-constrained processor in systems where high performance is required.

For this reason, vendors use to release several families of processors targeted to a particular scenario. This is what, for example, Intel Co. makes with the Centrino Mobile series, where enlarging the battery life of a laptop is the maximal constrain. On the other hand, Intel floods the market of high performance desktop computers with the Pentium 4 series. AMD fills its piece of the market cake with the Athlon 64 and Sempron series for mobile and desktop computers. Server systems are also aimed mainly with the AMD Opteron and the Intel Itanium 2 processors.

As we have shown in previous chapters, the Dynamic Vectorization can be easily tailored to solve a wide range of problems/necessities. This adaptation allows us to select which version of the mechanism fits best depending on the scenario constrains, being the following question the main motivation for this Chapter: which is the best variation of the mechanism for a given scenario?

To solve this answer, it is necessary, first, to define which constrains have to be fulfilled for a given baseline processor. We will base this study in three main requirements: hardware, performance and Energy-Delay^2 requirements.

The objective of this Chapter is not to provide a general ranking of mechanism, but report which mechanisms fit best for a given constrain.

7.2 MECHANISM COMPARISON

For simplicity, the comparison of mechanisms will be provided only for a subset of all the schemes presented in this dissertation. Table 7.6 summarizes this fact.

Figures of every constrain will be provided in next sections together with a partial ranking of mechanisms for every scenario.

7.2.1 Hardware

Even if the number of transistors grows in every new generation of processors, thanks to technology, hardware constrains differentiate whether a mechanism is implementable or not in a given processor.

Nowadays, hundreds of millions of transistors are available to construct whatever proposed idea. Very often, this transistor availability is translated into extending regular structures such as on-chip memory, from one processor model to another inside the same family, because it is easier to verify the correctness of the implementation. But structure enlargement must be performed carefully to tolerate the new access time. Wrongly designed structures can penalize dramatically the processor's cycle time.

As we annotate earlier in this dissertation (Section 2.2, the Dynamic Vectorization mechanism includes regular structures that fit nicely in 1Ghz processors. Even if frequency is increased by means of superpipelining, Dynamic Vectorization does not impact the cycle time of the processor by two reasons: structures are out of the critical path of the processor and, second, they can be easily pipelined.

Another important aspect is the control logic. To measure the complexity of a design it is necessary a detailed layout of the processor. Since the drawing of such scheme can take years, we have left it as future work. But just few words to justify that our design is implementable: in order to provide a processor with dynamic vectorization capabilities, only it is necessary to extend the control logic available in the processor to manage the new fields of existing structures. For new structures,

Problem	**Scheme**	**Description**	**Section**
General	DV	Scalar Dynamic Vectorization mechanism	Chapter 3
Branch mispredictions	CI	Control independence selection	Chapter 4
Memory gap	L2miss	Control-flow independence based mechanism for L2 miss loads	Section 5.4
	L2stall	ROB analysis at processor stall cycles	Section 5.5
	L2nostall	ROB analysis as soon as a L2 miss load is detected	Section 5.5
DV mispredictions	DAEC100	Replicas are not squashed until DAEC reaches a value of 100	Section 6.5
	Last reuse	Fine grain heuristic *Last Reuse*	Section 6.4.1
	Loop reuse	Coarse grain heuristic *Loop Reuse*	Section 6.4.2
	Criticality	Heuristic *Criticality*	Section 6.4.3

Table 7.6: Compared mechanisms

such as the SRSMT, the access logic is not more complex than the required logic to supply data to/from a 4-way data cache. In our studies, we obviate the required hardware to implement the control logic of the Dynamic Vectorization mechanism and its variations.

Table 7.7 shows the amount of extra hardware needed to implement the proposed mechanisms. For every mechanism, the description of the new hardware and the total amount of extra bytes needed are shown (we do believe that this is a good measure to estimate hardware).

From table 7.7 it is easy to see that the amount of extra hardware to implement the basic Dynamic Vectorization mechanism (12,25KB) its negligible and really affordable in current and future processors. In fact, the scheme that requires more hardware (Loop reuse) only needs 28,25KB, which is less than the size of L1 data caches in current processors. Furthermore, if we double the L1 data cache of the baseline processor of this thesis (64KB to 128KB), the IPC is only raised about 6%, whereas the Loop reuse heuristic provides 46% of speedup.

Summarizing, in scenarios highly constrained by hardware limitations, the basic DV mechanism is the best choice since it is the scheme that less transistors need to be implemented.

7.2.2 Performance

Multimedia is one of the most important classes of applications executed in nowadays processors. Since these programs are very performance-demanding, end users ask to processor's manufacturers for powerful systems in which playing audio and/or video or even generate large 3D worlds run smoothly and without lags. For this reason, desktop computers are upgraded once and again, incorporating the latest hardware and software releases to fulfill the performance requirements of those applications.

Due to the wide range of versions of the dynamic vectorization mechanism presented in this dissertation, a final comparison of performance is necessary. Figure 7.49 summarizes all those studies, following Table 7.6 (IPC for SpecFP2K for the

Scheme	**Hardware**	**Total (Bytes)**
DV	Stride predictor (64 sets x 2 ways x 20 bytes per entry) + register file upper level (768 positions x 8 bytes per position) + SRSMT (64 sets x 4 ways x 15 bytes per entry)	12544
CI	DV + rename map table extension (64 entries x 16 bytes per entry) + extended stride predictor (64 sets x 2 ways x 1 bit per entry) + NRBQ (64 entries x 16 bytes per entry) + CRP (16 bytes)	21792
L2miss	DV + rename map table extension (64 entries x 16 bytes per entry) + extended stride predictor (64 sets x 2 ways x 1 bit per entry) + extended rename map table (64 entries x 1 bit)	20760
L2stall	DV + IM (64 entries x 1 bit per entry) + DMT (64 entries x 1 bit per entry)	12560
L2nostall	DV + IM (64 entries x 1 bit per entry) + DMT (64 entries x 1 bit per entry)	12560
DAEC100	DV	12544
Last reuse	DV + 2-way set associative table with 64 entries with 2 bit counters + 32 bit tags	13088
Loop reuse	DV + branch predictor extension (64K entries x 2 bits)	28928
Criticality	DV + Icache extension (8192 instructions x 1 bit per instruction)	13568

Table 7.7: Hardware requirements of the proposed mechanisms

CI scheme is not provided).

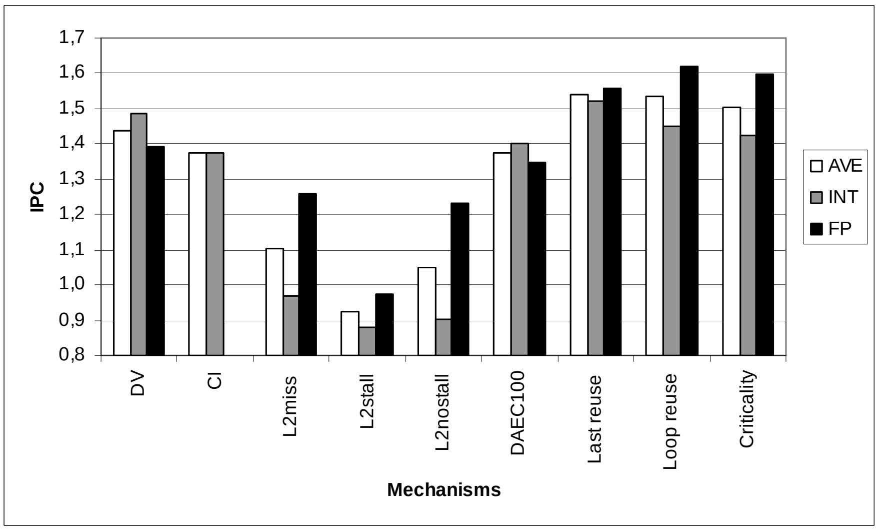

Figure 7.49: Performance comparison of the mechanisms

As shown in Figure 7.49 the schemes perform quite different. Averaged IPCs range from 0,92 (L2stall) to 1,54 (Last reuse). Selection focused on branch mispredictions or the memory gap degrade the performance of the basic mechanism since replication is only performed during short periods of time. In return, the heuristic-based schemes or the basic DV mechanism are continuously replicating instructions, increasing the chances of performance improvement.

Another important number on that Figure (7.49) is that no mechanism is able to perform better that the basic scheme for SpecINT2K. As previously said, in Chapter 4, replication for irregular codes must be as aggressive as possible to increase the chances of data reuse. On the other hand, heuristics-based schemes prove to outperform the DV baseline for SpecFP2K increasing until 16,3% (for Loop reuse) the speed up.

Even if we are discussing which mechanisms perform best, it is advisable to remember that the main objective of selection-based mechanism is to keep or increase as much as possible the performance with less resource requirements.

But lower performance does not necessarily means that a given scheme is worse than the DV mechanism. Another important factor must be had into account: the effectiveness executing instructions. Next section discusses this topic.

7.2.3 Energy-Delay^2

Hardware and performance are important requirements in current processors. The first one, hardware, decides which mechanism can be included in a layout. The second one, performance, ranks the processor in the market. But there are more important constrains when designing a processor.

Power dissipation [GH96] has become one of the most important problems in current processors. The rapid progress in technology scaling [Bor99], equal to quadrupling the number of transistors every three years, means that more temperature is generated per area unit, making a processor be provided with a more sophisticated thermal dissipation system.

There are a lot of work done in this topic. From mechanisms to dissipate efficiently the temperature [GH96], to schemes that reduce the activity [MKG98] to cool the chip.

This thesis does not provide low power designs for the DV mechanism, but will try to measure how effective [CL99] the basic scheme is, compared against the other proposed arrangements, executing instructions.

To measure this, we will use a well-known metric: energy-delay^2 (ED^2) [GH96]. This metric considerates how much energy an instruction needs to be executed related to the performance benefit that it generates. The less energy an instruction needs to obtain a given performance benefit, the better. Extrapolating to ED^2: the lower value of ED^2 the better.

Note that there is a substantial difference between Energy-Delay and Energy-Delay^2. The first metric is commonly used to measure low power, portable systems. Efficiency comes from increasing, for example, the battery life. The second metric, given that emphasizes the delay, is better for optimizing performance.

We will compute ED^2 in terms of #executed_instructions * CPI^2. We will assume that all the instructions waste the same amount of energy to obtain their results.

Even if this is not true, we are really confident that this is a good approach for the purpose of this study. *Delay* will be measured with the averaged number of cycles an instruction takes to completely execute.

For our study, a high value for ED^2 means that either a scheme creates an excessive amount of extra instructions or the ratio between performance an the number of extra instructions is not as high as the ratio obtained with the basic DV scheme.

Figure 7.50 shows the value of the ED^2 metric for the proposed mechanisms normalized to the basic DV scheme. The lower the value of the ED^2 for a given scheme, the better.

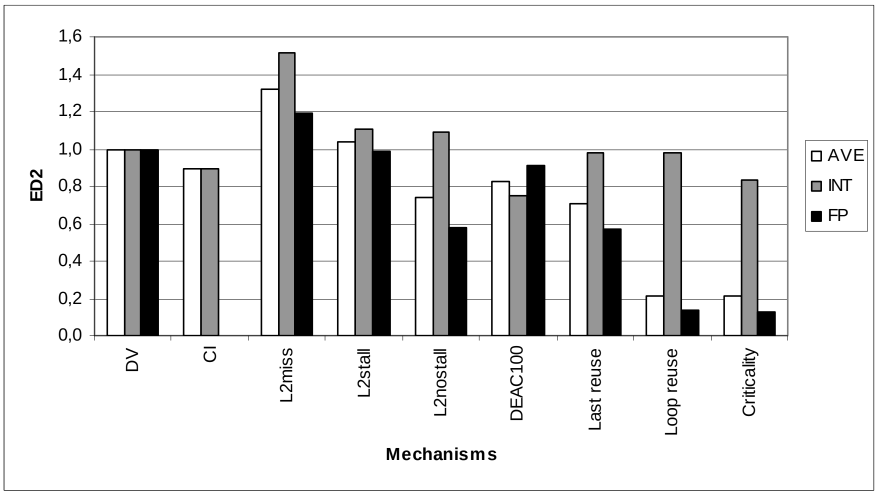

Figure 7.50: Energy-delay^2 comparison of the mechanisms

Only two schemes, L2miss and L2stall, are less energy-effective than the DV mechanism. The reason is that, for the first one, the obtained IPC is not high enough to justify the excessive amount of extra instructions created. The L2miss scheme creates 82 millions of instructions to obtain 1,1 IPC, compared against 1,4 IPC of the basic DV scheme. On the other hand, the L2stall mechanism, even if nearly creates extra instructions (4,9M of extra instructions), only obtains 0,92 IPC compared to 1,4 of the basic DV mechanism.

The good news is that the rest of mechanisms are more energy-delay^2 effective than the basic DV mechanism. As shown in Figure 7.50, Loop reuse and Criticality are the two schemes that are more energy-aware at replication. mainly due to SpecFP2K. Vectorization in regular codes is very effective since most data precomputed by replicas is validated. Moreover, when heuristic-based schemes that exploit the DLP are applied, latency (CPI) reduction makes instructions allocate resources for shorter periods of time then, reducing energy consumption.

On the other hand, energy-delay^2 is not substantially reduced for SpecINT2K. Since aggressiveness is the best way to improve performance for those benchmarks, and excessive amount of extra instructions is created. Fortunately, in this case, these extra instructions are fully justified. For example, the DAEC100 scheme reduces nearly one half the number of extra instructions, losing only 5% of IPC. This means that vectorization is more energy-efficiency. Instructions are better "vectorized".

7.3 SUMMARY

In this Chapter we have motivated the fact that a given mechanism, in this case Dynamic Vectorization, can be applied in a wide range of scenarios. Furthermore, for every scenario, some constrains must be had into account.

For this reason, we present a complete comparison of the mechanisms described in this thesis focusing in three important parameters: hardware requirements, performance and Energy-Delay^2.

The first parameter, hardware, is useful to decide if a mechanism is or not implementable. Even if millions of transistors are available, the size of structures cannot be infinite.

The second parameter ranks the mechanism for high-end computers where performance is the main objective.

Finally, we proportionate a brief description of the third parameter: ED^2. We justify that the creation of extra instructions is not for free. Misspeculated replicas consume energy to perform useless work. The ED^2 measures how effective, in terms of energy-consumption, the execution of an instruction is.

As previously said in the introduction of the Chapter, since scenarios vary from one to another, a global ranking of mechanisms has no sense.

Table 7.8 summarizes this chapter by showing the two best mechanisms for a given previously discussed requirement.

Requirement	**Scheme**
Hardware	Basic DV mechanism
	DAEC100
Performance	Last reuse
	Loop reuse
ED^2	Loop reuse
	Criticality

Table 7.8: Final summary of mechanisms

Chapter 8

Conclusions

8.1 FUNDAMENTALS OF THIS WORK

8.1.1 Background to this thesis

Vector processors have been shown to be very effective in exploiting the DLP present in regular codes. Vectorizing compilers are very efficient in detecting this DLP and passing this semantic information on to the processor through a subset of vector instructions of the ISA. A skilled programmer who has an in-depth knowledge of the underlying architecture can write code that clearly exposes the DLP of a regular application.

However, in irregular applications (integer codes) this DLP exploitation is very difficult. Compilers fail to detect DLP in this kind of application and avoid creating vector code. The main reason is that a compiler does not have enough knowledge of the behavior of an application to apply modifications that are not safe for all scenarios.

This problem is the motivation for this thesis. We wish to *detect the vector patterns of integer codes to clearly expose the DLP*. To accomplish this, since this detection has not been successful at compile time, we *develop a mechanism that detects DLP at runtime and creates vector code that executes in the vector resources* available in the processor. Furthermore, *this mechanism will be rearranged to aim at the most performance penalizing problems* in current superscalar processors: branch mispredictions and the memory gap.

8.1.2 Objectives of this thesis

The main objective of this thesis is to develop a mechanism that is able to detect and expose DLP by detecting vector patterns at runtime for both regular and irregular codes. With these patterns, the mechanism will speculatively create vector instructions that will be executed in the vector functional units of the processor, in order to prefetch and precompute data for their scalar counterparts. A second version of the mechanism will vectorize instructions in processors without vector capabilites.

Along this thesis, we realized that this mechanism can easily be tailored to several problems. As a result of studying the sources of performance improvement in the basic design, two new objectives emerged. The first one consists in alleviating the branch misprediction problem by selecting control-flow independent instructions to be vectorized. From the point of view of the dynamic vectorization mechanism this selection is beneficial since control-independent instructions are the instructions with the highest probability of reusing precomputed data which improves the resource management of the speculative dynamic vectorization mechanism.

The second objective tries to reduce the memory gap by exploiting the stall cycles of the processor caused by L2 miss loads, the inorder nature of the commit stage and the lack of entries in the reorder buffer. During these stall cycles the speculative dynamic vectorization mechanism prefetches and precomputes data for instructions out of the instruction window. This procedure enlarges the instruction window virtually.

Finally, the speculative dynamic vectorization mechanism must be resource-aware. By imposing this requirement, we ensure that the scheme can be implemented in current processors. For this reason, vector instruction creation has to be controlled by selecting the vectorizable instructions that have the greatest impact on performance.

8.2 ACCOMPLISHMENTS

8.2.1 Main contributions

The main contribution of this thesis lies in the claim that exploitable DLP exists, that it is only detectable at runtime (even in irregular codes) and that this parallelism can be used to translate the scalar code into an equivalent vectorial. This study has led us to create the *Speculative Dynamic Vectorization mechanism* (DV), which is able to create vector instructions from a fully optimized irregular code.

Furthermore, on the basis of in-depth study of the sources of performance improvement in the DV mechanism, new versions are derived. The first one deals with the branch misprediction problem in superscalar processors. Several instructions are completely independent to the control flow, and they compute the same results whatever the outcome of previous conditional branches is. Under branch mispredictions, these control-independent instructions are executed once and then again due to pipeline flushes to recover the state of the processor in the mispredicted branches. In this thesis we show that exploiting these instructions alleviates the penalty caused by branch mispredictions. We adapted the basic DV mechanism to focus on these instructions. Since one of the benefits of DV is the virtual enlargement of the instruction window, we claim that we have designed the first mechanism that is able to reuse data for control-flow independent instructions, even when they are out of the instruction window.

Taking advantage of the fact that speculative vector instructions do not occupy entries of the reorder buffer, we designed a set of mechanisms that are able to exploit he stall cycles of the processor due to L2 miss memory instructions. These mechanisms speculatively execute instructions during the cycles in which the processor is stalled, due to the fact that no entries in the ROB are available because this structure is completely full. With these mechanisms we demonstrate that the virtual enlargement of the instruction window is beneficial as previously claimed in the literature. Much more importantly, however, we claim that it is advisable to reuse data computed by speculative instructions during these cycles in order to further alleviate the memory gap.

Finally, based on the fact that the dynamic vectorization mechanism is register-hungry, we propose a set of heuristics to alleviate this problem. Therefore, we refine the basic DV mechanism to improve its benefits at a low cost. As a result, this gives a more powerful mechanism that further improves the basic DV scheme usign a negligible amount of resources.

8.2.2 Detailed breakdown of the contributions

Speculative dynamic vectorization

Chapter 3 demostrates an important fact: DLP is present in all kinds of codes. Even if a compiler fails to detect this parallelism, vector patterns exist in irregular codes and it can easily be detected at runtime. In fact, our studies show that nearly 30% of an irregular program can be vectorized using our mechanism.

In this chapter we present the basics of the Dynamic Vectorization mechanism. Two implementations of the mechanism are described. The first relies on the existence of vector resources to execute the speculatively created data. Speculative vector instructions are created as soon as a vector pattern, which is represented as a strided load, is detected at runtime. These speculative vector instructions prefetch and precompute data for their scalar counterparts. After this, instructions with precomputed data become validation instructions that check whether that precomputation has been performed correctly.

An evolved version of the mechanism is derived from the vectorial design due to the complexity of the execution engine that deals with both scalar and vector instructions. For this design, vector resources were removed from the processor.

Four main sources of performance improvement were detected: efficient management of the memory system, successful reuse of precomputed data, virtual enlargement of the instruction window and control-flow independence reuse of data.

Our studies show the benefits of the DV scheme. The exacerbation of the DLP, in conjunction with an optimal design of the L1 data cache port, leads to a net reduction of the number of accesses to the L1 data cache of nearly 40%. Furthermore, 24% of the scalar instructions become validation instructions, which means that they are not executed. If we add the control-flow independence reuse and the benefits of enlarging the instruction window, the performance is improved by nearly 40% on average for Spec2K, for configurations with long memory latencies.

Control-flow independence reuse via dynamic vectorization

Control-flow independence reuse is the first rearrangement of the basic Dynamic Vectorization mechanism. This selection scheme is based on the fact that vectorizing control-independent instructions only improves the effectiveness of the underlying replication mechanism since these instructions present a high rate of precomputed data reuse.

We have also shown that this scheme improves the Dynamic Vectorization mechanism which alleviates the pressure on the register file since less speculative work is created. Furthermore, we have shown that including the hierarchical register file is not only worth to reduce the design complexity but also to further alleviate the pressure on the scalar registers since replicas hold their results in the upper level of this register file, making the work of the critical lower level lighter.

Therefore, with a moderate amount of hardware, the control independence reuse scheme is able to outperform the superscalar baseline by about 17% (for our research we reduced the memory latency to emphasize the benefits of precomputation). Moreover, with a reduced number of registers distributed in a simple two-level hierarchical register file layout, this scheme performs nearly equally to the basic DV mechanism with an unbounded monolithic register file.

Furthermore, we have also shown that this is the first mechanism that is able to perform control-independence reuse out of the instruction window and that this reuse is very beneficial. Compared to other state-of-the-art schemes that are limited to ROB boundaries, such as Chou´s [CFS99] or Cher´s [CV01] mechanisms our mechanism obtains performance improvements of nearly 14%.

Overcoming the memory gap

To alleviate the memory gap, we presented two mechanisms whose main characteristic is the virtual enlargement of the instruction window.

The first mechanism, *L2miss* which is based on the control-independence scheme presented in Chapter 5, selects replicable instructions following an L2 miss load. The second mechanism, *L2stall*, uses the third version of the Dynamic Vectorization mechanism. In this case, the basics mechanism is implemented as a separate engine that communicates with the main core through the issue queue and the SRSMT table. This second mechanism analyzes the status of the reorder buffer under an L2 miss in order to replicate the mainstream.

We showed that both mechanisms outperform the baseline processor by taking advantage of the base sources of performance improvement of the basic Dynamic Vectorization mechanism.

Furthermore, we compared the mechanisms proposed with each other to show that a net reduction of resources is possible maintaining the performance. The *L2stall* mechanism is more cost-effective than the *L2miss* mechanism since the latter creates a negligible amount of extra instructions (13M and 82,7M of replicas for the L2nostall and the L2miss mechanisms respectively).

Finally, we demonstrated that the virtual enlargement of the instruction window is beneficial. Firstly, with small reorder buffers, the mechanisms presented achieve almost the same performance levels compared to processors with larger reorder buffers. Speculatively prefetching and precomputating data for instructions out of the instruction window overcomes the limitations of the reorder buffer meaning that a processor with 64 entries in the reorder buffer only loses 1% of IPC compared to a processor with 256 entries in the reorder buffer.

Secondly, reusing data precomputed for instructions out of the instruction window is beneficial. For this case, we used processor with Run-ahead as a baseline. We have shown that this effective reuse increases the performance by about 9%.

Cost-effective dynamic vectorization

In this chapter we describe a set of heuristics that is applied directly to the dynamic vectorization mechanism. We show the need for reducing the number of extra mispredicted instructions without penalizing the performance improvement of the mechanism due to a question of resource availability.

We present several families of heuristics divided according to their scope in *Fine grain* and *Coarse grain* heuristics depending on whether they affect just one instruction or a whole loop body.

Furthermore, we add a third heuristic based on *Criticality*. We define the concept of a critical instruction tailored to the Dynamic Vectorization mechanism.

The figures show that our heuristics effectively reduce the number of mispredicted extra instructions and that this reduction, in most cases, increases the speed-up of the basic Dynamic Vectorization scheme.

Finally, we show that the concept of criticality is positive since the Criticality heuristic improves the performance considerably, creating 64% less speculative instructions and validating only 12 millions instructions (50% less than the Dynamic Vectorization mechanism).

8.3 REMARKS AND FUTURE WORK

After the in-depth analysis of our research on different versions of the Dynamic Vectorization mechanism, new requirements emerge:

- **Simplification of the DV mechanism:** From a hardware logic point of view, some parts of the basic mechanism should be simplified, particularly the memory disambiguation algorithm. Every store committed must check its effective address against all the ranges of addresses of each vectorized load. If this address is inside any range, the resources of these loads are deallocated. Since these checks are costly, store commit requires extra latency. Rearranging this procedure would prevent stores from penalizing performance.

- **New register allocation policy:** More intelligent register allocation policies for replicas need to be developed. This will alleviate the pressure on the register file, since it is probable that fewer extra speculative instructions will be created. Policies regarding the code structures seem to be a good alternative.

- **Improve heuristics by gathering information of loops:** Section 6.4.2, the block information heuristic, motivates a new line of research. If the semantic information of the structure of a loop can be tracked at runtime, the effectiveness of vectorization can be improved by successive refinements of the speculatively created code, which will reduce resource requirements and result in better performance.

- **Vector code runtime optimizations:** After the vector instructions are created, several typical vector code optimizations could be applied, which would produce a better speculative code. A further step could be consider binary translation of scalar code at runtime, and even allow the created vector instruction to be executed in a non-speculative fashion.

- **Selective dynamic vectorization targeted to improve branch prediction:** As previously shown in this dissertation, the basic DV mechanism can easily be modified to aim any problem in current processors. In Chapter 4, a possible solution, based on control independence, for alleviating the branch misprediction penalty is presented. From another point of view, the DV scheme could be tailored to aim hard-to-predict branches towards improving their ratio of predictions by preexecuting the instructions leading to that branch.

- **Control-flow independence reuse for large instruction windows:** Concerning this point, we plan to apply the basic CI mechanism to large instruction window processors. In these processors the branch mispredictions penalize the performance dramatically, since a large number of instructions have to be squashed. From another point of view, the CI scheme can further enlarge the instruction window in these processors.

LIST OF TABLES

LIST OF FIGURES

REFERENCES

[AK87] J. R. Allen and K. Kennedy. Automatic translation of fortran programs to vector forms. In *ACM Transactions on Programming Languages and Systems*, 1987.

[AMD99] AMD. http://www.amd.com, 3dnow! technology manual. In *Technical Report*, 1999.

[Asa98] Krste Asanovic. Vector microprocessors. In *PhD Degree Dissertation, University of California at Berkeley*, 1998.

[BA97] D. Burger and T. Austin. The simplescalar tool set, version 2.0. In *Technical Report n. CS-TR-97-1342, University of Wisconsin-Madison*, 1997.

[BC91] J. Baer and T. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of International Conference on Supercomputing*, 1991.

[BDA01a] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically allocating processor resources between nearby and distant parallelism. In *Proceedings of the 28th International Symposium on Computer Architecture*, 2001.

[BDA01b] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of the International Conference on Microarchitecture*, 2001.

[BGS93] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high performance computing. In *Technical Report No. UCB/CSD-93-781, University of California at Berkeley*, 1993.

[Bor99] S. Borkar. Design challenges of technology scaling. In *IEEE Micro Volume 19, issue 4, pp 23-29*, 1999.

[CFS99] Y. Chou, J. Fung, and J. P. Shen. Reducing branch misprediction penalties via dynamic control independence detection. In *Proceedings of the 13th International Conference on Supercomputing*, 1999.

[CGVT00] J. L. Cruz, A. González, M. Valero, and N. Topham. Multiple-banked register file architectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[CHP97] P. Chang, E. Hao, and Y. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[CKP91] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[CL99] G. Cai and C. H. Lim. Architectural level power/performance optimization and dynamic power estimation. In *Cool Chips Tutorial in conjunction with MICRO 32*, 1999.

[COM$^+$04] A. Cristal, D. Ortega, J. F. Martínez, J. Llosa, and M. Valero. Out-of-order commit processors. In *Proceedings of the 10th International Symposium on High Performance Computer Architecure*, 2004.

[CSK$^+$99] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the 26th International Symposium on Computer Architecture*, 1999.

[CTWS01] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th International Symposium on Microarchitecture*, 2001.

[CV01] C. Cher and T. N. Vijaykumar. Skipper: A microarchitecture for exploiting control-flow independence. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, 2001.

[DM97] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the International Conference on Supercomputing*, 1997.

[ea95] Krste Asanovic et al. The t0 vector microprocessor. In *Hot Chips, Volume VII:187-196*, 1995.

[ea05] D. Pham et al. The design and implementation of a first-generation cell processor. In *Proceedings of the International Solid-State Circuits Conference*, 2005.

[Esp97] Roger Espasa. Advanced vector architectures. In *PhD Degree Dissertation, Universitat Politècnica de Catalunya*, 1997.

[FRB01] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th International Symposium on Computer Architecture*, 2001.

[GG97] J. González and A. González. Memory address prediction for data speculation. In *Proceedings of Europar*, 1997.

[GH96] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. In *IEEE Journal of Solid-State Circuits 31, N. 9*, 1996.

[GKMP98] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence estimation for speculation control. In *Proceedings of the 25th International Symposion on Computer Architecture*, 1998.

[Gon00] José González. Speculative execution through value prediction. In *PhD Degree Dissertation, Universitat Politècnica de Catalunya*, 2000.

[Int99] Intel. Pentium iii processor: Developer's manual. In *Technical Report*, 1999.

[Int02] Intel. Pentium ii processor - datasheets. In *Technical Report*, 2002.

[JG97] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[JLW01] R. D. Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. criticality. In *Proceedings of the 28th International Symposium on Computer Architecture*, 2001.

[Jou90] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, 1990.

[JRS96] E. Jacobsen, E. Rotenberg, and J. E. Smith. Limits of control-flow on parallelism. In *Proceedings of the 29th International Symposium on Microarchitecture*, 1996.

[Ken78] K. Kennedy. A survey of compiler optimizations techniques. In *Le Point sur la Compilation, (M. Amirchahy and N. Neel editors), INRIA, Le Chesnay, France*, 1978.

[LD97] Corinna G. Lee and Derek J. DeVries. Initial results on the performance and cost of vector microprocessors. In *Proceedings of the 13th International Symposium on Microarchitecture*, 1997.

[LKL$^+$02] R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

[LLVA98] D. López, J. Llosa, M. Valero, and E. Ayguadé. Widening resources: A cost-effective technique for aggressive ilp architectures. In *Proceedings of the 31st International Symposium on Microarchitecture*, 1998.

[LM96] C. K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[MKG98] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[MKSP05] O. Mutlu, H. Kim, J. Stark, and Y. N. Patt. On reusing the results of pre-executed instructions in a runahead execution. In *IEEE Computer Architecture Letters, vol. 4*, 2005.

[MLG92] C. T. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[MRH$^+$02] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th International Symposium on Microarchitecture*, 2002.

[MSWP03] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Run-ahead execution: An alternative to very large instructions windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, 2003.

[PGV02] A. Pajuelo, A. González, and M. Valero. Speculative dynamic vectorization. In *Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

[PGV04] A. Pajuelo, A. González, and M. Valero. Speculative execution for hiding memory latency. In *Workshop on MEmory performance, DEaling with Applications, Systems and Architectures*, 2004.

[PGV05a] A. Pajuelo, A. González, and M. Valero. Control-flow independence reuse via dynamic vectorization. In *Proceedings of the 19th IEEE International Parallel & Distributed Processing Symposium*, 2005.

[PGV05b] A. Pajuelo, A. González, and M. Valero. Cost-effective dynamic vectorization. In *Technical Report No. UPC-DAC-RR-2005-15*, 2005.

[PGV05c] A. Pajuelo, A. González, and M. Valero. Speculative execution for hiding memory latency. In *ACM SIGARCH Computer Architecture News, Volume 33, Issue 1*, 2005.

[PJS97] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity effective superscalar processors. In *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[RJS99] E. Rotenberg, Q. Jacobson, and J. Smith. A study of control independence in superscalar processors. In *Proceedings of the 5th International Symposium on High Performance Computing Architecture*, 1999.

[RS99] E. Rotenberg and J. Smith. A study of control independence in trace processors. In *Proceedings of 32nd Symposium on Microarchitecture*, 1999.

[RTDA97] J. A. Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin. On high-bandwidth data cache design for multi-issue processors. In *Proceedings of the 30th Symposium on Microarchitecture*, 1997.

[Rus78] R. M. Russell. The cray-i computer system. In *Communications of the ACM, 21(1) pp 63-72*, 1978.

[SC00] T. Sherwood and B. Calder. Loop termination prediction. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, 2000.

[SJ01] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model. In *Technical Report, Compaq Computer Corporation*, 2001.

[Spe00] Spec2000. Spec2000 benchmark suite. In *http://www.specbench.org/osg/cpu2000*, 2000.

[SS97] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[TG98] J. Tubella and A. González. Control speculation in multithread processors through dynamic loop prediction. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture*, 1998.

[Uht92] A. K. Uht. Concurrency extraction via hardware methods executing the statin instruction stream. In *IEEE Transactions on Computers, vol. 41*, 1992.

[VJM99] Sriram Vajapeyam, P. J. Joseph, and Tulika Mitra. Dynamic vectorization: a mechanism for exploiting far-flung ilp in ordinary programs. In *Proceedings of the 26th annual International Symposium on Computer Architecture*, 1999.

[WD94a] S. W. White and S. Dhawan. Power 2. In *IBM Journal of Research and Development, v. 38 n. 5, pp 493-502*, 1994.

[WD94b] S. W. White and S. Dhawan. Power2. In *IBM Journal of Research and Development, v.38, n. 5*, 1994.

[WM95] W. A. Wulf and S. A. Mckee. Hitting the memory wall: Implications of the obvious. In *ACM SIGARCH Computer Architecture News, vol. 23, n. 1*, 1995.

[WO01] K. M. Wilson and K. Olukotun. High bandwidth on-chip cache design. In *IEEE Transactions on Computers, vol. 50, no. 4*, 2001.

[ZC90] H. P. Zima and B. Chapman. Supercompilers for parallel and vector processors. In *ACM Press Frontier Series/Addison-Wesley*, 1990.

[ZS01] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 34th International Symposium on Microarchitecture*, 2001.