AN ALTERNATIVE TO BRANCH PREDICTION: PRE-COMPUTED BRANCHES

Lucian N. VINTAN*, Marius SBERA, Ioan Z. MIHU*, Adrian FLOREA***

"Lucian Blaga" University of Sibiu, Computer Science Department, Sibiu, ROMANIA
*** "S.C. Consultens Informationstechnik S.R.L." Sibiu, ROMANIA*
E-mail: lucian.vintan@mail.ulbsibiu.ro

Abstract: Through this paper we developed an alternative approach to the present – day two level dynamic branch prediction structures. Instead of predicting branches based on history information, we propose to pre - calculate the branch outcome. A pre - calculated branch prediction (PCB) determines the outcome of a branch as soon as all of the branch's operands are known. The instruction that produced the last branch's operand will trigger a supplementary branch condition estimation and, after this operation, it correspondingly computes the branch outcome. This outcome is cached into a prediction table. The new proposed PCB algorithm clearly outperforms all the classical branch prediction schemes, simulations on SPEC and Stanford HSA benchmarks, proving to be very efficient. Also, our investigations related to architectural complexity and timing costs are quite optimistic, involving an original alternative to the present-day in branch prediction approach.

Keywords: Multiple Instruction Issue, Pipelining, Dynamic Branch Prediction, Speculative Execution, Execution Driven Simulation, Performance and Complexity Evaluations

1. Introduction

Excellent branch handling techniques are essential for current and future advanced microprocessors. These modern processors are characterized by the fact that many instructions are in different stages in the pipeline. Instruction issue also works best with a large instruction window, leading to even more instructions that are "in flight" in the pipeline. However, approximately every seventh instruction in an instruction stream is a branch instruction which potentially interrupts the instruction flow through the pipeline [4,5,6,7]. As the pipeline depths and the issue rates increase, the amount of speculative work that must be thrown away in the event of a branch misprediction also increases. Thus, tomorrow's processors will require even more accurate dynamic branch prediction to deliver their potential performance [3,14,15,16,17].

Dynamic branch prediction forecast the outcome of branch instructions at run-time. This forecast, or prediction, may change for each occurrence of the branch even the dynamic context is the same. Dynamic branch predictors are composed of a single level, such as a classical Branch Target Cache (BTC), or even two levels, such as the Two-Level Adaptive Branch Predictors [8,9,10].

A BTC predicts (Taken/Not Taken and the corresponding Target Address) on the overall past behavior of the branch. In contrast, a Two-Level Adaptive predictor bases its prediction on either global history information or local history information. The first level history records the outcomes of the most recently executed branches (correlation information) and the second level history keeps track of the more likely direction of a branch when a particular pattern is encountered in the first level history. Global schemes exploit correlation between the outcome of the current branch and neighboring branches that were executed leading to the branch. Local schemes exploit the outcome of the current branch and its past behavior. Recently there has been interest in hybrid branch predictors where the fundamental idea is to combine different dynamic predictor schemes having different advantages, in a run-time adaptive manner [13].

2. The Pre - Computed Branch Algorithm

We suggest through this paper an alternative approach to the present – day dynamic branch prediction schemes. Instead of predicting based on history information, we propose to pre - calculate the branch outcome. A pre - calculated branch (PCB) determines the outcome of a branch as soon as all of the branch operands are known. This occurs when the last operand of the branch's instruction is produced at execution. The instruction that produced the last operand will trigger supplementary branch condition estimation and, after this operation, it correspondingly will compute the branch outcome (Taken/Not Taken). Similarly to branch history p rediction schemes, branch information is cached into a "prediction" table (PT), as it will be further presented. Through this method, excepted the first one, every instance of a branch can be computed and therefore correctly "predicted", before their issue.

In our PCB study we used MIPS-I microprocessor's Instruction Set Architecture (ISA) since a branch instruction has addressing modes with two register operands and no immediate operands. Considering for example the following MIPS –I code sequence:

ADD R9, R5, R7; //R9<-(R5) + (R7)

BNE R9, R8, offset; //if (R9!=R8) PC<-(PC) + offset

The first instruction (ADD) modifies the R9 content and therefore it directly influences the branch condition. That means that the ADD instruction will correspondingly modify R9 content in the branch prediction structures. After this operation the branch prediction structure estimates the condition and, at the moment when the branch instruction itself is encountered, its behavior will be perfectly known. Figure 1 depicts our new proposed branch prediction scheme. It uses two tables: the PT table and an extension of the register file called RU (Register Unit). As the reader can see further, PC doesn't indexes the RU table. It is used for some associative searches in PT table and also, in some certain cases, it will be updated into the LDPC field. We mention that the letters associated with the arrows in figures 1, 2 and 3 (a, b, c, d and e) represents sequential operations.

Each entry in the PT table consists of:

- a TAG field (the branch's PC high order bits)
- PC1 and PC2 – which are pointers to the last branch's operands producers (the PCs of the instructions that produced the branch's operands values)
- OPC – the branch's opcode
- nOP1 and nOP2 – the register names of the branch operands
- PRED – the branch outcome (Taken or Not-Taken) and a LRU field *(Least Recently Used)*

RU table maintains the register file meanings but additionally, each entry, has two new fields named LDPC and respectively RC.

- The "value" field contains the register data value
- LDPC – represents the most recent instruction label (PC) that wrote in that register.
- The RC field – is a reference counter that is incremented by one by each instruction writing in the attached register and linked by every branch instruction stored in PT table (therefore the instruction's label is necessary to be found in PC1 or PC2 field). The RC field is decremented by one when the corresponding branch instruction is evicted from the PT table. Therefore, if the RC field attached to a certain register is '0' it involves that in the PT table there isn't any branch having that register as a source operand.

In the newly proposed PCB algorithm, the PC of every non-branch instruction, that modifies at least one register, is recorded into the LDPC field belonging to its destination register. The first issue of a particular branch in the program is predicted with a default value (Not Taken). After branch's execution, if the outcome was taken, an entry in the PT table is inserted and the LRU field is correspondingly updated. The newly added PT entry fields are filled with the updated information from the branch itself (PC into TAG, OPC, nOP1, nOP2) and data from the RU table (LDPC into PC1 or PC2). Every time after a non-branch instruction - having the corresponding RC field greater than 0 - is executed, the PT table is searched upon its PC, in order to find a hit with the PC1 or PC2 fields (if RC=0, obviously it isn't any reason for searching the PT table). When a hit occurs, the branch stored in that PT line is executed and the corresponding result (taken/not -taken) is stored into the PRED bit. Next time when the program reaches again the same branch, the outcome of the branch is founded in the PT table, as it was previously calculated, and thus its direction it is surely known before branch's execution. In this way the processor knows for sure which of the program's path should be further processed. The only miss-predictions that may arise are coming from the initial learning (so named compulsory or cold miss-predictions) or from the fact that PT table has a limited size and therefore capacity miss-predictions may also occur.

However, the designer must be very careful about the pipeline timing. There are needed at least one and up to three cycles, depending on the pipeline length and structure, between the instructions that alter the registers and the corresponding branch instructions. This is because the branch may follow the instruction that produces its operands too closely in the program flow and thus the former instruction cannot finish its execution properly. The branch instruction cannot start its execution right away because it would trigger a *Read after Write* (RAW) hazard and it cannot be used the result from the prediction structure because it hasn't been yet calculated. So, we should postpone the branch processing few cycles, allow the previous instruction to finish and, after this, trigger the supplementary comparison. The minimum number of cycles that should separate the instruction that alter the registers from the corresponding branch instruction, analogously with the *Branch Delay Slot* term, we named PIDS (*Producer Instruction Delay Slot*). In order to fill this

PIDS we propose some program scheduling techniques, that will fill this PIDS when n ecessary, with control independent instructions (statically or dynamically). This was proven as being a feasible exciting solution, but we'll not focused on it during this work. Anyway, the PCB structure will therefore only help in those cases where the comparison process can be successfully moved several instruction slots ahead of the branch, without increasing the length of the schedule. The proposed PCB technique is then used if the comparison is far enough ahead, else conventional prediction might be us ed. Some previous valuable work about filling the PIDS with control independent instructions with very optimistic results and details could be found in [18].

We illustrate the working of this scheme using the example shown in Figure 2 and Figure 3. The Fig ure 2.A shows the sequential actions took after the execution of the instruction from "p1" address. The LDPC field corresponding to the destination register (R1 here) is filled with the instruction's PC (p) (the number that follows the PC label says that is the first encounter of that instruction; next time it will be 2 and so on). Because the RC field of the same register is '0' it means we have completed our actions related to instruction "p". Similar actions are followed for the instruction having the PC noted "c". After decoding the "b" branch, the PT table is searched for a hit on TAG, PC1, PC2 fields (in the "b" set). Due to a miss (this being the first instance of "b" branch) a default prediction is used. If after the "p" instruction's execution, its outcome is taken and a new line in the PT table is added; also the LRU field is correspondingly updated.

This time when the "p" instruction is issued again (Figure 3.A) the RC field attached to R1 register is greater then zero and the PT table is fully sea rched for a hit on PC1 or PC2 field. A hit is obtained and it triggers a supplementary branch execution (after obtaining the operands values from RU) and the result (taken/not taken) is correspondingly updated into the PRED field. Similarly actions are presented in Figure 3.B for the second issue of the "c" instruction. When the branch itself, about we are talking, is issued (Figure 3.C) the PT is searched into the "b" set. This time a hit occurs and the behavior of the branch "b" is extracted from the PRED field (Taken/Not Taken). This outcome is 100% accurate, because it has been correctly calculated in the previous described steps. For a more in depth understanding of the proposed PCB algorithm, we have provided a pseudo-code description in Appendix A.

3. Complexity Costs Evaluations

The global performance of a branch prediction scheme can be investigated from, at least, two points of view: prediction accuracy (local performance) and respectively architectural complexity (costs). The costs themselves ca n be split in two parts: the table's sizes and the time spent to access them. In order to evaluate the time corresponding to one branch prediction process (e.g. tables searches, supplementary branch's condition execution, etc.), we defined the next time qu otas:

T_{DM} – time needed for one direct mapped table access (RU)
T_{FA} – time needed for one fully-associative table access (PT)
T_{SA} – time needed for one set-associative table access (PT)
T_{EX} – time spent for one supplementary branch execution

Also we have considered:

N_B – the number of branch instructions
N_{NB} – the number of non - branch instructions
$N_{NB} = k_N * N_B$, where k_N is a statistical constant based on some program profiling ≈ 7
N_{NBL} – the number of non - branch instructions "linked" (through the RC field) with a branch instruction
$N_{NBL}=k_L*N_B$, where k_L is a statistical constant based on some program profiling ≈ 5
N_{NBEX} – the number of non - branch instructions followed by a "supplementary branch execution"
$N_{NBEX} =k_{ex}*N_B$, where k_{ex} is a statistical constant $\approx$ 1,3 (about 30% branches)

Now, the time spent in the branch evaluation process for one branch is formed by:

<u>A) Search time spent by a non - branch instruction</u>
$N_{NB}*T_{DM}$ – needed to check the RC field from RU
Every non - branch instruction that write s into a register triggers a search into the PT table for a hit with PC1 or PC2. To reduce these full table searches we have used instead this direct mapped table access to check the RC field (if this instruction is "linked") and proceed to the full table search only when the RC is not 0.

<u>B) Search time and execution time needed by a "linked" non - branch instruction</u>
$N_{NBL}*T_{FA(in\ PT)}+N_{NBEX}*(T_{DM(in\ RU)}+T_{EX})$, "linked" instructions search the PT table. When a hit arises, the operands values are taken from th e RU table and an execution follows (T_{EX})

<u>C) Search time needed by a branch instruction</u>
$N_B*(T_{DM(in\ RU)}+T_{SA(in\ PT)})$, when a branch is encountered a search in the PT table is performed to extract the corresponding prediction computed before

The overall tim e needed for one branch prediction is:
$$T= N_{NB}*T_{DM} + N_{NBL}*T_{FA}+N_{NBEX}*(T_{DM} + T_{EX}) + N_B*(T_{DM}+T_{SA})$$
$$\mathbf{T_{PT} = N_B*(}k_N* T_{DM} + k_L*T_{FA}\mathbf{+k_{EX}*(T_{DM} + T_{EX}) + T_{DM}+T_{SA})} \quad (1.1)$$

The time costs presented above we think that it should be necessary to be compared with a classic BTB having the same number of rows and a fully associative organization. For the BTB considered, the time needed to predict a branch is reduced to search the BTB at every branch instruction. So, the overall time in this case is $\mathbf{T_{BTB}= N_B*T_{FA}}$ (1.2)

Considering common sense values for constants involved as: $k_N \approx 7$, $k_L \approx 5$, $k_{ex} \approx 1{,}3$ and $T_{FA} = 4 * T_{DM}$, $T_{SA} = 1{,}2 * T_{DM}$, the (1.1) and (1.2) equations become:

$$T_{PT} = N_B * T_{DM} * 30{,}5 + N_B * T_{EX} * 1{,}3 \qquad (1.1')$$
$$T_{BTB} = N_B * T_{DM} * 4 \qquad (1.2')$$

At the first sight the time cost difference needed for one branch may seem overwhelming, but we think that other internal processes can hide some of the times from T_{PT}. So, the times wrote with italic font in the T_{PT} expression $N_B*(k_N* T_{DM} + k_L*T_{FA}+k_{EX})$ may overlap with the next instruction processing or data reuse process and the T_{PT} expression becomes now:

$$\mathbf{T_{PT} = N_B*(k_{EX}*(T_{DM} + T_{EX}) + T_{DM}+T_{SA})}$$

Using this new expression we have obtained:

$$T_{PT} = N_B * T_{DM} * 3{,}5 + N_B * T_{EX} * 1{,}3 \qquad (1.1'')$$
$$T_{BTB} = N_B * T_{DM} * 4 \qquad (1.2'')$$

Now the two expressions, in our opinion, are relatively comparable as processor time spent.

As we have stated above, the actions expressed by the times wrote with italic fonts in the T_{PT} expression may overlap with some other actions corresponding to the same non-branch instruction. While the instruction is executed (or even reused !) the RU table may be checked for the RC field and on a hit the PT table searched for PC1 or PC2 fields. All these operations can be done in parall el because these actions do not depend on each other, thus they are hidden into the real processor time consumed. The part from the T_{PT} expression that cannot yet be hidden is that which express the times involved in the supplementary branch execution: accessing the RU table for branch's operand values and the branch execution. It's quite obvious that we cannot offset these actions above the end of the current instruction's execution, when the instruction's result is produced. In place of trying to overlap these last actions with actions over the current instruction we could overlap them with the next instruction execution if they do not totally depend on each other. For this purpose we defined an average overlap probability (OP) which points out the overlapping degree with the next instruction's execution. After this (1.1'') and (1.2'') expressions becomes:

$$T_{PT} = N_B * T_{DM} * 3{,}5 + (1\text{-}OP)*N_B * T_{EX} * 1{,}3 \qquad (1.1''')$$
$$T_{BTB} = N_B * T_{DM} * 4 \qquad (1.2''')$$

The improvement brought by this scheme must be paid some way in costs. As we felt, if timing costs can be partially reduced by hiding them, the physical costs can not. Considering a register file RU with 32 registers and a PT table with M ($M = 2^j$) entries, the total size, in bits, is:

$\mathbf{D_{PT}}$=M·[(32-j)/*TAG*/ + 2·32/*PC1 and PC2*/ + 5/*OPC*/ + 2·5/*nOP1 and nOP2*/ + 1/*PRED*/ + 2/*LRU*/] + 32·(32/*LDPC*/ + 2/*RC*/) =
= **M·(114-j)+1088**

For a corresponding BTB having the features discussed above:

$\mathbf{D_{BTB}}$ = M·[(32-j)/*TAG*/ + 2/*prediction bits*/ + j/*LRU*/] = **M·34**

Considering tables (PT and BTB) with 1024 entries, we have obtained:

D_{PT} = 1024·(114-10)+1088=107584=105KBits and D_{BTB} = 1024·34=34KBits

4. Performance Evaluations through Simulation

The result for this second part of the paper were gather using a complex simulator built by the authors, on the kernel of the *SimpleScalar* simulation tool-set [1], an execution-driven simulator based upon the MIPS-I processor's ISA. The benchmarks used falls into two categories: the *Stanford HSA* (Hatfield Superscalar Architecture) benchmarks as described in [4, 11, 16], recompiled to run on *SimpleScalar* architecture and the *SPEC '95* benchmark programs [10] having as inputs, the files listed in Table 1. The benchmarks were run for maximum 500 millions instructions or to complet ion if they were shorter.

We performed several experiments to evaluate the newly proposed scheme. For this we have used table sizes of 128, 256, 512, 1024, 2048 entries having an associativity degree of 4. The results obtained on our PCB scheme were then compared with a BTB prediction scheme having an equivalent number of entries and two kinds of associativity degree: full associative and respectively 4-way set associative. For PCB we performed two experiments in order to evaluate the two ways of adding new entries in the PT table. First way is to add an entry in the PT table only if the branch was taken. The adopted strategy is to don't fill the table with branches that have a not-taken behavior (ANT=0). This solution reduces capacity misses, but we will have supplementary misses when the branch will be taken (end loop misses). The other way (ANT=1) is to add taken and not taken branches preventing the end loop misses. Of course, this will have a big impact on capacity misses when using small size tables.

Inserting entries in the PT table only when this is really necessary performs better on smaller tables because it reduces the capacity misses. In contrast, considering larger table sizes, where the capacity misses are not so frequent, adding every entry in PT reduce the end loop misses. The next experiment was to compare the newly proposed scheme (PCB) with similar classical dynamic prediction schemes. Figure 6 shows the amount of accuracy brought by the PCB scheme over two BTB schemes. The amount of "prediction" accuracy brought by the PCB scheme compared with a corresponding set associative BTB scheme using SPEC '95 benchmarks, is about 11%. As depicted in Figure 6, even with a full -associative BTB the PCB scheme performs better. The difference of accuracy between the PCB scheme and BTB schemes are even greater when using the Stanford benchmarks, about 18%, because these programs are more difficult to predict than SPEC benchmarks.

5. Conclusions and Further Work

The new proposed PCB algorithm clearly outpe rforms all the branch prediction schemes because it pre -computes the branch outcome before the branch will be really processed. From the pure "prediction" accuracy point of view this algorithm seems to be almost perfect. Similarly to branch history prediction schemes, branch information is cached into a "prediction" table (it doesn't really predict; more precisely, this table stores the branches behavior). Through this method, excepted the first one, every instance of a branch can be computed and therefore correctly anticipated, before its issue. The improvement in prediction accuracy brought by this scheme must be paid some way in timing and costs. Unfortunately, if the PCB's timing can be partially reduced by hiding it through some overlapping processes, the structural costs can not be reduced so easy. So, a PCB prediction scheme is about 105 KBits complex comparing with a full associative BTB scheme having only 34 KBits complexity at the same number of PT entries (1024 in this case).

As a further work we intend to measure the average PIDS (in cycles) based on SPEC '2000 benchmarks, and, as a consequence, trying to develop a software scheduler in order to fill – where it will be necessary - with some branch condition independent instructions these PIDS. Also we'll try to analyze in more depth other overlapping possibilities in order to reduce the PCB timing and also investigate the integration of the PCB scheme in some very powerful processor models, having some advanced architectural skills like value predic tion and dynamic instruction reuse concepts.

Acknowledgments

Our gratitude to Dr. Gordon B. Steven and Dr. Colin Egan from the University of Hertfordshire, England, for providing HSA Stanford benchmarks and for their useful friendship and support relate d to our Computer Architecture research during many years.

APPENDIX A

We are using the following notations and abbreviations in this annex:
PC – current instruction address
PC.nOP1 – register name for the first operand corresponding to the current inst ruction
PC.nOP2 – register name for the second operand corresponding to the current instruction
PC.OPCODE – instruction opcode corresponding to the current instruction
dimSet – the number of entries in a set
dimPT – the total number of PT entries
$PC_{m-1..0}$ – Least Significant m Bits of the PC
PT (Prediction Table) – set-associative organization after TAG and fully-associative after PC1 and PC2
To implement the PCB algorithm we have used the following helper functions:

- FOUND(j) - tests if a previous search in the PT table finished with success or not
- FIND_PT_ENTRY - it searches the PT table, in the PC's corresponding set, for a hit on the PC and PC1 and PC2 fields. When a hit occurs it returns the index of that entry in the PT table otherwise -1.
- ADD_PT_ENTRY - records a new entry in the PT table. The entry to be filled is selected using the FREE_PT_ENTRY function. If we had R0 as operand we will perform no decrementing because for the R0 register is useless to consider a RC field (there is no instruction to ha ve R0 register as destination). Now we can update the entry with the new data (TAG, PC1, PC2, nOP1, nOP2, OPC). Finally we have to link this entry with the corresponding operands by incrementing the RC field of those registers.
- FREE_PT_ENTRY - Its aim is to find a suitable entry in the PT table to be, first, evicted and then in that position to add a new entry.
- SCH_and_UPD_PT_TABLE - searches the entire PT table for a hit in the PC1 or PC2 fields. When a hit occurs the data stored into that entry (OPC, nOP1, nOP2) is used to execute a supplementary conditional operation. The result is then stored back in the PRED field of the same entry.

```
START:
0. FETCH_INSTR
1.DECODE_INSTR
2. IF isBRANCH(PC) THEN                    //this is a branch
3.        IF FOUND(FIND_PT_ENTRY(PC)) THEN
```

```
4.          PREDICTION=PT[FIND_PT_ENTRY(PC)].PRED //100% accuracy
5.      ELSE
6.          PREDICTION=NotTaken         //default prediction
7.          IF EXEC_BRANCH=TAKEN THEN
8.              ADD_PT_ENTRY(PC)
9.ELSE //not a branch instruction
10.     RD.REGVAL=EXEC_INSTR            //RD-destination register for the c urrent instruction
11.     RD.LDPC=PC
12.     if RD.RC >0 THEN
13.         SCH_and_UPD_PT_TABLE(PC) //search the whole PT table for PC1=PC or PC2=PC
                                     //on hit, update the prediction field of those entries
14. PC=PC+offset
15. [GOTO START]
```

Next we show the functions imple mentation

```
FOUND(j)
    IF j<0 THEN
        RETURN FALSE
    ELSE
        RETURN TRUE
END //FOUND

//searches for an entry in the PC set with (TAG=PC) and (PC1=PC.nOP1) and (PC2=PC.nOP2)
FIND_PT_ENTRY(PC)
    [stSet=dimSet*PC_{m-1..0}]'                 //first entry in the PC set
    [endSet=dimSet*(PC_{m-1..0}+1)]'            //first entry in the PC+1 set
    WHILE stSet < endSet DO                     //all this searches overlap
        IF (PT[stSet].TAG=PC) AND (PT[stSet].PC1=RU[PC.nOP1].LDPC) THEN
            IF NOT PC.OP2 THEN                  //there is no second operand
                RETURN stSet
            ELSE                                //there is a second operand
                IF (PT[stSet].PC2=RU[PC.nOP2].LDPC) THEN
                    RETURN stSet
        stSet++
    //end while
    RETURN -1
END //FIND_PT_ENTRY(PC)

//adds an entry in the PT table
ADD_PT_ENTRY(PC)
    IF PT[FREE_PT_ENTRY(PC)].nOP1 > 0  AND          //if this PT entry was taken and nOP 1 is not R0
      RU[PT[FREE_PT_ENTRY(PC)].nOP1].RC > 0 THEN    //don't go below 0
        RU[PT[FREE_PT_ENTRY(PC)].nOP1].RC--         //decrement the old refcount
    IF PT[FREE_PT_ENTRY(PC)].nOP2 > 0  AND RU[PT[FREE_PT_ENTRY(PC)].nOP2].RC > 0 THEN
        RU[PT[FREE_PT_ENTRY(PC)].nOP2].RC--         //decrement the old refcount
    PT[FREE_PT_ENTRY(PC)].TAG=PC
    PT[FREE_PT_ENTRY(PC)].PC1=RU[PC.nOP1].LDPC
    PT[FREE_PT_ENTRY(PC)].nOP1=PC.nOP1
    IF PC.OP2 THEN
        PT[FREE_PT_ENTRY(PC)].PC2=RU[PC.nOP2].LDPC
        PT[FREE_PT_ENTRY(PC)].nOP2=PC.nOP2
    ELSE
        PT[FREE_PT_ENTRY(PC)].PC2=-1
        PT[FREE_PT_ENTRY(PC)].nOP2=-1
    PT[FREE_PT_ENTRY(PC)].OPC=PC.OPCODE
    RU[PT[FREE_PT_ENTRY(PC)].nOP1].RC++             //increment the new refcount
```

```
        IF PC.OP2 THEN
                RU[PT[FREE_PT_ENTRY(PC)].nOP2].RC++          //increment the new refcount
END //ADD_PT_ENTRY


//full PT table search for PC in PC1 or PC2 fields
SCH_and_UPD_PT_TABLE(PC)
    [j=0]
    //this long time searches may overlap with EXEC_INSTR or data reuse process
    WHILE j<dimPT DO
            IF (PT[j].PC1=PC) OR (PT[j].PC2=PC) THEN
                    PT[j].PRED=EXEC(PT[j].OPC, RU[PT[j].nOP1].REGVAL,
                                               RU[PT[j].nOP2].REGVAL)
            j++
END //SCH_and_UPD_PT_TABLE
```

References

[1] *The SimpleScalar Tool Set.* Technical Report CS-TR-96-1308, University of Wisconsin-Madison, July, 1996 (www.cs.wisc.edu/~mscalar/simplescalar.html)

[2] Vintan L. - *Towards a High Performance Neural Branch Predictor*, Proceedings of The International Joint Conference on Neural Networks - IJCNN '99 (CD-ROM, ISBN 0-7803-5532-6), Washington DC, USA, 10-16 July, 1999

[3] B. Calder, D. Grunwald, D. Lindsay - *Corpus-Based Static Branch Prediction*, ACM Sigplan Notices, vol. 30, No. 6, pages 79-91, June, 1995, ISBN 0-89791-697-2

[4] L. Vintan - *Instruction Level Parallel Processors*, Romanian Academy Publishing House, Bucharest, 2000 (264 pp., in Romanian), ISBN 973 -27-0734-8

[5] G. Steven, C. Egan, L. Vintan - *Dynamic Branch Prediction using Neural Networks*, Proceedings of International Euromicro Conference DSD '2001,Warsaw, Poland, September, 2001

[6] G. Steven, C. Egan, W. Shim, L. Vintan - *Applying Caching to Two-Level Adaptive Branch Prediction*, Proceedings of International Euromicro Conference DSD '2001, Warsaw, Poland, September, 2001

[7] C. Egan, G. Steven, L. Vintan - *Quantifying the Benefits of Multiple Prediction Stages in Cached Two Level Adaptive Branch Predictors*, Proceedings of International Conference SBAC -PAD, Brasil, Braslia, September, 2001

[8] S. Sechrest, C. Lee, Mudge T. - *The Role of Adaptivity in Two-level Adaptive Branch Prediction*, 28th ACM / IEEE International Symposium on Microarchitecture , November 1995.

[9] T. Yeh, Y.N. Patt - *Two-Level Adaptive Branch Prediction*, 24th ACM / IEEE International Symposium on Microarchitecture, November 1991.

[10] T. Yeh, Y.N. Patt - *Alternative Implementation of Two-Level Adaptive Branch Prediction*, 19th Annual International Symposium on Computer Science, May 1995.

[11] G. Steven et al. - *A Superscalar Architecture to Exploit Instruction Level Parallelism*, Microprocessors and Microsystems, No 7, 1997.

[12] SPEC - *The SPEC benchmark programs* (www.spec.org)

[13] W.F. Wong – *Source Level Static Branch Prediction*, The Computer Journal, vol. 42, No.2, 1999

[14] J. Stark, M. Evers, Y. Patt - *Variable Length Path Branch Prediction*, ASPLOS VIII 10/98, CA, USA, 1998

[15] M. Evers, S. Patel, R. Chappell, Y. Patt – *An Analysis of Correlation and Predictability: What Makes Two Level Branch Prediction Work*, ISCA, Barcelona, June 1998

[16] L. Vintan, C. Egan - *Extending Correlation in Branch Prediction Schemes*, Proceedings of 25th Euromicro International Conference, Milano, Italy, 8 -10 September, IEEE Computer Society Press, ISBN 0-7695-0321-7, 1999

[17] L. Vintan – *Towards a Powerful Dynamic Branch Predictor*, Romanian Journal of Information Science and Technology (ROMJIST), vol.3, nr.3, pg.287-301, ISSN: 1453-8245, Romanian Academy, Bucharest, 2000

[18] Collins R. - *Exploiting Parallelism in a Superscalar Architecture*, PhD Thesis, University of Hertfordshire, U.K., 1996

[19] Steven G., Egan C., Anguera R., Vintan L. – *Dynamic Branch Prediction using Neural Networks*, Proceedings of International Euromicro Conference DSD '2001, ISBN 0-7695-1239-9, Warsaw, Poland, September, 2001 (pg.178-185)

Table 1. Benchmark programs and inputs

SPEC '95 benchmarks			**Stanford HSA benchmarks**		
Benchmarks	Input	Inst. Count executed	Benchmarks	Input	Inst. Count executed
Applu	applu.in	500000000	fbubble	Null	875174
Apsi	apsi.in	500000000	fmatrix	Null	824443
Cc1	1stmt.i	500000000	fperm	Null	581099
Compress95	bigtest.in	500000000	fpuzzle	Null	25271829
Fpppp	natoms.in	500000000	fqueens	Null	365205
Hydro	hydro2d.in	500000000	Fsort	Null	198305
Ijpeg	vigo.ppm	500000000	Ftower	Null	459788
Li	*.lsp	500000000	Ftree	Null	267642
Mgrid	mgrid.in	500000000			
Perl	scrabbl.pl	500000000			
Su2cor	su2cor.in	500000000			
Swim	swim.in	500000000			
Tomcatv	tomcatv.in	500000000			
Turb3d	turb3d.in	500000000			
Wave5	wave5.in	500000000			

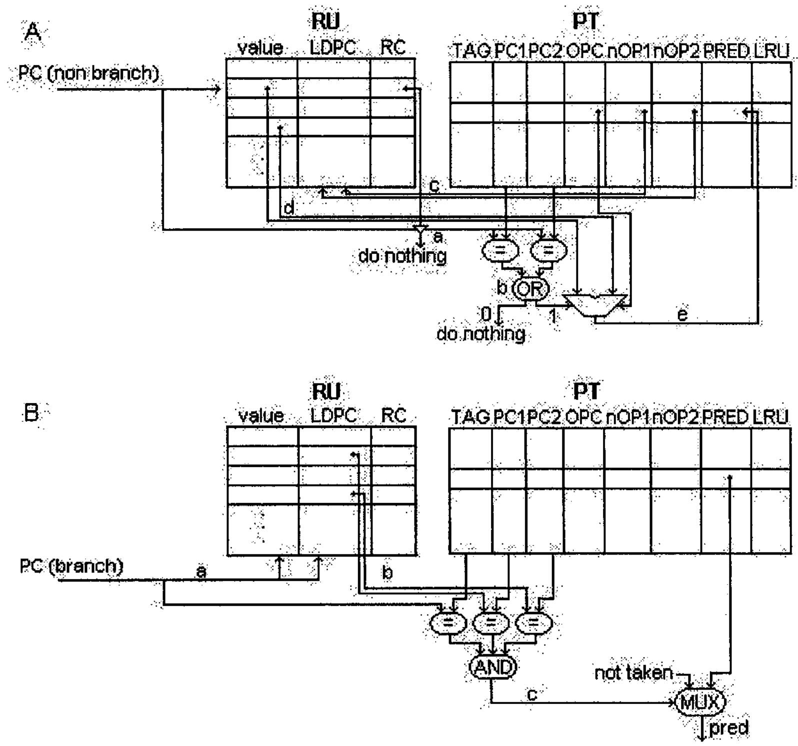

Figure 1. The new proposed prediction scheme. A) when a non-branch instruction is encountered; B) when a branch instruction is encountered

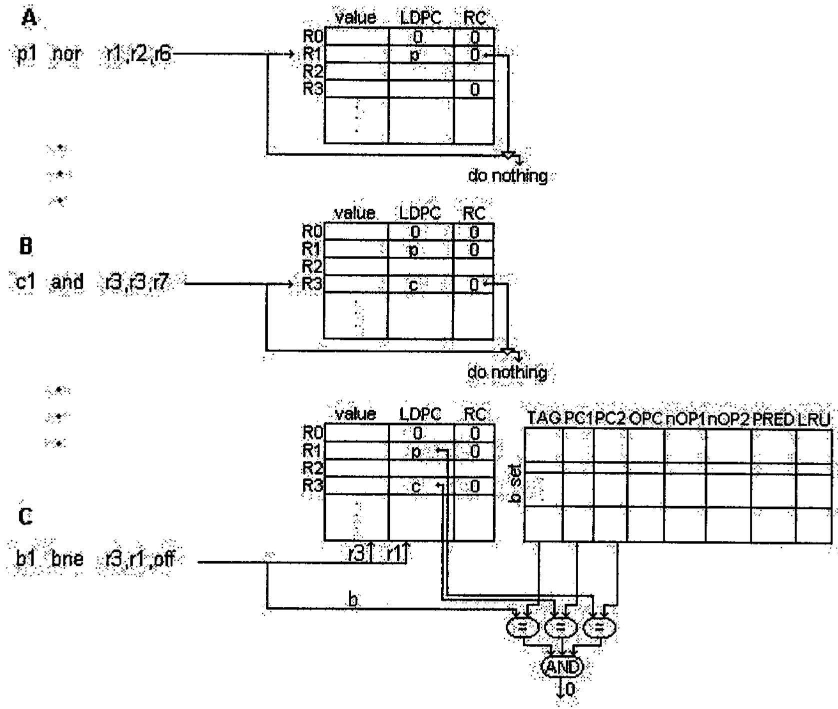

Figure 2. Example of the first instance of a particular branch; A), B) actions took when issuing non-branch instructions; C) actions took before b1 branch execution

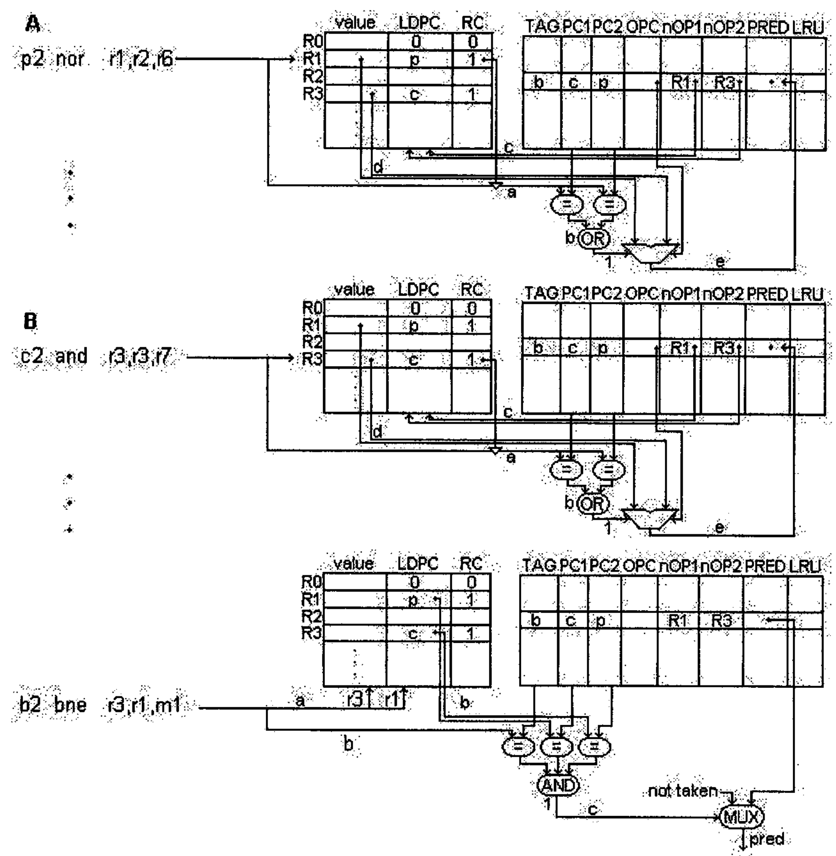

Figure 3. Example of the second instance of a particular branch; A), B) actions took when issuing non-branch instructions; C) actions took before b2 branch execution

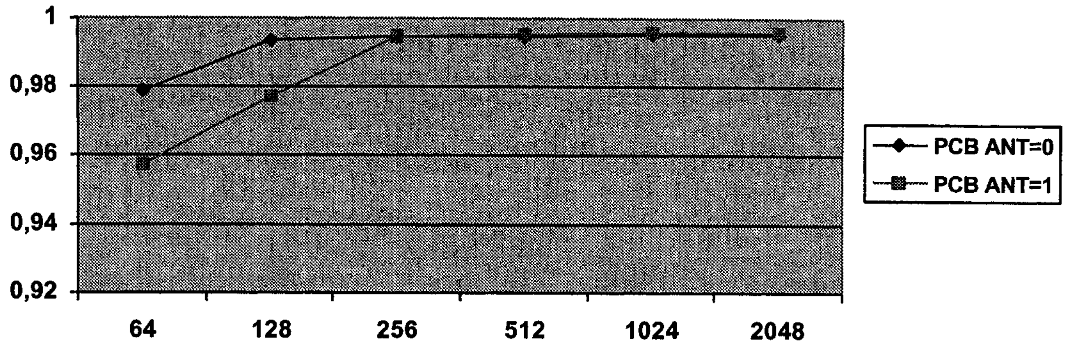

Figure 4. PCB's average "prediction" accuracy obtained on Stanford benchmarks

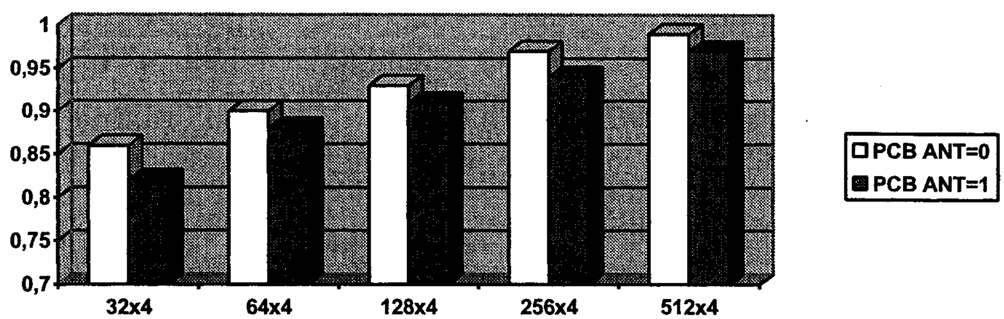

Figure 5. PCB's average "prediction" accuracy obtained on SPEC '95 benchmarks

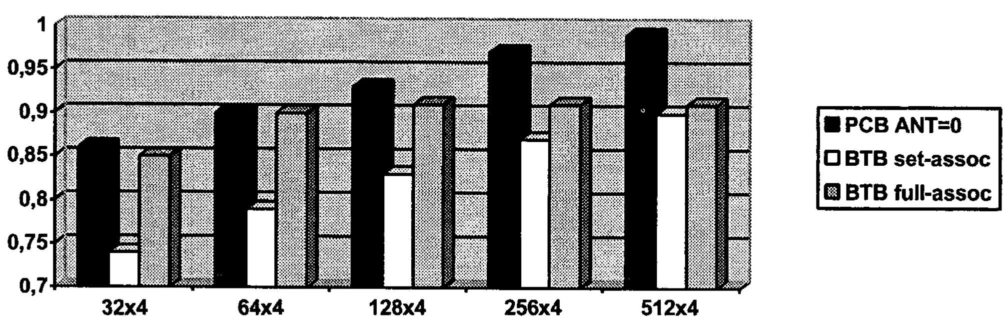

Figure 6. Average "prediction" accuracies obtained on SPEC '95 benchmarks