Predicting L2 Misses to Increase Issue-Queue Efficacy

Enric Morancho, José María Llabería and Àngel Olivé

Abstract — **The issue queue keeps the instructions that are waiting for the availability of input operands and issue slots. While some instructions remain for a few cycles in the issue queue, the instructions dependent on L2 misses may remain there for hundreds of cycles due to the L2 miss latency. Some authors have proposed mechanisms to extract these instructions from the issue queue. However, these mechanisms increase the issue-queue activity because the extracted instructions must be replayed, that is, issued twice (at least). Firstly, to be extracted from the issue queue. Secondly, after resolving the L2 miss, to be executed.**

We propose delaying the insertion of some instructions in the issue queue. After predicting which load instructions are going to miss in L2, the instructions dependent on these load instructions will be stored in an instruction buffer instead of being inserted in the issue queue. After resolving the miss, the instruction buffer will be traversed in order to insert in the issue queue the instructions dependent on the resolved memory access. The advantages of this proposal with respect to proposals that extract from the issue queue the instructions dependent on L2 misses are twofold. First, it avoids filling the issue queue with instructions dependent on L2 misses. Second, it reduces the amount of instruction replays.

The evaluations show that delaying the insertion of instructions in the issue queue reduces the amount of instruction replays between 27% and 31% in integer benchmarks and between 33% and 39% in floating-point benchmarks with respect to processors that extract from the issue queue the instructions dependent on L2 misses. The evaluations also show that this replay reduction does not harm processor performance.

Index Terms — **Instruction issue, L2 hit/miss prediction, Instruction replays.**

I. Introduction

In out-of-order processors, the issue queue is responsible for exposing the instruction level parallelism available in programs. A conventional issue queue keeps a subset of the in-flight instructions: the ones that have been renamed and that are waiting for the availability of either the input operands or the functional unit.

Instructions are inserted in the issue queue just after being renamed, and are extracted after being issued to the functional unit. Some instructions are extracted from the issue queue just after being issued. However, other issued instructions remain in the issue queue until the predicted latencies for their producer instructions have been checked because the issue queue is the recovery stage for latency mispredictions [6].

While some instructions remain in the issue queue for a few cycles, others remain there for a large time-span. Considering that the latency of L2 misses is about hundreds of cycles, instructions dependent on L2 misses may remain in the issue queue for hundreds of cycles. This large time-span produces a reduction in the effective capacity of the issue queue because the entries assigned to the dependent instructions will not be used to expose parallelism until the L2 miss gets resolved. Moreover, if the dependent instructions fill the issue queue, no parallelism will be exposed until the L2 miss gets resolved; consequently, the processor will be unable to issue instructions to the functional units.

To avoid filling the issue queue with instructions dependent on L2 misses, several authors propose extracting the dependent instructions from the issue queue and re-inserting (replaying) them again when the L2 miss has been resolved [8][12][17]. After extracting a dependent instruction, the freed issue-queue entry may be assigned to a younger instruction that, if independent, will be issued; otherwise, it will be also extracted to allow inserting a younger instruction, and so on. Although these mechanisms are able to increase processor performance, they also increase the issue-queue activity because the extracted instructions must be replayed; consequently, the issue-queue energy consumption increases. We will refer to these mechanisms as *early extracting mechanisms*.

We propose a mechanism that tries to delay the insertion in the issue queue of instructions dependent on L2 misses, that is, a *late inserting mechanism*. Early in the pipeline, we predict whether a load instruction will miss L2 or not. This information will be propagated to the dependent instructions in the Register Renaming stage. Only instructions predicted to be independent on L2 misses will be inserted in the issue queue; the remaining instructions (dependent on load instructions predicted to miss in L2) will be recorded in an Instruction Buffer located before the issue queue. When the L2 miss gets resolved, the Instruction Buffer is scanned at the same width as issue width in order to insert in the issue queue the dependent instructions. The goals of the proposed mechanism are: a) reducing the amount of instruction replays, b) do not harm

Authors are with the Computer Architecture Department, Universitat Politecnica de Catalunya, Spain. {enricm, llaberia, angel}@ac.upc.edu

E. Morancho and J.M. Llabería are also with the European Network of Excellence on High-Performance Embedded Architectures and Compilers (HiPEAC)

processor performance.

The evaluations compare a baseline processor with an *early extracting* mechanism versus the same processor extended with the *late inserting* mechanism. The *late inserting* mechanism reduces the number of instruction replays between 27% and 31% in integer benchmarks and between 33% and 39% in floating-point benchmarks with respect to the baseline processor. Our evaluations also show that the *late inserting* mechanism does not harm processor performance.

The remainder of this paper is organized as follows. Section II describes the baseline processor and how we extend it with the *late inserting* mechanism. Section III details the L2 hit-miss predictor we have developed. Section IV presents an evaluation of our proposal. Section V describes related works. Finally, Section VI summarizes the conclusions of this work.

II. PROCESSOR DESCRIPTION

In this section we describe the processors used in this work: the baseline processor and our proposed extension. The baseline processor uses an *early extracting* mechanism, that is, a mechanism that extracts from the issue queue the instructions dependent on L2 misses. Our proposed extension, the *late inserting* mechanism, allows delaying the insertion of some instructions in the issue queue.

A. Baseline processor

Fig. 1 shows a block diagram of the baseline processor. The core scheduler of the baseline processor uses an issue queue and a Recovery Buffer [10]. Instructions are extracted from the issue queue after being issued and are inserted in the Recovery Buffer in issue-order timing. The instructions remain in the Recovery Buffer until knowing if they are dependent on a load instruction that misses in L1. On a L1 hit, the instructions are discarded but, on a L1 miss they remain in the Recovery Buffer until knowing if the load hits L2. If the load hits L2, instructions dependent on the load are re-issued (replayed) from the Recovery Buffer when the cache-block is allocated in L1. To schedule the instructions, the Recovery Buffer uses the issue-order timing that has been recorded; while the Recovery Buffer is re-issuing instructions, no instructions are issued from the issue queue.

On a L2 miss, the load and its dependent instructions are discarded from the Recovery Buffer. Moreover, a bit vector with as many entries as physical registers is updated; the bit vector indicates which physical registers depend on L2 misses.

To re-issue (replay) these instructions when the cache block is allocated in L1, we use a buffer named Instruction Buffer (IB). The IB keeps all in-flight renamed instructions. Instructions can be inserted from IB to the issue queue. While instructions are inserted from the IB, no instructions are inserted in the issue queue from the Rename stage. We use an IB with as many entries as Reorder-Buffer entries. However, several implementations that optimize the size of this buffer have been published in the literature [2][17].

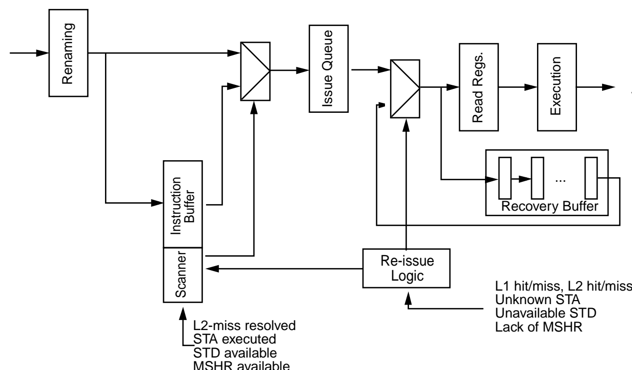

Fig. 1. Block diagram of the baseline processor

A module named *scanner* is responsible for inserting the instructions from the IB to the issue queue. On each L2 miss, the IB-entry identifier of the load instruction is recorded because, when the L2 miss is resolved, the scanner uses this identifier to start searching for instructions dependent on the load.

The scanner module analyses as many consecutive entries in IB as the issue width. Instructions dependent on the load (and not dependent on more unresolved L2 misses) are inserted to the issue queue. While the scanner analyses IB entries, other L2 miss can be resolved. If the latest resolved load is older than the instruction that is being analysed, the scanner goes back to the IB entry related to the latest miss and proceeds analysing from there. When the scanner reaches the youngest instruction in IB, scanning finishes. Moreover, the IB is also used for recovery actions: memory disambiguation and lacking of MSHR entries. According to the categorization presented in [7], the replay mechanism used by the baseline processor performs serial verification and selective replay.

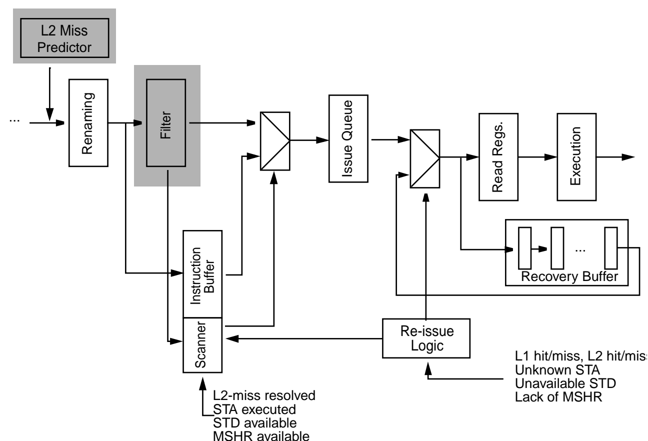

Fig. 2. Block diagram of the Late-Inserting mechanims (differences respect baseline processor have been shaded).

B. Extensions to the baseline processor: late inserting mechanism

Fig. 2 shows a block diagram of the processor with the *late inserting* mechanism. The proposed mechanism relays on predicting whether a load instruction will miss L2 or not. The L2 hit-miss predictor is accessed early in the pipeline using the PC of the load instruction. This prediction is propagated to the instructions dependent on the load instruction in the Rename phase. The prediction is used in the Filter phase. While the load instruction predicted to miss L2 is inserted to the issue queue, its dependent instructions are not. All instructions are kept in the IB. When the L2 miss gets resolved, the IB is scanned in order to insert the instructions dependent on the resolved L2 miss.

III. L2 HIT-MISS PREDICTION

This section details the L2 hit-miss predictor developed in this work. Before starting the development of the L2 hit-miss predictor, Table I shows the number of L2 misses per 1000 committed instructions in each analysed benchmark (benchmark selection and processor configuration is detailed in Section A).

Integer benchmarks	bzip2	crafty	eon	gap	gcc	gzip	
	0.3	6.4	1.2	4.0	1.1	1.2	
	mcf	parser	perl	twolf	vortex	vpr	
	47.2	7.7	2.6	18.0	6.7	12.1	
Floating-point benchmarks	ammp	applu	apsi	art	equake	facerec	fma3d
	40.5	75.0	33.1	155.8	6.1	51.4	12.5
	galgel	lucas	mesa	mgrid	sixtrack	swim	wupwise
	219.2	2.9	0.8	21.8	1.1	98.1	8.0

TABLE I Benchmark characterization according to the number of L2 misses per 1000 committed instructions

To avoid inserting in the issue queue instructions dependent on L2 misses we use a predictor that is accessed early in the pipeline. We adapt to L2 hit-miss prediction the perceptron predictor [4]. Several researchers have applied the perceptron predictor to branch prediction with promising predictability results [5]. The main drawback of the perceptron predictor is its latency; however, as our mechanism needs the L2 hit-miss prediction in the Register Renaming stage, we can tolerate perceptron latency.

Perceptron predictors use a global history register and a prediction table. The global history register is updated speculatively after performing every prediction; to recover the global history register after a branch missprediction, each conditional branch instruction checkpoints the current value of the global history register. The prediction table is updated at commit stage.

In the scope of L2 hit-miss prediction, misspredicting a load instruction of the correct execution path influences to the following hit-miss predictions of the correct execution path until the missprediction is extracted from the global history register. In the scope of branch prediction, this behaviour does not appear; after detecting a branch misprediction, the branch history register is restored when the fetch engine is redirected to the correct target. Consequently, we have evaluated the influence on predictor performance of correcting the history register as soon as the hit-miss prediction is known to be incorrect.

Fig. 3 plots the evaluations performed in our exploration of the design space of the predictor. The vertical axis shows the percent of dynamic load instructions. Each column is related to a predictor configuration. For each configuration, we distribute the dynamic load instructions into: L2 misses correctly predicted, L2 misses predicted to hit L2 and L2 hits predicted to miss L2. The remaining portion of each bar until the 100% stands for correctly predicted L2 hits. We present two group of results: for integer and for floating-point benchmarks.

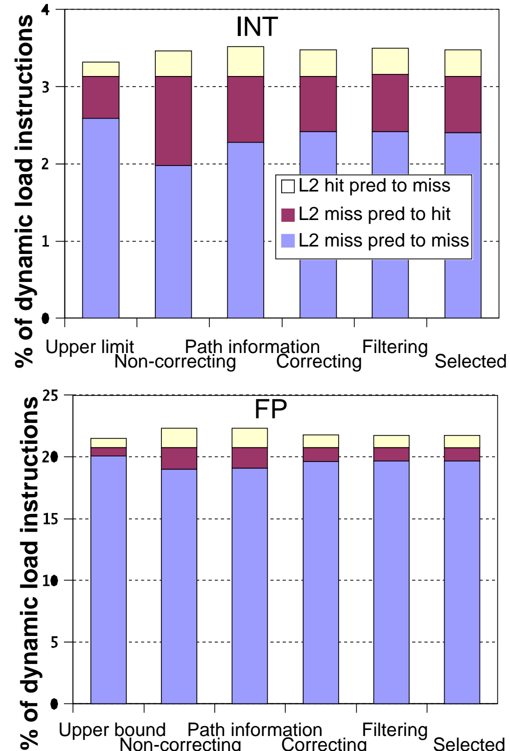

Fig. 3. Predictability results of several configurations of the perceptron predictor.

We describe the evaluations we have performed (unless noted, we use an unbounded prediction table, a 31-bit global history register and 7-bit counters).

- **Upper limit:** To establish an upper limit of the performance of the predictor, we simulated a perceptron updated at commit stage with correct L2 hit-miss information. Coverage is around 83% and 97% of the L2 misses in integer and floating-point benchmarks respectively.
- **Non-correcting global history register:** We conducted a simulation where the global history register is updated speculatively and it is not corrected on L2 hit-miss mispredictions. The increment on the amount of L2 misses predicted to hit is produced because each L2 hit-miss missprediction introduces a wrong value in the global history register that influences following predictions until the

misprediction is evicted from the global history register.

- **Extending the global history register with execution-path information:** The global history register is extended with 32 bits that represent the execution path. It is formed by concatenating the four least-significant bits of the instruction addresses of the latest eight predicted instructions. In integer benchmarks, L2-misses coverage is increased significantly.
- **Correcting the global history register**: The global history register is updated speculatively and it is corrected as soon as an L2 hit-miss prediction is known to be incorrect. Coverage is closer to upper limit (77% and 95% for integer and floating-point benchmarks). Correcting the global history register reduces the amount of L2 misses predicted to hit.
- **Filtering predictions:** A significant number of the static load instructions never miss L2 (70% and 50% for integer/floating-point benchmarks) and represent a significant amount of dynamic load instructions (around 30%). We have considered training the perceptron predictor only with the load instructions that have missed L2. The advantages of filtering are twofold: firstly, allows correlating load instructions using shorter global history registers; secondly, reduces the number of predictions performed by the perceptron predictor. Results show that filtering increments slightly the amount of L2 misses predicted to miss and reduces around 30% the amount of perceptron lookups.
- **Selected configuration:** After evaluating several predictor configurations, we have selected the following perceptron configuration: 256 table entries, 11-bit global history register, 7-bit counters and filtering using a 2Kbit table; the hardware budget of the configuration is less than 3Kbytes.

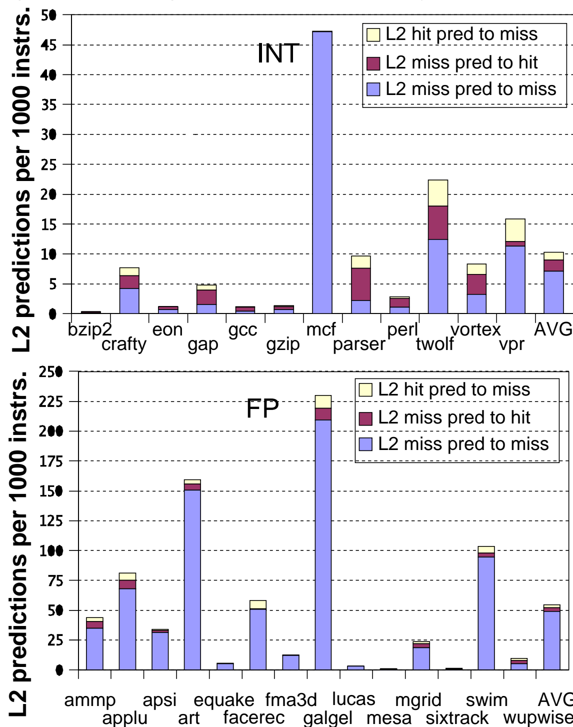

Fig. 4. Individual prediction breakdown

Fig. 4 shows individual predictability results with the selected predictor configuration. To simplify result comparison, the vertical axis stands for the amount of L2 hit-miss predictions per 1000 committed instructions and, for each benchmark, we present the same prediction breakdown as in Fig. 3.

IV. Evaluation

A. Evaluation environment

1) Simulator

To evaluate our proposals we will use several processor simulators derived from the SimpleScalar tool set [1]. This tool set offers a cycle-by-cycle simulator of a superscalar out-of-order processor with a two-level cache hierarchy. We have added to the simulator issue queues, L2 miss predictors and the mechanisms proposed to deal with instructions dependent on L2 misses.

2) Processor model

Table II details the main parameters of the evaluated processors. We have considered a large reorder buffer (2048 entries) to help the processor to tolerate the latency of L2 misses (400 cycles). Processor pipeline has two stages between instruction issue and the functional units.

Fetch width	4 instructions
Branch prediction	hybrid predictor 16-bit gshare & bimodal, 1024-entry BTB, 32-entry RAS, 10 cycles misprediction latency
Issue width	4 instructions
Issue Queues	2 queues (integer IQ and floating-point IQ), same size (20, 30, or 40 entries)
Commit width	8 instructions
Functional units	4 integer alus, 1 integer multiplier/divider, 2 FP alus, 1 FP divider, 2 memory ports
ROB	2048 entries
LSQ	1024 entries
Instruction L1	32 Kb, direct mapped, 2-cycle hit latency
Data L1	32 Kb, 2way, 2-cycle hit latency
Unified L2	256 Kb, direct-mapped, 9 cycle hit latency, 400 cycles miss latency
Instruction TLB	32 entry, 4-way
Data TLB	64 entry, 8-way
MSHR	10 entries

TABLE II Processor configuration

Benchmark description. To evaluate our proposal we simulate the execution of a representative interval of 100 million instructions of each Spec2000 benchmark [16]. Before starting each simulation, we have warmed-up caches and branch predictors. Binaries have been obtained compiling with full optimizations on an Alpha machine. Table I lists these benchmarks and details the L2 miss rate.

B. Results

1) Instruction replays

Fig. 5-a shows the number of instructions replayed by the baseline processor and by our proposal. The vertical axis shows the amount of instructions replays per 1000 committed

instructions for several issue-queue sizes. To remove the influence of the instructions of the wrong execution path, we consider only instructions issued of the correct execution path. Due to the biased behaviour of benchmark *mcf*, we present average results for all integer benchmarks but benchmark *mcf*, all integer benchmarks and all floating-point benchmarks. For instance, for floating-point benchmarks, while the baseline processor performs around 300 instruction replays per 1000 instruction commits, our proposal performs 200 replays per 1000 instruction commits. The reduction in the number of instruction replays is between 27%-31% in integer benchmarks but *mcf*, and between 33% and 39% in floating-point benchmarks.

Using our proposal, the reduction in the number of replays depends on the number of correct predictions and on the length of the chain of instructions dependent on the load instructions predicted to miss in L2. Consequently, using our proposal, the number of replays is almost no sensitive on the issue-queue size; however, using the baseline processor, the number of replays is sensitive to the issue-queue size.

Using larger issue queues allows inserting more instructions in the issue queue before the issue queue becomes full. If the inserted instructions are dependent on a load that misses L2, the mechanism will perform more replays. For instance, in average, in *apsi* benchmark, each load instruction predicted to miss L2 reduces 5 replays; however, in *parser* benchmark, each load instruction predicted to miss L2 reduces 40 replays.

Fig. 5-b and c details the replay reduction for each benchmark. We plot the number of replays performed by both the baseline processor and our proposal per 1000 committed instructions for several issue-queue sizes. Except two benchmarks, there is not a clear correlation between the issue-queue size and the number of replay reductions.

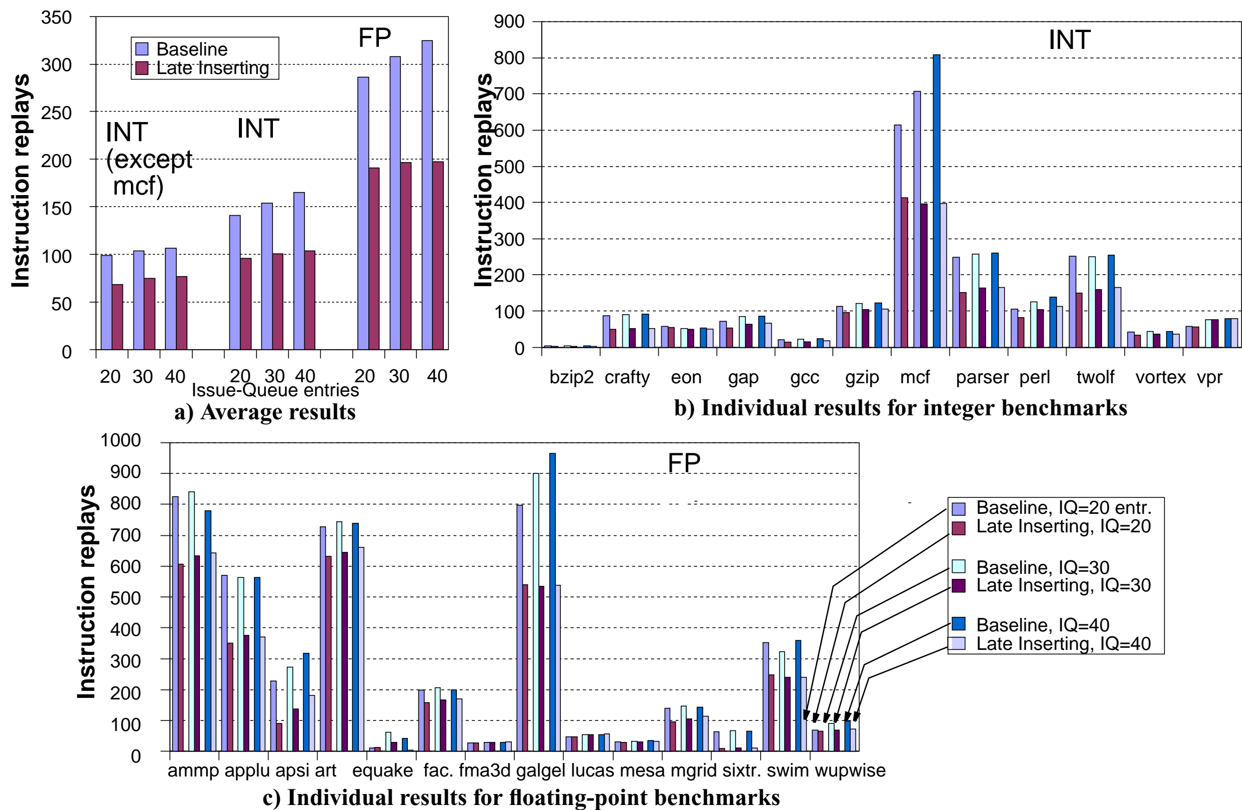

Fig. 5. Replays performed by both the baseline processor and our proposal per 1000 committed instructions

2) Processor performance

Although our proposal reduces the number of instruction replays, it may impact on processor performance because it may delay unnecessarily the insertion of some instructions in the issue queue. Our results show that, in average, our proposal does not harm IPC with respect to the baseline processor.

Fig. 6 shows, for each benchmark, the relative processor performance of our proposal with respect to the performance of the Baseline processor. The individual results show that the processor performance using our proposal is, except in *galgel* benchmark, at most, 1% worse than that of Baseline processor. In some benchmarks we obtain performance improvements around 2% (*apsi*, *mgrid* and *swim*) and, in *applu* benchmark, we obtain, at least, around 7% performance improvement.

The exceptional behaviour of *applu* benchmark can be explained because our proposal increases the average amount

of free entries in the 20-entry floating-point issue queue (from 5.8 to 8.9 entries) and reduces the amount of cycles the floating-point issue queue is full (from 56% to 38% of the execution cycles). Consequently, using our proposal, floating-point instructions independent on L2 misses may be inserted in the issue queue and may be issued before than using the Baseline processor and, according to the instant when misses get resolved, also be committed before.

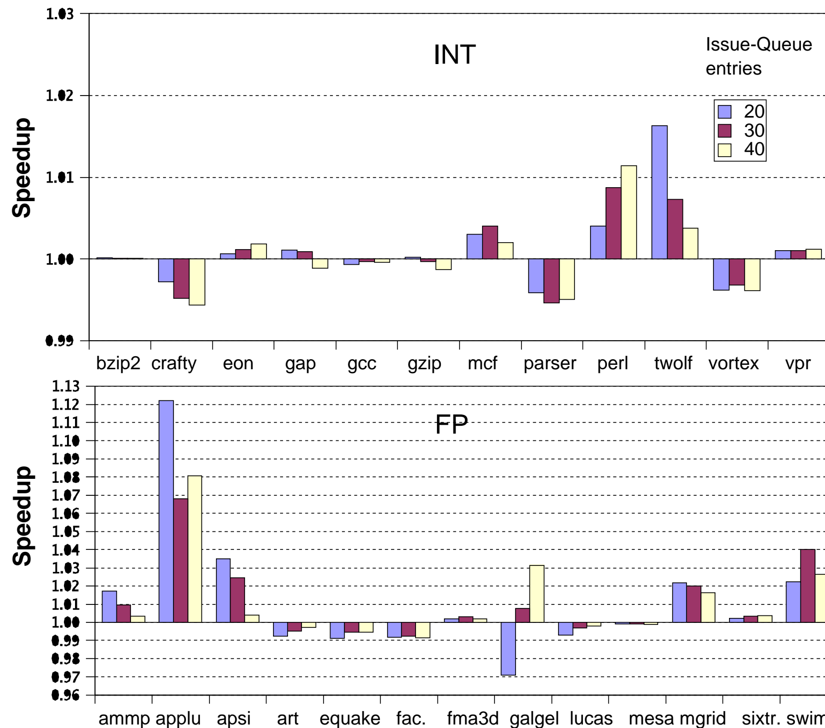

Fig. 6. Relative processor performance of our proposal with respect to the Baseline processor for each bench-

V. Related work

Several works ([11], [8], [12], [17], [2]) improve the management of the issue-queue entries assigned to instructions dependent on long-latency instructions. In thies works, instructions are extracted from the issue queue when an L1 or L2 miss is detected. After that, the extracted instructions are inserted in a buffer waiting for resolving the miss. Consequently, instructions dependent on misses are issued, at least, twice. Our proposal relays on delaying the insertion of instructions dependent on L2 misses in the issue queue in order to issue the instructions only once.

Kessler [6], Yoaz et al. [18] and Peir et al. [14] proposed the use of a L1 hit-miss predictor to improve the scheduling of the instructions dependent on load instructions. The proposals differ in the hit-miss prediction mechanism. While Kessler implements the predictor with a a 4-bit saturating counter, Yoaz et al. adapted a predictor used in branch prediction (the local predictor), and Peir et al. employed a bloom filter indexed by some bits of the effective address of the load instruction. Peir's proposal accesses the hit-miss predictor after issuing the load instructions and obtain the prediction just in time to prevent waking-up the instructions dependent on loads predicted to miss. Our proposal differs from all these proposals because we use L2 hit-miss prediction to delay the insertion of instructions in the issue queue .

Memik et al. [9] proposed the use of hit-miss prediction to reduce the access times and power consumption in processors with multi-level caches. After computing the effective address of a load instruction, they predict its location in the cache hierarchy; consequently, only the predicted level is accessed, reducing the access time and the power consumption with respect to a conventional hierarchy.

Liu et al. [15] perform L1- and L2 hit-miss prediction in order to estimate the issue cycle of all instructions. This estimation is used to preschedule the instructions. The authors use a hit-miss predictor feed by an address predictor of the effective addresses computed by the load instructions. Our work differs from this proposal in two main aspects: first, our hit-miss predictor does not rely on address prediction; second, we do not preschedule the instructions according to the prediction.

VI. Conclusions

The baseline processor considered in this paper is a processor that extracts from the issue queue the instructions dependent on load instructions that miss L2 and, after the L2 miss is

resolved, re-inserts the instructions into the issue queue in order to be re-issued (replayed). In this context, the goal of this paper is reducing the number of replays due to load instructions that miss in L2.

We have proposed predicting whether a load instruction will miss L2 or not, in order to delay the insertion into the issue queue of the instructions dependent on the load instructions predicted to miss L2. To perform L2 hit-miss prediction we use a perceptron predictor that is accessed in early stages of the pipeline using the instruction address of the load instructions. Our evaluations show a reduction in the number of instruction replays between 27% and 31% in integer benchmarks and between 33% and 39% in floating-point benchmarks without harming processor performance. Our future work includes evaluating the energy savings achieved through the reduction in instruction replays.

ACKNOWLEDGMENTS

This work was supported by the spanish government (TIN2004-07739-C02-01).

REFERENCES

[1] Todd Austin , Eric Larson and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling, Computer, v.35(2), p.59-67, 2002

[2] A. Cristal, D. Ortega, J. Llosa and M. Valero. Out-of-order commit processors. 10th HPCA, 2004

[3] Dan Ernst, Andrew Hamel and Todd Austin. Cyclone: a broadcast-free dynamic instruction scheduler with selective replay, Proceedings of the 30th annual international symposium on Computer architecture, June 09-11, 2003, San Diego, California

[4] Daniel A. Jiménez and Calvin Lin. Dynamic Branch Prediction with Perceptrons, Proceedings of the 7th HPCA , p.197, January 20-24, 2001

[5] Daniel A. Jiménez. Piecewise Linear Branch Prediction, Proceedings of the 32nd International Symposium on Computer Architecture, 2005

[6] R. E. Kessler. The Alpha 21264 Microprocessor, IEEE Micro, v.19 n.2, p.24-36, March 1999

[7] I. Kim and M.H. Lipasti. Understanding Scheduling Replay Schemes. In Proceedings of the 10th HPCA, 2004.

[8] Alvin R. Lebeck , Jinson Koppanalil , Tong Li , Jaidev Patwardhan and Eric Rotenberg. A large, fast instruction window for tolerating cache misses, Proceedings of the 29th annual international symposium on Computer architecture, p.59, May 25-29, 2002, Anchorage, Alaska

[9] Gokhan Memik , Glenn Reinman and William H. Mangione-Smith. Just Say No: Benefits of Early Cache Miss Determination, Proceedings of the The Ninth International Symposium on High-Performance Computer Architecture (HPCA'03), p.307, February 08-12, 2003

[10] Enric Morancho , José María Llabería and Àngel Olivé. Recovery Mechanism for Latency Misprediction, Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques, p.118, September 08-12, 2001

[11] T. Moreshet and R.I. Bahar. Complexity-Effective Issue Queue Design under Load-Hit Speculation, Proc. Workshop Complexity-Effective Design, 2002

[12] Onur Mutlu , Jared Stark , Chris Wilkerson and Yale N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors, Proceedings of the the 9th HPCA, p.129, February 08-12, 2003

[13] Subbarao Palacharla, Norman P. Jouppi and J. E. Smith. Complexity-effective superscalar processors, Proceedings of the 24th annual international symposium on Computer architecture, p. 206-218, June 01-04, 1997, Denver, Colorado, United States

[14] Peir, J., Lai, S., Lu, S., Stark, J., and Lai, K. Bloom filtering cache misses for accurate data speculation and prefetching. In Proceedings of the 16th international Conference on Supercomputing (2002), p. 189-198.

[15] Liu, Y., Shayesteh, A., Memik, G., and Reinman, G. . Scaling the issue window with look-ahead latency prediction. In Proceedings of the 18th Annual international Conference on Supercomputing (2004), p. 217-226

[16] Timothy Sherwood , Erez Perelman and Brad Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications, Proceedings of the 2001 PACT, p.3-14, September 08-12, 2001

[17] Srikanth T. Srinivasan , Ravi Rajwar , Haitham Akkary , Amit Gandhi and Mike Upton. Continual flow pipelines, Proceedings of the 11th international conference on Architectural support for programming languages and operating systems, October 07-13, 2004

[18] Adi Yoaz, Mattan Erez, Ronny Ronen and Stephan Jourdan. Speculation techniques for improving load related instruction scheduling, Proceedings of the 26th annual international symposium on Computer architecture, p.42-53, May 01-04, 1999