Master’s thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Marco Caccialino

Eliminating Unnecessary Broadcasts to Simplify Out-of-Order Instruction Scheduling

Master’s thesis in Faculty of Information Technology and Electrical Engineering
Supervisor: Rakesh Kumar
June 2020

NTNU
Norwegian University of
Science and Technology

Marco Caccialino

Eliminating Unnecessary Broadcasts to Simplify Out-of-Order Instruction Scheduling

Master's thesis in Faculty of Information Technology and Electrical Engineering
Supervisor: Rakesh Kumar
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Acknowledgement

I would like to thank my tutor Rakesh Kumar who spent his time and knowledge to guide me through this project and NTNU for having given me the opportunity to complete my thesis.

Summary

Over the past decades the industry has been pushing to find ways to achieve better performance and efficiency. The natural evolution of microarchitecture has introduced first the pipeline and then out-of-order superscalar processors to achieve a greater Instruction Level Parallelism. This newer iteration have, on the other hand, increased the complexity of the pipeline stage such as the issue stage.

This thesis focus on the above mentioned pipeline stage, analysing which are the aspect that add complexity and energy inefficiency. Using Sniper simulator, an x86 architecture will be used to track the usage of the wake up signal in order to find the instructions that really need to broadcast a signal. Lastly a simple hardware addition will be investigated as a possibility to reduce the broadcast width and simplify the wake up logic.

Table of Contents

List of Tables

List of Figures

Glossary

DNB Delay-and-Bypass. 24

FIFO First In First Out queue. v, 18, 19, 24, 25

FIFOrder FIFOrder MicroArchitecture. 24

FXA Front-end Execution Architecture. 24

IBDA Iterative Backward Dependency Analysis. 17

ILP Instruction level parallelism. 7, 24

IPC Instruction per Cycle. 23

IQ Instruction Queue. 24

IST Instruction Slice Table. 17

IXU In-order Execution Unit. 24

LTP Long Term Parking. 24

MLP Memory-Level Parallelism. 21, 23–25

MLP-Aware MLP-Aware Dynamic Instruction Window Resizing. 23

OXU Out-of-order Execution Unit. 24

RDT Register dependency table. 17

Chapter 1

Introduction

The word of computing devices has had since the beginning an insatiable demand for better performance year over year. Moore's law, since the 1960s, has predicted the growth of the number of transistors in computer chips that would lead to increasing performances at a lower cost.

Another correlated positive trend is expressed by Dennard's law, which describes how the power density in a chip stays constant since the growing density of transistor is compensated by the lower power they require to function. This abundance of resources brought to the introduction of new architecture like super-scalar out-of-order processor that could expose an higher Instruction Level Parallelism and therefore greater performances.

Around a decade ago the above-mentioned models stopped being valid due to technical limitation and caused designers to focus on other ways to reach an increase of performance. The whole industry has also been subject to a shift of the attention towards the energy efficiency of chips due to the growing number of small and portable device. This new environment is demanding for architecture that are more aware of the area and power consumption. This phenomenon eventually brought designer to even prefer simpler in-order scalar architecture respect to the more complex ones, scarifying performance in favor of energy efficiency.

Another way of facing this matter is, instead, to simplify and optimize the superscalar out-of-order architecture and in particular their most energy expensive component : the issue queue, which can make up 18-40% [3] of the whole chip consumption. The high complexity is due mainly to the wake up logic, to tell the entries their components are ready, and to the select logic which sends to execution the ready instruction.

A significant amount of work has been already done in order to simplify the issue stage, but the main focus so far has always been the select stage. Many proposal take into consideration the simplification of the out-of-order architecture [8][11][2][3] while others take the opposite approach, making the in-order more performing [4] [9].

In this thesis, instead, the major focus will be the wake up logic that none of the cited works have faced and therefore offers great possibility for improvements. In this project we will, in order, analyze the usage of the wake up, reduce the instruction that need to use this logic and imagine a simple additional hardware structure to further shrink the broadcasts needed.

1.1 Thesis goal

The goal of this thesis is to analyze and reduce the complexity of instruction wake-up logic in superscalar out-of-order processors. All the instructions finishing their execution need to broadcast wakeup signals (i.e. generally, their destination register IDs) so that their dependents in the issue queue can be waken up. Therefore, the number of wake-up signals need to be broadcasted depends on the maximum number of instructions that can finish execution in the same cycle. The goal of this thesis is to minimize the number of signal that need to be broadcasted per cycle, thereby reducing the number of comparisons required per issue queue entry and their associated area and energy cost.

Towards this end, we first study the distribution of number of instructions finishing execution in same cycle. We use this distribution to identify the minimal broadcast width (i.e. maximum number of wakeup signals broadcasted per cycle) with minimal impact of performance. To further reduce the broadcast width, we make a critical observation that an instruction does not need to broadcast its results if none of the instructions in issue queue depends on it. By avoiding to broadcast wakeup signals for such instructions we can reduce the broadcast width. To reduce the broadcast width even further, our key insight is that we can delay the broadcast of a wakeup signal if all the instructions needing this signal are the non-critical ones. Prior work has shown that not all instructions contribute to performance equally. Therefore, delaying the wakeup of non-critical instruction has the potential to reduce broadcast with a minimal performance penalty.

1.1.1 Requirements

- R1 *Gather information on broadcast usage*
- R2 *Study the instruction and their relation with the wake up signal*
- R3 *Develop a strategy to reduce complexity*

1.1.2 Contribution

- An overview on the use of wake up signal by instructions
- An evaluation on the necessary broadcast width
- A microarchitecture which exploits instruction criticality to reduce the broadcast width

Chapter 2

Background

This chapter presents the necessary background in how a processor works, forming the basis for the following research.

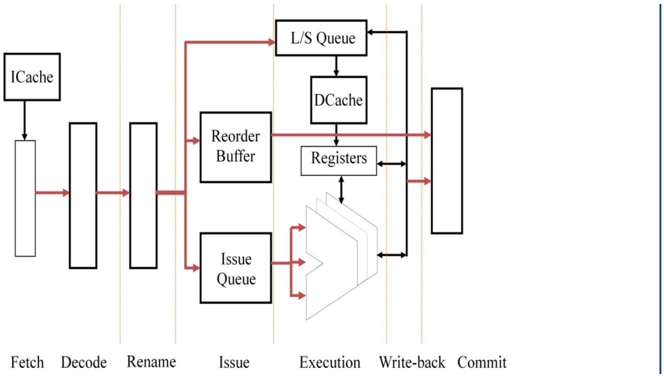

Figure 2.1: A general scheme of a pipeline design and major stages. Picture from [7]

2.1 Pipeline

The foundation of modern processor architecture is the pipeline which splits the execution of an instruction in multiple smaller stages that run independently at a higher frequency. This makes possible to execute many instruction at the same time in different stages, exposing what is defined as Instruction Level Parallelism. The processor therefore achieves an higher throughput because the smaller stages have a smaller execution time that allows the processor to run at an higher frequency. Since the architecture works as an assembly line each cycle it can deliver a complete instruction meaning that the total throughput is much higher.

One typical design of a pipeline is showed in Figure 2.1 and its main stages are:

- Fetch
- Decode
- Rename
- Execute
- Write-back
- Commit

Ideally the more the pipeline is split in smaller and numerous stages, higher is the frequency it could operate on; however, this is not entirely true because the time for each step to complete is anchored to the slowest of the stages.

Another factor that can affect the performance of the pipeline is the dependencies in between the instructions, for example: if I2 needs the data produced by I1 then the former needs to wait until the source instruction is completed and this could cause a stall.

2.1.1 Fetch and Decode

Fetch is the first stage of the pipeline and it's the one responsible of inserting the instruction into the processor. This stage computes the address of the next instruction and then it accesses the instruction cache to fetch it; this whole process for one instruction is usually completed in one cycle but it can be affected by branches which, due to their nature, prevent the next instruction address to be calculated in parallel (unless a speculative mechanism is introduced).

Decode is the stage which is in charge of interpreting the instruction previously fetched, so to understand what type of operation it describes, which execution unit it needs, which source operands it depends on as well as in which register the result will be written to.

2.1.2 Rename Stage

Strictly related to the decode is the rename stage. It is used to allocate physical registers to the decoded instruction that are not necessarily the one encoded in the instruction. It can happen that multiple instructions are meant to use the same architectural register even if the data meant to be written will not be shared, this creates a false dependency which is called *"Name dependency"*. The renaming logic solves this problem: it allocates the above-mentioned instruction in different physical registers and it keeps track of this allocation in dedicated tables [14]. In a superscalar (see subsection 2.2.1) processor there is the necessity to rename multiple instructions in the same cycle: to find the dependencies between them the source register of each entry is compared to the destination of the others. If an instruction's parent is in its group, the identifier of the physical register allocated to the parent overrides the identifier obtained from the rename map [14]. This logic is useful when many dependencies are exposed, therefore it finds a better use in out-of-order architecture rather than in-order.

2.1.3 Issue stage

The issue stage has two main purpose: 1) it is responsible for choosing which instruction will be sent to the Functional Units for execution and 2) it keeps track of the readiness of the source operands and wakes up instructions when all their source operands are ready. It represents one of the most complex and energy consuming components, especially in out-of-order architecture. As issue stage is the focus of this thesis, it will be described more in details in 2.2

2.1.4 Execute and write back

This is the stage at which the results are actually computed. The instruction, as we know, can be of different types and therefore need different functional unit for their execution: for example an integer operation needs different components compared to a memory operation. In modern processors, there are implemented different execution paths for integer, memory, floating point and branch instructions which also have different execution latencies.

Once the operation is completed the results are immediately available but they should then be written back to the register files before they can be used by the dependant instruction and this would cause a loss of 1 or more cycles. In modern processors, in order to maximize performance, the result can also be *forwarded* as soon as it is available after the execution stage through the *bypass network*

2.1.5 Commit

The commit is the last stage of a pipeline and it is in charge of actuating the modification derived from the execution of the instruction in previous stages. In modern architecture it is important to keep feeding the processor with new instruction to be executed even if it is not sure that their result will be used due to branches and exceptions. Therefore there is a difference between architectural state, which is the correct flow of execution, and speculative state which is the current flow of instruction that are being executed and will be committed into architectural state only in the commit stage.

In an out-of-order processor this is also the place where the instruction are finally put back in order before being written in memory and the resources in the Reorder Buffer and other structures are reclaimed.

2.2 Issue Stage Details

As this project aims to simplify instruction wake up mechanism, we discuss the issue stage in details.

2.2.1 Issue order

There are two main approaches to issue instructions in a processor: in-order and out-of-order. The former is more classic approach where the instructions are issued in the same order as they are fetched and it results in a simple hardware implementation. However, it is limited in its ability to extract Instruction level parallelism (ILP). With out-of-order issue, the instruction are executed as soon as they are ready, meaning as soon as their source operands are produced. Once they are executed they are then put back in order in the commit stage. While this solution is beneficial to the speed of the processor, it also results in a very complex implementation and the issue stage can become the critical path, limiting the clock frequency in the pipeline.

The issue stage also represents one of the most energy consuming component, the issue queue alone can count up for 18% [11] of the total energy in a processor due to a writing activity which is high energy demanding.

2.2.2 Scalar/Superscalar processor

Another major difference in processor microarchitectures is the issue width, in other words the number of instruction issued per cycle. A Scalar processor is able to issue only one instruction per cycle and work only on one piece of data at the time. The Superscalar processor, instead, can issue multiple instructions each cycle and work on multiple pieces of data [10] therefore it is able to achieve a throughput higher than one instruction/cycle unlike the former solution.

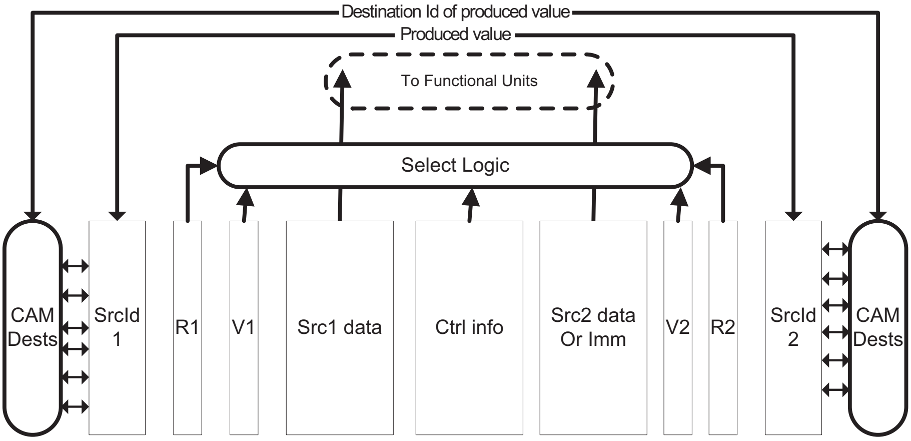

Figure 2.2: Example of Issue stage with operands read before the issue. The image is take from [7]

2.2.3 Reading Operands

An instruction can read its operands either before or after the issue stage:

Reading the operands before the issue stage

Figure 2.2 shows the necessary hardware blocks when the operands are read before the instruction is issued. *Src1Id* and *Src2Id* are the memory blocks used to keep the operand ID (after they have been renamed) of the instruction in the issue queue. Other two symmetric tables *Src1 data* and *Src2 data* are used to hold the values of correspondent operands of the instruction along with immediate value (in *Src2 data* only) when necessary. The validity bits are stored in other two tables *V1* and *V2*. When the operands of an instruction are produced then the latter is marked as ready for issue, marking the corresponding bits in the *R1* and *R2*.

Reading the operands after the issue stage

In case the operands are read after the issue, the hardware structure is simplified: The table *Src1 data* is not needed anymore while *Src2 data* is kept only to be used for immediate operands and is therefore reduced in size. The consequence of reading after is that there is an extra cycle between the instruction being ready and being issued.

On the other hand this design brings benefits such as the reduction of stages between renaming and queue allocation since data are yet to be read; this also causes a reduction of CAM memory needed.

Another big difference is in the number of the read port needed which is also proportional to area, power and access latency of the register file. With this implementation the number of read port is, in fact, linked to the issue width but it can be be also smaller with minimal impact on the performance. Most of operands is also read from the bypass logic making the number of read port needed even smaller.

For this project we will consider this implementation as the target architecture.

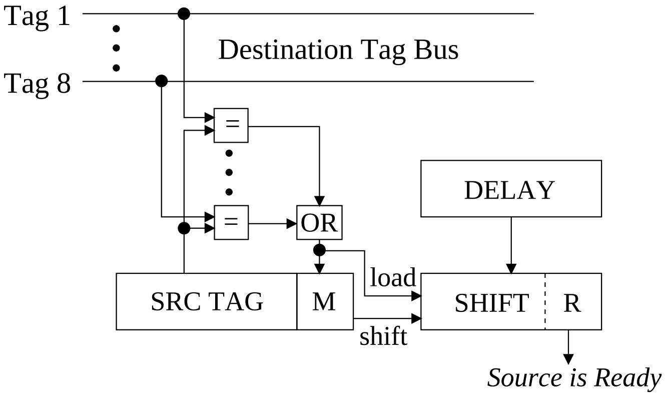

Figure 2.3: Conventional wakeup logic [13]

2.2.4 Wake up signal

Once one of the source operand has been produced a signal is sent to issue queue containing the ID and validity. The CAM logic is used to check that the ID is present in above-mentioned ID table and eventually set the corresponding validity bit.

In Figure 2.3 is reported an example of Wake up logic: the tag logic marks a match when the destination tag of a parent is broadcasted and the correlated bit is set.

Since the wakeup signal is produced only once it must be guaranteed that all the interested consumers get the information even if not in the issue queue, this brings to the use of the CAM memory also in the renaming tables. CAM memory, unfortunately, are known for taking more area and consuming more then a normal addressed memory like SRAM. This means that reducing their usage could certainly benefit the overall energy efficiency. In fact it has been found that the wake up logic represents up to **63%** of the whole issue logic [6] which can be roughly translated to **16%** of the total energy consumption of a processor.

The waking up signal is generated by an instruction when it finishes the execution to inform the dependant operands that they can be ready to execute. As shown in Figure 2.4 the wake up signal can be generated in advance, knowing how long it takes to a specific instruction to complete so to bring an higher overlapping of the pipeline stages. This implementation can be possible if a data bypass, discussed in subsection 2.1.4, is put in place. It allows data from the executed instruction to be used directly by the following, without storing it in the register files.

In addition, the number of comparisons required for each issue queue entry equals the number of source operands times the number of instructions finishing execution each cycle. Though the number of operands remains constant for an ISA, the number of instructions producing results increases with issue width of processor. Thus, the wakeup logic becomes more complex as the issue width increases.

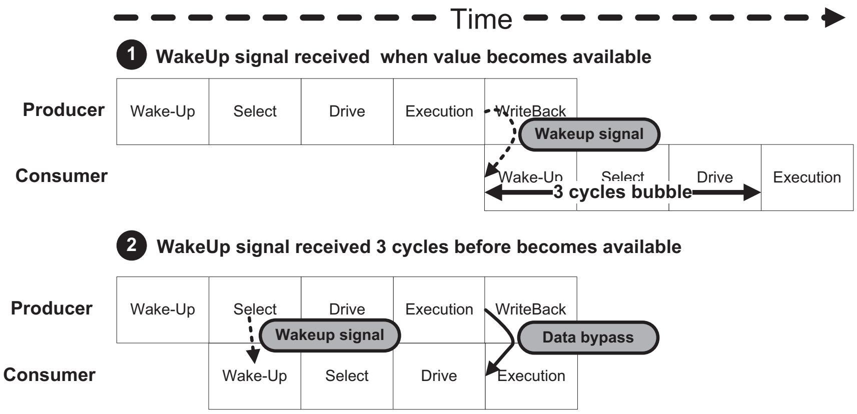

Figure 2.4: Example of the waking signal and the possible bypass implementation [10]

Chapter 3

Opportunity Analysis

This chapter analyzes the opportunity in reducing the broadcast width, i.e. the maximum number of wake-up signals broadcasted in a cycle, by either avoiding or delaying the broadcast of wake-up signals.

3.1 Instruction categories

The instructions can be categorized as follows based on whether or not they need to broadcast a wake-up signal:

- **ARITHMETIC**
- **LOAD**
- **STORE**
- **BRANCH**

An instruction needs to broadcast a wake-up signal, when it finishes execution, to wakeup the dependent instructions in the issue queue. However, if an instruction does not generate a date value, it would not have any dependents and hence it does not need to broadcast a wakeup signal. This is the case for branch instructions. Branch instructions only decide the control flow direction and do not have any data dependent instructions. Therefore, they do not need to broadcast a wakeup signal.

Store instructions write to memory and not to register file. The dependencies through memory are handled by load/store queue and not by the issue queue. Therefore, stores also do not need to broadcast a wakeup signal. Arithmetic and load instructions, on the other hand, do have dependent instructions in the issue queue that wait for their results. Therefore, these two category of instructions do not need to broadcast wake-up signals when they finish execution.

We assume that the baseline core does this distinction among instructions based on whether or not they need to broadcast wakeup signals. Next we show the fraction of instructions that does not need to broadcast wakeup signals in the baseline core.

3.2 Instruction distribution in baseline core

Figure 3.1 shows the dynamic instruction distribution in the above categories. As the figure shows, on average, about 9% of instructions are stores and a further 11% of instructions are branches. Therefore, a total of about 20% of instructions do not need to broadcast the wake-up signal in the baseline architecture. Looking at individual workloads, there are benchmarks like `omnetpp` and `perlbench` where instructions that do not need to broadcast wake-up signals (stores and branches) constitute about 35% of dynamic instruction stream. Therefore, these applications do not require a wide broadcast width.

However, for the majority of applications, the dynamic instruction stream is dominated by instructions requiring to broadcast wake-up signal. For example, in `gamess`, `GemsFDTD`, `gromacs`, `leslie3d`, `sphinx3`, `wrf`, and `zeusmp` nearly 90% of instructions need to broadcast the wakeup signals. Therefore, they all are likely to require wide broadcast width.

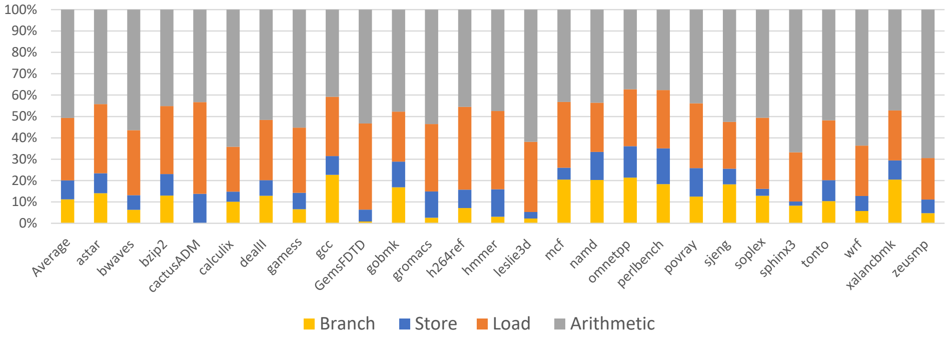

Figure 3.1: This graph shows the usage of the different categories of instruction for each benchmark

3.3 Instructions with no dependents in the issue queue

As noted earlier, our critical observation is that, if an instruction (load or arithmetic) does not have any dependent instruction in the issue queue it does not need to broadcast a wake-up signal on its completion. This is because it has no instruction to wake-up in the issue queue. If a dependent instruction later enters the issue queue, it will already have its operands present in the register file.

If we detect and avoid the wake-up signal broadcast for such instructions, we can further reduce the total number of instructions that need to broadcast wake-up signals. Figure 3.2 shows the fraction of dynamic instructions that have no dependent instructions in the issue queue when they finish execution. Note that the figure includes only loads and arithmetic instructions as stores and branches do not need to broadcast wake-up signal. On average the 7% of the instructions broadcast a wake up signal that does not wake-up any instruction in issue queue. There are some benchmarks like `hmmer` and `sphinx3` where this observation does no affect a significant slice of the instructions. On the other hand many workloads like calculix, wrf, xalancbmk and zeusmp present percentage even higher than 10%, suggesting that this observation could still lead to a meaningful reduction of the broadcast width when applied.

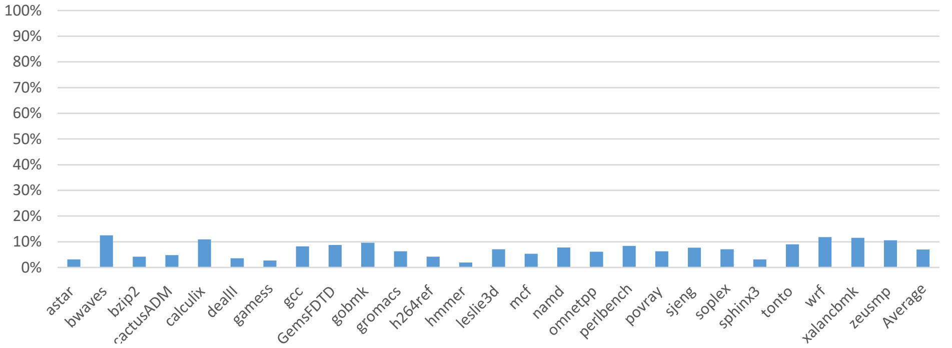

Figure 3.2: This graph shows the percentage of instruction that should not broadcast a signal because they do not have dependent. Store & Branch are not included because they do not need to broadcast a wake up signal

3.4 Instruction criticality to reduce broadcast width

To further reduce the number of instructions requiring to broadcast wake-up signal, we exploit the phenomenon of instruction criticality. Prior work [5, 11, 4] has shown that non all instruction contribute equally to the performance. Delaying the execution of non-critical instructions by some cycles hardly has any impact on performance. Our key idea is to delay the wake-up signal broadcast for non-critical instructions to reduce the broadcast width. Though this will delay their wake-up and execution, it might not have a big performance impact as these instructions are not critical to performance.

We describe the mechanism to identify the non-critical instructions and delay wake-up signal broadcast for them in section 4.2. Here we present the number of additional instructions that do not need to broadcast their wake-up signal immediately, rather when broadcast ports are available.

Figure 3.3 shows the number of instructions that generate results only for non-critical instructions. Hence, their wake-up signal broadcast can be delayed. In figure, for the sake of simplicity, the instructions with no dependent are excluded because already taken into consideration in Figure 3.2. The average number of instruction that can be delayed is around 41%, with conspicuous peaks given by benchmarks like castsADM, gromacs and mcf (79%, 66% and 61% respectively). This results meets our expectation and introduces a further idea to reduce the broadcast width.

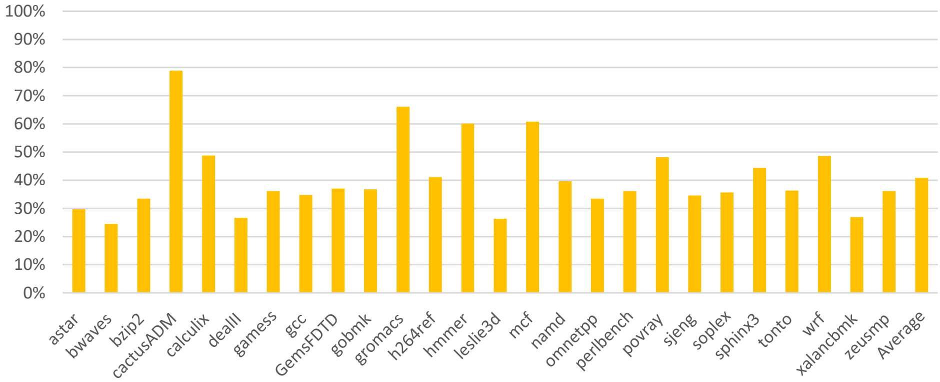

Figure 3.3: Percentage of instruction, out of the total, that are sending a wake up signal to a non Critical dependant. In this figure the percentage of instruction that have no dependent has been excluded to simplify the comparison

3.5 Combining all the results

Having seen the three categories of instructions, as mentioned above, whose wake-up signal broadcast can either be avoided or delayed, now we combine them together and present the overall results in Figure 3.4. On average the amount of instruction that do not need to broadcast wake-up signal immediately or never at all is around 68%, which supports our argument. In most of the workloads this number exceeds 50% while some like `cactusADM` and `mcf` even get above 90%, showing the best predisposition to benefit from a broadcast width reduction while keeping similar performance.

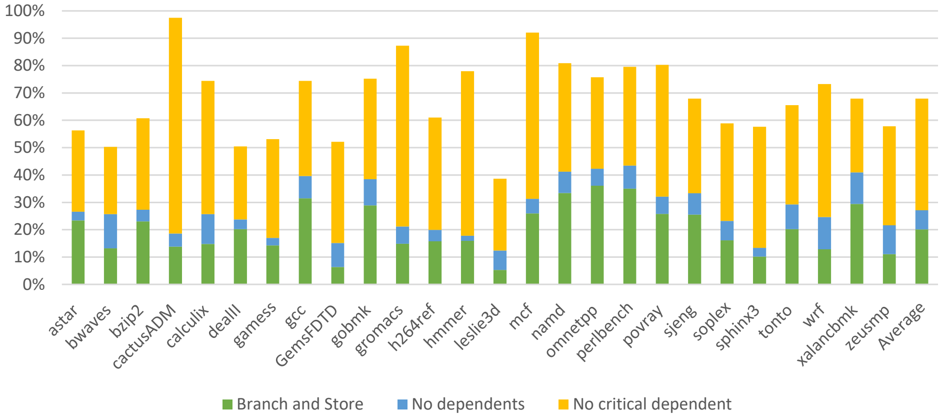

Figure 3.4: In this figure all the percentage of instruction (Store & Branch, No dependant, No critical dependent) discussed in previous point are combined

Chapter 4

Analyzing the broadcast width requirements

This chapter puts into practise the ideas of broadcast width reduction described in chapter 3, analysing the resultant behaviour for each of the previous proposal

4.1 Analyzing the broadcast width requirements

The major focus of this study is to analyze and reduce the broadcast width, which means reduce the number of wake up signal that can be broadcasted in one cycle. As mentioned before, in the baseline core, all the instruction broadcast a wake up signal apart from stores and branches. To understand further developments here we will show the initial situation.

4.1.1 Broadcast width for baseline core

Figure 4.1 shows the distribution of wake-up signals that need to be broadcasted each execution cycle. The benchmarks show very different distributions. In cases like `gcc` or `mcf` the majority of cycles (84% and 90% respectively) does not seem to need any broadcast which is likely due to frequent stalls caused by cache misses.

On the other hand cases like `cactusADM`, `gamess` and `hmmer` point out a far higher pressure on the broadcast logic, in particular the latter workload presents 38% of cycles with 2 wake up signal and 31% with 3 wake up signals. Considering all benchmarks the average of the distribution is fairly uniform with the cases from 0 up to 3 wakeup per cycle representing 90% of the total (40%, 18%, 18% and 14 % are the respective percentage in order from 0 to 3 wake up / cycle).

From the data presented so far we can deduce that more than 4 broadcasts per cycles are rarely needed. In fact, on average, 4 broadcast are used in 7% of the cycles while 5 and 6 represent 2% and 0.2% respectively. This shows that a broadcast width set to 4 would be already capable of covering 98% of the cycles in the baseline core.

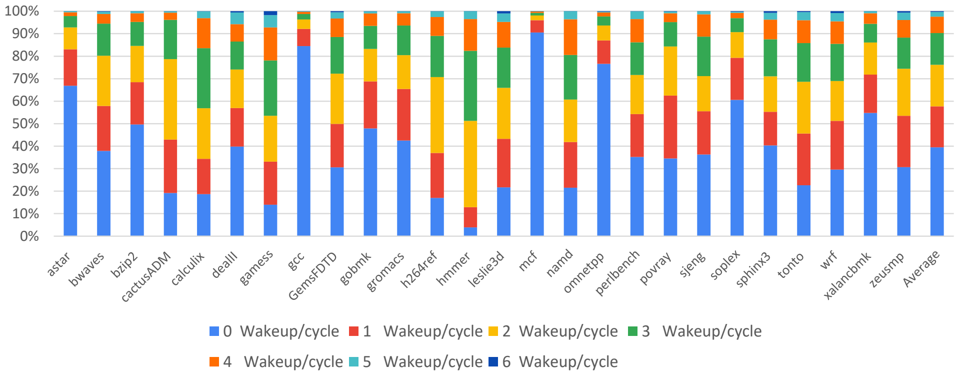

Figure 4.1: Broadcast usage per cycle in the baseline version with issue queue set to 168

4.1.2 Minimizing broadcast width

Given the data illustrated in Figure 4.1 we now explore what is the change in the distribution of broadcast when putting in practise the ideas discussed in chapter 3

Eliminating broadcast for instructions without dependents

In section 3.3 we found that, on average, 7% of the instruction still broadcast a wake up signal even if they do not have any dependent. In this section we explore the impact on the broadcast distribution if the above-mentioned instructions do not use a wake up signal.

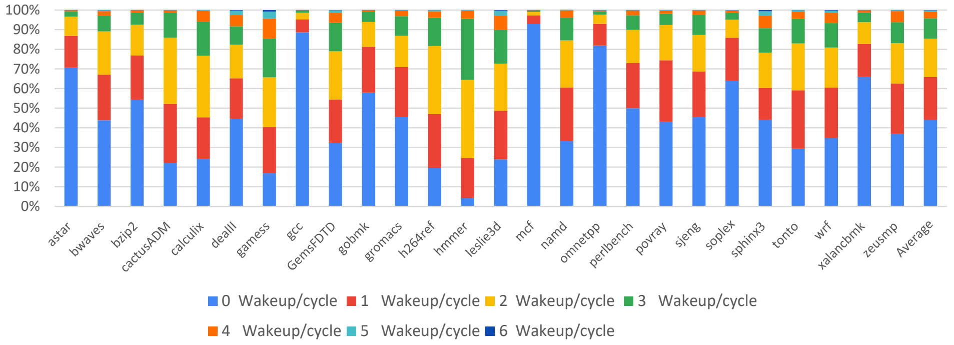

Figure 4.2: Broadcast usage per cycle with issue queue set to 168 when the instruction with no dependent do not send a signal

Figure 4.2 shows the new distribution and the improvement obtained in most of the workloads. We can observe that, as expected, since the number of broadcasts is lower, also the number of wake up signals per cycle should diminish.

The average shows that 96% of the cycles require 3 or less broadcasted signal respect to 90% showed in the previous scenario. It's worth noticing that the individual concentration are 44%, 22%, 19% and 10 % in order from 0 to 3 wake up / cycle, displaying a shift of the concentration toward the lower end.

In line with Figure 3.2 the benchmark `xalancbmk` shows one of the biggest improvement, with an increase of 11% of cycles without broadcast. `Astar`, on the other hand, has a small increase in both 0 and 1 wake up signal per cycles (3.70% and 0.14% respectively) as the Figure 3.2 suggested. Overall 4.2 confirms our prediction and shows a general improvement. In particular it appears that a broadcast width higher than 3 does not affect a great number of instruction and neither brings great benefit, therefore it could be set as a good compromise when considering this setting.

Delaying broadcast for non-critical instructions

In chapter 3 we found that the concept of instruction criticality could be exploited to reduce the number of broadcast per cycle. This is possible because the non critical instruction have a smaller impact on the performance and they can be delayed to reduce the pressure on the broadcast logic.

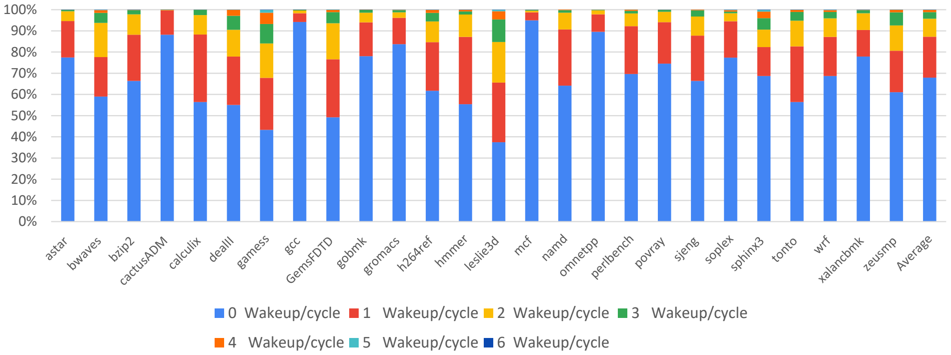

Figure 4.3: Broadcast usage per cycle with issue queue set to 168 when only the signals towards critical instructions are broadcasted

Figure 4.3 shows the simulation when only the wake up signals to Critical instruction are broadcasted immediately (non-critical are not shown in this figure). In this scenario the pressure on the broadcast logic is further reduced and, as the average data shows, the cases from 0 to 2 wake up per cycle signal gather 95% of the total cases. In detail the cycles with 0 wakeup represent the 67%, while the other from 1 to 3 make for 20%, 9% and 3% respectively.

Looking in details, the benchmarks `cactusADM` and `h264ref` show the biggest difference with respect to the section 4.1.2. This may be due to the fact that most of the instructions do not have a critical dependent in the issue queue.

Noticeably the workloads that less benefit from this scenario (in terms of increase of cycles without any wake up signal needed), are at the same which had already a low usage of the broadcast logic, gcc and mcf.

Lastly the decrease in the broadcast of wake up signal is aligned to the expectation given by Figure 3.4: the benchmark that have more instruction that broadcast to a non-critical dependent have also an higher improvement in Figure 4.3.

Given the data showed so far we can say that the broadcast width needed in this scenario is of 2 or 3 signal per cycle to have a minimal impact on the performances. This simulation helps to have an idea on the broadcast width reduction that could be possible in an ideal situation with only critical broadcast; in realistic scenario all the wake up signal should be considered. An additional component is needed to delay the broadcast when possible.

4.2 Critical instructions

To pursue the goal of reducing the broadcast width, in previous chapters we have introduced the concept of instruction criticality. Previous work [5, 11, 4] have shown that not all the instruction contribute equally to performance. In particular it has been noticed that the instruction which need to access the memory tend to have longer execution time than others and should therefore be prioritize. This is mainly caused by the growing amount of memory off-chip and more complex cache hierarchy.

Among the memory related instruction, our interest is towards the Loads instruction because we know that Stores, by definition, do not have dependent waiting for them in the issue queue. In addition, the instructions that generate addresses for loads are also categorized as critical as loads cannot execute until their address is available. Furthermore, instruction generating addresses for stores are also considered critical. This is because store addresses are needed to disambiguate younger loads.

4.2.1 Detecting critical instruction

To detect critical instruction, we use the Iterative Backward Dependency Analysis (IBDA) [4]. The main purpose of IBDA is to exploit the loops that are commonly present in code to mark the critical instruction using simple hardware structures: The Instruction Slice Table (IST) and Register dependency table (RDT). The first is used to store the address of the instruction marked as critical and the latter stores the address of the instruction which last wrote to the source register of a critical instruction to trace dependencies and identify address-generating instructions. The main idea is that on the first iteration of a loop the Loads are marked as Critical and treated accordingly, meantime IBDA will trace, for each of these Loads, the instruction which computed their addresses in the RDT. On the second iteration of the loop the IBDA, using the information collected previously, will also mark in the IST the address-generating instruction as critical. It has been found that with an IST of 128 entry the IBDA is capable of marking 99% of the relevant instruction [4].

4.3 Broadcast queue

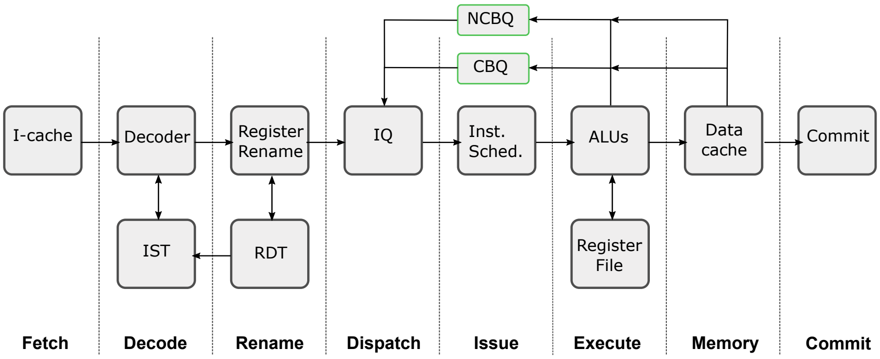

Figure 4.4: The microarchitecture used in this thesis which keeps some components of the Load Slice Core [4]. It introduces two additional FIFO queues (in green) to buffer the Broadcast directed to Critical Instruction (CBQ) and Non-critical instruction (NCBQ)

We have discussed so far that the criticality of an instruction is given by its impact on the execution performance. We have found as a result that definition can be applied to two category of instructions, Loads and their address-generators, because accessing the memory is a long-latency operation.

Given this distinction, in section 3.4 we have found out that the amount of instruction that broadcasts a wake up signal to a non-critical dependent is, on average, 41 %. The main idea is, in fact, to broadcast the wake up signal to critical instruction as soon as possible while the remaining wake up signals can be delayed and spread over the following cycle where the pressure on the broadcast logic is lower. By doing so, we expect to distribute the broadcast of signal more evenly over different cycles and therefore be able to reduce the broadcast width with minimal impact on performance.

To achieve this goal we introduce two simple FIFO queues to buffer the wake up signal informations : the first is used for wake up signals broadcasted to critical dependents and the second for signals to any other instruction. Each new cycle the queue for critical instruction will have higher priority; only when broadcast ports are left unused the second queue can then broadcast its signals.

These queues are introduced because they allow us to vary the broadcast width and understand its impact: when the number of broadcasted signals exceeds the slots available, they will be stored in the queues and handled in the following cycles. This allows us to evaluate which could be the performance loss associated to different broadcast width reductions in order to find which is the best compromise. Throughout our study we will assume that the FIFO queues introduced will have infinite entries.

Chapter 5

Evaluation

In the following chapter we will make a comparison in order to find which is the best compromise taking into consideration the broadcast width reduction and the eventual performance loss.

5.1 Methodology

The main tool that we have used to evaluate the proposed architecture is Sniper simulator [1] which works by expanding Intel's PIN tool with models for the core, memory hierarchy, and on-chip networks. The simulations have been performed using CPU2006 benchmark suite with reference inputs. To keep the simulation time reasonable, SimPoint methodology [12] is used to choose a single most representative region of 1 billion instructions in each application. For our experiment we have considered the following designs.

- The first is based on the Baseline core and introduces one unlimited FIFO queue where all the wake up signals exceeding the broadcast width will be stored.

- The second is an improved version of the first where the instruction with no dependent do not broadcast any signal in order to lower the pressure on the broadcast logic.

- The third is described in 4.3 and uses two different FIFO queue to manage critical and non-critical wake up signal.

5.2 Broadcast width of 4

We first set the broadcast width to 4 as we found in subsection 4.1.1 that the usage of more than 4 broadcasts per cycle is rare; therefore, we don't explore broadcast widths higher than four.

Figure 5.1 shows the performance loss caused by setting the broadcast width to 4 over an unlimited broadcast width design. The results in figure point out that the **baseline** version is, as expected, affected more than the other two designs, i.e. when the instructions with no dependents do not broadcast and when the critical instruction are prioritized. The average performance penalty in the baseline case is about **1%** while the other two designs show much lower performance loss: **0.03%** and **0.02%** respectively. These results show that the performance loss with a broadcast width of 4 is negligible compared to an unlimited broadcast width design.

Considering each benchmark individually we can see that *gamess* presents the peak delay in terms of additional cycles with respect to unlimited broadcast width in all the scenarios considered: 4.3%, 0.2% and 0.2% respectively. This numbers show that all the three microarchitecture, even in the worst case considered, have a limited performance loss, especially when excluding unnecessary broadcast when there are no dependents or when prioritizing the critical instructions.

It is worth noticing that the benchmarks that happened to have 5 or 6 broadcast per cycle more often in Figure 4.1 (`gamess`, `leslie3d`, `h264ref` and `wrf`) here show the highest delay as it can be expected. On the other side, workloads like `hmmer`, even showing an higher pressure on the broadcast logic (94% of the cycles require to broadcast at least 1 wake up logic) does not look to suffer from the reduction in broadcast width. This is because the broadcast width of four is wide enough to wake-up the majority of instructions on time.

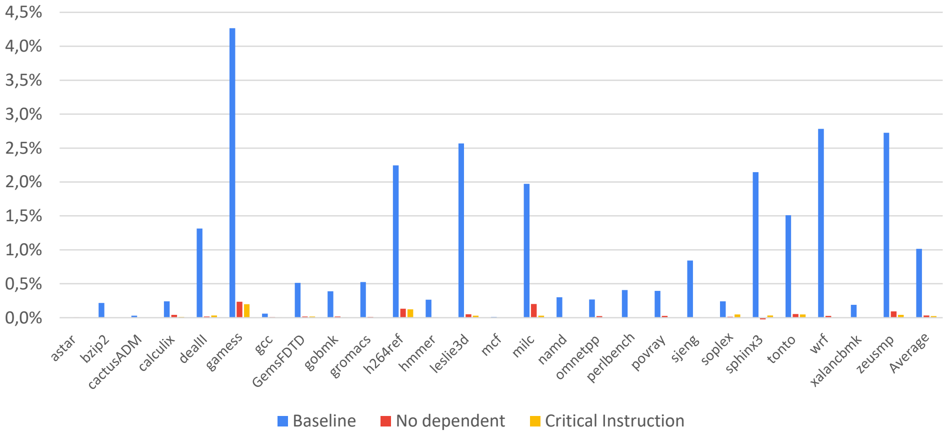

Figure 5.1: The figure shows the increase of the number of cycles, when the maximum number of broadcasted signals is set to 4, expressed as a percentage respect to the case with unlimited broadcast width

5.3 Broadcast width of 3

Figure 5.2 presents the performance loss when the broadcast width is set to 3 in terms of percentage of additional cycles needed to complete execution. In this scenario we found that *No Dependant* and *Critical Instruction* are slightly impacted in the performance, in fact their average delay gets to **0.51%** and **0.50%** respectively.

When looking at individual benchmark we found that the delay for the **baseline** is greater in the same workloads mentioned above (`gamess`, `leslie3d`, `h264ref` and `wrf`) but at the same time not all of them show the same slow down when looking at *No Dependant* and *Critical Instruction*(i. e. `wrf` has a pretty negligible delay both when it broadcasts wake up signals for critical instruction and when excluding the signal when no dependent is in the issue queue) . The *Baseline* maintains a similar behaviour to the previous case with a delay of 1.13%.

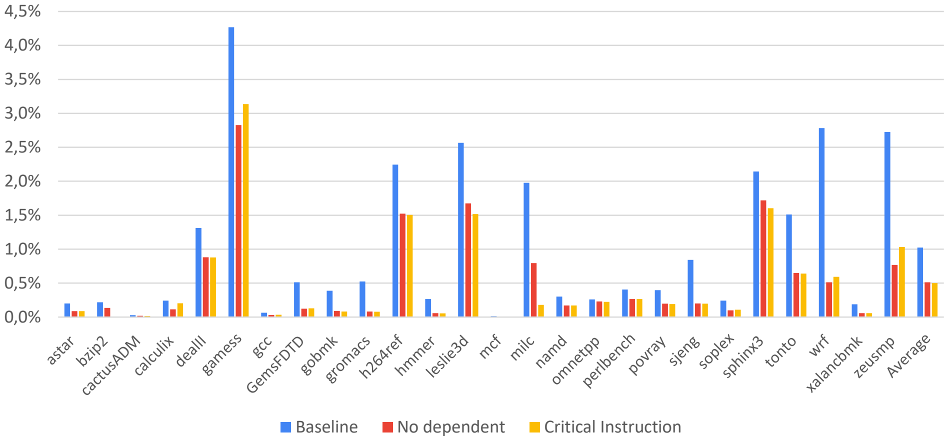

Figure 5.2: The figure shows the increase of the number of cycles, when the maximum number of broadcasted signals is set to 3, expressed as a percentage respect to the case with unlimited broadcast width

5.4 Broadcast width of 2

Figure 5.3 shows the results when the broadcast width is set to 2. A significant performance loss can be seen in the baseline core due to instructions not being waken up on time. The average slowdown for the baseline core is about 9%, with the slowdown being as high as about 24% in gamess. The same trend can be observed also when the design does not broadcasts results for non-dependent instructions. The average performance loss in this case is about 5.6%.

Our final design that considers instructions criticality shows 5.7% performance loss. When looking to benchmarks individually it can be noticed that, besides group of benchmarks already cited before, new critical cases emerged : `calculix` shows the biggest increase of delay probably because the short amount of wake up signals allowed exposes close-up dependency that cause the slow down. The narrow broadcast width also showed that in benchmarks like `gamess` or `calculix` the proposed microarchitecture with two queues is found to cause a greater delay than the second architecture with only one queue. This can be caused by the huge number of ”Critical” broadcast which, due to the prioritized queue, block the ”Non-critical” signals so long that the benefits brought by this architecture on the MLP are counterweight and result in a loss of performance.
Summary: To summarize, simply reducing the broadcast width of the baseline core, to reduce the complexity of wake-up logic, significantly hurts performance. As the results

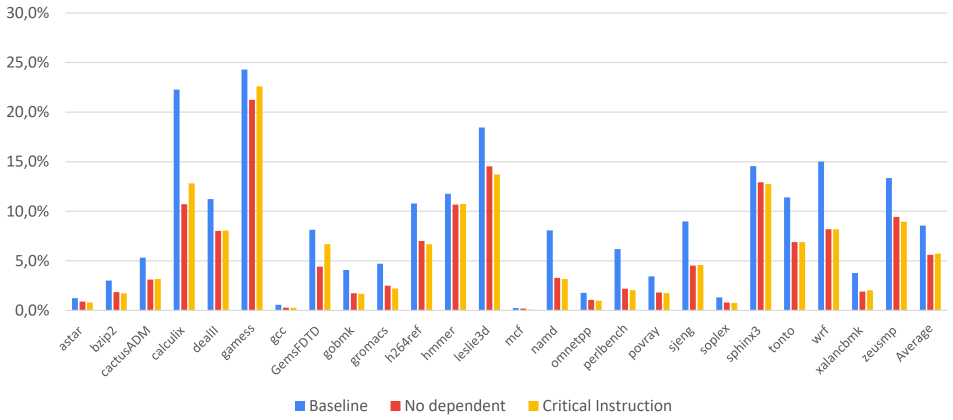

Figure 5.3: The figure shows the increase of the number of cycles, when the maximum number of broadcasted signals is set to 2, expressed as a percentage respect to the case with unlimited broadcast width

presented in this section show, reducing the broadcast width to 2 leads to a performance loss of about 9% and as high as 24% in some benchmarks. Our proposal, in contrast, aim to keep the performance penalty to minimal by taking advantage of instruction criticality and the observation that some instructions do not have their dependents in the issue queue when they finish execution.

From the data collected we can see that adding one queue and preventing instruction to broadcast when they have no dependent gives the best results at the minimum cost. Using two queues to prioritize the broadcasts causes, in some benchmarks, a meaningful loss with respect to the previous solution.

Chapter 6

Related Work

The issue stage, as mentioned before, is one of the most expensive component of a pipeline due to its high complexity. Many studies that have been carried out focus on this stage in order to find simpler architecture with a low impact on performance.

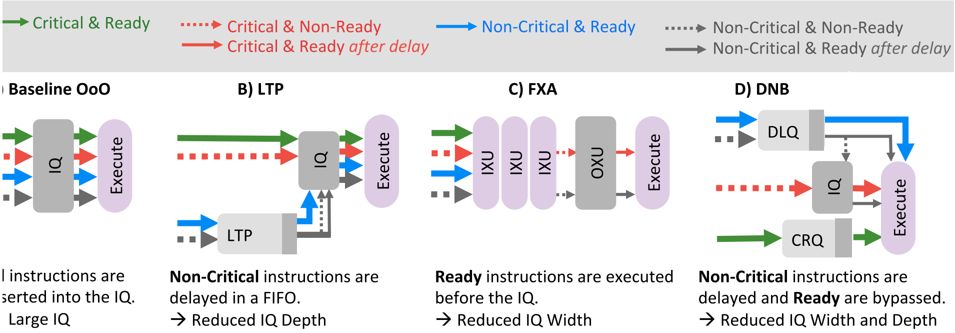

Table 6.1: Comparison withing different proposal to improve the issue stage. Picture from [3]

6.1 Works on out-of-order architecture

One of the first work presented on such topic has been the **MLP-Aware Dynamic Instruction Window Resizing** [8] that, as the name suggests, focuses on extracting Memory-Level Parallelism (MLP). The main observation is that the issue window influences the performance according to the type of workload: the *gcc* benchmark have a degradation of IPC when the windows size increases but, on the other hand, memory intense benchmark like *libquantum* benefit from a large issue window because of the better MLP. This brought to the idea of implementing an architecture which is capable of resizing the window and

have a large size only when necessary to extract MLP and a smaller window when the workload is more computing-intense.
Another important work is **Long Term Parking** [11], which first introduced the idea of separate queues for less important instructions. The idea is to identify which instruction can expose Memory-Level Parallelism and can therefore be considered worth of allocating the expensive IQ resources; all the other types should instead be *”parked”* in a simper and cheaper structure (like a FIFO) until they are ready. An instruction should allocate resources according to a metric based on its *Readiness* and its relation with Long Latency Instruction called *Urgency*, otherwise be ”parked”. This allows to reduce the dimension of the IQ and the Register File without affecting the performance.
Another proposed idea to simplify the issue stage is the **Front-end Execution Architecture** which focuses on reducing the issue width. The design includes a both an in-order execution unit (IXU) and an out-of-order execution unit (OXU) placed one after the other: the instruction will first enter the IXU and,, if ready, it will be executing using the FUs present at this stage; if not ready the instruction will be considered as a NOP and enter the OXU. The IXU in mainly acting like a filter for ready instruction that can instantly execute without wasting the issue resources of the OXU so that the latter can have a reduced width.
Taking on these previous ideas and merging some aspects of it another work has been presented: **Delay-and-Bypass** [3] The instruction are, in this case, separated according to a criteria based on both *readiness* and *criticality*: the resources of the IQ are allocated only to critical instructions waiting to be ready. When it is critical and ready it is issued directly to the execution units; non-critical are put in a simple FIFO queue. This evolution of the criteria allows DNB to reduce the IQ in size and width while keeping better performance than LTP.
The FIFOrder MicroArchitecture [2] focuses on using cheaper FIFO to deduct the pressure on the IQ so to improve energy efficiency up to 50% and performance [2]. To achieve that it categorizes the instruction according to their readiness at the dispatch stage : three of these types (Ready-at-Dispatch, Almost-Ready-at-Dispatch and Load-tail) are found to not benefit from the out-of-order execution and can be put in cheap separated FIFOs to avoid further stalls. The remaining category (which represents on average the remaining 33%) will allocate resources in the IQ. Respect to the cited FXA, this architecture proves to have better performance per energy when using three FIFOs and it does not require to replicate Functional Units. [2]

6.2 Work on in-order architecture

One other approach to the problem of complexity is to improve the performance and ILP of the in-order architecture that are by definition simpler than out-of-order architecture.
The Load Slice Core microarchitecture [4] focuses on extracting Memory-Level Parallelism: the instructions that are interested in this process (Loads and address generating) are marked as *critical* and therefore bypassed to another dedicated queue in order to avoid the stall that can occur in the normal stream. The structure used are still simple (RAMs and FIFOs) because the order is still kept as it is, generating ”slices” of instructions. This idea guarantees simple hardware but its performance cannot be compared to a real out-of-order processor because the dependencies in between instruction still frequently block

MLP generation.
An evolution on this architecture to overcome dependence-oblivious in-order slice execution has been proposed by **Freeway** [9]. The idea is that when executing the bypassed instruction mentioned above, the architecture should be aware of their dependency and, by using another FIFO structure, put them aside until they are available for execution. This whole mechanism allows the independent instruction to be executed out-of-order using a minimum amount of additional Hardware.

6.3 This work

All the projects and proposals that have been illustrated so far take different approaches to improve the issue stage : some start from a out-of-order architecture and try to reduce its complexity without losing performance, other try improve a simple and efficient in-order architecture to gain performance without over complicating. Most of this efforts have been made on the issue logic but none of this works takes into consideration the wake up logic which also represents a critical component. The use of CAM and RAM in the Issue Queue is strictly correlated to the employment of the wake up signals and, in particular, the area and energy consumption are proportional to the ports of the cited memories.

Chapter 7

Conclusion

The continuous demand for higher performance has caused processor design to change drastically over the year. Processors has evolved from simple single-cycle in-order execution machines to highly speculative out-of-order execution engines. Though this evolution has brought many fold performance benefits, it has also increased processor complexity significantly. Specifically, the instruction scheduling mechanism, i.e. the issue stage, is one of the most complex operations in contemporary processors. Its complexity stems from its need to *wake-up* instructions for execution when all their operands become available and *select* them for execution based on priority heuristics.

The goal of this thesis was to analyze and reduce the complexity of the instruction wake-up mechanism. We first established a reasonable baseline by assuming that store and branch instructions never need to broadcast wake-up signal as they do not have any data dependent instructions. Starting from this baseline, we further showed that there are other instructions for which the wake-up signal broadcast can either be completely eliminated or delayed, thus reducing the required broadcast width. For example, if an instruction does not have any dependent instructions in the issue queue when it completes its execution, it does not need to broadcast a wake-up signal at all. Furthermore, we exploited instruction criticality to delay the wake-up broadcast if all the dependents of an instructions are non-critical.

If more instructions finish execution in a cycle than the number of wake-up broadcasts supported by the processor, we buffer the extra broadcasts in two FIFO queues: one for critical and other for non-critical instructions. When a broadcast port becomes available, we issue broadcasts from these queue with instructions in critical queue getting higher priority than non-critical ones.

By reducing the number of instructions that need to broadcast wake-up signal immediately, we are able to reduce the broadcast width with various degree of success. When restricting the broadcast width to 4 and 3 the performance loss is, on average, negligible (around 0.02 % and 0.50% respectively). With the broadcast width set to 2 wake up signal per cycle the delay is more noticeable and stands at 5.6% with only one queue. We also found that implementing two queues of broadcast to prioritize the critical signal does not bring an substantial improvement and, when the broadcast width is set to 2, it causes substantial delays in some benchmarks.

We proved to some degree that is possible, implementing simple hardware, to reduce the broadcast width to simplify the wake up logic with a limited impact on the performances.

Bibliography

[1] The sniper multi-core simulator. Accessed: 2020-04-26.

[2] M. Alipour, R. Kumar, S. Kaxiras, and D. Black-Schaffer. Fiforder microarchitecture: Ready-aware instruction scheduling for ooo processors. In 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pages 716–721, 2019.

[3] M. Alipour, S. Kaxiras, D. Black-Schaffer, and R. Kumar. Delay and bypass: Ready and criticality aware instruction scheduling in out-of-order processors. In 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 424–434, 2020.

[4] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout. The load slice core microarchitecture. Portland, OR, USA, 06 2015.

[5] B. Fields, S. Rubin, and R. Bodík. Focusing processor policies via critical-path prediction. In Proceedings of the 28th Annual International Symposium on Computer Architecture, ISCA '01, page 74–85, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 0769511627. doi: 10.1145/379240.379253. URL https://doi.org/10.1145/379240.379253.

[6] D. Folegnani and A. González. Energy-effective issue logic. Gotemborg, Sweden, 06 2001.

[7] A. González, F. Latorre, and G. Magklis. Processor Microarchitecture An Implementation Perspective. Morgan & Claypool, 2012.

[8] Y. Kora, K. Yamaguchi, and H. Ando. Mlp-aware dynamic instruction window resizing for adaptively exploiting both ilp and mlp. In MICRO '46, Davis, CA, USA, 12 2013.

[9] R. Kumar, M. Alipour, and D. Black-Schaffer. Freeway: Maximizing mlp for slice-out-of-order execution. In HPCA 2019, Washington, DC, USA, 02 2019.

[10] D. M. H. Sarah L. Harris. Digital Design and Computer Architecture. Elsevier, 2016.

[11] A. Sembrant, T. Carlson, E. Hagersten, D. Black-Shaffer, A. Perais, A. Seznec, and P. Michaud. Long term parking (ltp): Criticality-aware resource allocation in ooo processors. In Micro 2015, Honolulu, United States, 12 2015.

[12] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS X, pages 45–57, 2002. ISBN 1-58113-574-2. doi: 10.1145/605397.605403. URL http://doi.acm.org/10.1145/605397.605403.

[13] J. Stark, M. D. Brown, and Y. N. Patt. On pipelining dynamic instruction scheduling logicthe load slice core microarchitecture.

[14] S. Zhang, C. Zhu, J. K. O. Sin, and P. K. T. Mok. A novel ultrathin elevated channel low-temperature poly-Si TFT. 20:569–571, Nov. 1999.

NTNU

Norwegian University of
Science and Technology