ADVERTIMENT. L'accés als continguts d'aquesta tesi doctoral i la seva utilització ha de respectar els drets de la persona autora. Pot ser utilitzada per a consulta o estudi personal, així com en activitats o materials d'investigació i docència en els termes establerts a l'art. 32 del Text Refós de la Llei de Propietat Intel·lectual (RDL 1/1996). Per altres utilitzacions es requereix l'autorització prèvia i expressa de la persona autora. En qualsevol cas, en la utilització dels seus continguts caldrà indicar de forma clara el nom i cognoms de la persona autora i el títol de la tesi doctoral. No s'autoritza la seva reproducció o altres formes d'explotació efectuades amb finalitats de lucre ni la seva comunicació pública des d'un lloc aliè al servei TDX. Tampoc s'autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant als continguts de la tesi com als seus resums i índexs.

ADVERTENCIA. El acceso a los contenidos de esta tesis doctoral y su utilización debe respetar los derechos de la persona autora. Puede ser utilizada para consulta o estudio personal, así como en actividades o materiales de investigación y docencia en los términos establecidos en el art. 32 del Texto Refundido de la Ley de Propiedad Intelectual (RDL 1/1996). Para otros usos se requiere la autorización previa y expresa de la persona autora. En cualquier caso, en la utilización de sus contenidos se deberá indicar de forma clara el nombre y apellidos de la persona autora y el título de la tesis doctoral. No se autoriza su reproducción u otras formas de explotación efectuadas con fines lucrativos ni su comunicación pública desde un sitio ajeno al servicio TDR. Tampoco se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al contenido de la tesis como a sus resúmenes e índices.

WARNING. Access to the contents of this doctoral thesis and its use must respect the rights of the author. It can be used for reference or private study, as well as research and learning activities or materials in the terms established by the 32nd article of the Spanish Consolidated Copyright Act (RDL 1/1996). Express and previous authorization of the author is required for any other uses. In any case, when using its content, full name of the author and title of the thesis must be clearly indicated. Reproduction or other forms of for profit use or public communication from outside TDX service is not allowed. Presentation of its content in a window or frame external to TDX (framing) is not authorized either. These rights affect both the content of the thesis and its abstracts and indexes.

Reusing cached schedules in an out-of-order processor with in-order issue logic

Oscar Palomar

Directors: Toni Juan i Juan. J. Navarro

Departament d'Arquitectura de Computadors

Tesi presentada per obtenir el títol de Doctor
per la Universitat Politècnica de Catalunya

Març 2011

Al Graeki, per ser com eres. Et recordarem sempre.

Für Neus. Danke für die deine Geduld, du hast sehr viel!
Danke auch für die gemeinsame Zeit, wir werden viel mehr Zeit zusammen verbringen.
Ich liebe dich.

A mi madre y al resto de la familia, por vuestra paciencia y los ánimos.

Als meus directors Toni i Juanjo, per tot el que m'heu ensenyat, pels bons consells
i l'ajuda en els moments difícils.

To the people in the BSC office. It is great to work with you!

A tots els amics, que no m'heu tingut en compte que sigui tant car veure'm.
No us ho he posat fàcil.

This work has been supported by the Ministry of Science and Technology of Spain under contract TIN2007-60625.

Ok
Here's what we're going to do
We all know the numbers right?
From zero to infinity
Whatever
Some other number with a mess of zeros behind it
Here's what we're going to do
We're gonna change the order
of these numbers
to make things interesting
Ok here we go:
1 Million and 1
Sixty-six
1 Billion, twenty-five, seventy-five thousand
1 Billion and eight, six, something
Zero
1 Million 1
Twenty-two
Seventy-five
Eleven
Eleven
Ok this is the new order
The New Number Order

New number order, from the album "1000 hurts" by Shellac

Contents

Chapter 1

Introduction

Modern processors use out-of-order processing logic to achieve high performance in Instructions Per Cycle (IPC) but this logic has a serious impact on the achievable frequency. In order to get better performance out of smaller transistors there is a trend to increase the number of cores per die instead of making the cores themselves bigger. Moreover, for throughput-oriented and server workloads, simpler in-order processors that allow more cores per die and higher design frequencies are becoming the preferred choice (IBM's Power6 [1] [1], Sun's Niagara 3 with 16 cores [5, 6]). Unfortunately, for other workloads this type of cores result in a lower single thread performance.

There are many workloads where it is still important to achieve good single thread performance. In this thesis we present the ReLaSch processor; its aim is to enable high IPC cores capable of running at high clock frequencies by processing the instructions using simple superscalar in-order issue logic and caching instruction groups that are dynamically scheduled in hardware after commit, that is, out of the critical path and only when really needed.

1.1 Objective

This thesis has several research goals:

- To show that the dynamic scheduler of a conventional out-of-order processor does a lot of redundant work because it ignores code repetitiveness.
- To propose a complete superscalar out-of-order architecture that reduces the amount of redundant work done by creating the schedules once in dedicated hardware, storing them in a cache of schedules and reusing the schedules as much as possible.
- To place the scheduler out of the critical path of execution, which should be enabled by the reduction of work that the scheduler must do. Thus, the execution path of our proposed processor can be simpler than that of a conventional out-of-order processor.

1.2 Out-of-order processors: an overview

This section presents a short introduction to out-of-order processors. It describes the concepts involved in the rationale behind our proposal. This overview is not intended to be a survey of all the

[1] Using a technology independent model, the Power6 processor doubles the frequency of the out-of-order Power5 [1]. The Power7 processor is out-of-order [2]. However, in its initial release its maximum available frequency (4.25GHz as of October 2010 [3]) is lower than that of the Power6 (5GHz as of April 2010 [4]).

possible ways to implement an out-of-order processor. It is a well-known topic in the computer architecture area, so we just give a short overview of the motivation to use them, an insight on the key implementation issues and a description of their main problems.

1.2.1 Concept

Out-of-order processors are able to execute the instructions of a program in a different order than that specified by the programmer. As long as the out-of-order processor respects the dependences between instructions, the outcome of the program will be indistinguishable from the result produced by a sequential (in-order) execution, that is, in the original order of the instructions.

1.2.2 Classification

In-order processors (IO). Also known as statically-scheduled processors. For this group of processors, all the instruction scheduling is done by the compiler. The microarchitecture issues groups of independent instructions that are consecutive in program order. At execution time, there are no changes on the order chosen by the compiler at compilation time and hence this mechanism is called statically-scheduled.

Out-of-order processors (OoO). Also known as dynamically-scheduled processors. On top of the static instruction scheduling performed by the compiler, the processor performs a second level of dynamic scheduling. Instructions are fetched following the predicted sequential order and put into an instruction queue. The microarchitecture groups the best independent instructions from the window dynamically and issues them for execution in a better order than the one chosen by the compiler based on the resources available.

On one hand, out-of-order processors can extract more instruction level parallelism (ILP) from the instruction stream. Furthermore, existing binaries can benefit from changes in the microarchitecture without recompiling. On the other hand, in-order processors are simpler and, therefore, smaller; additionally, they consume less power and either have a lower number of pipeline stages (lower penalties) or can be designed to run at higher frequencies.

1.2.3 Motivation

Out-of-order processors are attractive because they can achieve high speed-ups in IPC and execution time with respect to in-order cores. Whenever there is a data dependence with a long latency instruction, out-of-order processors are able to continue executing younger independent instructions, whereas in-order processors stall.

The compiler can schedule the instructions statically for a well-known in-order processor and remove these stalls, but still there are two important scenarios in which it is impossible:

a. The presence of instructions with variable latency, such as the accesses to the memory hierarchy, that have very different latencies on a miss and on a hit. Although the introduction of pre-fetch mechanisms can mitigate the problem, there is still much uncertainty in the latency of such instructions. Moreover, these instructions have a significant share of the instruction mix and their impact cannot be neglected. The flexibility of out-of-order processors enables them to find independent instructions that can be executed while waiting for cache misses.

b. Speculative execution through branch prediction. When instructions are executed speculatively the chances of finding additional independent instructions in the speculated path increase, which out-of-order processors can exploit. On the other hand, compilers must be conservative to

preserve the semantics of the program. They don't move instructions across basic blocks unless it is safe to do the optimization. Therefore, in most cases the independent instruction will not be statically rescheduled. With the use of techniques like predication the compiler can work around this problem, but it comes with an overhead in the code size and the total number of instructions executed.

To summarize, the flexibility of dynamic scheduling allows to adapt to changing situations such as variable instruction latency and changing execution paths that static scheduling cannot manage efficiently.

1.2.4 Implementation

This section presents the main structures needed to implement a generic out-of-order processor.

Dependences

The first thing to consider is what is a dependence and when instructions can be considered independent. Figure 1.1 illustrates the possible cases. The examples show cases in which only registers are involved. Memory and control dependences are left aside here for simplicity.

Both output and anti-dependences are considered "false" or "name dependences" since we can eliminate the dependence by changing the registers used by the instruction. On the other hand, a data dependence cannot be removed using this technique: the dependent instruction must wait until the needed value is produced.

Register Renaming

The register renaming logic eliminates the false dependences while preserving the semantics of the program. It maps the logical registers (the ones specified in the Instruction Set Architecture or ISA) to a larger set of physical registers. There can be several instances of the same logical register, although just one stores the architectural value. A new physical register is assigned to each destination logical register, while all reads to a given logical register are renamed to its most recently mapped physical register. Thus, the data dependences are preserved whereas false dependences are removed.

Physical registers are recycled after they are no longer needed. Depending on the implementation, that happens after the retirement of the instruction that writes the register or after the last read to the register.

The structures commonly used to implement register renaming are: a) the rename table with the most recent mapping of each logical register; b) the architectural rename table with the physical register that currently stores the architectural value of each logical register; and c) the free list that contains the free physical registers, that is, the registers available to be mapped to incoming instructions.

An important problem of register renaming is that two consecutive instructions may be dependent, so the renaming state after the first instruction is processed must be known to rename the second correctly. The consequence is that the renaming logic grows in complexity in superscalar processors, since the renaming logic must check any dependence between all instructions being mapped at the same time and rename them accordingly. Furthermore, the outcome of the renaming logic at a given cycle is used the next cycle. Such tight loop makes it challenging to pipeline the renaming logic without having an impact in performance.

a) Data dependence b) Output dependence c) Anti-dependence d) Independent code

```
A: MUL R1, R2, R3    A: MUL R1, R2, R3   A: MUL R1, R2, R3   A: MUL R1, R2, R3
B: ADD R3, R4, R5    B: ADD R4, R5, R3   B: ADD R4, R5, R1   B: ADD R1, R4, R5
```

Figure 1.1: The examples of code show, from left to right: a) Data or real dependence. If the instructions have the chance to execute out-of-order there is a Read-After-Write (RAW) hazard. b) Output dependence. The hazard it can create is called Read-After-Write (RAW). c) Anti-dependence. It is related with Write-After-Read (WAR) hazards. d) Independent code. The instructions can be freely reordered. The instructions in the examples indicate the destination register in the right-most position.

In-order retirement

All modern out-of-order processors retire the instructions in the original program order. The use of branch prediction already forces them to wait until the branch is resolved to retire the speculated instructions, even if they have finished execution many cycles before. Moreover, retiring the instructions in order forces exceptions to be precise. An exception is imprecise if the architectural state seen by the exception handler has been modified by instructions younger than the one that raised the exception, or if the state has not been modified by an older instruction. Both cases can occur in an out-of-order processor if instructions are allowed to retire out-of-order.

After instructions are renamed, they are inserted in the Re-Order Buffer (ROB), a First-In First-Out (FIFO) structure. An instruction stays in the ROB until it reaches the head of the buffer and it has completed execution with no exception or mis-speculation. The instruction retires (what is known as the Commit stage), updating the architectural state: that is, the destination register and the PC. It also updates the memory if the instruction is a store.

The architectural rename table tracks which physical registers hold the architectural state. The architectural rename table is updated in the Commit stage. Alternatively, dedicated architectural registers can be used. Additionally, the store queue keeps the information of all in-flight (non-committed) stores, including the address that is going to be accessed and the value that will be written. This is needed in order to update memory in the Commit stage.

A load instruction accesses the store queue before accessing the cache and checks if there is any older store that accesses the same address, a situation known as memory aliasing. If such an aliasing exists, the data is bypassed directly from the store queue instead of accessing the cache. In an out-of-order processor, memory instructions may calculate the addresses out-of-order too. To avoid stalling until the addresses of all older accesses are known, load instructions usually access the cache speculatively. When a store resolves its address, it checks if there is any younger load that has speculatively accessed the same address. The load is re-executed in case of an aliasing, typically forcing a pipeline flush. Thus, there is also a load queue that tracks all in-flight loads. The load and the store queues are sometimes merged into just one queue. Usually there is also a prediction mechanism to prevent speculative execution of loads that have been re-executed previously and thus reduce the number of pipeline flushes.

1.2.5 Issue logic

The issue logic is responsible for sending the instructions to the functional units (FU) when they are ready. The instructions are inserted in the issue queue after that have been renamed. The issue logic has to implement two main functionalities:

a. Wake-up: It tracks which instructions are ready to be executed (i.e. all its operands are available). This means that whenever there is a new result available, the logic has to check if it is

the operand of any instruction that is present in the issue queue. A common implementation uses comparators in each entry of the issue queue. It compares the physical register of each new result produced by the FUs with the physical source registers of the instructions in the queue. The tags that identify the instructions that produce each result are broadcasted to all the entries of the issue queue.

b. Select: From all the ready instructions in the queue in a given cycle, it has to select a group of them to execute in the FUs. The selection must respect the FU allocation rules to prevent any structural hazard. i.e., the number and type of FUs, the issue latency of operations previously sent to them, clustering of FUs, etc. The maximum number of instructions than can be selected per cycle defines the issue width of the processor. If more than one instruction can be selected, the processor is considered a superscalar core.

1.2.6 Main problems

It becomes apparent that out-of-order processors require several complex structures. Even worse, a technique like register renaming introduces a tight loop in the critical path of execution, since the physical register used by the instructions renamed in a given cycle are needed when renaming the instructions in the next cycle. This makes difficult to pipeline the renaming logic. Also, its complexity make these structures power-hungry.

An even more important problem is scalability. It has been shown that these structures don't scale well with quadratic increments in latency when the issue width or the length of the queues is increased [7].

1.3 Repeated issue in the out-of-order processors

Functions and loops are basic structures used to code algorithms. Most of the time, out-of-order issue logic processes a reduced amount of different instructions and many cycles it ends up issuing together the same groups of independent instructions. Eventually, the schedule adapts to new situations such as a cache miss or a change in the executed path.

The issue logic of an out-of-order processor is a mechanism designed to find each cycle the maximum available parallelism. All current implementations do not remember any information about the parallelism found before, the latencies of the memory instructions or the dependences found in the past. They do not benefit from the repetitive behavior of code, unlike other processor elements, e.g. caches or branch predictors, that indeed rely on this characteristic of the programs to perform as expected. The issue logic is the only part of the processor designed to efficiently execute valid random code.

To show that the issue logic repeats most of its work, we have captured the **issue-groups** (instructions issued in the same cycle) created by an out-of-order processor on simulations of the SPECcpu2000 benchmarks for 100 millions of instructions. The experimental environment is presented in detail in section 9.1. Figure 1.2 shows the percentage of unique issue-groups that add up 90% of the cycles. From this data, a 90/10-like rule of thumb can be postulated: 6% of the issue-groups appear 90% of the cycles. Regrettably, the issue logic is constantly creating the same issue-groups in the critical path of execution. These experiments are examined in more detail in chapter 2.

1.4 Proposal and thesis outline

We present the **ReLaSch** processor, named after Reused Late Schedules, in which the creation of issue-groups is removed from the critical path of execution. It uses a simple and small in-order

issue logic. It just wakes-up and selects the instructions of a single issue-group each cycle, instead of processing the instructions of a whole issue queue. The size of the issue-group is limited by the issue-width (four integer and two floating point instructions in the default ReLaSch and our reference processors), much smaller than the issue queue (20 integer and 15 floating point instructions in our reference out-of-order processor). Chapter 3 provides a general description of the ReLaSch processor.

A new logic at the end of the conventional pipeline schedules the committed instructions into **rgroups** (which are sequences of issue-groups). The default configuration of ReLaSch uses 256-instruction rgroups, with issue-groups that include up to four integer and two floating point instructions each. The new scheduler can be complex since it is not in the critical path of execution: our experiments show that even using a 20-stage scheduler does not affect the IPC achieved. Chapter 4 describes the scheduler.

The rgroups are stored in a cache. Whenever it is possible, an rgroup is read and its instructions executed; the schedules are reused, thus lowering the pressure on the scheduling logic. The cache is presented in chapter 5, while the new front-end that reads the rgroups is described in chapter 6. The in-order issue logic is detailed in chapter 7, while a description of the rest of elements of the processor is included in chapter 8.

In some cases, the ReLaSch processor is able to outperform a conventional out-of-order processor, because the post-commit scheduler has a broader vision of the code. For instance, while ReLaSch can schedule together two independent instructions that are distant in the code, a conventional out-of-order processor only issues them in the same cycle if both are present at the same time in the issue queue.

Conventional out-of-order processors use branch prediction and memory aliasing speculation to find more available instructions. Besides that, their issue logic adapts to variable latency instructions. The ReLaSch processor predicts the branch targets, memory aliases and latencies at scheduling time, out of the critical path. The prediction is based on the most recent executions seen at scheduling time. Conventional branch predictors make their prediction at execution time and can adapt it faster than ReLaSch. ReLaSch average branch misprediction rate is slightly higher than that of a conventional branch predictor.

Out-of-order issue logic reacts immediately to changes in code behavior. ReLaSch relies on the repetitive nature of the code. Therefore, on a change the schedule mispredicts or stalls due to an

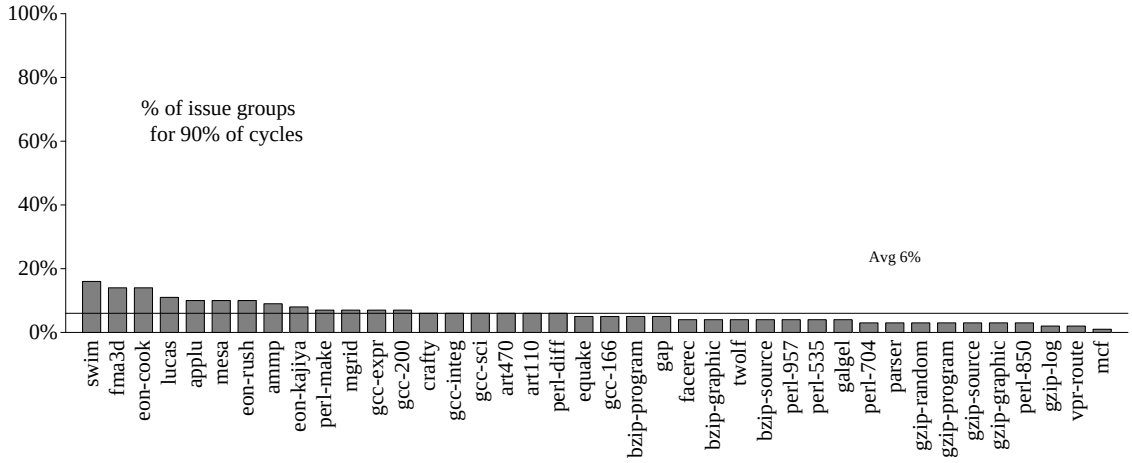


Figure 1.2: Number of unique issue-groups created by an out-of-order issue logic.

unexpected latency. To reduce the ammount of wasted cycles, it detects and replaces the rgroups that continuously fail.

The ReLaSch processor retains many techniques and elements from conventional out-of-order processors. It has a Reorder Buffer and a Commit stage to retire instructions in-order and provide precise interrupts. Additionally, it performs registers renaming to eliminate false dependences. Since the instructions of an issue-group are independent by construction, the renaming logic can process all the instructions of the issue-group in parallel and is simpler than the renaming logic of conventional out-of-order processors. ReLaSch doesn't detect the dependences between the instructions renamed in the same cycle.

Furthermore, most of the renaming process is performed by the scheduler and is removed from the execution pipeline. In particular, there is no loop in the renaming logic of the front-end; the instructions of an issue-group are renamed independently of the instructions of the previous issue-group. The absence of a tight loop reduces the complexity of this logic and eases pipelining it if required. An alternative design for the register file is shown in appendix A.

There is previous work that proposes moving the scheduling logic out of the critical path and/or trying to reuse instruction schedules [8], [9], [10], [11]. Other approaches simplify the issue logic of an out-of-order processor or reduce the size of its issue queue. In chapter 10 the main differences and similarities of ReLaSch with these proposals are highlighted.

The work presented here has been already partially published. [12] presents a study on the issue-groups generated by conventional out-of-order and in-order processors. [13] describes the whole ReLaSch processor and presents some experimental results. [14] provides an overview of ReLaSch and a preliminary study on the use of statically configurable maximum length of the rgroups.

1.5 Scope and results summary

Our experiments show that the ReLaSch processor achieves the same average IPC as our reference out-of-order processor and is clearly better than the reference in-order processor (1.55 speed-up). In all cases it outperforms the in-order processor and in 23 SPEC benchmarks out of 40 it has a higher IPC than the reference out-of-order processor. Chapter 9 presents the experiments that we have conducted in order to explore the design space of the many parameters of the ReLaSch processor. It also presents the experimental results of ReLaSch when compared with the reference in-order and out-of-order processor. Chapter 11 presents the conclusions.

The performance results presented in this work assume that all the processors use the same frequency. However, it would be reasonable to assume that the in-order processor and our proposed ReLaSch processor can achieve a higher frequency. This would translate into a higher performance speed-up over the reference out-of-order processor than the IPC speed-up shown here. We do not evaluate cycle time but describe the microarchitecture and compare the IPC. It is a proof of concept in order to validate the potential of our proposal.

After our experiments show good IPC results and back up our proposal, power consumption should also be taken into consideration. Power is not evaluated in this thesis, but it is part of the future work presented in chapter 11. We must take into account both the positive and negative impact in the power requirements of ReLaSch. On one hand, the in-order issue logic of ReLaSch is much simpler and it is very likely to be much less power-hungry than the issue logic of a conventional out-of-order processor. Moreover, the scheduler is not required to work each cycle, which can result in less energy consumed. On the other hand, the cache of schedules is a new source of energy consumption and is not present in a conventional out-of-order processor. Additionally, the structures needed by the scheduler increase our energy consumption. We expect that the reduction due to the in-order issue logic and the reduced criticality of the scheduler will outweigh the new sources of energy consumption.

To support this expectation, there is the fact that we have reduced the complexity in the critical

path of execution while adding new structures in a scheduler out of this critical path. Thus, aggressive power- and energy- saving techniques can be applied to the scheduler since they are unlikely to have a significant impact in performance. The exception is the cache of schedules that is a new source of energy consumption in the execution path. However, in this case we can also easily apply power- and energy- saving techniques when power consumption and battery lifetime have higher priority than performance. It is very simple to switch off portions of the cache and trade-off energy for performance. Also the scheduler can be switched off even when it would be possible to create new schedules. These power-saving techniques have some granularity and would be simple to implement. Finally, the scheduler can be designed to place the instructions following power-saving techniques. Taken all these ideas into account, we consider that ReLaSch has potential to be power-efficient.

Chapter 2

Motivation

In this chapter we study the repetitiveness of the groups of instructions that are selected by the issue logic. This repetitiveness enables designing a processor that reuses schedules that are created just once.

2.1 Reference processors

This section presents the reference processors used in this study.

The OoO processor

Figure 2.1 shows the pipeline of an improved out-of-order processor (OoO) based on the 21264 Alpha [15] (see appendix B), enhanced with better memory alias detection and branch target prediction. The pipeline is much like in the 21264 Alpha processor.

The Fetch stage reads the instructions from the instruction cache (Icache) and accesses the branch predictor. The result of the prediction is not known until one cycle later. The next stage completes the branch prediction and decodes the instructions. The Map stage renames the instructions and inserts them in the Reorder Buffer (ROB), in the issue queue and in the Load and Store queues (LQ and SQ) if needed. The integer and floating point instructions use separated issue queues. These three first stages process the instructions in-order.

The Issue stage wakes-up and selects out-of-order instructions that are present in the issue queue. It selects up to four integer and two floating point instructions per cycle. An instruction can be selected if its source registers are ready and a suitable Functional Unit is available. Older instructions in the queue have higher priority. Registers are read in the next stage and execution happens in the corresponding functional unit afterwards. Once the execution in the functional unit finishes, the registers are written during the Writeback (WB) stage, where the loads also start the access to the data cache (Dcache). The Writeback stage also checks for memory ordering violations with the help of the LQ and SQ.

Finally, instructions are retired in-order in the Commit stage. Once a completed instruction reaches the head of the ROB, the Commit stage sets its destination physical register as the architectural register for the corresponding logical register, checks for mispredictions and frees the entry used by the instruction in the ROB and in the LQ or the SQ if needed. It also performs the access to memory of store instructions.

Our OoO reference processor improves the original microarchitecture of the Alpha 21264 in two ways: First, the BTB is enhanced with a path-indexed table, used to predict the multi-target indirect branches. Second, the StWait bits, that are used in the 21264 Alpha to reduce the number of loads

that must be re-executed due to memory order violations, are substituted in the OoO processor by a better mechanism: the Store Sets [16]. With Store Sets, a load waits only until the aliased stores have committed, instead of waiting for all the in-flight stores, with a corresponding performance gain.

The ROB has 80 entries, the instruction queue has 20 entries for integer instructions and 15 for floating point instructions. The integer register file has 72 physical registers and the floating point register file has another 72. The LQ and the SQ have 32 entries each. The issue logic can process up to four integer and two floating point instructions per cycle. A complete description of the architectural parameters is shown in table 9.1 of chapter 9.

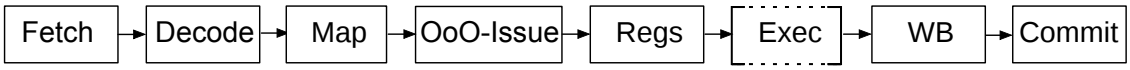


Figure 2.1: The pipeline of the OoO processor.

The IO processor

Figure 2.1 shows the pipeline of the in-order processor (IO) that we use as reference. It is also based on the 21264 Alpha but it has an in-order Issue stage. Besides, it does not rename the instructions, so the Map stage is removed from the pipeline. In the Decode stage, the instructions are inserted in the ROB (to support speculation) and in the issue buffer, that substitutes the issue queue of the Alpha 21264 processor. The issue logic selects up to four integer and two floating point instructions per cycle. The Commit stage is still needed to check the correctness of the branch prediction and the ROB is also used to allow the multi-cycle instructions to write the results out-of-order while maintaining precise exceptions. It uses the same Store Sets and the enhanced BTB of our reference OoO processor.

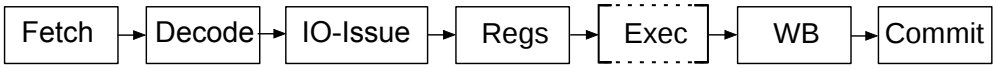


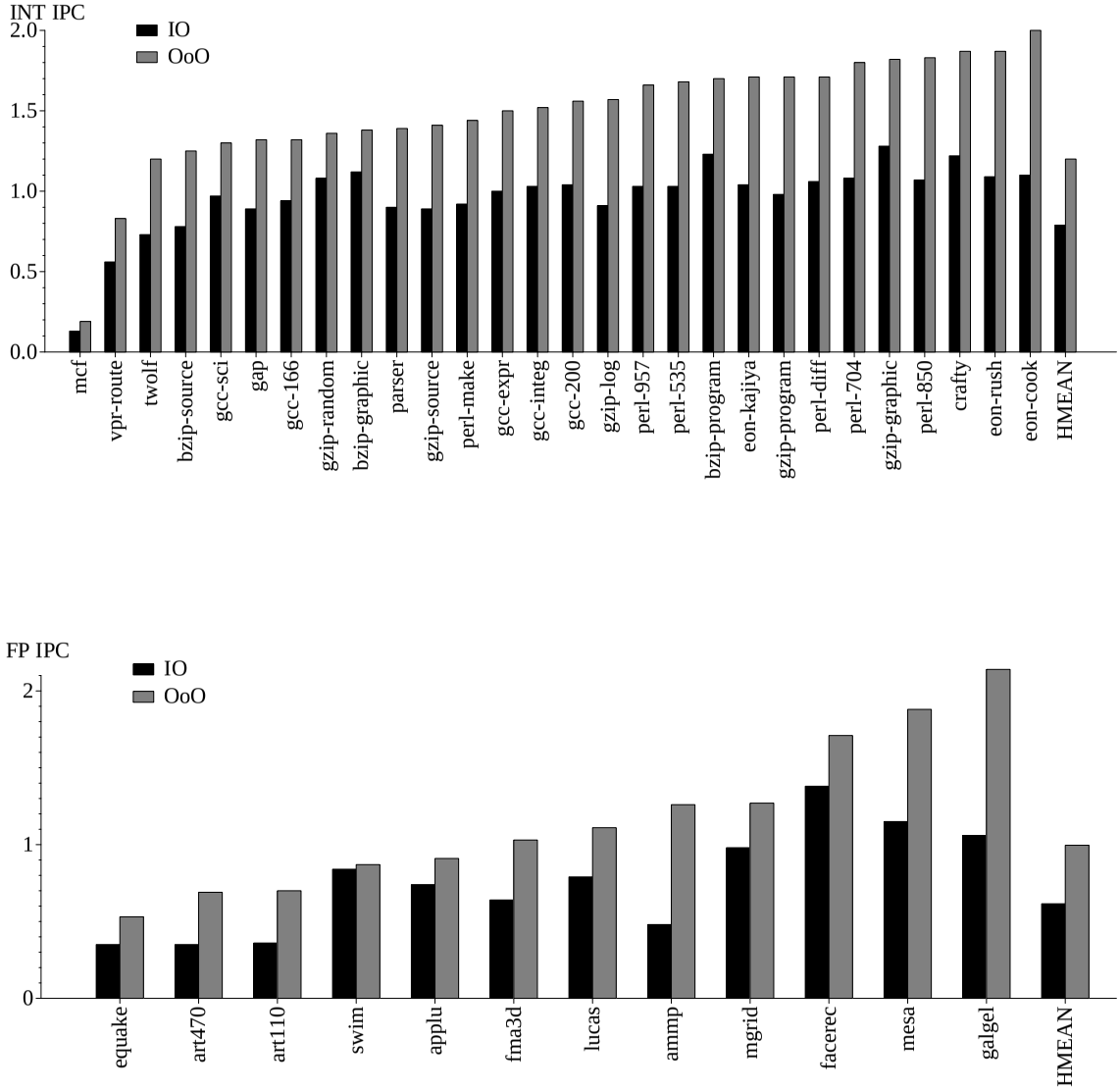
Figure 2.2: The pipeline of the IO processor.

2.2 Contribution of dynamic scheduling to performance

Figure 2.3 shows the IPC achieved by the two processors. In average, the IPC of the OoO processor is 56% higher than the IPC of the IO processor.

In the static scheduling approach, if an instruction stalls waiting for an operand that is not available yet, the processor does not allow any younger instruction to start its execution. In order to improve performance, the compiler takes into account the latencies when scheduling the dependent instructions, placing independent instructions between the producer and the dependent instructions. However, many times it does not find enough independent instructions to eliminate all the stalls in the IO processor. Furthermore, there are instructions with variable latency, such as memory instructions, which the compiler has to schedule assuming a fixed latency. While the OoO processor can adapt the schedule to the actual latency of these instructions, the IO processor stalls each time the actual latency is larger than the one expected by the compiler (for example, when a load misses in the Dcache but was scheduled assuming hit).

An additional problem is that a processor can issue together instructions from different basic blocs using a branch predictor. Both in-order and out-of-order processor can benefit from branch prediction. However, when the processor uses a branch predictor and executes instructions speculatively, new dependences appear across the basic blocks. Which dependences appear at execution time depend on



IPC	IO	OoO
HMEAN INT	0.97	1.50
HMEAN FP	0.76	1.17
HMEAN ALL	0.91	1.40

Figure 2.3: IPC of the IO and OoO processors.

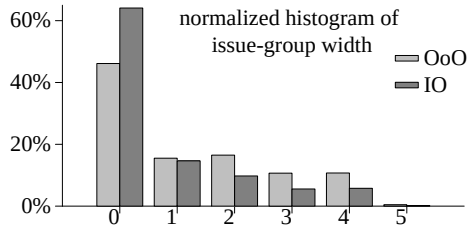


Figure 2.4: Distribution of the issue-group size depending on the instruction scheduling policy, in a four-integer and two-floating point issue processor.

which path follow the branches. The compiler cannot create an efficient schedule for all the possible combinations of basic blocs, so the in-order processor stalls often whereas the out-of-order processor is able to adapt its schedule to the actual dependences of the predicted path.

The register renaming logic assigns a new physical register each time a logical register is used as destination. Dependent instructions that use the logical register as source operand actually read the renamed physical register. Thus false dependences are eliminated. Dynamically-scheduled processors hugely benefit from this technique, since it is able to find much more independent instructions per cycle.

To regularly find enough ready instructions each cycle, the issue logic of out-of-order processors needs a large issue queue, from which it selects the instructions to issue. The issue queue stores the instructions after they are decoded and renamed and until they are issued. Instructions wait there for their source operands to become ready. The wake-up logic notifies to the instructions in the issue queue that a register is available. With a large issue queue, the issue logic finds more ready instructions per cycle, but it also grows the complexity of the wake-up and select logic [7].

2.2.1 Quantitative analysis of the behavior of the dynamic-scheduling logic

In a superscalar processor, a high IPC is achieved by issuing together as many instructions as possible. To measure how much of that work is redundant, we keep track of the issue-groups that are created during execution. We define an issue-group as the instructions that are issued in the same cycle. The issue-groups may include instructions from a mispredicted path.

Figure 2.4 shows the normalized histogram of the size of the issue-groups, for the IO and the OoO processors. Both processors can issue up to four integer and two floating point instructions per cycle. The ability of the dynamic scheduler to find independent instructions allows the OoO processor to have larger issue-groups in average. Besides, the in-order issue logic stalls frequently and most of the cycles is not able to issue any instruction at all.

We consider two issue-groups to be equal if they have the same number of instructions and the PCs of the instructions of both issue-groups match. Figure 2.5 shows how many unique issue-groups are created during the execution of each benchmark (100M instructions per benchmark). The figure is ordered by decreasing number of issue-groups created by the OoO processor. The rest of figures in this chapter follow this same order. From the figure, we can see that the number of issue-groups is much smaller than the total number of cycles. Having less restrictions to issue the instructions, the OoO processor creates a higher number of unique issue-groups during the execution of each benchmark.

Not all the issue-groups are executed the same number of times and we expect that a small percentage of issue-groups are executed frequently than the rest. Figure 2.6 shows how many unique issue-groups are needed to accumulate the 90% of the cycles (excluding the cycles where no instruction is issued). Figure 2.7 shows the same information as a percentage of the total number of unique issue-

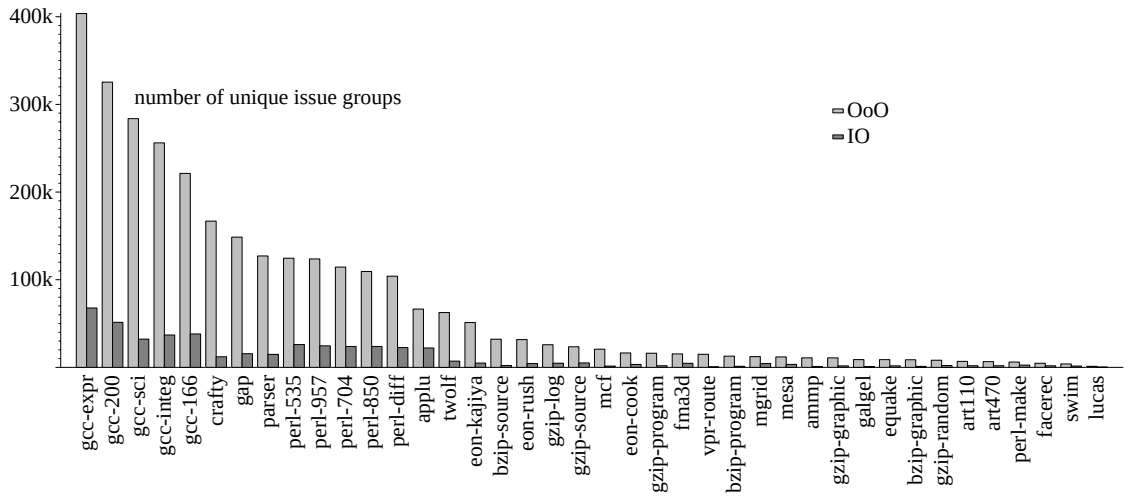


Figure 2.5: Number of unique issue-groups created by the OoO and IO processors (up to four integer and two floating point instructions per cycle).

groups created for each benchmark. As expected due the repetitive nature of code, the experiments show that the dynamic scheduler is constantly creating a reduced set of issue-groups, which appear again and again. Only from time to time it creates different issue-groups. In average, the 90% of the execution is covered with only a 6% of the total unique issue-groups.

2.3 Reusing the dynamic schedules

Given that most of the time the issue logic repeats the selection of instructions that it schedules, it seems reasonable that a new processor could use a dynamic scheduler only when new instructions are executed or there is change in the behavior of the code (for example, when a branch changes its behavior or a load misses in data cache). The issue-groups created then would be cached and the processor would read the issue-groups from the cache the rest of the time. When the issue-groups are not available in the cache, the processor would issue the instructions in-order, using a simple issue logic.

Such a processor is expected to achieve the IPC of an out-of-order processor when the issue-groups are read from the cache and the IPC of an in-order processor when the required issue-group is not available in the cache. There would be some additional penalties for changing the execution mode. So such a processor needs a high hit-rate in the issue-group cache to achieve a high IPC.

We have evaluated the miss-rate of a cache that stores the issue-groups as they are created by the issue-logic of the OoO processor (with a four-integer and two-floating point wide issue logic). The cache is not used in the experiment to feed the processor but it is just used to study the locality of the issue-groups. The PCs of the instructions in the issue-group are hashed. The result of the hash is used to index the cache. The cache is four-way set-associative and the number of issue-groups it can store varies from 1K to 8K.

Figure 2.8 shows the miss-rate of the cache. The results for the different sizes of the cache are shown overlapped in the figure. The results show that there is some degree of locality in the issue-groups created: in average, the 4K-issue-group cache has a 6% miss rate.

For each instruction in an issue-group the cache should store the PC and the encoded instruction. The encoded instruction is needed to execute it. We are assuming that the processor would use some

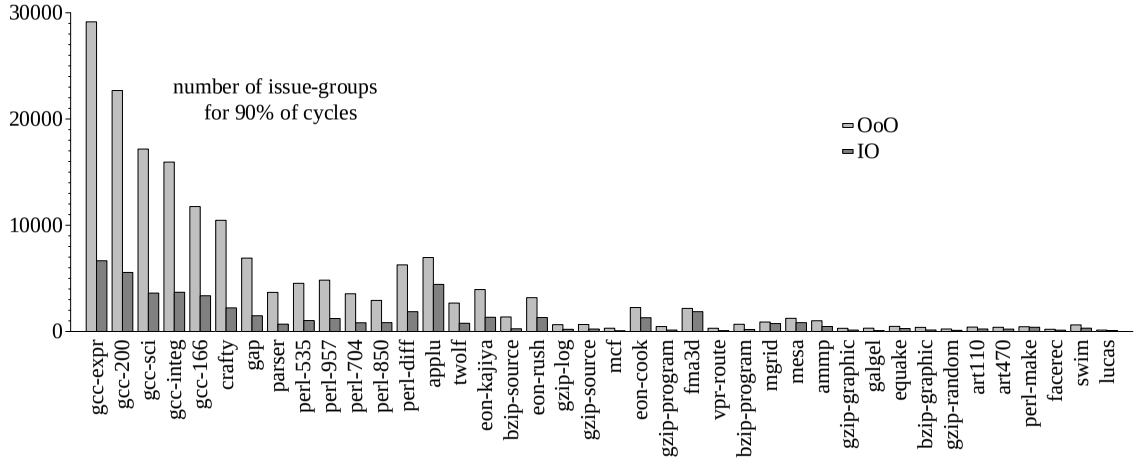


Figure 2.6: Number of unique issue-groups responsible of 90% of the total number of not-empty issue-groups created by the IO and the OoO processors.

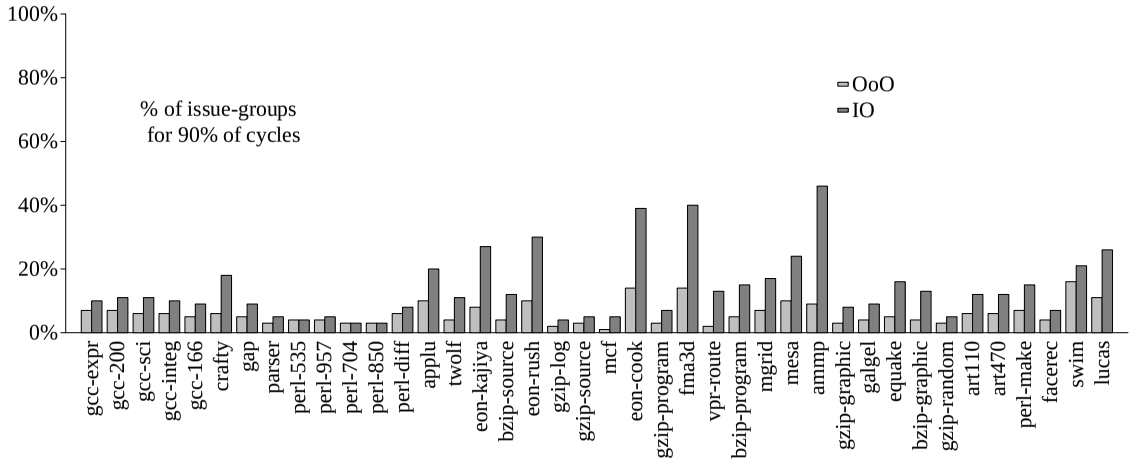


Figure 2.7: Percentage of unique issue-groups responsible of 90% of the total number of not-empty issue-groups created by the IO and the OoO processors.

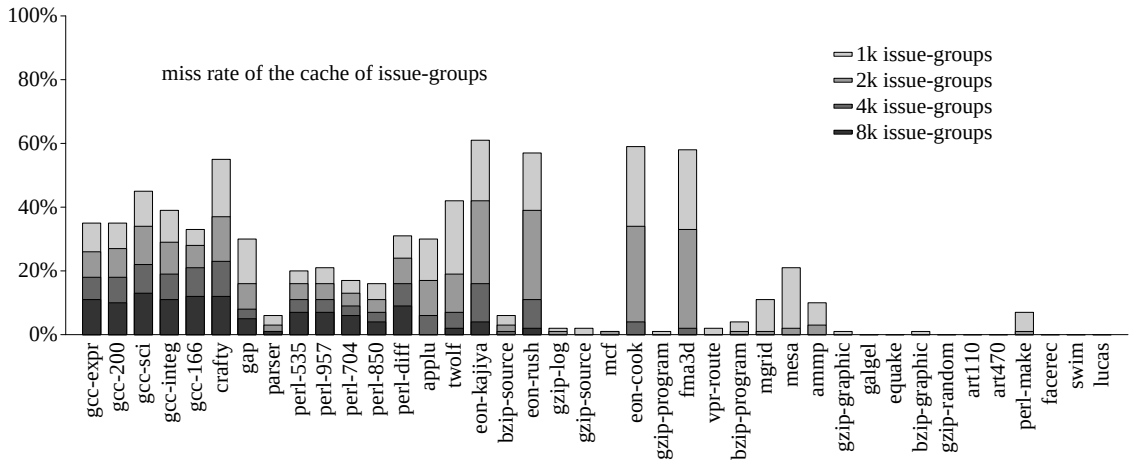


Figure 2.8: Miss rate of an issue-group cache for the OoO processor.

kind of prediction to decide which is the next issue-group to fetch and execute. Thus, the PC would be needed to check on the Commit stage whether the instruction is in the correct path and it should be actually executed. Therefore, the cache has to store 32 bits for the instruction and 46 bits for the PC, according to the Alpha 21264 parameters. With four integer and two floating point instructions per issue-group, that adds up to 468 bits per issue-group. The 4K-issue-group cache needs 234KB. The rest of configurations used in the experiments need between 58KB and 468KB.

Sequences or stand-alone issue-groups

The cache of issue-groups studied above stores the issue-groups individually. It is possible to design a processor that executes the issue-groups from such a cache. However, it yields much better results to store together sequences of issue-groups, that are always executed in the order that they are stored.

Using sequences of issue-groups simplifies two problems that are more difficult to solve in the stand-alone issue-group cache. First, the instructions can be renamed before they are stored in the cache, since dependences are well known within the scope of the sequence. Second, it is known which is the next issue-group to execute. The same issue-group can appear in several paths, so its successor can be different in each execution. With separated issue-groups, a predictor is needed to choose each cycle which is the next issue-group that will be executed. If sequences of issue-groups are used, this prediction is made when the issue-groups are created and not at execution time.

Another benefit is a reduction of overhead information. It is not required to store the PC of all instructions in the sequence but only the PC of the first instruction in the schedule and the predicted path of the branches included in it. As a drawback, there will be some redundant information when a given issue-group is stored several times in different sequences.

Taking all this into consideration, we have designed ReLaSch to work with sequences of issue-groups instead of stand-alone issue-groups.

Chapter 3

General description

This chapter presents the different elements of the ReLaSch processor. It is based on the 21264 Alpha processor. A high-level introduction to the ReLaSch processor and a brief description of some elements are followed by an explanation of the two operation modes of the ReLaSch processor. Finally, the new elements of the pipeline are introduced, highlighting how they cope with different issues such as dependences, memory aliasing or branch prediction.

3.1 The pipeline

The pipeline of the ReLaSch processor is shown in figure 3.1.c, along with the OoO (3.1.a) and IO (3.1.b) processors, already introduced in chapter 2. In ReLaSch, the Fetch, Decode and Map stages of the OoO processor (the Ifront-end) are coupled with additional logic to process the rgroups: Rfetch, Rdecode and Rmap (the Rfront-end). The Issue stage processes the instruction in-order just like the IO processor. Also, there is a new sequence of stages, the Rcreate logic. Besides, the Rcache is added to the processor.

This processor is based on schedule Reuse, so R identifies the schedules (rgroups) and the new elements of the pipeline. The instructions scheduled to be issued together in the same cycle are an issue-group. An rgroup is a sequence of issue-groups.

The Rcreate logic schedules committed instructions into rgroups. It applies a simple scheduling, makes predictions based on the last execution and partially renames the registers. The Rfetch logic accesses the Rcache and the Rdecode logic performs instruction decoding. The Rfront-end stages don't access the branch predictor since branch prediction is performed at scheduling time. The Rmap logic completes the renaming and inserts the instructions in the Reorder Buffer (ROB) and the issue buffer.

Out of the critical path

We have introduced new logic that performs the same tasks as the front-end of a conventional out-of-order processor: in this logic, the instructions are fetched, decoded, inserted in the ROB, their dependences analyzed, their registers renamed and selected for execution.

The aim is to perform as much as possible of those tasks out of the critical path. The Rcreate logic schedules the instructions and it is placed after the Commit stage, outside the critical path of execution of the instructions so its latency has little impact on performance, as will be shown in section 9.2.5. Also, scheduling is performed only when new rgroups are needed in order to reduce energy consumption. Therefore, the Rcreate logic can be more elaborated and complex than it would be if it had to schedule the instructions at execution time as conventional out-of-order processors do.

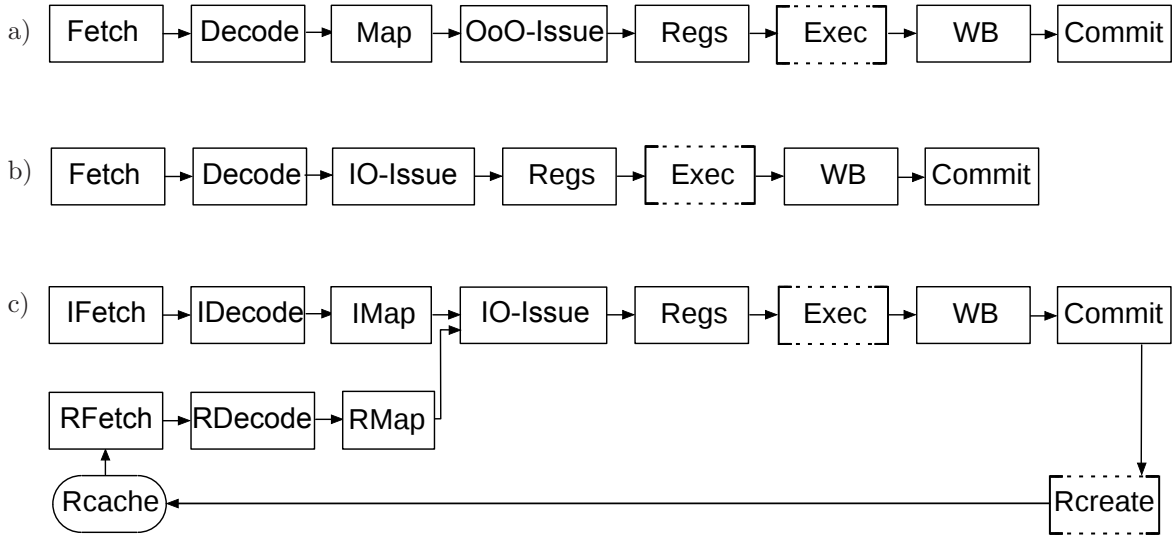


Figure 3.1: The pipeline of the (a) OoO, (b) IO and (c) ReLaSch processors.

On the contrary, the Rfront-end is part of the execution pipeline so its complexity can affect the cycle time of the processor. We must keep it as simple as possible. Besides, the Rfront-end processes the instructions just before they are issued, so any delay there implies further delays in execution and commit of the instructions that translates into performance degradation.

Therefore, the Rcreate logic makes all the decisions: it schedules instructions, renames registers, assigns identifiers in the queues, predicts the behavior of branches and memory instructions, etc. On its turn, the Rfront-end blindly processes in-order the instructions of the schedule: it simply reads and decodes the issue-groups and checks the availability of the resources and data needed by each instruction.

The Rfetch logic reads the rgroups from the Rcache and the rest of the Rfront-end processes them. The instructions in one issue-group are independent by construction, so the Rmap logic doesn't need to check if there is any dependence between them. It just has to check if the resources needed for each instruction are available. The resources have been already assigned by the Rcreate logic. These resources are the destination physical register, the entry in the ROB and the entry in the Load Queue (LQ) or the Store Queue (SQ), if it is processing a memory instruction.

Each Rcache line contains one rgroup. It stores information of hundreds of instructions, so an Rcache line is larger than usual cache-line sizes. However, it is not required to read an Rcache line in just one cycle. The rgroup is processed in-order, one issue-group per cycle at most, so the Rcache line can be read progressively, using a low bandwidth design.

Ensuring correctness

The Rmap logic processes the instructions in the order of the schedule. The instructions have been previously reordered by the Rcreate logic. The ROB is used to commit the instructions in-order. It provides support for precise interrupts and allows recovering from mispredictions. The Commit stage processes the instructions once they have completed their execution. Thus, an rgroup is not required to commit atomically. It may commit partially up to a mispredicted instruction. The identifier in the ROB of each instruction is assigned by the Rcreate logic and stored with the information of the schedule in the Rcache. Similarly, the LQ and the SQ allow to detect invalid reordering of memory

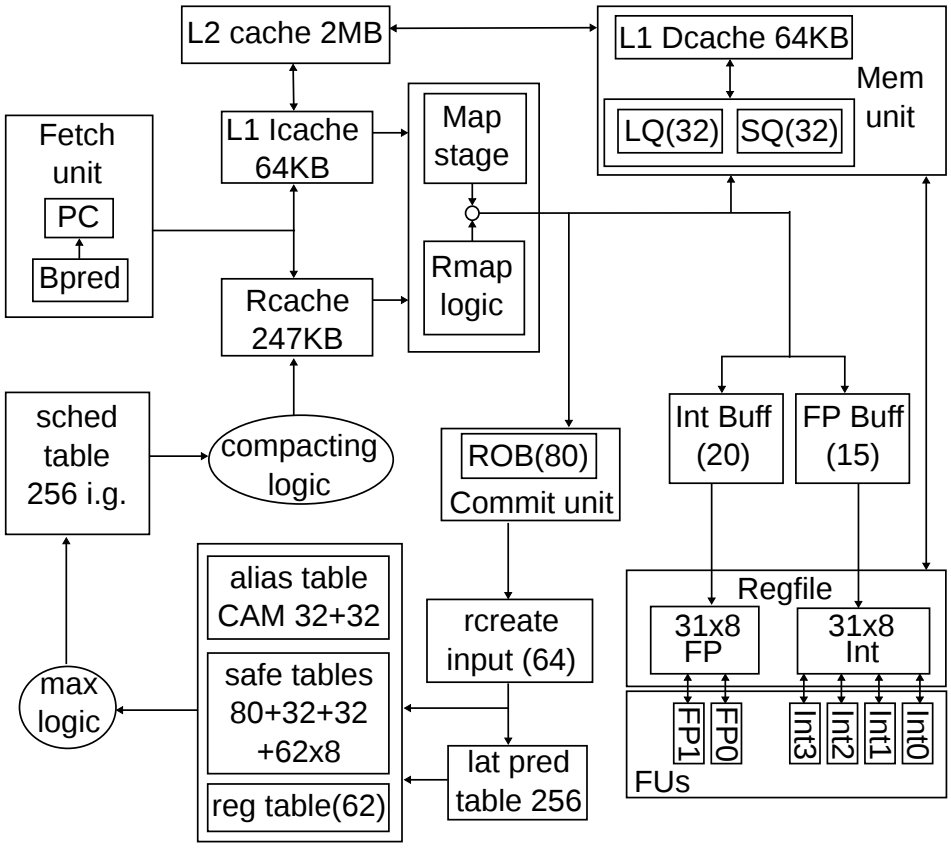


Figure 3.2: Block diagram of the ReLaSch processor.

instructions and enable bypassing data from a store to a load whenever it is possible. The identifier in these queues is also assigned by the Rcreate logic.

In addition to the tasks that the Commit stage performs in the OoO processor, in the ReLaSch processor it also accesses the Rcache in order to look for an appropriate schedule after a misprediction; besides, it notifies any misprediction in the rgroups to the Rcache and updates the branch predictor for the scheduled branches too. The committed instructions are sent to the Rcreate logic.

The register file

Unlike the reference OoO processor, that uses a shared pool of physical registers, ReLaSch uses a register file that has a separated set of physical registers for each logical register. It is based on the EC's register file [10]. We use such a register file because a ReLaSch processor modified to use a conventional register file has shown to yield a slightly lower IPC than the default configuration of ReLaSch. Besides that, register renaming is more complex with that register file, both at schedule and execution time. Furthermore, the Rcache requires storing more bits per instruction. Such alternative design is presented in detail in appendix A.

In the ReLaSch processor, the physical registers of a logical register are assigned sequentially and always in the same order by the Map stage or the Rcreate logic. Nevertheless, when the schedule is processed by the Rmap logic they are used out-of-order: in the order in which they appear as destination register in the schedule. The same physical register can be assigned multiple times within the same rgroup. Each physical register has two bits to indicate: a) whether it contains valid data; and b) whether the physical register is available as destination or it is busy.

Each logical register has a pointer that identifies which physical register currently stores its architectural value. The pointer is incremented when an instruction that writes that register commits.

3.2 Execution modes

The ReLaSch processor has two execution modes: a) the Icache mode, when the Ifront-end is used and the Rfront-end is inactive; and b) the Rcache mode, when the logic in the Rfront-end processes the instructions of an rgroup, stored in the Rcache, while the Ifront-end remains unused.

The more the processor is in the Rcache mode, the higher the IPC it achieves. So the processor is in the Rcache mode whenever it is possible, i.e. when a suitable rgroup is available. The initial mode of the processor is the Icache mode, since there is no available rgroup in the Rcache then.

The behavior of the Issue, Execute and Writeback stages is mode-independent. The Commit stage is mostly mode-independent but it updates the branch predictor for all branches executed in the Rcache mode.

Every cycle that the processor is in the Icache mode, the Fetch stage accesses the Icache and the Rcache in parallel with the current PC. The processor changes to the Rcache mode if the access hits in the Rcache, i.e. an rgroup begins in that PC.

In the Rcache mode, the Rfetch logic reads the content of the current rgroup. Once an rgroup has been completely fetched, another one is read from the Rcache if possible. Otherwise, the processor changes to the Icache mode.

Regardless of the mode, when the pipeline is flushed (on a branch misprediction or a memory order violation), the Rcache is accessed with the PC of the first instruction to be executed after the flush, either the outcome of the branch or the PC of the offending load. The processor executes the instructions in the Rcache mode on a hit and in the Icache mode otherwise.

3.2.1 Scheduler

The Rcreate logic can be either in the Schedule mode or in the Idle mode. It creates new rgroups only in the Schedule mode. The instructions executed in the Icache mode are always processed in the Schedule mode. When the processor changes to the Rcache mode, Rcreate changes to the Idle mode, but first it completes the current schedule. It changes back to the Schedule mode with an instruction executed in the Icache mode or to re-schedule the instructions of an rgroup that frequently aborts its execution (due to branch or memory-aliasing mispredictions). There is a saturating counter for each Rcache line to detect this kind of rgroups.

3.3 How does the ReLaSch processor work

3.3.1 Dependences and register renaming

The Rcreate logic places each instruction in the issue-group in which its source registers will be available at execution. Based on the information obtained from real execution, Rcreate knows the earliest that any instruction could have been scheduled. All the information needed to schedule an instruction is when are its sources and resources available. Knowing the latency of the operation, Rcreate can compute the earliest time the output of the current instruction will be available. The Rcreate logic schedules the committed instructions, taking into account their dependences, latencies, the FU usage and when the resources needed by the instruction are known to be free.

It also assigns the identifiers of the source and destination physical registers. These identifiers are stored with the instruction in the Rcache. A register that has not been used yet as destination within the rgroup (a live-in register) is assumed to be in the physical register 0 of its set. At execution time, the Rmap logic needs to adjust this renaming to the actual registers used by the previous rgroup, since the live-in registers may be not in physical register 0. So Rmap completes the renaming process at execution time, by simply adding an offset to the identifiers of the physical registers. Each logical register has its own offset. It is determined by the renaming of that logical register at the time the current rgroup began its execution. These offsets remain constant during all the execution of the rgroup so register renaming of an issue-group is completely independent of the renaming of the previous issue-groups in the rgroup. Therefore, the tight loop of the renaming logic that can be found in conventional out-of-order processors is not present in the ReLaSch processor.

The Rmap logic checks if the destination physical register is free, while the Issue stage is responsible for checking whether the source physical registers are ready. To know if the destination physical register is available, the Rmap logic checks its `busy` bit. The Issue logic checks the `valid` bit of the source physical registers.

3.3.2 Resource assignment

The Rcreate logic assigns the identifier in the ROB for each instruction in the rgroup, starting with identifier 0 for the first instruction. Similarly, each memory instruction gets an identifier in the LQ or the SQ. These identifiers are stored with the instruction in the Rcache.

The Rmap logic adapts the identifiers to the actual state of the ROB and queues using an offset. It also checks whether the desired identifiers are available. The Rmap logic checks the `busy` bit of the desired entry in the ROB and the LQ or the SQ. It stalls if any of the needed resources is not available. The Rmap logic also inserts the instructions in the issue buffer.

At scheduling time, Rcreate tracks the usage of the functional units in each issue-groups of the schedule. The Issue stage checks at execution time if the functional unit needed for each instruction is actually available.

To adjust the identifiers, the Rmap logic just needs to add an offset. Three different offsets are used, one for the ROB, one for the LQ and the last one for the SQ. The offset added to the identifiers is needed because they were assigned by Rcreate beginning with identifier 0, whereas the first identifier used at execution time should be that of the first available resource at the moment when the rgroup enters the Rmap logic. The identifier of that resource will be the offset during all the execution of the rgroup.

The Rcreate logic tracks in which issue-group it is safe to reuse each identifier in the ROB in order to avoid the possible deadlocks. Before assigning an identifier, the scheduler must take into account in which issue-group is scheduled the instruction that used it for the last time and in which issue-group the identifier is released. A deadlock could appear because the Rmap logic processes the instructions in the order of the schedule whereas the instructions commit in program order. The Rmap logic stalls when an instruction C needs a resource still assigned to an uncommitted instruction B. Any instruction A older than B in the program order must commit before B. If A is scheduled after C, the processor would enter a deadlock since Rmap stalls at C. The Rcreate logic tracks when are scheduled all the preceding instructions to avoid this situation. Section 4.2.3 explains this situation with more detail and our solution to avoid it.

3.3.3 Memory latency and aliasing prediction

In order to predict the latency of the memory instructions the Rcreate logic uses saturating counters. It schedules each load assuming either the L1 hit latency or the latency seen during the last execution. The counters capture the biased loads and the dependent instructions are scheduled accordingly.

The Rcreate logic uses the addresses accessed the last time to predict whether two memory instructions will alias at execution time. Unaliased instructions can be freely reordered, but two aliased instructions are scheduled to be issued maintaining their relative program order. As the SQ can usually bypass the data to an aliased load, maintaining the order at the Issue stage is usually enough.

However, bypassing is not possible in two cases: a) when the accesses have different sizes; and b) when there is a mismatch in the data type (i.e. floating point instead of integer). When any of these cases is detected, the Rcreate logic schedules the load instruction to be executed after the store commits. Besides, the identifier of the store in the SQ is copied with the information of the load in the Rcache. The Rmap logic adds the offset of the SQ to the identifier and the loads stalls at Issue until the aliased store has committed and it is safe to execute the load.

The Writeback stage checks if there has been any memory order violation. Whenever there is any mis-speculated load access, the Commit stage flushes the pipeline and execution restarts with the offending instruction.

3.3.4 Branch prediction

The Rcreate logic predicts at scheduling time that branches will follow at execution time the same path that they followed the last time. Each dynamic instance of a branch is predicted to repeat its most recent outcome. An rgroup can contain several copies of the same branch, each one predicted independently. It is not a one-bit history predictor and it can capture complex patterns since implicitly uses the whole rgroup as path.

Each conditional branch stored in the Rcache has a flag to indicate whether it is predicted as taken or not-taken. Besides, in each Rcache line there is a separated structure to store the target PC of the indirect branches of the rgroup. The number of this type of branches per rgroup is limited but less bits per rgroup are used in the Rcache.

The Rmap logic simply copies the taken flag of the conditional branches and the target PC of the indirect branches in the corresponding entry of the ROB. With this information and the result of the

execution, the Commit stage checks whether the prediction is correct just like it does for any branch executed in the Icache mode.

3.3.5 Bad rgroups

It may happen that the outcome of the branches and memory aliases captured by the Rcreate logic is actually not representative of the most common behavior of the instructions of a given rgroup and that the predictions made by Rcreate are not accurate. It is also possible that the program changes to another execution phase, with a change in the behavior of the control and memory instructions. In this situation, many rgroups repeatedly fail to be completely executed, either due to branch mispredictions or unexpected memory aliases. The high number of pipeline flushes will result in performance degradation.

These rgroups are detected and scheduled again in the Rcreate logic. The new schedule is expected to capture better the code behavior. To detect these cases, each rgroup stored in the Rcache has an associated saturating counter. The counter is increased when the rgroup is executed completely and all its instructions commit and decreased otherwise (when a branch or a memory alias was mispredicted and the uncommitted part of the rgroup must be flushed). It is also decreased if the rgroup completely committed but a load in the rgroup missed in the access to the L2 cache.

If a given counter becomes zero, its rgroup is marked to be rescheduled. The next time it is executed, its instructions carry a flag to notify the situation to the Rcreate logic, that will schedule a new rgroup from these instructions.

The Rmap logic marks the first instruction of each rgroup when inserts it in the ROB. The Commit stage uses this information to detect the beginning and the end of each rgroup and to access the Rcache to update the corresponding counter.

3.3.6 Rgroup identification

The PC of the first instruction of an rgroup is used to identify that rgroup in the Rcache. The PC is used to index the Rcache and also as the tag of the rgroup. The PC is required to match to consider the access as a hit in the Rcache. Besides this, the history bits of the branches on the path to the rgroup are used as a hint to decide between several rgroups with the same initial PC.

When the Rcreate logic closes an rgroup and stores it in the Rcache, it also includes the identifier of the next rgroup. It is used by the Rfetch logic to access the Rcache again once it has processed the rgroup. If an rgroup is executed completely, there is only one possible successor unless the last instruction is a branch. When the rgroup ends with a branch, the Rcreate logic predicts that the branch will behave just as at it did at execution time, as it does for all the other branches in the rgroup. The identifier of the next rgroup is formed by the next PC of the last instruction in the current rgroup and the history bits of the branches scheduled in it.

Once an rgroup has been processed, the Rfetch logic accesses the Rcache with this identifier to start fetching the next rgroup. Since the identifier is known in advance, the Rcache can be accessed before the current rgroup has been completely processed. Therefore, the latency to read the first chunk of data of the next rgroup can be overlapped with the last fetching cycles of the current rgroup.

3.3.7 Rules to close an rgroup

The Rcreate logic closes an rgroup under several conditions:

a. the maximum number of instructions per rgroup is reached;

b. due to its dependences or resource requirements, an instruction should be scheduled beyond the limit of the scheduling structure;

c. the maximum number of indirect branches is reached;

d. a trap instruction is scheduled; and

e. the buffer of committed instructions had an overflow and the next instruction to be scheduled was not committed immediately after the current one.

Once an rgroup has been closed, the Rcreate logic can either continue the scheduling process on a new rgroup or stop scheduling. If the next instruction to schedule was executed in the Icache mode, scheduling continues. If it was executed in the Rcache mode, scheduling continues just for one additional rgroup. After the new rgroup is closed, if the next instruction was executed also in the Rcache mode, Rcreate stops scheduling. The Rcreate logic starts scheduling again when a committed instruction was executed in the Icache mode or when the instructions of a "bad rgroup" are executed.

Chapter 4

The Rcreate logic

The Rcreate logic schedules the committed instructions into rgroups, trying to maximize performance while preserving the dependences and warranting a deadlock-free execution of the schedule. For each instruction, Rcreate also renames its registers and assigns the identifiers of the resources it uses at execution time: the identifier in the ROB and in the LQ or the SQ.

This chapter first introduces which structures the Rcreate logic uses to store the instructions and the schedule. Then it is explained how it creates a valid schedule of arithmetic instructions. This is followed by a description of the techniques used to improve the performance of the schedules. Then, the special treatment required by other types of instructions such as the memory or control instructions is examined. The policy for deciding when Rcreate must schedule and when the current rgroup must be closed is detailed afterwards. Finally, a block diagram of the Rcreate logic that summarizes this chapter is presented.

The Rcreate logic is pipelined in several stages. Since it is out of the critical path, the number of stages required has a low impact in the IPC achieved, as shown in section 9.2.5. Therefore, this chapter does not describe in detail how the Rcreate logic could be pipelined.

4.1 Storage structures

4.1.1 Rcreate_input buffer

Committed instructions are sent to the Rcreate logic, where they are copied into the `rcreate_input` buffer. This buffer is shown in figure 4.1. The instructions stored in this buffer are processed sequentially by the scheduler. The information stored in the buffer for each instruction is: the PC of the instruction, the PC of the next instruction, the encoded instruction, a flag to indicate whether the instruction was executed in the Rcache mode and another flag to know whether it is part of an rgroup that must be rescheduled. The conditional branches have a taken/not taken flag. The scheduler also needs the target PC of the indirect branches, but that it is already stored in the next PC field of the buffer. For memory instructions, it is also stored the address that was accessed, whether the access hit the cache and the latency of the access. Additional information such as the logical registers or the instruction type (load, store, conditional move, conditional and indirect branch) can be either stored in the buffer or decoded by the Rcreate logic.

The commit stage retires up to 11 instructions in a cycle in certain conditions, though this rate cannot be sustained. The scheduler assumes that two instructions that are consecutive in the `rcreate_input` buffer have committed sequentially and that there isn't any missing committed instruction. This is the normal case but buffer overflows must be taken into consideration. If the buffer is full when an instruction commits, the instructions already present in the buffer are not overwritten.

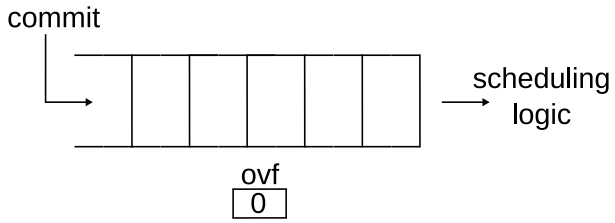


Figure 4.1: The committed instructions are stored in the `rcreate_input` buffer until they are scheduled. If the buffer is full when trying to insert a new instruction, the `ovf` flag is set.

Instead, the `ovf` is set to indicate that there has been an overflow. If this flag is set, all incoming committed instructions are dismissed and not inserted in the buffer, even if there is room for them. Otherwise, the assumption of the scheduler would be incorrect. Once Rcreate has scheduled all the instructions in the buffer it closes the current rgroup and clears the `ovf` flag. The instructions that commit from then on are allowed to be inserted in the buffer and will be later scheduled in a new rgroup.

Section 9.2.6 shows the impact in performance of the `rcreate_input` buffer. A buffer of 128 instructions is needed to eliminate all performance drops due to buffer overflows, even though most benchmarks perform well with a smaller buffer.

4.1.2 Sched table

The `sched` table stores the rgroup that is currently being created. It is shown in figure 4.2.a. It has one entry for each possible issue-group in the rgroup, 512 entries in our baseline configuration. As one issue-group corresponds to one cycle, an rgroup covers exactly as many cycles as issue-groups it can contain. The issue width of the processor determines the size of the issue-groups. In our baseline, it is four integer and two floating point instructions.

There are several fields for each instruction, as shown in figure 4.2.b. Besides the `valid` bit and the encoded instruction, for each instruction is stored its identifier in the ROB, the source and destination physical registers, the identifiers in the LQ and the SQ, whether a load is predicted to be aliased and whether a branch is predicted to be taken. Besides, each issue-group has some additional fields related with the usage of the functional units. These fields are explained in detail in the rest of this chapter.

The issue-groups in the `sched` table have identifiers 0, 1, etc. For each instruction, the Rcreate logic determines the identifier of the earliest issue-group in which an instruction can be scheduled, according to the dependences and latencies, the availability of the resources and the memory aliases. The `sched` table is accessed to store the instruction in its issue-group. When the current rgroup is closed, the whole content of the table is copied into one line of the Rcache.

4.2 Valid schedule of an arithmetic instruction

In order to schedule an arithmetic instruction correctly, we have to preserve dependences and assign resources properly. The case of the arithmetic instructions shows the common process applied to each instruction by the Rcreate logic. Other kinds of instructions require some additional processing by the scheduler, which is explained in sections 4.4, 4.5 and 4.6.

To simplify the explanation, the structures needed by the scheduler are introduced gradually. First, a basic approach is shown. Then, its weaknesses and problems are highlighted. Finally, a solution is presented.

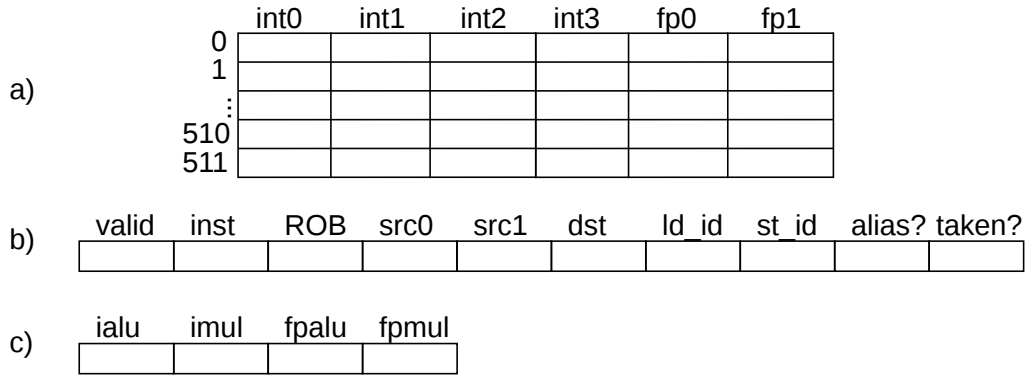


Figure 4.2: a) The `sched` table. b) The fields included in the slot for each instruction. c) Additional fields that are not included for each instruction but for each issue-group.

4.2.1 Dependences

The dependences of the instructions must be taken into account to create a correct schedule; that is, which physical registers are used and when are the operands ready. The Rcreate logic uses the `reg_info` table to record this information. The table is shown in figure 4.3. It has one entry for each logical register except for R31. In the Alpha ISA, this register always stores the value 0. Therefore, it is always available and it is never renamed. The table is indexed with the identifier of the source logical registers of the instruction. Each entry has two fields, `phy` and `read`. The `phy` field stores the identifier of the current physical register to which is renamed the logical register. This is the register that dependent instructions will read at execution time. The `read` field stores in which cycle at execution time the register is available as a source register. This cycle is indicated relative to the first cycle of execution of the rgroup. Both the `phy` field and the `read` field are initialized with value 0.

There is a 1:1 equivalence between the cycle in which an issue-group is executed and its identifier in the rgroup. Therefore, the maximum value of the `read` field of its source registers indicates in which issue-group should be scheduled the instruction, in order to not stall waiting for the source operand to be available.

Figure 4.3.a shows an example of how is the `reg_info` table used for the source registers. In the example, the table is accessed with the logical register 3 and the instruction is renamed to read the physical register 3 (R3.3), which is available in the issue-group 10. The second source operand is the logical register R2 and it is renamed to the physical register 1 (R2.1). Its `read` field is 7. Taking into account the information of both source registers, the instruction is scheduled in the issue-group 10.

Destination register

After a given instruction has been scheduled, the entry of its destination logical register in the `reg_info` table is updated. The identifier of the destination physical register is obtained from the value of the `phy` field plus one, modulo the size of the set of physical registers per logical register. The `read` field is updated with the cycle that corresponds to the issue-group in which the instruction has been scheduled plus its latency.

The example in figure 4.3.b shows that the destination logical register R1 is renamed to the physical register 0, assuming that the register file has sets of four physical registers (our baseline uses sets of eight physical registers). Since the instruction is scheduled in the issue-group 10 and assuming a latency of three cycles, the `read` field of the register R1 is updated to 13.

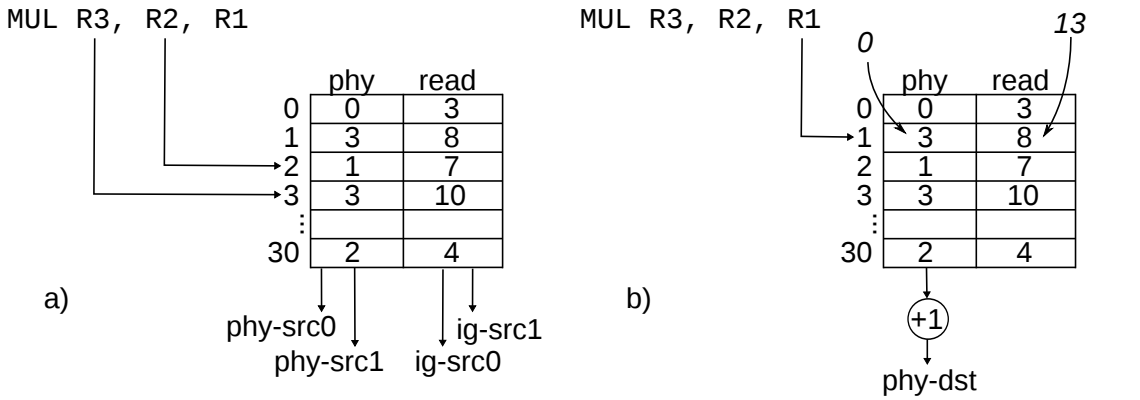


Figure 4.3: The `reg_info` table is used to rename the registers of the scheduled instruction and to know in which issue-group its source registers are available. The identifier of the source and destination logical registers is used to access the table. a) shows a read access to the table to the entries of the source registers. b) shows a read and write access to the entry of the destination register. The rightmost operand is the destination register.

The scheduler has to take into account in which cycle a physical register will be read for the last time. Otherwise, it could be used again as destination and be overwritten too early, leading to an invalid processor's state when the rgroup is executed.

Figure 4.4 illustrates the problem scheduling the example code of figure 4.5. It shows the contents of the `reg_info` table and the `sched` table after scheduling each instruction. To simplify the example, each logical register has a set of two physical registers. Also, all registers not explicitly written in the code are available at cycle 0. The ADD and MUL instructions have latencies of one and three cycles respectively.

The instruction A is scheduled in the issue-group 0 since its two operands are available then. Both operands are stored in the physical register 0 (R4.0 and R3.0). The `phy` field of its destination (R2) is incremented and its `read` field becomes 1 (issue-group 0 plus one cycle of latency). The instruction B has one operand (R2) that is ready at cycle 1 and the other (R1) at cycle 0. Therefore, it is scheduled in the issue-group 1. The `phy` field of register R0 is incremented and the value 4 ($1 + 3$ from the issue-group 1 and the 3 cycles of latency) is stored in the `read` field. When the instruction C accesses the `reg_info` table it finds that R2 is available at cycle 1 and R0 at cycle 4. So C is scheduled in the issue-group 4. The `read` field of the register R0 is updated to 7 ($4+3$) and the `phy` field is incremented, modulo 2. The instruction D can be scheduled in the issue-group 0 since all its source registers are available then. Finally, the instruction E is scheduled in the issue-group 1, when R2.0 is available.

Note that this is not a correct schedule, because E uses R2.1 as destination, just like the instruction A. Thus, at execution time the instruction C reads the value written by the instruction E in R2.1, instead of the result of the instruction A. Although not shown in the example, it would be possible to schedule an instruction even before the previous instruction that writes the same physical register (that is, to schedule E before A in the example).

The scheduler has to remember when each resource was used for the last time. By now, this problem can be solved with an additional `last_use` field for each physical register in the `reg_info` table. Each time a physical register is used as source or as destination, its `last_use` value is updated with the issue-group's identifier plus one, if this value is greater than the current value of `last_use`. Thus, the `last_use` field indicates in which issue-group is used that physical register file for the last time. The increment ensures that the register is read first and written afterwards. When an instruction is scheduled, the `last_use` field of its destination physical register is read; the instruction

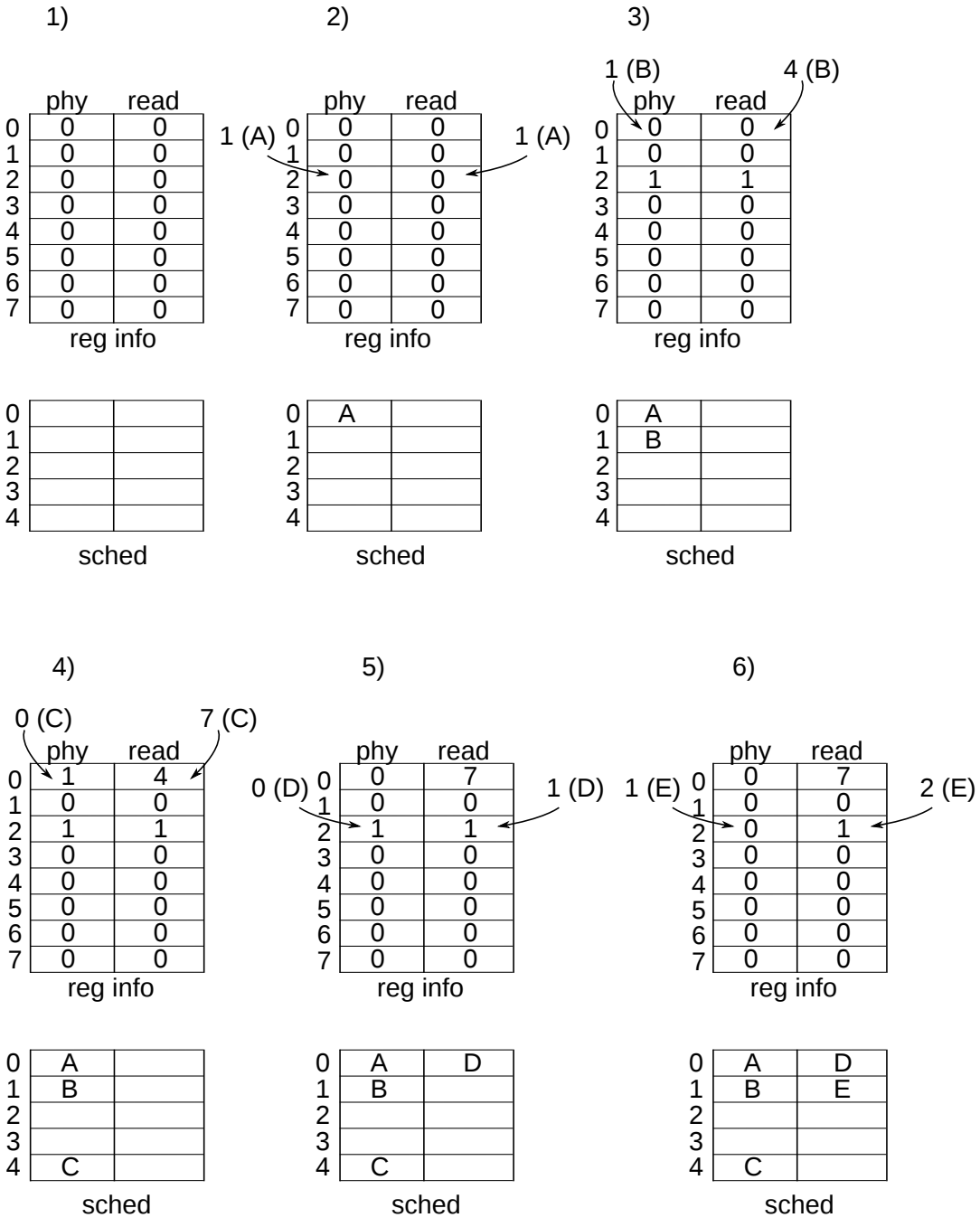


Figure 4.4: Evolution of the content of the `reg_info` and the `sched` tables throughout the scheduling process. ADD instructions have a latency of one cycle and MUL a latency of three cycles.

Example code:	Renamed code:
`A: ADD R4, R3, R2`	`A: ADD R4.0, R3.0, R2.1`
`B: MUL R2, R1, R0`	`B: MUL R2.1, R1.0, R0.1`
`C: MUL R2, R0, R0`	`C: MUL R2.1, R0.1, R0.0`
`D: ADD R5, R6, R2`	`D: ADD R5.0, R6.0, R2.0`
`E: ADD R2, R7, R2`	`E: ADD R2.0, R7.0, R2.1`

Figure 4.5: Example code, before and after renaming.

will be scheduled at least in the issue-group indicated by that field.

Figure 4.6 shows the same example of figure 4.4 with the additional field `last_use`. Note that the resulting schedule is exactly the same as before except for the instruction E, the offending instruction in figure 4.4. When the instruction E is scheduled, the `last_use` field of its destination register (R2.1) is equal to 5. So the instruction is scheduled in issue-group 5, even though its source registers are already available earlier. Thus the instruction C reads the proper content of register R2.1, preserving the dependence between instructions A and C.

Nevertheless, there are other problems related with the reuse of the physical registers that imposes additional restrictions to the scheduler. These problems are exposed and definitively solved in section 4.2.3 with the `safe` field that replaces the `last_use` field.

4.2.2 Reorder Buffer

Each instruction has an identifier in the ROB. This allows that the instructions commit in-order even though they are scheduled out-of-order in the rgroup. The number of instructions that the ROB can store is indicated by *ROB_size*. The Rcreate logic has the `ROB_id` register, that indicates which identifier in the ROB will be assigned to the next scheduled instruction. It is initialized with value 0. For each scheduled instruction, the register is incremented, modulo *ROB_size*. The identifier is stored with the rest of information of the instruction in the `sched` table.

The scheduler has to enforce that two instruction that have the same identifier in the ROB are not reordered in the `sched` table. It could be done with a table that stores the last use of each ROB identifier (`ROB_last_use`). An instruction would not be scheduled before the `ROB_last_use` issue-group of its ROB identifier. The `ROB_last_use` table would be updated with the issue-group's identifier of the scheduled instruction plus one (in order to avoid a race for the resource). However, the `ROB_last_use` table is not enough to create a correct schedule in all cases as it is shown in the section below.

4.2.3 Deadlocks

Reorder Buffer

The scheduler is not able to create correct schedules in all cases just using the information of the `ROB_last_use` table. In particular, the processor can enter into a deadlock when a wrongly-formed rgroup is executed. In this section, the code is first scheduled according to the dependences and the `ROB_last_use` table, then it is shown how the execution of the resulting schedule results in a deadlock. Finally, a solution is presented.

Scheduling The possibility of having deadlocks is illustrated with the example of figure 4.7. In the example we assume a four-entry ROB for simplicity. The identifiers are assigned in order, starting from 0. The identifier in the ROB for each instruction is indicated on the right of the renamed code. The ADD and MUL instructions have one and three cycles of latency respectively. The instruction B

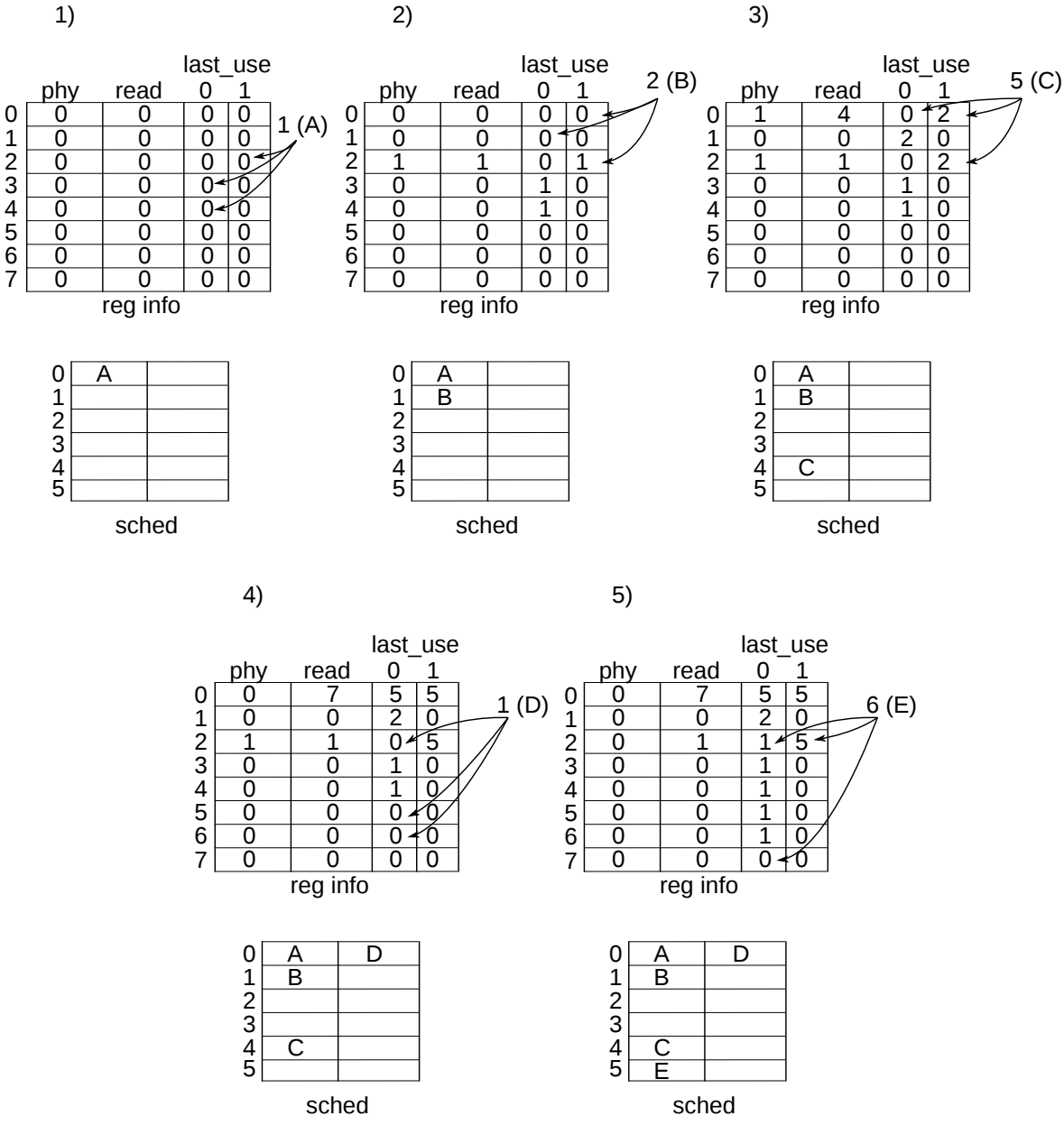


Figure 4.6: Evolution of the content of the `reg_info` table (with `last_use` fields for each physical register) and the `sched` table when the instructions in the example of figure 4.4 are scheduled. ADD instructions have a latency of one cycle and MUL a latency of three cycles.

Example code:

```
A: MUL R4, R3, R2
B: ADD R2, R1, R0
C: ADD R3, R1, R5
D: ADD R5, R6, R6
E: ADD R0, R7, R7
F: ADD R6, R1, R1
G: ADD R3, R4, R4
```

Renamed code and identifier in the ROB:

```
A: MUL R4.0, R3.0, R2.1; ROB[0]
B: ADD R2.1, R1.0, R0.1; ROB[1]
C: ADD R3.0, R1.0, R5.1; ROB[2]
D: ADD R5.1, R6.0, R6.1; ROB[3]
E: ADD R0.1, R7.0, R7.1; ROB[0]
F: ADD R6.1, R1.0, R1.1; ROB[1]
G: ADD R3.0, R4.0, R4.1; ROB[2]
```

Figure 4.7: Example of code that creates a deadlock, before and after register renaming.

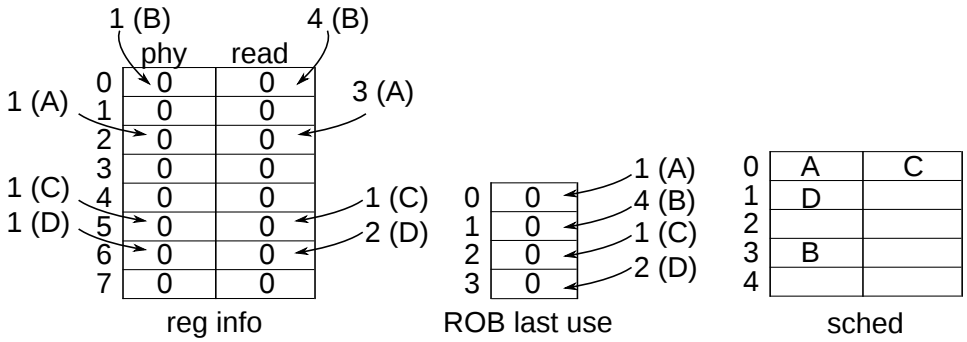


Figure 4.8: The state of the `reg_info`, `ROB_last_use` and `sched` tables after the instructions A to D are scheduled.

depends on the instruction A, the instruction D depends on C, the instruction E depends on B and the instruction F depends on D. The rest of instructions are independent.

Figure 4.8 shows the state of the `reg_info` table, the `ROB_last_use` table and the `sched` table after the first four instructions have been scheduled. The `last_use` fields of the `reg_info` table are not shown to simplify the figure since they don't have any effect in the resulting schedule in this case. Initially all fields have the value 0.

The instructions A and C are scheduled in issue-group 0, since their sources are available at cycle 0 and it is the first use of their `ROB_id`. The entries 0 and 2 of the `ROB_last_use` table are updated to 1. The instruction B is scheduled in the issue-group 3, in which the source register R2.1 will be available. The instruction D is placed in the issue-group 1, because it depends on R6.1. Their entries (1 and 3) in the `ROB_last_use` table are also updated, to store the values 4 and 2 respectively.

Figure 4.9 shows the state of the `reg_info` table and the `ROB_last_use` table after scheduling the rest of instructions. The instruction E uses the identifier 0 in the ROB, which was assigned previously to the instruction A, but must also wait for R0.1, that is available in the issue-group 4. Therefore, E is scheduled in the issue-group 4. Similarly, although the register R6.1 is available in the issue-group 2, the instruction F is scheduled in the issue-group 4, after the previous use of the ROB identifier 1 by the instruction B. Finally, the instruction G, that has all its sources available in the issue-group 0, would be scheduled in issue-group 1, since it reuses the ROB identifier of the instruction C and is independent with respect to the rest of instruction in the schedule.

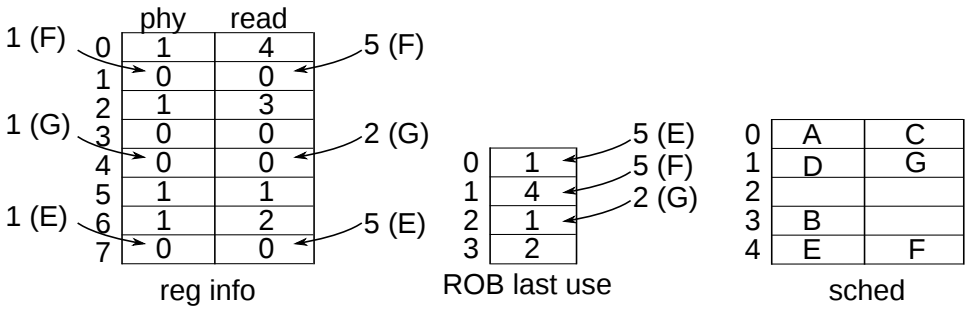


Figure 4.9: The state of the `reg_info`, `ROB_last_use` and `sched` tables after the instructions E to G are scheduled.

Execution The resulting schedule is not correct because G is placed before B. Let's see what happens at execution time. At cycle 0, both instructions A and C are processed by the Rmap logic and occupy the identifiers 0 and 2 in the ROB. At cycle 1, the instruction D is executed. However, the instruction G cannot be sent to issue: its identifier in the ROB (2) is already in use by the instruction C. Therefore, G stalls until its identifier in the ROB is freed. The instruction A can commit when it completes its execution. On the contrary, the instruction C must wait until B first enters the ROB and then commits, because instructions must commit in order. Unfortunately, the stall of the instruction G will prevent the instruction B to be executed. Since B will never commit, C will not be able to commit either.

Solution

In order to remove the deadlock in the example above, the instruction G must be scheduled after the instruction B. In general, an instruction I that reuses a given resource R must be scheduled not only after the previous instruction P that used R for the last time, but also after all instructions older than P. Since instructions are scheduled in-order, this can be accomplished by scheduling I after all the instructions already present in the `sched` table when P was scheduled. We will track this information for all the resources that can create a deadlock: the identifiers in the ROB, LQ, SQ and the physical registers. In the example, the instruction G must be placed after any instruction present in the rgroup when the instruction C is scheduled, including C itself.

To implement this new restriction, we introduce a new element in the Rcreate logic, the `safe` register. We also replace the `ROB_last_use` table with the `ROB_safe` table. Both new elements are shown in figure 4.10. Each identifier in the ROB has an entry in this table. Both the `safe` register and the entries of the `ROB_safe` table are initialized with 0. The `safe` register indicates which is the issue-group with a higher identifier that is not empty (there is at least one instruction scheduled in it). When an instruction is scheduled in a given issue-group *ig*, the `safe` register is updated if *ig+1* is greater than the current value of the `safe` register. The `safe` register is not used directly to schedule the instructions but only to update the `ROB_safe` table.

The scheduler uses the `ROB_safe` table just like it used the `ROB_last_use` table. The identifier in the ROB of the instruction is used to index the table and the value stored there is used to schedule the instruction. Figure 4.11 shows an example. There, the instruction currently being scheduled has the identifier 3 in the ROB and must be scheduled in the issue-group 10 or later. The table is updated for each scheduled instruction. The updated value of the `safe` register is copied into the `ROB_safe` entry that corresponds to the instruction's identifier in the ROB.

We show now how the same example code of the figure 4.7 is scheduled with the `safe` register and the `ROB_safe` table. Figure 4.12 shows the state of the `reg_info` table, the `ROB_safe` table and the

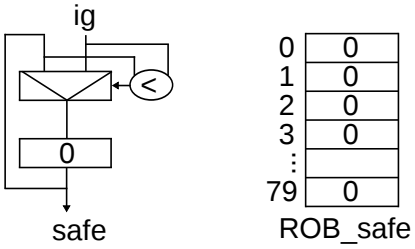


Figure 4.10: The `safe` register and the `ROB_safe` table. `ig` indicates the issue-group in which the current instruction has been scheduled.

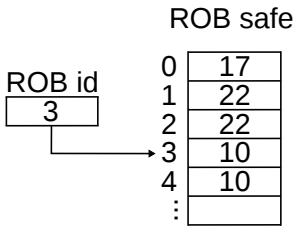


Figure 4.11: The `ROB_id` register is used to access the `ROB_safe` table.

`sched` table when the instructions A, B, C and D are scheduled. The content of the `reg_info` table is the same as in figure 4.8. After the instruction A is scheduled, the `safe` register takes the value 1 and so does `ROB_safe` entry 0. After the instruction B is scheduled, the `safe` register takes the value 4, as well as the `ROB_safe` entry 1. The instruction C does not modify the `safe` register but the `ROB_safe` entry 2 is updated to 4, which is the value already stored in the `safe` register. Similarly, with the instruction D, the `ROB_safe` entry 3 takes the value 4. All these instructions have found a 0 in their `ROB_safe` entries, so the new table has no influence in the scheduling of these instructions.

Figure 4.13 shows the state of the same tables when the instructions E to G are scheduled. The instruction E finds the value 1 in the `ROB_safe` entry 0. However, the dependences are more restrictive and it is scheduled in the issue-group 4. It updates the `safe` register and the `ROB_safe` entry 0 with the value 5. On the contrary, although the instruction F has its sources available at cycle 2, the `ROB_safe` entry 1 has the value 4, so it is scheduled in the issue-group 4. The `ROB_safe` entry 1 is updated with the value 5. Finally, the instruction G reads the value 4 in the `ROB_safe` entry 2, so it should be scheduled in issue-group 4. However, this issue-group is already full with the instructions E and F. Therefore, G is scheduled in the issue-group 5 and both the `safe` register and the `ROB_safe` entry 2 are updated to 6. Section 4.2.4 explains in detail how the scheduler deals with full issue-groups.

The only difference of the new schedule is the issue-group in which the instruction G is scheduled but that change is enough to remove the deadlock. When the rgroup is executed it does not stall at the issue-group 1, so some cycles later the instruction B can be executed and commit too, which finally allows the instruction C to commit on its turn. Eventually, the instruction G finds its ROB entry available and can be processed by the Rmap logic.

Register file

There is the same problem with the reuse of physical registers as with the identifiers in the ROB because registers are also freed in-order by the Commit stage. The problem has been solved using the `reg_safe` fields. There is one `reg_safe` field for each physical register. However, there is a slight difference with the reuse of identifiers in the ROB. While a ROB entry can be assigned again just

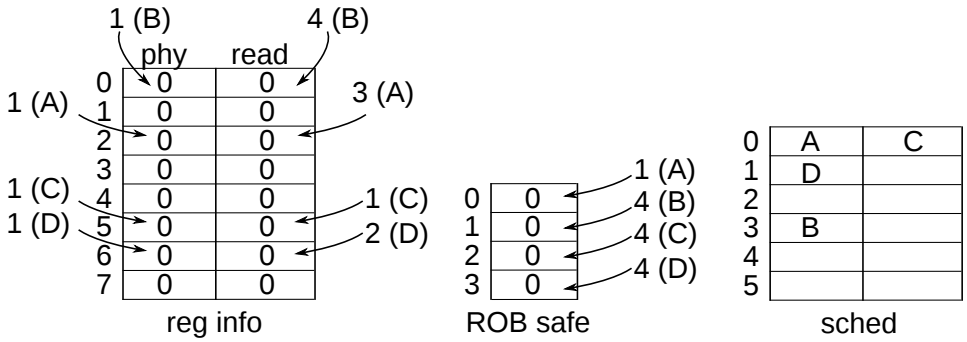


Figure 4.12: The state of the `reg_info` and `ROB_safe` tables after instructions A to D are scheduled.

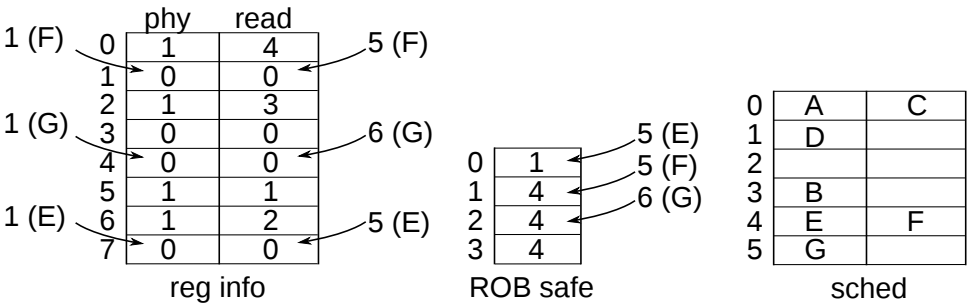


Figure 4.13: The state of the `reg_info` and `ROB_safe` tables after instructions E to G are scheduled.

after the last instruction that used it has committed, a given physical register can only be assigned again as destination when it does not hold the architectural value; i.e., when the next instruction that writes the same logical register commits. For example, the physical register 2 can be assigned again only after the instruction that writes the physical register 3 commits. If the physical register 2 had already been used as destination, it could happen that the physical register 2 was written but that the instruction that had written the physical register 3 didn't commit due to a miss-speculation. Then the architectural value would be lost and the processor's state after recovering from the miss would be invalid.

Figure 4.14 shows the new `reg_info` table, augmented with one field per physical register to indicate its `safe` value. These fields are called `reg_safe` though we use just "`safe`" in the figure for simplicity. In the figure, we assume that each logical register has a set of eight physical registers. Thus, the fields `safe` 0 to `safe` 7 correspond to the physical registers 0 to 7. The `safe` fields substitute the `last_use` fields in the `reg_info` table that were mentioned in section 4.2.1. All the `safe` fields are initialized with 0s. For each instruction, Rcreate assigns the physical *phy+1* register of its destination logical register, obtained from the value of the `phy` field of its logical register entry in the `reg_info` table. Therefore, it reads the `reg_safe` entry of the physical register *phy+1* to schedule the instruction. Once it has scheduled the instruction, it updates the `reg_safe` entry of the physical register *phy* with the value of the `safe` register, to correctly schedule the next instruction that reuses the physical register. Note that we only to consider the destination register here. For the source registers, the `read` field of the `reg_info` table is enough to ensure the correctness of the schedule.

Figure 4.16 shows how the code of figure 4.15 is scheduled according to the `reg_safe` fields. Each logical register has a set of two physical registers to simplify the example. The only dependent instruction is C, that reads the value written by B. Three instructions write the register R0. Initially,

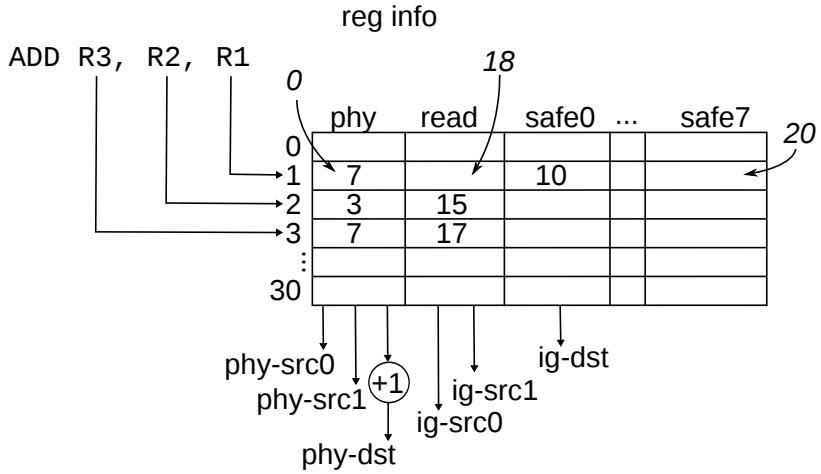


Figure 4.14: The `reg_safe` table is used to avoid deadlocks when assigning a physical register.

Example code	Renamed code
`A: ADD R0, R2, R0`	`A: ADD R0.0, R2.0, R0.1`
`B: MUL R2, R3, R1`	`B: MUL R2.0, R3.0, R1.1`
`C: ADD R1, R3, R3`	`C: ADD R1.1, R3.0, R3.1`
`D: ADD R7, R4, R0`	`D: ADD R7.0, R4.0, R0.0`
`E: ADD R3, R5, R0`	`E: ADD R3.0, R5.0, R0.1`

Figure 4.15: Example code for the use of the `reg_safe` table. ADD instructions have a latency of one cycle and MUL a latency of three cycles.

all `reg_safe` fields contains zeros.

The instruction A is scheduled in the issue-group 0 so the `safe` register is updated with the value 1. The `reg_safe` field of the register R0.0 is updated with the new value of the `safe` register (1). The instruction B can be scheduled in the issue-group 0 too. The `safe` register does not change, so the `reg_safe` field of the register R1.0 takes the value 1.

The instruction C is scheduled in the issue-group 3, in which its source register R1.1 is available. Thus, the value of the `safe` register is then 4 and the `reg_safe` field of the register R3.0 is also updated to 4. The source operands of the instruction D are available in the issue-group 0 but the `reg_safe` field of its destination physical register (R0.0) is 1. Therefore, it is scheduled in the issue-group 1. It updates the `reg_safe` field of the register R0.1 with the value of the `safe` register (4). The instruction E has its operands available in cycle 0 but the `reg_safe` field of R0.1 forces the scheduler to place it in the issue-group 4. The new value of both the `safe` register and the `reg_safe` entry for the register R0.0 is 5.

If the `safe` values had not been taken into account, the instruction E would have been scheduled before C, since the last use of its destination physical register is in the issue-group 0. So at execution time, the instruction D would not commit before the instruction C and the architectural value of the register R0 would still be in R0.1, which could not be assigned to E. Since the instruction E would stall, the instruction C would never be processed and the processor would enter into a deadlock. The actual schedule avoids this problem and is deadlock-free.

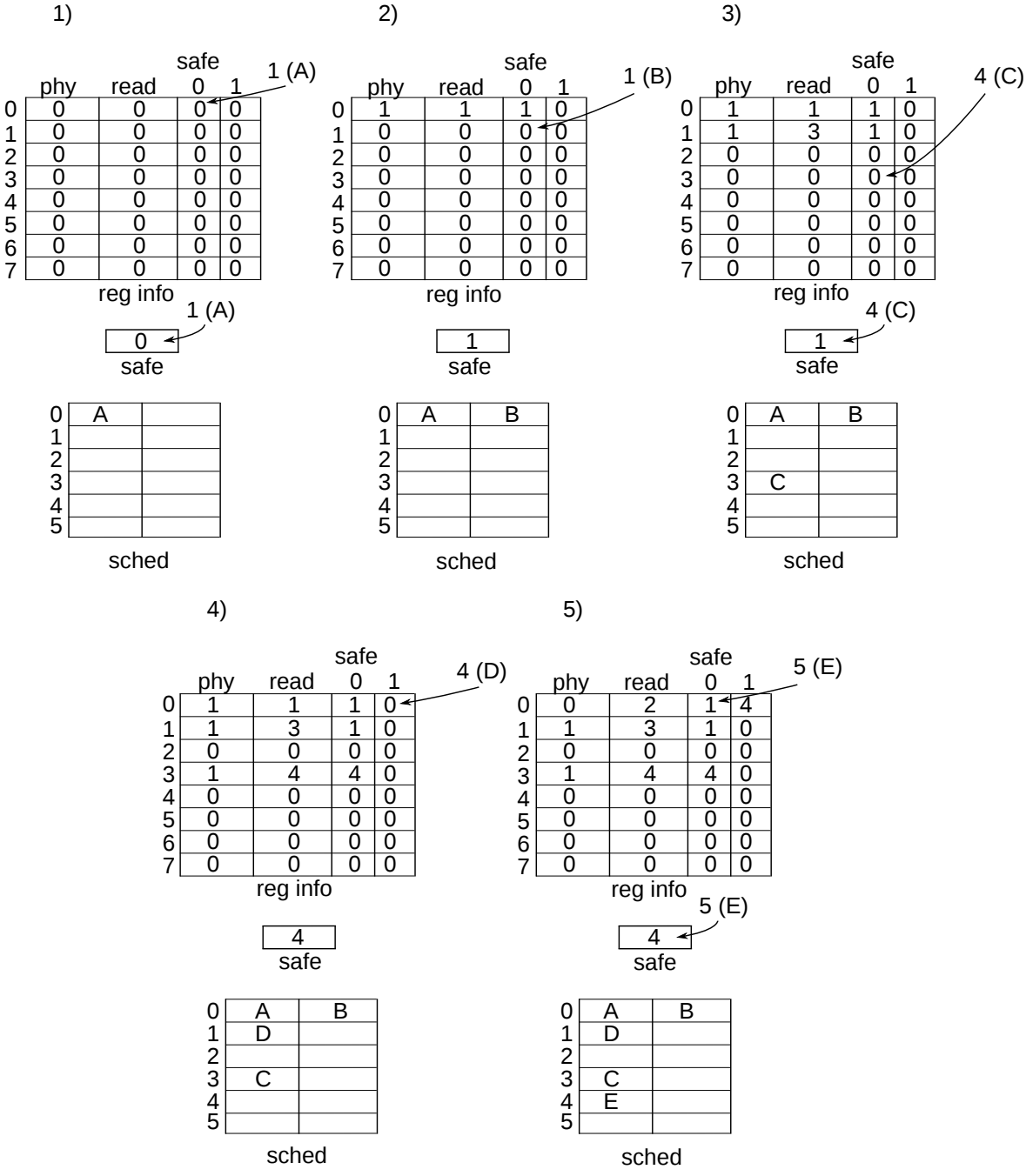


Figure 4.16: Evolution of the content of the `reg_info` table with `reg_safe` fields, the `safe` register and the `sched` table.

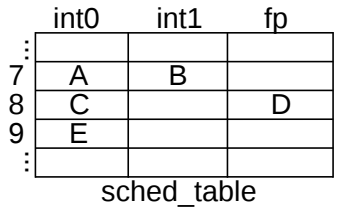


Figure 4.17: An example of how the Rcreate logic tracks whether there is room in the issue-groups.

4.2.4 Issue-groups

The maximum size of the issue-groups is fixed and is equal to the width of the issue logic. In our default configuration it can contain up to four integer and two floating point instructions. Therefore, it can happen that there is no room for an instruction in an issue-group *ig*, the earliest in which the instruction can be scheduled according to the `read` and `safe` fields. In that case, the scheduler tries to insert the instruction in the issue-group *ig+1*. If it is full too, the scheduler tries to insert the instruction in the issue-groups *ig+2*, *ig+3*, etc. consecutively. If all the issue-groups between *ig* and the end of the `sched` table are full, the instruction cannot be scheduled in the rgroup and the Rcreate logic closes the rgroup, as explained below in section 4.7.

Figure 4.17 shows an example, in which each issue-group can accommodate up to two integer and one floating point instructions. We assume that the current instruction must be scheduled in the issue-group 7, due to the values found in its `read` and `safe` fields and that it is an integer instruction. The instruction is actually scheduled in the issue-group 8 instead, since the issue-group 7 cannot hold more integer instructions. A floating point instruction cannot be scheduled in the issue-group 8, but it would be placed in the issue-group 9.

Most instructions are inserted in just one of the buffers (integer or floating point). However, some instructions are inserted in both buffers, for example the floating point memory instructions. In that case, the instruction is inserted just once in the rgroup but it occupies both entries of the issue-group. Therefore, it requires that its issue-group has room for one integer and one floating-point instruction. For instance, if we schedule a floating point store instruction over the schedule of the figure 4.17, it cannot be scheduled in the issue-group 7 because there is no place for an integer instruction there. It also cannot be scheduled in the issue-group 8 because there is already one floating point instruction. Thus, it would be scheduled in 9. No other instruction could be scheduled in the issue-group 9 thereafter.

4.3 Improving the performance of the schedule

The sections above have introduced the structures and methods used to create a valid and safe schedule. The dependences are taken into account and the reordered use of the resources never leads to a deadlock. However, there are several micro-architectural details that have been ignored so far that can make the rgroup stall often at execution time. This section explains how to minimize the stalls by improving the schedule.

4.3.1 Functional Units

One basic resource that we have deliberately ignored so far are the Functional Units (FU), where the instructions are executed. A given instruction type can be sent to one or more FUs and one FU can accept one or more instruction types. Each instruction type has an execution latency (the number of cycles needed to produce a result) and an issue latency (the number of cycles between

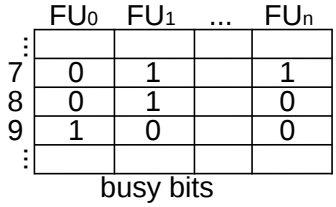


Figure 4.18: The `busy` bits for the different functional units in each issue-group.

two initializations, that is, between two instructions enter the same FU). The execution latency has already been used to update the `reg_info` table and schedule the dependent instructions.

If an instruction is scheduled in an issue-group in which it doesn't find an available FU at execution time, then it stalls until the required FU is free. To eliminate this stall, the Rcreate logic uses the `busy` bits, that are associated with each FU and with each issue-group of the `sched` table. Figure 4.18 shows an example. All the `busy` bits are initialized with 0s. The instruction type determines which bits must be checked. When the scheduler determines that an instruction should be scheduled at least in the issue-group *ig*, the `busy` bit of its FU in the entry *ig* of the `sched` table is accessed. If the bit is set, the scheduler tries to place the instruction in the issue-group *ig+1* instead and checks the corresponding `busy` bit. If that bit is set too, Rcreate tries to schedule the instruction in the issue-groups *ig+2*, *ig+3*, etc. If the instruction cannot be scheduled in any issue-group, the rgroup is closed, as explained in section 4.7. When the definitive issue-group *dig* is selected for the instruction, the corresponding `busy` bit of its FU is set in the entry *dig* of the `sched` table.

If the issue latency of the instruction is greater than one, the bits for the issue-groups from *ig* to *ig+issue latency-1* must be checked. Additionally, the corresponding `busy` bits are set when the instruction is scheduled. In the default configuration used in our experiments, most of the instructions have an issue latency of one cycle. Just the DIV and SQRT instruction types have a different value, 9 and 15 cycles respectively when using single precision and 12 and 30 with double precision. These latencies are chosen based on the documentation of the Alpha 21264 processor [15]. The scheduler accesses the `busy` bits of more than one issue-group only when it processes any of these instructions. A sequential approach that reads one bit per cycle can be used.

If an instruction can be scheduled in more than a single FU, the `busy` bits of all the possible FUs are checked. If the number of FUs where an an instruction can be executed is equal to the width of the issue-group, it is not necessary to have bits for them (assuming their issue latency is just one cycle). Our default configuration has four integer arithmetic and logic units with just one cycle of issue latency. Since the processor can issue up to four integer instructions per cycle, there is no need to have bits for these FUs in the `sched` table.

4.3.2 Safe_pos values

The `safe` register and its related tables have been introduced to keep track of the earliest issue-group in which it is safe to reuse a resource. In section 4.2.3, when the `safe` register has the value *ig+1*, it indicates that the issue-group *ig* is the highest issue-group where any instruction has been scheduled so far.

At execution time, an instruction scheduled in the issue-group *ig* will need several cycles to execute and commit. Therefore, an instruction that reuses a given resource (an identifier in the ROB, for example) that is scheduled in *ig+1* will be executed correctly and will be able to commit, but only after stalling for several cycles. These cycles of stall can be removed if the `safe` register and tables are updated taking into account the cycles that the instruction needs to go through the pipeline and commit.

Pipeline depth

When the processor is in the Rcache mode, an instruction goes through the following sequence of stages: the three Rfront-end stages (corresponding to the Rfetch, Rdecode and Rmap logic), and the Issue, Read, Execution (in the FU, where the instruction stays the cycles indicated by the execution latency), Writeback and Commit stages. The identifiers in the ROB, load queue and store queue, as well as the physical registers are freed in the Commit stage. All these identifiers are assigned by the Rmap logic. Assuming that a ROB entry that is freed by the Commit stage can be assigned again to a new instruction in the same cycle, a safe and stall-free schedule is created if the `safe` register is updated with *ig+execution latency+3*. The execution latency is that of the instruction scheduled at the issue-group *ig*.

Note that with this change, the `safe` register may be determined by an instruction i other than the instruction j scheduled in the highest occupied issue-group. This happens if the execution latency of i is greater than the latency of j.

Commit width

With the modification proposed in the section above, the `safe` register points to the highest issue-group in which an instruction in the rgroup finishes execution and commits. Such instruction may be a different instruction that the one that frees the resource, which may commit in a different cycle. In the example of section 4.7, the instruction G waited for the resource used by instruction C, but it must be scheduled after B and the `safe` value indicates the issue-group where it is scheduled. If the Commit stage could process just one instruction per cycle, C would not commit until one cycle later than B, so G would stall one cycle.

To take these extra cycles into account, the `safe` register is updated whenever the commit width is reached and it is known for sure that it will take an extra cycle to commit the younger instructions. In our default configuration up to 11 instructions can commit per cycle. Also, it can process only up to one control instruction per cycle. Therefore, the `safe` register is incremented in one unit after 11 scheduled instructions that haven't updated the `safe` register. Additionally, when a control instruction is scheduled in an issue-group that does not update the `safe` register, the content of the register is incremented and the counter is cleared. Figure 4.19 shows the logic involved.

4.4 Memory instructions

Memory instructions require some processing by the scheduler beyond what is done to the arithmetic instructions. The reason is that these instructions are inserted in dedicated queues at execution time (the loads in the LQ and the stores in the SQ) and need an identifier in these queues. Furthermore, they can have data dependences through memory, when two instructions access the same address (memory alias). In most cases, these dependences can't be resolved until execution time. The Rcreate logic must be able to accurately predict these dependences and schedule the instructions accordingly. Finally, their latency is variable and can change in each execution of the instructions.

4.4.1 Identifiers

Memory instructions are not only inserted in the ROB when they are executed but also in the LQ or the SQ. The processor uses these queues to detect when two memory accesses are aliased and bypass the data if possible. The number of instructions that can be stored in these queues is indicated in this text by *LQ_size* and *SQ_size*.

The Rcreate logic assigns the identifier in the corresponding queue of the memory instructions in the same fashion it assigns identifiers in the ROB. Thus, it has the `LQ_id` and the `SQ_id` registers to

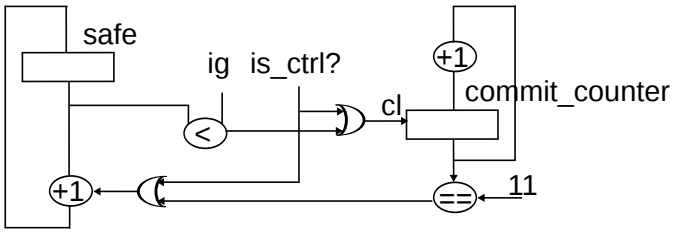


Figure 4.19: The value of the `safe` register is incremented to take into account the commit width. The `safe` register is incremented every 11 scheduled instructions that have not modified the register. It is also incremented after a control instruction is scheduled. The signal `is_ctrl?` indicates whether the instruction currently being scheduled is a control instruction. `ig` indicates the issue-group in which the current instruction has been scheduled.

know which is the next identifier that must be assigned for each queue. These registers are updated whenever a load or store instruction is scheduled. Like the identifiers in the ROB, the identifiers in the LQ and the SQ are assigned in program order by Rcreate, occupied at execution in the schedule order and freed when the instruction commits in program order. Therefore, two tables with the `safe` values are needed in order to have deadlock-free schedules: one table for the LQ (`LQ_safe`) and another for the SQ (`SQ_safe`). There is a `safe` field for each identifier in the LQ and the SQ. They are updated exactly in the same way as the `ROB_safe` fields are.

4.4.2 Addresses

Memory access ordering policy

Our architecture determines that the program order of all the accesses to the same memory address is preserved and the instructions are re-executed if needed to preserve the order. On the other hand, two memory instructions that access to different addresses can be freely reordered.

To consider two instructions as aliased, the size of the access (byte, word, double-word or quad-word) is taken into account, so a partial alias can be detected while accesses to disjoint parts of the same quad-word can be reordered. The memory addresses accessed are always naturally aligned to the size of the access, so the lower bits of the two addresses are ignored when they are compared to detect if two accesses are aliased. The number of ignored bits is determined by the size of the largest of the two accesses.

The load instructions perform the access to memory in the Writeback stage while the store instructions do it in the Commit stage. Since instructions commit in order, the relative order of the accesses of two store instructions is always preserved, regardless of the order in which they are issued. Both load and store instructions calculate the address in the Execution stage. The address is written in the corresponding entry in the LQ or the SQ in the Writeback stage. The store instructions also copy there the data to be written in memory.

When a load (*l1*) enters the Writeback stage, its address is compared with those of the other instructions in both the LQ and the SQ. Thus, it can be detected if there is any alias. When an alias is found, the processor proceeds as follows:

a. If there is a younger aliased load (*l2*) that has already read the data, *l2* is re-executed.

b. If there is an older aliased store, it bypasses the data from the SQ, instead of accessing the memory, which the store has still not updated.

c. Aliased older loads and younger stores are ignored.

When a store (*s1*) enters the Writeback stage, its address is compared with those of the instructions in the LQ. Thus, it can be detected if there is any alias. When an alias is found, the processor proceeds as follows:

a. If there is a younger aliased load that has already accessed the memory, the load is re-executed.

b. If there is a younger aliased load that has read the data through a bypass from the SQ, the action depends on the relative order of *s1* and the bypassing store (*s2*). If *s1* is older than *s2* nothing happens, since the load has read the data from the store on which it actually depends. Otherwise, the load is re-executed.

c. Aliased older loads are ignored.

Two memory accesses are required to have equal size and data type (integer or floating point) to bypass the data. Moreover, an aliased store-load pair cannot bypass the data if both reach the Writeback stage at the same cycle. If that happens the load is re-executed.

Scheduling aliased memory instructions

When a memory instruction commits and is inserted into the buffer to be processed by Rcreate, the address that was accessed is also copied in the `rcreate_input` buffer. The scheduler uses the addresses to predict whether two memory instructions will alias or not at execution time. Those instructions that have accessed the same address are supposed to alias again in following executions of the same instructions and the schedule is consequently restricted. Otherwise, they are considered as independent instructions.

The Rcreate has the `address_info` tables with an entry for each of the most recent memory instructions that have been scheduled in the current rgroup. It is not necessary to store the information about all the memory instructions in the rgroup but just for the last ROB_size instructions or for the last LQ_size loads and last SQ_size stores. Since an instruction already cannot be scheduled before the previous use of its identifier in the ROB, in the LQ or the SQ, older memory instructions can be ignored.

The `address_info` structure can be organized as a single table with ROB_size entries. Alternatively, it can be separated into two tables, the `load_address_info` table with LQ_size entries and the `store_address_info` table with SQ_size entries. The store instructions require more information that load instructions and $(LQ\_size + SQ\_size) < ROB\_size$ in our default configuration. Thus, separated tables is our choice for the `address_info` structure since it requires using less area. Furthermore, the consequences of aliasing with a load and with a store are different and each case requires a different logic. Keeping separated tables makes the logic simpler.

The section below present first a simpler version of the `address_info` structure that ignores the size of each access when aliased instructions are detected. All the instructions are considered to access quad-words and the three least significant bits of each address are ignored. Therefore, some accesses are considered as aliased when they actually access different bytes, words or double-words of the same quad-word. In the next section, the tables are modified to take into account the actual size of the accesses.

Size-ignoring address_info structure

The `load_address_info` and `store_address_info` tables are shown in figures 4.20 and 4.21 respectively. Each `load_address_info` entry stores the address that the corresponding memory instruction accessed at execution time. Each entry also has a `valid` bit and the `sched_pos` field, that indicates in which issue-group the load has been scheduled. Each `store_address_info` entry stores the address that was accessed, the size of the access and whether it writes an integer or a floating value. The entry

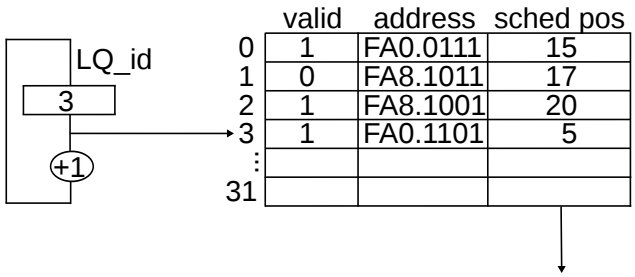


Figure 4.20: The `load_address_info` table allows detecting the aliased loads and scheduling them properly.

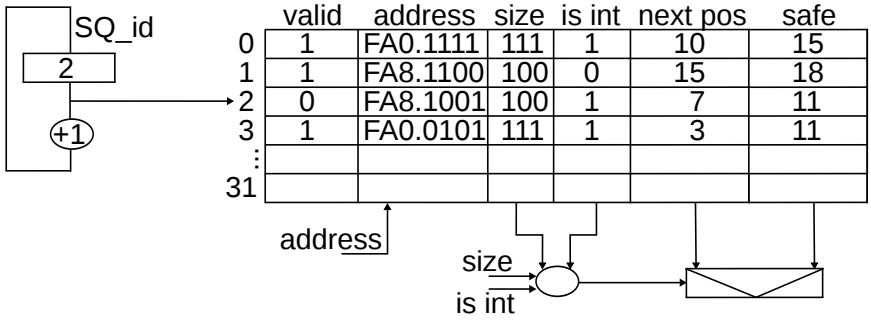


Figure 4.21: The `store_address_info` table is used to detect aliases between load and store instructions.

has a `valid` bit, the `next_pos` field that indicates the issue-group after the one in which the store has been scheduled and the `safe` field with the value of the `safe` register once the store was scheduled.

The `LQ_id` and the `SQ_id` registers, that are already used to assign the identifier in these queues, also indicate in which entry of the `load_address_info` or the `store_address_info` table the next memory instruction must be inserted. The corresponding register is incremented after a load or store is scheduled and inserted in the table. Thus, the insertion automatically invalidates the information of the previous instruction that used the same identifier in the LQ or SQ, since its entry is overwritten.

The information of a memory instruction after *ROB_size* scheduled instructions is not needed, since it cannot be in-flight and be aliased with younger instructions. If there have been less than *LQ_size* loads or *SQ_size* stores in the last *ROB_size* instructions, the information of the old instruction is still in the `load_address_info` or the `store_address_info` table. To invalidate it, the table should store the `ROB_id` of each instruction and compare it with the content of the `ROB_id` register, invalidating the entry on a hit. However, it is not strictly necessary to invalidate the entry because after *ROB_size* scheduled instructions, the values in the `sched_pos`, `next_pos` and `safe` fields of the old instructions cannot be greater than the value of the `ROB_safe` table accessed by the current instruction. As the old values do not have any effect in the scheduling process, it is simpler just to not invalidate them.

When a load instruction is being scheduled, its address is used to perform a CAM-access to both `address_info` tables. The `address` field of the entries are compared with the current address ignoring the three least significant bits. An entry hits if its address is equal and its `valid` bit is set. Up to one load and one store can hit in this version of the `address_info` structure. If no entry hits, the instruction is scheduled considering that it is data-independent to all previous memory instructions. Using the examples of figures 4.20 and 4.21, if the load instruction currently being scheduled has the address FA0.0101, the entry 0 of the `load_address_info` table hits. Besides, the load is aliased with

the entry 3 of the store_address_info table. The addresses have 16 bits to simplify the examples, with the upper 12 bits shown in hexadecimal and the lower four in binary.

If an entry in the load_address_info table hits, the current load is scheduled in the issue-group that is indicated by the sched_pos field of the entry. Therefore, both loads are scheduled in the same issue-group. In the example of figure 4.20, the instruction is scheduled in the issue-group 15.

If an entry in the store_address_info table hits, there are two possibilities:

a. The data can be bypassed at execution time (equal size and data type). The load is scheduled in the next_pos issue-group, which is one issue-group later than the store. At execution time, the load will arrive to the Writeback stage one cycle after the store and will bypass the data from the SQ.

b. The data cannot be bypassed, so the load will be re-executed if it reaches the Writeback stage before the store commits. Thus, the load is scheduled in the issue-group indicated by the safe field of the entry of the aliased store. That is, when the load reaches the Writeback stage the store has already committed, so the load can read the data from memory.

   In this case, the identifier in the SQ of the aliased store is copied with the load in the rgroup. This allows detecting at execution time whether that particular store has actually committed. If it hasn't committed yet, the load stalls in the Issue stage until the store commits.

   Additionally, the entry in the SQ_safe table of the aliased store is updated if its current content is less than the issue-group in which the load is scheduled. The reason is that if a store that is younger than the load reuses the identifier in the SQ and it is scheduled before the load, it may prevent the load to issue at execution time and lead to a deadlock.

In the example of figure 4.21, the store in the entry 3 of the table accesses a byte and writes an integer value, so if the aliased load performs the same kind of access, it is scheduled in the issue-group 3 (next_pos). Otherwise, it is scheduled in the issue-group 11 (safe).

After a load has been scheduled, it is inserted in the load_address_info table. If the access to the load_address_info table when scheduling the current load returned any aliased load in the table, the valid bit of its entry is cleared. Similarly, when a store is inserted in the store_address_info table, a CAM access to the address field of the table is performed with the address of the instruction. If there is a previous aliased store, it is invalidated. This enforces that there is at most one aliased instruction in each address_info table when they are accessed, thus simplifying the logic. The older information is not needed anymore; when the aliased instructions are two loads, the newer one cannot be scheduled before the older one and its sched_pos field is equally or more restrictive. When they are two stores, the newer one can be scheduled before the older one, and have a smaller next_pos field. However, it is the newer store that must bypass the data to a dependent load. If the data cannot be bypassed, the load must be scheduled after the newer store commits, which is specified by its safe field, so the information of the older store is not relevant anymore in any case.

Size-aware address_info structure

The address_info structure presented above effectively detects the memory aliases but some instructions are considered as aliased when actually they are not. That happens when two memory instructions access different parts of the same quad-word. The performance of the rgroup can be improved if the size of the accesses is taken into account. The drawback is that there can be up to 15 aliased loads and 15 aliased stores in the tables instead of just one. The logic required to handle that is more complex.

In our architecture, all addresses are naturally aligned to the size of their access. Therefore, the least significant bits of the addresses can be ignored when trying to detect if two instructions are

aliased. The actual number of bits ignored depends on the size of the largest of the two accesses. If one of the instructions accesses a quad-word, the lower three bits of the two addresses are ignored. Otherwise, two bits for a double-word access or one bit for a word access are ignored. If the two instructions access a byte, the whole addresses must match to consider them aliased.

The `load_address_info` and the `store_address_info` entries have a size field. The version of the `store_address_info` table presented in the section above already had a size field, used to know if data is bypassable. To detect an alias, the `address_info` tables are accessed with the address and the size of the current instruction. This size and that of the entry must be compared to know how many bits must be ignored. The largest access determines the number of bits that are ignored. A simple way to implement it is to encode the size of the accesses as a three-bit mask, where each zero correspond to a bit of the address that must be ignored, i.e. using the mask 000 for quad-words, 100 for double-words, 110 for words and 111 for bytes. A bitwise AND of the two size masks and the lower three bits of the addresses directly clears the desired number of bits.

For example, if one of the memory instruction accesses bytes and the other double-words, the AND of the two masks yields the resulting mask (100) that will be applied to the addresses. That is, the two least significant of the addresses will be ignored.

In the size-ignoring `address_info` tables, the aliased entries are invalidated when inserting a new instruction, so just one entry can hit per table in the next access. In the size-aware tables, the older aliased instructions can still be relevant later, if the size of their access is larger than that of the new instruction. For instance, let's consider the following code:

```
A: LDW 0(R1), R2
B: LDB 0(R1), R3
C: LDB 1(R1), R4
D: LDB 0(R1), R5
E: LDW 0(R1), R6
```

The instructions A and E access words, while B, C and D access bytes. Both B and C are aliased with A, but B and C are independent with respect to each other. D is aliased with A and B. E is aliased with all the previous instructions. When B is scheduled and the information of its access is inserted in the `load_address_info` table, the information of A is still needed, so its entry is not invalidated. When C is processed by the Rcreate logic, the entry of A in the `load_address_info` table hits and is used to schedule C. When D is scheduled, both the entries of B and A hit. The highest `sched_pos` value is used. The information of the instruction B is not needed afterwards, since any instruction that is aliased with B must also be aliased with D, so the entry of B in `load_address_info` table can be invalidated. When the instruction E is scheduled and the `load_address_info` table is accessed with its address, the entries of the instructions A, C and D hit. The highest `sched_pos` value is used and all these entries can be invalidated then.

Note that after the instruction C is scheduled, all the bytes accessed by the instruction A are aliased with posterior instructions. This makes A irrelevant, but we don't take into account this case, that requires tracking all bytes of each access independently.

Thus, an aliased entry can invalidated if its size is less or equal to that of the inserted instruction. Up to 15 entries in each table can hit: one quad-word load, two double-word loads, four word loads and eight byte loads. The quad-word load must be the older instruction, the double-word loads must be older than the word and byte loads, and the word loads must be older than the byte loads. A quad-word load can be aliased with all these instructions (which all would be thereafter invalidated). The logic must detect the higher value of the `sched_pos` of all the aliased loads. For each aliased store, the `next_pos` or the `safe` must be chosen, depending on whether the data can be bypassed. The most restrictive of all the matching entries is used to scheduled the current load.

Managing invalidations is more complex in this case than when the size of the access is ignored. The cost of not invalidating the older entries is that, in the worst case, all the entries in both tables

can be aliased with the current instruction. In our baseline, it means that up to 32 entries in each table can hit. Since it is a very unlikely case and we already have to deal with up to 15 hits per table, it seems reasonable to have a sequential logic to processes all the entries that hit. Thus, no entry is invalidated and up to 32 entries can match. In any case, whether the old entries are invalidated will not have any impact in the scheduling of incoming instructions.

4.4.3 Latency

Memory instructions have variable execution latencies and the Rcreate logic must predict the latency of each memory instruction in order to be able to schedule the instructions that depend on them. The latency of an access depends on the configuration and the state of the memory hierarchy. In our default configuration, the only fixed latency corresponds to the accesses that hit on the first cache level (L1), which have a three-cycle latency for the integer loads and four cycles for the floating point loads. In any other case, the exact number of cycles of the latency depends on the occupancy of the buses, the number of outstanding cache misses and which cache lines they involve, etc.

However, assuming that the access finds a completely clean state of the memory hierarchy and buses specified, a basic latency can be established for the cases when the access hits on the second cache level (L2) and when the data must be read from memory: 13 and 84 cycles respectively, with an extra cycle for floating point accesses.

Consequences of scheduling with a wrong latency

Although many programs have a very high L1 hit rate, it wouldn't be a wise decision to use the L1 hit latency for all loads since the penalty when a load misses is very high. The latency is used to know in which issue-group the result will be available and schedule there the dependent instructions. When a correct latency is chosen, the instructions are scheduled in the earliest issue-group where they don't stall. But when a load is scheduled with the hit latency and it actually misses at execution time, the first dependent instruction stalls until it can read its source register. And not only that instruction delays its execution; the issue logic processes the instructions in order, so many independent instructions that are scheduled in the next issue-groups have to wait until the memory access finishes. Furthermore, even when the first dependent instruction is scheduled later than just after the L1 hit latency cycles (for instance, because it has another source operand which is known to be available later), an independent instruction may stall waiting for the load to finish its access; that is, when an instruction reuses a ROB or queue entry or a physical register that has not been freed because the load has still not committed. It may be even a resource that is not used directly by the load instruction, but one used by an instruction that must commit after the load.

On the other hand, if a load is scheduled assuming the latency of a miss and it actually hits in the L1, the scheduler inserts its dependent instructions later than they could have been placed, with a potential performance loss. Again, not only the dependent instructions are affected, but also independent instructions that reuse a ROB entry, physical registers, etc. are scheduled later than needed, since the `safe` register is updated using the latency of the scheduled instruction.

Predicting the latency is more complex than just deciding whether the instruction hits or misses in the different levels of the memory hierarchy. For instance, when a memory instruction misses on the L1 and L2 caches but it is found that its cache line is already being retrieved from memory due to a previously missed instruction, the access can be considered as an L2 hit. But if the two instructions are executed within a few cycles, the second load will have near the same latency as the first one. Therefore, although it is correctly predicted as an L2 hit, the actual latency can be very different from the prediction. On the other hand, if the access is considered as an L2 miss, it can happen anyway that the second instruction is executed many cycles after the first one, so the actual latency is much closer to that of an L2 hit.

Latency predictor

Taking all this into account, we use a predictor to decide the latency of the load instructions, although [17] shows that this kind of prediction is a difficult problem. Besides, since our rgroups are reused many times, it can happen that a load hits on some executions of the rgroup but misses on others.

Rcreate tries to detect biased loads with the `lat_pred` table. It is a PC-indexed table of saturating counters. The table is accessed with the least significant bits of the PC of the instruction. When a load instruction commits and is inserted in the `rcreate_input` buffer, its entry also stores whether it hit on the L1 cache and its execution latency. When the load is scheduled, its counter is updated. The counter is increased on an L1 hit and right-shifted on an L1 miss in order to weight the misses more. The left-most bit determines the prediction: if set, the scheduler uses the L1-hit latency. Otherwise, it uses the latency seen in the last execution of the load, that is, the latency stored in the `rcreate_input` buffer.

Load instructions that frequently miss in the L2 cache are more likely to benefit from re-scheduling and updating the predicted latencies, so when a load executed in Rcache mode misses in the L2, its rgroup's counter in the Rcache is decremented.

4.5 Control instructions

4.5.1 Branch prediction

The stream of instructions that are executed after a control instruction can follow several paths. Conditional branches can be taken or not taken. Furthermore, the target PC of an indirect branch can be different in each execution. Once the Rcreate logic has scheduled a control instruction, we can consider three ways to continue the schedule:

a. Stop scheduling, with the control instruction as the ending instruction of the rgroup and use a branch predictor in the Rfront-end. In this case, the rgroup often contains less instructions than the maximum possible size. Since control instructions are very frequent in most codes, creating a new rgroup after each control branch leads to having many rgroups with very few instructions. The ILP that the scheduler can extract with such small rgroups is very small. If the Rcreate logic closes the rgroup only after the scheduling of a given number of branches, it reduces the problem of the size of the schedule, although the ILP extracted is still limited.

   Additionally, in this case a branch predictor in the Rfront-end is needed, in order to decide which path the Rfetch logic follows after processing the rgroup. This goes against our design principle to keep the critical path of execution as simple as possible and make most of the decisions in the Rcreate logic, out of the critical path of execution. Furthermore, the branches in the rgroup are reordered, so either: 1) the branch predictor is not accessed in program order, which can affect the accuracy of its predictions; 2) there is a list of the branches of the rgroup in the program order, stored along with the rgroup, which increases the ammount of information stored with each rgroup in the Rcache; or 3) the branches are scheduled in order, which imposes an unnecessary restriction on the scheduler, that can extract less ILP from the code.

b. Continue scheduling, using a branch predictor in the Rcreate logic to decide the path followed by the scheduler. The branch predictor is removed from the Rfront-end. The benefit from using a conventional branch predictor is that the schedule created contains usually the most probable path. The main disadvantage is that the scheduler is fed by the stream of committed instructions, so if the last execution of the branch didn't take this time the predicted path, the desired instructions are not available to the scheduler. In this case, the scheduler can either 1) stop scheduling (so it is actually case a) or 2) wait until the wanted path is executed and

its committed instructions enter the Rcreate logic, discarding the instructions committed until then. The second possibility can lead to lower IPC, because the skipped instructions will not be found in the Rcache and will be executed in Icache mode.

c. Continue scheduling from the path followed after the last execution of the branch. In this case, the size of the rgroup is not limited by the prediction, the ILP extracted from the scheduled path is high and the Rfront-end just needs to follow the path predicted by Rcreate.

   Note that, although the scheduler predicts that the branch will behave just like the last time, it is not a one-bit branch predictor since it implicitly incorporates history in the prediction: the path of the previous control instructions that have been scheduled in the rgroup. A branch can be scheduled with different behavior within the same rgroup. For instance, a conditional branch that is taken every other iteration of a certain loop, arrives to the Rcreate logic alternating the taken and not-taken case. The scheduler captures the pattern and the branch prediction will hit until the program exits the loop. More complex patterns would be captured as well. In average, the prediction is less accurate than that of a branch predictor accessed at execution time but the hit rate is usually close to that of a conventional branch predictor. The average misprediction rate is 5.21 for the default configuration of ReLaSch, while in the reference OoO processor it is 4.00. See table C.8 in appendix C for more details.

Our processor uses option c) because this scheme requires less complex logic in the execution pipeline, enables the scheduler to work with the current execution path and achieves acceptable branch misprediction rates. Thus, the Rcreate logic just continues scheduling from the path of committed instructions that follows a control instruction. The prediction is validated in the Commit stage against the actual behavior of the control instruction, just like it is checked for the prediction of the branches executed in the Icache mode. To do so, the conditional branches in the rgroup have a taken/not-taken bit, used to record which prediction was made. The indirect branches in the rgroup also store the predicted target PC.

Indirect branches

The PC of an instruction occupies many bits and most instructions do not need to store a target PC, so having a dedicated `target_PC` field for each instruction would result in many unused bits in the Rcache. Therefore, the target PC of all the indirect branches are stored separately in the Rcache, though this puts a limit on the number of indirect branches that can be scheduled in the same rgroup. To make the Rfront-end simpler, these target PCs are stored in the Rcache in the order of the indirect branches in the schedule, that is, out-of-order. Since at the time when a given branch is scheduled its relative order in the final schedule is not known (a younger branch can be scheduled before it), the actual order of the target PCs is not known until the rgroup is closed. The compacting logic (see section 4.7) stores the target PCs in the required order. Until that moment, the target PCs can be stored either: a) with the instruction, using a dedicated field for each instruction in each issue-group of the `sched` table; or b) in a separated queue structure in commit order, which requires having an additional field in the instruction to point to the corresponding entry in the queue. This field is smaller than the whole PC, so option b) is our choice.

There are two interesting cases which are not very frequent but have appeared in a number of experiments. One case is related with multi-target branches and the other with return addresses.

A conventional Branch Target Buffer (BTB), combined with a Return Address Stack, is a very effective way to predict the target PC. However, while a conventional BTB has a high misprediction rate for the indirect branches that continuously change their destination, the Rcreate logic detects the patterns in the target PCs when they exist and predicts that kind of branches correctly. The ReLaSch processor would have a significant advantage in applications with such kind of branches

if it was compared against a reference machine with a conventional BTB. Therefore, the reference OoO processor used in our experiments uses an enhanced BTB to correctly predict the multi-target indirect branches in order to compensate this advantage of the ReLaSch processor and make a fairer comparison.

In the second case, a return instruction is sometimes mispredicted by the Rcreate logic. although it is not a very frequent case, is always worse than the prediction made by a Return Address Stack, which has almost a 100% of accuracy. The misprediction can happen if there is a long function that is called from different points in the code. Short functions are typically in-lined in the rgroup but a long function can span through several rgroups, which are shared by the several call points. The return address predicted by the scheduler is wrong when the call point at scheduling time and the call point at the execution time are different.

4.6 Conditional move instructions

Conditional move instructions of the Alpha ISA move a source register to the destination if another source register is not zero and leave the destination register unchanged otherwise. As explained in section 4.2.1, the destination physical register is assigned by the Rcreate logic incrementing the `phy` field of the `reg_info` table, so the architectural value of the destination register is stored in different physical registers before and after the conditional move commits. This means that, in order to leave the destination register unchanged when the condition is false, it must be explicitly copied from the older architectural physical register to the new one.

The OoO, IO and ReLaSch processors execute the conditional moves with just one instruction (cf. appendix B). The execution pipeline reads the current value of the destination physical register after the conditional move instruction is issued, so the instruction actually has three source registers.

Therefore, the destination register of a conditional move involves using two different physical registers: a) the one that stores the previous value, identified by the value of the `phy` field of the logical register in the `reg_info` table. It is read by the instruction and it is considered as an additional source operand by the Rcreate and the Issue logic. b) The actual destination physical register $phy + 1$ in which the result is written.

When Rcreate schedules a conditional move, it copies in a dedicated field the value of the `phy` field of its destination register, before it is updated. It is used at execution time to access the register as an additional source register. The implementation of the issue logic or the register file could limit the number of conditional moves per issue-group since they require accessing more source registers per cycle and aren't a very frequent instruction. If just one conditional move is accepted per issue-group, a `cmov` bit would be added to each issue-group of the `sched` table, similar to the `busy` bits of the FUs. If the desired issue-group *ig* has its `cmov` bit set, the scheduler tries to insert the instruction in the issue-groups *ig+1*, *ig+2*, etc. Section 9.5.3 shows that limiting the number of conditional move instructions per cycle doesn't have any impact in performance.

4.7 Closing the rgroup

The following situations make the Rcreate logic close the current rgroup and stop scheduling instructions in it:

a. It has already scheduled the maximum number of instructions in the rgroup.

b. After taking into account the available field of the source registers, the `safe` field of its ROB entry, etc., it should schedule an instruction in an issue-group beyond the end of the `sched` table.

c. It has already scheduled the maximum number of indirect branches in the rgroup.

d. It has scheduled a system call instruction. In our implementation, the pipeline is flushed after a system call instruction commits, so it makes no sense to schedule instructions after it.

e. The `rcreate_input` buffer had an overflow and all the instructions there have been scheduled.

4.7.1 Compacting logic

After an rgroup is closed by Rcreate, it is sent to the Rcache to be stored there. The format in which an rgroup is stored in the Rcache differs from that of the `sched` table, so the Rcache line is not a direct copy of `sched` table. A compacting logic is needed, that sequentially reads the issue-groups of the `sched` table and inserts the instructions in the Rcache line or in an intermediate buffer with the same format. There, all the instructions are inserted in consecutive entries and the first instruction of each issue-group uses a flag to indicate the beginning of a new issue-group. The rgroup is stored in a compact format by removing the empty issue-groups and the empty slots in partially empty issue-groups. The compacting logic also creates the list of targets of the indirect branches following the actual relative order of the branches in the schedule. The targets are stored in the Rcache in that order.

Once the compacting logic has completely processed the `sched` table, the `sched` table is emptied (clearing all the `valid` and `busy` bits) to schedule there the instructions of the next rgroup. A given issue-group can be cleared just after it has been processed by the compacting logic. The `phy` field of the logical registers in the `reg_info` table as well as the `ROB_id`, `LQ_id` and `SQ_id` registers are cleared.

Furthermore, all the `safe`, `sched_pos` and `read` fields are cleared and the Rcreate logic invalidates the content of the `address_info` tables. Although it can make sense to keep this information from one rgroup to the next one in order to improve the performance of the schedules when they are executed back to back, the logic needed to do it is not simple and our experiments have shown little benefit from it.

4.8 Rgroup identification

To identify an rgroup, it is used the next PC of the instruction immediately preceding the beginning of the rgroup. Typically it is equal to the PC of the first instruction in the rgroup (see section 5.1 for details). When scheduling the first committed instruction after the processor is switched on or the first instruction inserted after an overflow of the buffer of committed instructions, the PC of this same instruction is used, since there is no information about the older instructions.

The Rcreate logic also has a history register, updated with the taken/not-taken bit of each scheduled conditional branch. The content of this register when the first instruction of an rgroup is scheduled is used along with the PC to identify that rgroup in the Rcache.

4.9 The Scheduling mode and the Idle mode

It is not necessary that the Rcreate logic is constantly scheduling instructions into new rgroups, because rgroups are meant to be reused. Thus, Rcreate can be either in the Scheduling or the Idle mode. The Scheduling mode corresponds to the behavior detailed in the rest of this chapter. In the Idle mode, the committed instructions are inserted in the `rcreate_input` buffer but are not scheduled.

The Rcreate logic performs the following tasks to all the instructions in the `rcreate_input` buffer, regardless of the mode:

a. If the current instruction is a load, it updates the predictor of the memory latency in order to predict in the Scheduling mode the latency of the loads using the most recent information.

b. If the instruction is a conditional branch, it updates the history register, which will be used to identify the next scheduled rgroup.

c. It stores the next PC, which indicates the instruction that must be executed just after it; it is used to identify the next scheduled rgroup.

Besides, in the Idle mode, the Rcreate logic checks if the current instruction should switch back to the Schedulitheng mode and start scheduling a new rgroup.

Rcreate is initially in the Scheduling mode. It can switch from the Scheduling to the Idle mode only after an rgroup has been closed. Just after closing an rgroup, the Rcreate logic checks if the next instruction that must be scheduled was executed in the Rcache mode. If this condition is evaluated as true after two closed rgroups in a row, the Rcreate logic changes to the Idle mode.

Once it is in the Idle mode, there are two cases that put Rcreate back in the Scheduling mode: a) when the instruction currently at the head of the buffer was executed in the Icache mode; and b) when the instruction was executed in the Rcache mode but it forms part of a "bad rgroup". As explained in detail below in section 5.3, the processor can detect when a given rgroup repeatedly fails to be completely executed, due to frequent mispredictions; it is marked as a "bad rgroup" and its instructions too at execution time; the entries in the `rcreate_input` buffer have a bit to mark such instructions. The instructions of the "bad rgroup" are re-scheduled taking into account the new behavior of branches and memory instructions, which are expected to have a lower misprediction rate.

4.10 Block diagram

Figures 4.22 and 4.23 show a block diagram of the Rcreate logic, as a summary of all the elements and logic used there. Figure 4.22 shows the blocks used to schedule an instruction. The `rcreate_input` buffer stores the committed instructions until they are scheduled. The Rcreate logic uses the `reg_info`, `ROB_safe`, `LQ_safe`, `SQ_safe`, `load_address_info` (`ld_addr` in the figure) and `store_address_info` (`st_addr` in the figure) tables to know in which issue-group should be scheduled the current instruction. The maximum value provided by these tables is used. `SQ_safe` is only accessed when the instruction is a store. `LQ_safe`, `load_address_info` and `store_address_info` are only accessed when the instruction is a load. The insert logic tries to place the instruction in the selected issue-group. If it is not possible because it is full or the corresponding FU is busy, it attempts to place it in the next issue-groups.

The `reg_info` table is used also to perform register renaming. It provides the identifiers of the physical registers used by the instruction. The `ROB_id` register provides the identifier used by the instruction in the ROB. The `LQ_id` register provides the identifier of the load instructions in the LQ and `SQ_id` provides the identifier of the store instructions in the SQ. All this information about the instruction is stored alongside with the encoded instruction itself in the chosen issue-group. The `safe` register is updated with the identifier of the issue-group and the latency of the instruction. The `lat_pred` table is used to select the latency of the load instructions. Also, the target PC of the indirect branches is stored in a dedicated buffer.

Figure 4.23 shows the blocks and logic required to update the state of the Rcreate logic after scheduling an instruction. The updated content of the `safe` register is used to update the `safe` field of the destination physical register in the `reg_info` table and also the corresponding entries in the `ROB_safe`, `LQ_safe` and `SQ_safe` tables. It is also used to update the `store_address_info` table when the instruction is a store, in order to indicate in which issue-group the instruction is expected to commit. The selected issue-group (`igroup` in the figure) plus the latency (`lat`) is used to update the issue-group in which the destination register is available. Besides, `igroup` is also used to update the `address_info` tables.

The Rcreate logic keeps a counter of the number of scheduled instructions and the number of indirect branches in the rgroup. The schedule is closed when the limit of instructions or indirect

branches per rgroup is reached or the selected issue-group is beyond the size of the `sched` table.

When the rgroup is closed, the compacting logic reads the whole content of the `sched` table. It filters out the unused issue-groups and unused slots in each issue-group. The compacted rgroup is sent to the Rcache. All the tables and registers of the Rcreate logic are reset afterwards, with the exception of the `lat_pred` table and the `rcreate_input` buffer.

When an rgroup is closed, the Rcreate logic checks the instruction in the head of the `rcreate_input` buffer in order to decide whether it should continue scheduling or it should switch to the Idle mode. There is a mode change if the instruction was executed in the Rcache mode. In the Idle mode, the instructions in the `rcreate_input` buffer are processed just in order to update the `lat_pred` table and to know whether the Rcreate logic should switch back to the Scheduling mode. The mode change happens when the current instruction either was executed in the Icache mode or forms part of a "bad rgroup".

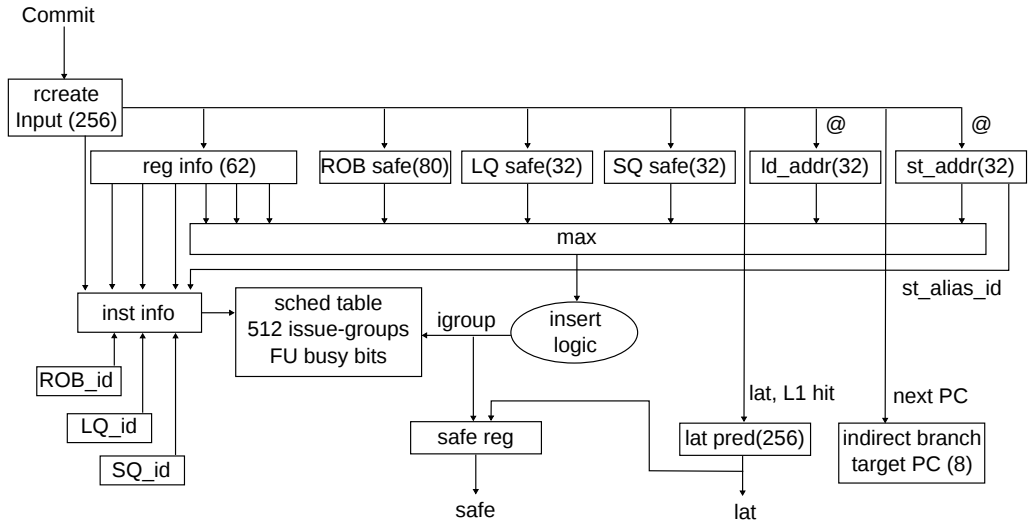


Figure 4.22: Block diagram of the Rcreate logic: blocks used to schedule an instruction.

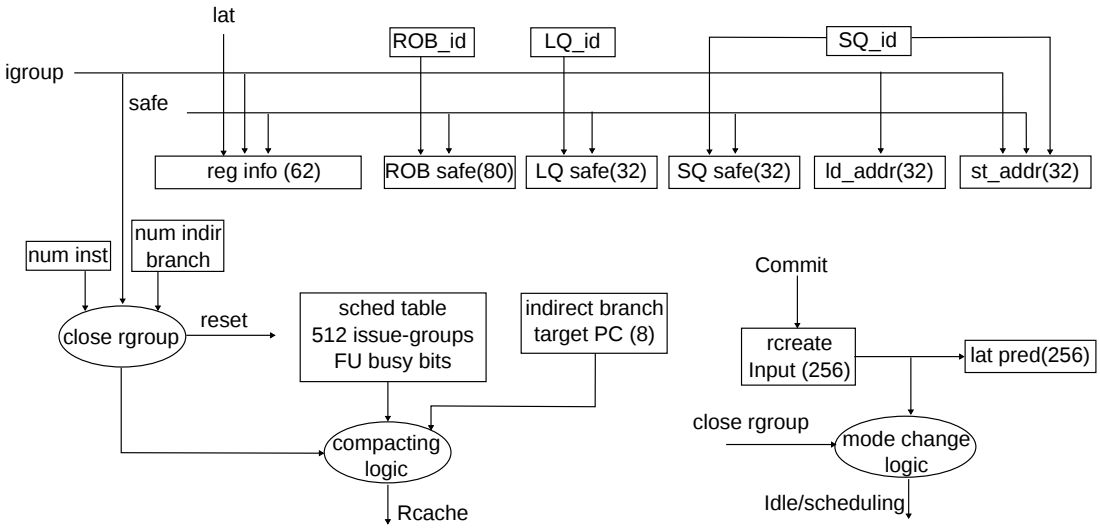


Figure 4.23: Block diagram of the Rcreate logic: logic required to update the state.

Chapter 5

The Rcache

The Rcache is the structure that stores the rgroups. The following sections explain how the rgroups are identified, how their information is stored, how frequently aborted rgroups are detected, and finally an estimation of the area that the Rcache requires.

5.1 Rgroup identifier

Each rgroup needs an identifier, which is used in the accesses to the Rcache to know if there is any rgroup that can be executed starting from the current PC. Each rgroup in the Rcache must use a unique identifier. For each control instruction, the Commit stage checks the correctness of its prediction. But it assumes that the instruction stored in the next ROB entry corresponds to the next PC in the predicted path or simply the next PC if the current instruction is not a branch. So we must enforce that the first instruction of an Rgroup corresponds to the next PC of the previous instruction. Therefore, the PC of the first instruction of an rgroup must be used to identify the rgroup, although it can be combined with other information such as history bits.

The following example illustrates how the rgroups are created and later found in the Rcache:

```
A: BNZ R0, D
B: MUL R1, R2, R3
C: ADD R4, R5, R4
D: AND R6, R7, R8
```

Let's assume that the first time this code is executed, the branch A is taken so the control flow goes to D. The processor is by now in the Icache mode. Since it is the first execution of these instructions a new rgroup (*r1*) is created out of them. This rgroup contains the instructions A and D among many other instructions. We assume it begins with an instruction older than A.

When the rgroup *r1* is executed and the branch A is taken, the instruction D and those that follow it in the schedule can commit. But if A is not taken, the prediction misses and execution restarts from the instruction B. The processor tries to execute then an rgroup that begins with the instruction B, but since this instruction has never been executed before, the instructions B, C, etc. are executed in the Icache mode this time. The Rcreate logic is likely to be in the Idle mode when B commits, because the processor was in the Rcache mode until then. In this case, the Rcreate logic schedules the committed instructions in a new rgroup (*r2*), that begins with the instruction B. The PC of B identifies the rgroup *r2*. The next time that A is not taken (or is predicted to do so in the Icache mode) the Rcache is accessed with the PC of the instruction B and the rgroup *r2* is executed.

5.1.1 Early retirement

It is better to not use directly the PC of the first instruction of the rgroup to identify it. The reason can be illustrated if we change the destination register of the instruction B of the previous example to R31. The register R31 always stores the value 0, so the instruction is actually a NOP and it is "early retired" before the Commit stage (see appendix B for details). Arithmetic and logical instructions as well as some kinds of load are early retired when they have R31 or F31 as destination.

```
A:  BNZ R0, D
B': MUL R1, R2, R31
C:  ADD R4, R5, R4
D:  AND R6, R7, R8
```

The instruction B' doesn't commit and it is never seen by the scheduler. Therefore, the rgroup (*r2'*) created after the misprediction of the branch A begins with the instruction C and its PC identifies that rgroup.

Each time that the instruction A is not-taken, the branch prediction of the rgroup *r1* misses and the Rcache is accessed with the next PC after the instruction A. Since it is the PC of the instruction B', the rgroup cannot be found. The processor goes then to the Icache mode, just to fetch the instruction B', which is later "early retired" and since the access to the Rcache returns that there is an rgroup that begins with the instruction C, the processor goes back to the Rcache mode to execute it. The problem is solved using the next PC of the instruction A to identify the rgroup the begins with the instruction C.

More generally, an rgroup is identified with the PC of the instruction (B) that follows the one (A) committed just before the first of the rgroup (C). When there is no early retired instruction just before the beginning of an rgroup, B is equal to C and the PC of the first instruction (C) identifies the rgroup. This is the most common case.

5.1.2 Index and tag

The least significant bits of the PC are used to index the Rcache, excluding the two least. The two least significant bits are always 0 because the instructions are encoded in 32 bits and are always aligned. The rest of bits of the PC are used as a tag for the rgroup. They must match to have a hit.

However, with this approach there can be in the Rcache only one rgroup that begins with a given instruction and it may be not enough in some cases. For instance, when the behavior of the branches in the rgroup depends on the path to the rgroup. So the history bits of the path to the rgroup are also part of the identifier of the rgroup, with a bit for each of the most recent conditional branches indicating whether they were taken or not taken. Siz bits of history are used in our default configuration.

The history bits are used to access the Rcache along with the PC. They form part of the tag although they don't actually must match to have a hit. The history bits are used only as a hint to choose between different rgroups that begin with the same PC. If there is an exact match, the matching rgroup is chosen. Otherwise, an rgroup with the matching PC is chosen even if the history bits are not equal. Since only the PC is used to index the Rcache, all the rgroups that begin with the same PC are stored in the same set.

Figure 5.1 shows an example of how the Rcache is accessed to look up an rgroup. 32-bit addresses are used in the example. The two least significant bits of the PC are ignored. The rest are used as index (the lowest six bits in the example, assuming 64 sets) and tag (the most significant 24 bits). The tag is compared with the ones stored in the indexed set. In the example, the Rcache is two-way set associative, so it must be compared with two tags. To have a hit, the tag must be equal and the corresponding `valid` bit must be set. In the example, an access with index 0 and tag 40FA73 would

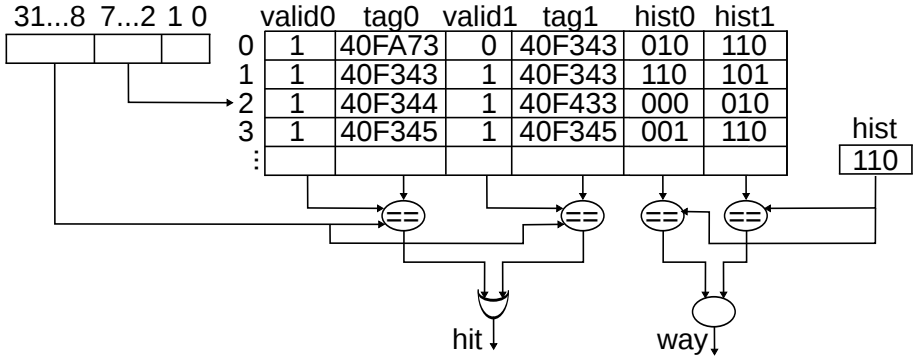


Figure 5.1: Example of an access to the Rcache.

hit, while an access with the same index and tag 407343 would miss because the `valid` bit for way 1 is cleared. Any access with index 0 and another tag would miss. Several tags can hit simultaneously, as would do an access with index 1 and tag 40F343 or with index 3 and tag 40F345. To select which line is actually fetched, the history bits are compared too. If just one tag matches the history bits are ignored. If there are more than one matching tags the rgroup with matching history bits is selected. If there are several rgroups with matching tag and none with a matching history, a pre-defined rule is used to select one of them. For example, the tag in a lower way. It is not possible that both the tag and the history bits of more than one rgroup match at the same time.

In the example a three-bit history register is used. An access with index 1 and tag 40F343 selects the rgroup in way 0 since it history bits are equal to the current history. An access with index 0 and tag 40FA73 selects the rgroup in way 0 even if its history bits don't match.

When the behavior of the branches is not correlated with the path to the rgroup and the Rcreate logic can capture it, the rgroup is usually completely executed, so it is not re-scheduled several times and inserted in the Rcache with different history bits. In this case, ignoring the history bits is the right choice. When the code includes history-correlated branches that are hard to predict, the instructions of the rgroup are re-scheduled and the new rgroup is inserted in the Rcache with different history bits. If the history bits matches the chosen rgroup is probably completely executed. Otherwise, it is likely that the chosen rgroup aborts its execution after a number of control instructions. However, our experiments (see section 9.3.5) show that the useful instructions that are executed anyway yield better IPC than if they are executed in the Icache mode, even a misprediction penalty is paid.

5.1.3 Replacement policy

When the Rcache must store a new rgroup, it first checks if there is already any rgroup with an identical identifier, that is, with exactly the same PC and history bits. If there is a match, the new rgroup replaces the older one. Otherwise, if there is any empty line in the selected set, it is occupied. Finally, if the set is full, the least recently used line is replaced.

5.2 Rcache line format

Each rgroup is stored in one Rcache line. Figure 5.2 shows the information stored for each rgroup in three parts: a) the control information, b) the target PC of the indirect branches, in their relative scheduled order, and c) the instructions. The sizes in the figure correspond to the default configuration of ReLasch presented in table 9.1.

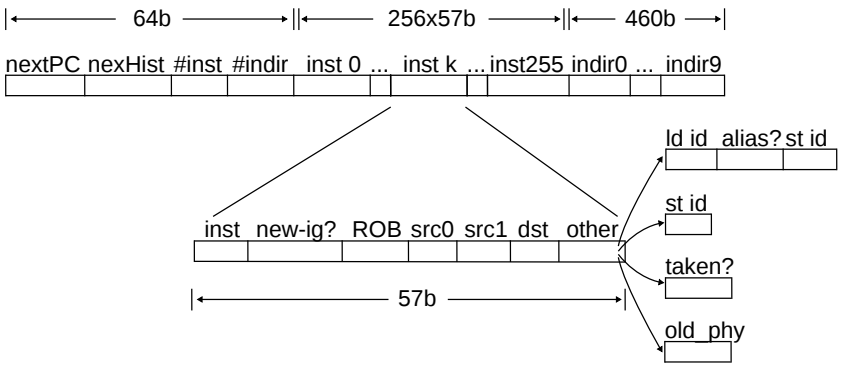


Figure 5.2: The information of an rgroup stored in an Rcache line.

The control information is the identifier of the next rgroup (formed by the PC of the instruction that follows the last one of the rgroup and the history bits of the last six branches), the size of the rgroup (the number of scheduled instructions) and the number of indirect branches. Although not shown here, the Rcache needs some additional information to manage the rgroups, such as the tag that identifies the rgroup, a valid bit and an LRU counter, as well as the saturating counter explained in section 5.3.

The predicted target PC of the indirect branches are stored separately. This imposes a limit to the number of indirect branches that can be scheduled per rgroup. A code with a very high number of these instructions is scheduled into many smaller rgroups, thus decreasing the IPC. Ten indirect branches per rgroup is more than enough for most benchmarks. Tables C.4 to C.7 in appendix C show the number of targets of static and dynamic indirect branches per each benchmark.

The instructions are stored as a sequence of issue-groups in the scheduled order. The empty slots in the issue-groups and the empty issue-groups are not stored in the Rcache. Instead, a bit is used to indicate that an instruction is the first one of an issue-group. Thus, for each instruction it is stored: the encoded instruction itself, its source and destination physical registers and its ROB identifier. For memory instructions, it is also stored the identifier in the LQ or the SQ. Besides, for some loads it is also used the identifier of an aliased store. To know when this field is required, an additional bit is used to validate it. A taken/not-taken flag indicates the predicted direction for each conditional branch. Conditional move instructions have also a field with the physical register that stores the previous value of the destination register. The fields that are only used by a specific type of instructions are overlapped to reduce the number of bits used. The instruction type that requires more extra bits is the load.

It could be useful to have also pre-decoded information: the logical registers of the source and destination registers, flags to indicate whether the instruction is a load, a store, a branch, etc. But it would require additional area and it is better to decode this information from the instruction each time it is executed. However, part of this information could be overlapped with the extra fields when the instruction is not a load.

5.3 Counters

Each rgroup in the Rcache has a saturating counter of five bits, used to detect the rgroups that usually do not commit all their instructions. The next time such an rgroup is read, its instructions have the "bad rgroup" flag set in their ROB entries. The Rcreate logic checks the flag and schedules a new rgroup when it detects these instructions. When an rgroup is read from the Rcache, its saturating

counter is accessed too. If it is equal to zero, the instructions fetched are marked as part of a "bad rgroup".

If an instruction commits and it is the last instruction of an rgroup, the Commit stage notifies to the Rcache that the rgroup has been completely executed. The identifier of the rgroup, that the Commit stage knows from the first instruction in the rgroup, is used to access the associated saturating counter and increment it. When a branch has been mispredicted or a load must be re-executed due to a memory alias and they were executed in the Rcache mode, the Rcache is accessed to decrement the saturating counter of its rgroup. Also when a branch is mispredicted and it was the last instruction of an rgroup the counter is decremented. Besides that, when an rgroup has been completely executed but one of its loads has missed in the L2 cache, the counter is decremented instead of being incremented.

5.4 Area and latency

The area needed to store the whole Rcache data depends on many parameters. The main ones are the number of rgroups that it can store and the number of instructions per rgroup. Also the exact size that occupies each instruction, which is determined by parameters such as the number of physical registers per logical register or the size of the ROB. Also the maximum number of indirect branches per rgroup affects the total area occupied.

In the default configuration of the ReLaSch processor, each instruction needs 32 bits for the encoded instruction, 7 bits for the ROB identifier (with 80 ROB entries), 9 bits for the physical registers (two sources and one destination register, with sets of 8 physical registers each one) and 1 bit for the new issue-group flag. There are also 11 extra bits, required by the load instruction to hold the LQ identifier (5 bits, 32 entries), the SQ identifier (5 bits, 32 entries) of an aliased store and a valid bit of the alias identifier. Actually, since loads only use one source register, 3 of these bits can be stored in the field used for one source register, reducing the size of the `other` field to 8 bits. When the instruction is not a load, part of the `other` field is used for the identifier in the SQ (stores), the taken/not-taken flag (for conditional branches) or the previous physical destination register (for conditional move instructions). All these fields add up a total of 57 bits per instruction.

Each rgroup has a list of the target PCs of its indirect branches. The addresses in the Alpha ISA have 64 bits, but actual implementations use fewer bits. The Alpha 21264 uses either 43-bit or 48-bit virtual addresses, which is configured with an internal control register. We assume that our addresses have 48 bits. Since instructions have 32 bits and are always naturally aligned, the lower 2 bits of a PC are always 0 and don't have to be stored in the Rcache. With 10 indirect branches per rgroup, and using 48-bit addresses, 460 bits are needed to store the target PCs.

Additionally, each rgroup has to store its size (8 bits for 256-instruction rgroups), the number of indirect branches (4 bits with 10 indirect branches at most) and the identifier of the next rgroup (52 bits, 46 for the PC and 6 for the history). It is a total of 64 control bits.

Each rgroup has an identifier, formed by a PC (46 bits) and a branch history (6 bits). The exact number of bits that must be stored as a tag depends on the number of bits of the PC used to index the Rcache. In the worst case, if the PC is hashed to index the Rcache or in a fully-associative Rcache, all the 46 bits of the PC must be used as tag. In our default configuration, where the bits are not hashed and the Rcache has 32 sets, the Rcache is indexed using 5 bits of the PC and the PC tag is reduced to 41 bits, plus the 6 history bits. Finally, each rgroup requires some control bits which are not visible outside the Rcache: a valid bit, the LRU bits and the 5-bit saturating counter. With the baseline 4-way associative Rcache, two bits indicate the LRU line, adding up a total of 8 invisible control bits.

In summary, a 256-instruction rgroup needs 14,592 bits to store the instructions, 460 bits for the target PC of the indirect branches and 64 control bits. Besides this, 55 internal bits are used. It is a total of 15,171 bits per rgroup, circa 1,897 bytes. A 4-way associative cache with 32 sets stores up

to 128 rgroups and it requires approximately 237KB. Section 9.3 explores many parameters that have an impact in the Rcache size as well as in the IPC of the processor.

The Rcache lines are longer than usual cache lines. However, a line is processed sequentially, one issue-group per cycle and there can be no random access in the middle of the line. The experiments explained in section 9.3.4 show that the Rcache is accessed once every 169 cycles in average, so any low bandwidth implementation of the Rcache would work. On the other side, the access to the tag information must be fast in order to detect when there is an rgroup available and change to the Rcache mode as fast as possible. Since the number of sets is small it should be easy to achieve it. In our default configuration, it takes 3 cycles until the hit is detected and the first issue-group is read by the Rfetch logic.

Chapter 6

The Rfront-end

In this chapter, the Rfront-end is described in detail. The Rfetch logic reads the Rcache, the Rdecode logic decodes the instructions and the Rmap logic checks, for each instruction, that its operands are ready and the required resources are available and then sends it to the Issue stage. The Rmap logic stalls when an operand or a resource is not available yet.

First, the Rfetch and Rdecode logic are described. Then, a simple, initial version of the Rmap logic is introduced. This version is able to process the rgroups and send the instructions to Issue, but it requires that the ROB is completely empty before sending the first instruction of an rgroup. It is followed by an explanation of how it deals with some instruction types that have specific requirements. Finally, Rmap is modified to allow it to start processing the instructions of an rgroup even if the ROB is not empty.

6.1 Rfetch and Rdecode

The Rfetch logic reads the Rcache. It takes many cycles to completely fetch an rgroup, which is done in parallel to the processing of its instructions by the rest of the pipeline.

After processing an rgroup, the Rfetch logic searches the next rgroup in the Rcache. The matching rgroup is fetched if the access hits; on a miss, the processor switches to the Icache mode. The identifier of the next rgroup is read from the Rcache with the information of the current rgroup. It is not necessary to choose between different rgroups, corresponding to different execution paths, after the end of a given rgroup: most of the times, the last instruction of the rgroup is not a control instruction; and when it is a branch, the target predicted at scheduling is likely to be correct, as the whole rgroup is implicitly used as history in the prediction.

The Rdecode logic decodes the instructions. In particular, the next stage (Rmap) requires at least the following decoded information for each instruction: the identifiers of the logical source and destination registers and whether an instruction is a load, a store, a conditional move, a conditional branch or an indirect branch. Further decoding can be performed also by the Rdecode logic, or later in the pipeline. For instance, which Functional Unit executes the instruction is not needed until the instruction is in the Issue stage.

The instructions are inserted in the `rgroup_buffer`, placed between the Rdecode and the Rmap logic, needed in case Rmap stalls (neither Rfetch or Rdecode have any hazard that must be solved by stalling).

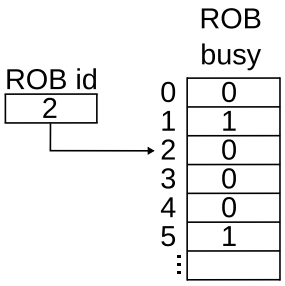


Figure 6.1: The `busy` bits of the ROB indicate if the desired identifier is available.

6.2 The Rmap logic with an empty ROB

The Rmap logic processes up to one issue-group per cycle. One issue-group is formed by up to six instructions, four integer and two floating point, which is equal to the issue width. The instructions in an issue-group are independent but are processed sequentially, in the order in which they are stored in the `rgroup_buffer`; if an instruction A cannot be sent to issue, the remainder of the issue-group must wait too, until A unblocks. This behavior allows an easy management of the `rgroup_buffer`, as it is only needed a pointer to the next instruction to process.

Each instruction in the rgroup has a flag that indicates if it is the first instruction of an issue-group. Each cycle, the Rmap logic processes the instructions on the head of the `rgroup_buffer` until it finds one with the flag set. Ideally, one issue-group should be processed per cycle, but the instructions with variable latency (mainly loads) may not follow the prediction of the scheduler. Therefore, sometimes the Issue stage stalls at an instruction dependent on the load or the Rmap logic stalls at an instruction that needs resources which are not freed until the load commits.

For each instruction, the Rmap logic needs to know the identifiers of the logical source and destination registers and whether an instruction is a load, a store, a conditional move or an indirect branch. This information was already extracted by the Rdecode logic and stored in the `rgroup_buffer`. Besides, the identifier in the ROB (`ROB_id`) and the identifiers of the physical source and destination registers are also needed. This information is stored with the instruction in the Rcache. Additionally, depending on the instruction type, the Rmap logic needs the identifier in the LQ or the SQ (for memory instructions), the previous destination physical register (for conditional moves), whether the instruction was predicted as taken (for conditional branches only) and the target PC (for the indirect branches).

Resources

There is a `busy` bit for each identifier in the ROB. It indicates if the identifier is already in use by any in-flight instruction or if it is available to be assigned. For each instruction, the Rmap logic checks that the `busy` bit is cleared for its identifier in the ROB. Figure 6.1 shows an example. The instruction uses the identifier 2 in the ROB, which is available. An instruction that uses the identifier 1 would stall at Rmap.

There are also `busy` bits for the identifiers in the LQ and the SQ, that are checked when a memory instruction is processed. Figure 6.2 shows an example. The entry 2 of the LQ has its `busy` bit cleared and the instruction can be processed and the instruction does not stall. An instruction that uses the identifier 3 in the LQ would stall until its entry is freed.

It is also necessary to check the availability of the destination register of each instruction. The register file is accessed using the identifier of the logical register (decoded in the previous stage) and, within its set of physical registers, the identifier of the physical register (assigned by Rcreate and stored

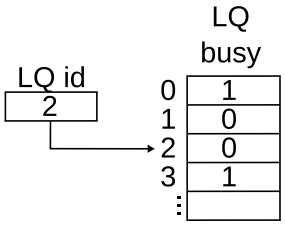


Figure 6.2: The `busy` bits of the LQ are used to know if the identifier used by a load instruction is available.

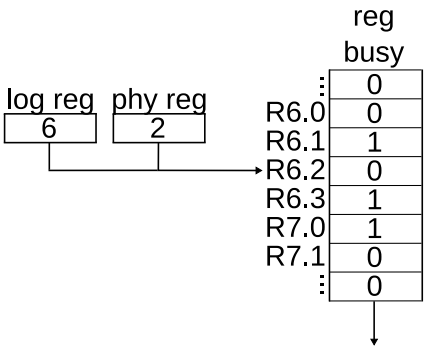


Figure 6.3: The Rmap logic checks the availability of the destination physical register. The identifier of the logical register (`log`) as well as that of the physical register (`phy`) are used to index the `busy` bit.

in the Rcache). There is a `busy` bit for each physical register that indicates whether the register is free. The `busy` bit is set when it is assigned as destination register, either by the Map stage or by the Rmap logic. It is not cleared until the physical register does not hold the architectural value of the logical register. When the instruction that writes the physical register *phy* commits, *phy* becomes the architectural register of the corresponding logical register *log*. When the next instruction that writes *log* (mapped to the physical register *phy+1*) commits, *phy+1* becomes the architectural register and the `busy` bit of *phy* is cleared.

Figure 6.3 shows an example of how the Rmap logic checks the availability of the destination physical registers. The logical register R6 is renamed to its physical register 2. The `busy` bit of the R6.2 is checked. Since it is cleared, the physical register can be assigned as destination. An instruction that uses the physical register R6.3 as destination would stall.

An instruction is sent to the Issue stage when satisfies all these requirements. The information of the instruction is copied on its entry in the ROB. The corresponding `busy` bit is set. The entry in the LQ or the SQ is also initialized. The `valid` bit of the physical destination register is cleared while its `busy` bit is set. Finally, the instruction is inserted in the integer or floating point issue buffer.

6.2.1 Indirect branches

The target of the indirect branches is predicted by the Rcreate logic and the prediction is stored in the rgroup. Using a field to store the target for each instruction would require a lot of area, that in most cases would not be used. Instead, these targets are not stored with the indirect branch but separately from the instructions in the Rcache line. Therefore, the number of indirect branches in an rgroup is limited.

The Rfetch logic copies all these targets in a buffer when it reads the rgroup. When the Rmap logic processes an indirect branch, the value on the head of the target buffer is copied with the instruction in the ROB and the head is advanced. Therefore, the order of the targets in the buffer is equal to that of the indirect branches in the rgroup. The target PCs are already stored in that order in the Rcache.

6.2.2 Identification of the rgroup

The Commit stage must know when begins and ends each rgroup in order to detect if all the instructions of an rgroup have committed. Some information is stored in the ROB entries to identify when an rgroup begins and ends. Each instruction processed by the Rmap logic has a flag to indicate that it is executed in Rcache mode. The first instruction in the rgroup also carries a copy of the identifier of its rgroup (actually, only the history bits, since the Commit stage already knows the PC). The new-rgroup flag of the first instruction of an rgroup is set. The Commit stage checks this flag of the committed instructions to know when a new rgroup starts.

6.3 The Rmap logic without restrictions on the ROB state

The version of the Rmap logic presented in the section above uses the identifiers in the ROB, LQ, SQ and the physical register just as Rcreate specified them. Since the Rcreate logic assigns the identifier 0 in the ROB to the first instruction scheduled in an rgroup, Rmap must wait until this ROB entry is freed to begin processing an rgroup. Furthermore, the live-in registers are renamed to the physical register 0 (of the corresponding logical register). However, the source data may be actually in any physical register of the logical register's set. Therefore, this simpler version of the Rmap logic must wait until all the instructions present in the ROB commit, when the architectural register contains the source data for sure. Then the architectural register can be copied to the physical register 0 and the processing of the rgroup can begin. Such behavior implies many wasted cycles when changing from the Icache to the Rcache mode and also between two consecutive rgroups executed in the Rcache mode.

A related problem appears when the Rfetch logic does not find the next rgroup in the Rcache and the processor changes to the Icache mode. The instructions executed in the Icache mode must read the live-out registers of the rgroup from the physical registers where the architectural registers will be stored once all the instructions in the rgroup commit. In this version of the Rmap logic, which is this register is not know until the last instruction of the rgroup commits. So the Map stage in the Icache mode must also stall until no instruction of the previous rgroup is in-flight.

In summary, two problems must be solved to eliminate these stalls: a) how to use the identifiers assigned by the Rcreate logic while part of the ROB and some physical registers are still in use by instructions that do not belong to the rgroup, and b) how to update the internal state of the processor to be able to insert new instructions after the whole rgroup has been sent to issue but part of it hasn't committed yet.

6.3.1 Adapting the identifiers

The instructions in the rgroup have several identifiers assigned by the Rcreate logic: the identifier in the ROB, in the LQ or the SQ (if it is a memory instruction) and the identifiers of the source and destination physical registers. All these identifiers must be adapted by the Rmap logic to the actual state of the ROB, the queues and the register file.

The `ROB_tail` register is a pointer to the first free ROB entry. It is used in the Icache mode to assign the identifiers in the ROB. In the Rcache mode we will use it also to find the actual entry used

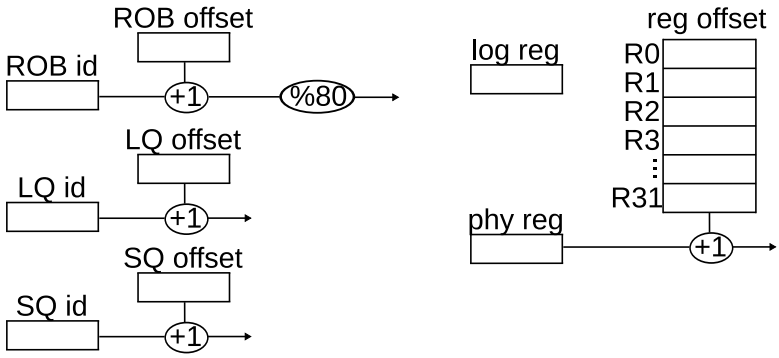


Figure 6.4: The Rmap logic must add an offset to the identifiers specified in the rgroup, before accessing the ROB, the LQ, the SQ and the physical registers. A modulo is applied to the add of the offset of the ROB, since the size of this structure is not a power of two.

by the instructions of the rgroups. The value of `ROB_tail` found when an rgroup starts execution will be used as an offset for the identifiers of that rgroup. The offset is added to the identifiers assigned by Rcreate, modulo *ROB_size*. The offset remains the same for all the instructions in the rgroup. The LQ and the SQ have similar pointers and the same mechanism is used.

An offset is also needed for each logical register in the register file. The Rcreate logic starts each rgroup assuming that the architectural values are in the physical register 0 of each logical register. But at execution time, the actual live-in values of the rgroup are in the currently mapped physical register of each logical register when the rgroup enters the Rmap logic. That mapping is indicated by the content of the `idx` fields at that moment (see section 8.1 for a detailed description of the register file). In the Icache mode, the `idx` fields are used to rename the registers. Therefore, the value of the `idx` field of each logical register found when the first instruction of an rgroup enters the Rmap logic is the offset that must be added to the identifiers assigned by Rcreate. Thus, all the `idx` fields are copied into the `reg_offset` table when a new rgroup enters the Rmap logic.

If the size of any of these structures is not a power of two (for example, the ROB has 80 entries in the default configuration of the ReLaSch processor), the modulo is calculated by additional logic after the addition. Figure 6.4 shows how the offsets are added to the identifiers in the Rmap logic.

6.3.2 Updating the structures

Although in the Rcache mode the `idx` fields are not used to rename the register, we want to update them at once after all the instructions of an rgroup have been processed by the Rmap logic. Thus, the next rgroup can use them as its own new offsets. Also, they will be used directly by the Map stage in case there is a change to the Icache mode. It happens the same with the `ROB_tail`, the `LQ_tail` and the `SQ_tail` registers.

In the Rcache mode, the entries in the ROB are occupied out-of-order, but after all the instructions have been processed by the Rmap logic, all the occupied entries are consecutive, with no holes between them. Just as if they had been assigned sequentially, the first free entry in the ROB after the rgroup has been processed is the entry just after the one used by the last instruction of the rgroup in program order. Rmap can calculate the identifier of this entry. Although the entries are assigned out of order, since all the instructions in the rgroup occupy an entry in the ROB, the size of the rgroup modulo *ROB_size* can be used to calculate the new value of `ROB_tail`.

For the LQ and the SQ, the Rmap logic can count how many of these instructions contains the rgroup. It would also do the same for each logical register, counting how many instructions in the

rgroup have it as destination register.

Alternatively, the rgroup could store how many loads and stores it has and the number of assignments of each logical register. These values would be added just once. Actually, it is not necessary to store the total number, but modulo *LQ_size*, *SQ_size* and the size of the sets of physical registers. For each rgroup in the Rcache, and using the values of the default configuration of the ReLaSch processor, the LQ would require 5 bits and the SQ another 5 bits (32 entries each). With 8 physical register per each logical register, 186 bits would be needed (3 bits per each one of the 62 logical registers).

Chapter 7

The Issue stage

The issue logic sends the available instructions to the functional units. First, the issue logic of the OoO processor is described. Second, the in-order issue-logic of the IO and the ReLaSch processors is introduced.

7.1 The out-of-order issue logic

The issue logic of the OoO processor uses two separated integer and floating point queues. Float-stores and float-to-integer instructions use both queues, whereas float-loads and integer-to-float use the integer queue only. The integer queue can hold up to 20 instructions and the floating point queue has room for 15 instructions.

One arbiter chooses up to four integer instructions. Older instructions are given more priority. An instruction in the queue is available if:

a. Its source registers are ready.

b. Its FU is available.

c. The aliased store has committed (only for loads that have a pending store of its store set).

Each queue entry independently checks the conditions for the instruction it stores. The selectable instructions send a request to the arbiter, that issues the oldest instructions that have an available FU.

Similarly, another arbiter chooses up to two floating point instructions. The instructions that use both queues must be selected by the two arbiters to be actually issued. With the separated queues, the integer arbiter doesn't need to access the floating point register file and its status bits.

Load hit speculation

The load instructions have a variable latency. The minimum latency of an access is three cycles when it hits the L1 cache. However, it is unknown whether the load hits the L1 or not until the end of the second cycle of the access. Therefore, in order to execute a dependent instructions as early as possible, the reference OoO processor speculates with the latency of the loads. It assumes that the loads will hit in the L1 and it issues the dependent instructions before the load latency is resolved. If the load hits, the data can be bypassed to the dependent instruction. If after two cycles it is discovered that the load actually misses in the L1, the instructions issued in the last two cycles are flushed from the functional units and re-issued in the next cycles.

To allow the re-issue of these instructions, the issue queues hold the instructions for two additional cycles after they are issued. The queue entries are freed and can be reused three cycles after the instruction is issued. In the rest of the text, the head of the buffer refers to the oldest instructions not issued yet, ignoring the issued instructions that may still be in the buffer waiting for load-hit resolution.

7.2 The in-order issue logic

In the in-order issue logic, the queues are replaced by buffers, maintaining the separation between integer and floating point instructions. Only the instructions in the head of the buffer can be issued. So instead of checking the readiness of all the 20 instructions in the queue and having an arbiter to choose four of them, there is a checking logic for just the four first entries of the integer buffer and the two first instructions of the floating point buffer.

Although each cycle only four and two instructions use the checking logic and can be issued, the issue buffer should be larger than just the size of one issue-group. The buffers are needed to decouple the Issue stage and the previous stage. They also enable load hit speculation by storing the instructions during two cycles after they are issued. In this chapter, the age of an instruction refers to when it was inserted. An instruction is older than another if it was inserted before. In the Icache mode, this order is equivalent to the program order but in the Rcache mode, the instructions are inserted in the schedule order. The head of issue buffer refers to the oldest instructions in the buffer that can be selected for execution. That is, the oldest four integer instructions and the oldest two floating point instructions.

The in-order Issue stage processes each buffer in-order but independently, unless an instruction is present in both buffers. Only the first instructions of each buffer are processed (four and two). For each of these instructions, the issue logic checks that :

a. The source registers are ready.

b. The FU is available.

c. All older instructions in the buffer have been issued or are issued in the same cycle.

d. The aliased store has committed (only for aliased loads).

e. The `new-issue-group` flag is cleared (except for the oldest instruction in the buffer).

When an instruction is present in both buffers, the two involved entries must be ready to allow issuing the instruction.

Figure 7.1 shows the access to the `valid` bits of the source register of one instruction. The register R7.1 is ready but R6.1 is not. Therefore, the instruction cannot be issued.

The `new-issue-group` flag is checked to avoid issuing together instructions of different issue-groups. When the instructions are independent, it would be correct to issue them in parallel when it is possible. Doing so typically improves IPC. However, if this is allowed an aliased store-load pair that the scheduler places into consecutive issue-groups on purpose can be issued simultaneously. Since both instructions arrive then to the Writeback stage at the same cycle, the data cannot be forwarded and the load is replayed. Our experiments (see details in section 9.4.1) show that the increase in load replays overweights the IPC gain, so the issue-group boundaries are respected. The `new-issue-group` flag is always 0 for the instructions executed in the Icache mode. The issue-logic of the IO processor does not use this flag at all.

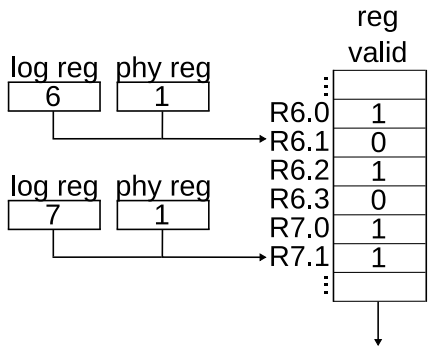


Figure 7.1: The `valid` bits of the register file indicate whether a given physical register is available.

7.2.1 Waking up the dependent instructions

The Alpha 21264 uses a scoreboard to know if the physical registers are available [18]. Each instruction in the issue queue access the scoreboard. Alternatively, the out-of-order issue logic [7] can use tag lines from the functional units to all the entries in the issue queue, to notify when the data can be bypassed.

In the ReLaSch processor, the check logic accesses the `valid` bit of the source physical registers for the instructions in the head of the issue buffer. To be able to bypass the data, the `valid` bit of the destination physical register is written two cycles (cycle $i-2$) before the end of the execution (at cycle i) of each instruction. Thus, the next cycle $(i-1)$ a dependent instruction can see that the `valid` bit of the register is set and it is issued, one cycle later (i) the dependent instruction receives the bypassed data and the next cycle $(i+1)$ can begin its execution. The register file is updated the same cycle $(i+1)$, and instructions issued from then on will read the register file. A dependent instruction that was issued in cycle i can one cycle later $(i+1)$ either read the bypassed value or the register file (if it is updated early in the cycle). To know where the source data is available, a bypass bit is set at the same time $(i-2)$ that the valid bit of the destination physical register. It is set until the last cycle when the bypass still can provide the data (that is, during 1 or 2 cycles). When an instruction is issued, the `valid` bit of its destination physical register is cleared.

There is a `valid` bit for each physical register of a logical register. In order to access it, the logical and physical register identifiers are used. The `valid` bits can be arranged either as a 1-bit table accessed with the concatenation of the two identifiers or as a table with one word per each logical register and then selecting the bit of the desired physical register.

7.2.2 Separated integer and floating point buffers

To ensure correct ordering of the instructions, some instructions are inserted in both the integer and floating point buffers (namely, the float-store and the float-to-int instructions). We call twin entries to the pair of entries used by that this kind of instructions. Such instructions are not allowed to be issued unless the check logic of both buffers assert that they are ready.

If an instruction is present in both buffers, the checking logic of its entry in the integer buffer must check that the twin entry of the instruction in the floating point buffer is in the head of the buffer and that the floating point checking logic asserts it as ready. Similarly, the floating point checking logic the readiness of the instruction in the integer buffer.

In general, it can be implemented assigning tags to the instructions in the issue buffers and performing a CAM-access to the entries in the head, that return a hit if there is a match and the instruction is ready.

Nevertheless, the CAM access is not actually needed, benefiting from the fact that the instructions are inserted in order in the buffers. If the oldest instruction in the floating point buffer has a twin in the integer buffer, it must be the oldest twin in the integer buffer. So actually it must check only whether there is a twin in the head of the integer buffer and if it is ready. If there are two twin instructions in the head of the floating point buffer, the second one can be issued if there are two ready twins in the integer buffer. The maximum number of twin instructions that are issued per cycle can be limited in order to reduce the complexity of this logic.

7.2.3 Tag broadcasting

In an out-of-order processor it is usual to broadcast the tags of producing instructions to all the instructions in the issue queue in order to wake up the dependent instructions. When an instruction is inserted in the issue queue, it must known in advance which results are already written in the register file and for which ones it has to wait the tag. The `rename` table can be extended to indicate if the renamed register is already available in the register file.

The ReLaSch processor could use also tag broadcasting of the instructions that complete execution. However, the renaming process cannot be used directly to obtain the pending results in a simple way. No `rename` table is accessed but just an offset is added to an already assigned identifier. Thus, an explicit access to the `valid` bits of the desired physical registers must be performed anyway after an instruction has been renamed. Furthermore, since only the instructions in the head of the buffer can be selected, it makes no sense to broadcast the tags to all the instructions in the buffer but just to the first four instructions in the integer buffer and the first two in the floating point buffer. Thus, the `valid` bits should be accessed once an instruction reaches the head of the buffer. Then, it becomes clear that the ReLaSch processor doesn't benefit from broadcasting the tags. Therefore, it simply accesses the `valid` bits of the source physical registers in order to know if an instruction is ready and it does it only when it is in the head of the issue buffer.

7.2.4 Conditional move instructions

The Alpha ISA includes conditional move instructions. With this kind of instructions, the destination register is also used as an implicit source register; the previous value of the register must be maintained if the condition is evaluated as false. However, the value is stored in a different physical register (the destination register of the conditional move), so it must be read from the current physical register and written in the new one.

The Issue logic checks the `valid` bit of the currently mapped physical register of the destination register before issuing a conditional move. The identifier of this register can be either stored in the rgroup with the information of the conditional move or it can be obtained subtracting a unit from the destination physical register. Since this field does not require extra space in the Rcache, the register is stored with the instruction in the rgroup.

Assuming that all the instructions in the issue buffers can be conditional move instructions, Three accesses to the `valid` bits would be required for each instruction in the head of the issue buffer. To avoid this situation, either the number of conditional moves can be limited or they can be implemented with a couple of standard two-source-register instructions, like the Alpha 21264 processor does (see B for details).

7.2.5 Issue buffer of issue-groups

The instructions of an issue-group are known to be independent, so it is not necessary to enforce them to be issued in order and an out-of-order issue logic could be used within an issue-group. The issue-group boundaries must be respected anyway to warrant the correctness of the execution. Thus,

the issue logic would process only the issue-group in the head of the buffer and its complexity would be much smaller than that of the issue logic of the OoO processor, that accesses the whole issue queue.

However, the issue logic must process also the instructions executed in the Icache mode, which often are not independent.

One possibility to ensure a correct execution is to consider each instruction as an independent issue-group, setting its `new-issue-group` bit. But this option unnecessarily penalizes the instructions executed in the Icache mode, since there are consecutive instructions are independent and can be issued together.

An alternative would be that the Map stage analyzes the dependences of the instructions it processes in the Icache-mode and sets the `new-issue-group` bit of the instructions whenever it is needed. That is, when an instruction depends on an older instruction that can be simultaneously in the head of the issue buffer. That is, an instruction that depends on any of the previous issue-width instructions.

For example:

```
A: ADD R3, R2, R1
B: ADD R6, R5, R4
C: ADD R9, R8, R7
D: ADD R1, R11, R10
E: ADD R4, R13, R12
```

Assuming that the issue logic can process up to four instructions per cycle, the instruction D must start a new Icache-mode issue-group, since it depends on A and both can be simultaneously in the head of the issue-buffer. Similarly, the instruction E depends on B, although the `new-issue-group` flag of D already prevents B and E to be processed simultaneously by the issue-group-based out-of-order issue logic, so E actually does not start a new issue group.

This solution makes the logic of the Map stage more complex but the instructions executed in the Icache mode are not penalized (beyond the fact that they have not been scheduled by the Rcreate stage). However, in some cases the instructions executed in the Icache mode still unnecessarily issue later than it would possible; when the first instruction in the Icache-mode issue-group stalls because it is not the oldest instruction processed by the issue logic this cycle, but its source registers are available and the instruction on which it depends was already issued. Using the same example as above, in a given cycle both instructions A and B are issued. The next cycle C, D and E could be issued, but only C is, because a new issue-group starts with D.

It is also possible to use an Icache-mode flag, to identify the instructions that must be issued in-order. However, then a more complex issue logic is needed, since both in-order and issue-group-based out-of-order issue logic are used, plus a choice logic to decide which issue logic must be used.

Similarly, two separated issue buffers and logic (for the Icache and the Rcache modes) can be used, but it makes the Issue stage more complex. Furthermore, to keep simple the choosing logic that decides which issue logic (Icache or Rcache) must process the instructions in a given cycle, it is possible that in some transition cycles no instructions can be inserted in the buffers, creating a bubble in the pipeline that hurts the IPC.

Due to all these problems, in our experiments we have only used an in-order issue logic for the ReLaSch processor. This is coherent with our principle of keeping the execution pipeline as simple as possible.

Chapter 8

Other elements

This chapter describes some other elements that have not been described in detail in the chapters above. The register file that is used in ReLaSch is completely different from the one used in the OoO and the IO processors. The Fetch, Map and Commit stages have some additional functionalities compared with those other processors in order to detect when the ReLaSch processor can switch to the Rcache mode and deal with the instructions executed in the Rcache mode when they commit.

8.1 The register file

The register file of the ReLaSch processor is based on the register file proposed in [10], where each logical register has its own set of physical registers. In our default configuration, all the sets of the physical registers have the same size (8 physical registers). This kind of register file simplifies both assigning the physical registers in the Rcreate logic and completing the rename in the Rmap logic.

Each logical register in the register file has two associated fields: `arch` identifies which physical register stores its architectural value and `idx` identifies the currently mapped physical register. The `idx` fields are equivalent to a conventional `rename` table. `idx` is used in the Icache mode to rename the source registers and in the Rcache mode to know the offset of each logical register.

In the Icache mode, the Map stage renames the source registers with the corresponding `idx` fields. The `idx` field of the destination logical register is incremented. The `arch` field of the destination register is incremented when the instruction commits.

In the Rcache mode, the rgroups already store the renaming information with the instructions. The `idx` fields are not needed then. The Rmap logic reads the `idx` fields of all the logical registers at once when a new rgroup starts execution, as explained in section 6.3.1.

Each physical register has two additional bits, `valid` and `busy`. The first one indicates whether the instruction that uses it as destination has already written the register, allowing the dependent instructions to read the value. The second one indicates whether the physical register is available to be assigned as destination or it is busy. The `busy` bit is set when a physical register *phy* is assigned as destination. When the instruction that writes *phy* commits, *phy* becomes the architectural register, so it still cannot be overwritten. When the next instruction that writes the same logical register commits, the value stored in the *phy* register is not needed anymore and its `busy` bit is finally cleared.

The issue logic uses the `valid` bit to know if a given source register is available and can be read. The Map stage and the Rmap logic check the `busy` bit to know if a given physical register can be used as destination.

When the pipeline is flushed, the `busy` bits of all the physical registers are cleared, except for those that store the architectural value. The `arch` field of each logical register indicates which are these

registers. The `arch` field is also used to update the `idx` field after a pipeline flush.

8.1.1 Common register file

The ReLaSch processor uses a register file that has a set of physical registers for each logical register. Such a register file can limit the performance achieved by the processor because there can be unused physical registers while the processor stalls waiting for a free physical register of a given logical register. Appendix A explains in detail the changes in the ReLaSch processor that are needed to use a conventional register file, that has a common pool of physical registers shared by all the logical registers. It also presents experimental results that show that the average performance achieved is very similar to that achieved by the default ReLaSch. Since the common register file requires more complex logic, we have decided to keep the set-based register file as the default for ReLaSch.

8.2 The Fetch stage

In the Icache mode, the Fetch stage accesses the Rcache and the Icache in parallel, with the PC of each fetched instruction. On a hit, the instructions that have been fetched and decoded but not mapped yet are removed from the pipeline and the processor switches to the Rcache mode. The Rfetch logic will then read the rgroup from the Rcache. The result of an access to the Rcache is not known until two cycles later, when the instruction that was fetched in parallel from the Icache enters the Map stage.

The Fetch stage updates a register with the history of the most recent branches. It is similar to the history register used by the Commit stage, but Fetch speculatively uses the outcome of the branch predictor to update the history register. The register is used to access the Rcache along with the current PC.

8.3 The Map stage

In the Icache mode, the Map stage is in charge of renaming the registers and inserting the instructions in the ROB. Memory instructions are also inserted in the LQ or the SQ. Besides, the Map stage inserts the instructions in the required issue buffer: integer, floating point or both. The identifiers in all these structures are assigned in order. It also checks that the desired destination physical register is free. All instructions are processed in order. If an instruction is not ready, e.g. because there isn't a free destination physical register, the ROB or the issue buffer is full, the Map stage stalls. No additional instructions are inserted in the ROB until the current instruction is ready and is sent to the Issue stage.

The `idx` field of the logical register is used to rename the instructions. The value of the `idx` field indicates the currently mapped physical register. The `idx` field of the destination register is incremented modulo the size of the set of physical registers. If the `busy` bit of that physical register is set, the stage stalls.

The instructions are inserted in the ROB, the LQ and the SQ in order. The stage increments the `ROB_head`, the `LQ_head` and the `SQ_head` registers when assigning an identifier in these structures. If the `busy` bit of the desired identifier is set, the stage stalls. It can only happen if the structure is full.

Assuming we are designing a superscalar processor, the stage processes more than one instruction per cycle. The renaming logic must check if there is any dependence between the instructions processed in the same cycle. However, it is not necessary to bypass that the same cycle the renamed register to a dependent instruction, since the in-order issue logic would stall the dependent instruction anyway in the next cycle.

Since we are using an in-order pipeline, register renaming is not actually needed except for WAW hazards, so it would be possible to substitute the Map stage for a simpler Insert stage. That stage would just insert the instructions in the ROB, LQ, SQ and issue buffers but it would not rename the registers. However, not renaming the registers introduces some problems regarding with the multi-cycle instructions and the use of separated integer and floating point queues, since in some cases the registers can be written out-of-order. To provide precise exceptions and be able to recover from mispredictions, the instruction would frequently stall to ensure that the registers are written in order and not speculatively. Alternatively, a structure like a future file [19] could be added to the register file to solve these problems. Since using the Map logic to rename the registers makes other parts of the processor simpler and uses already available resources instead of requiring additional ones (such as a future file), the ReLaSch processor has a Map stage that renames the registers in the Icache mode.

8.4 The Commit stage

The Commit stage of the OoO processor updates the architectural values of the register file, frees the resources used by the committed instruction and checks the correctness of the branch and memory aliasing predictions. It flushes the pipeline on a misprediction, correcting the speculative update of the branch predictor if necessary and performs the access to memory of the store instructions.

The Commit stage of the ReLaSch processor does all these tasks plus the following additional tasks:

a. Update the predictor for all the branch instructions executed in the Rcache mode (since the Fetch stage, that updates the predictor speculatively, didn't process the instructions in the Rcache mode).

b. Notify to the Rcache how do rgroups end (abruptly due a branch misprediction, a memory alias or to completion when the whole rgroup has committed).

c. Access the Rcache searching for a suitable rgroup after a pipeline flush.

d. Send the committed instructions to the Rcreate logic.

The branch predictor of the Fetch stage is only used in ReLaSch when the processor is in the Icache mode. In the Rcache mode, the branch prediction is performed in the Rcreate stage and stored with the rgroup. But to be accurate, a branch predictor needs to be updated with the last available information. Otherwise, it cannot adapt to changes in the behavior of the control instructions. So the control instructions that have been executed in the Rcache mode are also used to update the branch predictor. This is performed when the instructions commit because: a) the information is not speculative; b) the order of the branches in the schedule can be different from the program order, so the history information stored in the branch predictor can be updated with wrong information if it is updated in the Rfront-end. The branch predictor will not be used until the processor changes to the Icache mode, so there is no need for a fast update. It can take many cycles if required.

The Commit stage detects when an rgroup begins and finishes execution and notifies the end of each rgroup to the Rcache. To do so, the ROB entries have a flag to indicate whether an instruction is the first one of an rgroup, and another to indicate that they are executed in the Rcache mode. The ROB entry of the first instruction of an rgroup also stores the identifier of the rgroup The identifier of an rgroup is formed by a PC and history bits. The Commit stage already knows the PC and the next-PC of the instructions it processes. However, it cannot deduce the history bits that identify an rgroup because they are used only as a hint in the Rcache and they can differ from the actual history. When the Commit stage detects that the instruction in the head of the ROB (which is ready to commit) is the first one of an rgroup, it stores the identifier of the rgroup in a dedicated register.

When the Commit stage detects that an instruction is the last one of an rgroup, the rgroup has been completely executed. That happens when the next instruction is executed in the Icache mode or is the first one of a new rgroup. In that case, the Commit stage accesses the Rcache with the identifier of the rgroup. The Rcache updates the saturating counter associated with the rgroup. If the rgroup hasn't finished but a mispredicted branch commits or an aliased load that must be re-executed is processed by the Commit stage, the Rcache is accessed with the identifier of the rgroup to notify that the rgroup wasn't completely executed. The Commit stage also tracks whether a load instruction in the rgroup has missed in the L2 cache. In this case, the counter of the rgroup is decremented even if the rgroup is completely executed.

The Commit stage has a history register updated with the outcome (taken/not-taken) of the last conditional branches. It also updates a register with the architectural value of the PC after each committed instruction. Whenever there is a pipeline flush, the Commit stage uses the history and PC registers to access the Rcache and check if there is any available rgroup that begins with the next instruction that must be executed. On a hit, the Rfetch logic reads the rgroup; otherwise, the processor executes these instructions in the Icache mode.

The committed instructions are sent to the Rcreate logic, where they are scheduled. The instructions are inserted in the `rcreate_input` buffer; along with an instruction, it is also stored its PC and whether it was executed in the Rcache or the Icache mode. For some types of instructions, extra information from the execution is stored. Namely, for a memory instruction it is stored which address it accessed, the latency of the access and whether the access hit or missed in the L1 cache. Besides, the control instructions store if they were taken and their target PC.

Chapter 9

Design space and final results

This chapter presents an exploration of the design space of the ReLaSch processor. The ReLaSch Processor has a lot of parameters and most of them are not independent. Besides, a parameter can become predominant and hide the effect of other parameters. We have realized that a given parameter that has an important impact in performance with a given processor configuration, becomes nearly irrelevant with a different configuration.

We have followed an iterative approach: we choose first a default configuration and then try to refine it, by varying each parameter while fixing the rest, and choose the best design-point for that parameter. The resulting configuration becomes the new default configuration and the process begins again. This approach is a consequence of the evolution of the design, where new elements and techniques have been added progressively. The new parameters have been explored in the next iterations, possibly affecting the impact of the already explored parameters.

In this chapter we show the final iteration of this process. It begins with an already refined default configuration and all the parameters are explored individually while fixing the others. When a parameter can be highly dependent on another one (i.e. the size and the associativity of the Rcache), we vary both parameters simultaneously. We have grouped the explored parameters into sections according to the logic, block or stage that is more closely related to the parameter.

Section 9.6 shows the experimental results of the new default configuration.

9.1 Experimental set-up

We have modified the sim-alpha simulator [20] to model the ReLaSch and the reference OoO and IO processors. Sim-alpha is based on Simplescalar and was configured and validated against a real Alpha machine, the Compaq DS-10L Alphaserver [21]. The reference processors are much like an Alpha 21264 [15], enhanced with Store Sets [16] and an improved BTB, similar to the Intel Pentium M processor's target predictor [22], using path instead of history since it works better with our benchmarks [23]. The ReLaSch processor doesn't need to use the Store Sets and the improved BTB, since the rgroups already solve the same problems. So ReLaSch uses the simpler original Alpha BTB and StWait bits.

Table 9.1 shows the main simulation parameters. Any other parameter maintains the default sim-alpha/21264 value [20].

We use most of the SPECcpu2000 benchmarks [24], the eight missing ones (*apsi*, *perl-perf*, *sixtrack*, *vpr-place*, *wupwise* and the three *vortex* benchmarks) had compilation problems. The benchmarks were compiled with optimization flags. We used the default SPEC configuration for the peak performance evaluation, in which the optimization flag used varies depending on the benchmark. Most C benchmarks use -O4 while Fortran benchmarks use -O4 or -O5. The *eon* benchmark uses -O2 and

Parameter	OoO	IO	ReLaSch
Issue width: 4 Integer, 2 FP	*	*	*
Issue queue: 20 Integer, 15 FP	*		
Issue buffer: 20 Integer, 15 FP		*	*
ROB: 80 entries	*		*
Load Queue: 32 entries, Store Queue: 32 entries	*		*
Integer FUs: 4 alu, 4 multiply	*	*	*
Floating point FUs: 1 alu, 1 multiply	*	*	*
Data L1 cache: 2-way 64KB 3-cycle hit latency	*	*	*
Instruction L1 cache: 2-way 64KB 3-cycle hit latency	*	*	*
L2 cache: 2MB direct mapped 13-cycle hit latency, minimum 84-cycle miss latency (extra cycles if bus contention)	*	*	*
DTLB: 128 entries, ITLB 128 entries	*	*	*
Branch predictor: 4Kx2 choice predictor 4Kx2 global predictor 2-level local predictor (1K 10-bit history, 1K 3-bit counters)	*	*	*
BTB: 1024-entry 4-way PC-indexed	*	*	*
32-entry RAS	*	*	*
multi-target BTB: 1024-entry 4-way path-indexed	*	*	
Store Sets: 4K-entry SSIT, 128-entry LFST, 7-bit identifiers	*	*	
StWait: 1024 1-bit table			*
Register file: 72 Integer, 72 FP shared physical registers	*		
Register file: 8 physical registers per logical register			*
Rgroup: 256 instruction, 10 indirect branches			*
Rcreate: 4 instructions per cycle, 256 5-bit load latency predictors schedule table of 512 issue-groups `rcreate_input` buffer with 512 instructions			*
Rcache: 4 ways, 32 sets, 1897B per rgroup one 5-bit "bad rgroup" counter per line			*
Rcache latency: 3 cycles to read the fist issue-group one issue-group per cycle afterwards			*

Table 9.1: Main simulation parameters for the OoO, IO and ReLaSch processors.

perl has -O3 in the default SPEC peak performance configuration.

For each benchmark, a 100M-instruction segment is simulated. SimPoint [25] was used to find the most representative segment of each benchmark. Unless otherwise stated, the figures in this section show the speed-up in IPC obtained by the ReLaSch processor over the OoO processor, in order of increasing speed-up. The results of the FP and the INT benchmarks are shown in separated figures. Each bar starts at 1.0 speed-up. Speed-up lower then 1.0 indicates a performance loss with respect to the reference OoO processor. The default configuration is indicated in each figure with a **bold** label in the legend. Each figure is followed by a table that summarizes the average speed-up achieved in the INT and FP benchmarks and the average of all the benchmarks (ALL). In some cases, the table includes the results of some additional configurations that have been tested but that have been removed from the figures to simplify them. Some sections reference additional tables that present more details on the results. all these tables can be found in appendix C.

The simulator fetches the instructions from the binary even from the wrong path on a mispredicted branch. Although the simulator is not trace-driven, we use the EIO (External Input-Output) traces of the sim-alpha/simplescalar simulator to skip the execution of the instructions before the interval chosen by SimPoint. To create the EIO trace, first a complete functional simulation is performed and the architectural state of the processor at the beginning of the interval is stored in the trace. The trace includes the content of the registers and the memory addresses that have been written up to that moment. This state is read at the beginning of the simulation and the registers and memory are updated with the content of the file.

This is enough to execute the instructions of the interval, as long as there is no system call. The behavior of system calls depends on some status of the OS, such as the opened files, that is not part of the processor's state, so the system calls cannot be reproduced just from the state of the processor at the beginning of the interval. Therefore, when the trace is created, the whole interval is also simulated and the state of the processor is written not only at the beginning of the interval but also after each trap instruction in it.

Once the trace is created, the simulator read the EIO trace and updates the state with the content of the trace on a system call. The pipeline is flushed afterwards and instruction resumes with the instruction that follows the system call.

9.2 The Rcreate logic

This section presents the exploration of many parameters related with the Rcreate logic: the size of the rgroups, the `sched` table and the `rcreate_input` buffer, the memory latency predictor, the number of indirect branches per rgroup and the latency and the width of the Rcreate logic.

9.2.1 Rgroup size

Figure 9.1 shows the IPC speed-up achieved using several rgroup sizes (that is, the maximum number of instructions per rgroup). The size of the Rcache is approximately the same in all cases: when we double the number of instructions per rgroup, the total number of rgroups is halved as a consequence. So the Rcache stores up to 32K-instructions in each case. All these configurations of the Rcache are 4-way associative. The exact size in bytes differs due to overheads that depend on the total number of rgroups in the Rcache. For instance, the tags or the information about the branches of each rgroup, since the number of branches per rgroup is the same for all the configurations. The minimum size explored is 64 instructions per rgroup. In this case, the cache has 128 sets, so it stores up to 512 rgroups. The maximum size in the figures is 1024 instructions per rgroup, with an 8-set Rcache.

For the INT benchmarks, the scheduler extracts more ILP from larger rgroups. At the same time, th INT benchmarks also require having a high number of rgroups available in the Rcache. Thus, the

highest average performance is reached with the 256-instruction rgroups, where both parameters are balanced.

However, we find some benchmarks that have higher performance with 128-instruction rgroups: *eon-kajiya*, *twolf*, *crafty*, all the *gcc* benchmarks and most *gzip* and *bzip* benchmarks. These benchmarks have in common a high branch misprediction rate (see table C.9). The high number of taken paths leads to create many rgroups corresponding to those paths. Furthermore, in some benchmarks often only a prefix of the complete rgroup commits since a branch included in the rgroup is mispredicted. This leads to have a lower percentage of instructions executed in the Rcache mode (see table C.10). Nevertheless, in all cases the difference with the speed-up achieved with 256-instruction rgroups is small.

We find also the opposite case, when benchmarks perform better with 512-instruction rgroups: *mcf*, *vpr-route* and *perl-make*. The two latter benchmarks are more regular and benefit above all of having large rgroups. *mcf* has very high L1 and L2 cache miss-rates (59.6% L1 miss-rate and a total of 25M misses and 51.1% L2 miss-rate), so it spends most of the time simply waiting for the data in memory. In this situation, having a larger scope is beneficial. However, the impact in the absolute IPC is small. It just varies from 0.1432 to 0.1635. The speed-up achieved with this configuration is close to the speed-up achieved with 256-instruction rgroups in these cases too.

Almost all the FP benchmarks perform better with larger rgroups, so the highest average speed-up is achieved with 1024-instruction rgroups. Nevertheless, there is small difference with the speed-up achieved with 512-instruction rgroups. The exceptions are *ammp*, *lucas* and *mesa*, that achieve higher speed-ups with smaller rgroups.

From these results, it is clear that 256 instructions per rgroup yields the highest average speed-up (1.00). However, it seems worth considering a system with an adaptive rgroup size, where we can join two cache lines to hold a single, double-size rgroup when it is beneficial. A preliminary study for this has been presented in [14]. The results of the best configuration for each benchmark are combined, assuming that the compiler can detect this situation and set the rgroup size statically for each binary. The average results do not vary much (2% higher speed-up) but many benchmarks (29 out of 40) improve their results.

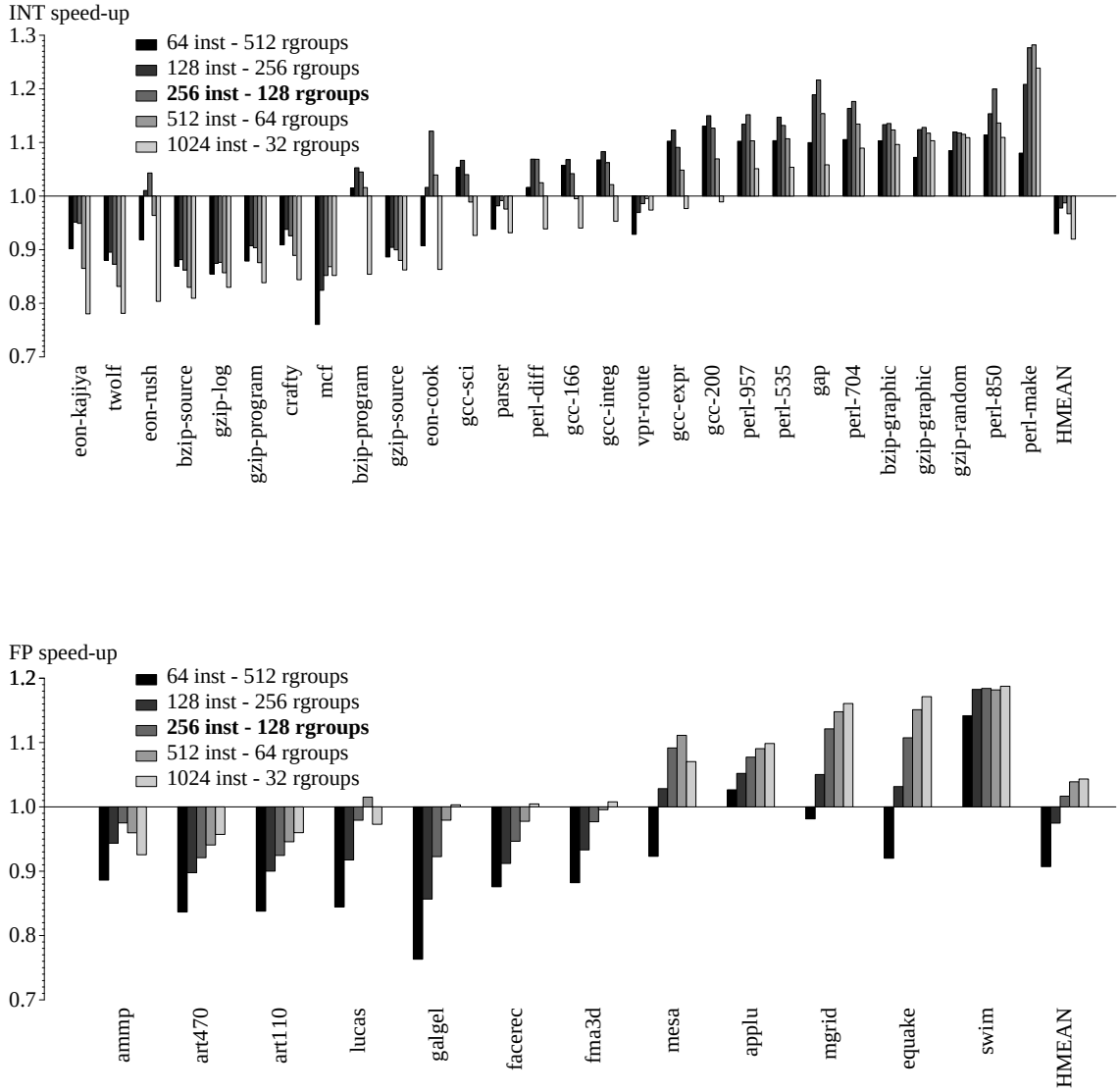
Small Rcache

In order to study the impact of the size of the Rcache in the results above, we have performed the same experiments with a smaller Rcache. In these experiments the Rcache stores up to 4K instructions, so it is eight times smaller than in the section above.

Figure 9.2 shows the results. The same patterns seen in the section above appear here even clearer. The FP benchmarks benefit from larger rgroups even with a small Rcache. The best average FP speed-up (0.99) is achieved with a 4-way 2-set Rcache with rgroups of 512 instructions. For the INT benchmarks it is better to use the Rcache with 128-instruction rgroups since it allows storing more different rgroups. It yields 0.87 average speed-up. The percentage of instructions executed in the Rcache mode drops drastically with the larger rgroups for the INT benchmarks, from 83% with 64-instruction rgroups to 56% with 1024-instruction rgroups (see table C.12 for more details).

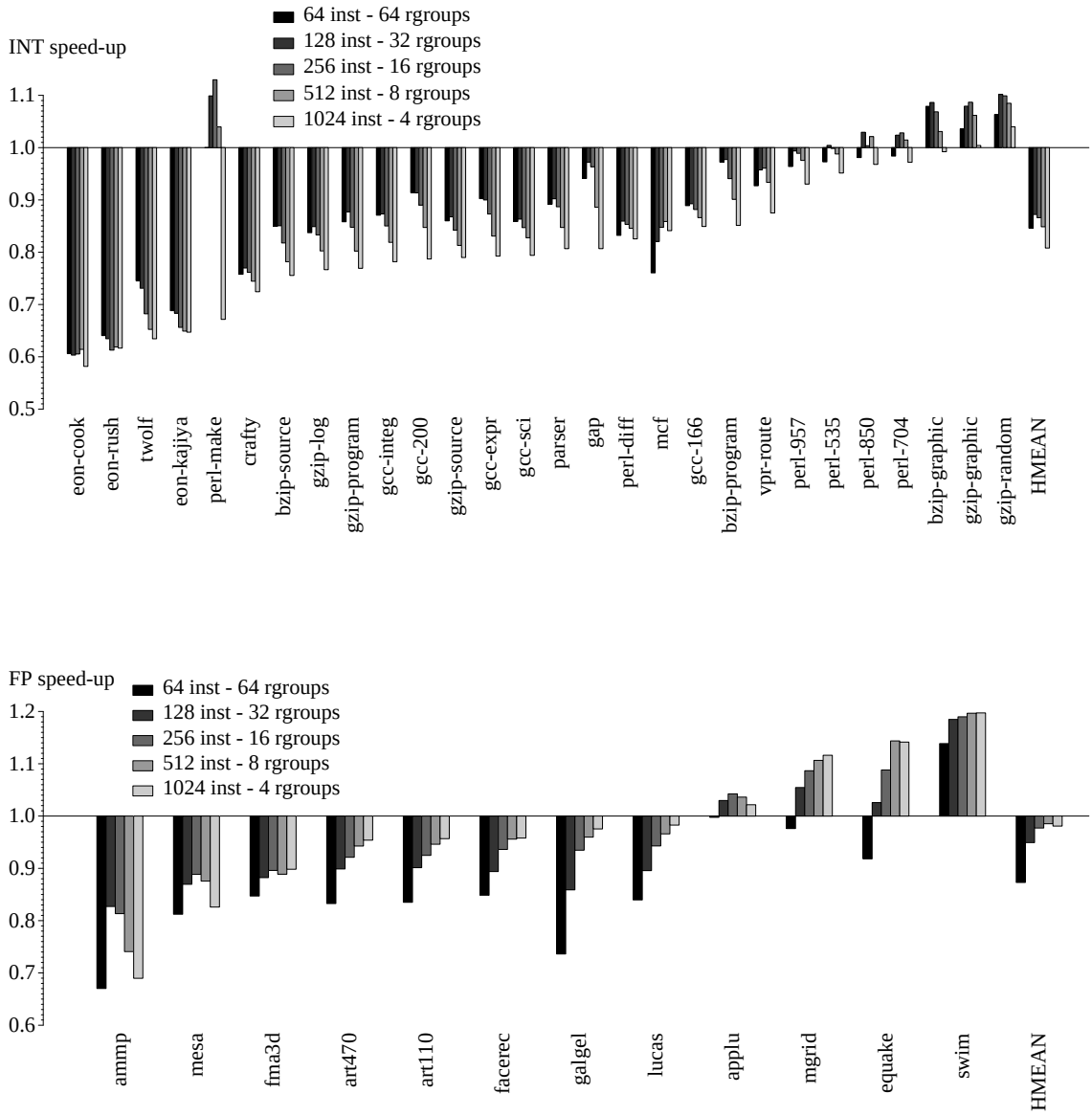
The INT figure shows that up to 5 applications achieve the highest performance with 64-instruction rgroups (*eon-rush*, *twolf*, *eon-kajiya*, *gcc-200* and *gcc-expr*). On the contrary, only 2 applications have their highest speed-up with more than 256-instruction rgroups (*eon-cook* and *mcf*). Nevertheless, the difference above the 256-instruction speed-up is small.

In the FP figure, one application finds its highest performance with 128-instruction rgroups (*ammp*) and two with 256-instruction rgroups (*mesa* and *applu*). All other applications yield their highest speed-up with larger rgroups.



instructions - rgroups	64 - 512	128 - 256	**256 - 128**	512 - 64	1024 - 32
HMEAN INT	0.93	0.98	0.99	0.97	0.92
HMEAN FP	0.91	0.97	1.02	1.04	1.04
HMEAN ALL	0.92	0.98	1.00	0.99	0.96

Figure 9.1: Speed-up with different rgroup sizes.



instructions - rgroups	64 - 64	128 - 32	256 - 16	512 - 8	1024 - 4
HMEAN INT	0.85	0.87	0.87	0.85	0.81
HMEAN FP	0.87	0.95	0.98	0.99	0.98
HMEAN ALL	0.86	0.90	0.90	0.89	0.86

Figure 9.2: Speed-up with different rgroup sizes in a small Rcache.

9.2.2 `sched` table

Figure 9.3 shows the IPC speed-up achieved using several sizes of the `sched` table (which defines the maximum number of issue-groups per rgroup), from 64 to 2048 issue-groups. The maximum number of instructions per rgroup is 256 in all examined configurations.

Using rgroups with at most 64 issue-groups limits a lot the IPC achieved by the ReLaSch processor. The scheduler closes an rgroup if the current instruction must be scheduled beyond the size of the `sched` table. With only 64 issue-groups in the `sched` table, the rgroups are scheduled with less than 256 instructions in many cases and the scheduler extracts less ILP. Note that, in order to schedule 256 instructions in 64 issue-groups, the scheduler must find four independent instructions per cycle in average, which is usually not possible. Table C.13 shows the average size of the rgroups created by the scheduler with several sizes of the `sched` table. With an `sched` table larger than 64 entries, the scheduler is usually able to find enough independent instructions to fill the rgroup. With a `sched` table of 256 issue-groups this parameter does not limit the performance of most benchmarks, which do not benefit from larger `sched` tables. The most notable exception is the *mcf* benchmark. It has a very high L2 miss rate which lowers its IPC to 0.14 in the OoO processor. Benchmarks with a high number of L2 misses and high latencies benefit from larger `sched` tables, since the resulting schedule is sparse.

This parameter has limited impact on some applications (i.e. *crafty*, *parser*, *bzip-program*, *gcc-integ*, *gzip-random*, *gcc-200*, *gzip-graphic* and *bzip-graphic*). All these applications have high branch misprediction rates, so executing short rgroups don't penalize them to a great extent.

9.2.3 Number of indirect branches

Figure 9.4 shows the speed-up using different maximum number of indirect branches per rgroup, including function calls and return instructions.

Using a maximum of more than 8 branches doesn't have much impact (the average speed-up does not grow beyond the 7-branch maximum). A maximum of less than 6 branches severely hurts the IPC of some benchmarks: all the *eon* and *perl* benchmarks along with *gap*, *mesa* and several *gcc* variants (*gcc-sci*, *gcc-166*, *gcc-expr* and *gcc-200*). The *perl* applications use indirect branches intensively. *eon* is an object-oriented application. Table C.14 shows the number of indirect branches executed during simulation of each benchmark.

9.2.4 Data cache latency prediction

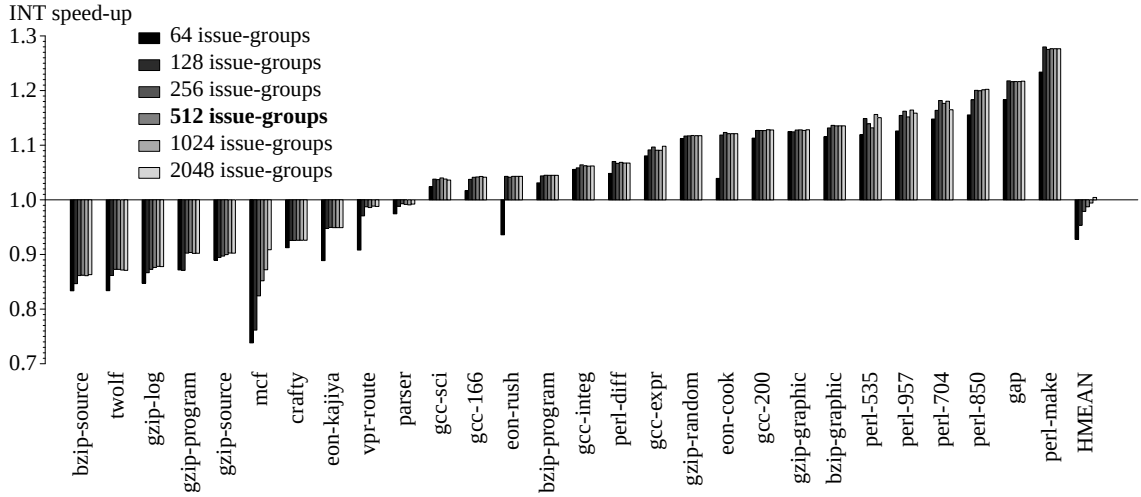
This section presents a study on how to predict the latency of the load instructions at scheduling time.

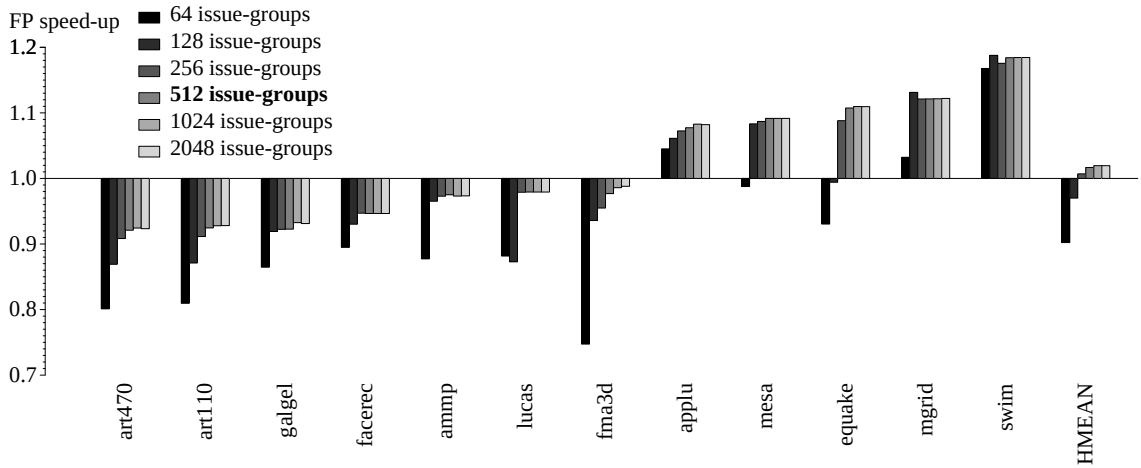
Figure 9.5 shows the speed-up achieved with different prediction schemes. On most configurations a table of saturating counters is used. The figure shows the results achieved with different table sizes. There are two configurations in which the latency is predicted using a fixed rule instead of a predictor structure: 1) the loads are always predicted to hit in the L1, so each one is scheduled assuming a latency of three cycles ("always hit" in the figure); 2) all loads are always scheduled using the latency seen at execution time ("exec lat" in the figure).

The table shows the average speed-up of some additional table sizes too.

The benchmarks show different behaviors: some of them (i.e. the INT applications *parser*, *vpr-route*, *gzip-random*, *gzip-graphic* and *perl-535* and the FP applications *galgel* and *applu*) work better with a small number of counters or the "always hit" configuration while others (i.e. the INT application *perl-make* and the FP applications *facerec*, *lucas*, *equake*, *mgrid*) benefit from having a higher number of counters or from using the execution latency always.

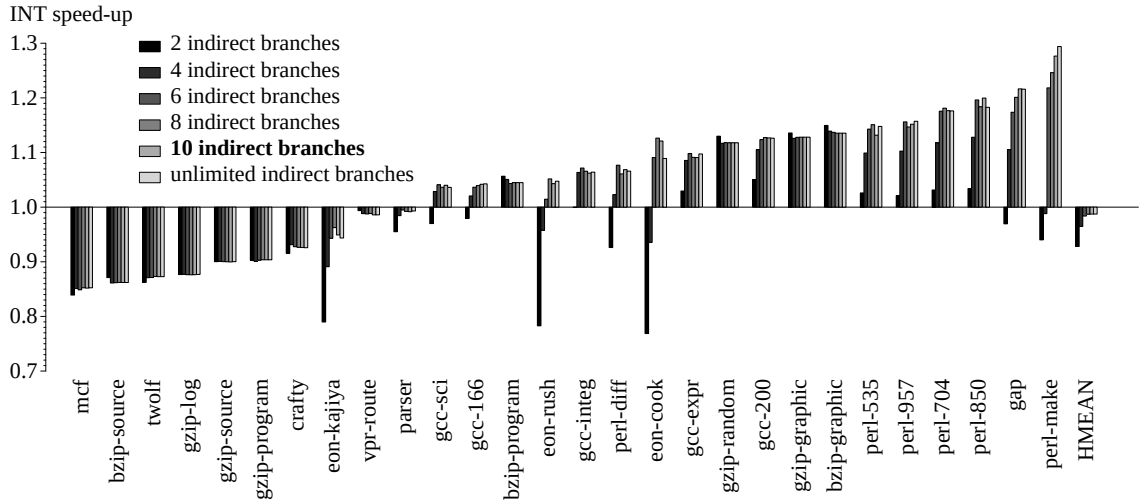
Table C.15 shows the miss rate in the data L1 cache and the L2 of all the benchmarks in the default configuration of ReLaSch.

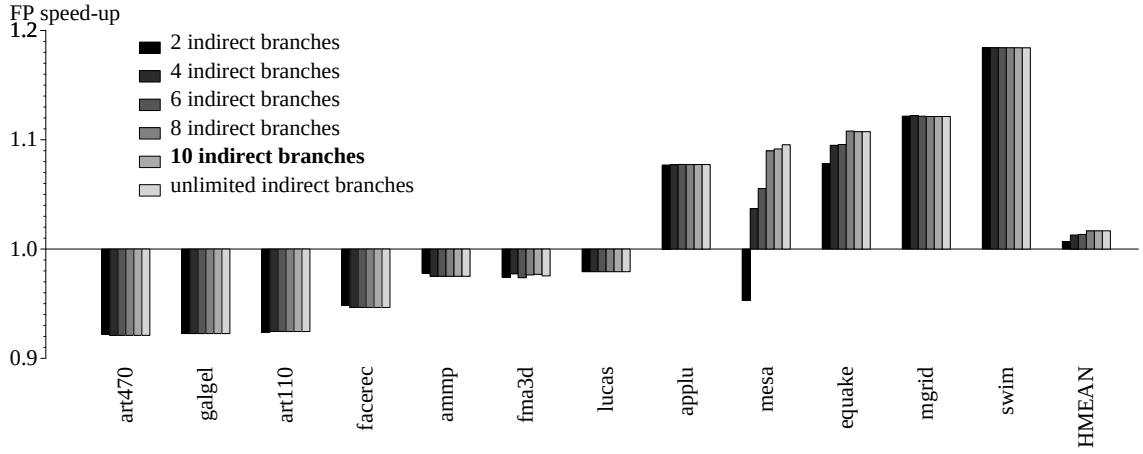




issue-groups	64	128	256	**512**	1024	2048
HMEAN INT	0.93	0.95	0.98	0.99	0.99	1.00
HMEAN FP	0.90	0.97	1.01	1.02	1.02	1.02
HMEAN ALL	0.92	0.96	0.99	1.00	1.00	1.01

Figure 9.3: Speed-up with different `sched` table sizes.





indirect branches	1	2	3	4	5	6	7	8	9	**10**	unlimited
HMEAN INT	0.87	0.93	0.96	0.96	0.98	0.98	0.99	0.99	0.99	0.99	0.99
HMEAN FP	1.00	1.01	1.01	1.01	1.02	1.01	1.02	1.02	1.02	1.02	1.02
HMEAN ALL	0.91	0.95	0.97	0.98	0.99	0.99	1.00	1.00	1.00	1.00	1.00

Figure 9.4: Speed-up with different maximum number of indirect branches per rgroups.

With respect to the predictor-less approaches, predicting always a hit usually hurts the performance of the benchmarks with a significant number of misses (i.e. *mcf*, *art-470*, *art-110*, *lucas*, *fma3d* and *equake*). On the contrary, some benchmarks (*gzip-log*, *gzip-source*, *gzip-program*, *perl-850*) have lower IPC when using always the execution latency. As mentioned above, in some cases the predictor-less approaches yield the highest speed-up. Nevertheless, for most examples the difference with the closest competitor is negligible. The exception is the FP application *applu*, that exhibits significant better performance with the "always hit" policy than with any other approach.

In general, the FP benchmarks benefit from using a higher number of counters or using always the execution latency. The opposed responses of the different INT benchmarks counteract each other, so the average speed-up does not change significantly with the number of counters. The "always hit" policy yields less speed-up in average as well as using always the execution latency, though in this case the difference is smaller. In order to maximize the overall, we choose to have a reduced number of predictors and discard the predictor-less approaches.

Data cache latency predictor parameters: number of bits per counter

Figure 9.6 shows the speed-up with different number of bits in each counter of the memory latency predictor. The threshold used in each case implies testing the left-most bit: 16 for the 5-bit counter, 8 for the 4-bit counter, etc. This parameter has almost no impact in performance, just the INT benchmarks *perl-535* and *perl-850* show a slightly better speed-up with 1, 2 or 3 bits than with larger counters.

Therefore, it seems better to use 1-bit counters instead of the larger counters of the current default configuration.

Data cache latency predictor parameters: threshold value

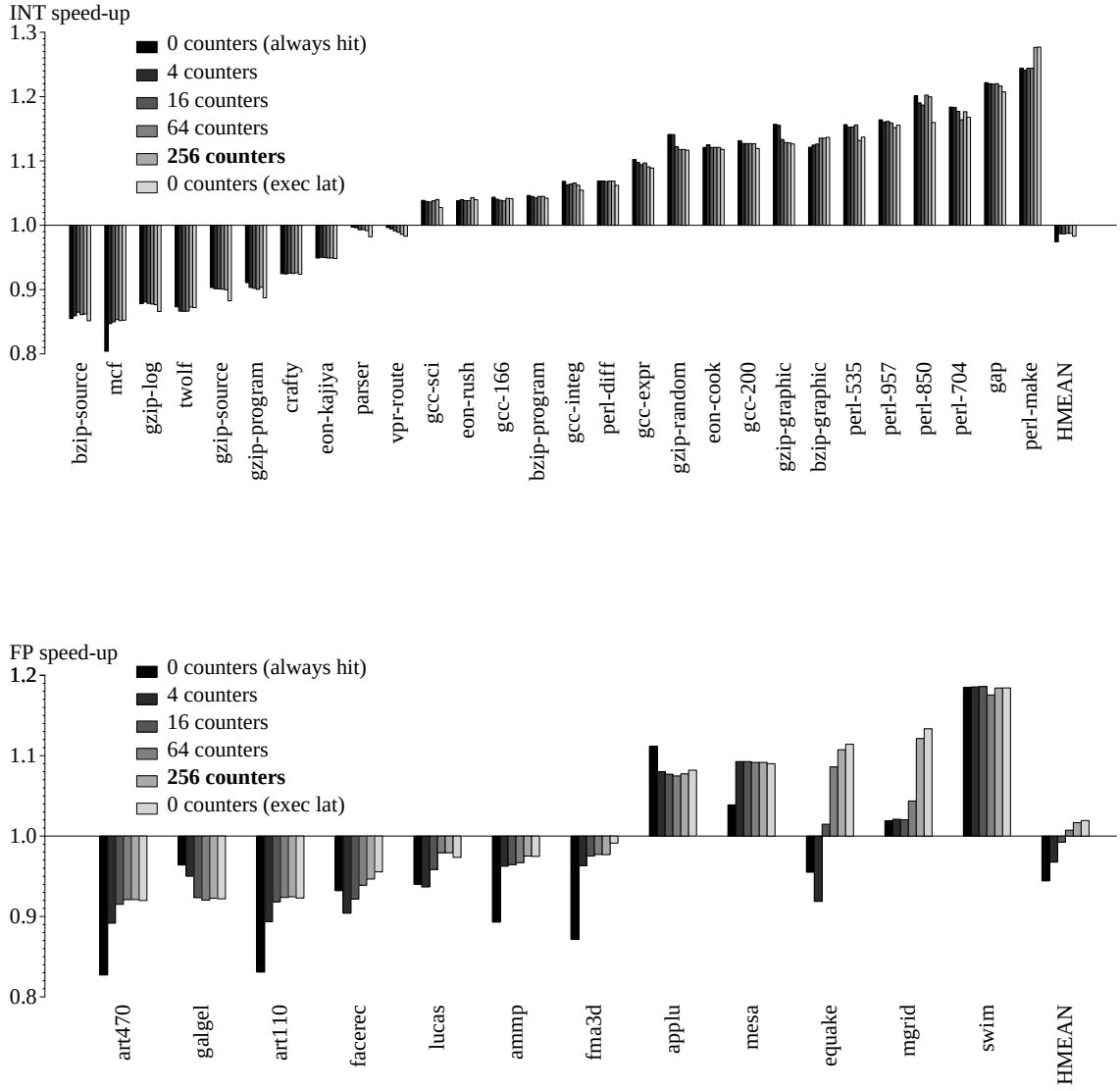
Figure 9.7 shows the speed-up using five-bit counters, with different prediction thresholds; that is, the lowest value that predicts a hit latency. The threshold indicates how many bits must be checked to perform the prediction. A threshold of 16 implies testing whether the left-most bit is zero. With a threshold of 8, the two left-most bits are tested. The speed-up of most benchmarks is almost independent of this parameter. FP benchmarks *fma3d* and *mgrid* benefit from higher thresholds, while several INT benchmarks (*gzip-log*, *gzip-source*, *gzip-program*, *parser* and *perl-850*) benefit from lower thresholds. The INT benchmark *perl-535* shows mixed behavior. If one-bit counters are used, this parameter is not relevant anymore.

Data cache latency predictor parameters: updating policy

Figure 9.8 shows the speed-up achieved with different methods for updating the counters of the memory latency predictor. The default configuration increases the 5-bit saturating counter on an L1 hit and shifts it to the right on a miss. This approach gives more weight to the misses. Decreasing on a miss and shifting to the left on a hit in order to weight more the hits is also considered here. Just increasing and decreasing is also evaluated, where hits and misses have the same weight. Finally, the results of the predictor-less approach "exec lat" are also shown as a reference.

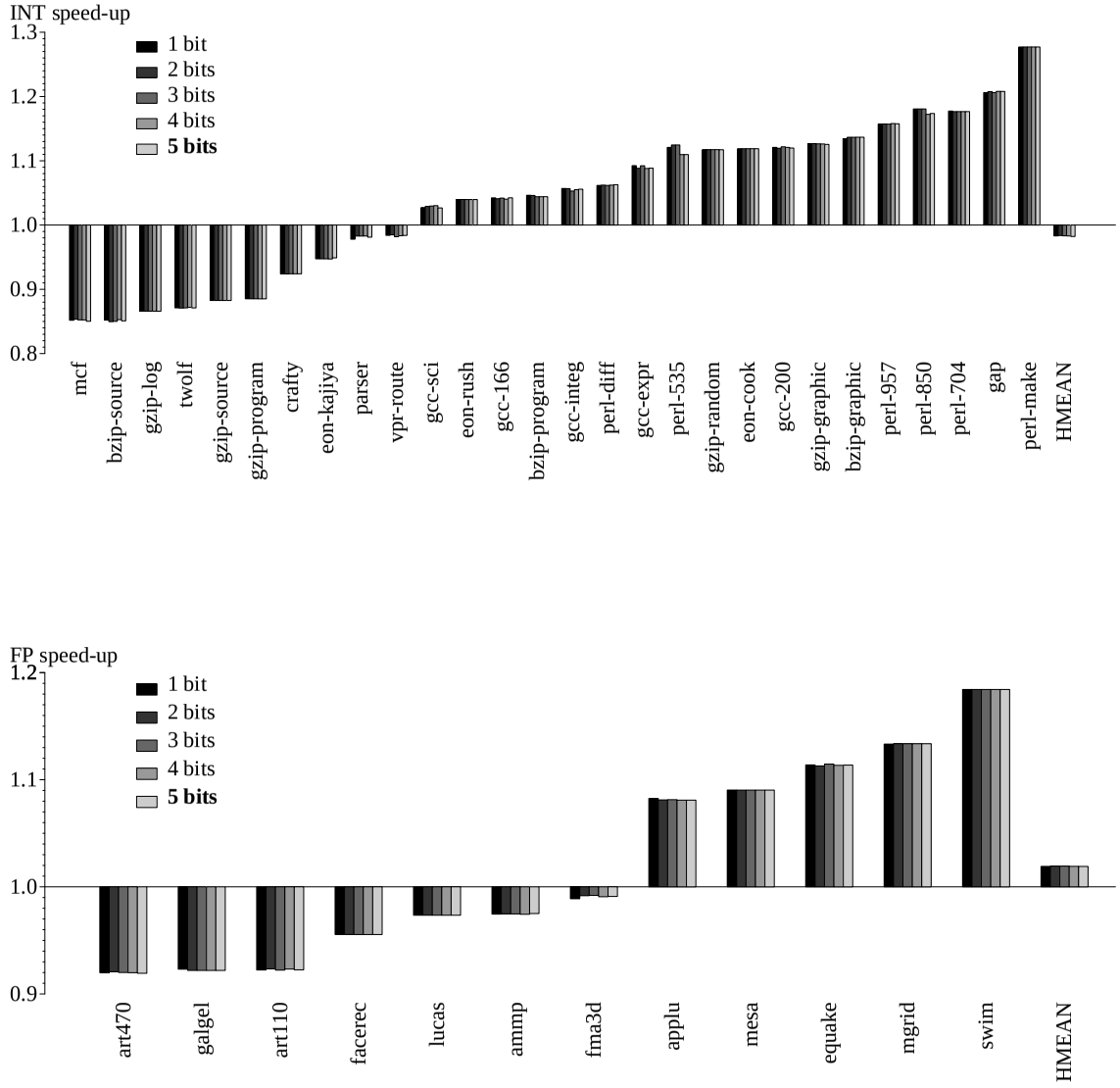
Some benchmarks show significant variation. The INT applications *bzip-source*, *gzip-log*, *gzip-source*, *gzip-program*, *gcc-sci*, *gzip-random*, *gzip-graphic*, *perl-850* and *perl-make* yield better performance if misses have more or equal weight. FP applications *facerec*, *lucas*, *fma3d*, *equake* and *mgrid* have better results when misses have more weight. *ammp* is the exception since it has a slightly better result if hits have more weight.

This parameter does not have a significant impact on the average performance. Furthermore, it does not have sense if one-bit counters are used.



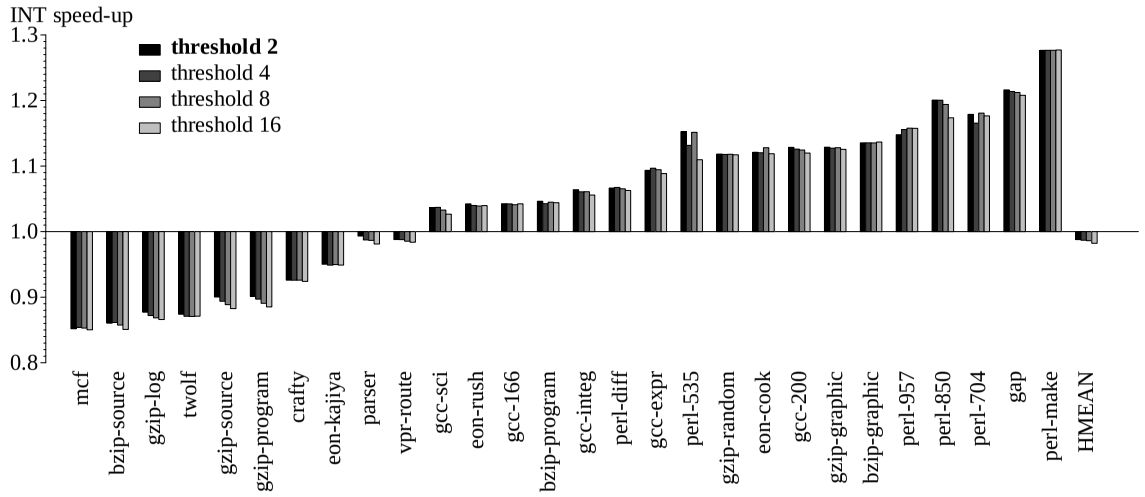
counters	hit	1	2	4	8	16	32	64	128	**256**	512	exec
HMEAN INT	0.97	0.98	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.98
HMEAN FP	0.94	0.97	0.97	0.97	0.97	0.99	1.00	1.01	1.01	1.02	1.02	1.02
HMEAN ALL	0.96	0.98	0.98	0.98	0.98	0.99	0.99	0.99	0.99	1.00	1.00	1.00

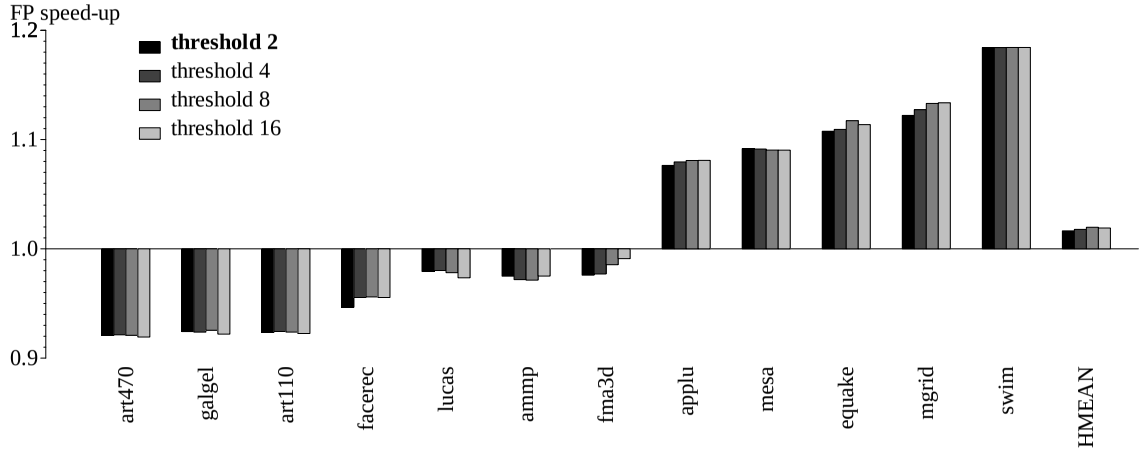
Figure 9.5: Speed-up with different sizes of the memory latency predictor.



counter size	1	2	3	4	**5**
HMEAN INT	0.98	0.98	0.98	0.98	0.98
HMEAN FP	1.02	1.02	1.02	1.02	1.02
HMEAN ALL	1.00	1.00	1.00	0.99	0.99

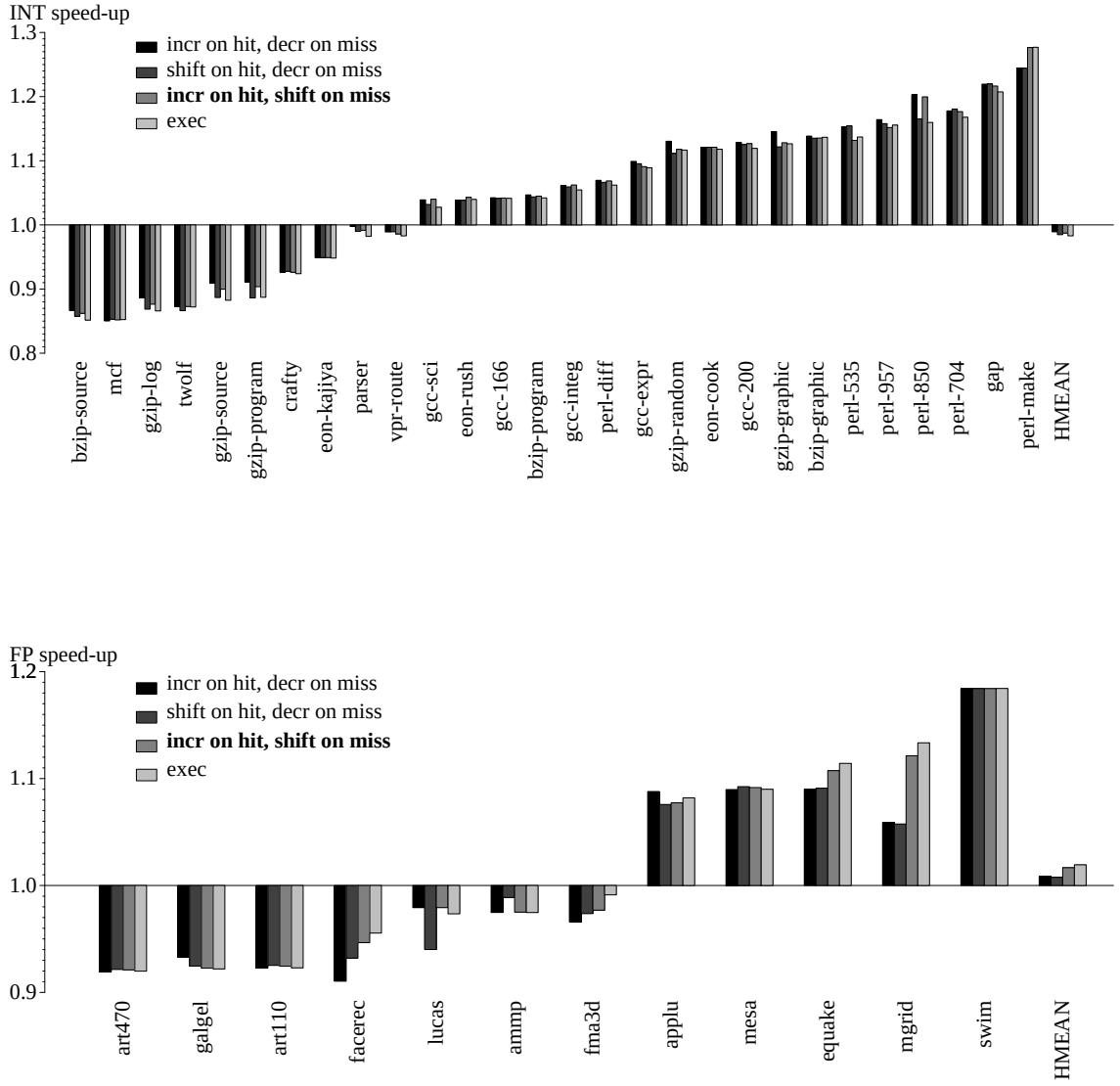
Figure 9.6: Speed-up with different sizes of the counters of the memory latency predictor.





threshold value	**2**	4	8	16
HMEAN INT	0.99	0.99	0.99	0.98
HMEAN FP	1.02	1.02	1.02	1.02
HMEAN ALL	1.00	1.00	1.00	0.99

Figure 9.7: Speed-up with different threshold values of the counters of the memory latency predictor using 5-bit counters.



policy	+1 on hit, -1 on miss	sh. on hit, -1 on miss	**+1 on hit, sh. on miss**	exec
HMEAN INT	0.99	0.98	0.99	0.98
HMEAN FP	1.01	1.01	1.02	1.02
HMEAN ALL	1.00	0.99	1.00	1.00

Figure 9.8: Speed-up with different methods for updating the memory latency predictor.

9.2.5 Pipelining Rcreate

Figure 9.9 shows the speed-up with different latencies of a fully-pipelined Rcreate logic. The Rcreate logic is placed out of the critical path of execution and only schedules instructions when it is necessary, so its latency doesn't affect the IPC of the ReLaSch processor, even with 100 cycles of latency there is little average degradation. Note that this latency includes the compacting logic and the writing latency of the Rcache.

9.2.6 Size of the `rcreate_input` buffer

Figure 9.10 shows the impact of the size of the `rcreate_input` buffer in the IPC speed-up. A 128-entry buffer is needed in order to allow the scheduler to process all the committed instructions. Many benchmarks don't have a performance degradation when smaller buffers are used, but the IPC drops drastically in some cases (*eon-rush*, *eon-cook*, *galgel*, *lucas*, *mesa*, *equake* and all the *perl* benchmarks). In general, the performance of most benchmarks degrades when a buffer of 16 entries is used.

In the current default configuration, the Rcreate logic accepts up to four instructions per cycle while up to 11 instructions can commit per cycle. Since the Ifront-end processes up to four instructions per cycle this completion rate cannot be sustained, but it still can overflow a 64-entry `rcreate_input` buffer, since the ROB holds up to 80 instruction that can commit in a burst.

These overflows reduce the IPC achieved in two ways: a) the rgroup being scheduled when the buffer overflows has less instructions than the maximum size, so the scheduler finds less ILP in it; b) there are committed instructions that are not processed by the Rcreate logic, so they are executed more often in the Icache mode.

The FP benchmark *facerec* shows an incoherent behavior, improving its IPC with a 32-instruction buffer over the IPC achieved with larger buffers. The IPC is worse with a 16-instruction buffer. Though the overflows of the `rcreate_input` buffer produced with a small buffer usually result in worse IPC, the rgroups created by the ReLaSch logic are different, and it is possible that they yield better IPC by better capturing the behavior of the branches or the memory instructions.

9.2.7 Width of the Rcreate logic

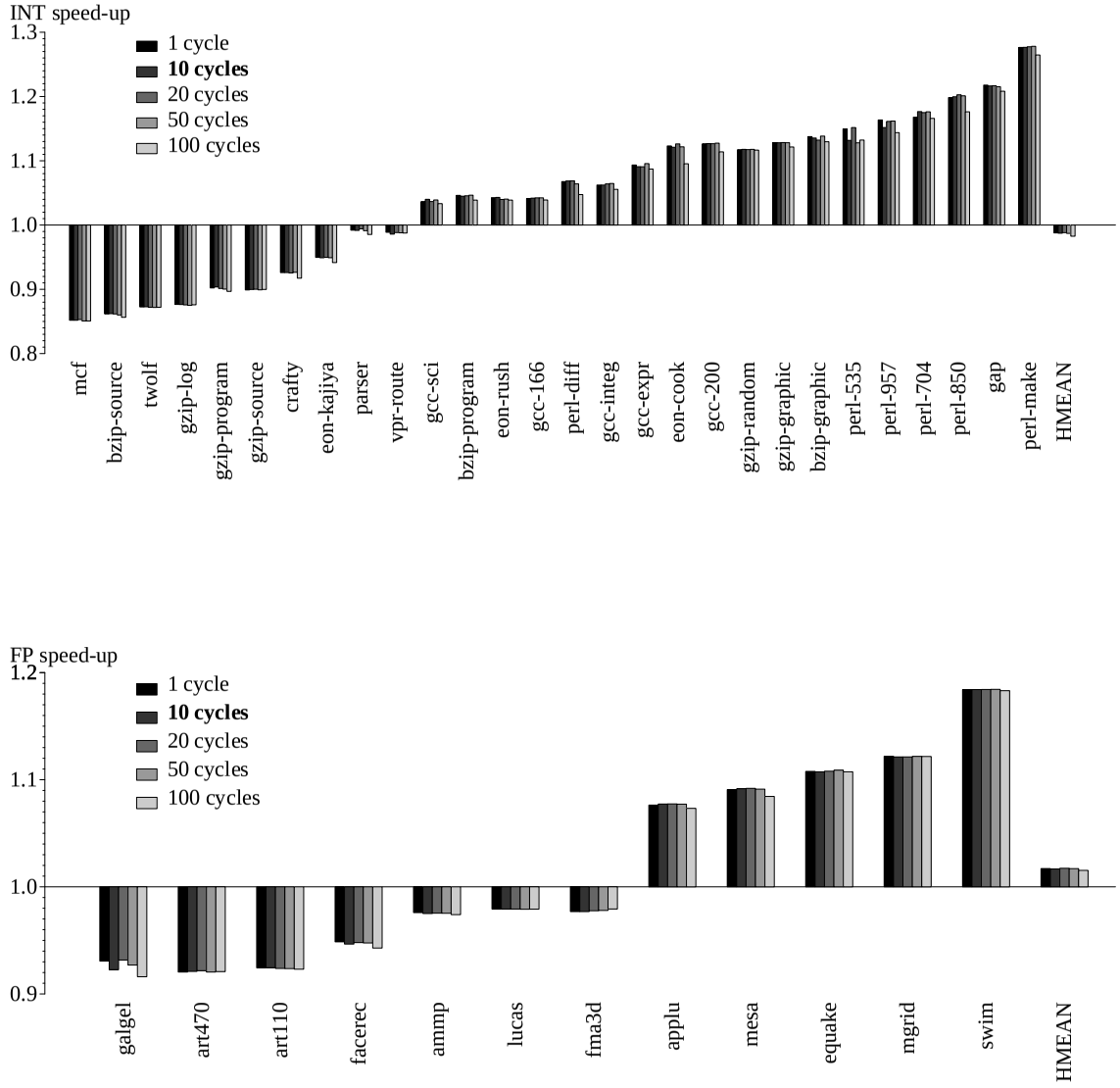
In the default configuration used in the experiments of this chapter, we have assumed a 4-instruction wide superscalar Rcreate logic. The reason to choose it was to enable the Rcreate logic to keep up the pace of the execution pipeline, which is limited by the 4-instruction wide front-end.

However, it would be challenging to implement the complex Rcreate logic in a superscalar fashion. We should add more ports to all the structures used and take into account the dependencies between instructions being scheduled at the same time. The complexity would lead to a very power-hungry logic.

Nevertheless, the main hypothesis used to design this processor is that we don't need to use the scheduling logic all the time and that it can be out of the critical path. Therefore, provided that the `rcreate_input` buffer has enough size in order to avoid overflows, a simple 1-instruction wide Rcreate logic should be able to yield almost the same performance as the superscalar scheduling logic, as long as the `rcreate_input` buffer is large enough to avoid the overflows.

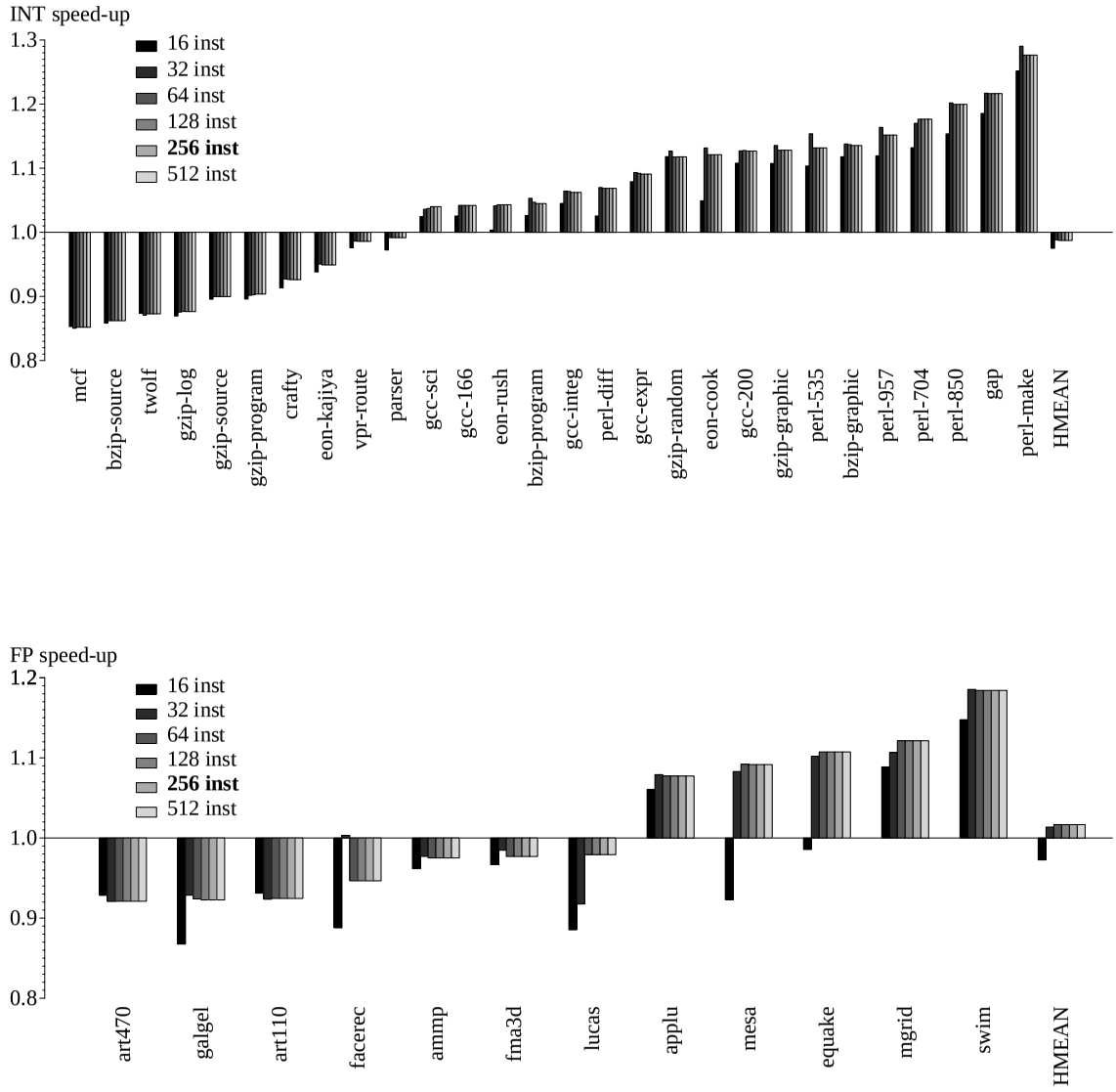
Figure 9.11 shows the speed-up with different widths of the Rcreate logic using a 256-entry `rcreate_input` buffer. It shows no significant variation. However, figure 9.12 shows the speed-up with different widths of the Rcreate logic but using a 32-entry `rcreate_input` buffer. In this case, the `rcreate_input` buffer suffers from frequent overflows with the smaller widths, which greatly reduces the IPC of some benchmarks.

In the new default configuration, we will assume that the Rcreate logic accepts one instruction per cycle and that it has a `rcreate_input` buffer that is large enough (512 entries).



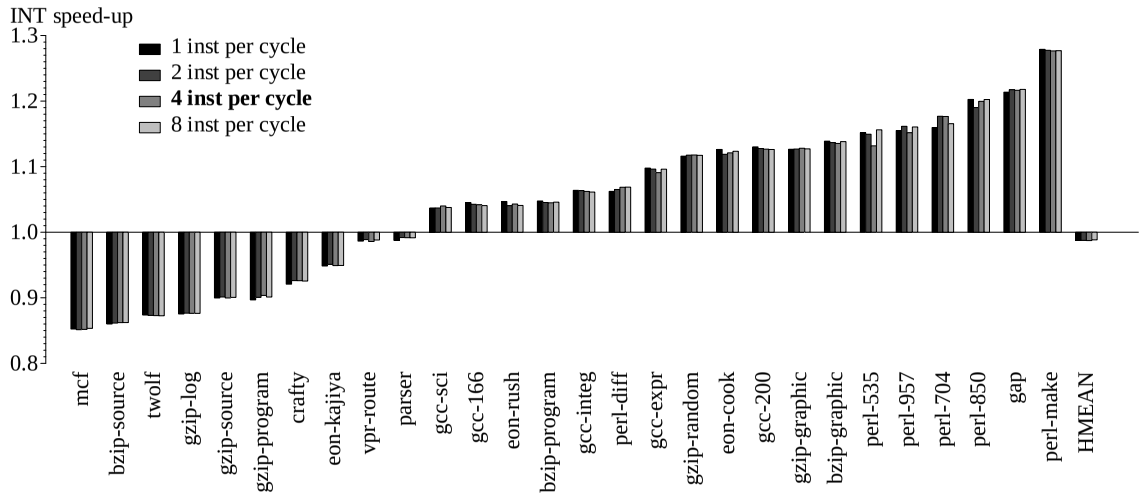
cycles	1	**10**	20	30	40	50	60	70	80	90	100
HMEAN INT	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.98
HMEAN FP	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02
HMEAN ALL	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99

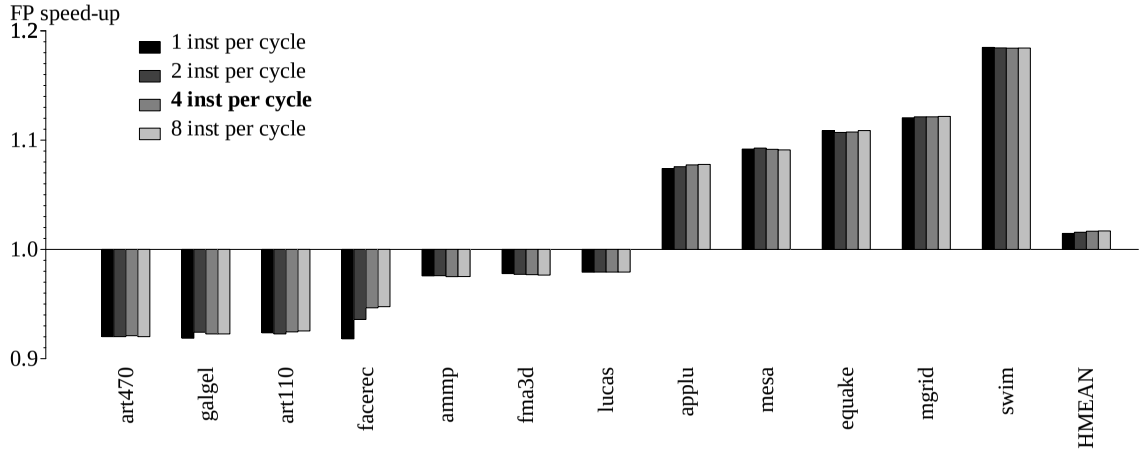
Figure 9.9: Speed-up with different latencies of a fully-pipelined Rcreate logic.



instructions	16	32	64	128	256	**512**
HMEAN INT	0.98	0.99	0.99	0.99	0.99	0.99
HMEAN FP	0.97	1.01	1.02	1.02	1.02	1.02
HMEAN ALL	0.97	1.00	1.00	1.00	1.00	1.00

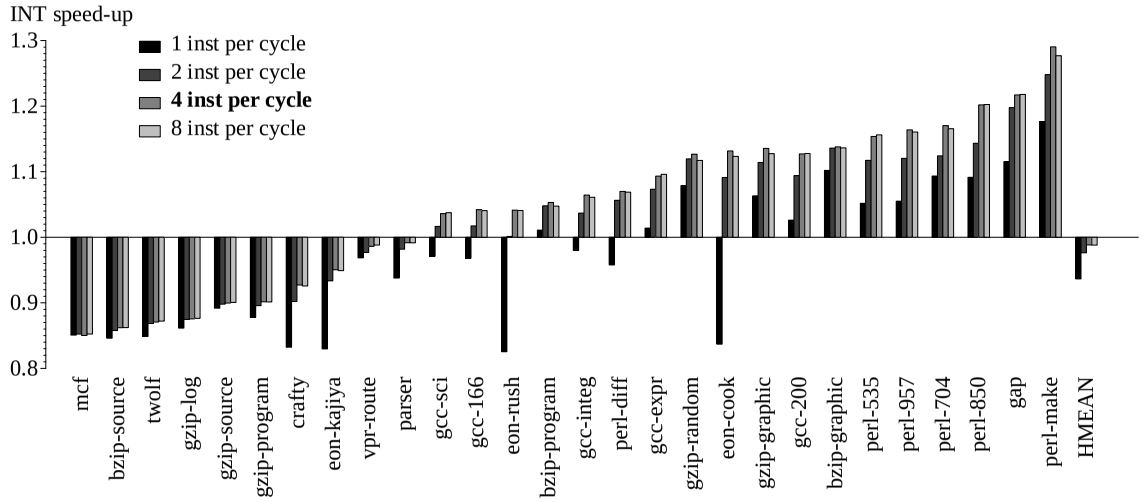
Figure 9.10: Speed-up with different sizes of the `rcreate_input` buffer.

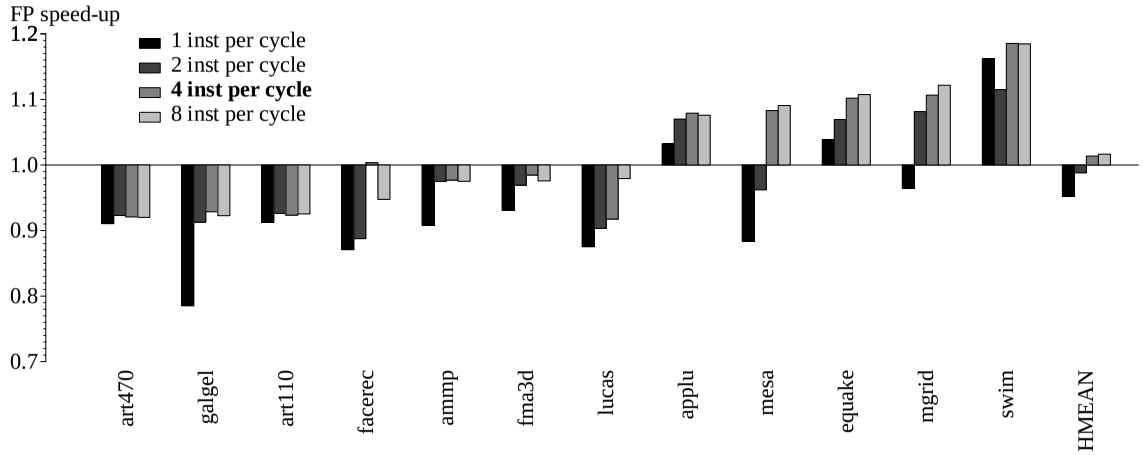




instructions per cycle	1	2	**4**	8
HMEAN INT	0.99	0.99	0.99	0.99
HMEAN FP	1.01	1.02	1.02	1.02
HMEAN ALL	1.00	1.00	1.00	1.00

Figure 9.11: Speed-up with different widths of the Rcreate logic using a 256-entry `rcreate_input` buffer.





instructions per cycle	1	2	**4**	8
HMEAN INT	0.94	0.98	0.99	0.99
HMEAN FP	0.95	0.99	1.01	1.02
HMEAN ALL	0.94	0.98	1.00	1.00

Figure 9.12: Speed-up with different widths of the Rcreate logic using a 32-entry `rcreate_input` buffer.

9.2.8 Rcreate-mode change policy

Figure 9.13 shows the speed-up when the Rcreate logic uses several different policies to change from the scheduling mode to the Idle mode. (that is, when instructions are executed from the Rcache instead of the Icache).

If we just close the current rgroup after scheduling the last instruction executed in the Icache mode ("close rgroup immediately" in the figure) the performance degrades severely. The reason is that, in this case, the Rcreate logic schedules many small rgroups and the processor sees small benefit from executing the rgroups.

In average, the best results are achieved when the Rcreate logic changes to the Idle mode once the current rgroup is closed and the instruction in the head of the `rcreate_input` buffer was executed in the Rcache mode (0 rgroups case in the figure). Scheduling one or two additional rgroups performance usually degrades, though there are a few exceptions (i.e. *perl-make*, *perl-850*, *galgel*, the *art* benchmarks, *facerec*, *fma3d* and *swim*), although the difference is small. If we go one step further and the Rcreate logic never changes to the Idle mode (it continues scheduling rgroups regardless of the execution mode), the results are worse in almost all cases, since it puts more pressure on the Rcache and it is more likely that we replace useful rgroups. Furthermore, it is more power hungry than changing to the Idle mode.

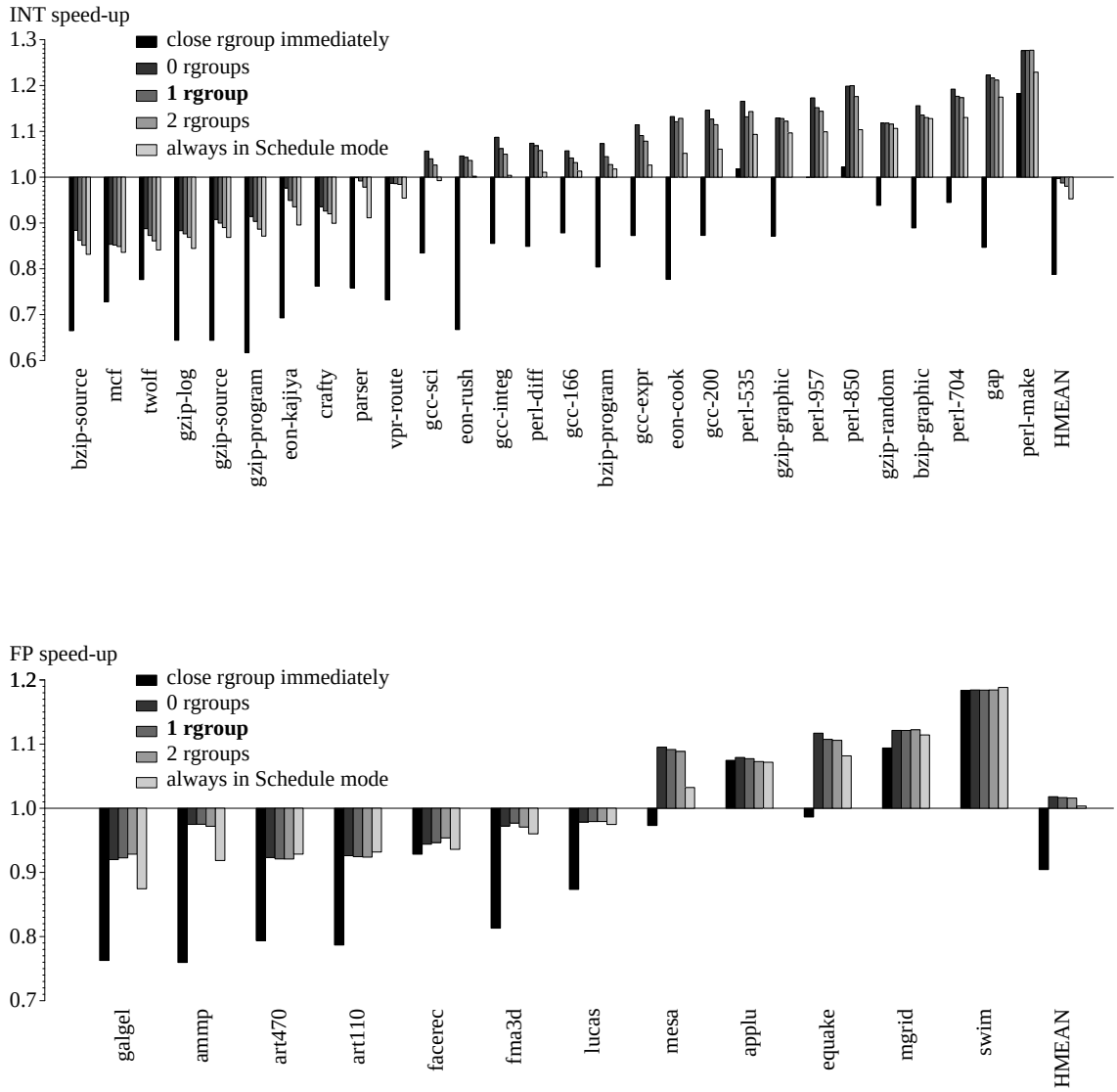
In the new default, the Rcreate logic switches to the Idle mode when an rgroup is closed and the next instruction was executed in the Rcache mode.

9.3 The Rcache

9.3.1 Rcache size

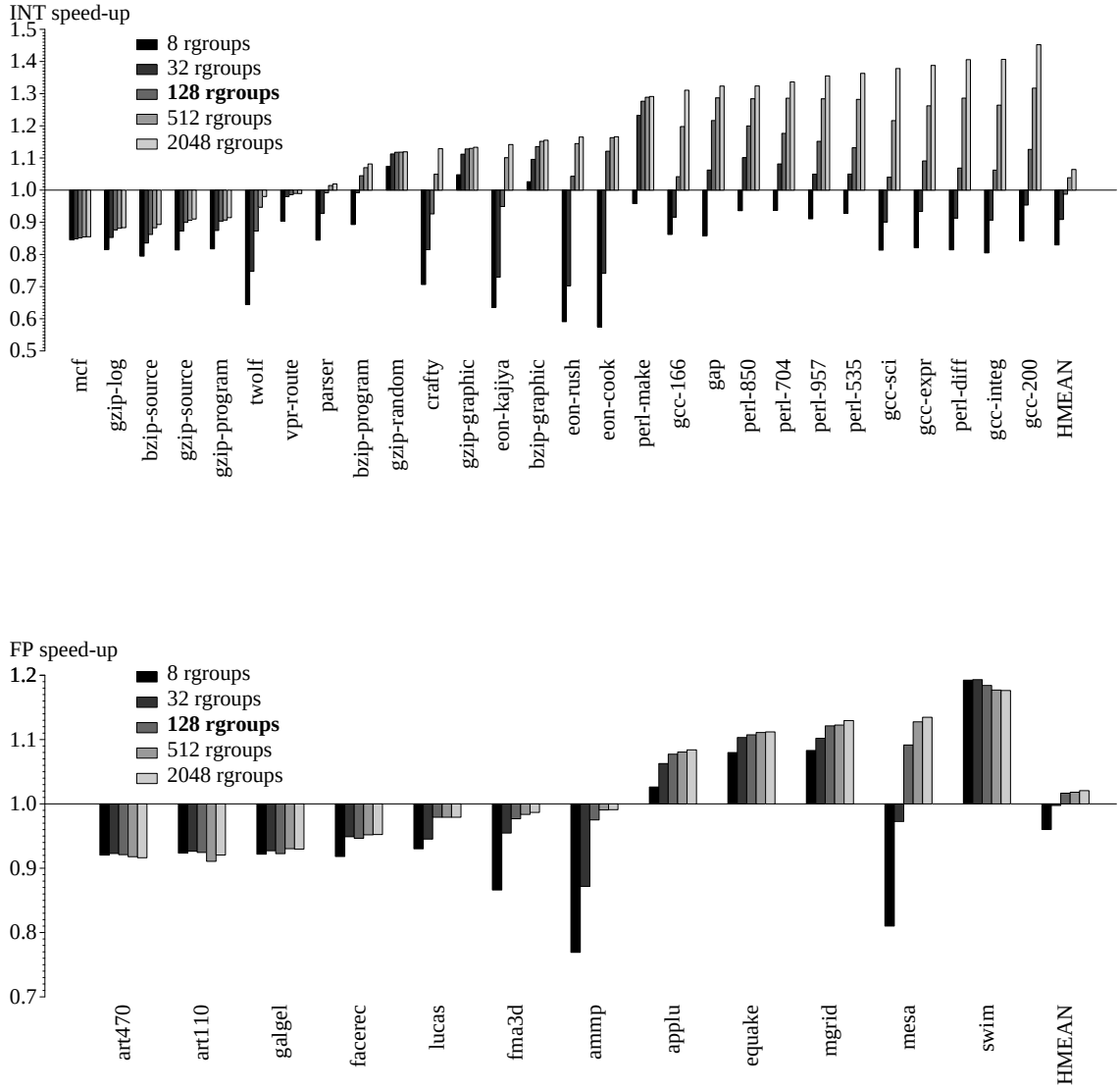
Figure 9.14 shows the impact of the size of the Rcache in the IPC speed-up. The number of cache lines is changed, thus changing the total number of rgroups that the Rcache can store. The caches in all the experiments are 4-way associative. As it could be expected, generally a larger Rcache improves performance, though in many cases only until a given point, when all the most used rgroups fit in the Rcache. The FP benchmarks *art110*, *art470* and *galgel* have enough with an Rcache that can store only four different rgroups. Most INT benchmarks always benefit from having larger Rcaches in the explored range. In particular, all the *gcc* and *perl* benchmarks (except *perl-make*) as well as *twolf*, *eon-kajiya* and *crafty* improve their performance with the largest Rcache explored (2,048 rgroups). The reason is that these benchmarks have a more complex branch behavior and execute through many different paths. The consequence is that they require a higher number of rgroups to cover all these paths. The rest of benchmarks stabilize their performance with an Rcache of at least 128 rgroups.

An exceptional case is the FP benchmark *swim*, that slightly reduces its IPC with larger Rcaches. This unexpected behavior is caused by worse memory latency prediction in the larger Rcaches, that results in more cycles lost waiting for the source registers or a busy resource. The *swim* benchmark has a high branch-prediction hit-rate (see table C.9), which keeps the Rcache counters high. Therefore, the rgroups are not usually marked as "bad rgroups" and re-scheduled. Thus, the prediction of the memory latency is not improved. With a smaller Rcache, the rgroups are re-scheduled as they are replaced and their instructions are executed in the Icache mode. A change in the re-scheduling policy could solve this problem if we are able to detect that the latency predictions are not very accurate. However, it would require additional detection logic. Furthermore, this problem affects the performance of just one benchmark, the performance degradation is small and in the worse case the benchmark still has good speed-up over the OoO processor.



policy	close rgroup immediately	0 rgroups	**1 rgroup**	2 rgroups	no Idle mode
HMEAN FP	0.90	1.02	1.02	1.02	1.00
HMEAN INT	0.79	1.00	0.99	0.98	0.95
HMEAN ALL	0.82	1.00	1.00	0.99	0.97

Figure 9.13: Speed-up with different policies to change the Rcreate mode upon a change to the Rcache mode.



rgroups	4	8	16	32	64	**128**	256	512	1024	2046
HMEAN INT	0.79	0.83	0.87	0.91	0.95	0.99	1.02	1.04	1.05	1.06
HMEAN FP	0.94	0.96	0.98	1.00	1.01	1.02	1.02	1.02	1.02	1.02
HMEAN ALL	0.84	0.87	0.90	0.94	0.97	1.00	1.02	1.03	1.04	1.05

Figure 9.14: Speed-up with different Rcache sizes.

9.3.2 Rcache associativity

Figure 9.15 shows the impact of the associativity of the Rcache in the performance achieved. Given a fixed size of the Rcache in number of rgroups (128), the set-associativity varies from 1-way (direct mapped) to 8-way. More associativity yields more average speed-up. Many benchmarks improve a lot when changing from direct mapped to 2-way. Some of them still improve significantly with more associativity though the average speed-up does not improve beyond 4-way set-associativity.

The FP benchmarks *art110*, *art470*, *galgel* and *lucas* are exceptional cases that show little or no improvement from associativity. The *bzip-graphic* benchmark performs better with a direct mapped Rcache. The reason is that it has some branches that are hard to predict and the rgroups that contain them are usually aborted at execution and re-scheduled. There can be several rgroups in a set with the same PC and different history bits. Thus, an rgroup r with a different PC may be replaced whereas in the Direct Mapped cache it would stay in the cache. The rgroup r may be placed in a different set in the Direct Mapped cache, increasing the number of instructions executed in the Rcache mode.

9.3.3 "Bad rgroup" counters

Figure 9.16 shows the speed-up with different sizes of the bad-rgroup counters in the Rcache, used to detect the rgroups that usually abort their execution. In general, INT benchmarks perform better with larger counters whereas some FP benchmarks (*art470*, *galgel*, *art110*, and *facerec*) benefit from smaller counters.

Almost all the benchmarks degrade their IPC when no counter is used (in this case, the rgroups are never marked as "bad rgroups"). The most relevant case is *gzip-program*, that nearly triplicates the number of branch mispredictions when the counters are removed (from 1.06M to 2.8M). On the other hand, the benchmarks *art470*, *art100*, *twolf*, *crafty*, *eon-kajiya* and *perl-diff* achieve the highest speed-up without counters in the Rcache, though the difference is small. The benchmarks *gzip-source*, *eon-rush*, *gcc-integ*, *eon-cook*, *gcc-200*, *fma3d* and *mesa* perform slightly better without counters than with some sizes of counters. Generally speaking, seems that it is beneficial to have between two and four bits per counter.

Figure 9.17 shows the speed-up when different policies to update the counters in the Rcache are used. Though the main reason to abort the execution of an rgroup is a mispredicted branch, the ReLaSch default configuration also decrements the counter on an aborted rgroup due to a load replay or on an rgroup that contains a load that has missed in the L2.

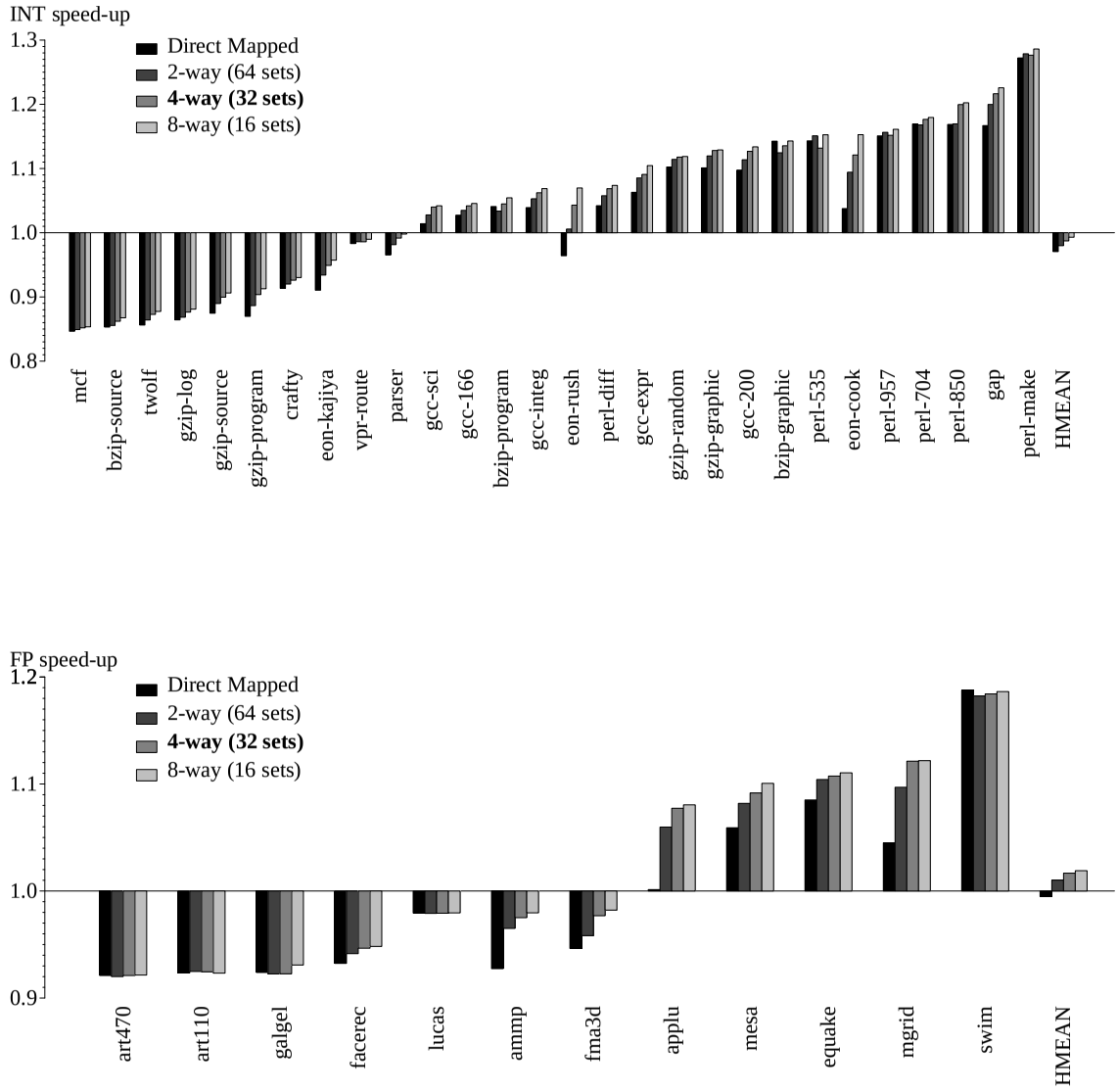
If the L2 miss is ignored, benchmarks *perl-704*, *facerec*, *lucas*, *equake* and *mgrid* decrease their IPC, while *perl-make*, *art470* and *art110* improve theirs. If the replay is ignored, *perl-850* has a decrease in IPC while *perl-535* improves its performance.

The overall effect of using the default updating policy is positive but small. Since it is easy to implement, the L2 misses and the replays are not ignored.

9.3.4 Rcache read latency

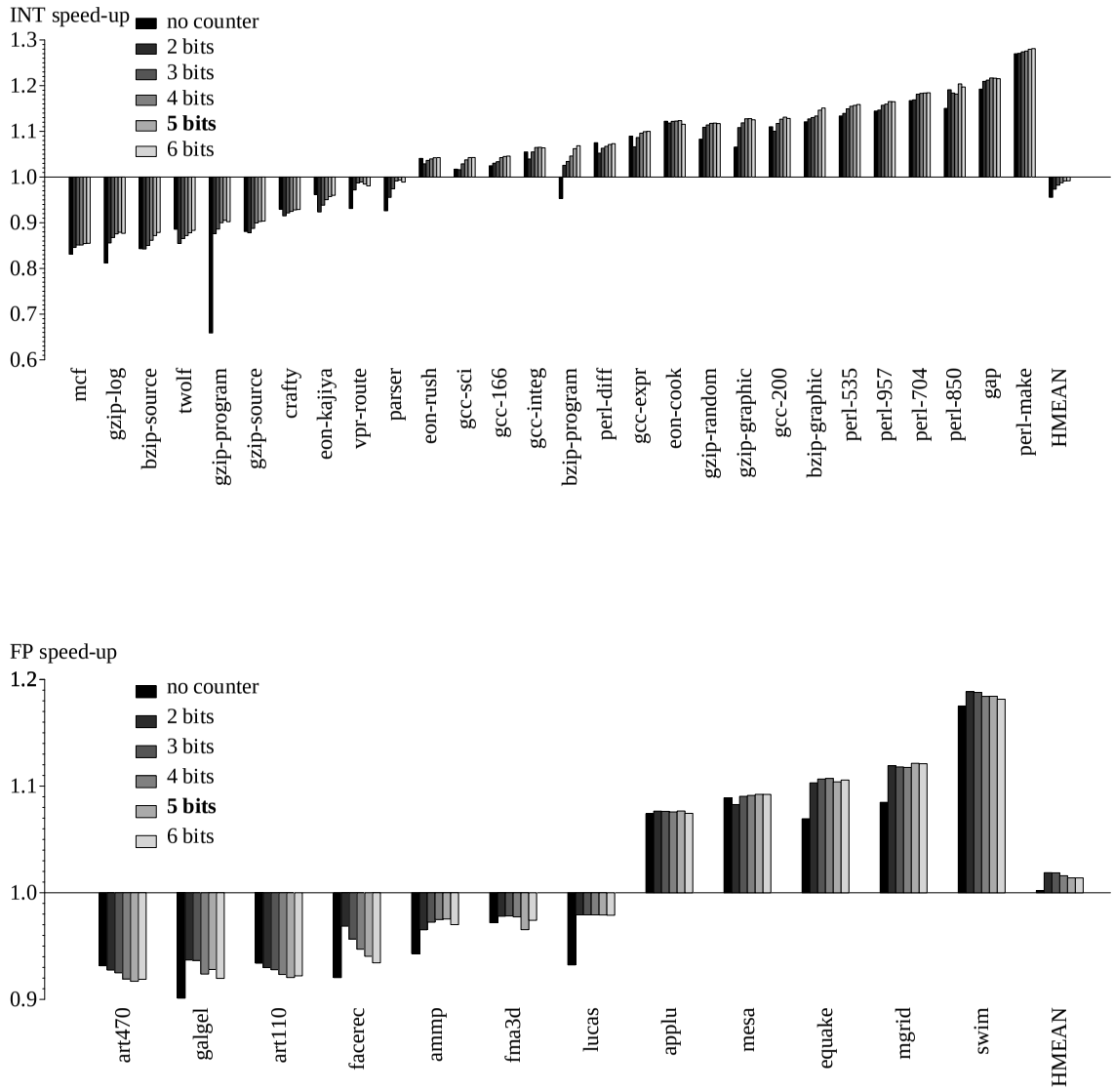
Figure 9.18 shows the speed-up with different latencies to read the first issue-group in the Rcache. It is only considered on a change to the Rcache mode, because when two rgroups are executed consecutively, the access latency of the second one can be overlapped with the last cycles of fetching of the first rgroup. This latency is also added after a pipeline flush when the processor is in the Rcache mode because in this case the latency cannot be hidden. The latency can be also seen as the impact of having additional stages in the Rfront-end logic.

This parameter has a small impact in most FP benchmarks (except *facerec*, *ammp* and *mesa*), but in all the INT benchmarks larger latencies significantly reduce the IPC achieved, so it is important to keep this latency as small as possible. The reason why INT benchmarks are more affected by this



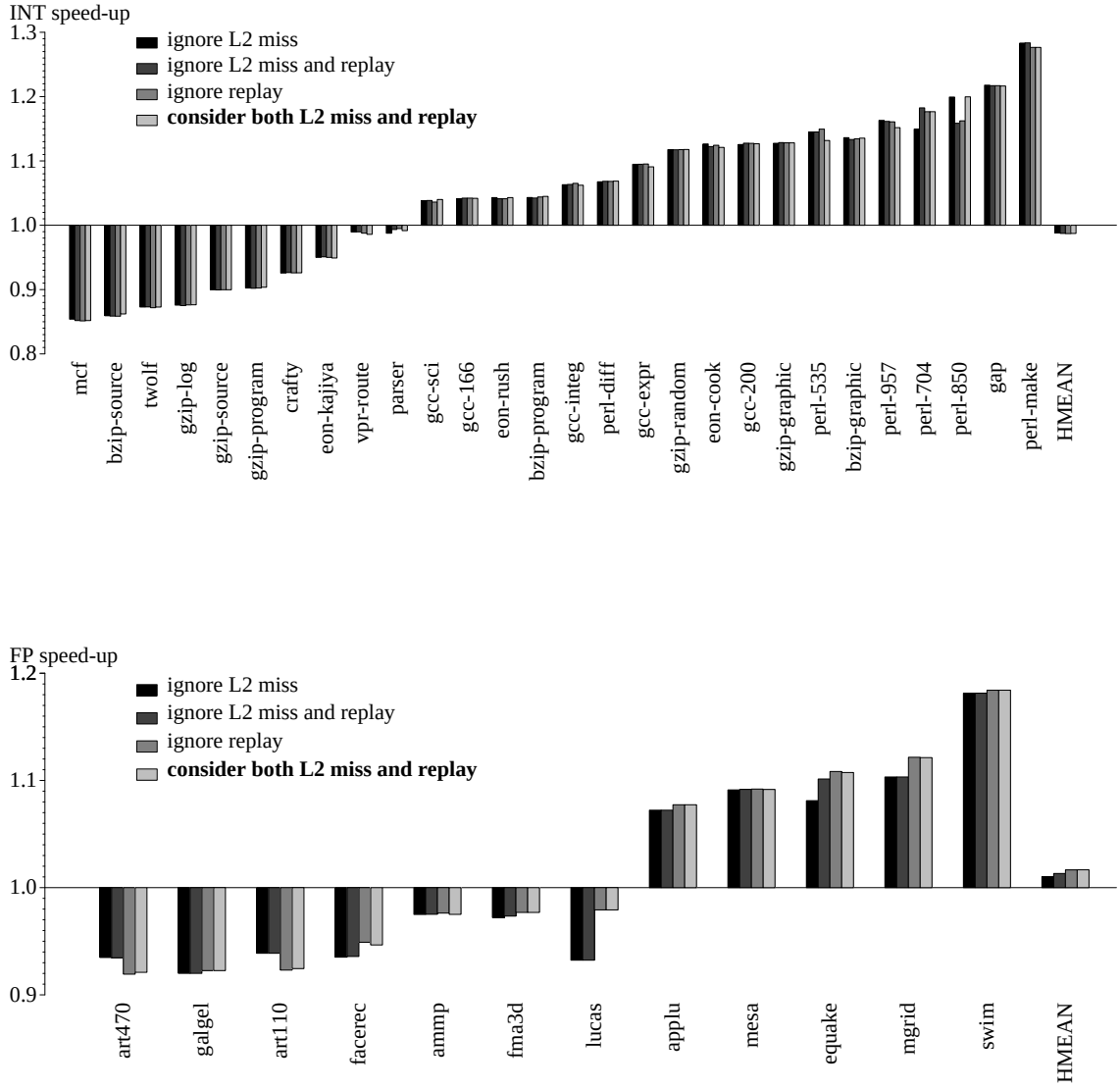
ways x sets	Direct Mapped	2x64	**4x32**	8x16
HMEAN INT	0.97	0.98	0.99	0.99
HMEAN FP	0.99	1.01	1.02	1.02
HMEAN ALL	0.98	0.99	1.00	1.00

Figure 9.15: Speed-up with different associativity of a 128-rgroups Rcache.



bits per counter	no counter	2	3	4	**5**	6
HMEAN INT	0.96	0.97	0.98	0.99	0.99	0.99
HMEAN FP	1.00	1.02	1.02	1.02	1.01	1.01
HMEAN ALL	0.97	0.99	0.99	1.00	1.00	1.00

Figure 9.16: Speed-up with different sizes and initialization values of the counters in the Rcache.



policy	ignore L2 miss	ignore L2 miss and replay	ignore replay	**consider both**
HMEAN INT	0.99	0.99	0.99	0.99
HMEAN FP	1.01	1.01	1.02	1.02
HMEAN ALL	1.00	1.00	1.00	1.00

Figure 9.17: Speed-up with different policies to update the counters in the Rcache.

latency is that, in general, they have worse branch misprediction rates. The pipeline is flushed more often and the latency is paid more often.

9.3.5 Rgroup-identifier: history bits

Figure 9.19 shows the speed-up achieved with different sizes of the history bits used to identify the rgroups in the Rcache.

This parameter has a small impact in the average speed-up, regardless of the number of history bits or whether history is used at all. The reason for this small impact is that history bits are used only as a hint instead of requiring them to match. In most cases just one variant of the history bits is enough to capture the behavior of the code. On the other hand, all rgroups that start with the same instruction are stored in the same set, since the PC of this instruction is used to index the Rcache. This is convenient to be able to fetch an rgroup with the same PC when history doesn't match. However, for benchmarks that create several rgroups starting from a given PC, it puts more pressure on the corresponding set. Actually, just four rgroups starting from that instruction with alternative execution paths can be present at the same time in the Rcache (assuming the default four-way set-associativity).

The FP benchmarks *facerec* and *fma3d* slightly benefit from using history (regardless of the number of bits), *galgel* benefits from not using history or using less bits, while *equake* improves its IPC when more bits are used (up to 8). Some INT benchmarks benefit from having between 2 and 6 history bits (*mcf*, *gap*, *perl-535*, the *bzip* and *gzip* benchmarks) but the impact is small. *perl-make* benefits from using more bits. Benchmarks *twolf* and *eon-kajiya* improve with a smaller history (or no history at all).

As a last attempt to improve the impact of the history bits, two alternative ways to use them are explored in the rest of this section: a) requiring an exact match to hit in the Rcache and b) hashing the history bits and the PC.

Figure 9.20 shows the speed-up with different sizes of the history bits if they are required to match to identify the rgroups in the Rcache, along with the default ReLaSch, that allows reading an rgroup with non-matching history bits, if the desired rgroup is not available. The results show very clearly that it is beneficial to execute the non-matching rgroups, as the default configuration does. It gets worse with more history bits, since there are more possible paths for each rgroup and it is less likely that the history bits match and the number of instructions executed in the Icache mode is increased.

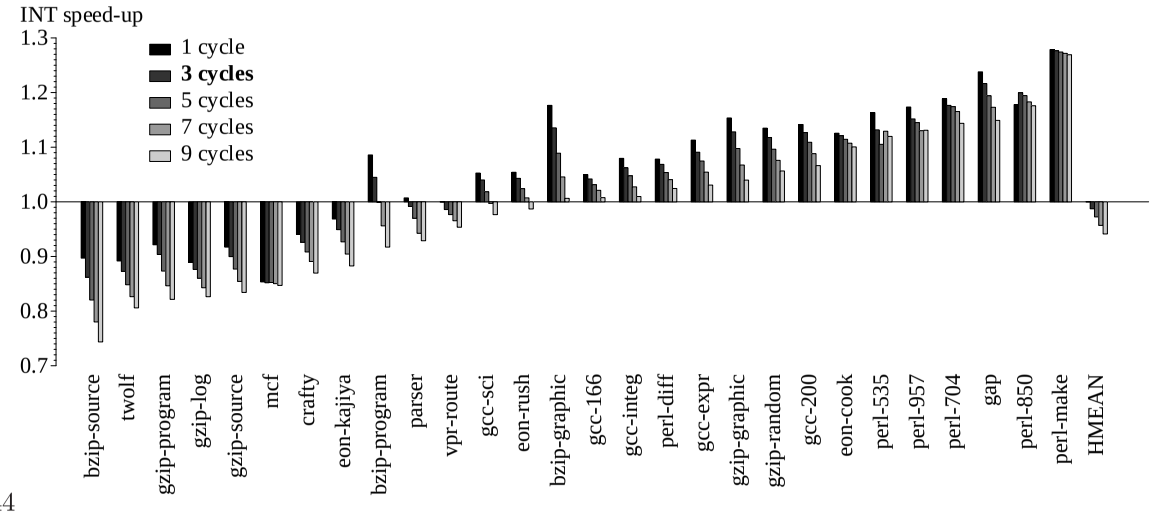
Figure 9.21 shows the speed-up with different size of the history bits used to identify the rgroups in the Rcache, when the history bits are hashed with PC to index the Rcache. The difference is small, though this yields in general lower IPCs than the default configuration, where only the lower bits of the PC are used to index the Rcache.

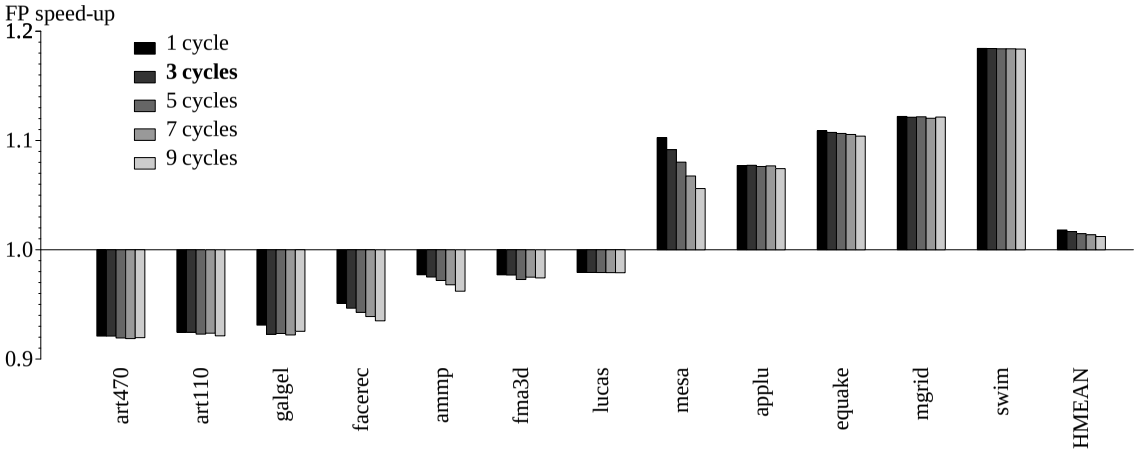
The benefit of using history is very small and since it makes more complex the access the Rcache and increases the number of bits that must be stored, it is not worthy to use them. Therefore, we will remove them in the new default configuration used in the final results.

9.4 The Issue logic

9.4.1 Issue-group boundaries

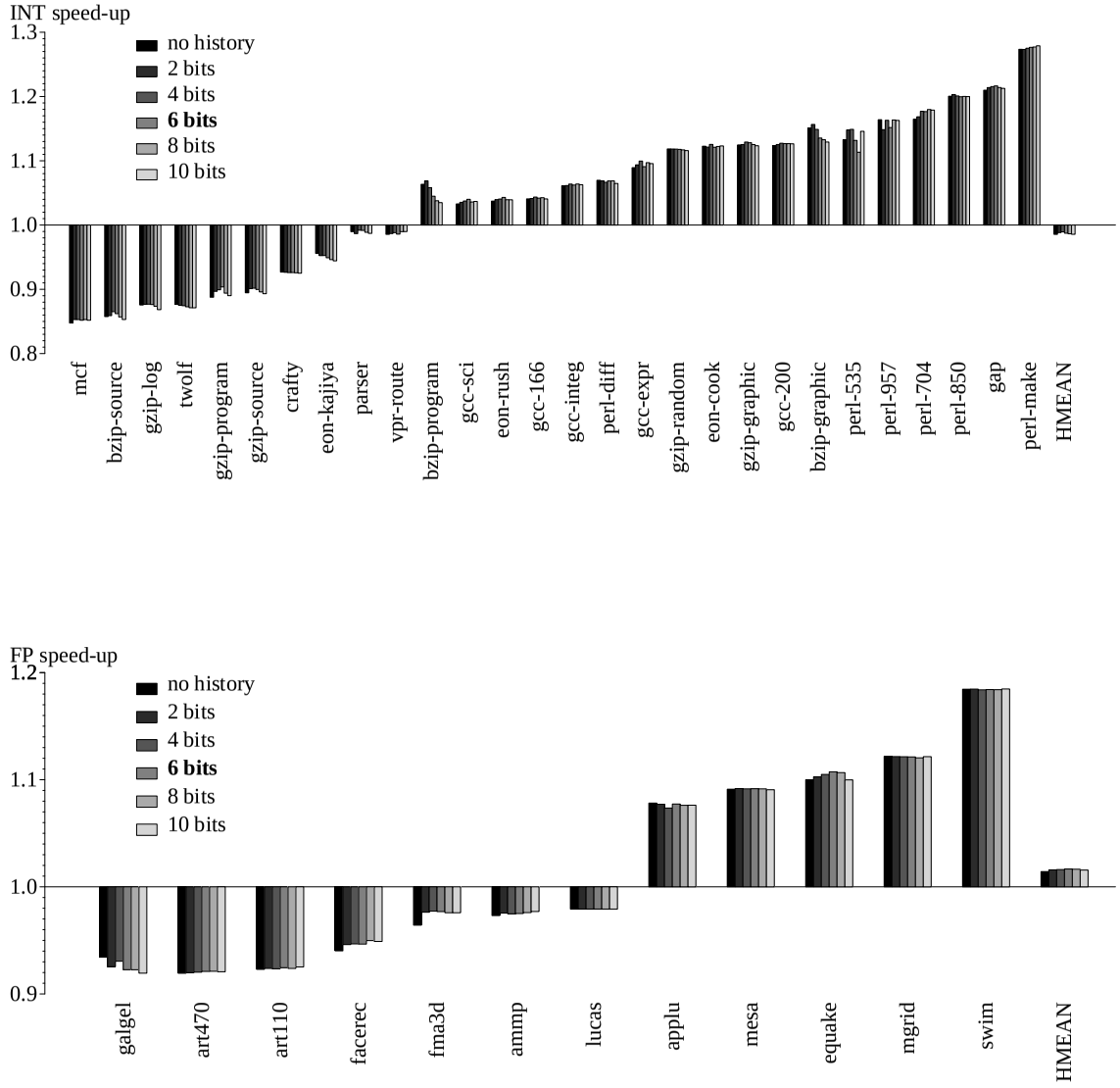
The beginning of a new issue-group is indicated with the information of each instruction in an rgroup. In the default configuration, the issue-logic prevents issuing instructions from different issue-groups in the same cycle. However, it is possible to ignore these bits and use the logic required to detect dependences in the Icache mode to find out register dependences also in the Rcache mode. Thus, if





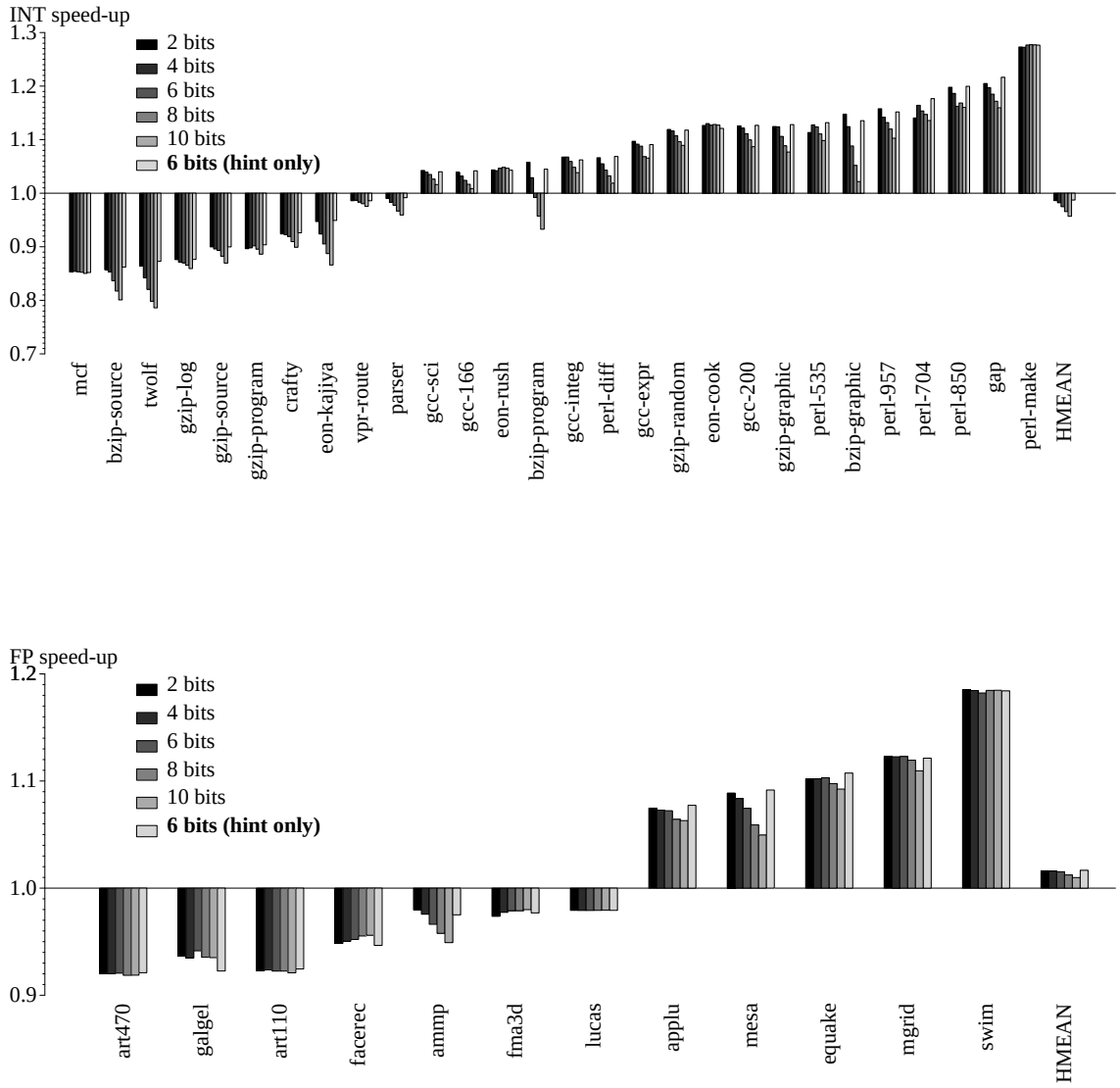
cycles of latency	10	9	8	7	6	5	4	**3**	2	1
HMEAN INT	0.93	0.94	0.95	0.96	0.96	0.97	0.98	0.99	1.00	1.00
HMEAN FP	1.01	1.01	1.01	1.01	1.01	1.01	1.02	1.02	1.02	1.02
HMEAN ALL	0.96	0.96	0.97	0.98	0.98	0.99	0.99	1.00	1.00	1.01

Figure 9.18: Speed-up with different read latencies to the first issue-group in the Rcache.



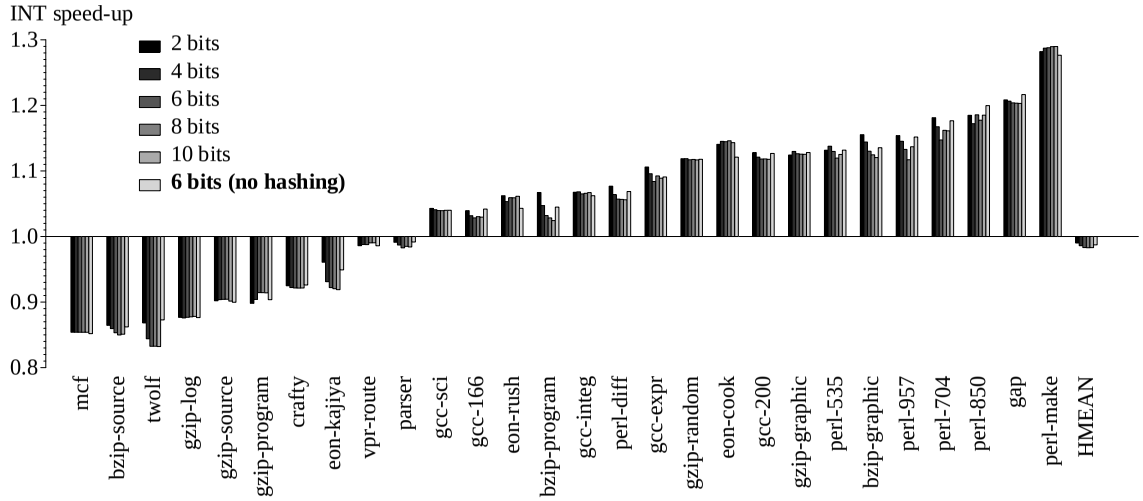
history bits	0	1	2	3	4	5	**6**	7	8	9	10
HMEAN INT	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99
HMEAN FP	1.01	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02	1.02
HMEAN ALL	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

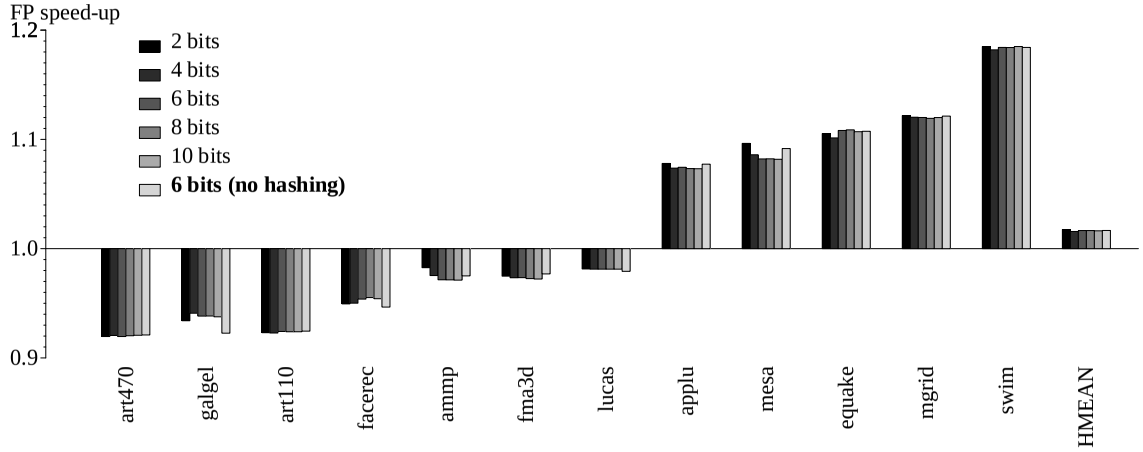
Figure 9.19: Speed-up with different sizes of the history bits used to identify the rgroups in the Rcache.



history bits	1	2	3	4	5	6	7	8	9	10	**6 (hint)**
HMEAN INT	0.99	0.99	0.99	0.98	0.98	0.97	0.97	0.97	0.96	0.96	0.99
HMEAN FP	1.02	1.02	1.02	1.02	1.02	1.02	1.01	1.01	1.01	1.01	1.02
HMEAN ALL	1.00	1.00	1.00	0.99	0.99	0.99	0.98	0.98	0.98	0.97	1.00

Figure 9.20: Speed-up with different sizes of the history bits if they are required to match to identify the rgroups in the Rcache. Default is 6 history bits used just as a hint (no matching requirement).





history bits	1	2	3	4	5	6	7	8	9	10	**6 no hash.**
HMEAN INT	0.99	0.99	0.99	0.99	0.98	0.98	0.98	0.98	0.98	0.98	0.99
HMEAN FP	1.02	1.02	1.02	1.02	1.01	1.02	1.02	1.02	1.02	1.02	1.02
HMEAN ALL	1.00	1.00	1.00	1.00	0.99	0.99	0.99	0.99	0.99	0.99	1.00

Figure 9.21: Speed-up hashing the history bits (of different sizes) used to identify the rgroups in the Rcache.

the instructions of an issue-group are ready before the scheduler expected, they would be executed sooner.

Figure 9.22 shows the speed-up with and without respecting the issue-group boundaries in the Issue stage, as defined by the `new-issue-group` bits. For most benchmarks it does not make any difference, but some INT benchmarks (*gap*, *perl-make*, *perl-diff* and the *eon* benchmarks are the most remarkable ones) have much more store-load replays when the issue-groups boundaries are ignored by the Issue stage, which decreases the IPC achieved. The scheduler places an aliased store-load pair in consecutive cycles, because a load is replayed if it is issued in the same cycle as the aliased store.

For instance, the *perl-make* benchmark increases the number of load replays from 2K to 325K (see table C.18 for details). Each replay has a penalty of 30 cycles, so the number of execution cycles is increased in more than 9 millions. The OoO processor has 3K replays in this benchmark.

9.4.2 Issue width

Figure 9.23 shows the speed-up with different issue widths of the ReLaSch and the OoO processors over the baseline OoO processor, that issues up to four integer and two floating point instructions per cycle. In general, both the OoO and ReLaSch processors have similar improvements when scaling from 4+2 to 8+4. When scaling to 16+8, the OoO still improves, with the exception of some benchmarks that do not have more instruction-level-parallelism that can be exploited. On the other hand, while all FP benchmarks improve their performance, many INT benchmarks perform significantly worse with the 16+8 issue logic in the ReLaSch processor. For instance, *gzip-program*, *gcc-sci*, *bzip-source*, *bzip-graphic*, *eon-kajiya* and *perl-diff*.

The reason for such unexpected results is the huge increase in load instructions that must be replayed in the wider issue configuration. The scheduler of the ReLaSch processor captures pairs of aliased loads and stores but not larger sets of aliased instructions, in which the specific pairs of aliased store-loads change in each iteration or function call. Capturing pairs is enough for most benchmarks in a narrow (four and two) issue-logic like in the default configuration, because it is less likely that a pair of these load and store instructions is reordered. But with a wider issue logic, it becomes more probable, causing the observed performance degradation. Compare tables C.16 and C.17 for more details.

In case a wider issue is the desired design point, the scheduler could be enhanced to cope with this problem. The OoO does not show such performance degradation because it implements Store Sets to deal with the aliased memory instructions, a technique that specifically addresses this problem. However, the complexity of its issue logic makes more challenging to implement a wider issue logic in a conventional out-of-order processor than in the ReLaSch processor, that has an in-order execution pipeline.

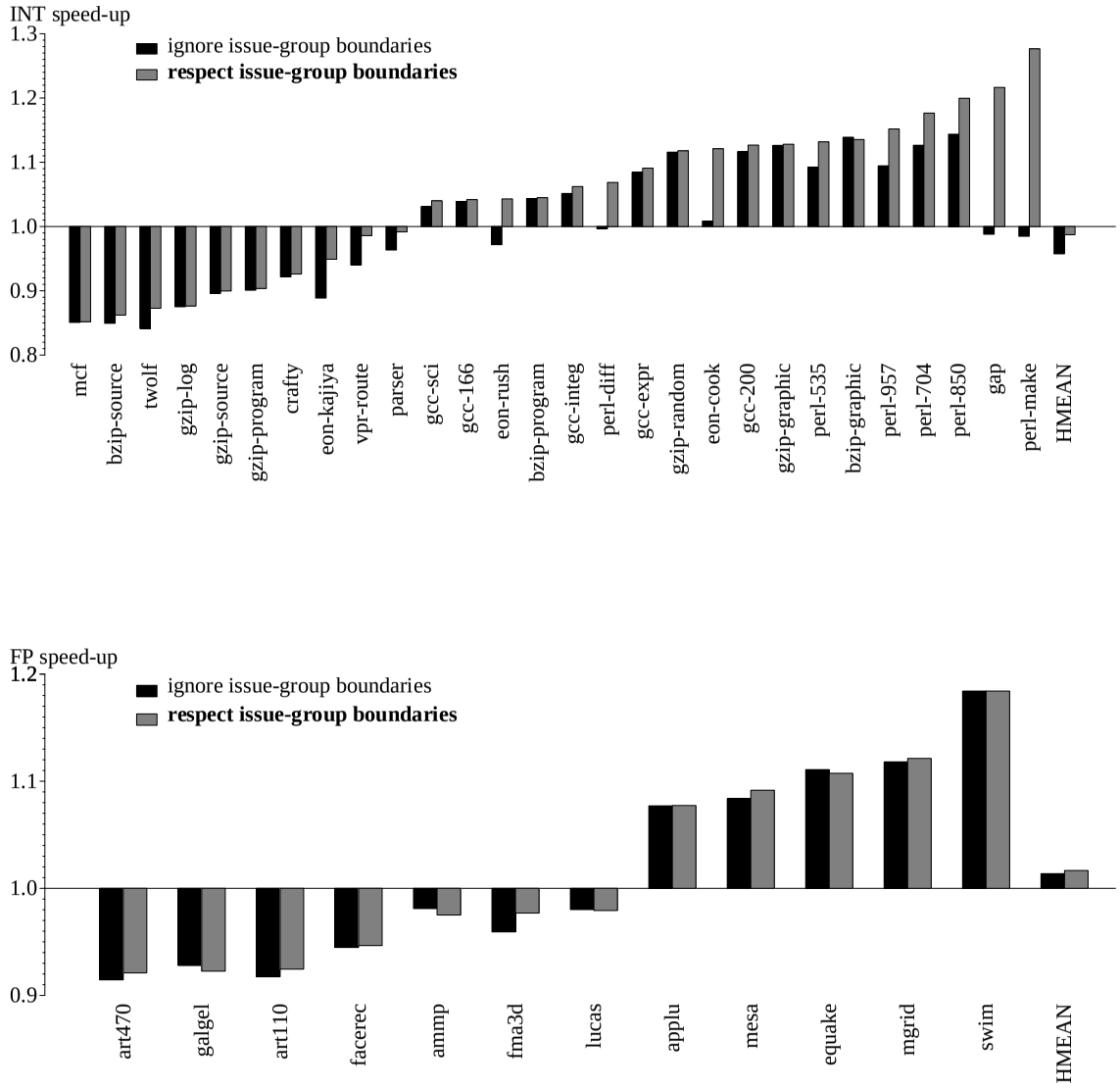
9.5 Other elements

9.5.1 The register file

Size of the register file

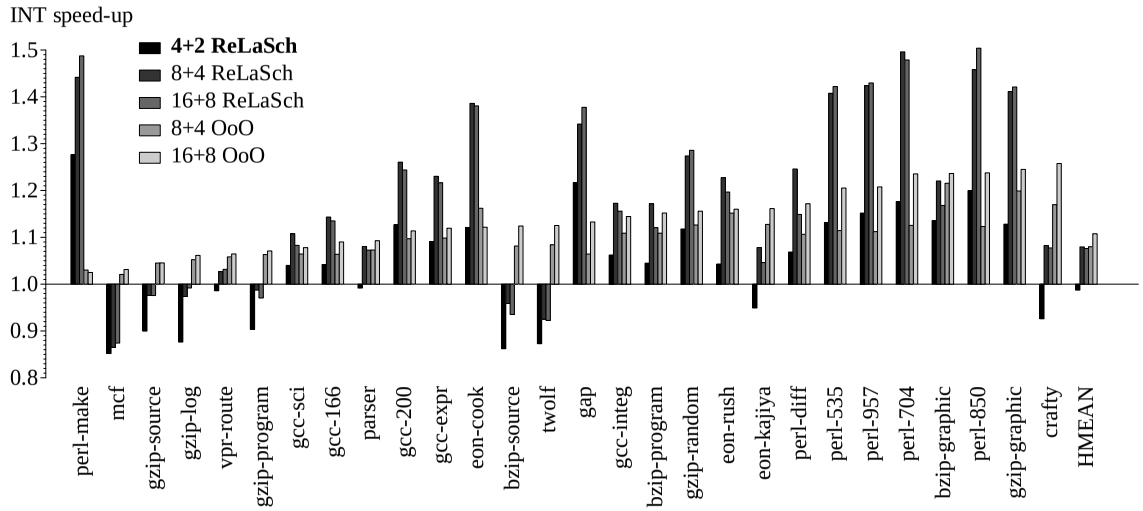
Figure 9.24 shows the speed-up achieved with sets of 4, 8 and 16 physical registers for each logical register in the register file. The default configuration for ReLaSch is 8 physical registers per logical register. At first sight, using 4 physical registers severely hurts the IPC achieved. The benchmarks that use just a small set of destination registers for most instructions are specially penalized, for instance the *gzip* benchmarks.

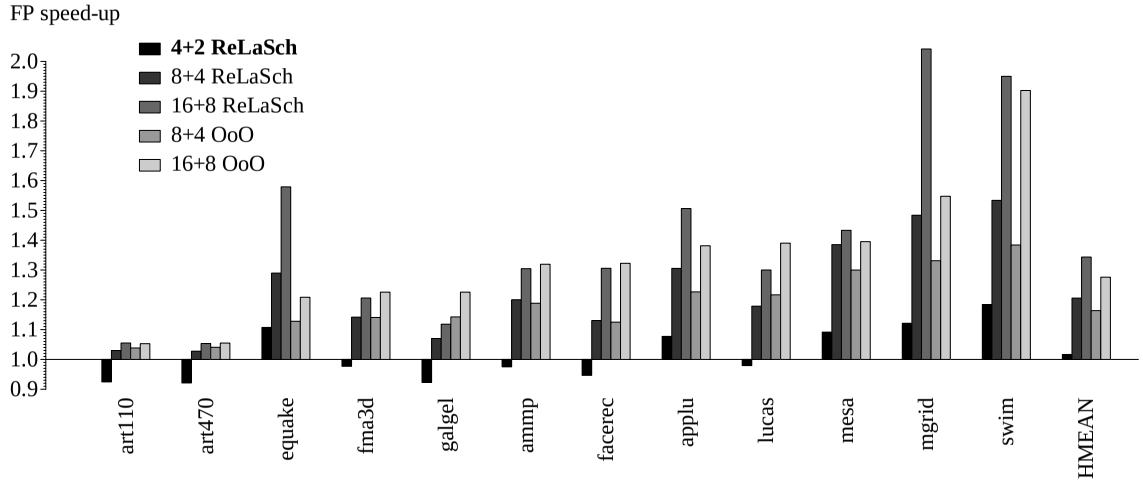
Some benchmarks (most noticeable *gzip-log*, *gzip-source*, *gzip-program*, *art470*, *art110* and *facerec*) benefit from increasing the number of registers to 16. However, using 16 physical registers does not



	ignore issue-group boundaries	**respect boundaries**
HMEAN INT	0.96	0.99
HMEAN FP	1.01	1.02
HMEAN ALL	0.98	1.00

Figure 9.22: Speed-up with/without respecting the issue-group boundaries in the Issue stage.





issue-width	**4+2 ReLaSch**	8+4 ReLaSch	16+8 ReLaSch	8+4 OoO	16+8 OoO
HMEAN INT	0.99	1.08	1.08	1.08	1.11
HMEAN FP	1.02	1.21	1.34	1.16	1.28
HMEAN ALL	1.00	1.12	1.15	1.11	1.16

Figure 9.23: Speed-up with different issue widths.

yield a significant overall benefit over the 8-physical registers default for the INT benchmarks, while the FP benchmarks do show an improvement in the average speed-up. The higher parallelism generally available in the FP applications puts more pressure to the register file, so these applications benefit from having more available registers.

ReLaSch vs. conventional register file

The use of such different register files in the OoO and the ReLaSch processors may have an impact in performance. To quantify it, we have simulated a version of the OoO processor that uses the register file of the ReLaSch processor. We show its speed-up over the baseline OoO processor. Figure 9.25 shows the speed-up of the default configuration of the ReLaSch processor and the OoO processor using the ReLaSch register file over the baseline OoO processor with a conventional register file.

The average IPC of the OoO processor with the ReLaSch register file is the same achieved with the conventional register file. Therefore, the ReLaSch register file has no relevant overall effect. However, the average hides that there are many benchmarks that show significant differences in speed-up. A closer look reveals several interesting results.

On one hand, there are some benchmarks that yield in this configuration between 90% and 95% of the IPC achieved by the baseline OoO. With the exception of *lucas*, all these benchmarks have a similar speed-up in ReLaSch. They are characterized by using a small set of logical destination registers, so they often find no free destination physical register and stall. For instance, in benchmarks *art470* and *art110*, four logical registers account for the destination register of nearly 90% of the integer dynamic instructions. Even though they are FP benchmarks, in both cases 54% of the committed writes are made to the integer register file. Similarly, in the *galgel* benchmark, eight logical registers accumulate 90% of the committed writes to the FP register file. The top three add up 44%. FP destination registers are a 63% of the total. Table C.19 shows the distribution of the writes among the whole logical register file. It shows how many logical registers are needed to accumulate 25%, 50%, 75% and 90% of all the writes. Results for the integer and floating point registers are shown separately.

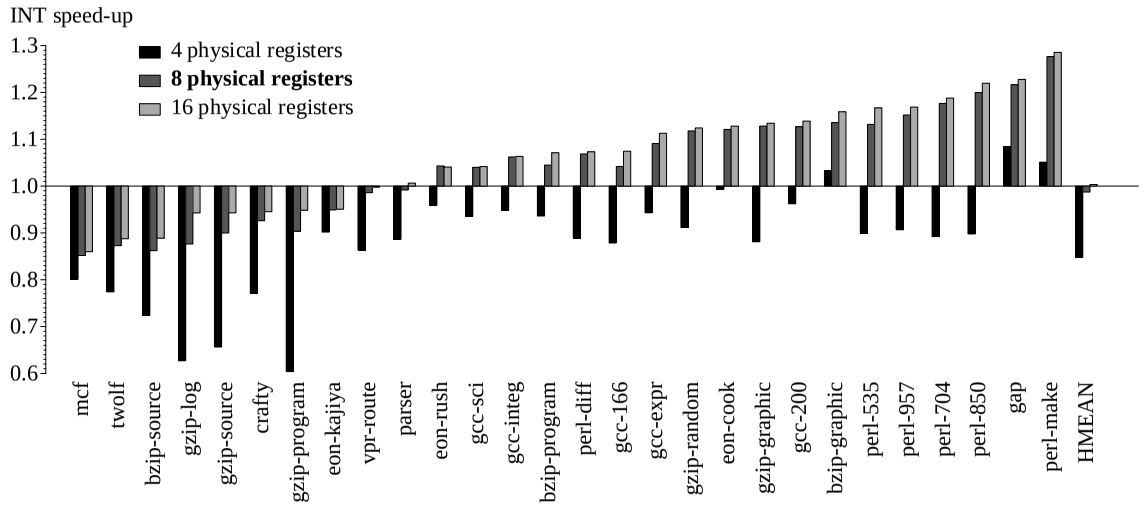
On the other hand, some benchmarks (*gzip-random*, *gzip-graphic*, *applu*, *mgrid* and *swim*) have a significant speed-up, up to 1.18 (*swim*). Since these benchmarks distribute their destination register among many logical registers, it is infrequent that they don't find an available destination physical register. In fact they benefit from the higher total number of physical registers of the ReLaSch register file. For instance, in benchmark *swim*, 23 logical registers are needed to accumulate the 90% of the FP destination registers and 11 register to add up to 50%. FP committed writes are a 73% of the total.

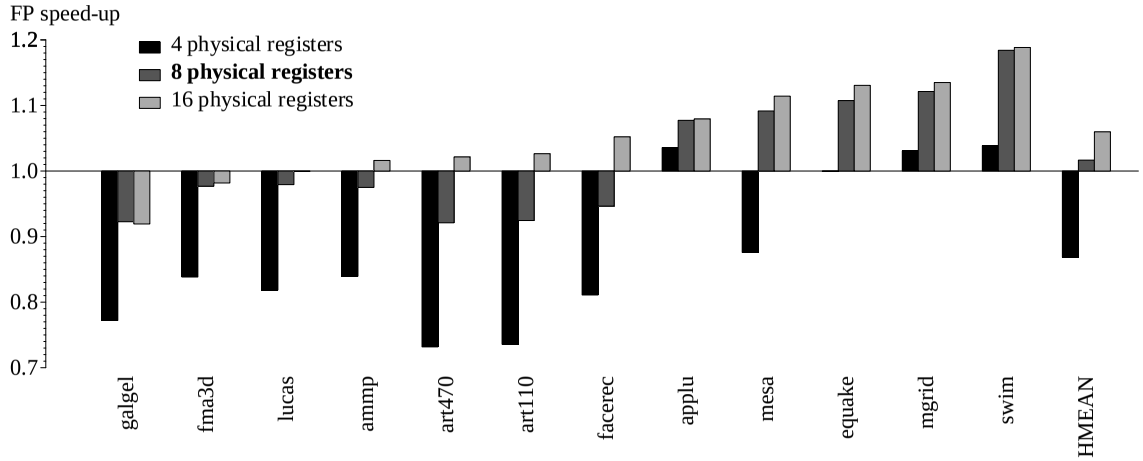
A processor that uses the ReLaSch register file benefits from distributing the use of registers across as many logical registers as possible. However, we have used directly the binaries generated by the compiler both for the ReLaSch and the OoO processor, oblivious to this property of the ReLaSch processor. The use of these binaries is advantageous for the OoO processor, since it uses a more flexible conventional register file. A possible future research direction is changing the register allocation policy of the compiler to improve the performance of the ReLaSch processor, distributing the allocated registers across as many logical registers as possible.

9.5.2 Store Sets and enhanced BTB

The OoO processor is based on the Alpha 21264 with two significant improvements: an enhanced BTB to predict multi-target indirect branches and the use of Store Sets instead of StWait bits. Neither of these two improvements is present in the ReLaSch processor but it uses the original Alpha 21264 BTB and StWait bits.

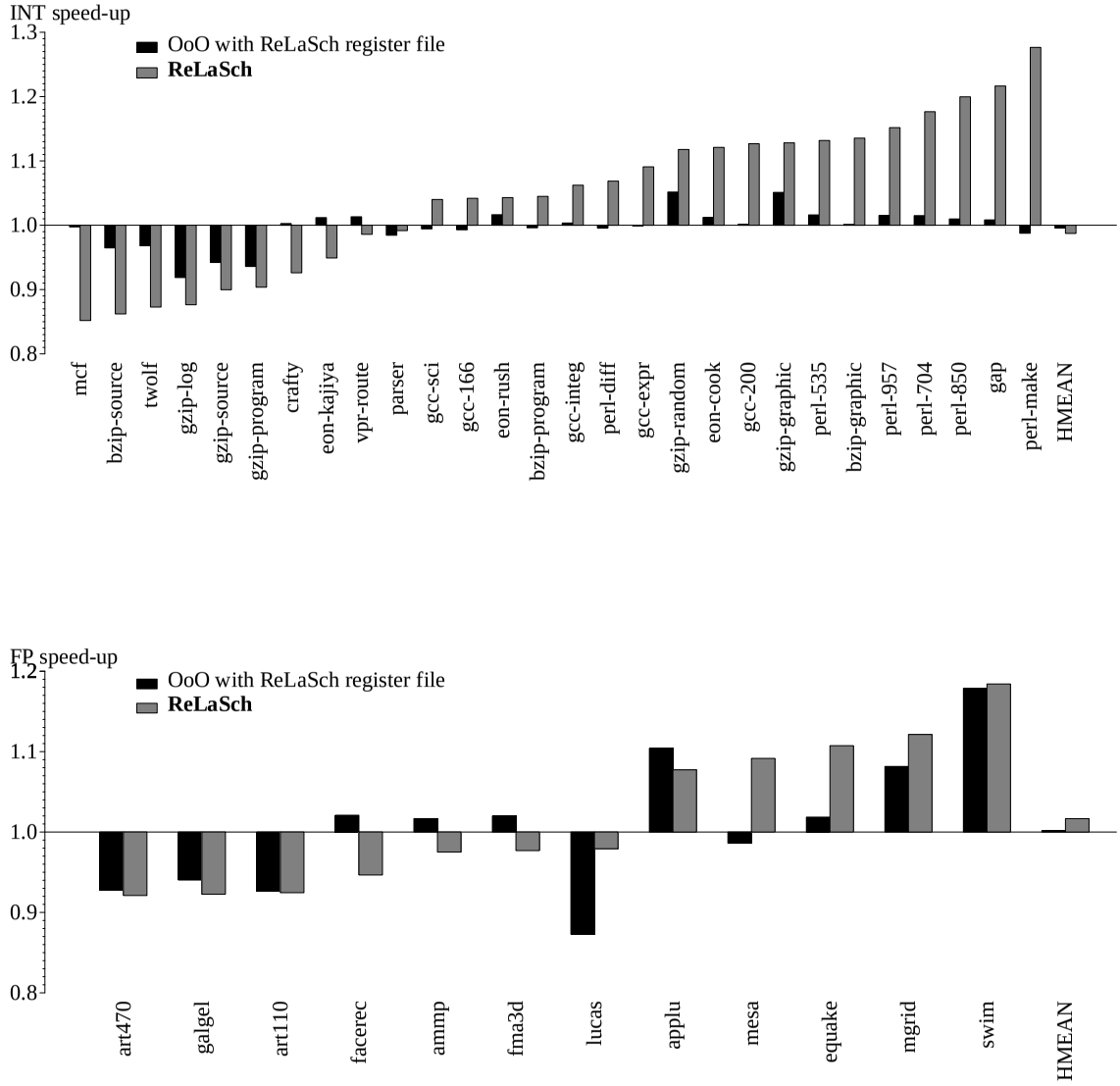
Figure 9.26 shows the speed-up over the baseline OoO processor of the default configuration of the ReLaSch processor and ReLaSch improved with Store Sets and the enhanced BTB of the OoO





physical registers	4	**8**	16
HMEAN INT	0.85	0.99	1.00
HMEAN FP	0.87	1.02	1.06
HMEAN ALL	0.85	1.00	1.02

Figure 9.24: Speed-up with different number of physical registers per logical register in the register file.



	OoO with the ReLaSch register file	**ReLaSch**
HMEAN INT	1.00	0.99
HMEAN FP	1.00	1.02
HMEAN ALL	1.00	1.00

Figure 9.25: Speed-up of ReLaSch and the OoO processor, both using the ReLaSch register file.

processor. Since both improvements are only used in the Icache mode, their impact in performance is negligible. Therefore, the ReLaSch processor doesn't need such complex structures.

To show the importance of these techniques in the OoO processor, figure 9.27 shows the speed-up over the baseline OoO of different configurations of the OoO processor: when the enhanced BTB is removed, when the Store Sets are removed and when both structures are removed. The conventional BTB of the Alpha 21264 or the StWait bits are used instead. Both techniques have more impact in the INT benchmarks. Particularly, the enhanced BTB has no effect at all in the performance of the FP benchmarks. The OoO processor drops to 0.93 of the IPC without the Store Sets in the INT benchmarks. The impact of the BTB is not so high in the average speed-up (0.97) since many benchmarks do not have a relevant number of multi-target indirect branches. However, those that are affected (mainly the *perl* benchmarks) show an important decrease in IPC when the original BTB of the Alpha 21264 is used. 12 benchmarks out of 28 achieve less than 90& of the IPC achieved with the baseline OoO that has both Store Sets and the enhanced BTB.

Table C.16 presents the number of replays of each benchmark in the OoO and the ReLaSch processor. Even though in general ReLaSch is able to capture the aliased load and store pairs and reduce the total number of replays, there are some notable exceptions. In particular, *bzip-source* has a huge increase in the number of replays. The reason is that there are sets of stores and loads that are aliased instead of just pairs of aliased instructions. Which particular load and store instructions are aliased changes every iteration. Our approach does not currently manage this case, unlike the Store Sets used by the OoO processor that explicitly targets this situation. Therefore, our schedules assume a single aliased pair and when any other alias is found at execution time, the execution of the schedule is aborted. We could extend our proposal to manage this case if it is frequent enough. The OoO processor does not have a very significant performance degradation without the Store Sets in *bzip-source* because the StWait bits allow much less aggressive speculation. Instead of being replayed, loads stall at the Issue stage until all older stores have committed.

9.5.3 Number of CMOV and INT+FP instructions

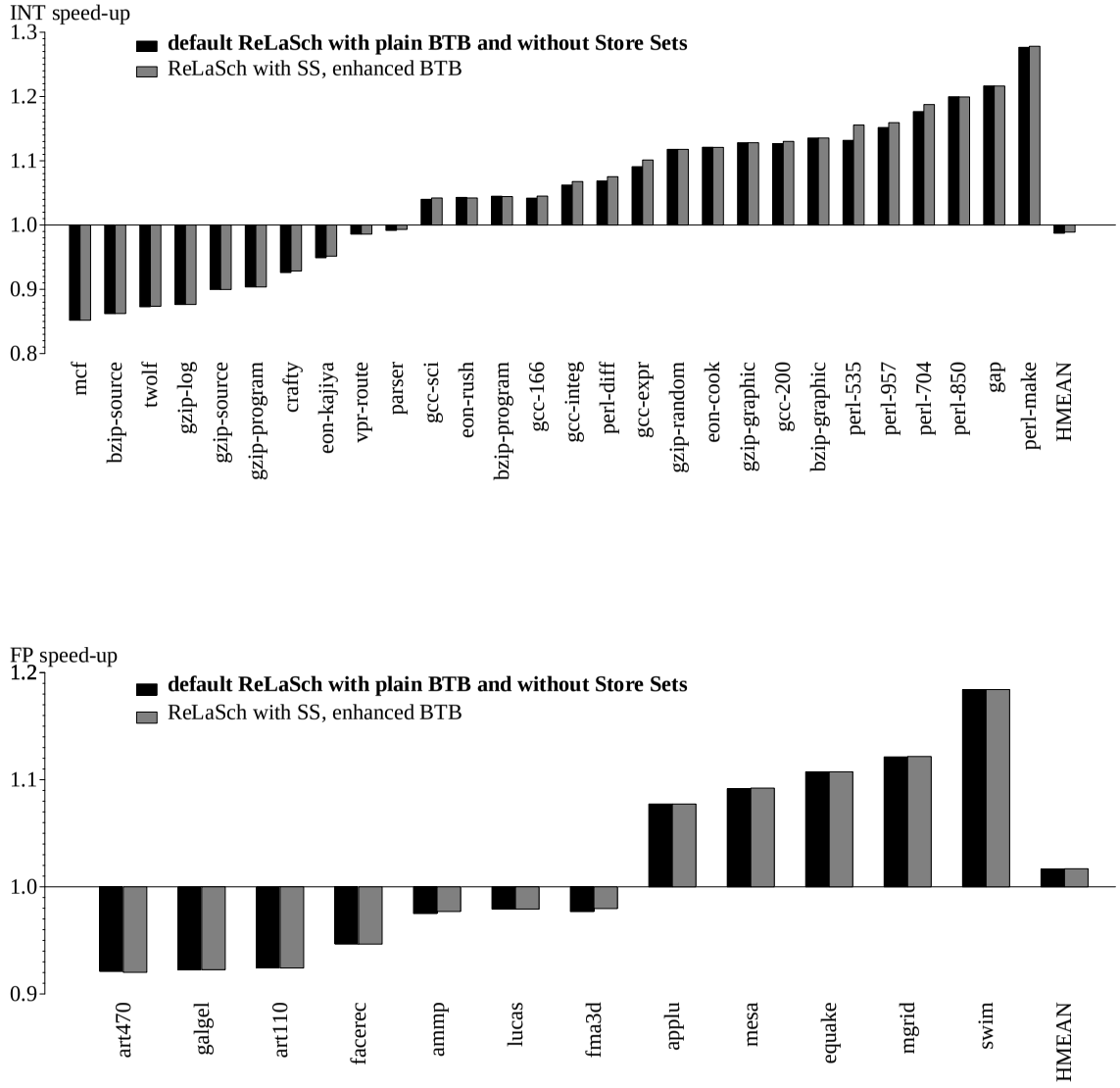
The CMOV instructions and the instructions that access both the INT and FP register files and are inserted in both issue queues require special handling, which is likely to lead to have a more complex issue logic. These two kinds of instructions are not very frequent, as can be seen in table C.3. Therefore, it may not be cost-effective to be able to process more than one of such instructions per cycle.

Figure 9.28 shows the speed-up with different maximum number of these instructions issued per cycle. As expected, the impact in performance of limiting the number of them that can process the issue logic each cycle is negligible.

9.5.4 Icache size

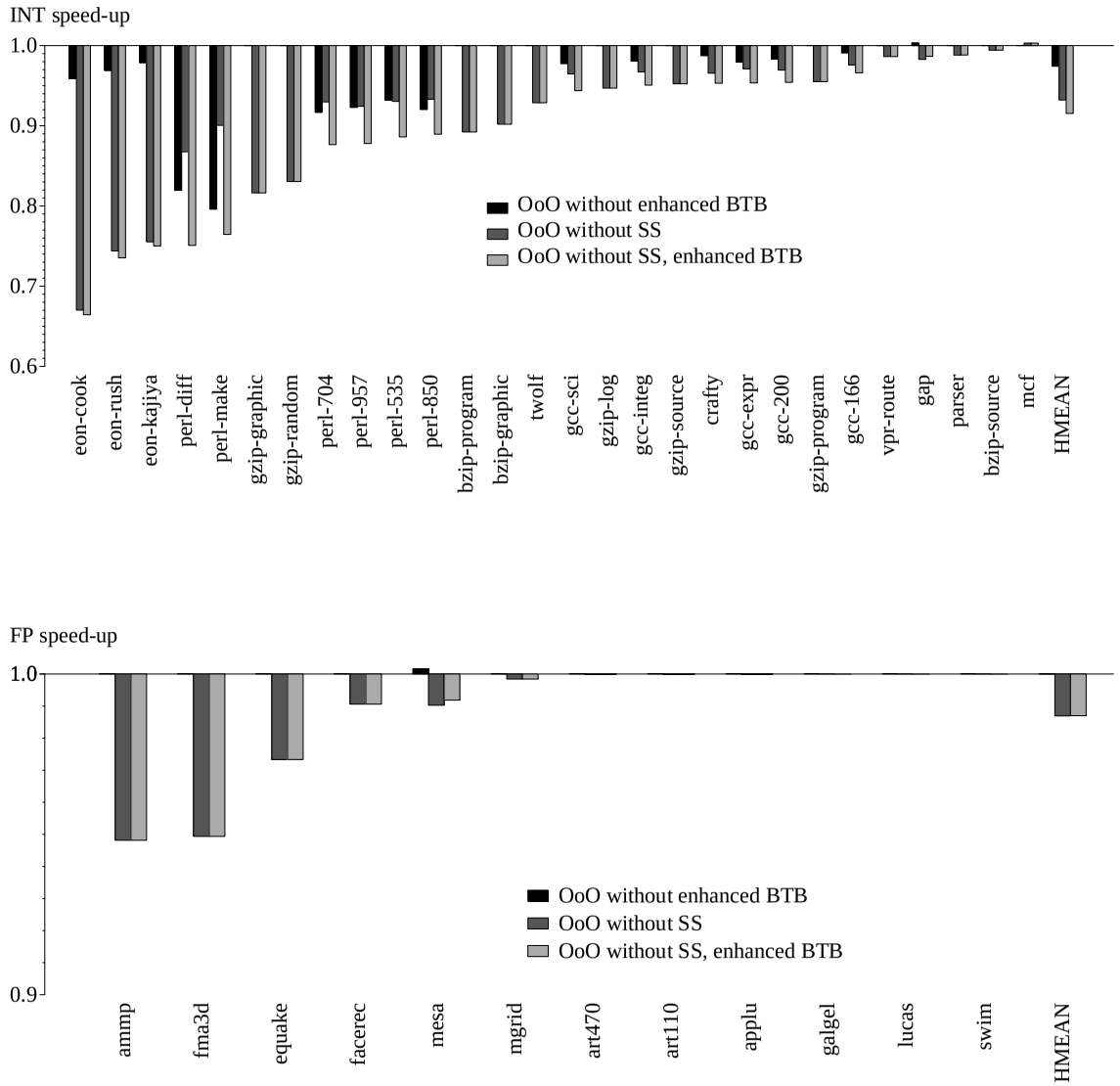
The ReLaSch processor executes most of the instructions in the Rcache mode. Therefore, it makes sense to reduce the size of the Icache in order to compensate the area required by the Rcache. Figure 9.29 shows the speed-up with different sizes of the L1 Icache in ReLaSch over the baseline OoO processor, which has a 64-KB Icache.

The FP benchmarks show almost no degradation even with an Icache that is 8 times smaller. On the other hand, many INT benchmarks have a higher amount of instructions executed in the Icache mode due to the presence of branches that the Rcreate logic is not able to predict correctly. These benchmarks benefit from larger Icaches, so reducing the size of the Icache degrades the IPC in most cases. However, it still seems acceptable to lower the average INT speed-up from 0.99 to 0.98 and reduce the Icache to 32KB. Halving it again reduces the speed-up to 0.96.



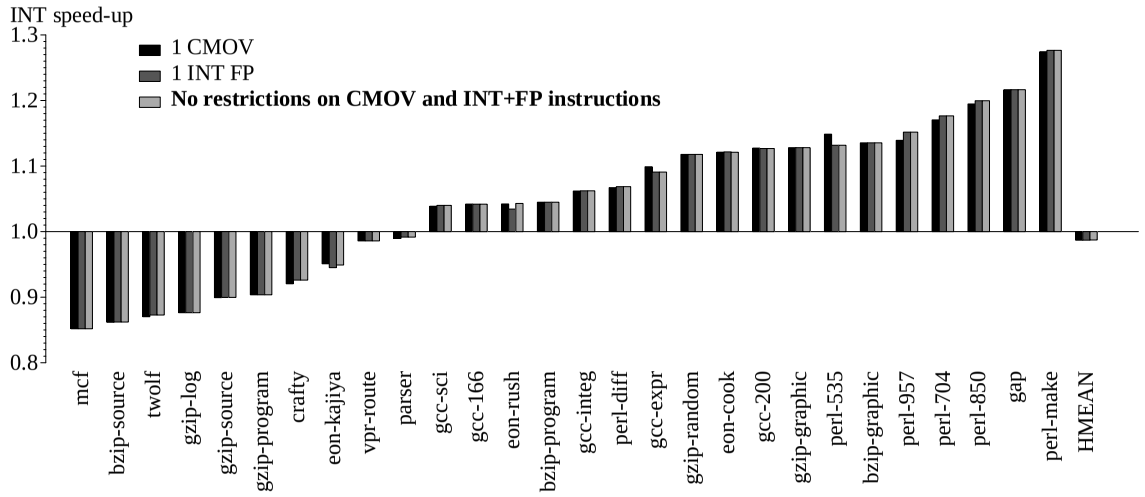
	ReLaSch	ReLaSch with SS, enhanced BTB
HMEAN INT	0.99	0.99
HMEAN FP	1.02	1.02
HMEAN ALL	1.00	1.00

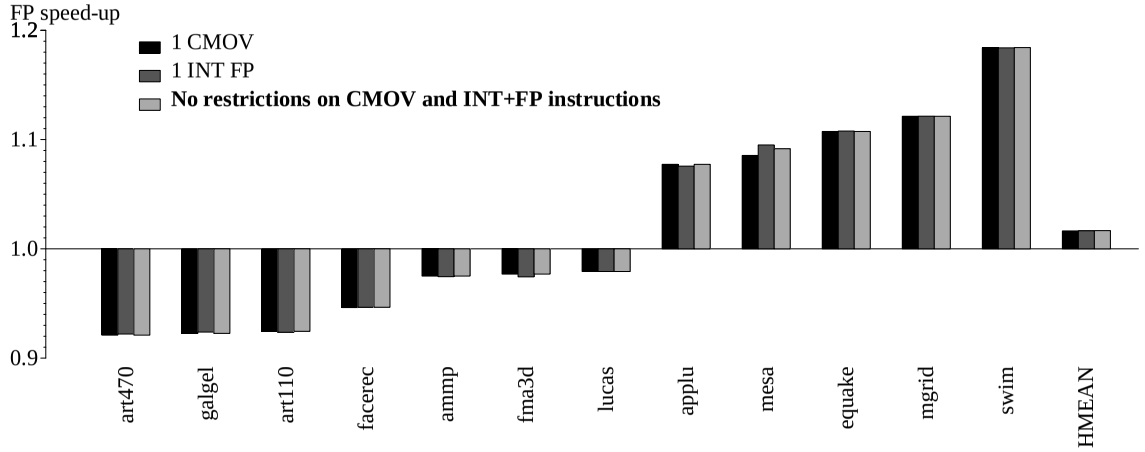
Figure 9.26: Speed-up of the ReLaSch processor with Store Sets and an enhanced BTB.



	OoO wo. enhanced BTB	OoO wo. SS	OoO wo. SS neither enhanced BTB
HMEAN INT	0.97	0.93	0.92
HMEAN FP	1.00	0.99	0.99
HMEAN ALL	0.98	0.95	0.94

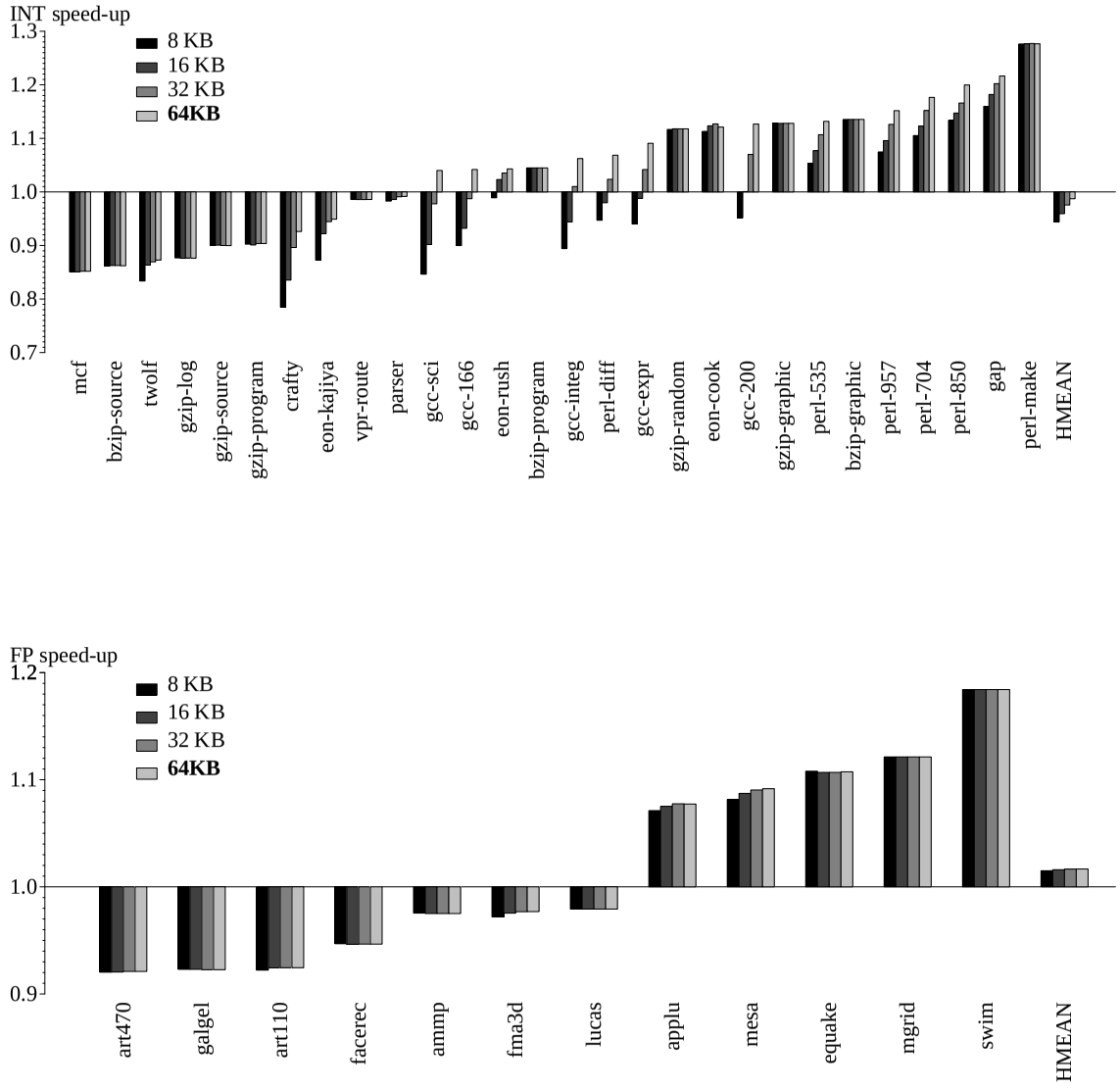
Figure 9.27: Speed-up of the OoO processor without Store Sets and without an enhanced BTB.





	1 CMOV	1 INT+FP	**no restrictions**
HMEAN INT	0.99	0.99	0.99
HMEAN FP	1.02	1.02	1.02
HMEAN ALL	1.00	1.00	1.00

Figure 9.28: Speed-up with different maximum number of CMOV and INT+FP instructions issued per cycle.



size in KB	1	2	4	8	16	32	**64**
HMEAN INT	0.91	0.92	0.93	0.94	0.96	0.98	0.99
HMEAN FP	1.01	1.01	1.01	1.02	1.02	1.02	1.02
HMEAN ALL	0.94	0.95	0.96	0.97	0.98	0.99	1.00

Figure 9.29: Speed-up with different sizes of the L1 Icache.

9.5.5 Branch predictor

The default configuration of ReLaSch uses the original Alpha 21264 branch predictor in the Icache mode. Since most instructions are executed in the Rcache mode, it is not necessary to use such a complex predictor, formed by a bimodal predictor, a history-based predictor and a choice predictor [26].

Figure 9.30 shows the speed-up achieved with a much simpler bimodal branch predictor. There is no performance degradation when the bimodal predictor is used. The size of the predictor (the number of counters) can be reduced without a significant impact in the average speed-up, though some INT benchmarks (i.e. *crafty*, *perl-diff* or the *gcc* benchmarks) yields less IPC.

9.5.6 ROB size

Figure 9.31 shows the speed-up achieved with different ROB sizes. The number of physical registers and the size of the LQ and SQ have been scaled in the experiments with a 128- or 256-entry ROB. The scaling is not strictly linear since, due a restriction of the simulator, the size of these structures must be a power of 2, while the default ROB has 80 entries. Thus, the ReLaSch configuration with a 128-entry ROB uses a 64-entry LQ and SQ and 16 physical registers per logical register. These values are doubled in the experiment with a 256-entry ROB. FP benchmarks clearly benefit from a larger ROB, while most INT benchmarks do not show much improvement after certain point.

For comparison, figure 9.32 shows the speed-up with different sizes of the ROB of the OoO processor for the INT and the FP benchmarks as a reference. The register file and the LQ and SQ queues have been scaled in the same way as in ReLaSch. Note that the order of the benchmarks and the range of the Y axis are different than in figure 9.31. The OoO processor achieves less average IPC from larger ROBs than ReLaSch does. Having a larger ROB allows exploiting the broader vision of the code of the Rcreate logic to a greater extent.

9.6 Final results

In this section we present the results of the ReLaSch processor with all the changes in the configuration that have been discussed in this chapter and that are summarized below. Most changes in the configuration aim to simplify the logic of the ReLaSch processor without compromising performance. Therefore, we don't expect to improve performance with the new default configuration. The changes are summarized in table 9.2.

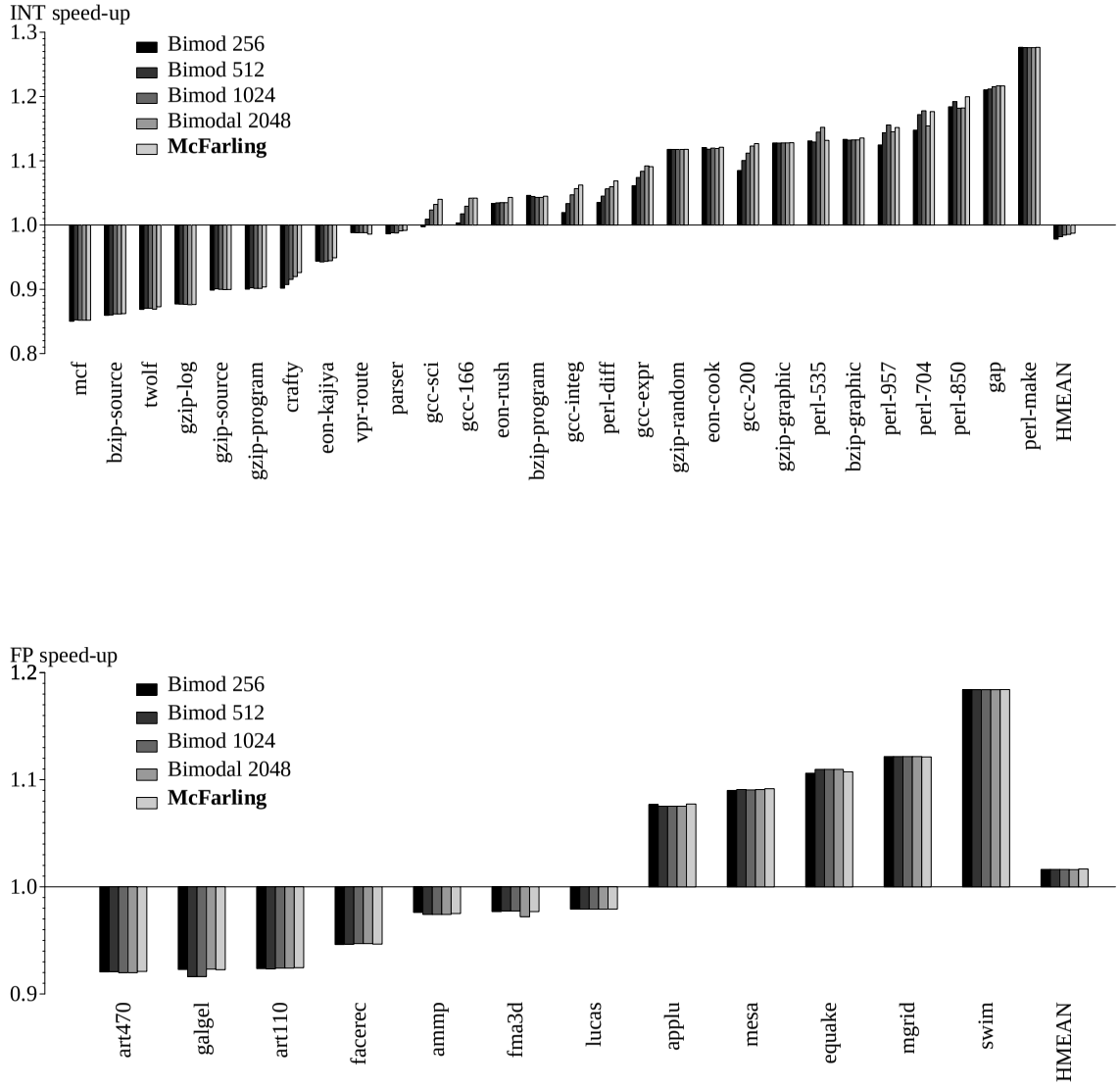
One of the most significant changes is the absence of history to identify the rgroups. Therefore, the history bits are not present in the Rcache and the logic used to detect if an access hits only uses the PC as index and tag. Besides, there is no need to remember the history in the Fetch and Commit stages, as well as in the Rfetch and the Rcreate logic.

When the Rcreate logic closes an rgroup it checks if the next instruction to schedule was executed in the Rcache mode. If it was, the Rcreate logic stops scheduling and goes to the Idle mode. The former version scheduled an additional rgroup before changing to the Idle mode.

The predictor used in the Rcreate logic to choose the latency of the load instructions has now 256 counters of one bit each instead of four-bit counters. Also, the counter in the Rcache used to detect the "bad rgroups" has four bits per cache line instead of five bits.

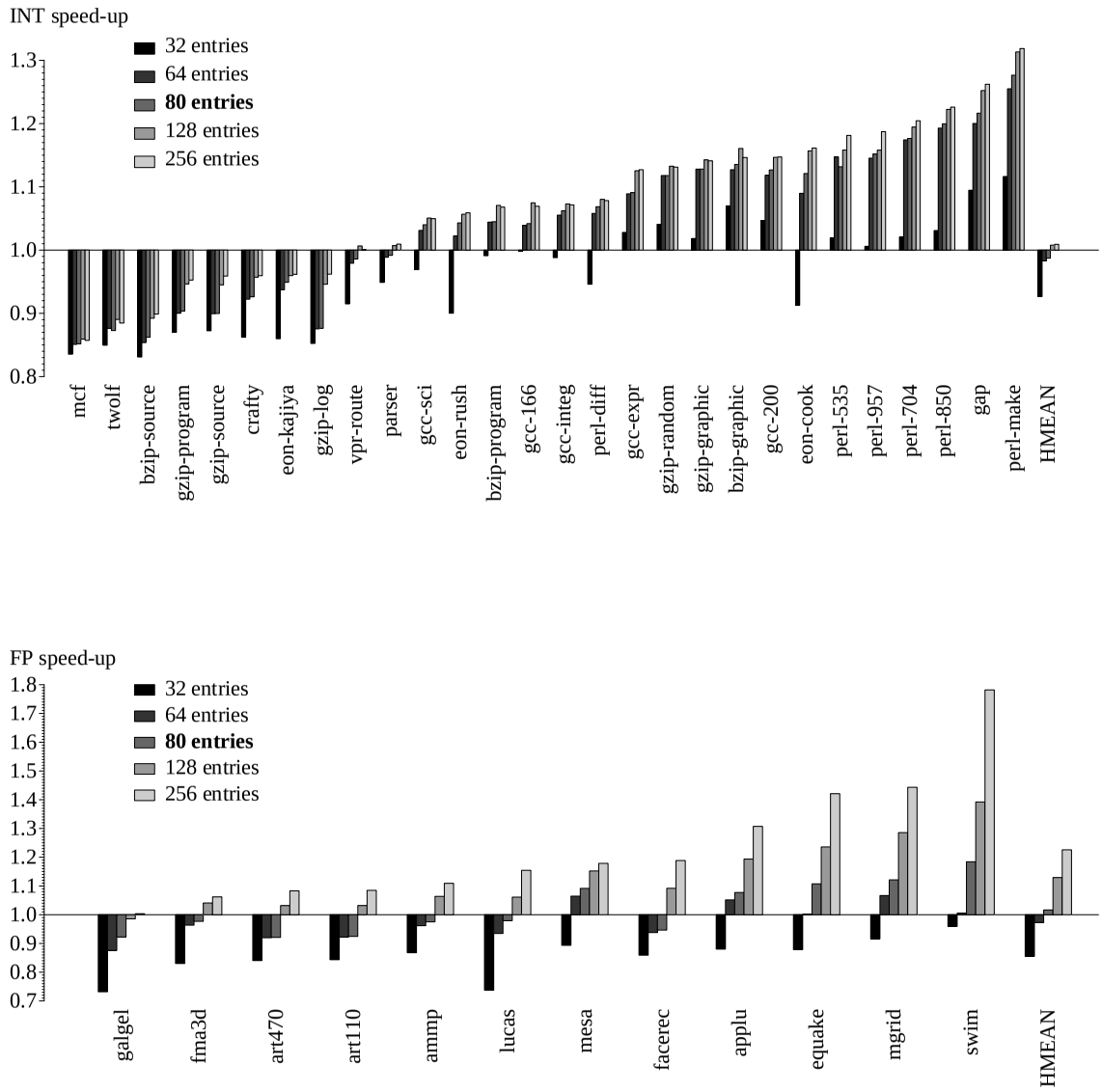
Some of the changes simplify the logic or reduce the area required by several structures, thanks to the fact that most of the time the processor executes instructions in the Rcache mode. Thus, a bimodal predictor of 2,048 entries is used instead of the much more complex McFarling predictor of the Alpha 21264. Besides, the Icache is reduced from 64KB to 32KB.

Up to one conditional move is allowed to issue per cycle, as well as up to one instruction that is present in both the integer and the floating point buffers. Both changes simplify the issue logic and



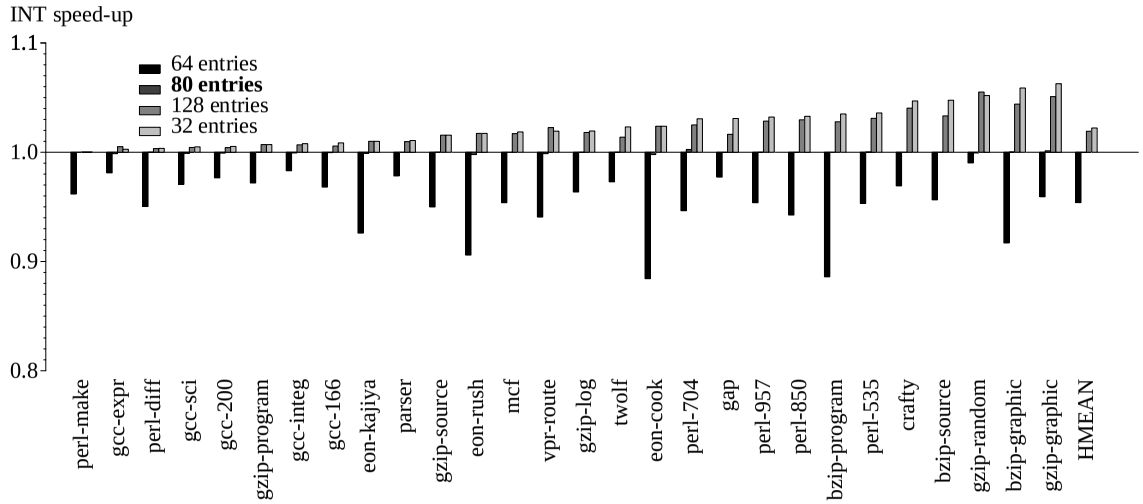
Bimodal predictor size	256	512	1024	2048	**McFarling**
HMEAN INT	0.98	0.98	0.98	0.99	0.99
HMEAN FP	1.02	1.02	1.02	1.02	1.02
HMEAN ALL	0.99	0.99	0.99	1.00	1.00

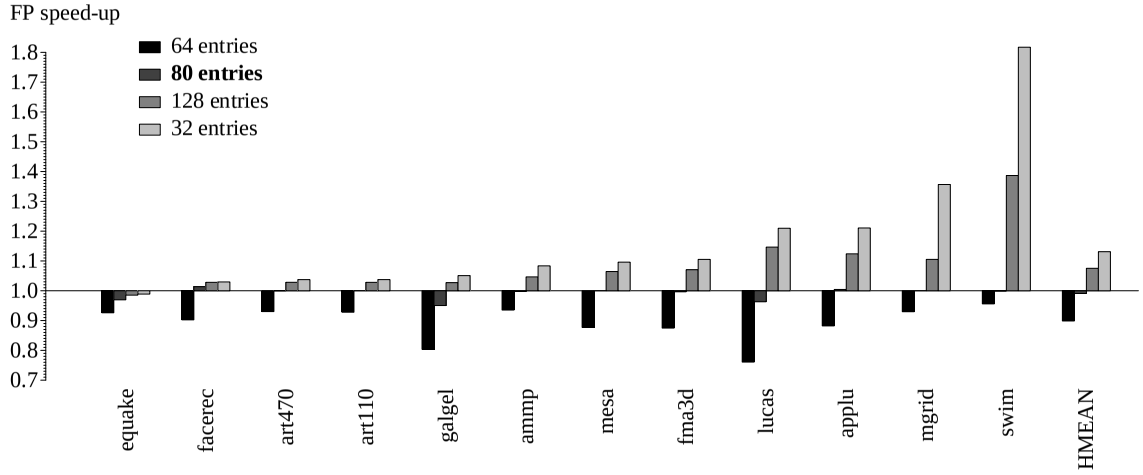
Figure 9.30: Speed-up with different branch predictors.



ROB size	32	64	**80**	128	256
HMEAN INT	0.93	0.98	0.99	1.01	1.01
HMEAN FP	0.86	0.97	1.02	1.13	1.23
HMEAN ALL	0.90	0.98	1.00	1.05	1.07

Figure 9.31: Speed-up with different sizes of the ROB of the ReLaSch processor.





ROB size	64	**80**	128	32
HMEAN INT	0.95	1.00	1.02	1.02
HMEAN FP	0.90	0.99	1.08	1.13
HMEAN ALL	0.93	1.00	1.04	1.06

Figure 9.32: Speed-up with different sizes of the ROB of the OoO processor.

Parameter	New Default	Old Default
Instruction L1 cache:	32KB	64KB
Branch predictor:	2K-entry bimodal	McFarling
Rgroup: indirect branches	8	10
Rcreate: instructions per cycle	1	4
Rcreate: load latency predictors	256 1-bit	256 5-bit
Rcreate: switch to Idle mode	after current rgroup	wait one rgroup
Rcache: "bad rgroup" counters	4-bit	5-bit
Rcache: History bits	No	6-bit history
Issue: Conditional move per cycle	Up to one	Unrestricted
Issue: FP+INT instructions per cycle	Up to one	Unrestricted

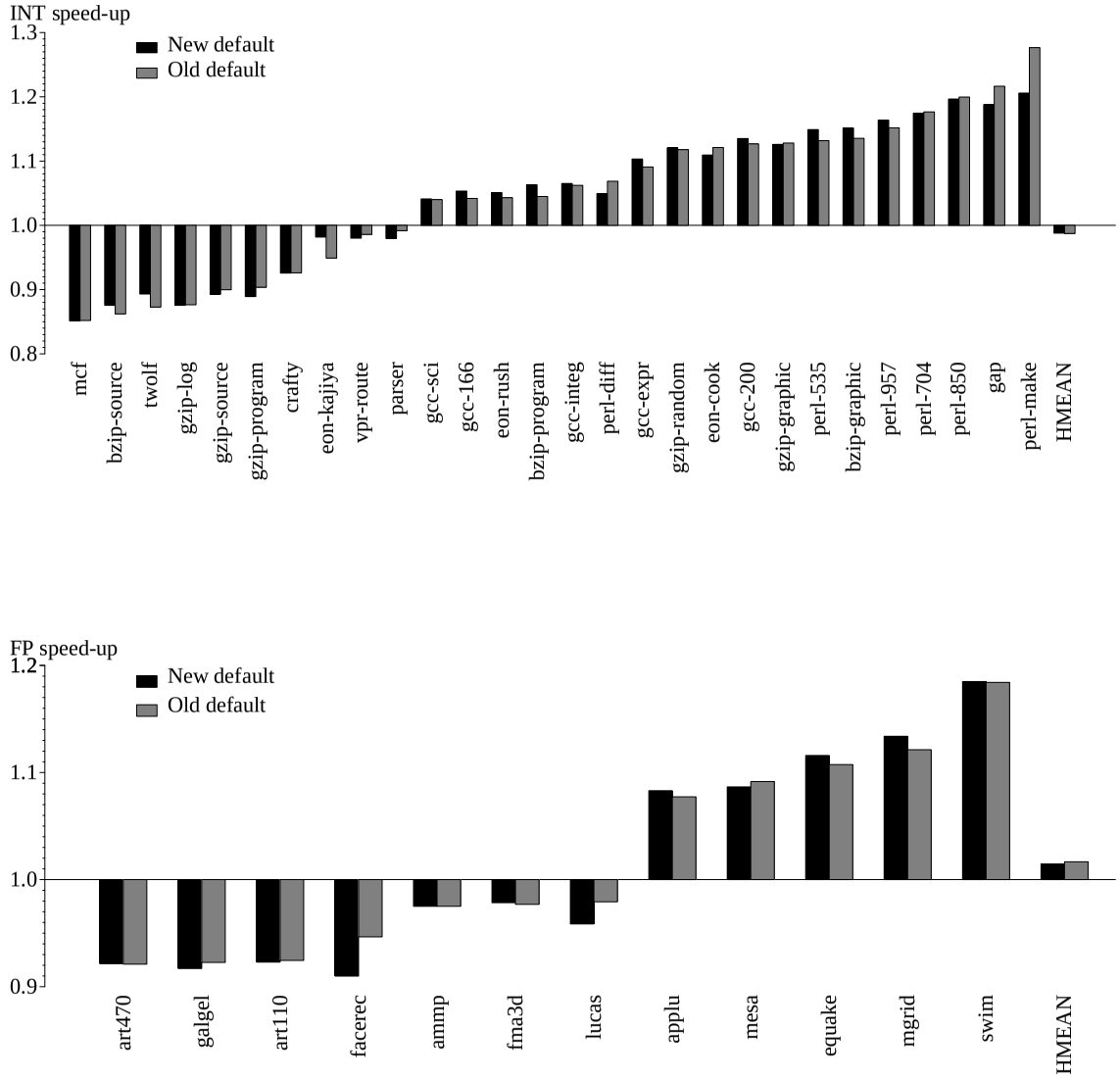
Table 9.2: Main changes in the default configuration of ReLaSch.

reduce the number of ports to the register file with respect to the old unrestricted version. Also, there can be up to 8 indirect branches per rgroup instead of 10.

The Rcreate logic processes up to one instruction per cycle instead of four. An `rcreate_input` buffer of 512 entries is still used to store the instructions until they are scheduled. Such a size is enough to compensate the lower width of the Rcreate logic when compared with the Commit stage (up to 11 instructions per cycle).

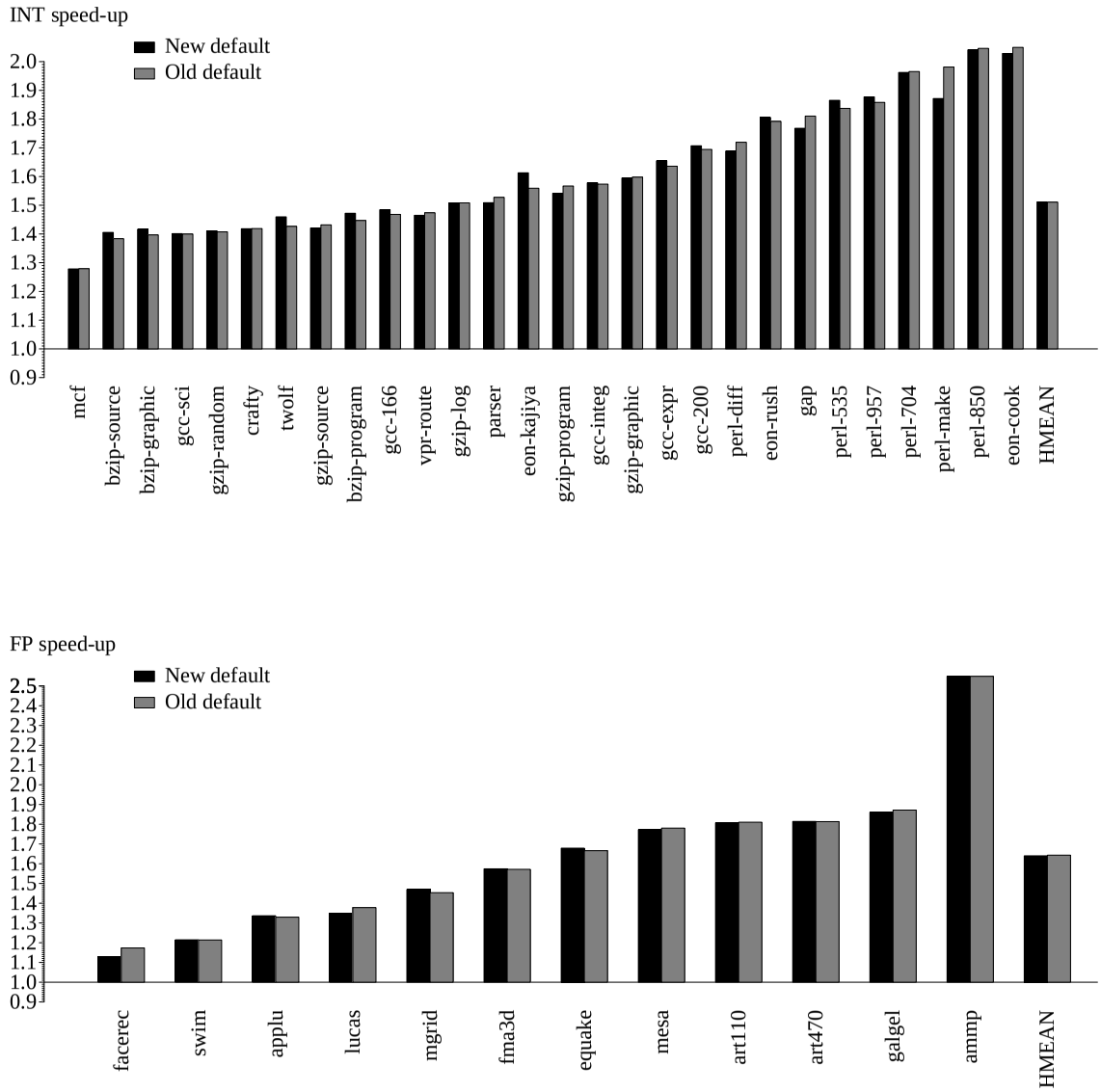
Figure 9.33 shows the results of the old and new default ReLaSch processors. The changes introduced in the configuration to simplify the processor have a negative impact in some of the benchmarks. The benchmark *perl-make* and *facerec* show the most noticeable performance degradation. The main reason is a significant increase in the number of mispredicted branches. Nevertheless, although many structures have been simplified or use less area, the processor has still has the same average IPC than the baseline OoO.

Figure 9.34 shows the speed-up of the two same configurations of ReLaSch when they are compared with the reference IO processor. The speed-up is very significant in the majority of benchmarks. ReLaSch performs better than IO in all cases and in average it has 1.55 speed-up over the IO processor.



	New default	Old default
HMEAN INT	0.99	0.99
HMEAN FP	1.01	1.02
HMEAN ALL	1.00	1.00

Figure 9.33: Speed-up with the new and the old default configurations of the ReLaSch processor.



	New default	Old default
HMEAN INT	1.51	1.51
HMEAN FP	1.64	1.64
HMEAN ALL	1.55	1.56

Figure 9.34: Speed-up with the new and the old default configurations of the ReLaSch processor when compared with the reference IO processor.

Chapter 10

Related work

There are other proposed processors that schedule instructions outside the critical path of execution, and cache the schedules to feed the pipeline later. There are also other attempts to simplify the issue-logic that don't imply caching the schedules. In this chapter, we first present the approach and objectives of those more similar to our work, followed by the related work that doesn't cache the schedules.

10.1 Caching proposals

10.1.1 DIF

DIF [8] stands for Dynamic Instruction Formatting. The proposal includes two cores: the primary engine (simple, in-order and super-pipelined) and the parallel engine (a VLIW-like core). It also places a translator between the two cores that has as input stream the instructions executed in the primary engine and schedules these instructions in a way that suits the parallel engine. The schedules, known as groups, are stored in the DIF-cache. The parallel engine is used whenever it is possible.

Since the primary engine offers a "safety net", the ISA of the parallel engine may not match the ISA offered to the programmer. For example, complex and infrequent instructions may be not implemented in the parallel engine but only in the primary engine. The presence of such an instruction forces to finish the current schedule. Also the encoding of the instructions can be chosen to maximize efficiency at execution time. It also can be changed in each generation of the design without modifying the ISA exposed to the programmer.

The groups are sequence of long instruction words (LIW). The results show experiments with up to eight LIWs per group and up to eight instructions (plus two branches) per LIW. The group is assumed to commit atomically, though it allows having several exits according to the outcome of several conditional branches. The registers in the groups are renamed using a two-level renaming scheme; an identifier is assigned at scheduling time and the identifier is updated later, at execution time. The register file has a fixed set of physical registers for each logical register. However, physical registers cannot be reused within a single group. They can only be written once per group. Instructions are speculated across branches. To be able to recover from speculation, an exit map of the live registers is stored for each conditional branch. Alternatively, it can be constructed at execution time.

The parallel engine can be designed to have homogeneous FUs (each FU can execute any instruction) or typed (each FU can execute only a subset of the instructions). The typed configuration can use a split schedule cache and register file and it is likely to have a more compact design, that allows having a larger number of FUs.

The scheduler uses a greedy algorithm to place the instructions. It tracks dependences and always uses a latency of either one or two cycles. Any instruction that has more than one cycle of latency is scheduled assuming to have two cycles of latency. The motivation is to generate compact schedules instead of sparse ones. At execution time, the instruction will stall until the operands are available.

To close a schedule, the scheduler inserts an unconditional branch in the last slot. It removes all unconditional branches in the stream of committed instructions of the primary engine. If the exit maps associated with each conditional branch are stored with the group, the number of conditional branches per group is limited. Indirect branches would be included in the schedule for object-oriented workloads. Since the benchmarks used in the experiments aren't object-oriented code, the proposal assumes an indirect branch finishes a group. The path predicted by the scheduler is the one learned at scheduling time. If the prediction turns out to be incorrect, it must be unlearned, using counters associated with each conditional branch in the group.

Memory instructions are scheduled using the addresses seen at scheduling time to predict if two instructions will alias. The instructions include the original order, which is used by a store and load queue to detect any unexpected alias at execution time.

The DIF processor is presented as an improvement over in-order processors. It is not compared with a conventional out-of-order processor but only with an in-order version of DIF, in which the scheduler schedules consecutive independent instructions into groups without reordering the original instructions.

The paper that presents DIF [8] evaluates eight SPECint95 programs, executing between 30 and 120 millions of instructions. Experiments are made with a trace-based simulator. As it could be expected, DIF has better results than the in-order DIF, up to 50%. Also, using larger groups is beneficial, as well as wider LIWs (up to four instructions plus two control instructions). The I-cache in the primary core can be very small without degrading the overall performance. A small DIF cache of 256x2 groups yields good performance for many of the benchmarks. Some other benchmarks benefit from having larger caches (1024x2). It is beneficial to allow basic block duplication (have several instances of the same basic block in different groups). In some cases it is equivalent to perform loop unrolling. The data cache used in the experiments has a one-cycle hit latency and only one-cycle miss penalty.

The methodology used in the experiments is weak. The authors compare DIF only to a in-order version of DIF. The groups used in the experiments include a relatively small number of instructions (up to 48). Also, the latencies for the data cache are completely unrealistic. In the worst case the processor stalls only for two cycles waiting for a cache miss. With such a small penalty, they are able to use a simplified scheduling algorithm that assumes either one or two cycles of latency for all instructions. They don't have to deal with complex prediction of load latencies, since the penalty for choosing a wrong latency is very small. A more realistic model of the memory system would result in very significant performance degradation.

DIF uses two different cores instead of sharing the execution pipeline like ReLaSch does. This implies a duplication of resources such as the functional units. Though the impact in area is not examined in the paper, it is clear that it cannot be an area-effective design. The organization of the register file is very similar to the one used in ReLaSch though a physical register cannot be reused within a group unlike ReLaSch. This is likely to lead to less populated schedules due to a lack of available physical registers. The reason why the physical registers cannot be reused is the exit maps of the conditional branches, that indicate which physical registers hold the architectural value on a group exit. If multiple uses of a given physical register are allowed, it may be ambiguous which one holds the architectural value. Our gradual commit using the ROB solves this problem and enables reusing a physical register within a schedule as much as it is required. Moreover, our design with a ROB and a single engine enables having in-flight instructions both from the Icache and the Rcache simultaneously, while DIF requires that a group commits and its exit map indicates the architectural values before it starts executing the instructions in the primary engine.

Though our very different initial approaches, DIF and ReLaSch share several design decisions such as predicting branches and memory aliases from their last execution seen at scheduling time. The coincidences strengthen our confidence on these decisions. All alternatives that we have explored yield worse results.

DTSLVIW

The DTSLVIW proposal [27, 28] is an independent proposal that resembles DIF. It also has two different cores; the primary one is a simple in-order scalar processor and the second one a VLIW core. A scheduler creates blocks of VLIW code from the stream of committed instructions of the primary core. These blocks are stored in a VLIW cache and feed the VLIW engine whenever it is possible. A main difference with DIF is that VLIW instructions are renamed to remove hazards and explicit copy instructions are placed in the program order to update the architectural registers. Thus, no renaming logic is needed at execution time in the VLIW core. Control speculation is handled in a similar way and copy instructions update the registers of the instructions scheduled before a branch. The scheduler uses an algorithm based on First Come First Served instead of the greedy approach of DIF.

10.1.2 rePLay

The rePLay hardware framework [9] aims to improve performance by dynamically optimizing the code. It relies on a conventional out-of-order pipeline to execute the instructions. The optimizer works on frames, which are sequences of instructions with a single entry point and a single exit point. Frames commit atomically; that is, either all the instructions in the frame or none of them commit. In order to construct large frames that span across many basic blocks, highly biased branches are promoted into assertions. At execution time, assertions fire a recovery mechanism in case the condition evaluates as false. The recovery mechanism undoes all the work done by the frame and sets the fetch mechanism to read the instructions of the frame from the instruction cache.

Traces commit atomically to allow more aggressive optimizations, requiring that useful instructions are re-executed after a mispredicted branch. The out-of-order pipeline schedules and renames the instructions.

The frame construction mechanism and the branch bias detection table are explained in great detail [9, 29]. Path history and branch PC are used to index the table. It stores a counter of the number of consecutive executions of a given branch with the same outcome. Whenever a counter reaches a fixed threshold its branch is promoted into an assertion and a frame is constructed from the stream of committed instructions. A frame ends when a non-promoted branch is inserted or a maximum number of instructions (256) is reached. Frames including less than 32 instructions or 5 blocks are dismissed.

The frame cache stores the frames once they have been constructed. Each frame occupies several cache lines in order to efficiently store frames of different sizes, though the experiments in [9] store one frame per entry.

At execution time, a conventional branch predictor is combined with a frame predictor and a selection mechanism.

Using a frame cache that can store up to 256 frames, a 64KB branch bias detection table (and 10KB for indirect branches), 16K-entry frame predictor and 6 bits of history, the rePLay framework is able to create frames of 88 instructions in average that cover 68% of the dynamic instruction count with a completion rate of 97.91%. The frame predictor hit rate is 81.26%.

The framework allows the frames to be optimized in hardware after they are constructed but before being stored in the frame cache [30]. Unoptimized frames still increase performance because the frame

cache works like a trace cache, executing 13% less execution cycles. When the frames are optimized, the performance gain is 21% less cycles than the baseline.

The optimized frames can use renamed registers [31]. The position of an instruction in the frame directly identifies its physical register. A Register Alias Table is used to communicate where are located the Live-In and Live-Out values.

Continuous optimization proposal [32] moved some optimizations present in the rePLay processor to the execution pipeline. The optimizations are not cached. They are dynamically applied to the stream of fetched instructions instead.

In-order rePLay

An in-order version of rePLay is presented in [33]. The frames are executed on a conventional in-order processor. After a frame is optimized a scheduler reorders the instructions to improve the performance of the in-order pipeline. Then the frame is stored in the frame cache.

The in-order processor and the scheduler is not described with much detail. Since frames commit atomically, the scheduler has a lot of freedom to reorder the instructions. It uses assertions to be able to speculate on memory aliasing and reorder load and store instructions.

With the Region Slip technique presented in the same paper, the in-order pipeline is enhanced with the Live-Out Map table, that indicates which instructions produce the values that are alive after exiting the last frame. It allows overlapping the execution of the last instructions of one frame with the first instructions of the next one.

The scheduler reduces the execution latency by 28% over the unscheduled code executed in the same in-order pipeline. Region slip improves these results and improves performance 26% over the plain scheduled in-order rePLay.

Comparison

The rePLay framework uses a conventional out-of-order issue logic that relies on a large issue queue, while the in-order rePLay uses an in-order issue logic. It has to store the live-out registers of each trace. Though the experimental methodology is different and direct comparison of the results is not very reliable, rePLay yields worse results than ReLaSch versus conventional out-of-order and in-order processors. Only highly biased branches can form part of a frame. Thus, the completion rate is very high, which is very energy-efficient since little energy is wasted in miss-speculated instructions. A similar filtering mechanism can be added to the ReLaSch processor too, in order to reduce energy consumption and lower the pressure on the Rcache.

The main goal of rePLay is to borrow compiler optimization techniques in order to dynamically reduce the number of instructions that are executed. ReLaSch does not use any of these techniques, because their use forces to have atomic commit: since not all the original instructions are executed, the semantics of the program inside the schedule are changed, even though the semantics are preserved on the schedule boundaries. However, it seems possible to apply some of this techniques in ReLaSch and force atomic commit to a selected number of rgroups that are know to be executed frequently and that always commit all their instructions.

10.1.3 Parrot

Parrot [34, 35] is similar to rePLay. It also optimizes the most frequently executed traces and caches them. The schedules are executed in an out-of-order pipeline afterwards. It aims to perform gradual optimizations of the code as a power-efficient way to improve the performance of the code.

It has two separated pipelines, hot and cold. Both are out-of-order but the hot pipeline executes the traces atomically and the cold pipeline is a conventional out-of-order pipeline that is used when

there isn't any available trace. Depending on the configuration, the hot and the cold pipelines can share some resources or can be completely separated, like the double core of the DIF processor. An additional similarity between these two proposals is that complex and infrequent instructions may be implemented only in the cold pipeline.

Indirect jumps and taken backward branches terminate a trace. A trace can have up to 64 uops.

The optimizer is placed after the commit stage of both pipelines. Parrot filters the traces, using two different thresholds, in order to schedule only the most frequently executed instructions. It has counters associated with the different execution paths, updated after commit. When the counter of a path reaches the first threshold, a trace of the decoded uops of that path is stored in the cache and is executed thereafter in the hot pipeline. The benefit is a reduction in the energy required to decode the instructions. When a trace executed in the hot pipeline reaches the second threshold, the trace is optimized and stored again in the cache. Thus, the optimizer processes only the parts of the code that are frequently executed, saving power.

The optimizations that Parrot performs are presented in [35] and include logic and arithmetic simplification, dead code elimination, SIMdification and pre-scheduling of the instructions among others.

The results of Parrot are compared with those of a conventional out-of-order processor without a trace cache and an optimizer. The 4-wide Parrot processor with a 128-entry trace cache achieves 17% speed-up over a conventional 4-wide out-of-order processor. The 8-wide Parrot processor with a 512-entry trace cache achieves 25% speed-up over a conventional 8-wide out-of-order processor.

10.1.4 Execution Cache / Flywheel

The Execution Cache (EC) [10] proposal fills a cache of traces generated by a conventional out-of-order issue-logic. Whenever it is possible, instructions are fetched from the EC and the front-end of the pipeline is switched off. The goal of the proposal is to reduce the energy consumed per instruction. The drawback is usually some performance degradation when the EC is used.

The Execution Cache is placed after the issue stage. A fill buffer captures the instructions issued and creates traces out of them. Since we schedule the instructions after commit, we have a broader vision of the code, that enables creating better schedules.

A trace contains up to 512 instructions but the actual maximum length can be changed dynamically according to the branch misprediction rate. A trace is closed on a function return or an indirect branch. Also when an existing trace starts in the current point of execution. Besides, a branch misprediction that occurs during the creation of a trace forces to discard it completely, since it would have instructions from both the correct and the wrong path. Instructions commit gradually. The instructions from just a single trace can be at the same time in the ROB.

The trace cache has a tag array and a data array. Each block in the data array stores up to 8 instructions. A trace typically occupies many blocks of consecutive sets. Each block contains a pointer to the way that stores the next block of the same trace in the next set. Since the sequence of sets to be accessed is known beforehand, a banked implementation of the data array allows accessing just the bank that contains the desired set. The rest of banks are gated, thus saving energy. Only the first access to a trace requires accessing all banks. The trace is formed by a sequence of Issue Units (similar to our issue-groups). A block of the Data Array can store an arbitrary number of Issue Units.

If a trace is aborted at execution (because it contains a mispredicted branch) M times in a row, it is invalidated and created again. The values used for M are 2 and 3.

The register file has a pool of physical registers for each logical register. The pool is implemented as a circular buffer. The renamed instructions in a trace assume that the physical register 0 stores the live-in values. To avoid having an offset per logical register and use adders (like we do in ReLaSch) or copying registers when a trace starts execution, all the identifier fields are xor-ed with the identifier of the current architectural physical register. This register becomes the register with identifier 0 and the

rest gets a unique identifier. The identifier of the physical registers are stored in a special field and an associative access is performed to determine where in the pool is a given physical register. The two drawbacks of this approach are the need for an associative access and that a trace must stall until all older instructions commit before starting execution, creating a pipeline bubble.

Each block in the trace cache stores eight instructions using 76 bytes. The instructions are stored already decoded using 48 bits. The experiments use an EC of 50KB and 100KB. The register file has four physical registers per logical register. A mix of benchmarks from SPEC95 and SPEC2000 is used to evaluate the proposal, simulating 50M instructions

The processor with a 100K EC is 8.5% slower than the baseline (a conventional out-of-order processor). When it uses a 50K EC, it is 9.8% slower. The processor with a 50K EC reduces a 35% the energy per committed instruction The processor with a 100K EC reduces a 31% the energy per committed instruction. The increase in power consumption for the larger EC is offset by the performance improvement. In the Energy x Delay product, the proposed architecture is 20% better than the baseline.

Flywheel

Flywheel [36, 37] introduces the EC in a processor that has different clocks in the front-end and the back-end of the pipeline. The processor has several execution modes that enable to both increase performance and save energy. The slowest clock is the one used by the select logic when it accesses the issue window. It determines the clock used by the back-end when the EC doesn't contain any suitable trace. The front-end can have a faster clock even though it accesses the issue window. Also, when the EC provides the instructions, the back-end can be clocked faster and the front-end can be switched off.

The other main difference with the original EC proposal is how register renaming is performed and the structure of the register file. Though it still uses a limited pool of physical registers to rename each architected register, it adds a new level of renaming. The register file is organized as a single continuous structure with all the physical registers. For each architected register, a table stores an offset to its first physical register and how many of them are available. This approach has two benefits: it removes the associative access to the register file and it enables to dynamically adapt to the usage of each architected register: if some registers are written much more often than others, the amount of physical registers in their pool can be incremented. The drawback is that all traces stored in the EC must be invalidated and created again whenever the configuration of the register file is changed.

The experiments use a 512-entry register file and a 128K EC. In average, 88% of the time the EC provides the instructions and in the worst case it happens 60% of the time. Using the same clock as the conventional out-of-order baseline that is determined by the select logic, Flywheel has an average 5% increase in performance due to its reduced misprediction penalty. When the clock is more aggressive (50% faster clock in the back-end with the EC than the select logic and also 50% faster in the front-end), Flywheel achieves an average 54% increase in performance. On the other hand, it increases the power in only 8%. In average, it requires 30% less energy than the baseline thanks to the reuse of traces.

10.1.5 CTS

The Converged Trace Schedules (CTS) proposal [11] uses a scheduler out of the critical path to speed-up an in-order pipeline. It applies software pipelining and loop unrolling to the most frequently executed loops. It breaks up the schedule into trace blocks, which are defined by the taken backward branches. Then it tries to find the largest repeating pattern in the sequence of blocks. This sequence is called a Converged Trace Schedule. The schedules are stored in a set of dedicated pages in virtual memory and a copy of the currently used blocks is stored in a small dedicated cache (8KB). The paper

proposes three different implementations of the scheduler: in a software thread running in the same processor as the optimized program (with the associated overhead when running the scheduler); in a software thread running in a different core (without the overhead); and in a hardware in a dedicated co-processor.

The CT-Schedule driver keeps track of all committed backward branches, counting the number of times they are executed and how many times they are taken in a row. When they reach a given threshold in both counters (50 and 5 respectively) the processor enters the scheduling mode, in which committed instructions are captured in the schedule window and then used by the scheduler. The driver stops trace generation when a converged trace has been created, when the backward branch is not taken or after a maximum number of iterations. Once a backward branch has triggered the scheduler, it is flagged in order to not trigger it again.

The schedule window contains up to 500 instructions. The instructions are annotated with register and memory dependences as well as the execution latency. The scheduler uses a greedy algorithm and assumes perfect renaming at execution time. In order to converge the traces, the scheduler follows additional rules: a) the branches are scheduled in order with respect to each other, b) the instructions cannot be speculated above a maximum number of branches (six in their baseline) and c) each static load is scheduled with a fixed latency inside the schedule. To choose the latency of each load, they use a heuristic to measure its criticality: how many of the following 100 instructions in the schedule depend directly or indirectly on the load. If at least 20% of them depend on the load, it is consider to be in the critical path and it is scheduled assuming a hit; otherwise, the miss latency is used.

Once a schedule has been created, the taken backward branches divide it into several trace blocks. The instructions of each trace block are hashed to produce a block ID. The sequence of IDs is processed by an algorithm that detects repeating sequences of IDs. It uses trees associated with each ID and counters to detect which is the most frequent pattern.

The register file is divided into several register windows. Each window has as many physical registers as logical registers are defined in the ISA. Each window is directly linked to an iteration of the loop. An instruction can be speculatively scheduled above several branches. The register window that the instruction uses is encoded as an offset and is determined by the number of those branches. At execution time, a mapping stage uses the offsets encoded in the instruction and the current iteration number to rename it (i.e. which window stores its source operands). It is not clear from the paper how WAR and WAW hazards are handled when they happen to be within a single loop iteration, since there is one copy of each logical register per window and each iteration is linked to a window.

The paper presents the results of CTS against a plain in-order processor. Also against an in-order processor plus a simple scheduler (similar to DIF) that does not converge traces and that has small number of instructions (up to 64). It also presents CTS enhanced with Pattern Trace Schedules (PTS), that stores the most frequent pattern when no trace converges after the scheduling process finishes.

The size of the CTS traces is extremely variable because it depends on the patterns that the scheduler is able to find. The scheduler is able to create scheduled of more than 900 static instructions that turn into 19,400 dynamic instructions at execution time though most traces are smaller. The percentage of instructions executed from the cache is as high as 98.3% and as low as 2%. It is 52% in average. CTS+PTS has in average 16% IPC speedup over the in-order baseline when the scheduler is implemented in hardware. The simple scheduler achieves 8% of speedup in average.

CTS filters the traces in order to schedule only the hottest parts of the code in order to lower the pressure on the cache and the scheduler. So the schedules don't attempt to cover all the instructions that are executed, with a potential performance loss that is supposed to be small since it is the less frequently executed instructions that are not scheduled.

On an exception, a misprediction or the end of a trace, CTS rolls back the register mapping to the last committed branch. Unlike other proposals with full atomic commit, CTS only has to re-execute all the instructions in the iteration after a load alias misprediction but not after a branch misprediction. On a load alias misprediction, ReLaSch has to start re-executing only from the load and doesn't

re-execute any instruction after a branch misprediction, just starts fetching from the target PC.

10.1.6 Comparing performance

Regarding the results of these proposals, only the in-order rePLay has an in-order issue-logic and compare their performance with a conventional out-of-order processor. Although it doesn't outperform the out-of-order IPC in any benchmark, it is close in some case. CTS also has an in-order issue logic and it yields 1.16 speed-up over an in-order conventional processor. This is less than what is achieved by ReLaSch and it is not compared with an out-of-order processor. DIF compares only to a simplified version of itself. The rest of proposals don't have an in-order issue logic.

10.1.7 Other caching proposals

There are other caching proposals that are presented here in less detail because we find them less closely related to our work.

The Turboscalar proposal [38] uses two pipelines: the Cold one is a conventional out-of-order superscalar pipeline; the Hot one is much wider (24 instructions per cycle) and shallow. The Hot pipeline read the instructions from a block-based trace cache with the instructions already decoded and renamed. Thus, it only has to access the register file and use a sparse crossbar to dispatch the instructions to the reservation stations, where instructions are executed out-of-order. The register file is organized as stacks of physical registers for each logical register. The position of the instructions in the blocks is arranged according to their type in order to match the corresponding dispatch unit, which allows to simplify the crossbar. The trace cache is filled with the instructions executed in the Cold pipeline.

MPS [39] places a scheduler between the main memory and the instruction cache. The instructions are then processed in order. Since the scheduler cannot use information gathered at execution time, the performance results are not close to those of a conventional out-of-order processor. MPS can speculatively schedule instructions across branches. It stores the original program order with the instructions, which is used to insert the instructions in a ROB. However, its lack of information from execution time forces MPS to not predict the outcome of indirect branches. The presence of such a branch ends a schedule. Schedules are also closed with backward branches to prevent loop unrolling.

The Hot Spots mechanism [40, 41] detects the regions of code that are most frequently executed. Then it creates sets of traces that are optimized to improve the performance of the Fetch mechanism: it relayouts the basic blocks to increase the frequency of the fall-through case, performs automatic in-lining and loop unrolling. The optimized traces are stored in a dedicated set of virtual memory pages called the code cache. The BTB is modified to access the code cache whenever a hot spot is reached.

Transmeta [42] performs binary translation from x86 instructions to a VLIW machine. The translated traces of code are cached and the traces are executed atomically. The Incremental Commit Groups proposal [43] improves a similar processor by allowing partial commit of the traces. Each trace is divided into several commit groups after instruction scheduling. Each commit group commits atomically. At execution time, a trace commits each group sequentially. A commit buffer and a speculative architectural register file is used to keep the architectural state of the trace. A trace predictor and a commit depth predictor are used to choose the next trace to execute and the number of commit groups that will be executed. If the prediction is correct two traces can be executed consecutively without paying any rollback penalty even if the first one is not completely executed. One significant difference with ReLaSch is that in Transmeta and the Incremental Commit Groups, translation is mandatory to execute any instruction, while in ReLaSch the scheduler is used only to improve performance and any instruction can always be executed in the Icache mode.

The Trace Line processor [44] adds renaming information to a trace cache in order to reduce the complexity of the renaming logic. The issue logic, the FUs, the register file and the renaming logic are partitioned into blocks. Bypassing is restricted to instructions executed in the same block. Thus, it can have a large issue window and a large register file with a reduced complexity. Each trace is assigned to an idle block after the map stage. A trace can have up to 16 instructions. The trace cache has register renaming information to bypass the map stage. Also, the instructions are stored already decoded. The proposal includes a dynamic loop detector that enables reusing a trace if it is already present in one of the blocks.

The instruction co-processor [45] is not strictly a caching proposal. It presents a co-processor that can be used to manipulate the instructions of the main processor. The co-processor takes as input the stream of committed instructions. It has its own very simple instruction set and can be used to perform many code transformation tasks. As an example, the implements the fill unit for a trace cache with the co-processor, as well as simple code optimization and data prefetching.

10.2 Non-caching proposals

10.2.1 Loop-based instruction reuse

The following related work present proposals that save energy by reusing the instructions already present in the pipeline whenever it is detected that they form a loop.

The Loop Processor Architecture [46] detects the presence of simple loops, buffers them and feeds the execution pipeline from the buffer instead of the front-end. A loop must have a single control path, but it can include branches that do not change their direction during execution. The main benefit of buffering a loop is the energy savings of switching off the front-end. Furthermore, since the loop buffer does not have to deal with alignments and taken branches, it can provide the instructions at a faster rate than the front-end, which accesses an instruction cache. The instructions are stored already decoded and renamed. The front-end can be switched off, including the register renaming logic. A flag in the rename table indicates whether the logical register is updated by an instruction in the loop or it is loop-independent. Instructions are renamed using virtual tags. Each tag has two parts: the root tag and the iteration tag. The root tag is assigned in a conventional way. The iteration tag is incremented in each iteration if the register is modified inside the loop. The buffering mechanism needs three iterations until it is able to provide the instructions to the back-end stages. The backward branch of the first iteration activates the buffering mechanism, the instructions of the second iteration are buffered and the third iteration is used to complete the detection of data dependences. A buffer of 128 instructions is used in the experiments. The activity of the front-end is reduced in average a 14% in the integer benchmarks and a 45% in the floating point benchmarks.

The following three proposals reuse the instructions already present in the issue queue or the ROB in order to reduce energy consumption in the front-end.

The issue queue reuse proposal [47] detects the loops that fit completely in the issue-queue. When that happens, it starts delivering the instructions from the issue-queue instead of using the front-end. Thus, the front-end can be gated off for power reduction. When the processor is in the reusing mode, the instructions are read from the issue queue, renamed and then inserted in the ROB. The instructions stay in the issue queue after being issued. The issue queue exits the reusing mode when a branch misprediction is detected. Several iterations of the loop can be present at the same time in the queue, thus dynamically unrolling the loop. It can also perform automatic function in-lining if the loop with the in-lined code fits in the issue queue. The branch predictor is gated with the rest of the front-end and the prediction used to fill the issue queue is reused. A small table remembers the loops that do not fit in the issue-queue in order to minimize unnecessary attempts to reuse a loop.

The trace reuse proposal [48] also reuses loop instructions to gate the front-end stages and deliver

instructions in a more power efficient way. In this case, the instructions are delivered from the ROB. Instructions from newer iterations are copied in new entries in the ROB and instructions are removed from the ROB as they commit. The reused instructions are sent to the conventional register renaming logic. The loop detection mechanism works on the basic block level. A FIFO buffer tracks which basic blocks are present at a given moment in the ROB. A CAM access is performed to the buffer for each branch target and reuse mode is activated on a hit. When the processor is reusing the instructions, the branch predictor is accessed once per basic block. This allows capturing complex loop patterns that include control instructions within the loop body.

The CLU proposal [49] also reuses loop instructions from the ROB buffer to reduce power. It is more restrictive than the trace reuse proposal discussed above since it can reuse only loops that are both tight (without control instruction in the loop body) and small (up to 16 instructions). Its distinctive feature is that each instruction in the loop is inserted just once in the ROB, so power savings come not only from front-end gating but by reducing the number of insertion in the ROB. A separated structure tracks the renaming information of all in-flight instances of the instructions, allowing several iterations to share the same ROB entries. Since it only captures tight loops, the branch predictor is not accessed while instructions are being reused.

10.2.2 Simplified issue

The following proposals show alternative ways to simplify the issue logic. They don't attempt to reuse the instructions already present in the pipeline.

Runahead [50] proposes to increase the performance of an in-order processor by pre-executing instructions on a cache miss. We consider that this technique is orthogonal to our proposal and could be implemented in it.

The Flea-flicker two-pass pipelining [51, 52] extends the idea of runahead execution and proposes an VLIW machine with two in-order pipelines. The first pipeline executes instructions greedily and speculatively. It does not stall whenever an instruction has not all its operands ready, typically due to a cache miss. The execution of that instruction is deferred to the second pipeline instead. Its destination register is marked as invalid to defer the execution of the dependent instructions too. The second pipeline executes instruction conservatively and maintains the architectural register file. It merges the results of the first pipeline to avoid executing the same instruction twice. A decoupling queue is used to communicate the two pipelines. The use of the two pipelines allows overlapping several outstanding misses and hide the long latency. This proposal was later implemented in a single in-order pipeline that performs multiple passes to the code [53, 54].

Wakeup prediction [55] simplifies the issue logic by removing the conventional wake-up logic, that tracks when the operands of each instruction are ready, and using a prediction mechanism instead. The Wakeup predictor returns a wakeup time for each decoded instruction. The instructions are then inserted in the self-schedule array, where they stay the predicted wakeup time. Instructions are then selected and sent to read the register file. If the operands are not ready yet, the instruction is replayed and inserted again in the self-schedule array, although the predicted wakeup time is doubled to reduce the number of unnecessary replays. In order to reduce the number of replays an additional allowance time is added to each prediction. The actual latency of the instruction is used to update the predictor, which is expected to stabilize after some time. The wakeup prediction mechanism removes the critical feedback look in the issue logic.

Cyclone [56] also predicts the latency of each instruction and selectively replays the instructions that are scheduled too early. Unlike Wakeup prediction, that uses previous observed latencies of an instruction to predict its latency, Cyclone tracks when are the registers available and schedules each instruction according to the availability of its operands. Another difference between the two proposals is the structure where the instructions wait during the predicted latency. In Cyclone instructions are inserted in the tail of a countdown queue. This queue shifts the instructions to the head one entry per

Name	Issue	I/C	S/O	RN	RF	C	Size	Aim	Reference
DIF	VLIW & scalar	C	S	Y	S	A[a]	6x8	IPC	IO-DIF
rePLay	OoO	C	O	N	C	A	256	#inst	OoO
rePLay-io	IO	C	S&O	Y	C	A	256	#inst & #cycles	OoO
Flywheel	OoO	I	S[b]	Y	S	I	512[c]	Power	OoO
CTS	IO	C	S&O	Y	W	L	400/20k[d]	IPC	IO & CTS wo. optimizer
Parrot	OoO	C	O	N	C	A	64	performance	OoO
ReLaSch	IO	C	S	Y	S	I	256	IPC	IO & OoO

[a] A schedule in DIF can have several exits.
[b] Flywheel schedules using a conventional issue logic.
[c] Maximum schedule size in Flywheel is adaptive.
[d] Schedule size in CTS is measured in static and dynamic number of instructions of the schedule.

Table 10.1: Summary of the main characteristics of the most relevant related work. The legend for the fields is: Issue: which kind of issue logic is used. I/C: Instructions are captured at Issue or Commit. S/O: Schedule or optimize. RN: Whether instructions are renamed by the scheduler (Yes/No). RF: Register file. Can be Common (C), organized in Sets (S) or organized in Windows (W). C: Commit type. Can be Atomic for the whole schedule (A), atomic for each Loop iteration (L) and per-Instruction (I).

cycle. When the instructions have been in the queue for half of the predicted time, they are moved to the main queue, where they spent the rest of the predicted time. The instructions in the main queue are shifted one position per cycle towards the register read and execute stages. In the register read stage, the ready bits of the desired physical registers are checked and the instruction is replayed if the operands are not ready yet. The destination physical register of a replayed instruction is marked as not ready in order to replay any dependent instruction too. Aliased loads and store instructions are tracked in a similar way. An important weakness of this proposal is that the prediction mechanism does not take cache misses into consideration.

Prescheduling [57] uses a Preschedule window formed by lines. These lines are read in order and feed an out-of-order issue logic. It allows reducing the size of the issue queue without degrading IPC. The preschedule logic processes the decoded instructions in execution time. It just takes into account the data dependences and does not deal with renaming or resource assignment because the out-of-order issue logic performs all these tasks. The preschedule logic assumes an L1-hit latency for all the load instructions.

10.3 Summary

Table 10.1 summarizes the main characteristics of ReLaSch and the more relevant proposals discussed above in order to help comparing all them.

Chapter 11

Conclusions and future work

11.1 Conclusions

In this work we have achieved the following main goals:

- To prove that the issue-logic of a conventional out-of-order processor does a lot of redundant work.
- To propose a microarchitecture that reduces the work to be done by the issue-logic by reusing dynamic schedules.
- To propose a simpler execution pipeline than a conventional out-of-order processor. The scheduler has been placed after the Commit stage, which allows using a fully in-order execution pipeline.

Regarding the lessons learned while doing this research, a very important decision has been to place the scheduler after the Commit stage. The scheduler has there a broader vision of the code that allows having better performance in some cases. It takes into account all the instructions in the schedule while a conventional out-of-order issue logic only sees the instructions that are available in the issue queue at a given cycle. Our experiments show that, using a cache of 128 schedules (of 256 instructions each), we achieve almost the same IPC than a conventional out-of-order processor with the INT benchmarks (99% of the average IPC) and 1.01 average speed-up in the FP benchmarks, outperforming the out-of-order processor in 23 out of 40 benchmarks.

Another key element that has allowed to achieve good results is the use of long rgroups, since the scheduler is able to extract more ILP and create better schedules. In the early stages of this work, we tried to develop a processor that reused individual issue-groups. Knowing which physical register to use was an important problem. Using a predictor in the renaming stage made the processor work but the performance results were unacceptable.

It is also important to perform a fine tuning of the schedules. There are many small things done by the scheduler to fine tune the processor that improve performance, even if some of them are not very effective when used alone. For instance, taking into account the issue latency of the different functional units. Another example is considering the commit rate of the scheduled instructions to update in which issue-group is released a given resource that is reused by a younger instruction.

Another significant step in the design of the processor was to remove the restriction that the instructions of a rgroup had to be alone in the execution pipeline, without any in-flight instruction from an older rgroup or from the Icache. This restriction implied paying a very heavy penalty because

it created bubbles in the pipeline between two consecutive schedules as well as before and after changing to the Rcache mode.

The ability to adapt to the variable latency of memory accesses is one of the most important strengths of a conventional out-of-order processor. We cannot have such ability but our latency predictor is generally able to capture the behavior of the memory accesses. Our initial attempts with simpler prediction strategies failed to generate proper schedules.

Detection of aliased memory instructions at scheduling time and how are they managed at execution time is also an important part of ReLaSch. Our excellent results in most of the benchmarks lead us to implement the Store Sets in the reference out-of-order processor in order to make a fair comparison. We also enhanced the reference out-of-order processor with better multi-target indirect branch prediction, because of our much better results when compared with the original reference processor.

The experiments have shown that using history bits to identify the schedules is not effective. Besides, the experiments have also shown that we can further simplify the processor without having a significant performance degradation. For instance, we can use a smaller instruction cache or a much simpler branch predictor. After simplifying the processor in this way, we have presented the results of the new default configuration, that achieves very similar results in performance.

Although the results presented here are in general compared with an out-of-order processor, we can also see ReLaSch as an enhancement over conventional in-order processors. In this case, it is a very effective way to improve the performance of an in-order core. Compared with a conventional in-order processor, we have 1.51 INT and 1.64 FP speed-up. Even with a smaller cache the improvement is significant.

As a summary of the performance achieved by the processor proposed here, figures 11.1 and 11.2 show the speed-up in IPC achieved with the new default ReLaSch when compared to the reference OoO and IO processors. All evaluated benchmarks are included, ordered from lower to higher speed-up.

11.2 Future research directions

The work presented here can be used as a starting point for new research. The following is not an exhaustive list but just some ideas for research directions based on this work, grouped into three categories:

- Cycle time: To create a detailed model to study to cycle time that ReLaSch can achieve. In this work we have assumed that ReLaSch and OoO work at the same frequency. But due to its in-order issue logic and execution pipeline, it seems reasonable to assume that it can be clocked at higher frequencies with the corresponding performance benefits.

- Energy/power aware ReLaSch.

  - To create a detailed model to study the power and energy consumption required by the ReLaSch processor; the study should evaluate the impact of the most relevant parameters such as the Rcache size and the rgroup size.

  - To study the performance and power trade-offs. Several ways or sets of the Rcache as well as the scheduler can be switched off to trade off some performance for energy savings. The gains and costs must be measured to define a policy for switching off these elements.

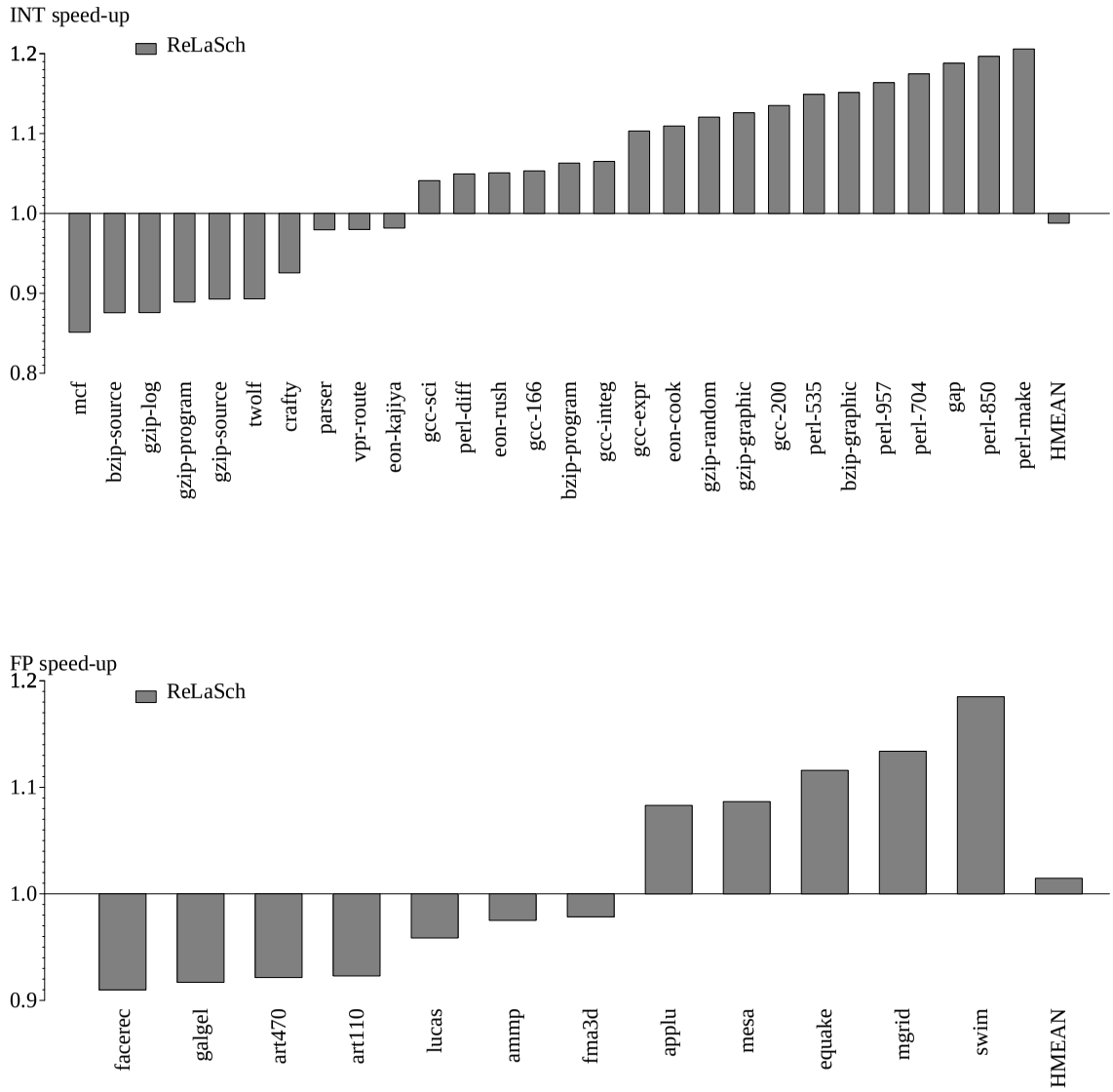
  - To study the impact of novel power and energy saving techniques. For example, the broad vision of the scheduler enables it to manage power saving techniques such as clock and voltage gating in a more efficient way. The scheduler knows at scheduling time whether a given functional unit is not going to be used for many cycles. Thus, it can insert such information in the rgroup to switch on and off these resources at the most appropriate

time. Furthermore, the scheduler can place the instructions later than the time when the operands are ready in order to maximize power savings.

  - Minimize voltage droops. When a sudden change in the activity of the processor creates a current peak it can produce a voltage droop and eventually require re-execution if the voltage margin is small. Such sudden change can be related with a long latency cache miss. When it is served after cycles of stall, the issue logic finds many ready instructions and the issue rate is very high. It has been shown that dynamic code transformation is useful to solve this problem [58]. The broader vision of the code that the scheduler has in ReLaSch can be used to schedule the instructions in the rgroups in a way that prevents the voltage droops.

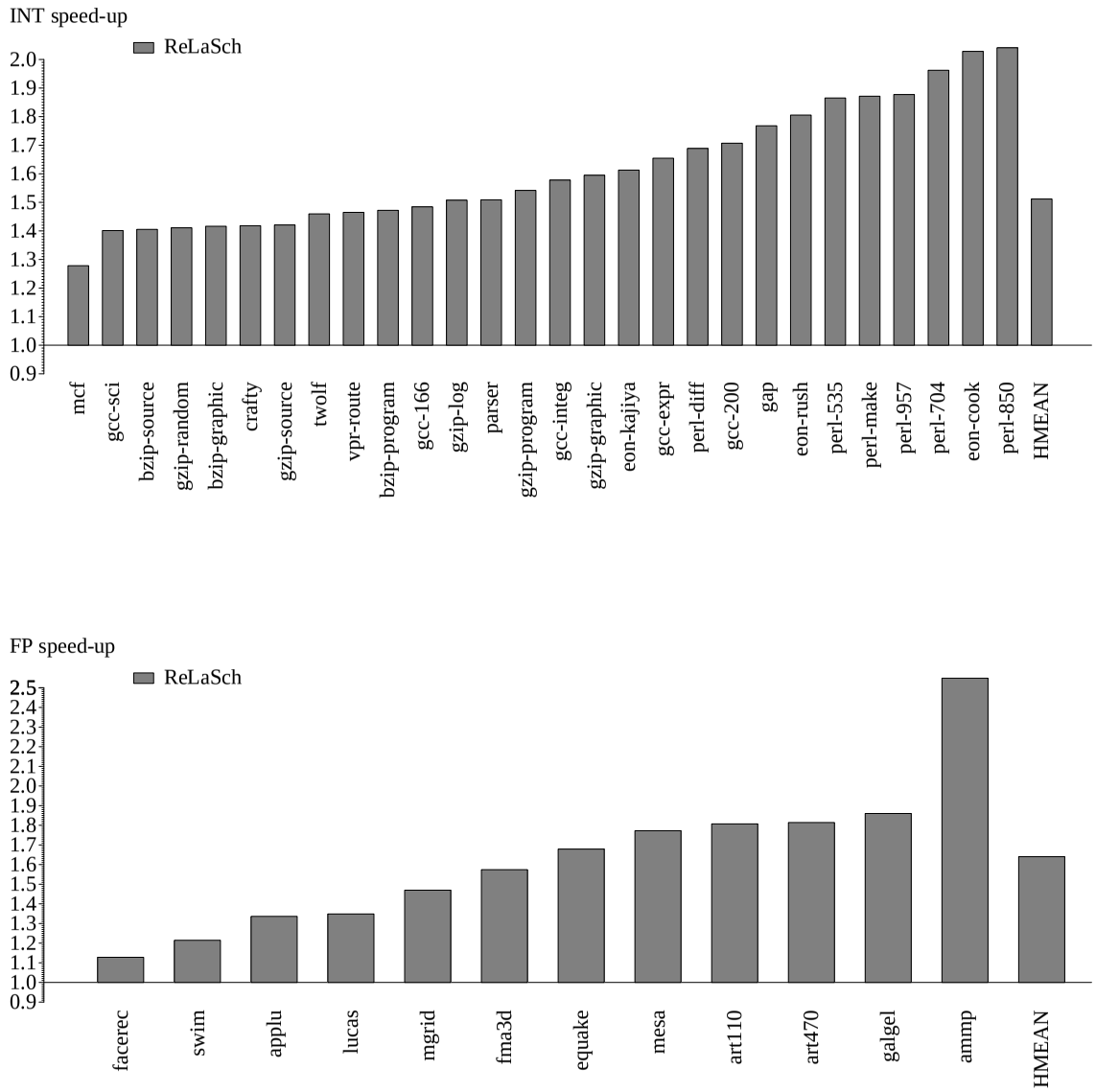
- IPC improvements.

  - To improve the branch misprediction rate. When compared to an aggressive conventional out-of-order processor, branch prediction is one of the weaknesses of ReLaSch. Though for many benchmarks our mechanism works well, some other benchmarks need to create a very high number of schedules, corresponding to the many paths that are executed. That puts a lot of pressure to the scheduler and the Rcache. It seems possible to improve the misprediction rate. For instance, using some kind of filtering technique in these cases to create only the schedules that cover very frequently executed paths.
  - To study the impact of dynamic optimizations. If frequently executed paths are known, dynamic code optimizations can be applied to rgroups that are known to execute always to completion if this is combined with atomic commit for these rgroups. There is previous related work [11, 9, 34] that can be helpful to choose which optimizations should be applied.
  - To improve the distribution of the accesses to the logical registers. In our experiments, we have used Alpha binaries just as they were generated by the compiler. Since our register file has a fixed set of physical registers for each logical register, it could be beneficial to change the binaries or force the compiler to distribute the writes across all the logical registers.
  - To improve the schedules whenever it is possible. For instance, the scheduler can detect the backward branch that ends a loop and finish an rgroup earlier. In that way, it would be easier to chain the execution of the rgroups that correspond to that loop. Similarly, the scheduler can be extended to include a mechanism similar to the store sets, to detect the cases when there isn't a single pair of aliased load-store, but a more complex set of aliased instructions.



	ReLaSch
HMEAN INT	0.98
HMEAN FP	1.02
HMEAN ALL	1.00

Figure 11.1: Speed-up of the ReLaSch processor over the reference OoO processor.



	ReLaSch
HMEAN INT	1.51
HMEAN FP	1.64
HMEAN ALL	1.55

Figure 11.2: Speed-up of the ReLaSch processor over the reference IO processor.

Appendix A

Common register file

The default ReLaSch processor presented in this work uses a register file that has a set of physical registers for each logical register. Such type of register file can limit the performance that can be achieved, because there can be unused physical registers while the processor stalls waiting for a free physical register of a given logical register. This appendix presents the changes to the ReLaSch processor that are needed to use a common register file. Such a register file offers a pool of physical registers shared by all the logical registers. It also presents experimental results that show that the average performance achieved is very similar to that achieved by the default ReLaSch. Since the common register file requires more complex logic, we have decided to keep the partitioned register file as the default for ReLaSch.

A.1 Description of the register file

The ReLaSch processor uses two separated register files, one for the integer registers and another for the floating point register. The rename structures that are presented here must be duplicated, one for each register file. Besides, the ReLaSch processor implicitly assumes the in-order processing of the instructions in the issue logic for the proper use of the registers. The use of separated integer and floating point issue buffers can change the relative order of a pair of integer and floating point instructions. To ensure the proper order in the accesses to the registers, the instructions that access both register files are inserted in both issue buffers. Float-store and float-to-int instructions already are inserted in both buffers in the baseline ReLaSch. In the ReLaSch version with a common register file, float-load and int-to-float are inserted in both buffers too. The motivation is explained in section A.2.2.

The logical registers R31 (integer) and F31 (floating point) always return the value 0, and a write to one of these registers has no effect. When the destination logical register of an instruction is either R31 or F31, it is not renamed to any physical register.

The register file is formed by a single pool of physical registers shared by all the logical registers. Each physical register has an associated `valid` bit that indicates whether a value has been written and is available for the instructions that want to read it. Each physical register has an identifier, that goes from 0 to *regfile_size-1*, where *regfile_size* is the number of physical registers in the register file.

The identifiers are always assigned in the same order. In section 6.3 we have seen that the Rmap logic needs to add an offset to the identifiers assigned by the scheduler. The identifiers of the common register file are assigned sequentially, always in the same order, because it enables using a single offset to update the identifiers at execution time. The `free_head` register is used in the Icache mode to do it. It points to the first free physical register, which is assigned as destination register when the next

instruction is mapped. The `free_head` register is incremented after each mapped instruction that has a destination physical register. The Rcreate logic assigns the identifiers also in the same sequential order, always beginning each rgroup with the identifier 0.

Each physical register has a `busy` bit to indicate whether a given physical register is available to be used as destination. Section A.2 shows how it is not needed for some sizes of the register file.

The architectural value of a given logical register must be safely stored until the next instruction that writes the same logical register commits. In the normal version of ReLaSch, where each logical register has it own private set of physical registers, the architectural value can be stored in one of the physical registers of the set indefinitely: this is possible because the physical register that stores the architectural value is not required to be assigned again until the logical register is not used again as destination. And when this happens, the architectural value will be stored in a different physical register after a number of cycles and the current architectural physical register is freed afterwards for sure.

On the contrary, with a common register file, where the physical registers are shared by all the logical registers and the physical registers are assigned always in the same order, a physical register *phy* that is used to rename the destination register of an instruction cannot hold indefinitely the architectural value of the destination logical register after the instruction commits. If none of the next instructions uses the same logical register as destination, the `free_head` register ends up pointing again to the physical register *phy*. In this case, the processor would either enter a deadlock or lose the architectural value by reusing the physical register *phy*.

Therefore, the architectural values are stored separately in a dedicated part of the register file. When an instruction commits, the value of the destination physical register is copied in the architectural register of its logical register. The architectural registers do not have an identifier that can be assigned as destination by the `free_head` register.

Register renaming in the Icache mode

The `rename` table is used in the Icache mode to know to which physical register must be mapped the source registers of a given instruction. The Icache mode uses a conventional renaming logic. It has an entry for each logical register, the `phy` field is used to store the identifier of the physical register, and the `arch` bit to indicate if the value is stored in the architectural register. When an instruction is mapped, the renaming logic reads the value in the `rename` table for each one of its source logical registers. It also updates the entry in the `rename` table of its destination logical register clearing the `arch` bit and copying the value of the `free_head` register into the `phy` field. The `free_head` register is incremented afterwards.

Figure A.1 shows an example of renaming in the Icache mode. The source logical register 2 is in the physical register 15, while the source logical register 3 is in the architectural register, since its `arch` bit is set. The destination logical register 1 is renamed to the physical register 17 and its `arch` bit is cleared. The `free_head` register is incremented afterwards.

The Issue stage knows that a source register is available when it is in the architectural register (the `arch` bit is set) or if the `valid` bit of the physical register is set. Regardless of the mode, when the instruction is finally issued, it reads the desired source register, either the physical register indicated in the `phy` field or the corresponding architectural register.

The `rename` table is also updated by the Commit stage. For each instruction, it accesses the `rename` table with the identifier of the destination logical register and the content of the `phy` field is compared with the identifier of the destination physical register used by the instruction. If they match, the `arch` bit is set. Otherwise, the `rename` table is left unchanged since there is a younger instruction that writes the same destination logical register. Figure A.2 shows an example of how the Commit logic accesses the `rename` table, where the destination physical register (17) of the committed instruction doesn't match with the current renamed physical register (22). It means that an in-flight

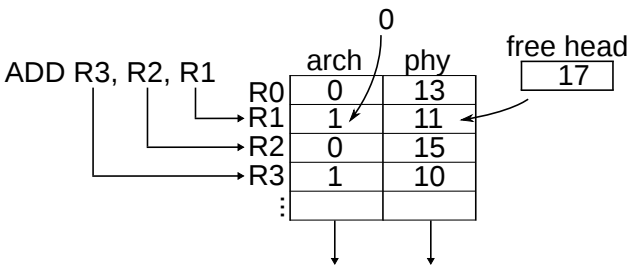


Figure A.1: In the Icache mode, the `rename` table is used to rename the registers. The `free_head` register indicates which is the next free physical register.

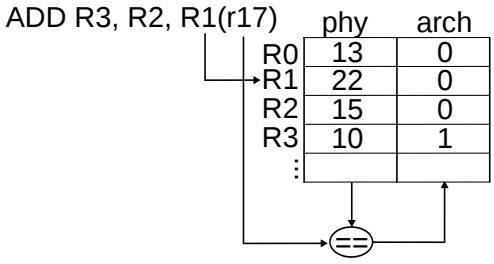


Figure A.2: An example of how the `rename` table is accessed with the destination logical register of each committed instruction.

younger instruction writes the same logical register. Therefore, the table is not updated in this case and the `arch` bit remains unchanged.

A.2 WAR and WAW hazards

Once a given instruction i has committed, all the instructions that depend on i that are renamed in the Map stage afterwards will read the architectural register, because they will find the `arch` bit set. The physical register *phy* that stored the speculative value will not be accessed by these instructions. However, an instruction j that depends on i that has been mapped before i commits is renamed to access *phy* because it accessed the `rename` table when the `arch` bit was still zero. In ReLaSch the source registers are read after the instruction has been issued, so the instruction will read *phy* even if the data is already available in the architectural register. If the instruction j stalls in the Issue stage (e.g. because it also depends on a long latency memory instruction), it may read the physical register *phy* many cycles after i commits and the content of *phy* has been copied to the corresponding architectural register.

This implies that a physical register cannot be reused immediately after commit when there is any pending read by a dependent instruction. The Icache mode ensures that just by issuing the instructions in-order. For the instructions executed in the Rcache mode, the Rcreate logic must track all the reads to each physical register before assigning the register again. The instruction that reuses a given physical register must be scheduled after the last read in schedule order of the previous use of the register.

Different approaches can be used depending on the relative size of the register file when compared to the size of the ROB. The sections below describe the differences between three cases: a) if there are at least twice as many physical registers as identifiers in the ROB; b) if there is at least as many physical registers as identifiers in the ROB and c; if there are less physical registers than identifiers

in the ROB. In the text below, *ROB_size* indicates the number of identifiers in the ROB and *RF_size* indicates the number of physical registers.

A.2.1 Twice as many physical registers as identifiers in the ROB

If $RF\_size \geq 2 * ROB\_size - 1$ then actually no special logic is needed to ensure correctness. In this case, a physical register can be assigned without any restriction because it is guaranteed that any older dependent instruction has read the old value in the physical register when it is overwritten.

The example in figure A.3 shows the worst case. To simplify the explanation, we assume that the code is executed in a version of the OoO processor that assigns the physical registers sequentially and has a conventional out of order issue logic. The instruction A uses the physical register *r0* as destination. The Map stage updates the entry for *R3* in the `rename` table to *r0*. The instruction A has a very high latency, so the Map stage completely fills the ROB, up to the instruction C, before A commits. All the dependent instructions mapped before then have read (or will read) the physical register *r0*. The instruction D uses the same identifier in the ROB as A, so D will stall until A commits and frees its entry in the ROB. When A commits, the `arch` bit of the logical register R3 is set, since no younger instruction has updated the entry for *R3* in the `rename` table. Thus, when the instruction D enters the processor, its source registers are renamed to the architectural register of *R3*.

The instruction B has a very high latency too, so the instruction C cannot be issued until many cycles later than it has mapped, since one of its source operands is not available. The instructions that follow it can be mapped and issued safely, since none of them is renamed to write *r0*. The instruction E cannot be mapped until B has committed and its ROB entry is freed. After the instruction B has committed, the instruction C finally has all its source operands available. However, the instruction C can actually stall for some additional cycles, if its functional unit is already in use by another instruction. In this case, the instruction E can advance it and write the result before the instruction C reads its source registers, so the instruction E shouldn't use *r0* as destination. The instruction F cannot be mapped before the instruction C commits so it is the first instruction that is guaranteed that cannot write *r0* before C has read the register. Therefore, it is safe to rename its destination register to *r0*.

With a register file that has at least $2 * ROB\_size - 1$ physical registers, each time a physical register is assigned as destination, all younger instructions that may read it have already committed. When such a register file is used, the Rcreate logic of the ReLaSch processor doesn't need to track when a physical register is read for the last time at scheduling time. There is still the limitation that the instruction that overwrites the physical register (F in the example) must be scheduled after the last read to it (C). However, this is already achieved through the `ROB_safe_pos` table, that is used to properly schedule instructions that reuse an identifier in the ROB.

A.2.2 As many physical registers as identifiers in the ROB

The ILP extracted by the dynamic scheduler in the OoO processor or by the Rcreate logic in the ReLaSch processor is limited by the size of the ROB. An instruction cannot be scheduled beyond *ROB_size* positions. When there are twice as many physical registers as identifiers in the ROB, many times most of the physical registers would remain unused. The example of A.3, in which the ROB is filled up and it takes many cycles since a physical register is assigned until it is read for the last time, is a corner case which is unlikely to happen often. The processor must be able to manage this situation correctly but we shouldn't optimize for this case. Having such a large register file is a waste of resources.

If a smaller register file is used, the processor must check that there isn't any pending read to a physical register before assigning it as destination. In general, the Rcreate logic cannot schedule an instruction that uses a given resource before it is safe to reuse it. In the ReLaSch processor with at

```
A: LD  0(R1),  R3 ; renames R3 to r0, L2 cache miss -> very high latency, ROB_id=0
...
B: LD  0(R2),  R4 ; L2 cache miss -> very high latency, ROB_id=78
C: DIV R3, R4, R5 ; reads R3 from r0, ROB_id=79
D: ADD R3, R3, R6 ; reads R3 from architectural register, ROB_id=0, writes r80
...
E: ADD R7, R8, R9 ; ROB_id=78
F: ADD R10, R11, R12 ; R12 in r0, ROB_id=79,
```

Figure A.3: Example of code that produces the worst case at execution time for a register file that is twice larger than the ROB. The right-most operand indicates the destination register. Only instruction A writes R3.

least *ROB_size* physical registers, the `safe_pos` of a physical register is at least the issue-group where it is read for the last time. This condition is already granted by the use of the `ROB_safe_pos` table.

No special logic is needed in the execution pipeline of the ReLaSch processor, neither in the Icache mode or the Rcache mode, thanks to the in-order processing of the instructions. If an instruction that reads a given physical register stalls at the Issue stage waiting for another source register or a functional unit, the instruction that reuses the same register as destination will also stall at the Issue stage, the Map stage or the Rmap logic. So the register cannot be overwritten before the previous value is read.

Since the issue logic uses independent integer and floating point buffers, the relative order of the instructions between the two buffers can be changed, even if the issue logic of both queues is in-order. In general the semantics of the program are preserved, but there are several instructions that access both the floating point and the integer register files. To ensure that they are ordered correctly with respect both the integer and floating point instructions, all the instructions that use both the integer and the floating point register files are inserted in both issue buffers in order to enforce a correct ordering of the accesses to the register files. These instructions are the floating point load and store and the conversion instructions that transfer a value from one register file to the other.

Another problem to solve is that it is possible that an instruction i that reads a physical register *phy* and an instruction j that will overwrite *phy* afterwards are simultaneously present in the issue buffer. They are processed in-order, so the accesses to the register are correctly ordered. However, the instruction j clears the `valid` bit of *phy* in order to prevent the dependent instructions to issue and access *phy* before it stores the new content. This bit is typically cleared in the Map stage but doing so now would also prevent the instruction i to issue, actually causing a deadlock. Thus, the `valid` bit of the physical register *phy* is cleared when the instruction j is issued.

Now there is another problem, since also a younger instruction that depends on instruction j can be present in the issue buffer at the same time as j. In the example of figure A.4, the three instructions can be present at the same time in the issue buffer and processed simultaneously by the check logic of the Issue stage. As explained above, B does not clear the `valid` bit of *r1* until it is issued to ensure that A can be issued and it reads the old value of *r1*. But then C sees that the `valid` bit of *r1* is set, so it could be issued that same cycle, though it actually must read the new value of *r1*, which is not available yet.

To solve this problem, the Issue stage could check if there is any instruction being issued that same cycle that overwrites a source register of a younger instruction. Whenever that happens, the younger instruction is not issued. If the issue logic can process up to four integer instructions, the check logic of the fourth instruction needs six comparators (three previous destination registers vs. two source registers). The check logic of the first instruction does not have any comparator. The

```
A: ADD R1, R2, R3 ; reads R1 from r1
B: ADD R4, R5, R6 ; renames R6 to r1
C: ADD R6, R7, R8 ; reads R6 from r1
```

Figure A.4: Example of code where both instructions that depend on the old value of a physical register and instructions that depend on the new value can be at the same time in the issue buffer.

second instruction needs two comparators (one for each source register) and the third instruction uses four comparators.

Actually, this is only a problem in the Icache mode, since instructions A, B and C would be scheduled in different issue-groups by the Rcreate logic. Since the Issue stage of the ReLaSch processor respects the boundaries of the issue-groups, the instruction C would not be issued at the same cycle as B even if the `valid` bit of *r1* is set. If the comparators are in the critical path, it is probably not cost-effective to use them, since they are needed only for the instructions executed in the Icache mode. In this case, an alternative is to create issue-groups from the instructions executed in the Icache mode, similarly to section 7.2.5. The Map stage would set the `new-issue-group` bit of the entry in the issue buffer of the instruction C. Thus the instruction would stall the cycle when B is issued, just like in the Rcache mode. Another alternative is to reduce the issue width when instructions are executed in the Icache mode. Since low ILP is extracted in this mode anyway and it is expected to be used less frequently than the Rcache mode, the impact in the IPC should be small.

A.2.3 Less physical registers than identifiers in the ROB

The identifiers in the ROB entries are freed in-order in the Commit stage and the Map stage and the Rmap logic stall the instruction that reuses them. Thus, no special logic is needed to detect if an instruction wants to overwrite the destination physical register of an uncommitted instruction if the number of physical registers is *ROB_size* or more like in the sections above.

This is not the case when there are less physical registers than identifiers in the ROB. If only the identifiers in the ROB are taken into account when mapping the instruction, two in-flight instructions may want to use the same physical register as destination. Therefore, each physical register has an associated `busy` bit that indicates whether the value has been committed or it is still speculative. It is set when it is assigned as destination register by the Map stage or the Rmap logic and it is cleared in the Commit stage. A given physical register cannot be assigned as destination as long as its `busy` is set, The instruction stalls until the value is committed and the `busy` bit is cleared.

A.3 The Rfront-end and the Rcreate logic with an empty ROB

To simplify the description of how are instructions scheduled and the rgroups executed, this section presents a simplified version of the logic that requires that the ROB is empty before processing the first instruction of the rgroup. The next section covers the general case.

A.3.1 The Rcreate logic

The Rcreate logic renames the source and destination registers of each instruction. In general, a source register can be in a physical register or in the architectural register. At execution time, the ROB will be empty when the first instruction of the rgroup is processed. Thus, its source operands will be stored in the architectural register. This will happen for all live-in values of the rgroup. When

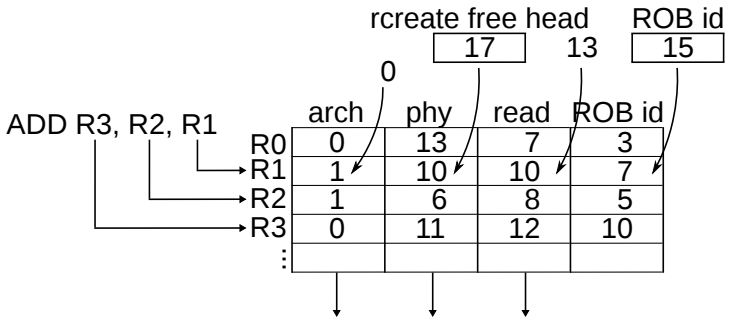


Figure A.5: An example of instruction renamed by the Rcreate logic using the `reg_info` table.

a source register is not a live-in value, its value will be stored either on the architectural register of in a renamed physical register, depending on whether the instruction that produces the value has committed or not. The Rcreate logic must detect which registers are written inside each rgroup, remember in which issue-group they commit and rename dependent instructions accordingly.

The `reg_info` table is used to rename the instructions in the scheduler. It is indexed with the identifier of the logical register. The `phy` field indicates which is the last physical register mapped to the corresponding logical register, the `arch` bit indicates whether the data is stored in the architectural or the physical register and the `read` field indicates the earliest issue-group in which a dependent instruction should be scheduled. As usual, the Rcreate logic accesses the `read` field of the source registers of each instruction and schedules it in the corresponding issue-group. The renamed source registers are stored with the instruction, copied from the `arch` bit and the `phy` field of the `reg_info` table.

Figure A.5 shows an example. The source logical register R3 is renamed to the physical register 11, which is available in the issue-group 12. The source logical register R2 is renamed to the architectural register, since its `arch` bit is set. This register is available in the issue-group 8. The instruction is scheduled in the issue-group 12, the maximum of 8 and 12.

To rename the destination registers, the Rcreate logic uses the `rcreate_free_head` register to track which physical register should be assigned next. The destination register of the current instruction is used to index the `reg_info` table. The content of `rcreate_free_head` is copied into the `phy` field of the indexed entry and the `arch` bit is cleared. As usual, the `read` field is updated with the proper issue-group. Since physical registers are assigned sequentially `rcreate_free_head` is incremented, modulo *regfile_size*.

In the example of figure A.5, the destination logical register R1 is renamed to the physical register 17. Thus, the `phy` field of the register R1 is updated to 17, the `arch` bit is cleared and the `read` field is updated to 13, assuming one cycle of latency for the ADD instruction.

The `arch` bit in the `reg_info` table of a logical register should be set as soon as possible in order to free the physical register and be able to reuse it. However, if it is set too early, it could happen at execution time that a dependent instruction reads the architectural register before it contains the committed value. Therefore, the `arch` bit must be set only when it is sure that at execution time the producer instruction will have committed when the instructions currently being scheduled are read their registers. That the producer has committed is guaranteed in two cases: a) after *ROB_size* scheduled instructions; and b) when the same physical register is assigned again. Which situation happens first depends on the relative size of the ROB and the register file, as well as the actual number of instructions that write a register.

In order to detect the first case, each entry in the `reg_info` table has an additional field (`ROB_id`). When an entry in the table is updated because a destination register has been mapped, the content

of the ROB_id register is copied to that field. In the example of figure A.5, the ROB_id field of the logical register R1 is updated to 15. Before scheduling an instruction, the Rcreate logic performs a CAM-access to the reg_info table with the content of the ROB_id register. The arch bit of any matching ROB_id field is set. Thus, the source registers of younger dependent instructions are renamed to the architectural register.

The second case doesn't require using any additional field because the reg_info table already tracks the usage of all physical registers. Thus, when a destination register is renamed, the Rcreate logic performs a CAM-access to the table with the content of the rcreate_free_head register. If there is any entry with a matching phy field, its arch bit is set.

An alternative to the CAM-access to the reg_info table is to use a separated table with one entry per physical register. Each entry stores the logical register, if any, to which is renamed the corresponding physical register. In this case, the table will be accessed with the content of the rcreate_free_head register. The value read there is used to index the reg_info table and set the corresponding arch bit. The table must be updated when the logical register is renamed. A similar table stores the destination logical register that corresponds to each identifier in the ROB, indexed with the ROB_id register. The ROB_id fields in the reg_info table are not needed in this case.

Reusing a physical register

The Rcreate logic must create a safe and deadlock-free schedule when reusing a given physical register *phy* as destination. Let us consider the previous (i) and the current (j) instructions that write *phy*. Three aspects must be taken into account to have a correct schedule:

a. The instruction i must update the register before j and i must be able to commit. In the example of figure A.6, the instruction C must be scheduled after the instruction B, but also after A. If the instructions are scheduled in the order B, C, A, the processor would enter into a deadlock at execution time, since the instruction C cannot be sent to issue until the instruction B commits and clears the busy bit of r1 and C cannot commit until the instruction A commits, which is blocked waiting that the instruction C is issued.

   Therefore, the instruction C must be scheduled in the issue-group indicated by the value of the safe_pos register at the time the instruction B was scheduled. If the number of physical registers is at least ROB_size, the ROB_safe_pos table already grants a correct ordering of the instructions. Otherwise, the phy_safe_pos table is used to store the safe issue-group for each physical register. For each scheduled instruction, the table is indexed with the identifier of its destination physical register, that is as indicated by the rcreate_free_head register. The value read from the phy_safe_pos table is used to schedule the instruction. Once it has been scheduled, the accessed entry in the table is updated with the content of the safe_pos register.

   Figure A.9 shows an example. The physical register 2 is assigned as destination so the instruction must be scheduled at least in the issue-group 5. According to the contents of the safe_pos register, the entry for physical register 2 is updated to 10. Thus, the next instruction that uses the physical register 2 as destination will be scheduled at least in the issue-group 10.

b. An instruction that depend on i and is younger than j reads the value produced by i through the physical register. In the example of figure A.7, the instruction C must be scheduled after the instruction B. If there are at least twice as many physical registers as *ROB_size*, the ROB_safe_pos table already enforces that the instruction that the instruction j is scheduled after the last instruction that depends on i. Otherwise, the phy_last_read table stores the issue-group with a higher identifier that contains an instruction that reads each physical register. The phy_last_read table is indexed with the source physical registers of every scheduled instruction. If the current value is less or equal than the identifier of the issue-group *ig* in which

```
A: ADD R0, R1, R2 ; R2 is renamed to physical register r0
B: ADD R3, R4, R5 ; R5 is renamed to physical register r1
...
C: ADD R6, R7, R8 ; R8 is renamed to physical register r1
```

Figure A.6: Example of code where a physical register is reused. C must be scheduled after A and B.

```
A: ADD R0, R1, R2 ; R2 is renamed to physical register r0
...
B: ADD R2, R3, R4 ; Reads R2 from physical register r0
...
C: ADD R5, R6, R7 ; R7 is renamed to physical register r0
```

Figure A.7: Example of code where a physical register is reused. Instruction B must read the value of r0 produced by instruction A.

the instruction has been scheduled, the entry is updated to *ig+1*. In the example of figure A.10, the instruction uses the physical register 2 as source and is scheduled in the issue-group 12. The entry in the `phy_last_read` table for the physical register 2 is updated to 13 since the current content is less than 12. Whenever a source register is renamed to use the architectural register, the `phy_last_read` table is not updated. The `phy_last_read` table is indexed with the destination physical register and the content is used to schedule the current instruction.

c. A instruction dependent on the instruction i and younger than j reads the value produced by i through the corresponding architectural register. In the example of figure A.8, the instruction C reads the architectural register. Therefore, it must be scheduled where it is sure that the instruction A has committed, which happens to be at least in the same issue-group of the instruction B. Otherwise, the instruction C could read the architectural value of the register R2 before the instruction A updates it.

   To enforce that, the `reg_info` table is extended with a `rewritten` field. The Rcreate logic already performs a CAM-access to the table with the content of the `rcreate_free_head` register to set the `arch` bit if there is any matching entry in order to properly rename the dependent instructions. Additionally, now the `rewritten` field is updated to indicate in which issue-group the current instruction has been scheduled. Now, both the `read` and `rewritten` fields of the source registers are used to schedule an instruction.

   Figure A.11 shows an example of how this field is updated. The entry of the logical register 1 matches the physical register 17, so its `arch` bit is set. Besides, the `rewritten` field of logical register 1 is updated to 12.

```
A: ADD R0, R1, R2 ; R2 assigned in physical register r0
...
B: ADD R3, R4, R5 ; R5 assigned in physical register r0
...
C: ADD R2, R6, R7 ; Reads R2 from architectural register for R2
```

Figure A.8: Example of code where a physical register is reused. Instruction C must read the value of R2 from the architectural register.

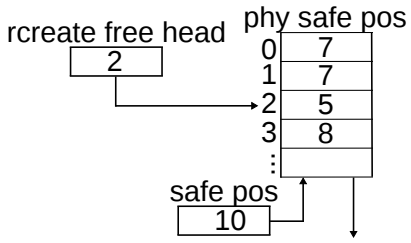


Figure A.9: The Rcreate logic uses the `phy_safe_pos` table to indicate in which issue-group it is safe to use a physical register as destination.

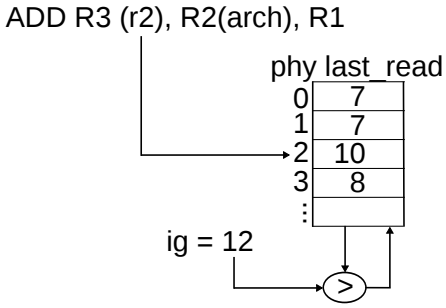


Figure A.10: The Rcreate logic uses the `phy_last_read` table to indicate in which issue-group is scheduled the most recent read to a physical register.

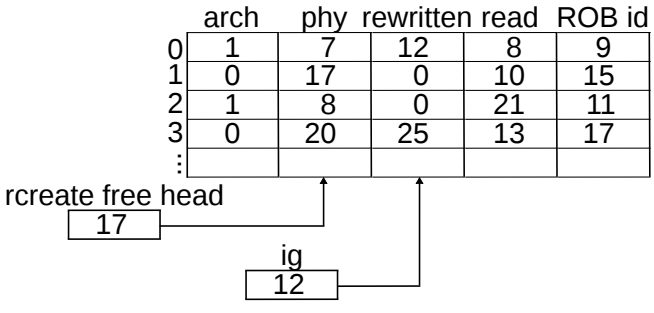


Figure A.11: The `reg_info` table is updated to use the architectural register when the instruction that writes a logical register commits. Incoming dependent instructions must be scheduled after the instruction that rewrites the physical register.

The text above assumed separated `phy_safe_pos` and `phy_last_read` tables as well as separated `read` and `rewritten` fields in the `reg_info` table to simplify the explanation. However, both `phy_safe_pos` and `phy_last_read` tables can be implemented as just one table with two fields, that is accessed with the physical register identifiers. Also, both the `read` and `rewritten` fields can be unified in a single field that indicates in which issue-group should be issued any instruction that wants to read the register. The field is updated whenever the new value is greater than the current one.

A.3.2 The Rmap logic

The Rmap logic is quite simple, since it is guaranteed that the ROB is empty before inserting the first instruction from the rgroup. Thus, the identifiers assigned by the Rcreate logic can be directly used.

A destination register is available if its `busy` bit is cleared. If the number of physical registers is at least equal to the *ROB_size*, it is not necessary to have `busy` bits, since the desired physical register is always available.

The next rgroup processed by the Rmap logic or the instructions executed in the Icache mode must wait until the current rgroup has been completely committed before entering the ROB. All the values will be stored in the architectural registers then. Therefore, there is no need to update the `rename` table in the Rcreate mode.

A.4 The Rfront-end and the Rcreate logic with a non-empty ROB

In the section above, the Rmap logic wastes many cycles waiting until the ROB is empty to start processing a new rgroup. This restriction degrades severely the performance that the processor can achieve. This section presents the changes needed to start an rgroup when the ROB is not empty.

A.4.1 The Rcreate logic

In the restricted version of the Rcreate logic presented in the section above, the scheduler assumes that all live-in values are stored in the architectural register when an rgroup starts execution. In the general version of the Rcreate logic presented in this section, that assumption is not true anymore. There can be older in-flight instructions that produce those live-in values. Usually it is not known until execution time whether a live-in value must be read from the architectural register or a physical register. To indicate which are the live-in values, the scheduler sets a flag when renaming a source logical register that has not been used yet as destination in the rgroup. The Rmap logic is responsible for renaming these registers properly.

Therefore, the `reg_info` table has an additional bit (`prev`) that indicates whether a logical register has been used as destination by any instruction in the rgroup or if at execution time it will contain a live-in value. Initially, the `prev` bit of all the entries in the `reg_info` table is set. The `prev` bit of the source registers is copied along with the `arch` bit and the `phy` field when the Rcreate logic renames an instruction. The Rcreate logic clears the `prev` bit of the destination logical register of each scheduled instruction.

After *ROB_size* scheduled instructions, it is sure that all the values written by the instructions older than the rgroup are committed and can be read from the architectural register. However, since there is no benefit from doing it and requires having some additional logic, the `prev` bits are not cleared after *ROB_size* instructions.

A.4.2 The Rmap logic

The Rmap logic has two additional tasks now: a) complete the renaming performed by the Rcreate logic in a way that allows to start processing an rgroup when the ROB is not empty; and b) update the state of the renaming structures so that Rmap can process the next rgroup (or the Map stage can process the instructions in the Icache mode) without waiting for the whole current rgroup to commit.

Completing the renaming

The schedule contains the instructions as they were renamed by the scheduler. A renamed source register has the `prev` and `arch` bits and the `phy` field. When the `prev` bit is zero, the `arch` bit determines whether the value to read is in the architectural register or the physical register indicated by `phy`. When `prev` is set, the value is produced by an instruction older than the rgroup. Depending on the status of that instruction, the architectural register or a physical register will be read. The `prev_rename` table is used to know which registers currently store the live-in values. The whole content of the `rename` table is copied into the `prev_rename` table when a new rgroup enters the Rmap logic. The Commit stage updates the `prev_rename` table in the same fashion as it updates the `rename` table; when an instruction commits, it compares the destination physical register with the content of the table for the destination logical register and sets the `arch` bit if they match.

When the `prev` bit of a source register is set, the Rmap logic accesses the `prev_rename` table with the identifier of the logical register and reads the `arch` bit and `phy` field stored there. If the `arch` bit is set, the instruction reads the architectural register. Otherwise, it reads the physical register indicated by the `phy` field.

When the `prev` bit of a source register is zero, the Rmap logic checks the `arch` bit of the instruction in the rgroup. If it is set, the instruction reads the architectural register. Otherwise, the `phy` field is used, but it has to be updated to the current state of the processor. The Rcreate logic starts assigning physical registers starting with the physical register 0, but that register may be the first free physical register when the rgroup starts execution. Since it is assumed that registers are assigned always consecutively, the identifiers assigned by the scheduler must be adapted to the current state. Therefore, an offset is added to the identifier stored in the `phy` field. There is a single offset for the whole register file. The offset does not change during the execution of an rgroup. Its value is the content of the `free_head` register when the rgroup enters the Rmap logic. It indicates which is the first free physical register at that moment. Thus, the source register is renamed to the physical register indicated by $(phy + offset) module regfile\_size$.

Figure A.12 shows an example. The physical register indicated by the rgroup is 3 and the offset is 20, so the actual register accessed is the physical register 23. The `prev_rename` table is accessed with the logical register 2. There, the `arch` bit is 0 and the `phy` field is 10. Since the `prev` bit stored in the rgroup is zero, the content of the `prev_rename` table is ignored for this instruction and the information stored in the rgroup prevails. The `arch` bit of the rgroup is zero so the physical register 23 is used.

The offset is also added to the identifier of the destination physical register. The register is available if its `busy` bit is zero. If it is set, the instruction stalls in the Rmap logic. As explained above, the `busy` bits are not needed if the number of physical registers is greater than or equal to *ROB_size*.

Figure A.13 shows an example. The destination physical register indicated in the rgroup is 46, which is renamed to the physical register 2, after the addition of the offset (assuming 64 physical register). Since the `busy` bit is zero, the physical register is available and the instruction is sent to the Issue stage.

Updating the structures

The Rmap logic must update the `rename` table and the `free_head` register to let the next instructions find a correct state to be correctly renamed without waiting for the whole current rgroup to commit.

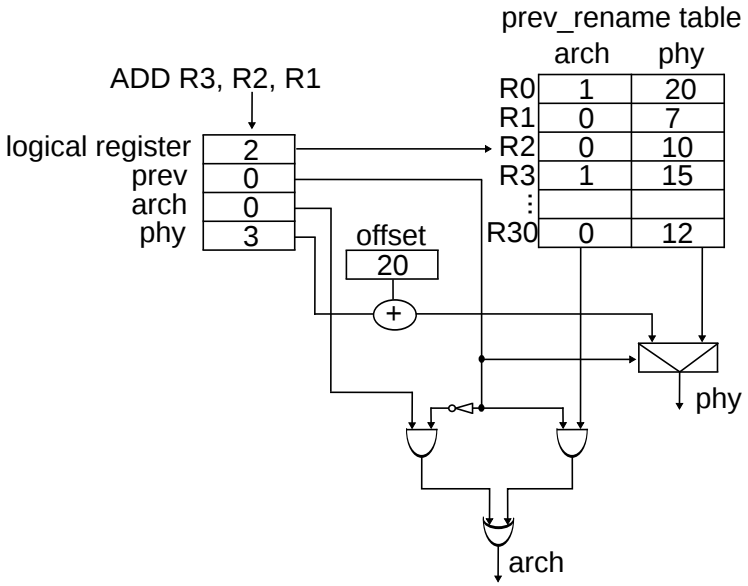


Figure A.12: The Rmap logic uses the information provided in the rgroup and the `prev_rename` table to complete the renaming of the instructions at execution time. The renaming information stored in the rgroup for the source register R2 (the logical register, the `prev` and `arch` bits and the `phy` field) are shown in the figure below the instruction.

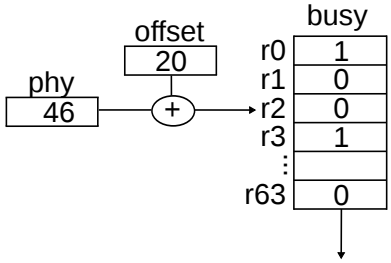


Figure A.13: The Rmap logic accesses the `busy` bit of the destination physical register to know if it is available.

If the next instructions are executed in the Icache mode, the Map stage uses the `rename` table and the `free_head` register to rename the registers. If it continues in the Rcache mode, these structures are needed anyway in order to update the `prev_rename` table and the offset for the next rgroup.

The Rmap logic can update the `free_head` register by incrementing it for each instruction that uses a destination physical register.

The `rename` table must contain the most recent renaming information of each logical register. However, the Rmap logic processes the instructions in the scheduled order and two writes to the same logical register can be reordered. Therefore, updating the `rename` table directly for each processed instruction is incorrect. The `rename` table must point to the physical register of the last in-flight instruction in program order.

The `next_rename` table is used to keep the renaming information that must be copied to the `rename` table when the last instruction of the rgroup has been processed by the Rmap logic. We want to keep the Rmap logic as simple as possible and the simplest way to detect at execution time which instructions must update the `next_rename` table is that they are already marked by the Rcreate logic. Rcreate knows which instructions in the rgroup produce the live-out values, the results that can be read by the next rgroup. Only the last write to each logical register in the rgroup can be read by instructions executed later. Furthermore, when a new rgroup starts execution, just the last $ROB\_size - 1$ instructions can still be in-flight and have not committed yet. The logical registers updated by older instructions must be read from the architectural registers. The Rcreate logic can detect which instructions from the last $ROB\_size - 1$ instructions generate a live-out value and set an additional `update_rename` bit for them. All other instructions will have this bit cleared. When the Rmap logic processes an instruction that has this bit set, it updates the `rename` table with the identifier of the destination physical register of the instruction.

The Commit stage updates the `next_rename` table as it does with the normal `rename` table. It is accessed with the destination logical register. If the `phy` field of the entry matches the destination physical register of the committed instruction, the `arch` bit of the entry is set.

Using a tag

The solution presented above uses a very simple Rmap logic to know which instructions must update the `next_rename` table. However, it requires an additional bit per instruction to be stored in the Rcache. It may be more cost-effective to detect these instructions at execution time in order to reduce the Rcache size. That could be implemented using tags to know the relative order of each instruction in the rgroup. Thus, each entry of the `rename` table has an additional `tag` field. This field is ignored when the table is used in the Icache mode. It is zeroed when a new rgroup enters the Rmap logic.

Each instruction in the rgroup has a tag. The tags are assigned sequentially in program order. For a given instruction with tag t, the Rmap logic indexes the `next_rename` table with the identifier of its destination logical register. If the current value of the `tag` field is less than the t, the instruction with tag t is younger. Thus, the `phy` and `tag` fields are updated and the `arch` bit is cleared. Otherwise, the `next_rename` table is left unmodified.

Figure A.14 shows an example. The entry of the logical register 2 is accessed. The current instruction has the tag 1.7, where the dot means that the two numbers are concatenated. Since it is greater than the current content of the `tag` field (1.0), the table is updated.

In order to generate the tags, the original order of the instructions must be known. The tag could be assigned by the Rcreate stage, but doing so makes no sense, since it would require more storage in the Rcache than the `update_rename` bit. If an rgroup has up to 256 instructions, each tag occupies one byte.

It is actually not necessary to store the tag in the rgroup. The original program order can be recovered from the identifier in the ROB that the scheduler already assigns to each instruction in the

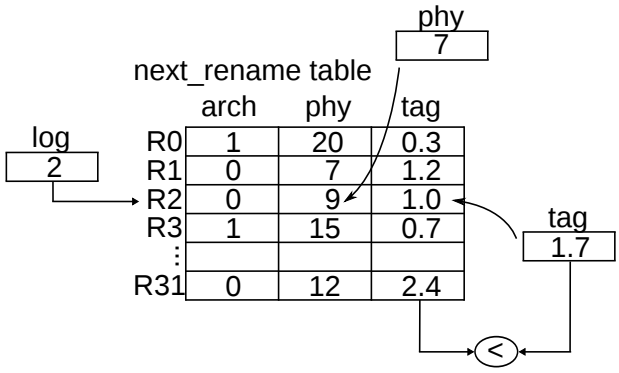


Figure A.14: The Rmap logic updates the `next_rename` table. It is indexed with the destination logical register. The logic updates the `tag` and `phy` fields and clears the `arch` bit if the new tag is greater than the current content of the field. The tag is formed by a counter (1 in the example) concatenated with an identifier (7 in the example).

rgroup. If an rgroup fits completely in the ROB, the identifiers in the ROB can be used directly as a tag.

However, if the ROB is smaller than the maximum number of instructions per rgroup, each identifier is used several times within the same rgroup and it cannot be used directly as a tag. The Rmap logic should count how many times has been reused an identifier and build the tag based on that. To do so, the Rmap logic can use a table with a counter for each identifier in the ROB, the `ROB_count` table. Thus, the tag of an instruction with the identifier in the ROB *id* is the content of the entry *id* of the `ROB_count` table concatenated with *id*. The `ROB_count` table is indexed with the `ROB_id` field of each instruction. All counters are reseted when a new rgroup enters the Rmap logic. For each Rmapped instruction, the counter of its identifier in the ROB is incremented.

Figure A.15 shows an example. The value read from the `ROB_count` table is concatenated with the `ROB_id` field of the instruction as read from the Rcache. In the example, the tag of the instruction is 1.7 and the counter of identifier 7 is incremented. Thus, the next instruction that has the identifier 7 in the ROB will have the tag 2.7.

Yet another way to obtain the tags is to count how many instructions of the current rgroup have committed in the `rgroup_committed` register and use this number to deduce the number of times that the *ROB_id* of the current instruction have been used within the rgroup. This solution reduces the number of counters from *ROB_size* to just 1, though it requires a more complex logic. The `rgroup_committed` register is incremented for every committed instruction of the rgroup. This register is initialized to zero when the rgroup enters the Rmap logic. The value of the register, modulo *ROB_size*, indicates which is the *id_next*, that is, the identifier in the ROB of the next instruction of the rgroup that will commit. The offset of the identifiers in the ROB is ignored here, so the first instruction uses the identifier 0. When a given instruction i with identifier *id* is processed by the Rmap logic, *id* is compared with *id_next*. If $id \geq next\_id$, it means that the identifier *id* has been assigned the same number of times as the *id_next* (including the instruction i). Otherwise, *id* is being assigned once more than *id_next*. To generate the tag, the Rmap logic uses the `num_ROB_reuse` register, that counts how many times the identifier 0 has been reused. It is initialized to zero and incremented each time that the `rgroup_committed` register is updated and the new value modulo *ROB_size* is 0. The tag is the content of the `num_ROB_reuse` register concatenated with *id* when $id \geq next\_id$. Otherwise, the tag is $num\_ROB\_reuse + 1$ concatenated with *id*.[1]

[1] Actually, only the last $ROB\_size - 1$ instructions in the rgroup can be in the ROB (not committed), when the first

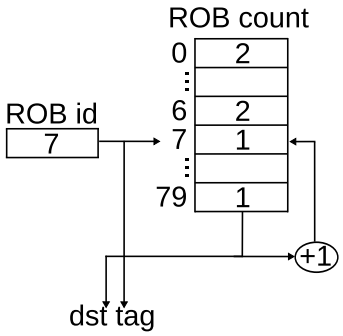


Figure A.15: The ROB_count table is used to create the tags of the Rmapped instructions.

Renaming loop

The Rmap logic reads the `prev_rename` table and updates the `next_rename` table. Thus, register renaming of the instructions of the rgroup do not depend on the changes made in these tables by the other instructions in the rgroup. Therefore, the tight logic loop that appears in the renaming logic of the out-of-order processors is not present in the Rmap logic. The `next_rename` table can be updated in a pipelined fashion without compromising the next instructions in the rgroup. However, it would have an impact when a new rgroup enters the Rmap logic.

A.5 The Rcache

The Rcache stores the renaming information of each instruction. When the common register file is used, the total number of physical registers determines how many bits are needed for the identifier of the physical register. With 72 physical registers, the identifier has seven bits.

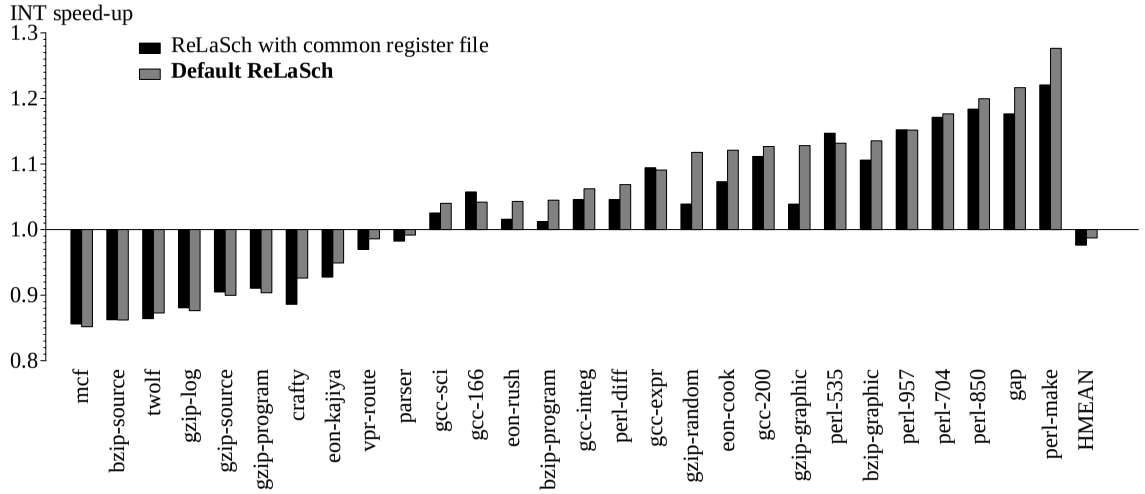
For a destination register, only the identifier of the physical register is stored. A source register can be in a physical register, in the architectural register, or in a place unknown until execution time. Thus, besides the seven bits of the identifier two extra bits (`arch` and `prev`) must be stored for each source register. However, if there are some unused identifiers (when the number of physical registers is not a power of two, as in our default configuration), a couple of unused identifiers can be used to encode the architectural and previous cases and use just seven bits per source register. The Rdecode logic would set the `arch` and `prev` bits accordingly. Besides, an additional `update_rename` bit is needed for each instruction, assuming that the Rmap logic does not use tags in order to update the `next_rename` table. This adds a total of 22 bits per instruction for the renaming information.

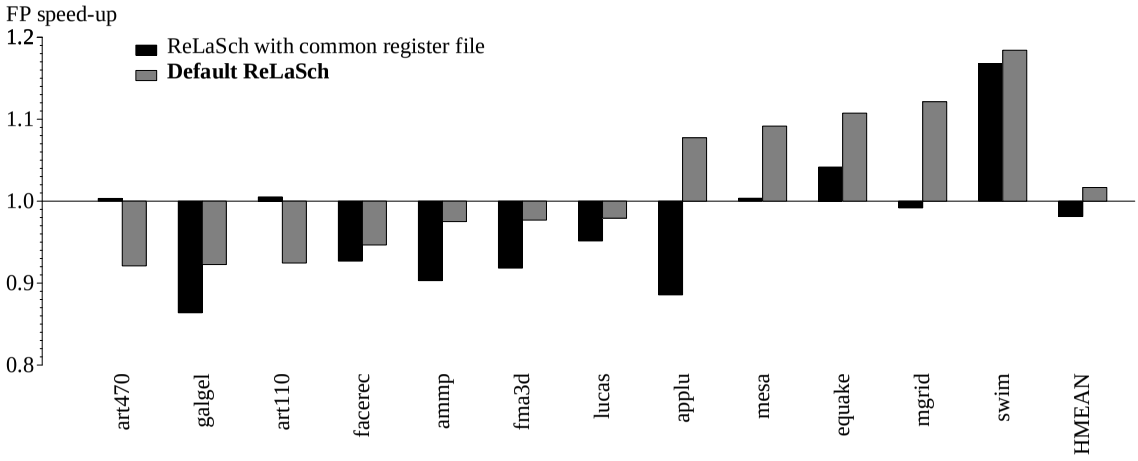
A.6 Experimental results

Figure A.16 shows the speed-up of the ReLaSch processor using the common register file as well as the default ReLaSch as defined in table 9.1.

---

instruction of the next rgroup is processed. So all the values not overwritten by the last *ROB_size* instructions are for sure in the architectural registers when the next rgroup begins its execution. Therefore, only the last instructions need to update the `rename` table and require a tag. Since the number of instructions that require a tag is less than the size of the ROB, there is no need to count the number of times an identifier has been used. However, to do so it is needed to detect which are the last *ROB_size* instructions in the program order. Though it is possible to detect is at execution time, it requires using a more complex logic than just counting the number of times the identifiers have been reused.





	ReLaSch with common register file	Default ReLaSch
HMEAN INT	0.98	0.99
HMEAN FP	0.98	1.02
HMEAN ALL	0.98	1.00

Figure A.16: Speed-up using the default and the common register file.

The ReLaSch processor with a common register file achieves a similar average speed-up as the default ReLaSch, though slightly lower. In some benchmarks (e.g. *gcc-166*, *art-470* and *art-110*) the ReLaSch processor with a common register file has better IPC than the default ReLaSch, since these benchmarks make an intensive use of a small number of logical registers. So in these benchmarks, the processor often stalls waiting for a free destination physical register when there is a fixed set of physical registers. In some other benchmarks (e.g. *ammp*, *applu*, *mesa*, *mgrid*, *gzip-random*, *eon-cook*, *gzip-graphic*, *perl-make*) the default ReLaSch achieves higher speed-up. That happens when the applications use a wide set of logical registers and benefit from having a larger total number of physical registers available. See table C.19 for details on the distribution of the writes to the logical registers.

We have chosen the register file with fixed sets of physical registers because the average IPC achieved is very similar in both cases and the renaming process with the common register file is more complex. In particular, the complexity of the Rmap logic when an rgroup can be processed even if there are older in-flight instructions. It is crucial to achieve good performance results. Since the Rmap logic is in the execution critical path, choosing the register file with fixed sets is coherent with our approach to keep the execution path as simple as possible.

Appendix B

The 21264 Alpha processor

In this chapter, the main characteristics of the 21264 Alpha processor are described. The ReLaSch processor, as well as the IO and OoO reference processors are based on the 21264 processor. Several papers and manuals related with the Alpha ISA and the 21264 processor have been used as reference during the research of this thesis [59, 15, 18, 60, 61, 62, 63].

The first section of this chapter introduces the Alpha Instruction Set Architecture, which is followed of a description of the more relevant aspects of the 21264 implementation of that ISA. Finally, the main differences between the processor and the simulator used in our experiments are presented.

B.1 The Alpha ISA

The Alpha AXP architecture is a 64-bit RISC architecture, with a limited instruction set and register-only operands. It has specific load and store instructions to read and write the memory space.

It has separated registers and instructions for the integer and the floating point data. There are 32 integer registers, R0 to R31. The register R31 always return the value 0 on a read. There are 32 floating point registers, F0 to F31. The register F31 always return the value 0 on a read. All the registers are 64 bits wide.

The Program Counter (PC) is a special register that indicates the next instruction to be executed. All instructions are encoded in 32 bits and are stored in memory in 32-bit aligned addresses.

The control instructions are divided into the conditional and the unconditional branches. The conditional branches test a condition on an explicitly coded integer or floating point register. All the conditional branches are relative to the PC, adding an immediate displacement. The unconditional branches can be relative or indirect. Indirect branches read the next PC from a register. The current PC+4 is stored in the destination register of the unconditional branch.

Each instruction has up to two source registers and one destination register of the general purpose registers. The PC register is implicitly used. Some instructions use an immediate operand.

B.1.1 Memory accesses

The Alpha ISA defines 64-bit virtual addresses to byte for the memory accesses. It supports 8- to 64-bit integer data types, as well as the S and T IEEE formats and several VAX floating-point formats (32- and 64-bit). The data is stored using the little-endian scheme.

All the addresses must be naturally aligned to the data size. An exception is generated otherwise. The LDQ_U load instruction ignores the lower bits of the address and can be combined with byte manipulation instructions to implement a fast access to unaligned data.

B.2 The 21264 Alpha processor

The 21264 Alpha processor implements the Alpha ISA. It is a 4-wide superscalar, with out-of-order execution pipeline and precise exceptions. It renames the registers to eliminate WAW and WAR hazards, and predicts the branches to execute the instructions after the branch speculatively. There are two levels of memory hierarchy between the processor and main memory. In the first level there are two separated caches, the Instruction Cache or Icache and the Data Cache or Dcache. Both have a size of 64KB and are 2-way set-associative. The second level is formed by a unified off-chip cache.

B.2.1 The pipeline

The pipeline of the processor is shown in figure B.2.1. The Fetch stage reads the instructions from the Icache and accesses the branch predictor. The result of the prediction is not known until one cycle later. The next stage completes the branch prediction, decodes the instructions and performs slotting (explained in detail in section B.2.2). The Map stage renames the instructions and inserts them in the Reorder Buffer (ROB), the Load and Store queues if needed and the issue queue. These three stages process the instructions in-order.

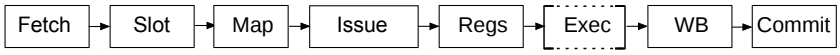


Figure B.1: The pipeline of the 21264 Alpha processor.

The Issue stage wakes-up and selects out-of-order instructions that are present in the issue queue. An instruction can be selected if its source registers are ready and a suitable Functional Unit is available. The registers are read in the next stage, and execution happens in the corresponding functional unit afterwards. Once the execution in the functional unit finishes, the registers are written in the Writeback stage, where the load instructions also access the Dcache. The Writeback stage also checks for memory ordering violations with the help of the Load and Store queues.

Finally, instructions are retired in-order by the Commit stage. Once an instruction that has finished execution reaches the head of the ROB, the Commit stage sets its destination physical register as the architectural register for the corresponding logical register, checks for mispredictions and frees the entry used by the instruction in the ROB and, when needed, in the Load or Store queues. The Commit stage also performs the access to memory of the store instructions.

The Fetch, Slot and Map stages can process up to four instructions per cycle. The Issue stage can process up to four integer and two floating point instructions per cycle. Up to four integer and two floating point instructions can write the destination physical register per cycle. Besides, up to two integer and two floating point loads can write their destination physical register per cycle. The Commit stage can sustain a maximum rate of eight instructions per cycle, although it is able to commit up to 11 instructions in a single cycle.

The processor uses an scoreboard to know when it can issue the instructions. They are inserted in the issue queues. There is an integer and a floating point queue. Each one has 32 entries, and in which queue is inserted an instruction depends on its destination register. Floating point store instructions and floating point to integer conversion instructions are inserted in both queues. When an instruction has all its source registers available, it notifies that it is ready to an arbiter. There are two arbiters for the integer queue. Each one selects up to two instructions per cycle to enter a functional unit and be executed. The arbiter of the floating point queue selects up to two instructions per cycle. Older instructions have higher priority in the arbitration logic.

The instructions are executed out-of-order, but commit in-order. Up to 80 instructions can be in-flight, mapped and inserted in the ROB but not committed yet. Many instructions that have R31 or F31 as destination register are early retired at the Map stage, before being inserted in the issue

queues, and do not commit. However, some load instructions that write these registers are not early retired, since they are used to perform software pre-fetching.

B.2.2 Slots and Functional Units

The integer functional units in the Alpha 21264 are grouped into two clusters: 0 and 1. It requires an extra cycle to bypass a result between clusters. Each cluster is further divided into two subcluster types: Upper and Lower. Therefore, there are four subclusters in total: Upper0, Upper1, Lower0 and Lower1. Some functional units are replicated in all the four subclusters, while others are present in only one or two subclusters. That imposes a restriction on which subclusters can be executed an instruction. Slotting is the assignment of a subcluster type (Upper or Lower) to an instruction. An instruction is slotted according to several rules that depend on the instruction's type and which are the types of the instructions fetched in the same cycle. This task is performed in the Slot stage. The actual cluster in which the instruction will be executed (0 or 1) is chosen at issuing time. There is an issue-arbiter for each cluster, so up to two instructions can be issued per cluster each cycle.

The OoO, IO and ReLaSch processors do not cluster the functional units and do not perform slotting. If these techniques were implemented in the ReLaSch processor, the Rcreate logic should take it into account when scheduling the instructions in the issue-groups. In particular, it should avoid placing in the same issue-group several instructions that can only go to the same subcluster (Lower0, Lower1, Upper0 and Upper1), more than two instructions to the same cluster (0 and 1) and more than two instructions to the same subcluster type (Lower and Upper). To do so, the Rcreate logic could use `busy` bits, similar to the ones assigned to the functional units, to follow the slotting rules. Furthermore, it should take into account, when scheduling a dependent instruction, that two instructions that are executed in different clusters need an extra cycle to bypass the data. The `reg_info` table could be expanded to remember in which cluster is executed the instruction that generates each logical register.

B.2.3 Branch prediction

The 21264 Alpha processor uses two structures to predict the branches. The line predictor is very simple and its accuracy is not very high, but it is fast and provides a prediction in the same cycle it is accessed. The branch predictor performs more accurate predictions, but it needs two cycles to return a result. So the Fetch stage accesses both predictors in parallel, and uses the line predictor to decide which instructions are fetched in the next cycle. The outcome of the access to the branch predictor is available in the Slot stage, and it is then compared with the prediction of the line predictor that was used the previous cycle. If the predictions differ, the instructions in the Slot and Fetch stage are dismissed and the Fetch stage reads the instructions from the path predicted by the branch predictor.

The branch predictor is a McFarling predictor [26], which contains three major elements: the local predictor makes a prediction based on the local history of the current branch; the global predictor uses the global history of the most recent 12 conditional branches; and the choice predictor selects the winning predictor when local and global predictions differ. It uses the global history and two-bit saturating counters, updated when one predictor misses and the other hits. To predict the target of the indirect branches, it uses a Branch Target Buffer and a Return Address Stack.

B.2.4 Memory accesses

The memory instructions are inserted in the Mbox structure at the same time they enter the Issue queues. The Mbox ensures the correct ordering of the load and store instructions. The Mbox stores the load instructions in the Load Queue and the store instructions in the Store Queue. Both queues have 32 entries, and the instructions are not removed from the entry until they commit.

Aliased memory instructions perform the accesses in order. Unaliased instructions can be re-ordered. Two memory instructions are aliased if there is at least one common byte in the accesses. Since accesses are naturally aligned, if the two instructions perform accesses of the same size, they must have exactly the same address to be aliased. If the two aliased instructions have different access sizes, the smaller one is one of the bytes, words or longwords of the bigger one, and have equal addresses if the lower bits are ignored.

The entries in the Store Queue contain the data that must be written into memory. The data can be bypassed to an aliased load, even if the load is smaller and it matches only partially the address of the store.

When the Mbox detects any violation of the memory ordering rules, the pipeline is flushed and the instruction is replayed (that is, re-executed). There are other situations that make the Mbox replay an instruction, related with the outstanding cache misses. When the Mbox detects an unaligned accesses, trap code is executed to solve the problem.

The Mbox uses the stWait table to detect the loads that frequently must be replayed due to an store-load order violation. If the stWait entry of a load instruction is set, the load is not issued until all older store instructions have committed. The bit is set after a store-load order trap, and the whole table is cleared every 16,384 cycles.

The Mbox also includes the Data TLB and controls the access to the L1 Data Cache. The Alpha ISA defines 64-bit virtual addresses but the 21264 processor uses either 43- or 48-bit virtual addresses. The size is configured with an internal register. They are translated into 44-bit physical addresses.

B.2.5 Register renaming

The Alpha 21264 has separated register files for the integer and floating point registers. Each register file has place for the architectural value of 31 registers (R31 and F31 occupy no physical registers) and 41 extra registers to store the result of the in-flight, uncommitted instructions, as well as eight additional registers for privileged code.

The source registers of each instruction are renamed to the physical register corresponding to the most recent write to the logical register. Destination registers are renamed to the registers present in the free-list. When an instruction commits, the physical register that had the previous architectural value of the destination logical register is inserted in the free-list.

B.2.6 Conditional moves

The conditional move instructions move register b to the destination register c if register a meets a condition. Register c is left unmodified otherwise. Since registers are renamed, the physical register where c is stored changes with the execution of the conditional move. Therefore, if the condition is false, the old content of c must be copied from one physical register to the other. That means that conditional moves have actually three physical registers. The Alpha 21264 processor assumes that each instruction has only up to two source registers per instruction. In order to implement the conditional move instructions, they are internally divided into two different instructions. Therefore, three different physical registers will be used to store the logical register c: phy (the original value), $phy1$ (the destination of the first internal instruction) and $phy2$ (the destination of the second internal instruction and the one that will store the architectural value of c once the conditional move commits). Besides, it is also added the `cmov` bit to each physical register.

The first instruction reads b and tests the condition. The result of the writes the result in the `cmov` bit of the physical register $phy1$. It also copies the current value of the c, stored in phy, into $phy1$. The second instruction reads the physical register that stores a and the physical register $phy1$. The `cmov` bit of $phy1$ is used to select the value that will be stored in $phy2$. The Commit stage takes into

account that the two instructions implement a single conditional move instruction and both commit at the same time.

B.3 The sim-alpha simulator

The sim-alpha is a simulator of the 21264 Alpha processor, that is intended to implement a realistic model of the processor. It has been verified against a real Alpha machine. However, there are some differences that are detailed in this section. Besides, the changes that we have made to the original code of the sim-alpha simulator are also described.

In the original sim-alpha code, the physical registers are not read after the Issue stage, but in the Writeback stage. The difference has no effect in the sim-alpha simulator, but it is a problem if it is required that the register are read in the issue order, as we do in the appendix A. So our simulator reads the registers one cycle after the instruction has been issued, just as the 21264 processor does.

The instruction LDS, that reads an IEEE floating point value from memory, had a bug in the original code, that didn't detect correctly the NaN case. It has been corrected in our simulations.

Some instructions of the Alpha ISA are not implemented in sim-alpha. The unimplemented instructions are: a) those that use the VAX floating point formats (LDF, LDG, STF, STG, ADDF, etc.); b) barrier instructions (EXCB, MB, WMB and TRAPB); and c) instructions that help to manage the cache and perform pre-fetching (ECB, FETCH, WH64). However, software pre-fetching can also be done using load instructions to the R31 register. Overflow and underflow checking is also unimplemented. Therefore, the Floating Point Control Register is ignored by the floating point instructions. The floating point instructions just check that it is not dividing by 0 or performing a square-root on a negative number, and abort the execution of the program whenever that happens. The memory instructions used to access shared data in a multiprocessor environment (LDx_L and STx_C) are implemented just like normal load and store instructions.

The documentation of the 21264 Alpha states that it can sustain a rate of 8 committed instructions per cycle, although up to 11 instructions can commit in a single cycle. The Commit stage in the sim-alpha retires up to 11 per cycle. Besides, a control instruction can commit only if it is the first instruction retired in a given cycle. This implicitly limits the number of committed control instructions to one per cycle. As far as we know, this limitation is not documented, but it makes sense since when control instructions commit, they must update the branch predictor, which in the access in the Fetch stage is already limited to one instruction per cycle.

Sim-alpha is not a full system simulator and system code is not simulated cycle by cycle. The pipeline is flushed on a system call; then the architectural state is used to perform a call to the OS of the machine where the simulator is being run, and the architectural state of the simulated processor is updated as the OS would have done. In some cases the simulator itself updates the architectural state directly, without a call to the OS.

The Store Queue in the Mbox allows bypassing the data of an store to an aliased load that access only a part of the store data. Sim-alpha requires that the size of the two accesses are equal, and that the data type (integer or floating point) also matches. Otherwise, the load is re-executed. The original code was not able to forward the data to an LDS instruction either; this limitation has been removed in our experiments. Furthermore, the original code considered as aliased some memory instructions that accessed different parts of the same quadword, because it ignored the actual size of the access. Although it has no effect on the architectural state of the processor, some instructions are re-executed unnecessarily. The code has been modified in our simulations to detect only the actual aliases.

Appendix C

Benchmarks

This chapter presents characterization information about the benchmarks used in this Thesis. It also presents some additional information of some experiments presented in chapter 9. When the data presented depends on the configuration of the processor where the benchmark runs (i.e. branch misprediction rate), the default ReLaSch configuration as defined in table 9.1 is used.

	memory	control	fpalu	intalu
ammp	35.18	8.01	36.01	20.81
applu	40.28	0.55	51.53	7.64
art110	32.26	13.98	19.75	34.02
art470	32.33	13.95	19.64	34.08
equake	44.38	4.07	33.51	18.04
facerec	25.70	7.14	18.07	49.09
fma3d	44.39	4.05	30.58	20.98
galgel	42.68	5.13	25.42	26.76
lucas	22.28	2.88	43.75	31.09
mesa	37.27	8.95	12.70	41.08
mgrid	36.68	0.29	57.65	5.38
swim	28.45	0.48	45.62	25.44
FP	35.16	5.79	32.85	26.20
bzip-graphic	42.33	10.49	0.00	47.17
bzip-program	42.93	9.92	0.00	47.15
bzip-source	34.20	14.62	0.00	51.18
crafty	33.65	11.88	0.00	54.48
eon-cook	46.74	12.02	12.22	29.02
eon-kajiya	47.96	11.62	13.19	27.23
eon-rush	47.23	12.24	11.91	28.62
gap	35.04	16.48	0.00	48.48
gcc-166	38.58	14.84	0.00	46.58
gcc-200	38.97	16.77	0.00	44.26
gcc-expr	38.36	17.11	0.00	44.53
gcc-integ	42.66	15.70	0.00	41.64
gcc-sci	41.76	17.85	0.00	40.39
gzip-graphic	29.42	10.33	0.00	60.25
gzip-log	26.33	13.54	0.00	60.13
gzip-program	19.53	15.29	0.00	65.17
gzip-random	29.93	9.80	0.00	60.28
gzip-source	23.14	13.27	0.00	63.59
mcf	36.46	20.93	0.00	42.61
parser	31.01	16.46	0.00	52.54
perl-diff	43.57	14.50	0.06	41.87
perl-make	36.94	14.17	1.62	47.26
perl-535	44.45	14.37	0.03	41.15
perl-704	45.31	14.19	0.03	40.48
perl-850	44.37	14.46	0.04	41.14
perl-957	44.58	14.36	0.04	41.02
twolf	29.31	12.63	4.62	53.44
vpr-route	42.53	10.95	5.87	40.65
INT	37.76	13.96	1.77	46.51
ALL	36.98	11.51	11.10	40.42

Table C.1: Instruction mix

	FP load	INT load	FP store	INT store
ammp	15.94	8.65	9.41	1.18
applu	27.72	1.97	10.58	0.00
art110	14.97	9.09	8.20	0.00
art470	14.99	9.27	8.06	0.00
bzip-graphic	0.00	27.73	0.00	14.61
bzip-program	0.00	27.90	0.00	15.03
bzip-source	0.10	26.19	0.00	7.91
crafty	0.03	28.28	0.00	5.33
eon-cook	13.61	12.92	11.97	8.24
eon-kajiya	14.50	12.39	12.79	8.28
eon-rush	13.59	12.38	13.11	8.15
equake	26.36	13.30	3.66	1.06
facerec	12.29	11.69	0.00	1.72
fma3d	20.69	8.12	12.32	3.26
galgel	21.89	15.23	5.56	0.00
gap	0.01	24.87	0.00	10.16
gcc-166	0.02	20.30	0.01	18.25
gcc-200	0.71	25.05	0.01	13.19
gcc-expr	0.60	25.34	0.01	12.41
gcc-integ	0.75	26.13	0.01	15.78
gcc-sci	0.04	25.93	0.01	15.79
gzip-graphic	0.00	21.20	0.00	8.22
gzip-log	0.00	21.59	0.00	4.74
gzip-program	0.00	16.35	0.00	3.18
gzip-random	0.00	19.40	0.00	10.53
gzip-source	0.00	16.84	0.00	6.30
lucas	11.92	8.60	1.76	0.00
mcf	1.50	30.53	0.00	4.44
mesa	5.44	19.64	2.10	10.09
mgrid	29.36	2.47	4.67	0.19
parser	0.08	21.53	0.00	9.39
perl-diff	0.06	27.15	0.05	16.31
perl-make	1.06	25.12	0.42	10.34
perl-535	0.03	26.91	0.02	17.48
perl-704	0.02	27.14	0.02	18.12
perl-850	0.03	26.63	0.03	17.68
perl-957	0.03	27.07	0.03	17.45
swim	17.87	4.11	6.48	0.00
twolf	0.43	21.89	0.04	6.95
vpr-route	9.59	21.31	1.06	10.57
FP	18.29	9.34	6.07	1.46
INT	2.03	23.07	1.41	11.24
ALL	6.91	18.96	2.81	8.31

Table C.2: Number of each memory instruction type (in millions).

	fcmov	icmov	conv
ammp	0	0	864
applu	0	539,404	9
art110	1,080,000	534,910	554,691
art470	1,060,000	533,794	553,598
bzip-graphic	0	28,635	0
bzip-program	0	141,047	0
bzip-source	0	157,450	24
crafty	0	2,333,381	31,329
eon-cook	288,992	294,678	709,921
eon-kajiya	340,536	171,316	662,806
eon-rush	310,277	223,028	625,554
equake	0	8	0
facerec	0	314,592	1,947
fma3d	147,848	505,236	73,926
galgel	0	543,043	0
gap	0	89,813	213
gcc-166	0	214,238	5,132
gcc-200	0	378,243	18,417
gcc-expr	1	515,720	16,517
gcc-integ	4	600,948	29,039
gcc-sci	0	435,278	35,666
gzip-graphic	0	1,593,085	0
gzip-log	0	170,482	0
gzip-program	0	355,696	0
gzip-random	0	1,868,542	0
gzip-source	1	488,814	0
lucas	4,639,409	0	0
mcf	0	117,171	0
mesa	188,798	1,024,696	1,455,050
mgrid	0	167,519	0
parser	0	3,965,244	79,182
perl-diff	624	1,646,727	5,894
perl-make	211,804	2,931,234	353,008
perl-535	207	1,865,642	2,946
perl-704	154	2,042,901	2,214
perl-850	113	1,903,788	1,620
perl-957	192	1,882,806	2,732
swim	0	1,613,849	0
twolf	2	4,038,293	2,089,248
vpr-route	0	0	1,681,454
FP	593,005	481,421	220,007
INT	41,175	1,087,650	226,890
ALL	206,724	905,781	224,825

Table C.3: Total number of conditional moves and conversion instructions between integer and floating point benchmarks.

targets	1	2	3	4	5	6	7	8	9	>=10
ammp	1	0	0	0	0	0	0	0	0	0
applu	24	7	1	2	0	0	0	1	0	0
art110	6	0	0	0	0	0	0	0	0	0
art470	6	0	0	0	0	0	0	0	0	0
bzip-graphic	6	0	1	0	0	0	0	0	0	1
bzip-program	5	1	0	1	0	0	0	0	0	1
bzip-source	10	1	1	0	0	0	0	0	0	1
crafty	42	27	12	3	3	5	1	2	1	9
eon-cook	65	8	6	3	1	0	1	0	1	2
eon-kajiya	76	23	2	7	1	3	0	1	0	3
eon-rush	67	20	4	6	1	0	1	0	1	3
equake	4	0	0	0	0	0	0	0	0	0
facerec	3	2	0	0	0	0	0	0	0	0
fma3d	15	0	0	0	0	1	0	0	0	0
galgel	1	0	0	0	0	0	0	0	0	0
gap	172	70	31	29	21	14	6	5	5	16
gcc-166	197	65	38	24	12	12	9	8	5	32
gcc-200	161	80	49	43	19	18	13	12	4	32
gcc-expr	220	111	61	44	24	22	22	11	7	53
gcc-integ	159	83	33	28	22	8	11	8	3	19
gcc-sci	132	57	29	25	20	7	8	5	6	19
gzip-graphic	13	3	3	0	0	0	0	0	0	1
gzip-log	43	10	6	2	1	0	0	0	0	1
gzip-program	12	5	3	0	0	0	0	0	0	1
gzip-random	33	6	3	1	0	0	0	0	0	0
gzip-source	55	15	8	4	0	0	0	0	0	1
lucas	0	0	0	0	0	0	0	0	0	0
mcf	21	7	1	0	0	0	0	0	0	0
mesa	47	3	0	0	0	0	0	0	0	0
mgrid	25	3	1	2	0	0	0	0	0	0
parser	119	75	29	15	1	3	4	2	1	12
perl-diff	222	39	18	12	7	5	5	5	3	16
perl-make	51	3	3	0	0	0	0	0	0	1
perl-535	231	49	25	14	8	7	3	2	4	18
perl-704	218	49	18	13	6	5	2	4	2	18
perl-850	226	42	18	13	6	5	2	3	2	18
perl-957	219	49	17	14	6	4	2	5	2	18
swim	1	0	0	0	0	0	0	0	0	0
twolf	67	18	13	8	0	0	0	0	0	0
vpr-route	9	3	1	0	0	0	0	0	0	0

Table C.4: Number of static indirect branches with k different targets.

number of targets (up to)	10	20	30	40	50	60	70	80	90	>=100
ammp	1	0	0	0	0	0	0	0	0	0
applu	35	0	0	0	0	0	0	0	0	0
art110	6	0	0	0	0	0	0	0	0	0
art470	6	0	0	0	0	0	0	0	0	0
bzip-graphic	7	0	1	0	0	0	0	0	0	0
bzip-program	7	0	1	0	0	0	0	0	0	0
bzip-source	12	1	0	0	0	0	0	0	0	0
crafty	96	1	2	3	0	2	1	0	0	0
eon-cook	86	1	0	0	0	0	0	0	0	0
eon-kajiya	114	2	0	0	0	0	0	0	0	0
eon-rush	100	3	0	0	0	0	0	0	0	0
equake	4	0	0	0	0	0	0	0	0	0
facerec	5	0	0	0	0	0	0	0	0	0
fma3d	16	0	0	0	0	0	0	0	0	0
galgel	1	0	0	0	0	0	0	0	0	0
gap	355	8	1	3	1	0	1	0	0	0
gcc-166	372	23	2	2	2	1	0	0	0	0
gcc-200	407	15	3	0	3	3	0	0	0	0
gcc-expr	527	36	4	3	1	1	2	0	0	1
gcc-integ	356	13	2	3	0	0	0	0	0	0
gcc-sci	292	11	2	2	1	0	0	0	0	0
gzip-graphic	19	1	0	0	0	0	0	0	0	0
gzip-log	62	1	0	0	0	0	0	0	0	0
gzip-program	20	1	0	0	0	0	0	0	0	0
gzip-random	43	0	0	0	0	0	0	0	0	0
gzip-source	82	1	0	0	0	0	0	0	0	0
lucas	0	0	0	0	0	0	0	0	0	0
mcf	29	0	0	0	0	0	0	0	0	0
mesa	50	0	0	0	0	0	0	0	0	0
mgrid	31	0	0	0	0	0	0	0	0	0
parser	251	6	1	0	0	2	0	1	0	0
perl-diff	317	11	3	0	0	0	0	0	1	0
perl-make	57	1	0	0	0	0	0	0	0	0
perl-535	344	13	2	1	0	0	0	0	1	0
perl-704	320	11	2	1	0	0	0	0	1	0
perl-850	320	11	2	1	0	0	0	0	1	0
perl-957	321	11	2	1	0	0	0	0	1	0
swim	1	0	0	0	0	0	0	0	0	0
twolf	106	0	0	0	0	0	0	0	0	0
vpr-route	13	0	0	0	0	0	0	0	0	0

Table C.5: Number of static indirect branches with k different targets.

targets	1	2	3	4	5	6	7	8	9	>=10
ammp	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
applu	44.22	14.34	2.39	22.71	0.00	0.00	0.00	16.33	0.00	0.00
art110	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
art470	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
bzip-graphic	36.69	0.00	24.98	0.00	0.00	0.00	0.00	0.00	0.00	38.33
bzip-program	0.00	48.91	0.00	18.81	0.00	0.00	0.00	0.00	0.00	32.28
bzip-source	5.85	3.58	90.56	0.00	0.00	0.00	0.00	0.00	0.00	0.00
crafty	7.97	13.96	10.30	3.50	2.75	10.00	2.67	7.17	0.08	41.60
eon-cook	37.41	21.42	7.97	5.50	4.47	0.00	1.67	0.00	3.19	18.37
eon-kajiya	28.27	29.84	0.49	12.71	1.83	2.42	0.00	1.52	0.00	22.93
eon-rush	26.13	29.32	3.58	9.20	3.29	0.00	1.30	0.00	2.57	24.61
equake	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
facerec	88.66	11.34	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
fma3d	50.20	0.00	0.00	0.00	0.00	49.80	0.00	0.00	0.00	0.00
galgel	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
gap	5.91	10.59	12.01	16.14	5.48	4.42	1.93	2.86	2.22	38.45
gcc-166	8.88	9.84	7.86	10.46	9.47	8.47	2.76	2.53	4.79	34.94
gcc-200	5.51	8.37	12.86	11.39	10.29	6.50	6.42	8.39	3.08	27.19
gcc-expr	4.08	6.10	13.73	9.13	7.17	9.38	7.49	5.10	4.63	33.18
gcc-integ	4.30	4.42	13.01	9.25	17.87	3.08	11.33	5.69	5.99	25.06
gcc-sci	6.50	5.40	13.82	11.53	8.65	6.14	3.17	6.26	4.89	33.63
gzip-graphic	24.00	37.58	0.78	0.00	0.00	0.00	0.00	0.00	0.00	37.65
gzip-log	29.18	17.10	6.55	0.69	25.25	0.00	0.00	0.00	0.00	21.23
gzip-program	22.80	22.48	1.56	0.00	0.00	0.00	0.00	0.00	0.00	53.17
gzip-random	38.10	0.01	61.88	0.01	0.00	0.00	0.00	0.00	0.00	0.00
gzip-source	32.95	23.43	1.13	8.53	0.00	0.00	0.00	0.00	0.00	33.96
lucas	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
mcf	0.56	91.52	7.92	0.00	0.00	0.00	0.00	0.00	0.00	0.00
mesa	81.06	18.94	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
mgrid	50.88	21.05	6.58	21.49	0.00	0.00	0.00	0.00	0.00	0.00
parser	14.06	26.28	15.21	6.86	2.23	5.95	1.75	0.02	0.55	27.10
perl-diff	16.51	10.15	1.81	3.50	1.40	0.99	7.56	16.15	0.95	40.99
perl-make	61.55	3.85	0.01	0.00	0.00	0.00	0.00	0.00	0.00	34.59
perl-535	15.17	1.47	27.82	1.20	1.53	1.01	2.20	0.09	1.66	47.85
perl-704	12.68	1.60	30.51	1.02	1.32	0.74	2.32	0.21	1.16	48.45
perl-850	12.88	1.67	30.07	1.28	1.47	0.83	2.33	0.20	1.01	48.27
perl-957	15.90	2.07	25.82	1.99	1.73	0.77	2.15	0.48	1.44	47.63
swim	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
twolf	25.48	25.42	24.37	24.74	0.00	0.00	0.00	0.00	0.00	0.00
vpr-route	21.71	20.64	57.65	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table C.6: Percentage of dynamic indirect branches with k different targets.

(up to)	10	20	30	40	50	60	70	80	90	>=100
ammp	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
applu	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
art110	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
art470	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
bzip-graphic	61.67	0.00	38.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00
bzip-program	67.72	0.00	32.28	0.00	0.00	0.00	0.00	0.00	0.00	0.00
bzip-source	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
crafty	58.40	0.24	2.01	10.55	0.00	22.44	6.36	0.00	0.00	0.00
eon-cook	84.43	15.57	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
eon-kajiya	80.02	19.98	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
eon-rush	75.39	24.61	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
equake	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
facerec	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
fma3d	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
galgel	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
gap	62.86	10.37	0.70	8.35	0.98	0.00	16.73	0.00	0.00	0.00
gcc-166	67.41	22.43	2.26	5.05	1.01	1.84	0.00	0.00	0.00	0.00
gcc-200	78.06	12.36	1.54	0.00	6.20	1.84	0.00	0.00	0.00	0.00
gcc-expr	69.18	20.05	2.01	1.69	3.24	1.93	1.09	0.00	0.00	0.80
gcc-integ	75.17	18.13	2.83	3.87	0.00	0.00	0.00	0.00	0.00	0.00
gcc-sci	68.90	20.86	0.75	7.76	1.74	0.00	0.00	0.00	0.00	0.00
gzip-graphic	62.35	37.65	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
gzip-log	78.77	21.23	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
gzip-program	46.83	53.17	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
gzip-random	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
gzip-source	66.04	33.96	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
lucas	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
mcf	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
mesa	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
mgrid	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
parser	76.74	4.89	0.05	0.00	0.00	18.31	0.00	0.01	0.00	0.00
perl-diff	59.48	6.93	21.63	0.00	0.00	0.00	0.00	0.00	11.96	0.00
perl-make	65.41	34.59	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
perl-535	52.54	34.55	0.89	0.85	0.00	0.00	0.00	0.00	11.17	0.00
perl-704	52.38	37.08	0.80	0.73	0.00	0.00	0.00	0.00	9.01	0.00
perl-850	52.70	36.92	1.09	0.89	0.00	0.00	0.00	0.00	8.41	0.00
perl-957	53.47	33.24	1.12	0.99	0.00	0.00	0.00	0.00	11.19	0.00
swim	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
twolf	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
vpr-route	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table C.7: Percentage of dynamic indirect branches with k different targets.

	ReLaSch	OoO
ammp	2.29	2.06
applu	5.31	0.81
art110	0.41	3.33
art470	0.43	3.33
equake	3.45	2.13
facerec	1.79	1.18
fma3d	0.73	0.34
galgel	0.61	0.47
lucas	0.00	0.00
mesa	3.07	2.74
mgrid	5.40	1.89
swim	0.61	0.30
FP	2.01	1.55
bzip-graphic	13.35	10.41
bzip-program	13.52	10.06
bzip-source	16.79	10.48
crafty	9.36	7.10
eon-cook	1.22	1.39
eon-kajiya	8.71	5.93
eon-rush	4.21	3.21
gap	4.86	5.23
gcc-166	6.20	5.80
gcc-200	5.82	5.64
gcc-expr	6.20	5.97
gcc-integ	6.93	6.34
gcc-sci	8.32	7.73
gzip-graphic	6.76	5.23
gzip-log	5.63	3.65
gzip-program	6.96	4.40
gzip-random	6.53	5.35
gzip-source	7.84	5.04
mcf	3.73	2.43
parser	5.46	3.57
perl-diff	4.51	2.94
perl-make	0.41	0.16
perl-535	3.63	2.47
perl-704	2.63	2.04
perl-850	2.00	1.68
perl-957	3.32	2.45
twolf	12.73	8.84
vpr-route	6.95	5.88
INT	6.59	5.05
ALL	5.22	4.00

Table C.8: Branch misprediction rate (in number of misprediction per 100 branches) of the OoO and ReLaSch processors

	64 - 128	128 - 64	256 - 32	512 - 16	1024 - 8
ammp	2.33	2.32	2.29	2.75	3.19
applu	7.29	6.87	5.31	4.31	3.67
art110	1.09	0.86	0.40	0.21	0.13
art470	1.03	0.86	0.42	0.20	0.13
equake	3.75	3.49	3.45	3.65	3.74
facerec	2.76	2.33	1.78	1.21	0.69
fma3d	1.59	0.85	0.73	0.63	0.46
galgel	1.45	1.09	0.61	0.47	0.55
lucas	0.00	0.00	0.00	0.00	0.00
mesa	3.60	3.26	3.07	3.04	3.25
mgrid	6.98	6.64	5.39	5.84	5.74
swim	0.50	0.56	0.60	1.02	0.94
FP	2.70	2.43	2.00	1.94	1.87
bzip-graphic	12.34	12.60	13.34	13.57	13.96
bzip-program	12.39	12.73	13.52	13.99	17.70
bzip-source	15.72	16.24	16.79	17.65	17.49
crafty	9.59	9.32	9.35	9.34	9.13
eon-cook	3.03	2.05	1.21	2.07	2.73
eon-kajiya	9.42	8.62	8.71	9.02	7.83
eon-rush	5.72	4.53	4.20	4.69	5.12
gap	5.61	5.02	4.86	5.20	5.52
gcc-166	6.21	6.06	6.19	6.28	6.14
gcc-200	5.92	5.83	5.81	5.91	6.01
gcc-expr	6.21	6.15	6.19	6.24	6.35
gcc-integ	7.00	6.91	6.93	6.83	6.91
gcc-sci	8.41	8.27	8.31	8.31	8.25
gzip-graphic	5.74	6.01	6.76	7.11	7.17
gzip-log	5.01	5.22	5.63	6.59	7.41
gzip-program	6.04	6.32	6.96	7.99	8.93
gzip-random	5.20	5.68	6.53	6.83	6.59
gzip-source	6.40	6.98	7.84	8.74	8.99
mcf	3.52	3.49	3.73	4.12	5.25
parser	6.44	5.73	5.46	5.55	6.10
perl-diff	5.05	4.66	4.51	4.38	4.51
perl-make	0.97	0.64	0.40	0.91	1.14
perl-535	3.06	3.09	3.63	3.22	3.14
perl-704	2.50	2.45	2.63	2.70	2.59
perl-850	2.03	2.48	2.00	2.82	2.19
perl-957	3.04	3.57	3.32	3.53	3.29
twolf	12.47	12.55	12.73	12.94	12.35
vpr-route	6.77	7.00	6.95	6.98	7.50
INT	6.49	6.44	6.59	6.91	7.15
ALL	5.35	5.23	5.21	5.42	5.57

Table C.9: Branch misprediction rate with different rgroup sizes.

	64 - 128	128 - 64	256 - 32	512 - 16	1024 - 8
ammp	99.90	99.68	99.44	98.58	96.46
applu	97.91	95.75	98.02	98.41	98.09
art110	99.97	99.89	99.79	99.51	99.09
art470	99.97	99.87	99.80	99.53	99.12
equake	99.97	99.97	99.97	99.95	99.93
facerec	99.99	99.97	99.92	99.73	99.27
fma3d	95.93	93.68	97.49	98.01	97.61
galgel	100.00	99.99	99.98	99.93	99.76
lucas	100.00	100.00	99.97	99.94	99.94
mesa	99.59	99.21	98.39	97.27	91.39
mgrid	98.91	97.93	99.73	99.53	99.65
swim	99.88	99.63	99.69	99.48	99.41
FP	99.33	98.80	99.35	99.16	98.31
bzip-graphic	99.98	99.88	99.37	98.60	96.85
bzip-program	99.97	99.86	99.23	97.76	79.28
bzip-source	99.84	99.27	97.77	94.79	88.58
crafty	93.02	88.83	82.74	74.03	62.02
eon-cook	99.22	98.36	98.54	93.73	76.89
eon-kajiya	97.38	94.25	90.13	78.27	56.49
eon-rush	98.41	96.78	94.86	87.89	65.01
gap	98.00	96.42	94.93	90.53	81.82
gcc-166	93.58	90.04	84.72	77.26	65.79
gcc-200	95.08	91.93	87.23	79.87	68.90
gcc-expr	94.56	91.57	87.02	79.76	69.28
gcc-integ	94.18	90.64	85.57	77.88	67.11
gcc-sci	92.90	88.60	82.25	72.01	58.50
gzip-graphic	99.96	99.90	99.75	99.10	98.42
gzip-log	99.93	99.81	99.43	98.41	96.00
gzip-program	99.96	99.92	99.76	98.95	96.72
gzip-random	99.98	99.92	99.80	99.55	98.92
gzip-source	99.95	99.88	99.55	98.40	95.75
mcf	99.98	99.94	99.77	99.13	97.74
parser	99.82	99.58	98.68	96.59	90.68
perl-diff	95.68	92.81	89.21	83.87	75.41
perl-make	99.91	99.72	99.66	99.44	96.59
perl-535	97.38	95.69	93.57	90.17	85.36
perl-704	98.01	96.72	95.30	92.41	88.65
perl-850	98.47	97.33	96.36	94.07	90.41
perl-957	97.41	95.77	93.96	90.30	85.10
twolf	98.37	95.76	91.16	82.56	69.13
vpr-route	99.99	99.96	99.85	99.56	98.43
INT	97.89	96.40	94.29	90.17	82.14
ALL	98.32	97.12	95.81	92.87	86.99

Table C.10: Percentage of instructions executed in Rcache mode with different rgroup sizes..

	64 - 16	128 - 8	256 - 4	512 - 2	1024 - 1
ammp	2.90	2.78	3.06	3.14	2.54
applu	10.14	8.44	8.82	7.34	6.18
art110	0.97	0.84	0.44	0.32	0.40
art470	0.97	0.83	0.46	0.35	0.42
equake	4.04	4.29	4.86	4.15	4.36
facerec	4.65	2.86	1.83	1.67	1.65
fma3d	3.43	1.93	1.30	1.06	1.31
galgel	2.39	1.16	1.09	1.14	1.70
lucas	0.00	0.00	0.00	0.03	0.00
mesa	5.68	5.15	4.37	3.76	3.57
mgrid	7.16	10.64	24.11	17.47	6.80
swim	1.35	0.58	0.58	0.89	0.72
FP	3.64	3.29	4.24	3.44	2.47
bzip-graphic	13.03	13.23	13.44	13.20	12.60
bzip-program	13.79	13.93	14.12	13.73	13.62
bzip-source	15.67	15.89	16.08	16.08	14.86
crafty	11.35	10.31	9.56	8.87	8.17
eon-cook	9.99	7.99	7.08	3.65	3.14
eon-kajiya	12.20	10.86	10.42	7.75	6.65
eon-rush	11.54	9.97	9.13	5.90	4.64
gap	7.17	6.64	6.29	5.96	6.07
gcc-166	7.80	7.24	6.95	6.62	6.40
gcc-200	7.66	7.16	6.83	6.50	6.42
gcc-expr	8.14	7.68	7.39	7.09	7.00
gcc-integ	8.97	8.39	8.07	7.73	7.45
gcc-sci	10.28	9.69	9.43	8.93	8.80
gzip-graphic	6.92	7.01	7.11	7.31	7.30
gzip-log	5.53	5.55	6.01	6.42	6.42
gzip-program	6.77	6.95	7.67	8.58	8.65
gzip-random	6.12	6.11	6.62	6.62	6.42
gzip-source	7.66	8.02	8.56	8.82	8.48
mcf	3.82	3.77	4.39	5.13	5.29
parser	7.24	6.73	6.31	6.50	5.69
perl-diff	6.32	5.63	5.36	4.90	4.90
perl-make	2.34	1.66	1.89	2.37	6.65
perl-535	3.92	3.66	3.67	3.41	3.40
perl-704	3.43	3.18	3.04	2.94	2.96
perl-850	3.04	2.75	3.50	2.37	2.62
perl-957	4.02	3.75	3.61	3.32	3.52
twolf	13.81	12.89	12.16	10.86	9.66
vpr-route	6.91	7.17	7.15	7.29	7.65
INT	8.05	7.64	7.57	7.10	6.98
ALL	6.73	6.33	6.57	6.00	5.63

Table C.11: Branch misspredict rate with different rgroup sizes in a small Rcache.

	64 - 16	128 - 8	256 - 4	512 - 2	1024 - 1
ammp	84.11	91.09	87.47	78.57	68.84
applu	68.87	72.14	79.79	72.51	59.79
art110	99.73	99.40	99.16	98.93	98.47
art470	99.73	99.41	99.18	98.96	98.45
equake	99.64	99.62	99.65	99.59	98.09
facerec	98.01	98.23	96.96	95.21	92.39
fma3d	65.63	70.17	60.35	52.79	52.24
galgel	99.95	99.57	99.31	98.77	98.65
lucas	99.81	99.73	98.80	98.83	99.02
mesa	81.26	75.35	68.22	58.52	45.44
mgrid	84.92	86.46	81.10	85.62	80.29
swim	95.84	97.71	97.60	94.17	94.99
FP	89.79	90.74	88.97	86.04	82.22
bzip-graphic	98.49	95.97	90.39	80.18	65.90
bzip-program	97.35	93.18	84.99	73.54	59.11
bzip-source	95.89	91.30	83.87	74.95	63.26
crafty	61.27	54.26	45.55	36.73	27.94
eon-cook	55.81	45.60	39.41	28.89	19.67
eon-kajiya	54.83	45.23	35.01	21.97	16.01
eon-rush	56.09	46.35	35.56	24.31	17.88
gap	81.82	76.28	70.91	57.77	42.70
gcc-166	71.09	65.61	60.19	54.69	49.86
gcc-200	72.80	66.47	59.29	49.53	38.10
gcc-expr	73.49	67.11	59.34	49.39	39.23
gcc-integ	70.77	64.39	57.19	47.95	38.66
gcc-sci	64.54	58.03	50.93	41.37	31.66
gzip-graphic	99.24	98.74	97.15	94.13	86.41
gzip-log	98.41	96.33	92.28	86.09	77.29
gzip-program	99.13	97.67	93.71	88.55	81.11
gzip-random	99.55	99.38	98.14	95.61	88.26
gzip-source	98.55	96.16	90.34	82.65	74.04
mcf	99.20	98.22	97.23	94.91	90.87
parser	95.06	90.92	85.20	75.45	60.72
perl-diff	77.44	73.04	68.57	65.13	61.55
perl-make	94.06	92.85	91.34	82.56	32.23
perl-535	87.36	84.80	81.80	79.55	75.22
perl-704	89.65	87.52	85.00	83.20	78.73
perl-850	90.12	88.28	85.69	83.78	78.98
perl-957	86.32	83.56	80.26	77.64	72.64
twolf	68.93	56.20	38.76	24.23	14.18
vpr-route	99.59	98.18	95.95	88.64	75.15
INT	83.46	78.99	73.36	65.84	55.62
ALL	85.36	82.51	78.04	71.90	63.60

Table C.12: Percentage of instructions executed in Rcache mode with different rgroup sizes in a small Rcache.

Schedule window size	1024	512	256	128	64
ammp	256	254	239	191	81
applu	256	254	237	175	78
art110	254	232	132	39	17
art470	254	233	138	40	19
equake	256	253	181	79	22
facerec	256	256	256	203	93
fma3d	256	250	227	165	27
galgel	256	254	241	158	75
lucas	256	256	256	159	76
mesa	256	256	256	248	151
mgrid	256	255	254	201	108
swim	256	256	248	166	120
FP	256	251	222	152	72
bzip-graphic	256	256	256	252	202
bzip-program	256	256	256	255	207
bzip-source	256	256	254	239	173
crafty	256	256	256	254	188
eon-cook	252	252	252	245	164
eon-kajiya	251	251	251	242	162
eon-rush	251	251	251	244	167
gap	238	238	238	235	155
gcc-166	254	254	251	245	186
gcc-200	254	254	254	252	190
gcc-expr	255	255	254	251	188
gcc-integ	255	255	255	253	187
gcc-sci	254	254	253	251	182
gzip-graphic	256	256	256	255	183
gzip-log	256	255	253	238	149
gzip-program	256	256	256	248	153
gzip-random	256	256	256	254	178
gzip-source	256	255	254	241	148
mcf	187	120	65	35	16
parser	247	246	244	220	135
perl-diff	250	250	250	248	162
perl-make	246	246	245	229	123
perl-535	246	246	245	232	149
perl-704	246	246	244	231	147
perl-850	246	247	243	231	149
perl-957	246	246	244	229	147
twolf	256	256	255	215	111
vpr-route	256	255	250	183	106
INT	250	247	244	232	157
ALL	251	248	238	208	132

Table C.13: Average number of instructions per created rgroup with different sizes of the `sched` window.

	executed indirect branches
ammp	0
applu	120
art110	0
art470	0
equake	0
facerec	0
fma3d	0
galgel	0
lucas	0
mesa	523,380
mgrid	105
swim	0
FP	43,634
bzip-graphic	0
bzip-program	0
bzip-source	0
crafty	211,124
eon-cook	561,897
eon-kajiya	631,394
eon-rush	574,323
gap	1,342,228
gcc-166	202,461
gcc-200	318,877
gcc-expr	356,727
gcc-integ	383,016
gcc-sci	489,928
gzip-graphic	11
gzip-log	197
gzip-program	6
gzip-random	19
gzip-source	73
mcf	12
parser	10
perl-diff	1,214,101
perl-make	1,554,373
perl-535	964,047
perl-704	980,575
perl-850	912,529
perl-957	982,500
twolf	7,583
vpr-route	0
INT	417,429
ALL	305,290

Table C.14: Number of executed indirect branches in 100M instructions

	L2 misses	Data L1 misses	L2 miss rate	Data L1 miss rate
ammp	478,792	3,922,800	0.19	0.10
applu	1,905,695	6,105,058	0.62	0.15
art110	2,804,513	13,041,451	0.19	0.40
art470	2,870,890	13,165,247	0.20	0.41
equake	2,691,729	12,193,075	0.52	0.26
facerec	365,822	3,114,141	0.73	0.11
fma3d	1,825,569	4,401,353	0.70	0.10
galgel	325,860	3,063,139	0.18	0.07
lucas	1,528,037	2,652,152	0.70	0.12
mesa	100,771	426,252	0.30	0.01
mgrid	963052	1,449,571	0.67	0.04
swim	2,040,543	4,532,540	0.55	0.16
FP	1,491,773	5,672,232	0.46	0.16
bzip-graphic	237,185	657,784	0.37	0.01
bzip-program	108,926	505,748	0.23	0.01
bzip-source	283,041	2,160,881	0.14	0.04
crafty	17,105	507,852	0.04	0.01
eon-cook	1,489	165,923	0.02	0.00
eon-kajiya	1,479	287,585	0.01	0.01
eon-rush	1,487	386,484	0.01	0.01
gap	167,410	703,303	0.29	0.02
gcc-166	276,814	12,187,762	0.08	0.29
gcc-200	67,660	2,441,603	0.06	0.06
gcc-expr	88,165	2,194,467	0.08	0.05
gcc-integ	79,346	3,268,064	0.05	0.07
gcc-sci	109,856	5,551,730	0.05	0.12
gzip-graphic	16,469	1,126,487	0.01	0.03
gzip-log	192,401	2,412,670	0.12	0.08
gzip-program	10,905	1,058,301	0.01	0.05
gzip-random	498,891	5,082,308	0.21	0.15
gzip-source	342,530	3,661,222	0.17	0.13
mcf	8,915,967	24,856,530	0.51	0.60
parser	139,079	1,839,503	0.11	0.05
perl-diff	17,617	560,488	0.03	0.01
perl-make	66,027	92,043	0.73	0.00
perl-535	85,877	917,505	0.12	0.02
perl-704	67,423	744,142	0.11	0.02
perl-850	75,235	765,027	0.15	0.02
perl-957	85,960	868,920	0.11	0.02
twolf	84,619	4,584,773	0.03	0.12
vpr-route	687,430	3,064,092	0.20	0.06
INT	454,514	2,951,900	0.14	0.07
ALL	765,692	3,768,000	0.24	0.10

Table C.15: Number of misses and miss rate in the data L1 cache and the L2 cache.

	OoO			ReLaSch		
	diffsize	store	load	diffsize	store	load
ammp	1,042	1,842	24	11,672	3,871	62
applu	2	91	16,213	0	1,622	0
art110	403	609	21	113	2,847	0
art470	351	609	24	99	3,078	0
equake	0	72	201,238	1	96	2
facerec	0	961	1,356	0	6,155	0
fma3d	0	1,028	7,200	0	9,064	0
galgel	0	16	0	0	0	0
lucas	0	0	0	0	0	0
mesa	16	577	5,010	6,262	1,930	0
mgrid	3	552	11	0	353	0
swim	0	0	0	0	0	0
FP	151	530	19,258	1,512	2,418	5
bzip-graphic	190	2,177	146,056	0	4,623	2,228
bzip-program	44	2,935	4,198	0	19,765	14,067
bzip-source	27	1,094	57,232	15	362,889	91,021
crafty	2065	10,214	1,910	2,955	7,838	650
eon-cook	971	5,711	1,040	3,299	6,577	67
eon-kajiya	3,696	20,291	263	16,461	65,019	1,268
eon-rush	1,668	24,357	2,660	10,456	35,156	146
gap	90	9,325	102,268	83	1,428	1,168
gcc-166	572	7,377	46,963	973	6,972	8,521
gcc-200	721	5,428	42,987	1,154	7,589	5,864
gcc-expr	850	5,479	39,691	2,615	7,777	5,973
gcc-integ	916	5,954	35,436	1,246	9,901	4,467
gcc-sci	1,388	8,781	48,725	3,680	15,397	9,928
gzip-graphic	1	542	60,685	0	4,116	112
gzip-log	39	1,042	5,488	20	719	131
gzip-program	9	528	13,218	0	322	2,227
gzip-random	1	459	59,681	0	6,133	159
gzip-source	19	944	14,527	22	401	4,200
mcf	40	2,156	98,642	42	19,570	40,285
parser	220	1,779	2,386	29	4,554	3,447
perl-diff	992	9,724	21,408	72	13,904	1,293
perl-make	203	3,311	447	25	1,786	29
perl-535	647	7,855	14,845	691	7,557	354
perl-704	469	6,429	13,074	503	5,642	234
perl-850	389	5,673	12,815	353	4,816	361
perl-957	602	7,930	16,277	710	7,772	397
twolf	1,580	10,650	87,701	6,407	23,867	63,474
vpr-route	0	1,732	2,024	0	1,841	1,457
INT	657	6,067	34,023	1,850	23,355	9,412
ALL	506	4,406	29,594	1,749	17,074	6,590

Table C.16: Number of load replays in the ReLaSch and OoO processors. The diffsize column indicates replays of a load in an aliased store-load pair in which bypassing is impossible due to a mismatch is the size of the accesses. The load column indicates replays of a load in an aliased store-load pair that has been executed out-of-order. The store column indicates replays of a load in an aliased store-load pair that has been executed out-of-order.

	OoO			ReLaSch		
	diffsize	store	load	diffsize	store	load
ammp	620	2,327	49	19,451	30,299	84
applu	15	171	32,578	3	3,433	1
art110	224	1,327	49	48	10,442	0
art470	201	58,737	248	46	9,310	0
equake	0	172	126	29	16,758	210,817
facerec	1	1,256	234	46,013	8,945	3,242
fma3d	0	29,218	64,104	2,437	28,821	1,627
galgel	0	84	0	19	203	1,268
lucas	0	0	1	0	0	0
mesa	157	1,979	76,411	18,048	14,457	0
mgrid	8	654	66,437	0	4,022	0
swim	0	0	0	0	0	0
FP	102	7,994	20,020	7,174	10,558	18,087
bzip-graphic	11,075	5,871	13,373	33,675	321,212	104,669
bzip-program	4,329	6,705	6,168	25,906	341,614	190,594
bzip-source	457	3,530	108,870	977	1,194,216	409,697
crafty	5,156	28,594	14,091	8,122	27,727	25,074
eon-cook	5,903	19,798	102,319	271,401	66,038	2,215
eon-kajiya	19,065	31,088	56,959	199,964	309,144	52,645
eon-rush	6,608	21,356	59,230	147,124	209,994	16,410
gap	521	35,424	135,007	9,468	13,083	12,689
gcc-166	1,854	25,977	130,035	8,170	39,208	46,996
gcc-200	1,729	22,792	101,589	10,363	49,744	43,104
gcc-expr	2,030	24,672	82,476	10,933	47,039	40,108
gcc-integ	2,670	32,345	82,526	12,709	65,811	37,230
gcc-sci	3,373	41,743	108,882	17,103	82,361	61,181
gzip-graphic	6	1,477	39,577	6	29,169	4,964
gzip-log	268	3,599	10,097	103	8,214	3,582
gzip-program	26	1,777	31,777	0	33,717	1,2951
gzip-random	2	991	44,867	6	30,246	4,894
gzip-source	131	3,199	45,236	88	42,452	15,710
mcf	261	6,588	202,056	351	139,075	134,048
parser	1,759	3,758	7,020	2,220	34,475	31,475
perl-diff	1,446	30,658	62,537	12,694	217,240	17,529
perl-make	234	6,227	33,545	438	3,303	292
perl-535	1,972	23,411	37,943	3,043	35,709	5,269
perl-704	1,558	18,555	30,396	1,754	29,241	3,025
perl-850	1,181	17,529	35,751	1,528	22,714	2,859
perl-957	1,796	23,438	38,826	2,467	33,734	4,258
twolf	2,328	28,594	180,291	9,888	118,854	10,9701
vpr-route	209	4,648	28,458	112	23,485	28,414
INT	2,784	1,6941	65,354	28,236	127,458	50,771
ALL	1,979	14,257	51,753	21,918	92,388	40,966

Table C.17: Number of load replays in the ReLaSch and OoO processors with an issue-width of 16 integer and 8 floating point instructions. The diffsize column indicates replays of a load in an aliased store-load pair in which bypassing is impossible due to a mismatch is the size of the accesses. The load column indicates replays of a load in an aliased store-load pair that has been executed out-of-order. The store column indicates replays of a load in an aliased store-load pair that has been executed out-of-order.

	default			ignore		
	diffsize	store	load	diffsize	store	load
ammp	11,672	3,871	62	11,257	5,448	49
applu	0	1,622	0	1	2,568	0
art110	113	2,847	0	113	132,714	0
art470	99	3,078	0	99	134,408	0
equake	1	96	2	0	9,874	2
facerec	0	6,155	0	0	17,113	0
fma3d	0	9,064	0	0	123,573	0
galgel	0	0	0	0	1,164	0
lucas	0	0	0	0	0	0
mesa	6,262	1,930	0	6,291	12,926	0
mgrid	0	353	0	0	9,159	0
swim	0	0	0	0	0	0
FP	1,512	2,418	5	1,480	37,412	4
bzip-graphic	0	4,623	2,228	1	6,933	2,907
bzip-program	0	19,765	14,067	0	19,682	11,976
bzip-source	15	362,889	91,021	14	439,470	93,556
crafty	2,955	7,838	650	3,200	16,112	681
eon-cook	3,299	6,577	67	6,330	185,230	25
eon-kajiya	16,461	65,019	1,268	17,288	233,681	1,273
eon-rush	10,456	35,156	146	4,875	191,344	124
gap	83	1,428	1,168	85	320,242	1,905
gcc-166	973	6,972	8,521	966	13,282	8,312
gcc-200	1,154	7,589	5,864	991	25,528	5,671
gcc-expr	2,615	7,777	5,973	2,720	27,454	5,831
gcc-integ	1,246	9,901	4,467	1,096	30,602	4,895
gcc-sci	3,680	15,397	9,928	3,700	32,742	9,471
gzip-graphic	0	4,116	112	1	9,778	126
gzip-log	20	719	131	19	10,355	189
gzip-program	0	322	2,227	0	6,852	1,731
gzip-random	0	6,133	159	0	13,388	174
gzip-source	22	401	4,200	30	14,086	4,212
mcf	42	19,570	40,285	39	462,005	36,193
parser	29	4,554	3,447	18	92,424	3,728
perl-diff	72	13,904	1,293	82	119,888	1,845
perl-make	25	1,786	29	32	325,247	16
perl-535	691	7,557	354	661	86,255	720
perl-704	503	5,642	234	461	71,497	283
perl-850	353	4,816	361	435	58,053	342
perl-957	710	7,772	397	659	87,434	699
twolf	6407	23,867	63,474	6,798	204,817	59,406
vpr-route	0	1,841	1,457	0	315,290	1,613
INT	1,850	23,355	9,412	1,804	122,131	9,211
ALL	1,749	17,074	6,590	1,707	96,715	6,449

Table C.18: Number of load replays in ReLaSch when the boundaries of the issue-group are ignored and in the default configuration. The diffsize column indicates replays of a load in an aliased store-load pair in which bypassing is impossible due to a mismatch is the size of the accesses. The load column indicates replays of a load in an aliased store-load pair that has been executed out-of-order. The store column indicates replays of a load in an aliased store-load pair that has been executed out-of-order.

	INT				FP			
	25.00%	50.00%	75.00%	90.00%	25.00%	50.00%	75.00%	90.00%
ammp	3	6	12	18	5	10	16	20
applu	3	6	10	15	7	14	22	27
art110	1	2	3	5	4	9	17	22
art470	1	2	3	5	5	9	16	22
equake	2	5	10	16	3	7	12	18
facerec	4	6	13	19	3	5	8	11
fma3d	4	8	16	22	5	12	20	26
galgel	4	8	13	18	2	4	7	9
lucas	2	4	7	12	4	8	14	20
mesa	5	11	19	25	3	8	15	22
mgrid	4	9	17	22	6	14	22	27
swim	2	4	6	8	5	12	18	24
bzip-graphic	4	9	15	20				
bzip-program	4	9	16	21				
bzip-source	4	9	16	21				
crafty	3	9	17	23				
eon-cook	3	7	13	19	2	5	9	14
eon-kajiya	3	7	13	20	2	5	10	15
eon-rush	2	7	13	19	2	5	10	15
gap	4	9	15	21				
gcc-166	3	8	14	20				
gcc-200	5	10	17	23				
gcc-expr	5	10	17	23				
gcc-integ	4	10	17	23				
gcc-sci	4	10	16	22				
gzip-graphic	4	7	16	20				
gzip-log	2	4	7	14				
gzip-program	2	4	6	13				
gzip-random	3	7	12	18				
gzip-source	2	5	9	16				
mcf	2	5	9	13				
parser	2	5	12	18				
perl-diff	4	8	15	21				
perl-make	3	7	13	19	2	3	5	8
perl-535	3	7	13	19				
perl-704	3	7	13	19				
perl-850	3	7	13	19				
perl-957	3	7	13	20				
twolf	3	8	14	17	1	2	3	4
vpr-route	2	6	12	19	1	1	2	3

Table C.19: Number of logical registers that accumulate the indicated percentage of writes. Lower numbers indicate that writes are concentrated in a small number of logical registers. Results are separated for the integer and floating point register files. Whenever an INT benchmark has less than 1M writes in the floating point register file, only its values for the integer register file are shown.

Bibliography

[1] H.Q. Le, W.J. Starke, J.S. Filds, F.P. O'Conell, D.Q. Nguyen, B.J. Ronchettti, W.M. Sauer, E.M. Schwarz, and M.T Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Devevlopment*, 51(6):639–662, November 2007.

[2] R. Kalla and B. Sinharoy. Power7: IBM's next generation power microprocessor. In *IEEE Symposium on High-Performance Chips (Hot Chips 21)*, 2009.

[3] IBM. Power systems facts and features. POWER7 blades and servers. POB03022-USEN-06, October 2010.

[4] IBM. POWER6 systems facts and features. POB03004-USEN-14, April 2010.

[5] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, March/April 2005.

[6] S. Patel, S. Phillips, and A. Strong. Sun's next-generation multi-threaded processor - rainbow falls: Sun's next generation CMT processor. In *IEEE Symposium on High-Performance Chips (Hot Chips 21)*, 2009.

[7] S. Palacharla, J. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, pages 206–218, 1997.

[8] R. Nair and M.E. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, pages 13–25, 1997.

[9] S.J. Patel and S.S. Lumetta. rePLay: A hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):590–608, jun 2001.

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

[11] S. Narayanasamy, Y. Hu, S. Sair, and B. Calder. Creating converged trace schedules using string matching. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 210–221, 2004.

[12] O. Palomar, T. Juan, and J.J. Navarro. A case for simplifying the dynamic-scheduling engine of current microprocessors. In *XIV Jornadas de Paralelismo*, pages 369–374, 2003.

[13] O. Palomar, T. Juan, and J.J. Navarro. Reusing cached schedules in an out-of-order processor with in-order issue logic. In *Proceedings of the 27th International Conference on Computer Design (ICCD'09)*, pages 246–253, 2009.

[14] O. Palomar, T. Juan, and J.J. Navarro. A configurable cache of schedules. In *6th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems ACACES 2010 (Poster session)*, 2010.

[15] Compaq Computer Corporation. *Compiler writer's guide for the Alpha 21264*, June 1999. EC-RJ66A-TE.

[16] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, pages 142–153, 1998.

[17] T.C. Mowry and C.K. Luk. Understanding why correlation profiling improves the predictability of data cache misses in nonnumeric applications. *IEEE Transactions on Computers*, 49(4):369–384, April 2000.

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

[19] James E. Smith and Andrew R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th International Symposium on Computer Architecture (ISCA)*, pages 36–44, 1985.

[20] R. Desikan, D.C. Burger, S.W. Keckler, and T.M. Austin. Sim-alpha: a validated, execution-driven Alpha 21264 simulator. Technical Report TR-01-23, Dep. of Computer Sciences, UT-Austin, 2001.

[21] Compaq. Compaq AlphaServer DS10L Systems, Technical Summary, 2003.

[22] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2), May 2003.

[23] P.Y. Chang, E. Hao, and Y.N. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, pages 274–283, 1997.

[24] SPEC CPU 2000 web page. http://www.spec.org/cpu2000/ (14 March 2011).

[25] 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)*, pages 45–57, Oct 2002.

[26] S. McFarling. Combining branch predictors. Technical Report Tech. Note TN-36, Compaq Computer Corp. Western Research Laboratory, 1993.

[27] A. Ferreira de Souza and P. Rounce. Dynamically scheduling the trace produced during program execution into VLIW instructions. In *Proceedings of the Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, pages 248–257, 1999.

[28] S. Jee and K. Palaniappan. Dynamically scheduling vliw instructions with dependency information. In *INTERACT '02: Proceedings of the Sixth Annual Workshop on Interaction between Compilers and Computer Architectures*, page 15, Washington, DC, USA, 2002. IEEE Computer Society.

[29] S.J. Patel, T. Tung, S. Bose, and M.M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. In *Proceedings of the 33th International Symposium on Microarchitecture (MICRO)*, pages 303–313, 2000.

[30] B. Fahs, S. Bose, M.M. Crum, B. Slechta, F. Spadini, T. Tung, S.J. Patel, and S.S. Lumetta. Performance characterization of a hardware mechanism for dynamic optimization. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO)*, pages 16–27, 2001.

[31] B. Slechta, D. Crowe, B. Fahs, M. Fertig, G.A. Muthler, J. Quek, F. Spadini, S.J. Patel, and S. Lumetta. Dynamic optimization of micro-operations. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 165–176, 2003.

[32] B. Fahs, T.M. Rafacz, S.J. Patel, and S.S. Lumetta. Continuous optimization. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pages 86–97, 2005.

[33] F. Spadini, B. Fahs, S. Patel, and S.S. Lumetta. Improving quasi-dynamic schedules through region slip. pages 149–158, 2003.

[34] R. Rosner, Y. Almog, M. Moffie, N. Schwartz, and A. Mendelson. Power awareness through selective dynamically optimized traces. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*, pages 162–175, 2004.

[35] Y. Almog, R. Rosner, N. Schwartz, and A. Schmorak. Specialized dynamic optimizations for high-performance energy-efficient microarchitecture. In *Proceedings of the 2nd International Symposium on Code Generation an Optimization (CGO)*, pages 137–150, 2004.

[36] E. Talpes and D. Marculescu. Increased scalability and power efficiency by using multiple speed pipelines. In *Proceedings of the 32th International Symposium on Computer Architecture (ISCA)*, pages 310–321, 2005.

[37] E. Talpes and D. Marculescu. Power reduction through work reuse. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 340–345, 2001.

[38] B. Black and J.P. Shen. Turboscalar: A high frequency high ipc microarchitecture. In *Workshop on Complexity-Effective Design - Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*, 2001.

[39] S. Banerjia, S.W. Sathaye, K.N. Menezes, and T.M. Conte. MPS: Miss-Path Scheduling for Multiple-Issue Processors. *IEEE Transactions on Computers*, 47(12):1382–1397, dec 1998.

[40] M.C. Merten, A.R. Trick, E.M. Nystrom, and R.D. Barnes. A hardware mechanism for dynamic extraction and relayout of program hot spots. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, pages 59–70, 2000.

[41] M.C. Merten, A.R. Trick, R.D. Barnes, E.M. Nystrom, C.N. George, J.C. Gyllenhaal, and W.W. Hwu. An architecture framework for runtime optimization. *IEEE Transactions on Computers*, 50(6):567–589, June 2001.

[42] J.C. Dehnert, B.K. Grant, J.P. Banning, R.Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing^{TM}Software: Using speculation, recovery and adaptive retranslation to address real-life challenges. In *Proceedings of the 1st International Symposium on Code Generation an Optimization (CGO)*, pages 15–24, 2003.

[43] M.T. Yourst and K. Ghose. Incremental commit groups for non-atomic trace processing. In *Proceedings of the 38th International Symposium on Microarchitecture (MICRO)*, pages 67–80, 2005.

[44] S. Vajpeyam and T. Mitra. Improving superscalar instruction dispath and issue by exploiting dynamic code sequences. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, pages 1–12, 1997.

[45] Y. Chou and J.P. Shen. Instruction path coprocessor. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, pages 270–281, 2000.

[46] A. García, O.J. Santana, E. Fernández, P. Medina, and M. Valero. LPA: A first approach to the loop processor architecture. In *Proceedings of the International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, pages 273–287, 2008.

[47] J.S. Hu, N. Vijaykrishnan, S. Kim, M. Kandemir, and M.J. Irwin. Scheduling reusable instructions for power reduction. In *Proceedings of the Design, Automation and Test in Europe Conference*, pages 148–153, 2004.

[48] C. Yang and A. Orailoglu. Power-efficient instruction delivery through trace reuse. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 192–201, 2006.

[49] F. Pratas, G. Gaydadjiev, M. Berekovic, L. Sousa, and S. Kaxiras. Low power microarchitecture with instruction reuse. In *Proceedings of the Conference on Computing Frontiers*, pages 149–158, 2008.

[50] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th International Conference on Supercomputing*, pages 68–75, 1997.

[51] R.D. Barnes, E.M. Nystrom, J.W. Sias, S.J. Patel, N.Navarro, and W.W. Hwu. Beating in-order stalls with "flea-flicker" two-pass pipelining. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, pages 387–398, 2003.

[52] R.D. Barnes, J.W. Sias, E.M. Nystrom, S.J. Patel, N. Navarro, and W.W. Hwu. Beating in-order stalls with "flea-flicker" two-pass pipelining. *IEEE Trans. Computers*, 55(1):18–33, 2006.

[53] R.D. Barnes, S. Ryoo, and W.W. Hwu. "flea-flicker" multipass pipelining: An alternative to the high-power out-of-order offense. In *Proceedings of the 38th International Symposium on Microarchitecture (MICRO)*, pages 319–330, 2005.

[54] R.D. Barnes, S.Ryoo, and W.W. Hwu. Tolerating cache-miss latency with multipass pipelines. *IEEE Micro*, 26(1):40–47, January-February 2006.

[55] T.E. Erhart and S.J. Patel. Reducing the scheduling critical cycle using wakeup prediction. In *Proceedings of the 10th International Conference on High Performance Computer Architecture (HPCA)*, pages 222–231, 2004.

[56] D. Ernst, A. Hamel, and T. Austin. Cyclone: A broadcast-free dynamic instruction scheduler with selective replay. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, pages 253–262, 2003.

[57] P. Michaud and A. Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 27–36, 2001.

[58] K.M. Hazelwood and D. Brooks. Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 326–331, 2004.

[59] Compaq Computer Corporation. *Alpha Architecture Handbook*, October 1998. EC-QD2KC-TE.

[60] R.E. Kessler, E.J. McLellan, and D.A. Webb. The Alpha 21264 microprocessor architecture. In *Proceedings of the 16th ICCD*, pages 90–95, 1998.

[61] T. Fischer and D. Leibholz. Design tradeoffs in stall-control circuits for 600MHz instruction queues. In *Proceedings of the International Solid-State Circuits Conference*, pages 232–233, 1998.

[62] B.A. Gieseke, R.L. Allmon, D.W. Bailey, B.J. Benschneider, S.M. Britton, J.D. Clouser, H.R. Fair III, J.A. Farrell, M.K. Gowan, C.L. Houghton, J.B. Keller, T.H. Lee, D.L. Leibholz, S.C. Lowell, M.D. Matson, R.J. Matthew, V. Peng, M.D. Quinn, D.A. Priore, M.J. Smith, and K.E. Wilcox. A 600MHz superscalar RISC microprocessor with out-of-order execution. In *Proceedings of the International Solid-State Circuits Conference*, pages 176–177, 1997.

[63] M. Matson, D. Bailey, S. Bell, L. Biro, S. Butler, J. Clouser, J. Farrell, M. Gowan, D. Priore, and K. Wilcox. Circuit implementation of a 600MHz superscalar RISC microprocessor. In *Proceedings of the International Conference on Circuit Design*, pages 104–110, 1998.