# Recovery Mechanism for Latency Misprediction

Enric Morancho, José María Llabería and Àngel Olivé

*Departament d'Arquitectura de Computadors*
*Universitat Politècnica de Catalunya, Barcelona, Spain*
*{enricm,llaberia,angel}@ac.upc.es*

## Abstract

*Signalling result availability from the functional units to the instruction scheduler can increase the cycle time and/or the effective latency of the instructions. The knowledge of all instruction latencies would allow the instruction scheduler to operate without the need of external signalling. However, the latency of some instructions is unknown; but, the scheduler can optimistically predict the latency of these instructions and issue speculatively their dependent instructions.*

*Although prediction techniques have great performance potential, their gain can vanish due to misprediction handling. For instance, holding speculatively scheduled instructions in the issue queue reduces its capacity to look-ahead for independent instructions.*

*This paper evaluates a recovery mechanism for latency mispredictions that retains the speculatively issued instructions in a structure apart from the issue queue: the recovery buffer. When data becomes available after a latency misprediction, the dependent instructions will be re-issued from the recovery buffer. Moreover, in order to simplify the re-issue logic of the recovery buffer, the instructions will be recorded in issue order.*

*On mispredictions, the recovery buffer increases the effective capacity of the issue queue to hold instructions waiting for operands. Our evaluations in integer benchmarks show that the recovery-buffer mechanism reduces issue-queue size requirements about 20-25%. Also, this mechanism is less sensitive to the verification delay than the recovery mechanism that retains the instructions in the issue queue.*

## 1. Introduction

In dynamically-scheduled superscalar processors, instructions wait in the issue queue for the availability of operands and functional units [8][13][16][17]. To issue instructions out-of-order to the functional units, the issue queue has two components: a) *wakeup logic* and b) *select logic*. The *wakeup logic* keeps monitoring the dependencies among the instructions in the issue queue and, when the operands of a queued instruction become available, this logic will mark the instruction as ready. The *select logic* selects which instructions will be issued to the functional units on the next cycle.

Considering only instructions with known latency, a mechanism that counts latencies and wakes-up dependent instructions can be included in the issue logic. However, to deal with instructions with unknown latency, the functional units must send a signal to the *wakeup logic*; then, with high clock rates, wire delays may forbid back-to-back execution of dependent instructions [1][10]. Therefore, a valuable mechanism that deals with unknown-latency instructions is latency prediction. If the predicted latency is optimistic, instructions dependent on the predicted instruction can be scheduled speculatively; however, a recovery mechanism is needed on mispredictions to nullify and to re-issue the speculatively issued instructions.

A simple alternative is squashing; all the instructions younger than the mispredicted instruction are flushed from the processor, and these instructions are later re-fetched from the instruction cache. This process is identical to the branch-misprediction recovery mechanism.

However, to reduce the penalty of the recovery process, finer recovery mechanisms are needed. For instance, they can benefit from the fact that the instructions that must be re-issued have already been fetched. In this case, the mechanism must provide storage to keep, until the prediction is verified, the speculatively issued instructions.

A conventional solution is to maintain the chain of speculatively issued instructions (and probably other independent instructions) in the issue queue [9][14] until latency prediction is verified. However, unless increasing the issue-queue size, processor performance can suffer because this solution reduces the capacity of the scheduler to look-ahead for independent instructions. On the other hand, increasing the issue-queue size will be limited by future wire delays [1]. Then, as the issue queue is in the critical path, this solution is a limited alternative.

Another approach consists in extracting the issued instructions from the issue queue after being issued, and storing them in a recovery buffer, apart from the issue queue, until latency prediction is verified. Then, new instructions can be inserted in the freed issue-queue entries and the look-ahead capacity of the issue queue is maintained. On a misprediction, the re-issue is performed from the recovery buffer and, to reduce the complexity of the re-issue logic of the recovery buffer versus that of the issue queue, the recovery buffer maintains the relative issue cycles between the instructions. Moreover, on mispredictions, the recovery buffer increases the amount of in-flight instructions because it holds issued instructions dependent on latency-mispredicted instructions.

A scope where this work can be applied is load-use delay. Load instructions have unknown latency because it depends on the location of the data in the memory hierarchy. Moreover, tag-checking is in the critical path to wake up the dependent instructions; also, in first-level caches, data-array contents can be obtained before tag-checking result [12]. Consequently, waiting until tag-checking to wake up the dependent instructions can reduce the performance. For instance, in a 4-way processor executing integer benchmarks, performance degradation is about 6% when load-use delay increases from 3 to 4 cycles.

This paper applies latency prediction in the instruction scheduler, and evaluates the performance of the recovery-buffer mechanism versus keeping the speculatively issued instructions in the issue queue. Moreover, two issued-instruc-

tion nullification policies are evaluated: a) nullifying all the instructions potentially dependent on the mispredicted instruction, b) nullifying only the chain of instructions dependent on the mispredicted instruction.

The evaluations are focused on load-latency prediction [4][5][7][9] and, as high first-level-cache hit rates are expected, the prediction is that all load instructions are hits in data cache; as a side effect, cache tag-checking can be moved out of the critical path. Evaluations show that the recovery-buffer mechanism outperforms the conventional recovery mechanism. And, for integer benchmarks, the recovery-buffer mechanism allows a issue-queue-size reduction about 20-25% without performance decrease.

The rest of this paper is organized as follows. Section 2 characterizes the recovery process for latency mispredictions. Section 3 outlines the recovery process when speculative instructions are retained in the issue queue. In Section 4 the recovery process using the recovery buffer is designed. Section 5 gives performance results of the recovery-buffer mechanism compared with the conventional recovery mechanism. Finally, Section 6 presents the conclusions of this work.

# 2. Background

Figure 1 shows two cases where latency prediction is profitable (stages between instruction-fetch stage and rename stage are not shown as they are not relevant to this work). In Figure 1.a, as tag-checking is performed before waking-up the dependent instruction, it increases load-use delay if data-array access is a cache hit. Using latency prediction, tag-checking can be decoupled from data availability, and thus load-use delay is reduced by one cycle. Other pipeline design (Figure 1.b), includes a stage between the issue queue and the functional units. In this case, to support back-to-back execution of dependent instructions, *wakeup logic* must wake-up the dependent instructions before tag-checking.
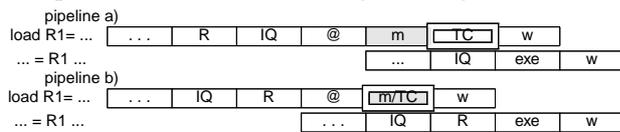


**Figure 1:** Pipeline designs without latency prediction. Stages: read registers (R), issue queue (IQ), compute address (@), execute (exe), data-array access (m), tag-checking (TC), write registers (w). Pipelines: a) Registers are read before IQ stage; tag-checking is performed one cycle after data availability. The issue queue stores the values of the registers or a functional-unit identifier. b) Registers are read after IQ stage; tag-checking and data availability performed on the same cycle.

In both cases, predicting hit latency is useful to execute the chain of dependent instructions without delay if memory access is a cache hit. Without latency prediction, load-use delay is three cycles; with latency prediction, load-use delay is two cycles. However, a recovery mechanism is required on a latency misprediction because the dependent instructions use incorrect data in their computations. From now, the pipeline design b) of Figure 1 is used on the examples of this paper.

For instance, latency prediction is also useful in: a) way prediction in associative caches, b) bank prediction in multi-banked caches, c) designs where the physical registers are read after the issue stage in a pipelined way, d) first-level cache with ECC correction logic, e) pipelining the scheduling logic [15].

Figure 2 is used to introduce the terminology of this paper. Assume that on cycle 1 a load instruction is issued with a data-cache latency of two cycles and a tag-check latency of three cycles. When hit latency is predicted, the speculative instructions potentially dependent on the load instruction, directly or through a dependent chain, are issued on cycles 3, 4 and 5 (shadowed instructions). We name these cycles *speculative window* (SW); that is, the cycle range from waking-up the first potential dependent instruction until tag-checking. We name *verification delay* to the duration of the speculative window (three cycles in the example). Also, we name *independent window* (IW) to the cycle range between issuing the load and the beginning of the speculative-window; the instructions issued during the independent window are independent on the load. An instruction is inside a window if it is issued during a cycle of the window. A *wave of instructions* represents all the instructions issued during a cycle.
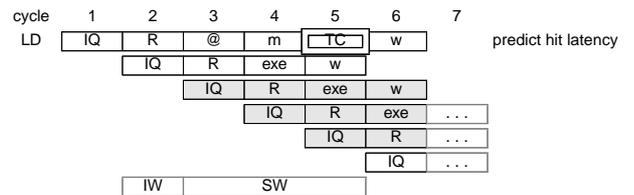


**Figure 2:** Instruction flow after issuing a load instruction (LD) on cycle 1 with predicted hit latency. IW is the Independent Window, SW is the Speculative Window.

## 2.1. Recovery on a mispredicted latency

Known scopes where the processor executes instructions speculatively are branch prediction and memory-dependence prediction.

Branch prediction is used to speculatively execute predicted-path instructions. These instructions may be issued before issuing the predicted branch instruction. The prediction is performed by the fetch unit and the verification is performed by the branch instruction when it is executed. On a misprediction, wrong-path instructions are squashed and the fetch unit is redirected to the new path.

Memory-dependence prediction is used to execute a load instruction and its dependent instructions before knowing the addresses accessed by older store instructions. The speculative instructions are issued after issuing the predicted load instruction. The prediction is performed by a load instruction and the verification by an older store instruction. On a misprediction, the instructions that must be nullified are the same that must be re-executed.

Usually, these predictions rely on a general recovery mechanism that flushes-out the entire instruction pipeline [8][9].

Unlike the previous prediction types, latency prediction shows **all** the following characteristics:
- The verification of the prediction is performed by the predicted instruction.
- The speculative instructions are issued after issuing the predicted instruction.
- When a latency-predicted instruction is issued, the cycles where the dependent instructions can be speculatively issued are known (the speculative window).
- On a misprediction, the instructions that must be re-executed are the same that are nullified.

These characteristics allow the design of a simple recovery mechanism that is slightly aggressive from a performance point of view[1]. When a misprediction is detected, all instructions issued inside the speculative window are nullified and their dependent instructions are slept. After that, the nullified instructions are re-issued in proper time: the instructions independent on the latency-predicted instruction are re-issued on next cycles, and the dependent instructions will be re-issued when data is available. In next sections we analyse two structures for keeping the instructions while predicted latency is verified: the issue queue and the recovery buffer.

The previous approach losses on every misprediction a number of cycles equal to the speculative-window size. Figure 3 shows an example where 3 cycles are lost on a misprediction. A better mechanism is also evaluated in this paper; the mechanism only nullifies the instructions dependent on the mispredicted instructions.
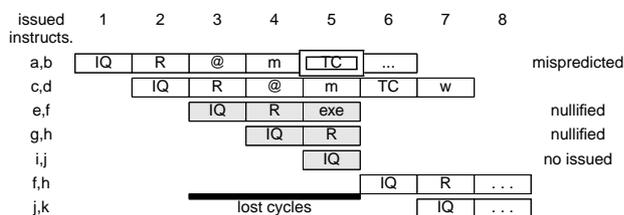


**Figure 3:** Load instruction **a** is a latency-mispredicted instruction. On cycle 5, the misprediction is detected, instructions **i** and **j** are not issued and do not wakeup their dependent instructions; also instructions **e**, **f**, **g**, **h** are nullified and their dependent instructions (**i**, **k**, **n** and **m**) are slept in the issue queue. On cycle 6, nullified instructions **f** and **h** are re-issued and their dependent instructions are waken-up. On cycle 7, instruction **j** is re-issued and instruction **k** is issued. Instructions dependent on load instruction **a** are re-issued (not shown) once the memory hierarchy provides data.
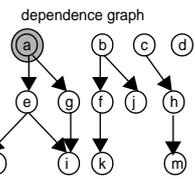
Our evaluations assume that no instruction is issued on cycles where latency mispredictions are detected; that is, on the last cycle of the speculative window of a mispredicted instruction, the instructions selected to be issued are not issued (cycle 5 in Figure 3).

In summary, this paper presents evaluations of two approaches:
- A conservative approach, named non-selective, that assumes that all issued instructions are inside a potential speculative

---

1. Alpha 21264 processor handles this situation with a minirestart mechanism. All integer instructions issued during the speculative window are "pulled back" into the issue queue to be re-issued later [9].

window. On mispredictions, it nullifies all the instructions inside the speculative window.
- An aggressive approach, named selective, that considers only the instructions dependent on latency-predicted instructions. On mispredictions, it nullifies only the instructions of the speculative window dependent on mispredicted instructions.

These approaches represent two extreme cases, although several intermediate approaches could be designed.

## 2.2. Base Pipeline and Issue Queue

**Base Pipeline** (Figure 4). After fetching the instructions, they are decoded and renamed. A renamed instruction resides in the issue queue until its source operands have been computed and it has been selected for execution. After it has been executed, it is marked in the ROB (reorder buffer) as complete. After that, it is committed when all previous instructions in program order have been marked as complete and have been committed. When an instruction is committed, the architectural state is updated with the speculative state and resources are freed. The ROB records all in-flight instructions.
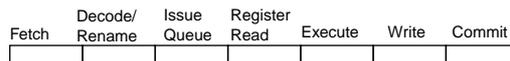


**Figure 4:** Base processor pipeline.

**Base Issue Queue.** The issue queue includes a dependence matrix (Figure 5) to track dependencies among instructions. The matrix has as many rows as the number of instructions analysed simultaneously for scheduling, and as many columns as the number of physical registers (registers for short).

The columns are wires that cross all rows and each row includes a bit for each column. Each column marks the data availability of a register. Each column is set by a count-down latency counter or by a shift register connected to the column.

When an instruction is inserted in an issue-queue entry (row), the bits related to the source operands of the instruction are set. Also, the latency counter related to the destination register is initialised to the instruction latency.

Each crosspoint of the dependence matrix contains a logical circuit that determines if the required source operand is ready. For each row, the outputs of these logical circuits are used to compute a *ready bit* that indicates if the instruction is ready to be selected by the *select logic*. *Ready bits* are evaluated every cycle.
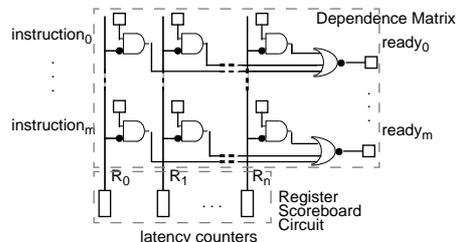


**Figure 5:** Dependence-matrix structure.

When an instruction is issued, the latency counter related to its destination register is decreased on every cycle. Then, when latency lapses, the column will be set to mark the availability of the result.

# 3. Keeping issued instructions in the issue queue

In regular operation, without latency prediction, instructions are removed from the issue queue as soon as they are issued. However, with latency prediction, some instructions must be re-issued when a misprediction is detected.

To perform a fast recovery, a possibility is keeping each issued instruction in the issue queue until the instruction is known to be unnecessary for a recovery action [9]. During these cycles, the issued instruction should be nonvisible to the *select logic*. Then, a *no-request bit* is added to each dependence-matrix row; the bit is set when its instruction is issued.

When an issued instruction is known not to be needed in a latency-mispredicted recovery action, it can be removed from the issue queue. Otherwise, on a misprediction, it must be nullified. It is made visible again to the *select logic*, and its destination register is set as not available to delay the issue of its dependent instructions until it has been re-issued.

Two control circuits perform these operations: the *removal circuit* and the *register-scoreboard circuit*. Figure 6 shows the interface between them and other issue-queue elements.
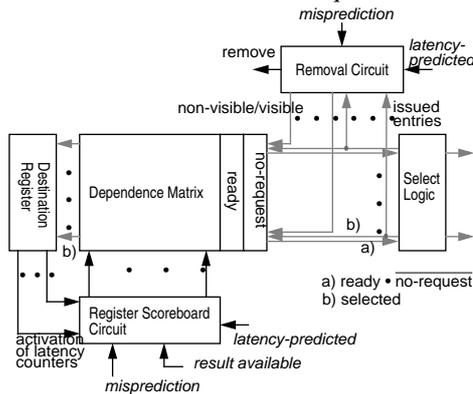


**Figure 6:** Issue-Queue structure.

The *removal circuit* is used to remove issued instructions from the issue queue (if it is safe to remove them), as well as to make them visible again (if they must be re-issued).

The *register-scoreboard circuit* is used for activating latency counters (for each issued instruction) or for unsetting columns (for each nullified instruction). As *ready bits* are re-evaluated every cycle, nullified instructions will re-evaluate their ready bits and all instructions dependent on the nullified instructions will be slept.

On every cycle, the *removal circuit* is being aware of the issued instructions and the *register-scoreboard circuit* is notified of their destination registers. Both circuits are also notified of which issued instructions are latency predicted and of the results of the prediction verifications. Moreover, the register-scoreboard circuit keeps track of the mispredicted-data availability.

This paper presents the evaluation of two extreme approaches that differ on two aspects:
• When can an instruction be removed from the issue queue.
• Which instructions are nullified on a misprediction.

## 3.1. Non-selective nullification

The simplest recovery mechanism conservatively assumes that all the instructions are issued inside a potential speculative window and, on a misprediction, they are dependent on a latency-predicted instruction. Then, all issued instructions are retained in the issue queue during a number of cycles equal to the verification delay minus one. After that, if no latency-misprediction is detected, the instructions can be removed from the issue queue. Otherwise, on a misprediction, all the instructions issued on the speculative window of the mispredicted instruction must be nullified and re-issued (for instance, instructions **e**, **f**, **g** and **h** in Figure 3). Consequently, the columns related to these instructions must be reset and the issue-queue entries must be made visible again.

To perform both actions, the *register scoreboard circuit* and the *removal circuit* respectively track the destination registers and the issue-queue entries of the instructions issued each cycle. The information related to a cycle can be discarded as soon as the wave is outside any speculative window.

On a misprediction, both circuits aggregate the information related to the speculative window of the mispredicted instruction. After that, the *register scoreboard circuit* clears the columns related to the destination registers of the instructions to be nullified, and the *removal circuit* unsets the no-request bits of the issue-queue entries of these instructions. Moreover, the *register scoreboard circuit* also clears the destination register of the mispredicted instruction.

Among nullified instructions, there may be instructions independent on the mispredicted instruction. These instructions will immediately compete to be selected for issue because their source operands are still available (for instance, instructions **f** and **h** in Figure 3).

A possible implementation of the tracking mechanism uses bit vectors; every cycle, a bit vector is allocated in every circuit. Every bit vector of the *register scoreboard circuit* has as many bits as physical registers; setting its i-th bit indicates that the instruction that produces the i-th register has been issued on the related cycle. Every bit vector of the *removal circuit* has as many bits as issue-queue entries; setting its j-th bit indicates that the instruction allocated in the j-th issue-queue entry has been issued on the related cycle. The amount of bit vectors of each circuit is equal to the verification delay minus one.

On a misprediction, both circuits aggregate the information by OR-ing the bits vectors related to the cycles of the speculative window. The resultant bit vectors are used to clear the columns and the no-request bits on a single cycle.

## 3.2. Selective nullification

The previous mechanism is simple but conservative because it assumes that all the issued instructions are inside a potential speculative window and, on a misprediction, independent instructions inside this speculative window are also nullified. A more selective mechanism keeps in the issue queue only instructions dependent on a latency-predicted instruction not yet verified, and nullifies just these instructions on a misprediction. For instance, in Figure 3, this mechanism does not

nullify instructions **f** and **h**, and retains in the issue queue only the instructions **e** and **g**.

We suppose that the cycle following the issue of an instruction is used to compute the dependence of the issued instructions on a latency-predicted instruction not yet verified. Then, independent instructions are removed from the issue queue one cycle after issuing them, and dependent instructions are kept in the issue queue while the prediction is not yet verified.

*Register-scoreboard circuit* tracks dependencies and notifies them (not shown in Figure 6) to the *removal circuit* to track the issue-queue entries of the dependent instructions. To do so, each circuit uses bit vectors with the same size that in the previous subsection, but managed differently. When a latency-predicted instruction is issued, a bit vector is allocated in each control circuit; bit vectors of the *register-scoreboard circuit* are initialised by setting the destination register of the instruction. These bit vectors are updated in successive cycles with the destination registers and the issue-queue entry numbers of the dependent issued instructions.

*Register-scoreboard circuit* tracks dependencies using as inputs the identifiers of the source operands of the issued instructions. If any source operand is marked in the bit vector of a latency-predicted instruction, the control circuit sets the destination register of the issued instruction in this bit vector. Then, a bit vector shows the registers dependent on the related latency-predicted instruction.

On mispredictions, each circuit uses the bit vector related to the mispredicted instruction. Bit vectors are freed as in the non-selective mechanism. Thus, the amount of bit vectors of each circuit is equal to the issue-width of latency predicted instructions times the verification delay minus one.

# 4. Keeping issued instructions in the recovery buffer

Keeping speculatively-issued instructions in the issue-queue reduces its capacity to look-ahead for independent instructions. This section develops a recovery mechanism that keeps issued instructions in an structure apart from the issue queue while they can be nullified: the recovery buffer.

Figure 7 shows the placement of the recovery buffer in the pipeline. Every cycle, instructions can be issued from the issue queue, from the recovery buffer or from both structures to the execution pipelines; in the latter case, each pipeline is fed prioritarily from the recovery buffer.
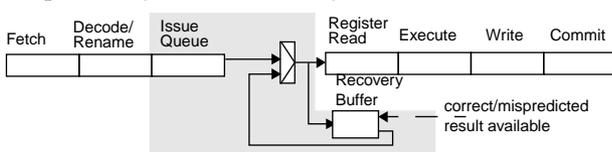


**Figure 7:** Placement of the recovery buffer in the processor pipeline.

After issuing the instructions, they are removed from their source structures and are stored in the recovery buffer, and they remain there while they can be nullified.

Each recovery-buffer entry stores all the instructions issued on the same cycle, i.e., an instruction wave. If no instruction is issued on a cycle, the related recovery-buffer entry is kept empty. Thus, the recovery-buffer entries are time-ordered in issue order; that is, the relative issue cycles among instruction waves are maintained. For instance, on cycle 5 of Figure 3, the recovery buffer holds the following instruction waves: (**e**, **f**) and (**g**, **h**).

When a prediction is verified and it turns out to be correct, the recovery-buffer entries related to the speculative window of the latency-predicted instruction are freed. However, on a misprediction, the instructions dependent on the mispredicted instruction are retained in the recovery buffer until they can be re-issued. For instance, in example of Figure 3, instruction waves (**e**) and (**g**) would be retained.

For each latency-mispredicted instruction, the recovery buffer identifies the range of recovery-buffer entries related to the instruction. Then, when the result of a mispredicted instruction is available, the re-issue logic of the recovery buffer scans the entry range (one entry per cycle) related to the instruction to re-issue its dependent instructions. For instance, in example of Figure 3, the re-issue logic scans the entries that hold the instruction waves (**e**) and (**g**).

As in the previous models, on cycles where a misprediction is detected (that is, the last cycle of the speculative window of the mispredicted instruction), the instructions selected to be issued are not issued and remain in their source structure. Also, in the issue-queue structure, instructions dependent on the nullified instructions are slept until nullified instructions are re-issued (Section 3).

Figure 8 shows the interface between the issue queue and the recovery buffer. The *removal circuit* is not shown because issued instructions are always removed from the issue queue without waiting for the prediction verification; also, the *no-request* bits are not needed.
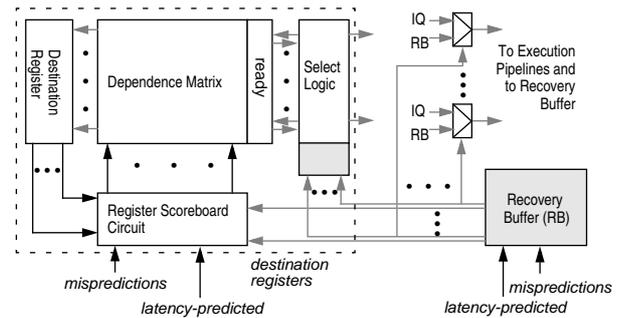


**Figure 8:** Interface between the issue queue and the recovery

Execution pipelines can be feed from both the issue queue and the recovery buffer. The multiplexers are controlled by signals generated by the recovery buffer. Also, these signals are used by the *select logic* to avoid selecting some instructions due to the higher priority of the instructions re-issued from the recovery buffer.

To wake-up the issue-queued instructions dependent on the re-issued instructions, the recovery buffer notifies every cycle the destination registers of the re-issued instructions.

The re-issue logic of the recovery buffer has a lower complexity than the issue logic of the issue queue because the former takes advantage of the scheduling performed when the

instructions were previously issued. This logic is described in the next section.

## 4.1. Recovery-Buffer organization

The recovery buffer has three instruction storage components (Figure 9): *pending-verification buffer*, *first-level buffer* and *second-level buffer*. The pending-verification buffer stores latency-predicted instructions not yet verified. The first-level buffer stores issued instructions potentially dependent on the instructions stored in the pending-verification buffer. The second-level buffer stores issued instructions dependent on latency-mispredicted instructions.

The issued instruction waves are stored in the first-level buffer and they are removed from it when they are outside all the potential speculative windows. Then, the number of cycles that an instruction wave remains in this buffer is fixed and equal to the verification delay minus one.

When an instruction wave leaves the first-level buffer, each one of the instructions is either moved to the second-level buffer or discarded. Moreover, the latency-predicted instructions of the wave are either moved to the second-level buffer or to the pending-verification buffer. These decisions are taken by considering if the instructions are included in the speculative window of a mispredicted instruction and if they are dependent on a mispredicted instruction.
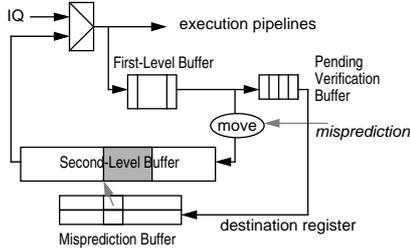


**Figure 9:** Recovery-Buffer organization.

The number of entries of the pending-verification buffer is equal to the duration of the independent window. Then, when a latency-predicted instruction leaves this buffer, its prediction is verified.

On a misprediction, the recovery buffer allocates an entry in a structure named *misprediction buffer*. This entry stores the destination register of the mispredicted instruction and a pointer to the first entry of the second level buffer related to the mispredicted instruction.

After that, during a number of cycles equal to the verification delay minus one, the instruction waves that leave the first-level buffer are analysed looking for instructions dependent on the mispredicted instruction. The dependent instructions are moved to an empty entry of the second-level buffer, and the independent instructions are discarded. Concurrently, execution pipelines are fed with ready instructions

For instance, in Figure 3, instruction **a** is moved to pending-verification buffer on cycle 4. If **a** is a latency-mispredicted instruction, instructions **e** and **g** are moved to second-level buffer on cycles 6 and 7.

**Re-issue logic of the recovery buffer.** The idea is to take advantage of the scheduling performed when the instructions

were previously issued. The re-issue logic is based on the fact that the recovery-buffer entries are time ordered and only one entry is analysed on a cycle. Then, the re-issue logic does not need to account explicitly for instruction latencies. It is enough to account for the status (availability) of the physical registers.

The status of the physical registers can be maintained locally because the recovery buffer analyses all issued instructions. When an instruction is issued, its destination register is marked as available in the recovery buffer. Also, the recovery buffer is notified of the misprediction; then, the status of the destination registers of the nullified instruction can be updated locally as not-available.

Figure 10 shows the re-issue logic of the recovery buffer. We distinguish three components: two dependence matrices (similar to the matrix described in Section 2.2) and a register scoreboard circuit without latency counters. A dependence matrix is used by instruction waves leaving the first-level buffer, and the other one is used by instruction waves re-issued from second-level buffer. The number of rows of both dependence matrices is equal to the processor issue width. Register-scoreboard circuit controls columns of both dependence matrices. A column is set or unset in both matrices at the same time.
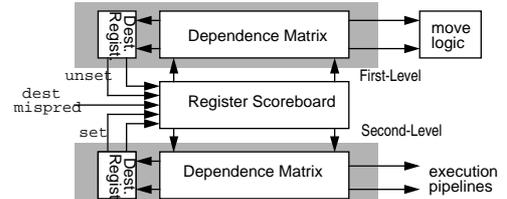


**Figure 10:** Re-issue logic of the recovery buffer.

Some input signals of the register-scoreboard circuit are generated by the first-level buffer to unset columns. Other inputs are generated by the second-level buffer to set columns. The remaining inputs are used to set/unset columns related to the destination register of mispredicted instructions.

When an instruction wave leaves the first-level buffer, it is scanned using the associated dependence matrix. This matrix has two functions: updating the status of the destination registers of the wave and, after detecting a misprediction, discriminating dependent from independent instructions.

The dependence matrix associated to the second-level buffer is used when the result of a mispredicted instruction is available. This matrix has two functions: checking the availability of the source registers of an instruction wave and updating the status of the destination registers of the re-issued instructions.

When a latency-predicted instruction leaves the pending-verification buffer, its prediction has been verified. On a misprediction, the register-scoreboard circuit unsets the column associated to its destination register.

**First-level buffer.** When an instruction wave leaves the first-level buffer, the columns related to the destination registers of the wave are set, and the wave is analysed in the dependence matrix of the first-level buffer. We differentiate two cases. In the first case, the instruction wave is not included in the spec-

ulative window of a mispredicted instruction. In this case, the dependence matrix indicates that all source registers of the instructions are available.

In the second case, the instruction wave is included in the speculative window of a mispredicted instruction. As the column associated to the destination register of the mispredicted instruction has been previously unset, the dependence matrix discriminates between instructions dependent and independent on the mispredicted instruction. Columns associated to the destination registers of the dependent instructions are unset. By unsetting the columns, the chain of instructions dependent on the latency instruction is detected on consecutive cycles.

Moreover, the output of the dependence matrix is used to indicate to the move logic which instructions of an instruction wave must be stored in the second-level buffer.

**Second-level buffer.** The dependence matrix associated to the second-level buffer is used to re-issue instructions when the result of a mispredicted instruction is available. From misprediction buffer is obtained a pointer to a second-level buffer entry and the identifier of the destination register of the mispredicted instruction. This register identifier is used to set the associated column of the matrices. After that, from the pointed entry, a fixed range of second-level buffer entries is scanned to re-issue the chain of dependent instructions. Each cycle is scanned a single second-level buffer entry.

Note that on a cycle can be detected as many mispredictions as the number of latency-predicted instructions simultaneously issued. Also, a new misprediction can be detected while other is being processed, that is, while moving dependent instruction form first-level buffer to second-level buffer. Then, in the scanned range of second-level buffer entries may be stored instructions dependent on several mispredictions. Moreover, data can return from memory hierarchy in an order different from misprediction-detection order.

The role of the second-level-buffer dependence matrix is to identify ready instructions in the scanned range of entries. For this, in each cycle is analysed an instruction wave. Ready instructions are re-issued and the columns associated to their destination registers are set.

## 4.2. Recovery buffer with selective nullification

This mechanism nullifies only instructions dependent on a latency-mispredicted instruction. First, we describe the management of the storage components of the recovery buffer and their sizes. Second, we use an example to show the detail of the mechanism. Finally, we present an optimization in the re-issue logic to avoid performance degradations.

**Storage components of the Recovery Buffer.** All storage components are handled using the FIFO policy. The instructions remain in the first-level buffer and in the pending-verification buffer for a fixed number of cycles. Instruction waves remain in the first-level buffer for a number of cycles equal to the verification delay minus one. Latency-predicted instructions remain in the pending-verification buffer a number of cycles equal to the duration of the independent window. The second level buffer and misprediction buffer are handled as a circular queue and its storage is freed, after re-issuing the instruction, using a FIFO policy.

Each misprediction requires a number of second-level-buffer entries equal to the verification delay minus one.

In the worst case, when the speculative windows of the mispredictions do not overlap, the maximum number of second-level buffer entries needed is equal to the number of pending mispredictions supported by the processor times the verification delay minus one. Although second-level-buffer size can be large, the accesses to this buffer can be pipelined without performance degradation.

**Example.** Figure 11 shows an example where the speculative windows of the mispredicted instructions (**a** and **c**) overlap by two cycles; instructions issued on cycles 8 and 9 can be dependent on both latency-predicted instructions.

On cycle 9, instruction **a** leaves the pending-verification buffer and a misprediction is detected. In this moment, the instructions waves stored in the first-level buffer are (**d**), (**e**) and (**f**). A misprediction-buffer entry is allocated to store the destination register of the mispredicted instruction and the current tail pointer to the second-level buffer.

During a number of cycles equal to the verification delay minus one, a second-level-buffer entry is allocated every cycle; also an instruction wave is analysed every cycle. Every entry will store the instructions of the analysed wave that are dependent on the mispredicted instruction.
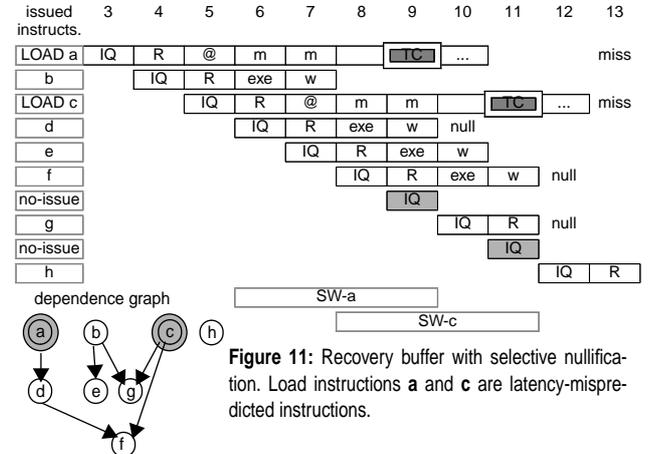
| issued instructs. | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LOAD a | IQ | R | @ | m | m | | TC | ... | | | | miss |
| b | | IQ | R | exe | w | | | | | | | |
| LOAD c | | | IQ | R | @ | m | m | | TC | ... | | miss |
| d | | | | IQ | R | exe | w | null | | | | |
| e | | | | | IQ | R | exe | w | | | | |
| f | | | | | | IQ | R | exe | w | null | | |
| no-issue | | | | | | | IQ | | | | | |
| g | | | | | | | | IQ | R | null | | |
| no-issue | | | | | | | | IQ | | | | |
| h | | | | | | | | | | IQ | R | |

dependence graph

SW-a
SW-c

**Figure 11:** Recovery buffer with selective nullification. Load instructions **a** and **c** are latency-mispredicted instructions.

On cycle 10, instruction **d** is inserted in second-level buffer. On cycle 11, the misprediction of instruction **c** is detected and a misprediction-buffer entry is allocated. Also, because instruction **e** is independent on the misprediction, a second-level-buffer entry is left empty. On cycle 12, instruction **f** is moved to second-level buffer; note that this instruction is in the speculative windows of both mispredicted instructions. On cycles 13 and 15, second-level-buffer entries are kept empty. On cycle 14, instruction **g** is moved.

When the result of the mispredicted instruction **a** is available (cycle $n$), instruction **d** will be reissued. On cycle $n+1$ no instructions will be re-issued because the related second-level buffer entry is empty. On cycle $n+2$, instruction **f** is not re-issued because it must wait until the availability of the result of the mispredicted instruction **c**.

### 4.2.1. Re-issue optimization

In some cases, when the result of a mispredicted instruction is available the first entries of the scanned range of second-level-buffer entries are empty. One case is produced when the speculative windows of several latency-mispredicted instructions overlap (note that instruction-issue is stalled when a misprediction is detected). Another case is produced when no instruction dependent on the latency-mispredicted instruction has been issued on the first cycles of its speculative window.

The cycles used to scan the first empty entries are not needed for a proper scheduling of the next not empty entries; consequently, these cycles constitute a delay. This situation can be critical if the latency of the instructions to be re-issued is long (for instance, floating-point operations).

Figure 12 shows an example where on the overlapped cycle between two speculative windows (cycle 9) no instructions are issued due to the mispredicted instruction **a**. Moreover, on cycle 10, no instructions are issued due to the mispredicted instruction **b**. These cycles are the first cycles of the speculative window of instruction **d**.

In the second-level buffer, the relevant entry ranges stores the following instruction waves: {(**h**), (**e**), (**f**), (**-**), (**-**) and (**g**)}. When data of mispredicted instruction **d** is available, the entry range {(**-**), (**-**), (**g**)} is scanned to re-issue the dependent instructions. Consequently, no instructions will be re-issued from second-level buffer on the first and the second cycle of the scanning process.

**Figure 12:** Recovery buffer with selective nullification. Example that shows the re-issue optimization. Loads **a**, **b** and **d** are latency-mispredicted instructions. On the first cycles of the speculative window of instruction **d**, no instructions are issued.
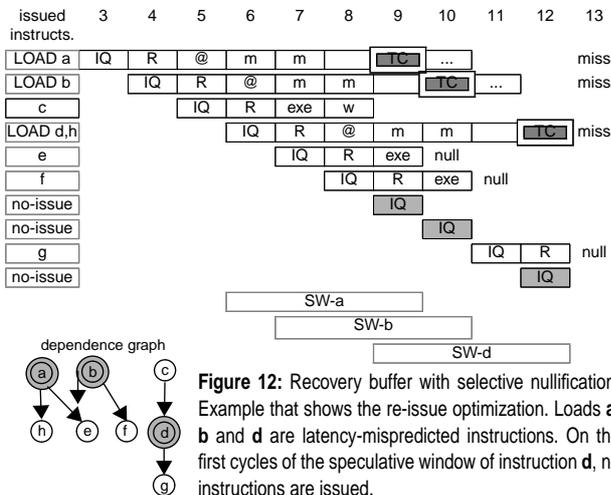
In order not to delay the re-issue of instructions, the cache controller can notify to the second-level buffer data arrival several cycles in advance. Then, re-issue-logic can skip the first empty entries of the entry range.

### 4.3. Recovery buffer with non-selective nullification

This mechanism nullifies all the instructions issued inside the speculative window of a latency-predicted instruction when a misprediction is detected. The nullified instructions independent on the misprediction are re-issued without delay from the first-level buffer. Concurrently, the dependent instructions are moved to the second-level buffer.

An example is shown Figure 13. The latency prediction of

instruction **a** is verified on cycle 5. A misprediction is detected and the instruction waves (**e**, **f**) and (**g**, **h**) are nullified. On cycle 6, the instruction wave (**e**, **f**) is analysed in the dependence matrix associated to first-level buffer; after that, instruction **e** is recorded in the second-level buffer and instruction **f** is re-issued from the first-level buffer. Analogously, on cycle 7, the dependent instruction **g** is moved to the next entry of the second-level buffer and instruction **h** is re-issued. Concurrently, some instructions are issued from the issue queue (**j** and **k** on cycles 6 and 7).
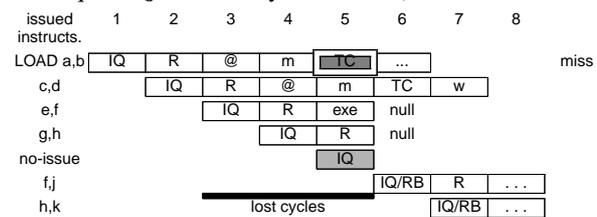
**Figure 13:** Recovery buffer with non-selective nullification. Load **a** is a latency-mispredicted instruction. IQ/RB stands for cycles where instructions are issued from the issue queue and re-issued from the recovery buffer.

### 4.4. Effect of wrong-path instructions on recovery-buffer structures

When a branch misprediction is detected, wrong-path instructions are squashed and their physical destination registers are freed. Age identifiers are used to detect the instructions to be squashed, and shadow map tables are used to re-establish the mapping from architectural to physical registers.

Recovery-buffer structures also require attention when a branch misprediction is detected. The local status of the physical registers must be repaired and the wrong-path instructions stored in these structures must be squashed. These actions are not performed immediately, they are performed concurrently with the regular operation of the recovery buffer.

In recovery buffer, an age identifier is stored with each instruction[2]. Also, a new recovery-buffer structure (the squashed-range buffer) holds the range of age identifiers of the squashed instructions. An entry of this structure is freed when the age identifier of a committed instruction is younger than the youngest limit of the range.

Regular operations of the recovery buffer that analyse instruction waves are: a) when an instruction wave leaves the first-level buffer and b) when an instruction is re-issued from the second-level buffer. In both cases, the local status of the physical registers is updated. To squash wrong-path instructions, the age identifiers of the analysed instructions are checked with entries of the squashed-range buffer. If the age identifier is included in an squashed range, the instruction is squashed and neither recorded in second-level buffer nor re-issued.

When a mispredicted instruction must be squashed, previous regular operations squash neither it nor its dependent

---

2. This identifier can be the processor age identifier, or the recovery buffer can build a local age identifier from the processor age identified.

instructions (all of them are wrong-path instructions). This case is managed by the freeing algorithm of the second-level buffer. As this buffer and the misprediction buffer are managed as circular queues, the age identifier of the latency-mispredicted instructions will achieve the head entry of the misprediction buffer. Then, the age identifier of the head entry is compared with the squashed-range-buffer entries. If the age identifier is included in an squashed range, the related instruction is squashed.

The local status of the physical registers freed by squashed instructions is repaired in time by the algorithm described in Section 4.1; this algorithm updates the local status of a physical register when its producer instruction leaves the first-level buffer. The freed physical registers are assigned to a producer instruction in the new path, and younger instructions use it as a source register. When the producer instruction leaves the first-level buffer, before its consumer instructions, the local status of the register is repaired. Before this time, no instruction needs to check the local status of this register.

# 5. Evaluation
## 5.1. Simulation environment
Our evaluations used a cycle-by-cycle simulator derived from SimpleScalar 3.0 tool set (Alpha ISA) [2]. The simulated processor had a separate reorder buffer, two issue queues, physical register files like Alpha 21264 processor. Also, the issue width is four integer instructions and two floating-point instructions, the issue policy always selects the oldest ready instructions for execution, and there is a pipeline stage, between issue queue and the functional units, devoted to reading registers (like Figure 4). Table 1 summarizes the baseline processor.

**Table 1:** Baseline processor model.

| Issue Queues | 20-entry integer issue queue/ 15-entry FP issue queue |
|---|---|
| Reorder Buffer | 128 entries |
| Load-Store queue | 64 entries |
| Instruction Fetch | up to 4 consecutive instructions per cycle |
| Decode width | up to 4 instructions per cycle |
| Issue width | up to 4 integer instructs., up to 2 FP instructs. |
| Commit width | up to 4 instructions per cycle |
| Functional Units | -4 Integer ALU's, 1-cycle latency<br>-1 Integer Mult/Div, 3/20-cycle, fully pipelined/not pip.<br>-1 FP Adder, 4-cycle, fully pipelined<br>-1 FPMult/Div, 4/12-cycle, fully pipelined/not pip<br>-2 memory acesses (any combination of loads and stores) |
| Branch Prediction | Hybrid predictor: local($2^{15}$ entries)<br>+ gshare(15 bits) + selector<br>Speculatively updated in decode stage<br>1024-entry, 4 way BTB |
| First-Level Cache | Separated caches, Direct mapped, 2-cycle latency |
| Second-Level Cache | Unified, 1 Megabyte, Direct-mapped, 12-cycle latency |
| Main Memory | 80-cycle latency |

In current processors, the integer issue queue typically does not exceed 20 entries. For instance, 20 entries in Alpha 21264, 18 entries in AMD Athlon and 20 entries in Intel P6. Also, the number of in-flight instructions (reorder-buffer size) is typically 2 to 4 times bigger. In this paper, the issue-queue sizes of the baseline processor model are 20-entry integer issue queue, and 15-entry floating-point issue queue.

The evaluations varied the following parameters: issue-queue sizes, verification delay and first-level cache size. The verification delay was defined in Section 2 as the duration of the speculative window; we evaluate 2-cycle, 3-cycle and 4-cycle verification delays. We present results only for 64-Kbyte first-level caches, but we have also performed evaluations using 32-Kbyte first-level caches and the behaviour is qualitatively similar.

## 5.2. Benchmark description
To perform our evaluations we collected results for the SPEC95 benchmarks. Programs were compiled on an Alpha 21164 processor using the full optimizations provided by the native compiler.

Each benchmark was executed using the reference data set (cp-delcl.i input file was selected for gcc benchmark), but our simulators focused on an interval of the execution. They discarded the initial part of the execution of the benchmarks and then started a detailed simulation for a limited number of committed instructions. To decide the amount of instructions to be discarded and to be simulated, we performed an analysis on temporal behaviour. Table 2 shows the interval selected for each benchmark.

**Table 2:** Simulation intervals for SPEC95 programs (millions of instructions discarded / millions of committed instructions simulated) and miss rate in a direct-mapped 64K first-level data cache.

| SPEC INT | | | SPEC FP | | |
|---|---|---|---|---|---|
| Benchmark | Interval | %miss | Benchmark | Interval | %miss |
| go | 4000/1000 | 1.9 | tomcatv | 1500/500 | 18.6 |
| m88ksim | 1000/1000 | 0.7 | swim | 500/500 | 6.1 |
| gcc | 0/finish | 2.2 | hydro2d | 500/500 | 15.0 |
| compress | 4500/2500 | 12.5 | mgrid | 0/500 | 4.0 |
| li | 2500/1000 | 3.3 | applu | 500/500 | 7.1 |
| ijpeg | 1000/1000 | 0.6 | turb3d | 0/500 | 4.1 |
| perl | 1000/1000 | 0.5 | fpppp | 0/500 | 0.2 |
| vortex | 1000/7000 | 3.0 | wave5 | 1500/500 | 7.5 |

## 5.3. Results
We present the results of the integer benchmarks separately from the results of the floating-point benchmarks because the behaviour of the recovery-buffer mechanisms depends on the computational latency of the instructions.

In figures are used the following acronyms for identifying the evaluated mechanisms: a) IQNS: keeping in the Issue Queue with No Selective nullification, b) IQS: keeping in the Issue Queue with Selective nullification, c) RBNS: Recovery Buffer with No Selective nullification, and d) RBS: Recovery Buffer with Selective nullification.

### 5.3.1. Integer benchmarks
In all the evaluations performed in this section we used a 10-entry floating-point issue queue and the following integer issue-queue sizes: 15, 20 and 25 entries. First, the sensitivity of the evaluated mechanisms to the verification delay is showed. In Figure 14, each graph is related to an issue-queue size, the horizontal axis stands for the verification delay and the vertical axis for the harmonic mean of the IPC's of the integer benchmarks.

The sensitivity to the verification delay of RBNS and RBS remains small when the verification delay or the issue-queue

size increases. Also, performance differences between RBNS and RBS are small (less than 1.3%). As the instructions are removed from the issue queue after issuing them, the differences arise because RBNS re-issues nullified instructions independent on the mispredicted instructions.
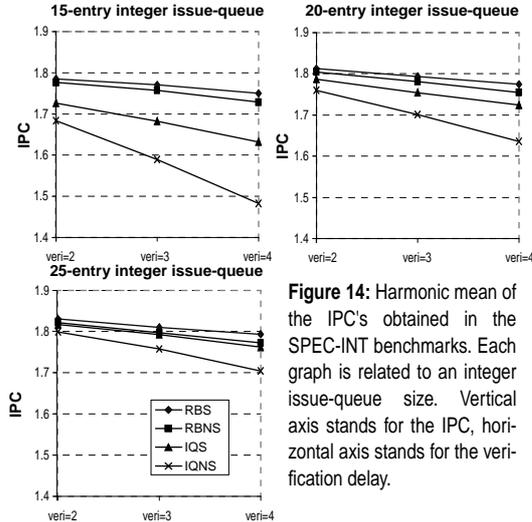


**Figure 14:** Harmonic mean of the IPC's obtained in the SPEC-INT benchmarks. Each graph is related to an integer issue-queue size. Vertical axis stands for the IPC, horizontal axis stands for the verification delay.

However, the sensitivity to the verification delay of IQNS and IQS is very significant for the 15-entry issue queue, and decreases as the issue-queue size increases. This result shows that keeping all instructions in the issue queue a number of cycles equal to the verification delay (IQNS) can reduce significantly the performance.

IQS retains in issue queue only instructions dependent on the latency-predicted instructions but its performance is smaller than the performance of RBNS, although both are close in a 25-entry issue-queue. The recovery buffer allows freeing the issue-queue entries of some of the instructions dependent on the mispredicted instructions. These freed entries can be assigned to new instructions, increasing the look-ahead capacity of the instruction scheduler. Moreover, this capacity overcomes the lost issue slots for re-issuing independent instructions.
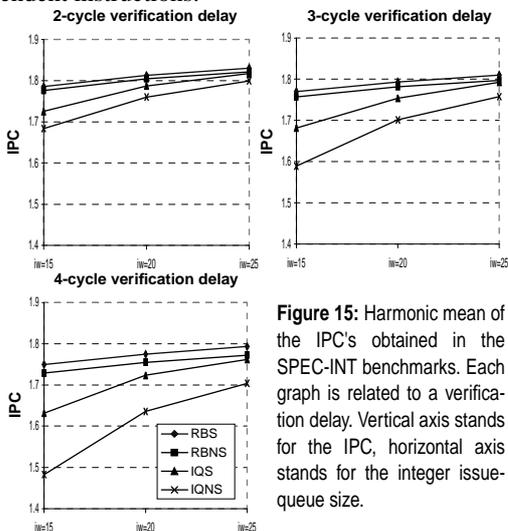


**Figure 15:** Harmonic mean of the IPC's obtained in the SPEC-INT benchmarks. Each graph is related to a verification delay. Vertical axis stands for the IPC, horizontal axis stands for the integer issue-queue size.

The capacity of the recovery buffer to free issue-queue entries allows the use of a smaller issue queue. Figure 15 shows almost the same information than Figure 14 but grouping data by verification delay, and putting the issue-queue size in the horizontal axis. In all cases, RBNS and RBS can obtain the same performance than IQNS and IQS, but with a reduction of the issue-queue size around 20% to 25%[3].

The implementation of the selective nullification in the issue queue may be critical with intensive one-cycle operations. Comparing non-selective mechanisms, the longer the verification delay, the larger the issue-queue size can be reduced. Then, RBNS is an attractive solution.

### 5.3.2. Floating-point benchmarks

In all the evaluations performed in this section we used a 20-entry integer issue queue and the following floating-point issue-queue sizes: 10, 15 and 20 entries.

Floating-point benchmarks show (Figure 16 and Figure 17), in the evaluated mechanisms, a different behaviour than integer benchmarks. Non-selective mechanisms are sensitive to the verification delay while selective mechanisms do not. This fact produces that IQS performance is better than RBNS performance when issue-queue size increases.
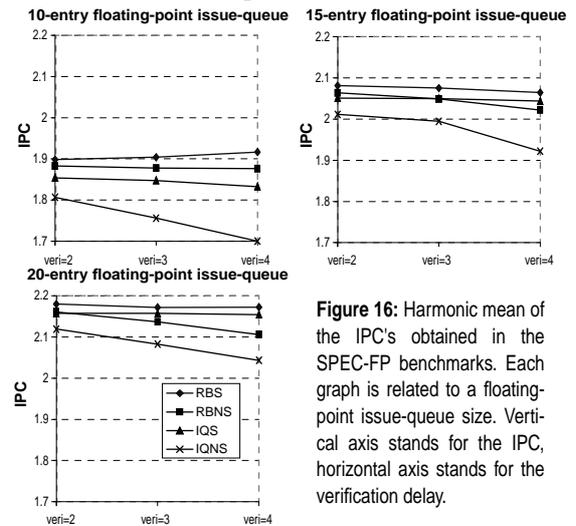


**Figure 16:** Harmonic mean of the IPC's obtained in the SPEC-FP benchmarks. Each graph is related to a floating-point issue-queue size. Vertical axis stands for the IPC, horizontal axis stands for the verification delay.

The best (RBS) and the worst (IQNS) mechanisms are the same for both classes of benchmarks. Also, RBS is almost insensitive to the verification delay while IQNS is very sensitive to it.

The behaviour of the floating-point benchmarks is related to the computational latency of the floating-point instructions; four-cycle latency for FPadder and FPmultiplier are used, and the evaluated verification delays range from two to four cycles. These latencies forbid the existence of a chain of dependent instructions larger than one instruction in the speculative window of a FPload instruction. Therefore, only the first data-flow level dependent on a FPload is included in its speculative window. Other data-flow levels are held in issue-

---

queue entries and the scheduler can not look-ahead because the issue queue is full. Then, the capacity of the recovery buffer to store dependent instructions is slightly used.

Also, the value retrieved by a load instruction is used by few instructions, that is, its fan-out is small. Then, in a non-selective mechanism, a large number of the independent instructions[4] are often re-issued. As the latency of the re-issued instructions is long in floating-point benchmarks, the execution of the chain of dependent instructions is significantly delayed, and potential ILP is lost.

The implementation of selective mechanisms for integer-benchmarks can be critical because the latency of most ALU operations is one cycle. However, as the computational latency of the FP operations is longer, the implementation of selective mechanisms for them is less critical.
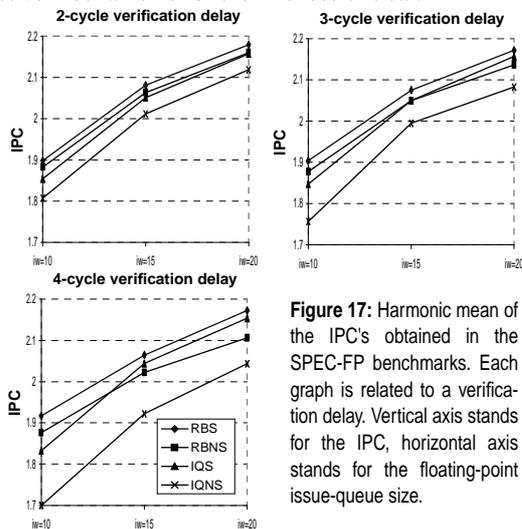


**Figure 17:** Harmonic mean of the IPC's obtained in the SPEC-FP benchmarks. Each graph is related to a verification delay. Vertical axis stands for the IPC, horizontal axis stands for the floating-point issue-queue size.

## 6. Conclusions

This paper addresses recovery mechanisms to deal with latency-predicted instructions; it compares the performance of a conventional mechanism that keeps issued instructions in the issue queue versus a mechanism that stores these instructions in a recovery buffer apart from the issue queue. Also, it compares selective versus non-selective instruction nullifications on mispredictions.

We designed a recovery-buffer mechanism and we evaluated it in the context of load-latency prediction. Our results show that, under the same nullification conditions, the recovery-buffer mechanism outperforms the mechanism that retains the instructions in the issue queue and, moreover, it is less sensitive to the verification delay of the predictions. For integer benchmarks, the mechanism allows a reduction in the issue-queue size around 20-25% without performance decrease. The recovery-buffer mechanism allows the issue-queue logic to free entries and to insert new instructions in the issue queue; therefore, it increases the capacity of the scheduler to

look-ahead for independent instructions.

Also, we show that for issue queues that feed functional units with intensive one-cycle latency operations, the simple recovery-buffer mechanism with non-selective nullification is an attractive solution. On the other hand, for issue queues that feed functional units where the latency of most instructions is long, the use of selective nullification is preferable. Note that, in this case, selective nullification is not critical due to the long latency of the operations.

Further work is needed to evaluate the use of the recovery-buffer mechanism in other kinds of prediction scenarios, such as address prediction and value prediction. Also, we are interested in studying how the recovery-buffer mechanism can be integrated into mechanisms that perform a dynamic data-flow pre-scheduling [11] or that use an issue queue for accounting latency mispredictions or for waiting the result of unknown-latency instructions [3].

## Acknowledgments

## References

[1]V. Agarwal, M.S. Hrishikesh, S.W. Keckler and D. Burger. *Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures.* Proc. of the Int. Symp. on Computer Architecture, pp. 248-259, 2000

[2]D. Burger and T.M. Austin. *The Simplescalar Tool Set Version 2.0.* T.R. 1342, Computer Science Department, University of Wisconssin, june 1997

[3]R. Canal and A. González. *A Low-Complexity Issue Logic.* Proc. of 14th Int. Conf. on Supercomputing, pp. 327-335. may 2000

[4]D. Carmean. *Inside the Pentium 4 Processor Microarchitecture.* Intel Developer Forum, Fall 2000

[5]K. Diefendorffc. *Hal Makes Sparcs Fly. Sparc64 V Employs Trace Cache and Superspeculation for high ILP.* Microprocessor Report, Vol 13(15), pp 5-13, 1999

[6]J.A. Farrell and T.C. Fischer. *Issue Logic for a 600 MHz Out-of-Order Execution Microprocessor.* IEEE Journal of Solid-State Circuits, Vol 33(5), pp 707-712, 1998

[7]T. Horel and G. Lauterbach. *UltraSparc III: Designing Third-Generation 64-Bit Performance.* IEEE MICRO,Vol. 19, pp. 73-85, may-june 1999

[8]D. Hunt. *Advanced Performance Features of the 64-bit PA8000.* Proc. of the COMPCON, pp.123-128, 1995

[9]R.E.Kessler. *The Alpha 21264 Microprocessor.* IEEE MICRO,Vol. 19, pp 24-36, march-april 1999

[10]D. Matzke. *Will Physical Scalability Sabotage Performance Gains?* IEEE Computer Vol. 30, n 9, pp. 37-39, 1998

[11]P. Michaud and A. Seznec. *Data-flow Prescheduling for Large Instruction Windows in Out-of-Order Processors.* Proc. of the Int. Symp. on High Performance Computer Architecture, pp. 27-36, 2001

[12]MIPS. *MIPS R4000 Microprocessors User's manual*

[13]S. Palacharla, N.P. Jouppi and J.E. Smith *Complexity-Effective Superscalar Processors.* Proc. of the Int. Symp. on Computer Architecture, pp 206-218, 1997

[14]E. Rotenberg, Q. Jacobson, Y. Sazeidas and J. Smith. *Trace Processors.* Proc. of the Int. Symp. on Microarchitecture, pp. 138-148, 1997

[15]J.Stark, M.D. Brown and Y.N. Patt. On Pipelining Dynamic Instruction Scheduling Logic. Proc. of the Int. Symp. on Microarchitecture, pp. 57-66, 2000.

[16]R.M. Tomasulo. *An Efficient Algorithm for Exploting Multiple Arithmetic Units.* IBM Journal of Research and Development, vol 11, pp 25-33, jan. 1967

[17]K.C. Yeager. *The MIPS R10000 Superscalar Microprocessor.* IEEE MICRO,Vol. 16, n 2, pp 28-41, april 1996

---

4. Using a 15-entry floating-point issue queue and a 4-cycle verification delay, 85% of the nullified instructions on floating-point benchmarks are independent on the mispredicted instructions. In integer benchmarks, using a 20-entry integer issue-queue and a 4-cycle verification delay, this percentage drops to 53%.