Control-Flow Speculation through Value Prediction for Superscalar Processors

José González and Antonio González
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya,
{joseg,antonio}@ac.upc.es

Abstract

In this paper, we introduce a new branch predictor that predicts the outcomes of branches by predicting the value of their inputs and performing an early computation of their results according to the predicted values. The design of a hybrid predictor comprising our branch predictor and a correlating branch predictor is presented. We also propose a new selector that chooses the most reliable prediction for each branch. This selector is based on the path followed to reach the branch. Results for immediate updates show a significant improvement with respect to a conventional hybrid predictor for different size configurations. In addition, the proposed hybrid predictor with a size of 8 KB achieves the same miss ratio as a conventional one of 64 KB. Performance evaluation for a dynamically-scheduled superscalar processor, with realistic updates, shows a speed-up of 11% despite its higher latency (up to 4 cycles).

1. Introduction

Branch prediction is one of the key issues on the design of superscalar processors. Building highly accurate branch predictors along with powerful fetch engines may become a challenge for future microprocessors with large instruction windows. Large windows may be useless if processors are not fed continuously with sufficient instructions from the correct path.

Data dependences also limit the Instruction Level Parallelism (ILP) due to the serialization that they impose on the execution of programs. Data value speculation can eliminate this serialization by predicting the inputs/outputs of some instructions and by speculatively executing some sections of the code. Data value speculation is implemented by means of a value predictor and a speculative execution engine.

In this work, we propose to take advantage of the potential of value predictability to enhance branch prediction. We show that branch outcomes can be predicted by predicting their inputs through a conventional value predictor and executing speculatively those branch instructions in an early stage of the pipeline. The mechanism we propose consists of a hybrid predictor which combines a correlating branch predictor (e.g. *gshare)* with a value predictor engine plus the additional logic to detect the instructions that generate the inputs of branches. The selector we propose is based on the path followed to reach the predicted branch.

The results for immediate updates of the tables show that the proposed scheme outperforms the accuracy obtained by a hybrid predictor consisting of a *bimodal* and a *gshare* for all the size configurations considered in this work. More precisely, the proposed scheme achieves the same miss ratio as a conventional hybrid predictor of 64KB, using only 4KB if the value predictor is not considered (because it is already implemented for other purposes), or 8KB if it is taken into account. The conclusions are similar for other correlating predictors, such as the *agree* and the *bimode* predictors.

We have also evaluated both hybrid schemes in a dynamically-scheduled superscalar processor with realistic updates. Results show that the proposed scheme also improves the misprediction rate for realistic updates, and the IPC is increased by 11% on average, even if the latency of the new predictor is assumed to be 2 cycles more than that of the conventional predictor.

The remainder of this paper is organized as follows. Section 2 presents the motivation of this work and reviews some related work. The proposed branch predictor is explained in Section 3. Section 4 analyzes its performance and finally, Section 5 summarizes the main conclusions of this work.

2. Motivation and Related Work

Branch predictors presented so far rely on the fact that the outcome of a branch may be correlated with its own behavior [25], the path followed by the program [19] or the behavior of previous branches [15][28]. Some other proposed predictors include different improvements to the previous ones, in order to avoid aliasing [16][26], to make a better use of the history [11] or to combine different predictors[2][4][15].

However, the outcome of a branch ultimately depends on its source operands. If a branch could know the value of its inputs when it is fetched, the outcome could be computed with ease. Nonetheless, the branch input operands are not usually available at the decode stage. Predicting such values may be an alternative approach to improve the accuracy of branch predictors.

Data speculation has been used so far to deal with data dependences by means of either predicting data dependences [3] [7][17] [18] [27] or the values that flow through such dependences [6] [8] [9] [13] [21] [22] [23].

The aim of this work is to present a new application of value prediction. We propose to use the predictability of values to predict the inputs of branches and execute them according to their predicted inputs. That is, data value prediction will be used for control-flow speculation. There are some scenarios where this technique has a high potential, for instance in codes that traverse a list, searching for an element. Assume that we have the following piece of code:

```
void func (searched_elem) {
  l=first_element
  while (l->elem!=searched_elem) {
    l=l->next;
  }
...
```

If the value of `searched_elem` for successive activations of `func` do not follow any regular pattern, the branch corresponding to the last iteration of the `while` statement will likely be mispredicted most of the times by a correlating branch predictor. However, as far as a branch predictor based on value prediction is concerned, if the values of the `elem` field of successive list elements follow a predictable pattern, such branch will be correctly predicted since the inputs are predictable, regardless of the random behavior of the input parameter of the function.

Using value prediction to predict branches requires a mechanism that identifies which instructions generate the inputs of branches and predicts their values, as well as a few functional units to execute branches speculatively according to their predicted inputs.

Since the accuracy of the value predictors proposed so far is lower than that of conventional branch predictors, branch prediction cannot only rely on value prediction. However, since value predictors and correlating branch predictors exploit two different phenomena, a synergetic effect may be expected when they are combined together. Therefore, we propose a hybrid predictor, that consists of a correlating branch predictor (e.g. *gshare*) and a data value predictor.

Values are used to predict branches in [14]. In that work, the compiler predicts the outcome of a branch taking into account the values of some registers available 16 cycles ahead of the branch. Using profile information, the compiler detects which registers and which values of such registers will likely determine the outcome of a branch. Then, it inserts some instructions in order to speculatively compute the branch direction according to that information. Values involved in the branch prediction must be known some cycles before the branch, so value prediction is not applied in that work. The potential of improving branch prediction by adding data value speculation was suggested in [23], but no particular scheme was proposed. In [5], the execution of the path that leads to some branches is decoupled in order to obtain the outcome before the actual execution. Value prediction is used to speed up the execution of this path. However, only very few static branches are considered (0 to 12 depending on the application), and they are tagged by hand. A similar technique is proposed in [20], where the virtual calls are predicted by dynamically identifying the sequence of operations that compute their targets and pre-computing them.

3. Branch Prediction Mechanism

In this section we present the proposed branch predictor. First, we review the simulated instruction set architecture. Then, we explain the mechanism that predicts the outcomes of branches by executing them with their predicted inputs. We also describe the selector used to choose between the two predictions generated for each branch. Finally we discuss some implementation issues for a dynamically-scheduled superscalar processor.

3.1. Branch Prediction through Value Prediction (BPVP)

The particular implementation of the Branch Prediction through Value Prediction unit (BPVP) depends on the instruction set architecture (ISA). The implementation presented in this work is based on the ISA of the Alpha architecture [24]. Conditional branches of this ISA have only one source register operand, whereas comparisons can have two inputs. Minor modifications would be needed to extend this mechanism to different ISA's. In particular, for those ISA's where branches have two inputs, they would be managed as the comparisons in the current implementation.

The mechanism proposed in this work has a different behavior depending on the branch that is to be predicted:

- For a branch whose input is produced by an arithmetic or load instruction, the mechanism stores the PC (Program Counter) of the producer of the input so that when the branch is fetched, it obtains this PC, access the value predictor, predicts the input and computes the outcome according to the predicted input.
- For a branch whose input is produced by a compare instruction, predicting the input of the branch would just be like predicting the outcome of the branch (both the input and the output are booleans). Instead, the mechanism predicts the inputs of the comparison when it is fetched, executes the comparison according to the predicted inputs and stores the speculative result in a table. When the dependent branch is fetched it gets this result from the table.

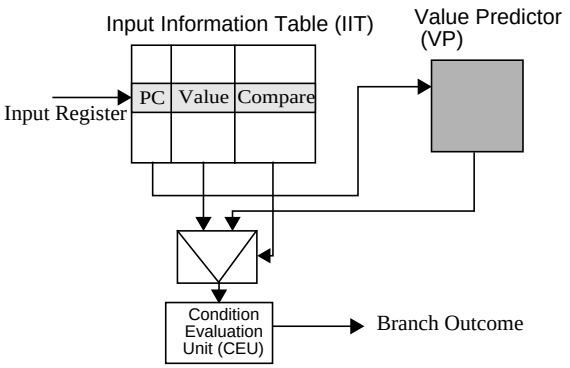

Figure 1. Block diagram of the branch predictor based on value prediction (BPVP)

Figure 1 shows the implementation of the proposed mechanism. The *BPVP* mechanism consists of an *Input Information Table (IIT)*, a *value predictor* and a special purpose functional unit which can perform comparisons. The *IIT* stores all the information that a branch needs to be predicted. It is indexed by a logical register identifier, thus it has *#logical regs* entries. Each entry has three fields:

- PC: stores the PC of the latest instruction that had that logical register as destination. This information is used by branches and comparisons when they have to predict their inputs, in order to access the value predictor.
- Value: stores the boolean value computed speculatively by the latest compare instruction whose destination register was the one corresponding to this entry.
- Compare flag: this field is set when the latest instruction that has written to the register corresponding to this entry is a compare instruction.

These hardware structures operate as follows, depending on the type of instruction being decoded:

- Load or arithmetic instructions: the entry of the *IIT* indexed by the destination register is updated with the current PC and the compare flag is set to 0.
- Compare instruction: the *BPVP* predicts its inputs and produces its output by performing the comparison according to the predicted inputs. In order to do that, the *IIT* is accessed twice. First, the PCs of the instructions that produce the inputs are obtained by reading the *IIT* entries corresponding to the source registers. Then, the *value predictor (VP)* is used to predict the values associated with these PC's. The predicted inputs are forwarded to the *condition evaluation unit (CEU)*, where the result of the comparison is computed. The *IIT* is then indexed by the destination register identifier in order to store the result of the speculative computation in the *value* field. In addition, the compare flag is set to 1.
- Branch instruction: the *IIT* is accessed using the source register identifier. If the compare flag is set (i.e., the input value has already been predicted by a compare instruction) the outcome is computed in the *CEU* according to the speculative result of the comparison (stored in the *value* field). If the compare flag is not set, the PC of the producer of the input is obtained from the *IIT*, and its value is predicted by the *VP*. Then, the speculative branch outcome is computed in *CEU*.

Note that the *IIT* stores the PC of the producers of the inputs of branches and comparisons. One may think that these inputs could be predicted just with the PC of branches and comparisons. However, these inputs may have different producers in successive executions depending on the path followed to reach them. Thus, using the PC of their producers is more effective, since different paths use different PC's and this provides a more precise information to the value predictor.

The *BPVP* can be easily extended to allow multiple predictions per cycle, since a branch does not need the prediction of a previous branch that could be fetched in the same cycle.

In this work, two different value predictors have been considered: the *context-value predictor* [21] [22] and the *stride predictor* [6] [8]. Value predictions are performed when a branch or compare instruction is fetched. The predictor is updated once the inputs of these instructions are available. Thus, the PC's of the producers must be kept in some structure (for instance, in the reservation station entry of the branches and compare instructions), in order to pass all required information (PC and correct value) to the *value predictor*.

The major drawback of this approach to predicting branches is its latency, since several accesses to tables have to be performed in order to predict a branch. On the other hand, predicting only the results of those instructions that produce the inputs of branches and comparisons (instead of the result of all the instructions), significantly reduces the cost of the Value Predictor. There is another approach, detailed in [10], that uses a *Speculative Register File* to store the predicted values, so that, comparisons and branches obtain their speculative input from it. That implies a reduction in the number of indirections, since the *IIT* is not needed anymore, and thus, the latency of the *BPVP* is decreased.

3.2. Selector

The selector is a key part of a hybrid predictor. In this case, it has to choose the most reliable option between the predictions generated by a correlating predictor (*CorrPred*) and the *BPVP*.

Figure 2 shows the structure of the proposed 2-level selection mechanism. The first level is a single history

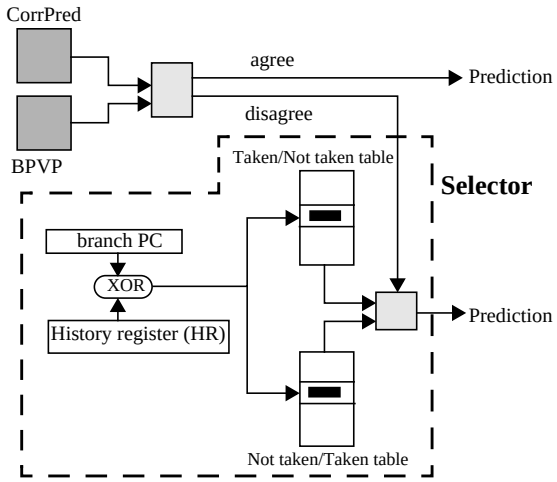

Figure 2. Block diagram of the proposed hybrid branch predictor

register (HR) and the second level consists of two tables of counters. Each table is associated to one of the possible combinations when both branch predictors disagree. For instance, the Taken/Not Taken table is used when the *CorrPred* predicts taken and the *BPVP* predicts not taken. Preliminary analysis showed that having two different tables provided more accuracy than having one table of the same total size.

Each entry of these tables has a two-bit, up-down saturating counter. High values in a counter give more confidence to the *CorrPred,* whereas low values give confidence to the *BPVP*.

The selector is used only when the predictors disagree. The selector can be accessed at the same time as the branch predictors are working. The history register is *XOR-ed* with the least-significant bits of the branch PC and both tables are accessed. Depending on the two predictions, the counter from the corresponding table is selected. If the counter has a value greater than 1, the *CorrPred* is chosen, otherwise the *BPVP* is selected.

A novel feature of the selector is the history register. It holds a part of the PC from each control transfer instruction. That is, for each fetched control instruction, the history register is shifted *m* bits to the left and the *m* least-significant bits of the PC are shifted in (in this work we assume *m*=2). The reason for using this history register instead of a traditional one is because one of the predictors is based on value prediction, and the path that leads to the branch instruction may supply very useful information. For instance, assume that we have the following piece of code (very similar to one of the most frequent branches in the *go* program):

```
void func (int a) {
    load b
    cmp a,b
    bne target
    ...
```

It can be observed that the branch is dependent on one of the parameters of the function. This function is called in different parts of the program, and therefore the value of *a* is generated by different instructions. For some paths, *a* may be highly predictable whereas for others it may not. By recording some bits of the PC of the `call func` instructions, the more predictable paths can be identified. The proposed mechanism uses different entries of the selector for the same branch depending on the path followed to reach it. Thus, the entries corresponding to a branch with very predictable inputs will assign more confidence to the *BPVP*. In [19], Navir proposed using the path that leads to a conditional branch as a part of the history register of a *correlating predictor*.

Once the correct outcome of a branch is computed, the corresponding counters are updated by increasing the counter if the *CorrPred* hits, or decreasing the counter otherwise. Note that, since the selector is only used when the predictors disagree, only one of the predictors hits.

3.3. Implementation Issues in a Superscalar Processor

Since each prediction may involve up to two table lookups (*IIT+selector* in parallel followed by the *VP*) and a CEU operation, it is unlikely that it will be implemented in a single cycle. We assume that it is pipelined into several cycles, up to a maximum of four.

For evaluation purposes, we consider that the *CorrPred* (for these experiments, we will use the *gshare* [15], the *agree* [26] and the *bimode* [12]) performs its prediction in one cycle. The processor will follow the path indicated by the *CorrPred*. In the meantime, the *BPVP* and the selector tables are accessed in parallel, so that, some cycles later (up to three), the processor will have the prediction made by the *BPVP* and the result of the selector. If the *BPVP* agrees with the *CorrPred* or the selector assigns more confidence to the *CorrPred*, no actions are taken. However, if both predictors disagree and the selector has chosen the *BPVP*, the prediction carried out by the *CorrPred* is reversed, the instructions already fetched are discarded and the fetch starts again on the path predicted by the *BPVP*. From now on, we will refer to the number of cycles that the *BPVP* takes as the *BPVP latency*.

There are some other issues that must be taken into account:

- The proposed hybrid predictor will be compared to a hybrid predictor composed of a *bimodal* and a *gshare*. For this latter predictor, we assume that predictions can be made in one cycle.
- The history register of the *CorrPred* is updated speculatively with the result of the predictor chosen by the selector.
- For the proposed hybrid scheme, the value predictor is updated with the correct value once the compare instruction or the branch have their inputs available.
- The counters of the selectors employed in both hybrid schemes are updated at commit time.

4. Results

In this section we analyze the performance of the hybrid predictor presented in this work. First of all, the misprediction rate obtained by the hybrid predictor *BPVP+gshare* is compared with that obtained by a hybrid predictor consisting of a *bimodal* [15][25] and a *gshare*. We have chosen this hybrid predictor since it includes a branch predictor that works well for branches that correlate with themselves (*bimodal*) and a predictor that works well for branches that correlate with others (*gshare*). The *bimodal* predictor is implemented by means of a table of two-bit, up-down saturating counters indexed with the least significant bits of the branch PC.

We also evaluate the combination of the *BPVP* with other branch predictors. In particular we consider the *agree* predictor and the *bimode* predictor. From now one we will refer to the combination of the *bimodal* plus the *gshare, bimode or agree* as *2bit+CorrPred,* and he combination of *BPVP* plus *gshare, bimode* or *agree* as *BPVP+CorrPred*, where *CorrPred* is a particular correlating predictor

Regarding the selector of the *2bit+CorrPred* hybrid predictor, we first considered the one proposed in this work. However, preliminary studies demonstrated that for the *2bit+CorrPred* predictor the two-level selector proposed in [2] performs better than our selector and therefore, for the *2bit+CorrPred* we use that selector. The first level of it consists of the same history register as that used by the *gshare* predictor. The second level is a table with 2 counters per entry. Each counter is associated with a predictor and they are used to assign confidence to the two predictions made for each particular branch.

4.1. Methodology

We performed two set of experiments. First, in order to evaluate the potential of our scheme, we obtained a trace of instructions using the *sim-safe* utility of the *SimpleScalar/ Alpha* tool set [1]. In these experiments, the different predictors are updated with the actual result immediately after the prediction. In a second set of experiments, we used the *sim-outorder* simulator of the SimpleScalar in order to study the behavior of the different schemes when they are implemented in a superscalar processor. In this case, the tables are updated when the actual value is available.

The five programs from SpecInt95 that exhibit the highest branch misprediction rates have been considered. Table 1 lists the programs along with their input sets and the number of instructions executed. All programs were run to completion. We compare predictors that have the same total capacity in their tables. In order to compute the capacity of each predictor, the selector tables and the two-bit counter tables have been considered but the global history registers have been neglected (their contribution is in fact negligible).

Table 1. Benchmark programs, along with their input sets, number of instructions executed, and number of conditional branches executed.

Program	Input set	#dyn.inst. (in Millions)	#dyn. cond. branches (in Millions)
perl	scrabbl.in	46.7	5.2
compress	40000 e 2231	169.6	12.6
li	7 queens	242.7	32.0
gcc	genrecog.i	145.4	19.3
go	9 9	145.6	15.3

4.2. Results Using Immediate Updates

This Section presents misprediction rates when the tables are immediately updated with the correct value, without taking into account the timing of a superscalar processor. The objective is to evaluate the potential of the predictor when it is isolated from other aspects of the processor microarchitecture.

4.2.1. Results for an Oracle Selector. The potential of the *BPVP* when combined with *gshare, agree* or *bimode* is shown in this subsection. Before considering any particular selector, one may be interested in evaluating the potential improvement that the *BPVP* can provide to branch prediction. We therefore considered an *oracle* selector, which always chooses the right predictions when the two predictors disagree. The results are shown in Figure 3. For each particular size, the left column represents the combination *BPVP+gshare*, the middle column corresponds to the combination *BPVP+agree* and the right column represents *BPVP+bimode.* For each configuration, each predictor occupies half of the total capacity.

It can be observed (white bars) that the *BPVP* can correctly predict the majority of branches mispredicted by *gshare*. This benefit is quite important for programs where *gshare* does not perform very well, such as *go*, for which the *BPVP* reduces the miss rate by about 62% for the 1KB configuration. Regarding the *agree* and *bimode* predictors,

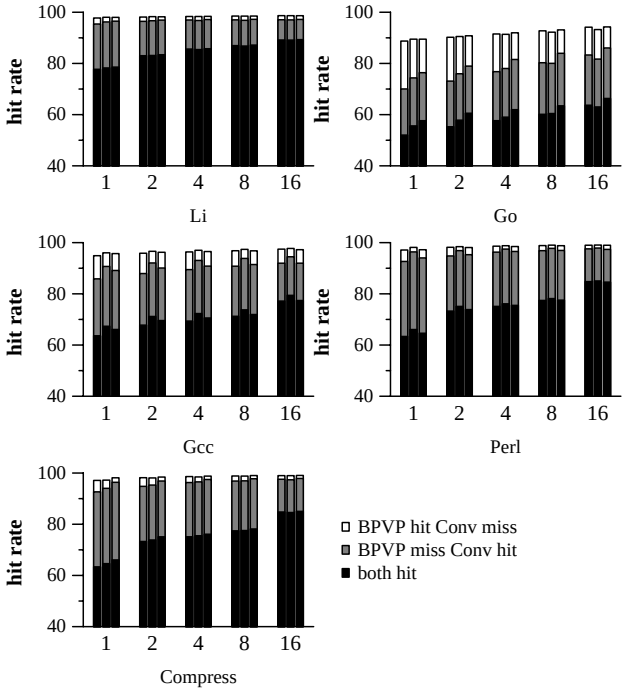

Figure 3. Hit rate for an oracle selector.

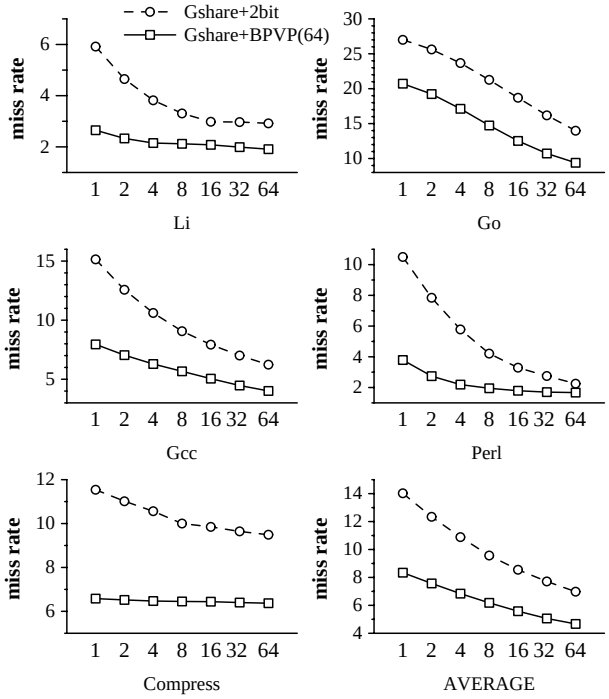

Figure 4. Misprediction rates for both hybrid predictors. The size of the value predictor is not taken into account.

one can observe that in general they slightly outperform the *gshare* (compare grey columns). The *BPVP* can also enhance the accuracy of these particular predictors, achieving an overall hit rate greater than that achieved by the *BPVP+gshare*. This means that for some of the most powerful predictors based on branch correlation, predicting branches based on value prediction can still provide significant improvements.

4.2.2. Results Using a Realistic Selector. In this subsection, the hit rates of the different predictors with a realistic selector are compared. We first analyze the performance of the hybrid branch predictors assuming that the processor already has a *value prediction unit*. Note that value prediction is a mechanism that can be used to speed-up different parts of the execution, in addition to branches. Therefore, one may assume that future microprocessors will probably incorporate a value predictor for other purposes different from branch prediction. This same value predictor can be used by the *BPVP* scheme. Under this scenario one may argue that the contribution of the value predictor to the total cost of the branch predictor can be neglected since it is not a new feature required by the mechanism.

Figure 4 shows the misprediction rates for this scenario using *gshare* as a correlating predictor and a context-value predictor for the *BPVP*. This figure depicts the results for the five evaluated programs (*li, go, compress, gcc* and *perl*) and the arithmetic mean. The ordinate axis represents the total size (in KiloBytes) of the predictor. Note that for the *2bit+gshare* predictor, a quarter of the total capacity is devoted to each predictor, and a half corresponds to the selector (for instance, for a 32KB configuration, the *gshare* table has 8 KB, the *2bit* predictor has 8 KB and the selector tables have 16 KB). On the other hand, for the *BPVP+gshare* predictor, the total size of the predictor is split into two equal parts, corresponding to the *gshare* and the selector. The size of the context-value predictor is 64 KB. Results show that *BPVP+gshare* significantly outperforms *2bit+gshare* providing a very important reduction in the miss rate for all considered sizes. For instance, the miss ratio is almost halved in the 64 KB configuration. On average, a *BPVP+gshare* of 4 KB obtains about the same performance as a *2bit+gshare* of 64 KB.

Figure 5 shows the miss rates when the hardware of the *value predictor* is included in the total size of the *BPVP+gshare* (for instance, for a 32 KB configuration, the value predictor has 8 KB, the *gshare* table has 8 KB and the selector tables have 16 KB). This would be the analysis when we include a *value prediction unit* only dedicated to predict branches. Two different value predictors are considered: a stride predictor and a context-value predictor. For the three hybrid predictors the size of the selector represents a half of the total size. It can be observed that the combination of a *BPVP+gshare* significantly outperforms *2bit+gshare* for all size configurations, in spite of the fact that the hardware of the value predictor is included. On average, a *BPVP+gshare* of 8KB has about the same performance as a *2bit+gshare* of 64KB. Moreover, the miss ratio obtained by the *BPVP+gshare* when a stride predictor

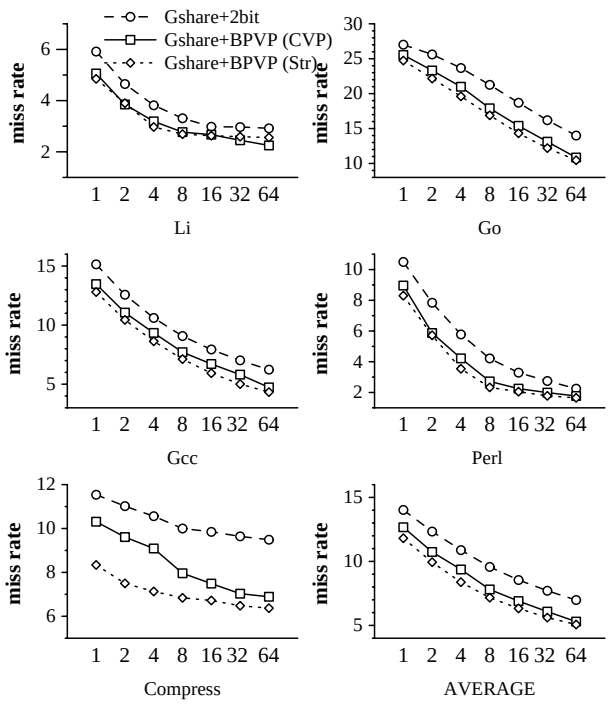

Figure 5. Misprediction rates for *2bit+gshare* and *BPVP+gshare* predictors. The *BPVP* predictor includes either a context-value predictor (CPV) or a stride predictor (Str)

is used is lower than that obtained for a context-value predictor. The reason is that for a moderate hardware cost, stride predictors perform better than context-value predictors [22].

The conclusions are similar if we consider more recent correlating predictors such as the *agree* and the *bimode* instead of the *gshare*. Table 2 summarizes the average misprediction rates obtained by each hybrid predictor (including those which have the *agree* and *bimode*) as a function of the size, when the size of the value predictor is included in the branch predictor. It can be observed that no matter the accuracy of the baseline predictor, the *BPVP* can significantly improve its performance. In general, a *BPVP+CorrPred* of 8KB has about the same miss ratio as a *2bit+CorrPred* of 64KB (see shaded cells). Further details about the behavior of the *2bit+ agree, 2bit+bimode, BPVP+agree* and *BPVP+bimode* for each particular program can be found in [10].

We have also run experiments comparing hybrid predictors with the combination *gshare+bimode* or *gshare+agree* against *BPVP+gshare* or *BPVP+bimode* and we have observed similar improvements. The reason is that the *agree* and the *bimode* predictors have the aim of predicting the same type of branches as the *gshare*, but reducing the interferences. The scope of these predictors are those branches that are correlated, whereas the aim of the *BPVP* is to predict those branches that depend on highly

Table 2. Average misprediction rate for each hybrid predictor depending on the size

	1 KB	2 KB	4 KB	8 KB	16 KB	32 KB	64 KB
2bit+gshare	14.02	12.34	10.89	9.57	8.55	7.71	6.98
2bit+agree	12.08	10.95	9.90	8.97	8.21	7.60	7.06
2bit+bimode	12.01	10.67	9.52	8.52	7.68	7.01	6.43
BPVP+gshare	11.81	9.94	8.38	7.17	6.33	5.61	5.07
BPVP+agree	10.35	9.04	7.88	6.98	6.30	5.77	5.35
BPVP+bimode	10.14	8.75	7.63	6.74	6.00	5.46	5.01

predictable inputs, which is a different philosophy. Therefore, the *BPVP* complements a *correlating predictor* much better than other *correlating predictors.*

4.3. Results for Realistic Updates in a Superscalar Processor

This section presents an evaluation of the *BPVP+gshare* by comparing it with the *2bit+gshare* in a dynamically-scheduled superscalar processor.

Table 3 summarizes the main parameters of the simulated processor. We have modified SimpleScalar in order to have a more powerful fetch engine, so that taken conditional branches that are correctly predicted can fetch the correct path in the next cycle. Regarding the predictors' latencies, the *2bit+gshare* can predict the branch in the

Table 3. Simulation parameters

Fetch engine	Fetches up to 8 instructions per cycle, allowing 2 taken branches. 8 cycles branch misprediction penalty
Instruction cache	64 KB two-way associative
Execution engine	issues up to 8 instruction per cycle, 128-entry reorder buffer, 64-entry load/store queue, loads are executed when the addresses of all previous stores are known
Functional Units	6 integer alu, 1 integer mult. 2 memports, 6 FP alu, 1 FP mult
Latency	1 cycle IntAlu, 2 cycles FP alu, 1 cycle load/store, 3 cycles IntMult, 4 cycles FP mult
Data Cache	64 KB two-way associative, 32byte/line, 256 KB 4-way unified L2, 6 cycles miss latency. 18 cycles L2 miss
Virtual Memory	4 KB pages, 30 cycles TLB miss

fetch stage (one-cycle latency) whereas we have considered a *BPVP* latency ranging from 1 to 4 cycles, i.e., the *gshare* predicts in the fetch stage and up to 3 cycles later the selector and the *BPVP* compute their result, so that the prediction of the *gshare* is maintained or reversed.

Figure 6 shows the IPC obtained for each program as well as the harmonic mean, using 8 KB branch predictors. The *BPVP* uses a stride predictor and its hardware is included in the 8 KB. Table 4 shows the speedup obtained by the *BPVP+gshare* for different *BPVP* latencies with

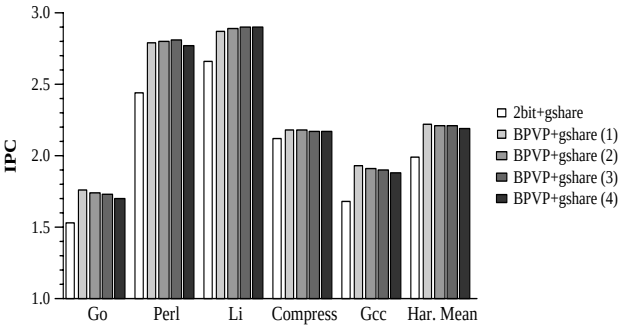

Figure 6. IPC for 8 KB predictors. The *2bit+gshare* latency is 1 cycle whereas the *BPVP+gshare* latency ranges from 1 to 4 cycles.

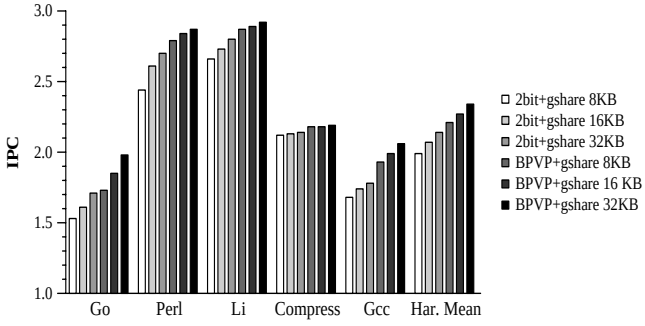

Figure 7. IPC obtained for a *2bit+gshare* and *BPVP+gshare* as a function of their size. The latency of the *BPVP+gshare* is 3 cycles

respect to the *2bit+gshare* with a 1-cycle latency. The misprediction rates for immediate updates (those obtained in the Section 4.2.2) and those obtained when the predictors are implemented in a superscalar processor are also shown.

Table 4. Speedup of the *BPVP+gshare* with respect to *2bit+gshare*, for latencies varying from 1 to 4 cycles for the *BPVP+gshare*, and a fixed latency of 1 cycle for the *2bit+gshare,* along with the misprediction rates for both immediate updates (Figure 5) and realistic updates. The misprediction rate of the stride predictor is also presented.

	Go	Perl	Li	Compress	Gcc	AVG.
Speedup (1)	15%	14%	8%	3%	15%	11%
Speedup (2)	14%	15%	9%	3%	14%	11%
Speedup (3)	13%	15%	9%	3%	13%	11%
Speedup (4)	11%	14%	9%	2%	12%	10%
2bit+gshare,MissRate RealUp	24.41	8.16	6.38	10.75	14.53	12.84
2bit+gshare,MissRateImmUp	21.26	4.21	3.31	10.00	9.07	9.57
BPVP+gshare, MissRate RealUp	18.54	5.06	4.31	9.46	10.45	9.56
BPVP+gshare, MissRate Immediate Up	16.89	2.33	2.69	6.84	7.12	7.17
Value predictor, MissRate Real update	64.48	61.15	52.61	49.84	67.88	59.19
Value predictor MissRate Immediate update	56.13	54.14	48.07	42.04	56.13	51.30

Results in Figure 6 and Table 4 show that the proposed hybrid scheme achieves a significant improvement with respect to the *2bit+gshare* in spite of its higher latency (about 11% increase in IPC on average). In fact, the degradation caused by increasing the predictor latency from 1 to 3 cycles is almost negligible, which allows for a pipelined implementation of the *BPVP*. We can also observe that the increase in misprediction rate when the tables are realistically updated is more or less the same for both hybrid predictors.The average misprediction rate goes from 9.57% to 12.84% for the *2bit+gshare*, and from 7.17% to 9.56% for the *BPVP+gshare*. The lowest speedup is experienced by *compress*, which is due to the negative effect of realistic updates on the value predictor whereas the *2bit+gshare* is hardly affected.

It is also interesting to analyze how the predictor size affects the IPC. Figure 7 shows the IPC achieved by both hybrid predictors when their sizes go from 8 KB to 32 KB. For the *BPVP+gshare* the *BPVP* latency considered is 3 cycles. It can be observed that for all programs the IPC achieved using a *BPVP+gshare* of 8 KB is higher than that obtained using a *2bit+gshare* of 32 KB.

Finally, it is important to remark that not only is the IPC important, but also the misprediction rate. The IPC is affected by many features of the microarchitecture other than branch prediction accuracy. If future microprocessors remove those other bottlenecks, the contribution of the branch predictor may become more important. Some results presented in [9] point towards this direction.

5. Conclusions

In this paper, a new approach to performing branch prediction has been proposed.

This approach, called *Branch Prediction Through Value Prediction* (*BPVP)*, predicts the outcomes of conditional branches by predicting the values of their inputs and performing an early computation of branches according to those predictions.

In order to predict values, we considered both a context-value predictor and a stride predictor. We proposed a hybrid predictor consisting of a *gshare* and a *BPVP*. The aim of this combination is to capture with the *BPVP* those branches that a *gshare* cannot predict. We have also proposed a new selector that is based on the path followed by the program to reach the predicted branch.

We have compared the *BPVP+gshare* with a *bimodal+gshare*. Results using immediate updates show significant improvements for all considered sizes. For instance, a *BPVP+gshare* of 8 KB achieves about the same misprediction rate as a *bimodal+gshare* of 64 KB, even when the cost of the value predictor (which may be used for other purposes) is included in the branch predictor. We have also observed similar conclusions when a *bimode* or an

agree are used instead of a *gshare*. Average reductions of about 40% in misprediction rate are observed.

Finally, we have evaluated the proposed predictor in a dynamically-scheduled superscalar processor with realistic updates of the tables and realistic *BPVP* latencies. Results show a significant reduction in the misprediction rate and, therefore, an important increase in the IPC. For instance, we have observed that having a *BPVP+gshare* of 8 KB obtains a higher IPC than a *bimodal+gshare* of 32 KB.

6. Acknowledgments

The authors would like to thank Nigel Topham for his valuable comments on a draft version of this paper. We also thank Todd Austin and Doug Burger for theAlpha version of the Simplescalar.

This work has been supported by the grants CYCIT TIC98-0511, and 1996FI-03039-APDT, the ESPRIT project MHAOTEU (EP24942), and the CEPBA.

7. References

[1] D. Burger, T.M. Austin,"The SimpleScalar Tool Set, Version 2.0", Technical Report #1342, University of Wisconsin-Madison, Computer Sciences Department, June 1997.

[2] P.-Y. Chang, E. Hao and Y.N. Patt, "Alternative Implementations of Hybrid Branch Predictors", in *Proc. Int. Symp. on Microarchitecture*, pp. 252-257, 1995.

[3] G.Z. Chrysos and J.S. Emer, "Memory Dependence Prediction Using Store Sets", in *Proc. Int. Symp. on Computer Architecture*, pp. 142-153, 1998

[4] M. Evers, P-.Y. Chang and Y.N. Patt, "Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches", in *Proc. Int. Symp. on Computer Architecture*, pp. 3-11, 1996

[5] A. Farcy, O. Temam, R. Espasa, T. Juan, "Dataflow Analysis of Branch Mispredictions and Its Applications to Early Resolution of Branches", in *Proc. Int Symp. on Microarchitecture*, 1998.

[6] F. Gabbay and A. Mendelson, "Speculative Execution Based on Value Prediction". Technical Report, Technion, 1997.

[7] J. González and A. González, "Memory Address Prediction for Data Speculations", in *Proc. EURO-PAR'97 Conference*, pp. 1084-1091, Aug. 1997.

[8] J. González and A. González, "Speculative Execution via Address Prediction and Data Prefetching", in *Proc. International Conference on Supercomputing*, pp 196-203, 1997

[9] J. González and A. González, "The Potential of Data Value Speculation to Boost ILP", in *Proc. International Conference on Supercomputing*, 1998.

[10] J. González and A. González. "Control-Flow Speculation through Value Prediction for Superscalar Processors". Technical Report #UPC-DAC-1998-46, 1998. ftp://ftp.ac.upc.es/pub/reports/DAC/1998/UPC-DAC-1998-46.ps.Z

[11] T. Juan, S. Sanjeevan and J.J. Navarro, "Dynamic History-Length Fitting: A third level of adaptivity for branch prediction", in *Proc. Int. Symp. on Computer Architecture*, 1998.

[12] C. Lee, I. Cheng, K. Cheng and T. N. Mudge, "The Bi-Mode Branch Predictor", in *Proc. Int. Symp. on Microarchitecture*, pp 4-131997.

[13] M.H. Lipasti, *Value Locality and Speculative Execution*, Ph. D. thesis, Carnegie Mellon University, 1997.

[14] S. Mahlke and B. Natarajan, "Compiler Synthesized Dynamic Branch Prediction", in *Proc. Int. Symp. on Microarchitecture*, 1996.

[15] S. McFarling, "Combining branch predictors", Technical Report #TN-36, Digital Western Research Laboratory, 1993.

[16] P. Michaud, A. Seznec and R. Uhlig, "Trading conflicts and capacity aliasing in conditional branch predictors", in *Proc. of the Int. Symp. on Computer Architecture*, pp 292-303, 1997.

[17] A. Moshovos and G.S. Sohi. "Streamlining Inter-operation Memory Communication via Data Dependence Prediction", in *Proc. Int. Symp. on Microarchitecture*, pp 235-245, 1997.

[18] A. Moshovos, S. Breach, T. Vijaykumar and G. Sohi, "Dynamic Speculation and Synchronization of Data Dependences", in *Proc. of the Int. Symp. on Computer Architecture*, pp. 170-180, 1997

[19] R.Nair, "Dynamic path-based branch correlation", in *Proc Int. Symp. on Microarchitecture*, pp 15-23, 1995.

[20] A. Roth, A. Moshovos and G.S. Sohi. "Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation", in *Proc. Int. Conference on Supercomputing*, pp. 356-364, 1999.

[21] Y. Sazeides and J.E. Smith, "The Predictability of Data Values", in *Proc. Int. Symp. on Microarchitecture*, pp 248-258, 1997.

[22] Y. Sazeides and J.E. Smith, "Implementations of Context Based Value Predictors", Technical Report #ECE-TR-97-8, University of Wisconsin-Madison, 1997.

[23] Y. Sazeides and J.E. Smith, "Modeling program predictability", in *Proc. of the Int. Symp. on Computer Architecture*, 1998.

[24] R.L.Sites, "Alpha Architecture Reference Manual", Digital Press, 1992.

[25] J.E. Smith, "A study of branch prediction strategies", in *Proc. of the Int. Symp. on Computer Architecture*, pp. 135-148, 1981.

[26] E. Sprangle, R.S. Chappel, M. Alsup and Y.N. Patt, "The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference", *Proc. of the Int. Symp. on Computer Architecture*, 1997.

[27] G. Tyson and T. Austin. "Improving the Accuracy and Performance of Memory Communication Through Renaming", in *Proc. Int. Symp. on Microarchitecture*, pp 218-227. 1997.

[28] T.Y. Yeh and Y. N. Patt, "Two-level adaptive branch prediction", in *Proc. Int. Symp. on Microarchitecture*, pp. 51-61, 1991.