Split Last-Address Predictor

Enric Morancho, José María Llabería, and Àngel Olivé
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
c/ Jordi Girona 1-3, Campus Nord, Edifici D6
08034 Barcelona, Spain
{enricm, llaberia, angel}@ac.upc.es

Split Last-Address Predictor

Enric Morancho, José María Llabería, and Àngel Olivé
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
c/ Jordi Girona 1-3, Campus Nord, Edifici D6
08034 Barcelona, Spain
{enricm, llaberia, angel}@ac.upc.es

Abstract

Recent works have proposed the use of prediction techniques to execute speculatively true data-dependent operations. However, the predictability of the operations do not spread uniformly among them. Then, we propose the use of run-time classification of instructions to increase the efficiency of the predictors. At run time, the proposed mechanism classifies instructions according to their predictability, decoupling this classification from prediction table. Then, the classification is used to avoid the unpredictable instructions from being candidates to allocate an entry in the prediction table.

The previous idea of run-time classification is applied to the last-address predictor (Split Last-Address Predictor). The goal of this predictor is to reduce the latency of load instructions. Memory access is performed after the effective address is predicted concurrently with instruction fetch, after that, next true data-dependent instructions can be executed speculatively. We show that our proposal applied to the last-address predictor captures the same predictability than the last-address predictor proposed in literature, increases its accuracy, and reduces its area-cost a 19%.

1 Introduction

The hardware parallelism of processors is not fully exploited because data and control dependencies limit the instruction-level parallelism of programs. Dependencies impose an execution order between instructions that must be preserved to guarantee the semantic correctness of programs.

To reduce the restrictions imposed by dependencies, several techniques have been proposed [8]. For instance, prediction techniques; they predict the result of an operation prior to execute it. They have been mainly used to reduce the influence of conditional branches [11]. In recent works, prediction techniques have also been applied to predict values or addresses to speculatively issue dependent operations [2][7][9][10].

Prediction techniques can be classified as stale-less and as state-needed. State-less prediction does not use results of previous executions of the operation to predict it. Some examples of this kind of prediction are static branch prediction, prefetch on miss, and the use of a simple operation to predict the effective addresses accessed by load instructions [2].

In state-needed prediction, the results of previous instances of an operation are recorded. Later, applying a prediction formula, they are used to predict the result of the next instance of the operation. This process is dynamic and adaptive since it collects and studies information at run time. Some examples of state-needed prediction are dynamic branch prediction [11], stride-address prediction [3] and value prediction [7][9][10].

Prediction tables are used to record the information needed to perform the predictions. Usually, direct mapping is employed to relate an operation to a table entry; as these tables are indexed using some significant bits of the program counter, they can be accessed early in the pipeline.

Some works [5][7][9][10] have evaluated the potential performance improvement of the speculative execution of instructions that data-depend on a predicted operation. On a right prediction, the effective latency of the operation can be reduced several processor cycles. However, on a misprediction, the operation must be re-executed, and dependent instructions must be flushed out and re-executed. Then, as these recovery operations could have a penalty of some processor cycles, the predictor should perform a minimum number of mispredictions.

A decrease in performance is observed if a predictable operation is replaced in the prediction table by an unpredictable operation. The predictability of operations do not spread uniformly among them (Section 2.1, [5]), then, classifying operations according to their predictability may avoid the unpredictable operations from being candidates to allocate an entry in the prediction table, and allows a better use of the prediction-table resources.

To filter the unpredictable operations, we propose the use of a dynamic mechanism for classifying operations. This mechanism is decoupled from prediction table, and it selects the operations that can allocate an entry in the prediction table. This proposal will be called *Split Predictor* and it will be applied to a last-address predictor. Performance studies show that this proposal captures the same predictability than the last-address predictor proposed in literature, increases its accuracy, and reduces its area-cost a 19%.

This paper is organized as follows. Section 2 shows an analysis of the distribution of the address predictability. Section 3 presents a dynamic mechanism for classifying load instructions. Section 4 describes the *Split Last-Address Predictor*, and in Section 5 it is evaluated. Section 6 reviews related works and, finally, the conclusions of this work are summarized in Section 7.

2 Prediction model

This section shows the distribution of address predictability among load instructions using the last-address predictor in some benchmarks. This predictor performs a trivial computational operation, the predicted address is equal to the last generated address. It can be implemented using a prediction table where each entry contains the last address computed by a load instruction related to this entry.

Table 1: Benchmark analysed, input data set, number of static load instructions (present in the binary file and executed at least once), number of dynamic load instructions, and maximum address predictability capturable by the last-address predictor. Measures taken on an Alpha processor.

Benchmark	Input data set	Stat. load pres/exec ($\times 10^3$)	Dyn. load ($\times 10^9$)	Last address predic.
go	Reference	21 / 16.3	8.5	52.66
mksim	Reference	13 / 3.7	19.3	71.89
gcc	cp-decl.i	97 / 21.3	0.1	64.86
compress	Reference	4 / 0.7	12.5	57.66
li	Reference	7.7 / 2.4	18.6	35.55
ijpeg	Reference	15 / 3.9	7.1	21.71
perl	primes.pl	43 / 5.1	3.0	79.22
vortex	Reference	43 / 19.5	22.8	62.65
anagram	25K-word lexicon, 3 input words	3 / 0.8	0.005	63.31
bc	Primality test	5.7 / 1.8	0.004	56.66
ks	100 nodes, 50 nets graph	5 / 1	0.005	6.65

To evaluate the effectiveness of an address predictor, we define the address predictability captured by an address predictor as the percent of the number of correct predictions out of the number of executed load instructions. Using an unbounded prediction table, Table 1 shows the address predictability that can be captured by the last-address predictor in the spec-95 and other integer benchmarks [1]. In this work we focus on integer codes because they are more sensitive to the effective latency of load instructions than floating-point codes [8].

Table 1 shows that, except for *ks* benchmark, the last-address predictor predicts correctly between 21% and 79% of effective addresses computed by load instructions.

In this work we propose an implementation of the idea of run-time classification applied to the last-address predictor. The same idea can be applied to another typical computational predictor, the stride-address predictor [3][7].

2.1 Distribution of last-address predictability

This subsection presents an analysis of the contribution of the load instructions to the overall last-address predictability (throughout this paper, it will be referred as predictability). We show that the predictability of load instructions do not spread uniformly among them. That is, a significant number of static load instructions is highly predictable or highly unpredictable.

For each load instruction, we have evaluated its predictability using an unbounded prediction table, then we have grouped load instructions in ranges of predictability. Figure 1 presents the distribution of the static load instructions according to their predictability. It fades out from the highly predictable load instructions (90%-100% range, at the bottom of each bar), to the highly unpredictable ones (0%-10% range, at the top of each bar). Between 25% (*anagram*) and 60% (*ijpeg*) of static load instructions can be classified as highly predictable, and between 20% (*go*) and 45% (*ks*) as highly unpredictable.

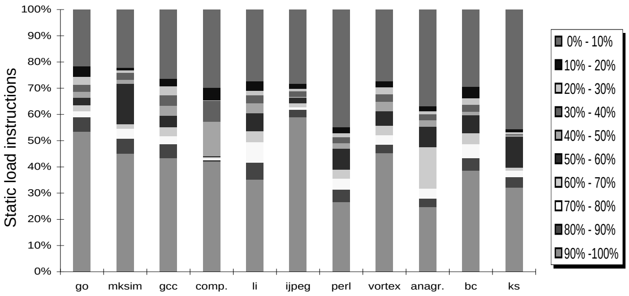


Fig. 1. **Static load-instruction distribution according to their predictability.**

To show the contribution of each predictability range on the number of executed load instructions, every static load instruction has been weighted with its execution frequency; Figure 2 shows this distribution. In some benchmarks,

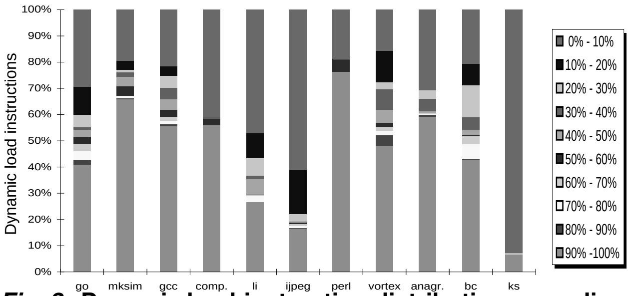


Fig. 2. **Dynamic load-instruction distribution according to their predictability.**

highly predictable and highly unpredictable load instructions represent almost all executed loads (*compress*, *perl*, *ks*); in other benchmarks, load instructions with a medium predictability account for a significant proportion of the number of executed load instructions (*go*, *vortex*, *bc*). The static and the dynamic load-instruction distributions can differ significantly (for instance, in benchmark *ijpeg*, 60% of static load instructions are highly predictable but they only represent 16% of executed load instructions).The absolute contribution of every predictability range to the captured predictability show us the significance of medium-predictable load instructions to the predictability. Left bars related to benchmarks in Figure 3 present the contribution of every load-instruction range to the predictability captured by a last-address predictor. The main contribution is made by static load instructions in range 90%-100%, they account between 75% and 97% of the overall predictability. The static load instructions in range 70%-90% represent a small contribution, they account, at most, for the 8% of the overall predictability. Remaining overall predictability is produced by static load instructions that belong to lower predictability ranges; for instance, the contribution of load instructions in 10%-60% range to the overall predictability varies from 18% to 2%.

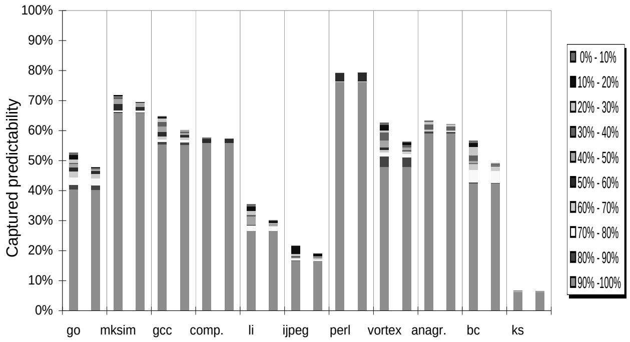


Fig. 3. **Predictability captured by the nonclassifying last-address predictor (left bar) and by the classifying one (right bar).**

3 Classification of load instructions

In Section 3.1 we will present a mechanism to classify load instructions according to their address predictability. After that, in Section 4, we will present a predictor that uses load-instruction classification to filter unpredictable load instructions from being candidates to allocate an entry in a bounded prediction table; this classification will allow us to increase the performance of the predictor, because our address predictor will record addresses computed only by load instructions classified as predictable.

We use two-bit saturated counters as a dynamic classifying mechanism. A counter is assigned to each prediction-table entry. Every time a load instruction computes the same effective address than its previous execution, the counter value will be increased by one, otherwise it will be decreased by one. Also, to predict a load instruction, its saturated counter value must be greater than one.

Classifying counters detect load instructions that, on a burst consecutive executions, compute the same address; these bursts will be called predictable bursts. When a predictable burst is detected, the load instruction is classified as predictable and, on next executions, it is predicted; when the predictable burst finishes, the instruction is classified as unpredictable until a new predictable burst is detected. As the classifying predictor needs some executions of the load instruction to detect a predictable burst, some predictability of the load instruction is not captured. Moreover, when a short predictable burst is detected, it can be over or almost finished. Short predictable bursts are mainly produced by load instructions with medium or low predictability.

Two-bit saturated counters classify correctly highly predictable and highly unpredictable load instructions, but the ones with medium or low predictability can be classified in a wrong way. Then, the predictability captured by the classifying predictor can be smaller than the predictability captured by the nonclassifying one. Using unbounded tables, Figure 3 shows the predictability captured by the nonclassifying (left bar) and by the classifying predictor (right bar) distributed according to the load-instruction predictability ranges. The decrease in the captured predictability attributable to the classifying mechanism is related with the contribution of load instructions with medium or low predictability to overall predictability, the classifying predictor is not able to fully capture their predictability. This reduction can account up to 15% of the predictability captured by the nonclassifying predictor.

3.1 *N*-bit classifying mechanism

All bits of the effective addresses are needed to perform a memory access, but are not needed to classify a load instruction. We propose using few bits of the effective addresses (*N* bits) as input of the load classifier.

To classify a load instruction as nonpredictable by a last-address predictor, it is sufficient to detect that one bit of the last computed addresses is different. It follows that few bits (*N*) of the computed addresses can be enough to classify load instructions (for instance the least-significant bits). If one of these bits is different, the classification will be correct (equal to the classification performed comparing all bits of the computed addresses). Wrong classifications will be produced, for instance, if addresses follow an arithmetic progression and the analysed bits are not modified; this load instruction will be classified as predictable but it is not.

Also, the mapping of language data types onto architectural data types performed by the compiler can produce that some low-order bits of the computed addresses are not significant for classifying most load instructions.

Then, we propose a classifying mechanism named <*N*,*k*>, where *N* is the number of bits used to classify load instructions, and *k* is the number of low-order bits discarded of the computed addresses. The classifier skips the *k* low-order bits of the computed addresses and then selects the *N* low-order bits.

To compare the classifying mechanism that use all bits of the effective address (<all,0>) and a <*N*,*k*> classifying mechanism, we define the similarity between them as the percent of coincidences of the classifying counters (every time a load is executed, the classifying-counter values related to this load are checked) out of the number of executed loads.

In this work we have used load-instruction traces taken from an 21164 Alpha-AXP processor. As the fundamental unit of data of Alpha architecture is 8 bytes [4], we have evaluated the discarding of the 3 low-order bits of the computed addresses for classifying load instructions.

Figure 4 shows the average similarity in all benchmarks between <all,0> and two *N*-bit mechanisms: no-skipping (<*N*,0>) and 3-bit skipping (<*N*,3>). Mechanism <*N*,3> does not take advantage of selecting more than five bits due to eight-byte unaligned computed addresses; its similarity graph is saturated about 90%. To achieve a similarity greater than 90%, the three low-order bits must be selected as is shown in the graph of <*N*,0>.

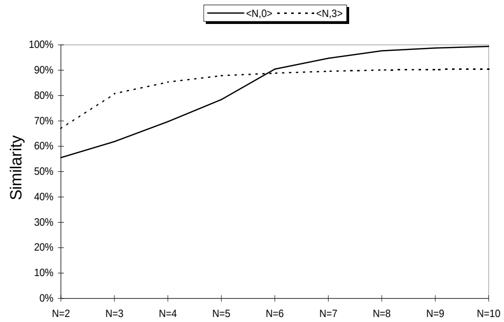


Fig. 4. **Average similarity in all benchmarks between two *N*-bit classifying mechanisms (<N,0> and <N,3>) and the <all,0> classifying mechanism.**

Cases <3,3> and <4,3> obtain a high similarity (over *80%)* with <all,0>. Moreover, <4,3> achieves almost the same similarity than <6,0>. We will use the <3,3> classifier mechanism in our proposed predictor; the similarity of this classifier is about 80%.

To improve the similarity between our proposed classifier and the <all,0> mechanism, the operation code of load instructions can be used to decide dynamically the number of skipped bits. In this paper we do not use this improvement.

4 Split Last-Address Predictor

This section describes a predictor mechanism with run-time classification. It takes advantage of two considerations: a) few bits of the effective addresses are enough to classify precisely load instructions, and b) it is necessary to record the whole effective address accessed by a load instruction only when this load is predictable. Following these considerations, we propose to split the prediction table into two tables: the Classification Table (CT) and the Address Table (AT). CT is used to classify dynamically load instructions according to their predictability. AT stores the last effective addresses computed by predictable load instructions. Figure 5 shows an scheme of the mechanism, it will be named *Split Last-Address Predictor* or *Split Predictor*.

To classify dynamically load instructions, the predictor uses the <*N,k*> strategy described in Section 3.1. CT is direct mapped and each entry contains two fields: a two-bit saturated counter, and *N* bits of the effective address. The counter is used to classify load instructions continuously, that is, each executed load updates the CT.

AT is also direct mapped and each entry contains a complete effective address and a *ct_tag*; this tag identifies the CT entry related to an AT entry.

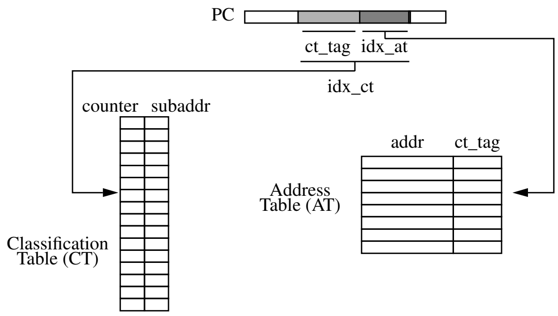


Fig. 5. Split Last-Address Predictor **scheme.**

The proposed predictor avoids the placement of highly unpredictable load instructions in the AT using the information recorded in CT. Load instructions that can be placed in AT are filtered using CT: their saturated counter must be greater than 1. That allows to the predictable load instructions to continue placed in AT, and gives more chances to exploit their predictability.

```
/* Predicts an effective address
Output variables:
-predicted: a prediction
is suggested
-pred_addr: predicted address
*/

void Prediction(PC) {
 idx_at = INDEX_AT(PC);
 idx_ct = INDEX_CT(PC);
 ct_tag = CT_TAG(PC);
 predicted =
   ((AT[idx_at].ct_tag==ct_tag)
   &&
   (CT[idx_ct].counter > 1));
 pred_addr = AT[idx_at].addr;
}

Note: Operators ++ and -- update
the counter in a saturated way.
```

```
/* Updates CT and AT */

void Update(PC, addr) {
 idx_at = INDEX_AT(PC);
 idx_ct = INDEX_CT(PC);
 ct_tag = CT_TAG(PC);
 subaddr = SELECT_N_BITS(addr);
 if (AT[idx_at].ct_tag==ct_tag) {
  if (AT[idx_at].addr==addr)
      CT[idx_ct].counter ++;
  else CT[idx_ct].counter --;
  AT[idx_at].addr = addr;
 }
 else {
  if (CT[idx_ct].counter > 1) {
    AT[idx_at].ct_tag = ct_tag;
    AT[idx_at].addr = addr;
  }
  if (CT[idx_ct].subaddr==subaddr)
      CT[idx_ct].counter ++;
  else CT[idx_ct].counter --;
 }
 CT[index_ct].subaddr = subaddr;
}
```

Fig. 6. Split Last-Address Predictor **pseudo-code.**

The predictor works as follows. When a load instruction is fetched, the appropriate CT and AT entries are selected, and the *ct_tag* field is checked to determine if the AT entry is related to this CT entry. If so, the counter value in CT is used to decide if the load is predicted, otherwise it is not predicted. The procedure `Prediction` in Figure 6 shows the pseudo-code related to these actions.

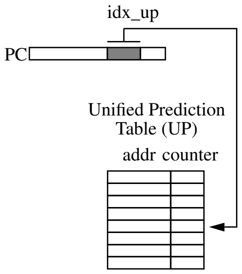
The prediction tables are updated after the address stage of the pipeline (procedure `Update` in Figure 6 shows its pseudo-code). On an AT hit, the selected entry is updated using the computed effective address. On an AT miss, the counter value in CT entry is checked to decide if the current AT entry is replaced.

5 Performance evaluation

5.1 Baseline Predictor

As a baseline last-address predictor we will use the *Unified Predictor* (UP). It employs a direct-mapped prediction table where each field contains the last effective address computed by the load instruction related to the entry, and a two-bit saturated counter. This counter is used to decide if a load instruction can be predicted, and it is updated using the <all,0> classifying mechanism. Load instructions are allocated in the UP using an always-allocate policy. Figure 7 details the scheme and the pseudo-code of this predictor.

The *Unified Predictor* is similar to the predictors used in some previous works [7][9], and our proposed predictor (*Split Predictor*) will be compared to the *Unified Predictor.*




```
void Prediction(PC) {
  idx_up= INDEX_UP(PC);
  predicted = UP[index_up].counter > 1;
  pred_addr = UP[index_up].addr;
}

void Update(PC, addr) {
  idx_up = INDEX_UP(PC);
  if (UP[idx_up].addr == addr)
       UP[idx_up].counter ++;
  else UP[idx_up].counter --;
  UP[idx_up].addr = addr;
}
```

***Fig. 7. Unified Predictor* scheme and pseudo-code.**

5.2 Simulation environment

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, with OSF1 V3.2). Then, they have been instrumented with ATOM (this tool is able to instrument user-level code, but does not instrument operating-system code) to evaluate the performance of the predictors. Benchmarks were run until completion.

5.3 Working-set size of load instructions

We evaluate the working-set size of load instructions of a benchmark as the minimum two-power size of an LRU fully associative table with a 99% hit rate. Table 2 presents a classification of the benchmarks according to their working-set size. In this paper we present results for large and extra-large benchmarks. The others benchmarks show similar behaviours when the working-set size of load instruction is considered.

Table 2: Benchmark classification according to their working-set sizes of load instructions.

Class	Benchmarks	Working-set size
Small	*compress, anagram, ks*	≤ 128
Medium	*li, ijpeg, perl*	256 - 512
Large	*mksim, bc*	1.024
Extra-Large	*go, gcc, vortex*	2.048 - 8.192

The predictability captured by a predictor is related with the load-instruction miss rate in the prediction tables. Then, we will show the miss-rate improvement obtained in the *Split-Predictor* when the amount of CT entries is increased. From these results and the area-cost model of the predictors, we will obtain a ratio between the amount of CT entries and AT entries that improves the miss rate and saves area cost respect the *Unified Predictor.*

5.4 Miss rate of the *Split Predictor*

Using the *Unified Predictor* every load instruction is allocated in UP, but using the *Split Predictor* only a portion of all load instructions are allocated in AT, because CT filters the load instructions that can be placed in AT. So, we expect that CT-filtering will reduce the amount of capacity misses in AT compared to the capacity misses in UP for the same amount of AT and UP entries.

To measure this reduction, we evaluate the miss rate of both predictors. The miss rate of the *Unified Predictor* is the percent of load instructions that miss in UP out of the total number of load instructions. On the other hand, two cases produce a miss in the *Split Predictor*: a) a load instruction that misses in CT and b) a load instruction that hits in CT, that is classified as predictable by CT, and that misses in AT. The miss rate of the *Split Predictor* is the percent of misses in the *Split-Predictor* tables out of the number of load instructions.

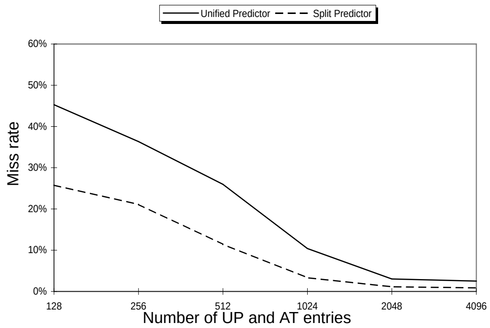


***Fig. 8.* Miss rate in the *Unified Predictor* and the *Split Predictor* in benchmark *go*. UPs and ATs are fully associative with an LRU replacement policy. CTs are unbounded.**

First, we evaluate the miss rate of both predictors when they are implemented using fully associative ATs and UPs with LRU replacement policy, unbounded CTs, and the <all,0> classifying mechanism; to compute both miss rates we have employed fully tagged tables. These miss rates can be assumed as capacity miss rates. Figure 8 shows the miss rate of some configurations of the *Unified Predictor* and the *Split Predictor* in benchmark *go*; horizontal axe represents the amount of AT and UP entries, vertical axe shows the miss rate. The remaining benchmarks exhibit similar behaviours but in a different table-entry range.

From Figure 8, recording the same number of effective addresses, the *Split Predictor* has a lower miss rate than the *Unified Predictor* because CT filters the load instructions that can be placed in AT. As AT and UP become larger, the miss-rate reduction is less significant due to the increment of capacity in both tables. Similar miss rates are achieved using a *Split Predictor* and an *Unified Predictor* where AT entries is equal to half of UP entries; for instance, the *Split Predictor* with 512 AT entries and the *Unified Predictor* with 1.024 UP entries achieve about a 10% miss rate.

Next, we evaluate the influence of the mapping policy and the CT size on the miss rate; ATs, UPs and CTs will be direct mapped. Bounded and direct-mapped CTs increase the miss rate of the *Split Predictor* due to capacity and conflict misses in CT.

Figure 9 displays the miss rate of some configurations of the *Split Predictor* and the *Unified Predictor* in benchmark *go* for several table-entry ratios (CT entries/AT entries); from now on they will be called ratios. Horizontal axe represents the number of UP and AT entries, vertical axe shows the miss rate. A configuration with ratio equal to 1 is a degenerated *Split Predictor* configuration; then the minimum analysed ratio is 2 and the maximum is 16. Also, in Figure 9 is shown the miss rate for unbounded CT tables.

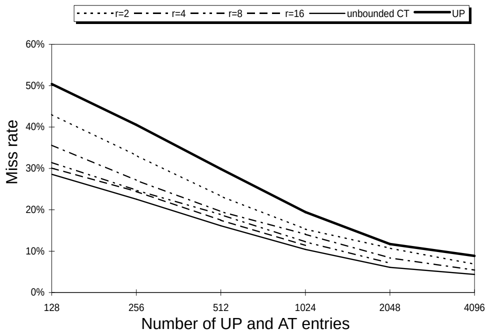


***Fig. 9.* Miss rate in the *Unified Predictor* and the *Split Predictor* in benchmark *go*. UPs, ATs and CTs are direct mapped. Several CT entries/AT entries ratios (r) are shown.**

For benchmark *go*, the *Split Predictor* with a ratio equal to 2 shows a miss reduction about 20% in the whole range of evaluated configurations. Conflicts in CT between unpredictable load instructions do not influence classification, but conflicts between predictable load instructions classify CT entry as unpredictable. Then, increasing the ratio, a finer classification and an additional miss-rate decrease can be achieved. That is, the classification will be improved when at least one of two load instructions that previously collide in a CT entry is predictable. However, it improves the miss rate when only one of the load instructions that collide in an AT entry is predictable. Also, for large ratios, the reduction in misses is gradually fewer. There are a few conflicts in CT but remain conflicts in AT between predictable load instructions.

The area cost of a predictor is defined as the amount of bits that it stores. Following expressions evaluate the area cost of every predictor as a function of the number of table entries.

$$UnifiedPredictorAreaCost = (64+2) \times UPentries$$

$$\begin{array}{c} SplitPredictor \\ AreaCost \end{array} = (2+3) \times CTentries + \left(64 + \log_2\left(\frac{CTentries}{ATentries}\right)\right) \times ATentries$$

We are assuming 64-bit logical addresses,
two-bit saturated counters and a three-bit classifying mechanism.

Area distribution between the *Split Predictor* and the *Unified Predictor* is very different. *Split Predictor* saves area respect the *Unified Predictor* to store addresses. This area saving is used to classify more finely the load instructions. Then, the classification improves the utilization of the AT entries.

In this paper, we evaluate *Split-Predictor* configurations with an area cost smaller than the area cost of the *Unified Predictor* configuration with UP entries=2×AT entries. Between them, we select the configuration with the biggest amount of CT entries to obtain the finest possible classification. From area-cost expressions, we obtain a ratio equal to 8 (CT entries/AT entries=8). These configurations represent an area-cost saving of 19% respect the *Unified Predictor* with UP entries=2×AT entries.

Unbounded CTs obtain the maximum miss-rate reduction respect the *Unified Predictor*. For benchmark go (Figure 9), proposed configurations of the *Split Predictor* achieve about an 80% of the maximum miss reduction. For the other benchmarks, this reduction is similar considering the working-set size of load instructions, Also, because the biggest working-set size of load instructions in analysed benchmarks is about 8K load instructions, we limit the amount of CT entries in our evaluation to 8K entries.

5.5 Predictability captured by the predictors

In this section, the predictability captured by the *Split Predictor* and the *Unified Predictor* are evaluated for several configurations and is shown the relation between configurations.

We will name the *Split Predictor* configurations as {AT entries, CT entries}. From previous observations we select the following configurations: {256, 2.048}, {512, 4.096}, {1.024, 8.192} and {2.048, 8.192}. From Figure 4 we select the <3,3> classifying mechanism; this mechanism achieves a high similarity with the <all,0> classifying mechanism.

Figure 10 shows the normalized predictability captured by every predictor configuration. The predictability is normalized to the predictability captured by a last-address predictor with an unbounded prediction table and without classifying counters (Table 1). Notice that classifying counters reduce captured predictability (Figure 3). Horizontal axe represent the number of AT and UP entries, vertical axe shows normalized captured predictability, graph lines connect the *Unified Predictor* results and the *Split Predictor* results.

For 2.048 AT entries and UP entries, the performance increment achieved by the *Split Predictor* in benchmarks *mksim* and *bc* is smaller than in the remaining benchmarks. In former benchmarks, AT size is bigger than their working sets, so the *Split Predictor* only reduces conflict misses.

The others benchmarks have a working-set size bigger than 2.048 load instructions, so the *Split Predictor* reduce capacity misses. For instance, in *go* benchmark, the $\{256, 2.048\}$ *Split Predictor* captures 65% more predictability than the *Unified Predictor* with 256 entries; on the other hand, storing both predictors 2.048 effective addresses the captured-predictability improvement is 10%.

From Figure 9, the absolute difference between the *Split Predictor* with unbounded CT and the *Unified Predictor*, is bigger for lower UP sizes. This potential increase in performance is observed in Figure 10 that shows a very significant increase of predictability. Also, the miss rate for large *Unified Predictor*'s is lower and the reduction in miss rate obtained by the *Split Predictor* achieves a few increment in predictability.

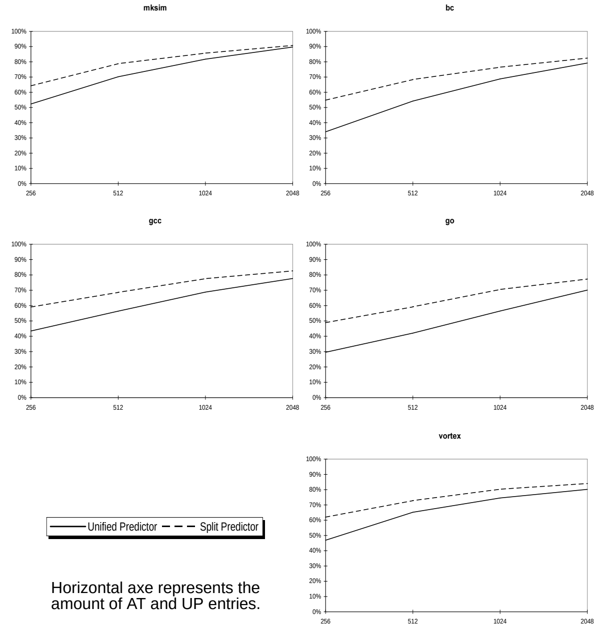


Fig. 10. **Normalized captured predictability by the *Unified Predictor* and the *Split Predictor* in the selected benchmarks.**

Differences in predictability smaller than the 3% in almost all the benchmarks are observed between *Split Predictor* configurations and an *Unified Predictor* with UP entries=2×AT entries. Moreover, the *Split-Predictor* configuration needs only an 81% of the area-cost needed by the *Unified-Predictor* configuration.

5.6 Accuracy of the predictors

Another benefit of the *Split Predictor* is related with the amount of mispredictions. Address predictors are designed to capture as much address predictability as possible, and they also should mistake the minimum number of predictions because every misprediction could have a penalty of some processor cycles.

To compare the correctness of the predictors, we define the accuracy achieved by a predictor as the percent of correct predictions out of the total number of predictions. Figure 11 shows the accuracy achieved by the *Split Predictor* and by the *Unified Predictor* in selected benchmarks.

In almost every benchmark, any *Split-Predictor* configuration achieve a higher accuracy than the most accurate *Unified-Predictor* configuration. It is due to several factors: a) the classification performed by the *Split Predictor* is more precise than the one performed by the *Unified Predictor* due to the bigger number of classifying counters, and b) as AT is partially tagged, the *Split Predictor* can detect some conflicts in AT between CT entries, preventing probable mispredictions.

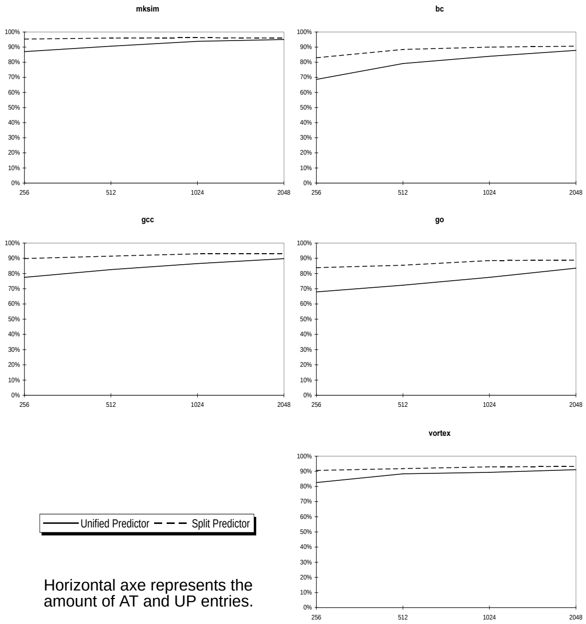


Fig. 11. **Accuracy achieved by the *Unified Predictor* and by the *Split Predictor*.**

Moreover, the accuracy of the *Unified Predictor* is sensitive to the amount of UP entries, while the accuracy of the *Split Predictor* is almost independent of AT entries. Comparing the $\{256, 2.048\}$ *Split Predicto*r to the 256-UP entries *Unified Predictor*, the accuracy is increased from 8% (*vortex*) to 16% (*go*); for bigger ATs and UPs, the difference between their accuracy decreases.

6 Related works

Some works propose address or value predictors to execute speculatively true data-dependent instructions [7][9]. They propose predictors similar to the *Unified Predictor*, and analyse the potential IPC improvement that can be achieved using the predictors. In this work we have proposed a predictor with a smaller area-cost that captures the same predictability.

The use of program profiling to collect information that describes the predictability of the instructions in a program has been proposed in [6]. Profile information is used to classify statically the instructions according to their value predictability. Then, the compiler, insert hints that are used by a hardware predictor to determine if a table entry should be allocated to this instruction.

Static classification has some drawbacks: a) it needs a profile execution, b) static classification is not binary compatible, c) the decision to predict an operation lasts all program execution because the classification is static, and d) the accuracy of this classification is highly sensitive to the contribution of medium-predictable load instruction to the overall predictability. Our work avoids these disadvantages.

7 Conclusions

The streams of effective addresses generated by load instructions in integer benchmarks exhibit a significant tendency to be predictable. Using the last-address predictor, an average 50% of the computed effective addresses can be predicted correctly in integer benchmarks, but the tendency of load instructions to be predictable do not spread uniformly among them.

Most address-predictor mechanisms use tables for recording information of past executions of load instructions to predict future effective addresses. These predictors use an always-allocate policy, it follows that a highly unpredictable load instruction can replace a highly predictable one in the prediction table, decreasing the performance of the predictor.

We propose to split the recorded information into two tables: the Classification Table and the Address Table. The Classification Table classifies at run time every load instruction according to its predictability using few bits of the computed addresses. The Address Table stores information needed to predict predictable load instructions. The Classification Table will filter the unpredictable load instructions to be placed in the Address Table.

Performance studies show that the *Split Predictor* applied to a last-address predictor captures the same predictability than the last-address predictor, but with an area-cost reduction of 19%. Moreover, the *Split Predictor* improves the accuracy of the last-address predictor.

The proposed idea of run-time classification can also be applied to stride-address predictors and to value predictors. In these cases, to reduce the area requirements of the classifying mechanism, we can perform a discrete classification of the instructions, that is, only some executions of the instructions will update their classifications.

Acknowledgements

This work was supported by the Ministry of Education and Science of Spain under grant CICYT TIC-0429/95, and by the CEPBA (European Center for Parallelism of Barcelona).

References

[1] T.M. Austin, S.E. Breach and G.S. Sohi. (1994). Efficient Detection of All Pointer and Array Access Errors. In, *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*.

[2] T.M. Austin, D.N. Pnevmatikos and G.S. Sohi. (1995). Streamlining Data Cache Access with Fast Address Calculation. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 369-380

[3] J.L. Baer and F.T. Chen. (1991). An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of Supercomputing'91 Conference*, pp. 176-186

[4] D. P. Bhandarkar. (1996). *Alpha, implementations and architecture*. Digital Press.

[5] F. Gabbay. (1996). *Speculative Execution based on Value Prediction*. Electrical Engineering Department (Israel Institute of Technology). EE-TR-1080Split

[6] F. Gabbay and A. Mendelson. (1997). Can Program Profiling Support Value Prediction? In *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 270-280

[7] J. González and A. González. (1997). Speculative Execution via Address Prediction and Data Prefetching. In *Proceedings of the 11th International Conference on Supercomputing*, pp. 196-203

[8] J. L. Hennessy and D. Patterson. (1995). *Computer Architect a Quantitative Approach*. Second Edition. Morgan Kaufnab Publishers, Inc.

[9] M. H. Lipasti, C. B. Wilkerson and J. P. Shen. (1996). Value Locality and Load Value Prediction. In *Proceedings of the 7th ACM International Conference on Architecture Support for Programming Languages and Operating Systems*, pp. 138-147

[10] M. H. Lipasti and J. P. Shen. (1996). Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture*, pp. 226-237

[11] S. McFarling. (1993). *Combining Branch Predictors*. Western Research Laboratory (Digital). WRL-TN-36