

Looking at History to Filter Allocations in Prediction Tables

Enric Morancho, José María Llabería and Àngel Olivé

Departament d'Arquitectura de Computadores

Universitat Politècnica de Catalunya, Barcelona, Spain

{enricm,llaberia,angel}@ac.upc.es

Abstract

Dependencies between instructions impose an execution order that must be preserved to guarantee the semantic correctness of programs. Recent works propose the use of prediction techniques to speculatively execute dependent operations, showing a significant increment in IPC.

We propose a mechanism that reduces the area cost of a typical address predictor: the last-address predictor. Our proposal classifies load instructions at run-time and records the classifications in a table with more entries than the prediction table. Moreover, it uses this information to initialize its confidence information and to filter the allocation of the load instructions in the prediction table. Using direct-mapped tables, our proposal captures a similar predictability and increases the accuracy of the typical address predictor, and represents around a 40% area-cost saving.

1. Introduction

Dependencies between instructions restrict the instruction-level parallelism, and make difficult for the processor to exploit the available hardware parallelism. To overcome dependencies, several kinds of hardware techniques have been proposed, ie register renaming, out-of-order execution [7] and prediction techniques.

Prediction techniques predict the result of an operation prior to execute it. After that, the processor uses the prediction to speculatively issue dependent operations. Prediction techniques have been successfully applied to reduce the influence of control dependencies [11], but recently, some works have proposed the application of prediction techniques also to reduce the influence of data dependencies; these proposals have been evaluated to predict addresses computed by load instructions [2][6] and results computed by modifying-register instructions [10].

A predictor predicts a pattern of the operation results; to predict more patterns, the predictor must be more sophisticated; eg an hybrid predictor [1][2].

Predictor resources are used efficiently when they are assigned to predictable instructions. Filtering techniques [3][15] are applied to avoid some allocations in the prediction tables. Filtering is valuable to increase the performance of the predictor or to reduce its area cost.

We evaluate a technique that records classification information of the instructions evicted from the prediction table to filter the allocation of the unpredictable instructions in the prediction table. We will focus on the simplest predictor used by the hybrid predictors, the last-result predictor, and on predicting the addresses computed by the load instructions. The same idea can be applied to other predictors as stride and value predictors.

This paper is organized as follows. Section 2 shows the prediction model used in this work. Section 3 presents a benchmark characterization. Section 4 points some

considerations used to design our proposal (*Looking-Backward Predictor*). Section 5 evaluates some performance metrics of our proposal, and compares them with the ones of a simple address predictor and other filtering predictors. Section 6 reviews related works. Finally, Section 7 summarizes our conclusions.

2. Last-Address prediction model

The last-address predictor [16] predicts addresses computed by the load instructions. It performs a trivial operation: the predicted address is equal to the previous address computed by the same load instruction. These previous addresses are recorded in a prediction table where each load is related to a set of table entries.

As every misprediction could have a penalty of some processor cycles, a prediction only takes place when it exists high confidence in the correctness of the prediction. Then, we will add a two-bit saturated counter to every prediction-table entry; these counters will be named confidence counters. Every time a load instruction computes the same address than the one computed in its previous execution, its confidence-counter value is increased by one, otherwise it is decreased by one. A load instruction is predicted only when its confidence-counter value is greater than one.

Proposed implementations of predictors based in the last-address prediction model [6][9] employ a direct-mapped table, named Address Table (AT), indexed with the least-significant bits of the PC. Every AT entry contains the last address computed by the most-recently executed load instruction mapped to the entry, a two-bit confidence counter, and a tag used to detect mapping conflicts. Load instructions are allocated in the AT using the *always allocate* policy; on a miss the load instruction is not predicted, the address field is updated and its confidence-counter value is set to one. This predictor will be named *Base Predictor* (BP); Figure 1 shows its scheme.

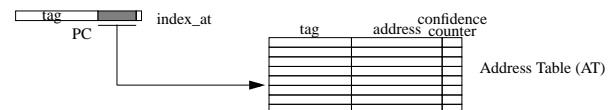


Figure 1: Base Predictor scheme.

To evaluate the performance of an address predictor, we use two measures: (a) the **captured address predictability**, defined as the percent of correct predictions out of the number of executed load instructions, and (b) the **accuracy**, defined as the percent of correct predictions out of the number of predictions performed. Table 1 shows the address predictability and the accuracy obtained using a BP with an unbounded AT in the integer benchmarks of the spec-95 suite. We focus on integer codes because they are more sensitive to the load-instruction latency than floating-point codes [7].

Binaries used in this work have been obtained compiling with the `-O4` switch of the `cc` native compiler of the machine

(an Alpha 21164 processor), and linking statically. Then, they have been instrumented with ATOM to evaluate the performance of the predictors. Benchmarks were run until completion, and loads to zero registers have been ignored.

Table 1: Benchmark description, input data set, number of static load instructions executed at least once, number of dynamic load instructions, and performance (address predictability and accuracy) obtained using the BP with an unbounded AT. Measures taken on an Alpha processor.

Benchmark	Input Set	Static Loads exec. ($\times 10^3$)	Dyn. loads ($\times 10^9$)	Last address predict.	Accur.
<i>go</i>	Reference	16.3	8.5	47.81	92.40
<i>mksim</i>	Reference	3.7	19.3	69.53	96.89
<i>gcc</i>	cp-decl.i	21.3	0.1	60.11	95.03
<i>compress</i>	Reference	0.7	12.5	57.27	99.72
<i>perl</i>	primes.pl	5.1	3.0	79.19	97.61
<i>vortex</i>	Reference	19.5	22.8	56.28	93.90
<i>li</i>	Reference	2.4	18.6	30.20	94.11
<i>jpeg</i>	Reference	3.9	7.1	19.11	93.14

Base Predictor flaws

Proposed last-result predictors [2][6] similar to the BP do not record confidence information of the operations evicted from the AT; every time an operation is allocated in the AT, the predictor must start from scratch the evaluation of the confidence measure on the operation. This fact decreases the predictability captured by the predictor, specially if the number of executions of an operation from its allocation in the AT until its eviction (the hit-burst length) is small compared with the number of executions of the learning phase of the confidence measure. In next section we will evaluate the typical hit-burst lengths and we will propose a mechanism to reduce its influence on the predictor performance.

Another captured-predictability decrease is observed if an unpredictable operation evicts a predictable one from the AT. We will propose the use of a mechanism that filters the allocation of the unpredictable operations in the AT.

3. Program characterization

The performance of a bounded BP is related to the AT miss rate and to the number of consecutive executions of a load instruction from its allocation in AT until its eviction (the hit-burst length). A dynamic load instruction that does not match with the tag recorded in its related AT entry (an AT miss) will not be predicted. Then, to predict their future executions, the predictor start-up will waste, at least, the next execution of this load instruction. Moreover, in a mapping-conflict at the AT entry, the load instruction will be evicted from AT, the hit burst of the load instruction will finish, and the whole process must start again on the next execution of this load instruction.

Miss rate and working-set size evaluation

We have evaluated [14] the AT miss rate using both direct-mapped AT's and fully associative AT's with the Least Recently Used (LRU) replacement policy.

Then, we define the working-set size of static load instructions of a benchmark as the minimum two-power number of AT entries needed to achieve a 1% miss rate in a fully associative AT with LRU replacement policy.

After working-set size measures we classify benchmarks in four classes: small (*compress*; size ≤ 128), medium (*li*, *jpeg* and *perl*; between 256 and 512), large (*mksim*; 1.024) and extra-large (*go*, *gcc* and *vortex*; ≥ 2.048).

The working-set size of a benchmark is related to the number of AT entries needed by a BP to achieve a performance similar to the one obtained using an unbounded AT. In previous evaluations [2][3][6][9][15], typical AT sizes range from 1.024 to 4.096 entries and, in some cases, mapping is four associative. These sizes are big enough to capture the whole working set of load instructions of most integer benchmarks. In this paper, we also evaluate smaller AT sizes to know the behaviour of our proposed predictor when it is pressured by capacity and conflict misses, because the goal is filtering the allocation of unpredictable load instructions in the AT.

Hit-burst length evaluation

We define the hit-burst length as the number of consecutive executions of a static load instruction that are hits in AT. A hit burst of length N involves $N+1$ consecutive executions of the load instruction: the first one produces the allocation of the load instruction in AT (a miss in AT), and the remaining N executions are AT hits.

On a replacement in an AT entry, its confidence counter is set to one; then, the next execution of the allocated load instruction will not be predicted. The loss of predictability due to this initialization can be large if, usually, a load instruction is evicted from AT after few executions of this load instruction. For instance, consider a load instruction 100% predictable, and let its hit-burst length be two; the predictor is able to predict only one of the two executions of the hit burst, ie, predictability loss is 50%. On a hit-burst of length 10, the predictability loss is 10%.

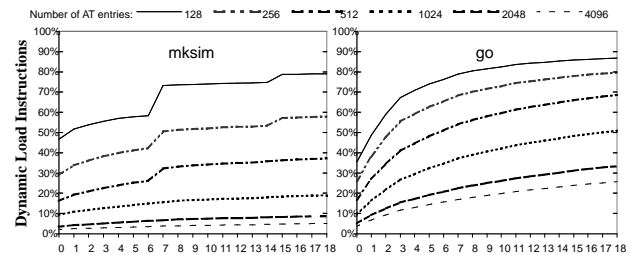


Figure 2: Cumulative dynamic-load distributions according to hit-burst lengths using direct-mapped AT's. Vertical axes stand for the percent of dynamic load instructions, horizontal axes stand for the hit-burst length. The horizontal axis is cut at hit-burst length 18, but all graphs saturate at 100%. Every line is related with a number of AT entries.

We have evaluated the number of occurrences of every hit-burst length, and they have been weighted by its number of involved executions. Figure 2 shows cumulative distributions of the hit-burst lengths in some benchmarks using several direct-mapped AT's. As the number of AT entries increases, the number of misses decreases, and the hit-burst lengths become larger.

We have observed that a significant amount of dynamic load instructions are related with short hit bursts. For instance, in benchmark *go*, for a 1.024-entry AT, 29% of dynamic load instructions is related with hit bursts of, at most, length four. To capture the potential predictability of these hit bursts, a predictor must be able to predict the load instructions as soon as they are allocated in AT.

Moreover, we have noticed that the percentage of dynamic load instructions related to zero-length hit bursts is significant in some benchmarks. For instance, in *mksim* and *go* using a 1.024-entry AT, they represent about 10% of the dynamic load instructions. These hit bursts are produced by load instructions that are allocated in AT and, before their next

execution, they are evicted. Although several benchmarks exhibit a similar miss rate, their hit-burst distributions present significant variations (Table 2).

Table 2: Distribution of dynamic load instructions related with hit-burst of length zero, up to four and up to ten executions in some configurations and benchmarks (numbers represent percents of dynamic load instructions). Miss rate in some benchmarks and configurations of a direct-mapped AT.

	256-entry <i>li</i>	1.024-entry <i>mksim</i>	2.048-entry <i>go</i>
hit burst length	0	7.54	9.4
	≤ 4	16.41	13.39
	≤ 10	23.65	16.89
Miss rate	12.69	11.63	11.72

To fully exploit the predictability available in short hit bursts, we propose recording confidence information of the evicted load instructions. This information will be used to initialize the confidence-counter value of the AT entries when the load instructions are reallocated in AT.

Predictability analysis

In [16] has been showed that the predictability of the load instructions do not spread uniformly among them. We have evaluated which load instructions contribute to the overall predictability. The main contribution is made by the highly predictable load instructions (between 82% and 97% of overall predictability) but, in other benchmarks (*go*, *vortex* and *gcc*), medium-predictable load instructions represent a significant contribution (up to 18% of overall predictability).

The allocation of unpredictable load instructions in AT does not contribute to the predictability captured by a predictor and, moreover, can hurt some predictability if it evicts a predictable load instruction. In this work, we present a mechanism that uses the confidence information of the evicted load instructions also to guide the replacement algorithm in AT.

4. Looking-Backward Predictor

In this section we present some design considerations that should be considered to design an address predictor. These considerations have been evaluated using unbounded prediction tables; detailed results can be obtained in [14]. After that, we describe the *Looking-Backward Predictor*.

4.1. Design Considerations

- **Recording the classification of the evicted load instructions:** Hit-burst length evaluations suggest us the addition of a Classification Table (CT) to the BP. CT records classification information of the evicted load instructions (predictable/unpredictable); the number of CT entries will be larger than the number of AT entries and CT will be indexed using the PC of the load instruction. CT information will be used to set the confidence counter of a load instruction on reallocation in AT. It will allow the predictor to exploit the predictability available in short hit bursts.
- **Filtering the allocation in AT by means of the CT:** We have evaluated the following filtering: a load instruction classified as unpredictable by CT will not replace a load instruction classified as predictable by AT; in any other case, the replacement will be performed. However, our evaluations show that this replacement algorithm is rough for medium-predictable load instructions, because reclassification is obstructed by predictable load instructions allocated in AT that have not been executed

for a long time.

- **Re-evaluation of the classification of the load instructions classified as unpredictable:** An opportunity to reclassify load instructions classified as unpredictable by CT is when they collide in AT with a predictable load instruction that has not been executed for a long time. We add a saturated 2-bit collision counter at every AT entry; its value is decreased on AT hits, and increased when unpredictable load instructions miss in AT. When the counter achieves its maximum value, the load instructions classified as unpredictable are allocated in AT to be reclassified
- **CT initialization:** Initializing all the CT entries as unpredictable is valuable because it filters load instructions with zero hit-burst lengths.

4.2. Predictor design

Figure 3 shows a scheme of the *Looking-Backward Predictor*. It uses two prediction tables: the AT and the CT; both tables will be direct mapped.

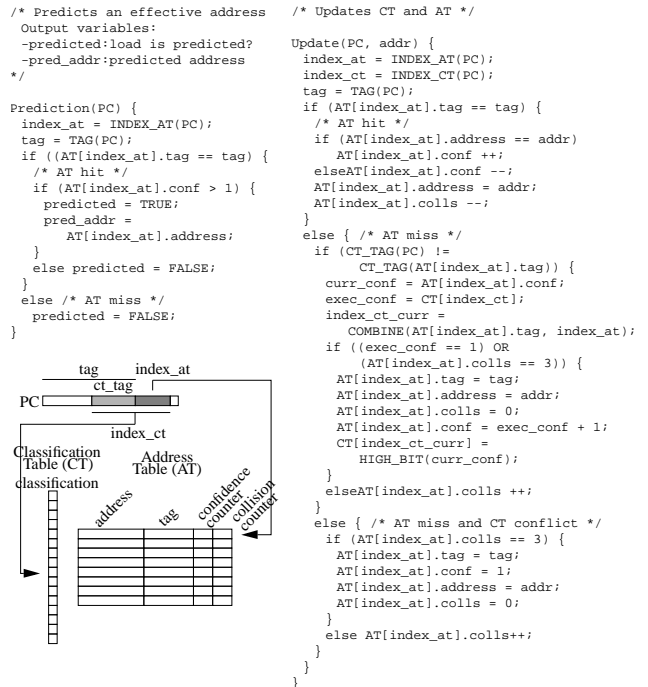


Figure 3: Looking-Backward Predictor scheme and pseudo code (operators ++ and -- update counter in a saturated way).

Each AT entry contains four fields: an address, a tag, a confidence-counter value and a collision-counter value. The address will be used to predict future addresses, the tag contains the bits of the PC that do not index AT (the lower bits of the tag identify the CT entry related to the AT entry), the confidence counter is used to decide if the load instruction allocated in the AT entry must be predicted, and the collision-counter value reflects the execution ratio between the allocated load instruction and the unpredictable ones that collide at the same AT entry. From now, the ratio between the number of CT entries and AT entries will be named the table-size ratio.

Each one-bit CT entry records the classification of a load instruction. We have decided the use of one-bit CT entries because recording the whole confidence counter is not a cost-effective alternative. Results were similar but, for a 1.024-entry AT and a table-size ratio of 8, two-bit CT entries

represent a cost increment around 10% (Section 5.1 presents an area-cost evaluation).

The predictor works as follows. When a load instruction is fetched, the predictor checks if the tag recorded in AT matches with some bits of the PC (AT hit). If so, the confidence-counter value is used to decide if the load instruction must be predicted. The procedure `Prediction` in Figure 3 shows the pseudo-code related to these actions.

The prediction tables are updated (using procedure `Update` in Figure 3) after the address stage of the pipeline, that is, immediately. Note that in the last-address predictor, misprediction-propagation issues are less important than in context predictors [1]. On an AT hit, the confidence-counter value is increased as in the BP; moreover, the computed address is recorded in the address field of the AT entry, and the collision-counter value is decreased by one. Note that in a implementation most prediction-table accesses can be performed in parallel.

On an AT miss, we use a simple replacement mechanism in AT. A load instruction classified by CT as predictable replaces immediately the load instruction allocated in the AT entry. However, if the load instruction is classified as unpredictable by CT, the replacement is only performed if the collision-counter value is 3; otherwise, the collision-counter value is increased by one.

On a AT replacement, AT fields are updated according to the executed load instruction, the collision counter is reset and the classification related to the evicted load instruction is recorded in CT. Values 0 and 1 classify the load instruction as unpredictable; values 2 and 3 as predictable.

On an AT miss produced by a load instruction that also collides in CT with the load instruction allocated in AT, the replacement mechanism only considers the collision counter to decide the allocation of the load instruction. On a replacement, the confidence counter will be set to one.

We will evaluate several table-size ratios: 2, 4, 8 and 16. Increasing table-size ratio produces a finer classification. That is, the classification will be improved when a predictable load instruction do not collide in CT using the larger ratio. The increase in captured predictability depends on the conflicts in AT; that is, for an AT size and large table-size ratios, the captured predictability saturates.

Our predictor takes advantage of the CT in two ways: a) to initialize the confidence-counter of AT entries on replacements, and b) as an input of the replacement mechanism of AT. As a result, filtering increases the hit-burst lengths related to the predictable load instructions, and reduces the loss of predictability produced by the learning phase of the confidence mechanism.

5. Performance evaluation

This section presents an evaluation of our proposal, comparing its area cost, predictability and accuracy with the ones of the BP. Moreover, we will compare our proposal with other filtering address predictors [3][15].

5.1. Area cost

The area cost of a predictor is estimated as the amount of bits of information stored in it. Following expressions of area cost of the BP and the *Looking-Backward Predictor* are function of the number of table entries; we assume 64-bit addresses, 2-bit confidence and collision counters, and 1-bit

CT entries.

$$\frac{\text{BasePredictor}}{\text{AreaCost}} = (64 + 2 + \text{tagbits}) \times \text{ATEntries}$$

$$\frac{\text{LookingBackwardPredictor}}{\text{AreaCost}} = 1 \times \text{CTEntries} + (64 + 2 + 2 + \text{tagbits}) \times \text{ATEntries}$$

5.2. Captured Predictability

We present a comparison between the predictability captured by the BP and the *Looking-Backward Predictor*.

We have performed simulations in a wide range of tagbits for every AT size to detect the number of tagbits needed to saturate the performance of the predictors [14]. We conclude that, in the evaluated benchmarks, both predictors should use up to 17 bits to tag and index AT.

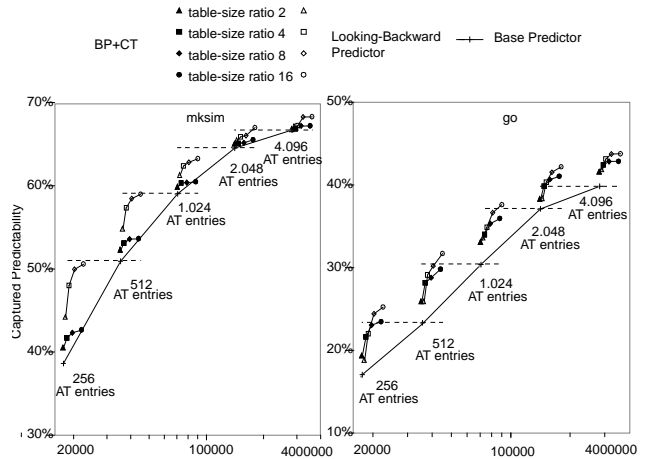


Figure 4: Predictability captured by the BP, the BP+CT and the Looking-Backward Predictor. Horizontal axes stand for the area cost (logarithm scale), vertical axes stand for captured predictability. A line connects the predictability captured by BP configurations. Other lines connect results for BP+CT (black points) and Looking-Backward Predictor (white points) for an AT size and several table-size ratios.

To show the influence on the performance of remembering the classification of the evicted instructions and filtering the allocation of unpredictable instructions, we present results for three cases: (1) the BP, (2) a BP with a CT that is only used to set the confidence counter on AT allocations, and (3) the *Looking-Backward Predictor*. Case (2) will be named BP+CT. Figure 4 shows these results for some benchmarks.

Comparisons between predictor configurations with the same number of AT entries

BP+CT always outperforms BP because it fully exploits the hit bursts of the predictable load instructions. The benefits of the BP+CT are significant in configurations with a great amount of short hit bursts (Table 2).

Differences between BP+CT and the *Looking-Backward Predictor* are due to the filtering performed by means of the CT. Filtering increases the captured predictability in almost all benchmarks. In some cases (*mksim*), the filtering effect is remarkable because there are a significant proportion of zero-length hit-bursts.

For every number of AT entries, the predictability captured by the *Looking-Backward Predictor* increases as the table-size ratio increases. The captured predictability saturates for large table-size ratios because it is limited by the number of AT entries and the mapping policy.

Comparisons between predictor configurations with a different number of AT entries

We compare the BP versus *Looking-Backward Predictor* configurations with half AT entries. From Figure 4, the *Looking-Backward Predictor* can capture more predictability than the BP with twice AT entries. Doubling the number of AT entries of the BP increases its captured predictability only if doubling produces that a predictable load instruction does not conflict with other load instructions of the same program-execution context. But in some cases, doubling the AT does not remove all the conflicts in an AT entry. The *Looking-Backward Predictor* tries to eliminate conflicts in AT by filtering the allocation of unpredictable load instructions; this filtering is successful if there is only one predictable load instruction colliding at the same AT entry. When two predictable load instructions collide at the same AT entry, the *Looking-Backward Predictor* can not capture their predictability, but doubling the AT may eliminate this conflict. In this case, the BP can outperform the *Looking-Backward Predictor* (*mksim* with 2.048 AT entries).

Most *Looking-Backward Predictor* configurations with table-size ratio 8 or 16 capture as much predictability or outperform the BP with twice AT entries. These results show that there is a significant proportion of conflicts between predictable and unpredictable instructions, then, doubling the AT of the BP is not a cost-effective solution.

Using as a reference the area cost of a BP with N AT entries, the area cost of a *Looking-Backward Predictor* with N/2 AT entries and table-size ratio 8 is around a 43% smaller; for table-size ratio 16 the reduction is about 36%.

Benchmarks *vortex* and *gcc* exhibit a similar behaviour than *go* due to their large working set of load instructions and the significant influence of medium-predictable load instructions. Benchmark *perl* is similar to *mksim* due to its large amount of conflicts in AT and its large predictability. Benchmarks *compress*, *li* and *jpeg* present saturated results due to their small working-set size.

The *Looking-Backward Predictor* can increase the performance of the BP as much as doubling its AT and with a smaller area-cost increment. Finally, our proposal gives a wide range of configurations to obtain the best fit to the available area.

5.3. Comparison of filtering strategies

We will compare our filtering strategy with the ones proposed in [3][15]. Their replacement algorithm uses confidence information of the allocated load instructions, and mapping-conflict counters. For this, a replacement counter is added to every AT entry of the BP; this counter is decreased on a correct prediction and increased on a misprediction or on a miss. Replacement takes place on saturation of the replacement counter; after that, the counter is set to zero. In this paper, we will name a predictor with this replacement algorithm a *Conservative Predictor* because it prioritizes the instructions allocated in AT versus a colliding instruction that misses in AT (even if it is predictable). That is, the *Conservative Predictor* does not use classification information of the instructions that collide as our replacement algorithm does. We will employ 3-bit replacement counters for the comparison because we have observed that they saturate the predictability captured by a *Conservative Predictor*.

We classify the benchmarks into two classes: (A) benchmarks with a high amount of load instructions that are

highly predictable (*mksim* and *perl*) and, (B) the remaining benchmarks. Now, we will comment the behaviour of the filtering strategies in both classes; we choose as representative of each class the benchmark with the largest working set of load instructions: *mksim* and *go*.

Figure 5 shows the predictability captured by the BP, the *Conservative Predictor* and our proposal for different numbers of AT entries using direct-mapped tables in benchmarks *mksim* and *go*. To evaluate our proposal, we have selected a table-size ratio equal to 16.

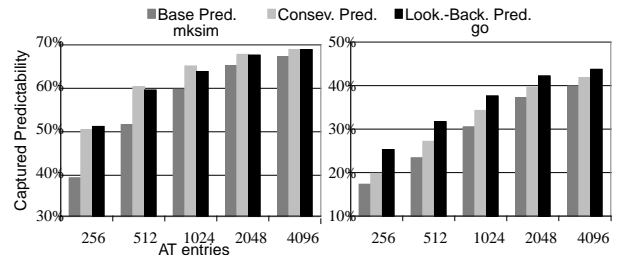


Figure 5: Predictability captured (vertical axis) by the BP, the *Conservative Predictor* and our proposal in benchmarks *mksim* and *go* using direct-mapped tables. Horizontal axis shows the number of AT entries of the configurations.

In *mksim*, most conflicts are between predictable load instructions. Both the *Conservative Predictor* and our proposal capture more predictability than the BP because they filter conflicts between predictable load instructions. However, for some AT sizes, our proposal shows a slightly decrease (2%) versus the *Conservative Predictor*.

The way how both replacement algorithms filter the predictable load instructions is different. The *Conservative Predictor* favours the allocated load instructions, then, a highly predictable instruction allocated in the AT will only be evicted on saturation of the replacement counter. On the other hand, our proposal filters the allocation of predictable load instructions as a side effect of initializing CT entries as unpredictable: a potentially predictable load instruction is classified in CT as unpredictable. Then, potentially predictable instructions that have a few conflicts with an allocated predictable instruction are not reclassified as predictable; if they have a bigger number of conflicts, their reclassification will be only delayed.

In both replacement algorithms, the allocation of predictable or potentially predictable instructions is favoured by collisions produced by unpredictable instructions. But, in our proposed algorithm, a instruction classified as predictable is allocated in AT without delay.

In benchmark *go*, both the *Conservative Predictor* and our proposal capture more predictability than the BP. However, our algorithm is able to capture more predictability than the *Conservative Predictor* because it prioritizes the predictable load instructions, that is, the allocation of load instructions classified as predictable is not delayed. On the other hand, the *Conservative Predictor* can delay the allocation of these load instructions in an execution-context change or in the same execution context. It is not delayed only if it collides with an unpredictable load instruction that has its replacement counter saturated. On the same execution context, unpredictable instructions can produce a delay chain that makes difficult the allocation of a predictable instruction.

We conclude that our replacement algorithm perform better than the replacement of the *Conservative Predictor* on benchmarks with a significant amount of conflicts between predictable and unpredictable load instructions. On

benchmarks with a conflicts between predictable load instructions, the loss of predictability of our proposal respect the *Conservative Predictor* is small.

To increase the performance of our proposal in benchmarks with a high number of conflicts between predictable load instructions we can take advantage of the results obtained in [8], and [12], but the evaluations of these possibilities is beyond the scope of this paper.

Comparing the area cost of both the *Conservative Predictor* and the *Looking-Backward Predictor* for the same number of AT entries, our proposal represents an increment around 20%. However, it is able to obtain the performance of a BP with twice AT entries; that represent a decrease around 40%.

5.4. Accuracy

We have also compared the accuracy of the BP, the *Conservative Predictor* and the *Looking-Backward Predictor*. Our results show that, for the same number of AT entries in most cases, the *Looking-Backward Predictor* is more accurate than the *Conservative Predictor*, and both are more accurate than the BP. The only exception is benchmark *mksim* for 256 and 1.024 AT entries. It is due to the large difference between the number of predictions performed by both predictors (Figure 4).

5.5. Associative mapping

We have compared the performance of the predictors using associative AT's. First, we have compared the *Looking-Backward Predictor* versus the BP. Our results show that, for the same number of AT entries and associativity, our proposal outperforms BP. Moreover, in benchmarks *go*, *gcc* and *vortex*, a direct-mapped *Looking-Backward Predictor* outperforms a two-way BP. Comparing the *Looking-Backward Predictor* versus the *Conservative Predictor*, we obtain than our proposal is better in benchmarks with a high percentage of medium-predictable load instructions. In benchmarks with a high percentage of highly predictable load instructions, the *Conservative Predictor* outperforms our proposal.

6. Related works

Some works that use the BP to speculatively execute dependent instructions [6][9] have shown a potential IPC improvement. Then, the BP used in these works can be replaced by the *Looking-Backward Predictor* because it exhibits a similar performance using a smaller area-cost.

Lipasti et al. [10] propose a value predictor that decouples the classification information from the value information, but classification information is not used to filter the allocation of value information.

Morancho et al. [13] record in the Classification Table some bits of the effective address to estimate the classification of the load instructions. This idea has also been applied to a context predictor in [1]. This proposal reduces the area cost of the BP around 19% and maintains its performance, while the *Looking-Backward Predictor* reduces the BP area cost up to 40%.

Mechanisms that detect the usefulness of the predictions to reduce the execution time have been proposed in [3][15]. These mechanisms are used to select the instructions that must be inserted in the AT. The *Looking-Backward Predictor* can improve their performance because we filter the allocation of unpredictable load instructions in the AT.

Filtering the allocation of information in the prediction tables has been used in the context of branch predictors [4][5]. These works propose hybrid predictors with priority

allocation in prediction tables, but, when no predictor is predicting a branch instruction with strong confidence, all predictors in the hybrid predictor are updated simultaneously, and resources are allocated for a branch instruction in all prediction tables. Our proposal can be included and used to filter the allocation of information in some prediction tables.

7. Conclusions

The addresses computed by the load instructions in the integer benchmarks exhibit a significant tendency to be predictable, but this tendency do not spread uniformly. Then, we propose classifying load instructions according to their predictability and the use of this information by the predictor to guide the replacement algorithm.

We have shown that with an area-cost saving around 40% respect the BP, the *Looking-Backward Predictor* captures similar predictability than the BP.

Our proposal can also be applied to other predictors as stride, value and branch target predictors, to reduce their area cost.

Acknowledgements

This work was supported by the Ministry of Education of Spain under grant CICYT TIC98-0511-C02-01, and the CEPBA (European Centre for Parallelism of Barcelona).

References

- [1] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz and U. Weiser. (1999). Correlated Load-Address Predictors. In *Proceedings of the 26th ISCA*.
- [2] B. Black, B. Mueller, S. Postal, R. Rakvic, N. Utamaphethai and J.P. Shen. (1998). Load Execution Latency Reduction. In *Proceedings of the 12th ICS*, pp. 29-36
- [3] B. Calder, G. Reinman and D.M. Tullsen. (1999). Selective Value Prediction. In *Proceedings of the 26th ISCA*.
- [4] K. Driesen and U. Holzle. (1998). The cascaded predictor: Economical and Adaptive Branch Target Prediction. In *Proceedings of the 31th Micro*.
- [5] A.N. Eden and T. Mudge. (1998). The YAGS Branch Predictor Scheme. In *Proceedings of the 31th Micro*.
- [6] J. González and A. González. (1997). Speculative Execution via Address Prediction and Data Prefetching. In *Proceedings of the 11th ICS*, pp. 196-203
- [7] J.L. Hennessy and D. Patterson. (1995). *Computer Architect a Quantitative Approach*. Second Edition. Morgan Kaufnab Publishers, Inc.
- [8] T. Juan, S. Sanjeevan and J. Navarro. (1998). Dynamic History-Length Fitting: A Third Level of Adaptivity for Branch Prediction. In *Proceedings of the 25th ISCA*, pp. 155-166.
- [9] M.H. Lipasti, C. B. Wilkerson and J.P. Shen. (1996). Value Locality and Load Value Prediction. In *Proceedings of the 7th ASPLOS*, pp. 138-147
- [10] M.H. Lipasti and J.P. Shen. (1996). Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Micro*, pp. 226-237
- [11] S. McFarling. (1993). *Combining Branch Predictors*. Western Research Laboratory (Digital). WRL-TN-36
- [12] S. McFarling. (1991). *Cache Replacement with Dynamic Exclusion*. Western Research Laboratory (Digital). WRL-TN-22
- [13] E. Morancho, J.M. Llabería and Á. Olivé. (1998). Split Last-Address Predictor. In *Proceedings of PACT'98*, pp. 230-237.
- [14] E. Morancho, J.M. Llabería and Á. Olivé. (1999). *Discrete Last-Address Predictor*. Universitat Politècnica de Catalunya, Depart. d'Arquitectura de Computadors, UPC-DAC-1999-1.
- [15] B. Rychlik, J.W. Faistl, B.P. Krug and J.P. Shen. (1998). Efficacy and performance impact of value prediction. In *Proceedings of the PACT'98*.
- [16] Y. Sazeides and J.E. Smith. (1996). The Predictability of Data Values. In *Proceedings of the 29th Micro*, pp. 238-247