Efficiency of Pre-Computed Branches

Mohammed Aamer, Kevin Lux, Ravi Mistry, Brian Mulholland
aamer@seas.upenn.edu, luxk@saul.cis.upenn.edu, ravimist@seas.upenn.edu, mulholla@seas.upenn.edu

Dept. Of Computer Science
University of Pennsylvania
3330 Walnut Street
Philadelphia, PA 19104

ABSTRACT
In this paper we discuss an alternative to the existing branch prediction algorithms, called pre-computed branch prediction. This method does not actually involve prediction, as the branch's outcome is calculated as soon as its last operand is known. We attempted to validate the claim that pre-computed branches give a higher rate of performance than the existing branch prediction methods, by implementing the pre-computed branch algorithm in SimpleScalar. After analyzing our data, we concluded that one does get a higher hit rate and IPC if pre-computed branches are used, which results in increased performance.

1. INTRODUCTION
Branching is a big problem in computer architecture, especially with modern processors that run at such high clock speeds and contain increasingly deeper pipelines. Branches interrupt the control flow of a processor, which can then decrease performance significantly if that branch is not handled correctly. There are numerous methods used today to try and minimize this negative effect on performance. We designed this project after reading a paper that discusses one approach to solving this problem, called "An Alternative to Branch Prediction: Pre-Computed Branches"[6]. Pre-Computed Branch Prediction uses dedicated hardware to keep track of the locations of branches and their operands. Then whenever that branch is encountered again, the pre-computed branch hardware can figure out its outcome as soon as its arguments are computed. This means that except for the very first time a branch is encountered, and except for structural hazards due to the prediction table size, the processor can know with 100% certainty whether the branch is taken and what the target of the branch is.

To investigate the ideas of their paper, we first modified the SimpleScalar sim-func simulator to see what kind of speedup we could expect from implementing the pre-computed branch architecture. We then emulated the pre-computed branch architecture in the SimpleScalar sim-DLX simulator to see if our actual results matched our expected results. Lastly, we ran the six integer benchmarks with pre-computed branches and compared the results to those obtained by using the traditional branch prediction algorithms.

2. BACKGROUND
In most programs, between 10 to 20% of the instructions are branches. There are methods that we can use to try and predict whether a branch is taken, but ultimately they are just guesses, which have to guess incorrectly a portion of the time. If the processor predicts incorrectly, but it has already begun to execute instructions from the incorrect branch, then it is forced to flush all of the wrongly-predicted instructions after the branch and start over. This results in a big loss of performance. And it's not enough to simply predict whether a branch is taken or not - we must also be able to figure out the value of the new PC, or else we will have to stall. Therefore, good branch handling methods are essential if we are going to get the most out of a processor.

There are many methods used in modern processors to try and avoid the potential performance hit caused by branches. If no tricks were used at all, we would have to wait until the end of the execution stage of the pipeline every time a branch instruction was encountered, to know exactly what instruction to fetch next. Assuming that 15% of the instructions are branches, and the execution stage was the third stage of the pipeline, this could cause a 30% increase in the CPI of a processor.

2.1 Branch Prediction Methods
Speculative execution is used to try and avoid these control stalls. With speculative execution, the processor guesses whether or not the branch is taken, and if so what its target is. Once the processor figures out this information, it can begin executing at that point immediately. Later on, when the result of the branch is actually known, the processor can see whether its guess was correct or not. If the guess was correct, the processor can continue to execute. But if the guess was incorrect, the processor must now flush its pipeline of all the incorrect instructions it has been executing, as well as undo any incorrect changes that were made.

Branch prediction is the method used to "guess" whether or not a branch is taken. There are two kinds of branch prediction, static and dynamic. Static branch prediction schemes always do the same thing: always predict taken, always predict not taken, always predict certain opcodes taken, etc. Always predicting taken isn't a terrible idea, because about 65% of branches are taken - but this still means a 35% misprediction rate. Dynamic branch prediction, on the other hand, uses special hardware to determine whether a branch is taken.

One kind of dynamic branch prediction is a Branch History Table (BHT). The BHT is a 1-bit wide table that is indexed by the least significant bits of a branch PC. When a branch is encountered, the BHT is checked to see whether that branch was taken or not taken the previous time it was encountered. If there is a misprediction, the BHT is updated. However, multiple PCs may map to the same BHT entry, and in addition it will mispredict every time the branch outcome switches. To improve on this model, two-bit saturating counters were introduced. The two-bit

counters perform better on average because they must mispredict twice in order to change their prediction. Correlating predictors are another dynamic branch prediction method. Correlating branch predictors base their predictions on the outcomes of branches that occurred recently in the program, because there is often some correlation between the current branch and the previous few branches. Hybrid predictors have multiple predictors, each using a different predictor scheme. There is a "chooser" which selects which scheme to use for each branch. If the chosen predictor is wrong for a given branch, the chooser is flipped for that branch and the other scheme is used until the next misprediction.

Predicting whether or not a branch is taken is just one aspect of speculative execution - you also need to know what the target PC is. A branch target buffer (BTB) is used to keep track of previous branches and their targets. When an instruction is fetched, its address is used as an index into the BTB table. If there is a match, we know that the instruction is a branch. If that branch is predicted as taken, then the branch destination is obtained from the BTB, and the processor begins to fetch and execute at that new instruction. Certain BTBs not only store the destination address, but also store the entire instruction at that address. This is called a Branch Target Cache (BTC). However, BTBs and BTCs can be rather large, and therefore expensive to implement - in terms of both money and chip space.

2.2 Need for a More Efficient Algorithm

Moore's Law states that the number of transistors on CPUs doubles about every 18 months which eventually translates to double the performance in approximately every two years [8]. To keep up with the Moore's law we would need more efficient branch prediction techniques.

Modern processors use pipelining to exploit parallelism and improve performance. Conditional branches in the instruction stream decrease performance by causing pipeline flushes in case of mis-prediction. Efficient branch prediction mechanisms can overcome this limitation by predicting the outcome of the branch before its condition is resolved. As a result, instruction fetch is not interrupted as often and the window of instructions over which instruction level parallelism can be used increases. In fact, accurate branch predictors can eliminate most of these pipeline stalls and are thus critical to realizing the performance potential of a processor.

As the design trends of modern superscalar microprocessors move toward wider issue and deeper super-pipelines, effective branch prediction becomes essential to exploring the full performance of microprocessors. Processors with a simple five-stage pipeline typically have a two-cycle branch penalty. For a four-way superscalar design, however, this could mean a loss of eight instructions. If the pipeline is extended, the branch penalty usually increases, resulting in the loss of even more instructions. Since programs typically encounter branches every 4–6 instructions, inaccurate branch prediction causes severe performance degradation in highly superscalar or deeply pipelined designs. Improving branch prediction accuracy by using new more efficient algorithms is important for reducing these larger mis-prediction penalties.

To increase the branch prediction accuracy different branch prediction techniques have been developed as mentioned above. These branch prediction schemes can achieve average prediction accuracy in the range of 80-95% depending on the type of prediction and the program being executed. Although it must signify a good branch prediction performance, there is a growing need for new techniques that can be used to improve it even further, in order to stay with new advances being made in the field of pipelining. One of the effective techniques that can be used in this regard is pre-computed branch algorithm.

3. PRE-COMPUTED BRANCHES

As we have seen, there is a need to develop a branch prediction algorithm that outperforms today's traditional branch prediction algorithms like two bit counters, two-level adaptive branch predictors [1,2,3,4] and hybrid/tournament predictors [5]. This paper will demonstrate an algorithm that is an alternative to the traditional prediction algorithms. In this paper we detail and implement the pre-computed branch prediction algorithm described in [6]. Instead of predicting the branch outcome, a pre-calculated branch prediction (PCB) determines the outcome of the branch as soon as all the operands of the branch instruction are known. This outcome is then cached in to a 'prediction' table (PT). The main idea of the pre-calculated branch algorithm is that each branch, when first encountered, inflicts a capacity miss; then this branch is cached into the prediction table, the result of any subsequent non-branch instruction that modifies any operand of a cached branch instruction is stored in an extension of the register table called the 'register unit' table (RU). The outcome of the branch is then recalculated and cached back into the PT. Through this algorithm, except for the first time and except for capacity misses due to the size of the prediction table, every branch will be predicted with 100% accuracy.

3.1 Implementation Methodology

Two tables are required to implement pre-computed branches- the 'register unit' table (RU) and the 'prediction' table (PT). The RU table is an extension of the register file. The RU maintains the register file meanings, but in addition to the register file, each entry in the RU also has two new fields named LDPC and RC. The RC field is a reference counter that gets incremented by one whenever an instruction, whose label is in the PC1 or PC2 field of the PT, writes into the attached register. Whenever a branch instruction is evicted from the PT table, the RC field of its corresponding operates are decremented by one. Therefore, if a register has a RC field of 0 then it would mean that there is no branch instruction in the PT table that uses that particular register as an operand. The RU table and the PT table can be seen in Figure 1(a and b).

The PT table stores the pointers to the last branch's operand producers (PC1 and PC2) along with the TAG of the branch. The PT also stores the branches opcode in OPC and the register names of the branches operands in nOP1 and nOP2 respectively. PRED shows the path the branch will take when it is encountered next (taken or not-taken). The LRU field is used to keep track of the least recently used branch. The size of the PT table is an important factor in calculating pre-computed branches. In the structure of the table it is apparent that the PC is not used to index the RU table, it is instead used for some associative searches in the PT table and in some cases, as we will see soon, it is updated into the LDPC field. We will now explain the implementation of the pre-computed branch algorithm by demonstrating how the algorithm handles branch instructions and non-branch instructions.

3.1.1 Branch Instructions

Figure 1a details the logic required to handle branch instructions with the RU and PT tables. We will first explain what happens when a branch is encountered for the first time and then explain what happens when it is encountered again. When any branch is encountered the PT table is searched for hit on TAG, PC1 and PC2 fields, if the branch is encountered for the first time it will not exist in the PT. The first issue of every branch in the program is always predicted to be not-taken. After execution, if the branch was not-taken then nothing is done, but if it was taken then its TAG, OPC, OP1 and OP2 are placed into the PT table. The PC1 and PC2 values are filled by the LDPC of the operands from the RU table, and lastly, the LRU field is updated appropriately. When the branch instruction is issued again, the PT is searched for a hit on TAG, PC1 and PC2. Each time a hit occurs and the branch outcome is available from the PRED field, this outcome is 100% accurate.

3.1.2 Non-Branch Instructions

The logic for when a non-branch instruction is encountered is different from that of a branch instruction, as seen in Figure 1b. Every non-branch instruction that writes something into a register triggers a search into the PT table for a hit with PC1 or PC2. But, if the RC of those registers is 0, then the PT table is not searched since there will be no branches in it that use the register as an operand. If the RC is greater than 0, then PT table is searched for a hit on PC1 or PC2 field. If a hit is obtained then the updated operand value from RU is obtained and a supplementary branch execution is performed, the obtained result (taken or not-taken) is updated into the PRED field.

From the above description of the implementation of pre-computed branches, it is obvious that the only mispredictions that occur by using our algorithm, are when branches are encountered for the first time (compulsory misses) and when branches are replaced due to the limited size of the PT table (capacity misses).

4. IMPLEMENTATION IN SIMPLESCALAR

Our purpose is to enable sim-func to count the number of instructions between when the last register needed to compute the branch was written, and when the branch was executed. This result would be the ideal theoretical number of instructions before a branch, that its outcome was available. For sim-DLX, we needed to hack it to enable pre-computed branch functionality and then also needed to hack it to count the number of instructions in advance a branch outcome is actually available. This number would be the actual value as opposed to the theoretical value calculated through sim-func. We then compare the theoretical value and the actual value and also compare the results of the traditional branch algorithms with pre-computed branches in sim-DLX.

4.1 Sim-func

In sim-func, we first check to make sure that both the input registers are not empty. After that we compare the two input register's PCs and use the most recent instruction number to update the number of instructions before the branch that the operand was available. If we are writing the registers, then we update the time the register was last written.

Then we make the following output via print_counter when execution of a benchmark has completed:

branch_vars_avail : displays how many instructions are branches and how many are non-branches that change the branch instructions operands
branch_insn : the total number of branch instructions
branch_vars_avail : the number of instructions a branch outcome is available before the actual branch is encountered

4.2 Sim-DLX

In sim-DLX, we first implemented the above code from sim-func to calculate the actual number of instructions before a branch that its outcome was available. As is mentioned in the section on Pre-computed branches, we needed to implement the RU and the PT tables in sim-DLX to enable pre-computed branch functionality. We created the RU and the PT structures in machine.h. The RU has the same number of entries as the register file and is an array. RU[1] is the same as Register[1] and so on. The RU holds a reference count, which counts how many entries in the prediction table reference a particular register. This is used so that when a register is written, it can be astute about recomputing branches (If there's no reference count, don't recompute). The RU also contains a LDPC field, which contains the PC that last wrote the register. This is used so that the prediction table can determine if the same situation exists when recomputing a branch. For the PT table, we create a structure that contains nOP1 and nOP2 which are the register numbers that are the inputs to the branch. i.e. bne r1, r2, pc (nOP1=1, nOP2=2). It also contains PC1 and PC2, which were the last PCs of the instructions that wrote the above registers before the branch was executed. (In other words, the PCs that generated the values used to compute the branch.) The PT also contains the OPC (opcode of the branch), the prediction, LRU (for determining which entry to kick out when full) and a tag (the PC of the branch for matching later).

We use these two structures in sim-DLX, and set the size of the RU to the size of the register table (MD_TOTAL_REGS) and the size of the PT table to 256 entries. The following prototypes are added for the PT table-

```
int find_pt_entry(void);
int free_pt_entry( void );
void add_pt_entry(md_addr_t addr, struct predec_insn_t *pdi);
void update_pt_table();
```

Then in sim-DLX we zero out the reference count on all registers in the RU table and also zero out the registers, LRU and tag info in the PT table. We then set the branch predictor option to none. And then enable the statistics of the pre-computed branches to be displayed at the end of the simulation along with other results. We choose to display:

sim_sample_pcb_computed,
sim_sample_pcb_misses,
sim_sample_pcb_hits,
sim_sample_pcb_hitrate.

to compare the obtained results with the other branch prediction algorithms and then to analyze them. Then, we implemented the actual pre-computed branch algorithm in sim-DLX as was explained in section 3. Since, the algorithm has already been explained in detail we only present the flow, in pseudo-code, that explains what we did:

```
if (instruction is a branch)
        check the prediction table using the PC
        if (found an entry)
                update LRU
                return correct prediction
```

```
        else
                guess branch not taken
if (branch taken)
        add new pt entry for this pc
else
        if (output register is valid)
                ru[register].ldpc = pc
                if (ru[register]rc &gt; 0)
                        update pt table for register

find entry - compare tags of current pc (of the branch)
if the tags match, and the last written PCs of the registers match
return the index
add new pt entry
        get an available entry
        copy the PC to tag, opcode to OPC, register numbers to
        nop1, nop2
        copy the LDPC for registers to pc1, pc2
        set LRU
        set PRED to true
update pt table
        find all entries where nop1 or nop1 = written register
        recompute those branches
```

5. EXPERIMENTAL RESULTS

We had to make decisions about which parameters to use for our modified sim-func and sim-DLX. For sampling we decided to use the following parameters:

-insn:sample:first 40000000:0:10000000

-insn:sample 490000000:0:10000000

The above numbers were not chosen for any particular reason, just to make sure that the simulations would not take as long as they would without sampling. They were used consistently across all simulations.

We had decided to run sim-func and sim-DLX on the six integer benchmarks because they are less loop intensive and more procedure intensive than the floating point benchmarks. Therefore, they are more prone to being mispredicted than floating point benchmarks. Thus we believed that they would test the performance of a branch prediction algorithm more accurately.

5.1 Theoretical Implementation

We ran our modified sim-func using the above sampling arguments on each of the integer benchmarks. The following are the results we obtained:

Sim-func

Benchmarks	# of instructions
gcc.eio	171
bzip2.eio	58773
crafty.eio	1249
eon.kajiya.eio	8382
twolf.eio	30
vpr.route.eio	322

Table 1

The results we got are the theoretical number of instructions a branch outcome is available before the branch is encountered. These are the number of instructions the outcome is available before the pre-computed branch algorithm is implemented and before latency is brought into the picture. Therefore, when we graph these results with the results obtained from sim-DLX, we will be able to see how much the actual implementation varies from the theoretical one.

5.2 Actual Implementation

To simulate the branch prediction algorithms (other than pre-computed branches) we have to use the unmodified sim-DLX since in our hacked version of sim-DLX we set all branch prediction algorithms to none. The following are the cache parameters we used in our sim-DLX simulations of one bit counter, two bit counters, two level predictor and hybrid predictor. They are held constant through all the six integer benchmarks.

The configuration of the caches used here are: 16KB, 32B line, 2-way set-associative L1 data cache with LRU and 16KB, 32B line, 2-way set-associative L1 instruction cache with LRU. The TLB data and instruction cache are both 32-entry, 2-way set-associative, 4K page with LRU replacement policy. The L2 cache is 512KB, has 64B lines and is 4-way set-associative with LRU. L2 has hit latency of 10 cycles and TLB has miss latency of 10 cycles.

We have used the following branch prediction methods to compare with the pre-computed branch results- one bit counter, two bit counters, two level predictor and hybrid predictor. We now detail the processor configuration used in each case. The cache configuration mentioned above is common to all the cases.

One bit counter- A BHT with 1024 entries and a 1-bit predictor is used to predict branches.

Two bit counter- A 2 bit BHT with 1024 entries is used.

2-level predictor- We use a BHR with 2 bits of branch history and 1024 entries XORed with the PC. We also use a 2 bit BHT with 1024 entries.

hybrid predictor- We used a hybrid of the 2-level predictor mentioned previously and a chooser with 1024 entries and a 2 bit predictor.

pre-computed branches- We used our modified sim-DLX and passed each of the integer benchmarks one after the other using only the sample parameters mentioned at the start of this section.

After obtaining results for each of the six benchmarks, we tabulated the hit rates and the IPCs of the above mentioned prediction methods to analyze the performance of pre-computed branches against the popular branch prediction algorithms. The following tables display our obtained results:

Sim-DLX
Hit rate

Benchmarks	1 bit counter	2 bit counter	2 level	Hybrid	PCB
gcc.eio	0.8512	0.8830	0.8400	0.9073	0.8677
bzip2.eio	0.9555	0.9687	0.9669	0.9745	0.9483
crafty.eio	0.8152	0.8613	0.8465	0.9019	0.9269
eon.kajiya.eio	0.7838	0.8481	0.9172	0.9379	0.9070
twolf.eio	0.7467	0.8125	0.7957	0.8537	0.9581
vpr.route.eio	0.9065	0.9247	0.9433	0.9464	0.9472
Average	0.84	0.88	0.88	0.92	0.93

Table 2

Sim-DLX
IPC

Benchmarks	1 bit counter	2 bit counter	2 level	Hybrid	PCB
gcc.eio	0.7182	0.7212	0.7174	0.7235	0.8557
bzip2.eio	0.9454	0.8469	0.8467	0.8473	0.9587
crafty.eio	0.8121	0.8149	0.8139	0.8183	0.9454
eon.kajiya.eio	0.8069	0.8090	0.8193	0.8207	0.9330
twolf.eio	0.8019	0.8037	0.8054	0.8084	0.9216
vpr.route.eio	0.8367	0.8367	0.8388	0.8391	0.9445
Average	0.82	0.81	0.81	0.81	0.93

Table 3

The above tables enable us to effectively compare the branch prediction algorithms, which we do in the proceeding section.

When we run the benchmarks on the modified sim-DLX we also get the number of instructions the branch outcome is available before the branch is encountered on the screen along with the hit rate and the IPC. We tabulate this data in the following table for comparison with table 1.

Sim-DLX

Benchmarks	# of instructions
gcc.eio	20
bzip2.eio	81571
crafty.eio	88
eon.kajiya.eio	25
twolf.eio	2
vpr.route.eio	28

Table 4

6. ANALYSIS OF RESULTS

Using our modified sim-func, we calculated the average number of instructions before branch execution that we knew its outcome. But this is only the theoretical maximum - the actual number of instructions before execution that the outcome was known through our pre-computed branch method, implemented in a modified sim-DLX, is less than the theoretical maximum as seen in Figure 2. This can be explained by the latency that occurs when using the pre-computed branch algorithm. Our results show that we can still obtain the branch's outcome far enough in advance that we can still achieve some performance improvement.

The latency is due to the time spent searching through the pre-computed branch structures. When a branch instruction is encountered, we must access the entire PT table and an extended register table (RU), searching for a match for the tag of that branch. Also, if a capacity miss is encountered, that means we must take the time to fill up the prediction table with the branch's tag, OPC, OP1 and OP2 , this results in higher latency.

Using pre-computed branches, the hit rate obtained was more when compared with the hit rates of other mainstream branch prediction methods as evident in Figure 3. On average, the hit rate obtained from pre-computed branches was slightly higher than the hybrid prediction algorithm, but when compared to the 1-bit counter, 2-bit counter, or 2-level methods, the difference was significant. The increase in hit rate in our pre-computed branch method can be explained because our only branch mispredictions come from capacity or compulsory misses. If we were using a prediction table larger than 256 entries, the capacity misses would decrease significantly. However, this could introduce a higher latency, so some compromise between the two would have to be reached.

There were some small discrepancies in hit rate when applied to certain benchmarks. For example, our hit rate for the bzip benchmark was actually the lowest out of all the branch prediction methods. This anomaly may be explained due to the fact that the pre-computed branch algorithm is only able to predict conditional, unconditional, and procedural branches, where the branch destination is explicitly specified. The only remaining branch type requiring a predicted target address for the pre-computed branches is an indirect jump where the destination may be specified by values computed during execution. When an indirect branch is encountered, no branch prediction of any kind is being used, thus decreasing the hit rate proportional to the amount of indirect branches encountered. However, we could accommodate this by using a very small branch target buffer, dedicated to predicting only the outcome of indirect jumps [7].

The performance might also have been lower due to the presence of Producer Instruction Delay Slots (PIDS). These slots deal with the minimum number of cycles that should separate the instruction that modify the operands of a branch and the actual branch. If the operand modifying instruction is too close to the corresponding branch then the branch would have to postpone processing for a few cycles, till the operand modifying instruction is finished. But, a BTB could be used to handle operand modifying instructions that are in the PIDS, this would result in no cycles being lost due to the branch having to wait for an operand to be ready. Thus, the performance loss that had occurred due to the presence of PIDS can be rectified by introducing BTBs. Whenever an algorithm is able to reduce the amount of mispredictions that occur, the IPC invariably increases. This occurs because less cycles are wasted recovering from an

incorrect prediction. Therefore, since our pre-computed branch algorithm reduces the number of branch mispredictions, not only is the hit rate increased, but the IPC is also improved. This is apparent in figure 4, where the IPC of our algorithm is consistently much higher than the IPC for every other algorithm, for each benchmark.

7. COST EVALUATION

To completely analyze the benefits of using the pre-computed branch algorithm over the other algorithms, we have to analyze the prediction accuracy in the form of hit or miss rates and we also have to judge costs in the form of hardware costs and architectural complexity. Results also show that the pre-computed- Branch architecture performs better than an architecture using only a BTB, and has significant hardware savings. This is particularly true for larger programs more representative of modern applications [7].

If the PT table is large, then only a small number of branches will be evacuated by the LRU algorithm, but the trade-off is that the latency required to go through the PT table will be high. If the table is small, then there is a high probability for a branch that might be encountered soon to be evacuated and the trade-off will be that latency will be low.

By using the pre-computed branch algorithm we can reduce the misprediction rate, which saves both the time and power necessary to flush the pipeline when recovering from a misprediction. This translates into lower power consumption than the other branch prediction algorithms, since they mispredict more often. As more and more processors are being incorporated in mobile devices like laptops, there is an ever-increasing need for effective power consumption without losing performance. The use of the pre-computed branch method can meet that need well.

8. CONCLUSION

After a careful analysis of the results, we can conclude that pre-computed branches outperform the existing branch prediction methods by resulting in a higher hit rate and higher IPC. We also demonstrated the latency that occurs by the introduction of the prediction table and the extension of the register file; this is apparent from the difference between the theoretical and actual results. When using pre-computed branches, an important trade off exists - the size of the prediction table versus latency. We have also noticed that not using BTBs to handle indirect branches and PIDS can prevent us from achieving higher performance. There are also other benefits to using pre-computed branches, such as power consumption and hardware cost.

Acknowledgments

We would like to acknowledge Dr. Amir Roth for providing us with invaluable advice and frequently guiding us in the right direction.

References

[1] Tse-Yu Yeh, Yale N. Patt - *A comparison of dynamic branch predictors that use two levels of branch history*, Proceedings of the 20th annual international symposium on Computer architecture, San Diego, California, United States

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

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

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

[5] Po-Ying Chang, Eric Hao, Yale N. Patt - *Alternative implementations of hybrid branch predictors*, Proceedings of the 28th annual international symposium on Microarchitecture, Ann Arbor, Michigan, United States

[6] Lucian N. Vintan, Marius Sbera, Ioan Z. Mihu, Adrian Florea – *An Alternative To Branch Prediction: Pre-Computed Branches*, ACM SIGARCH Computer Architecture News, June 2003.

[7] Brad Calder, Dirk Grunwald – *The Precomputed Branch Architecture*, UCSD Technical Report, March 1997.

[8] Gordon E. Moore - *Cramming more components onto integrated circuits*, Electronics (Volume 38 Number 8), April 1965.

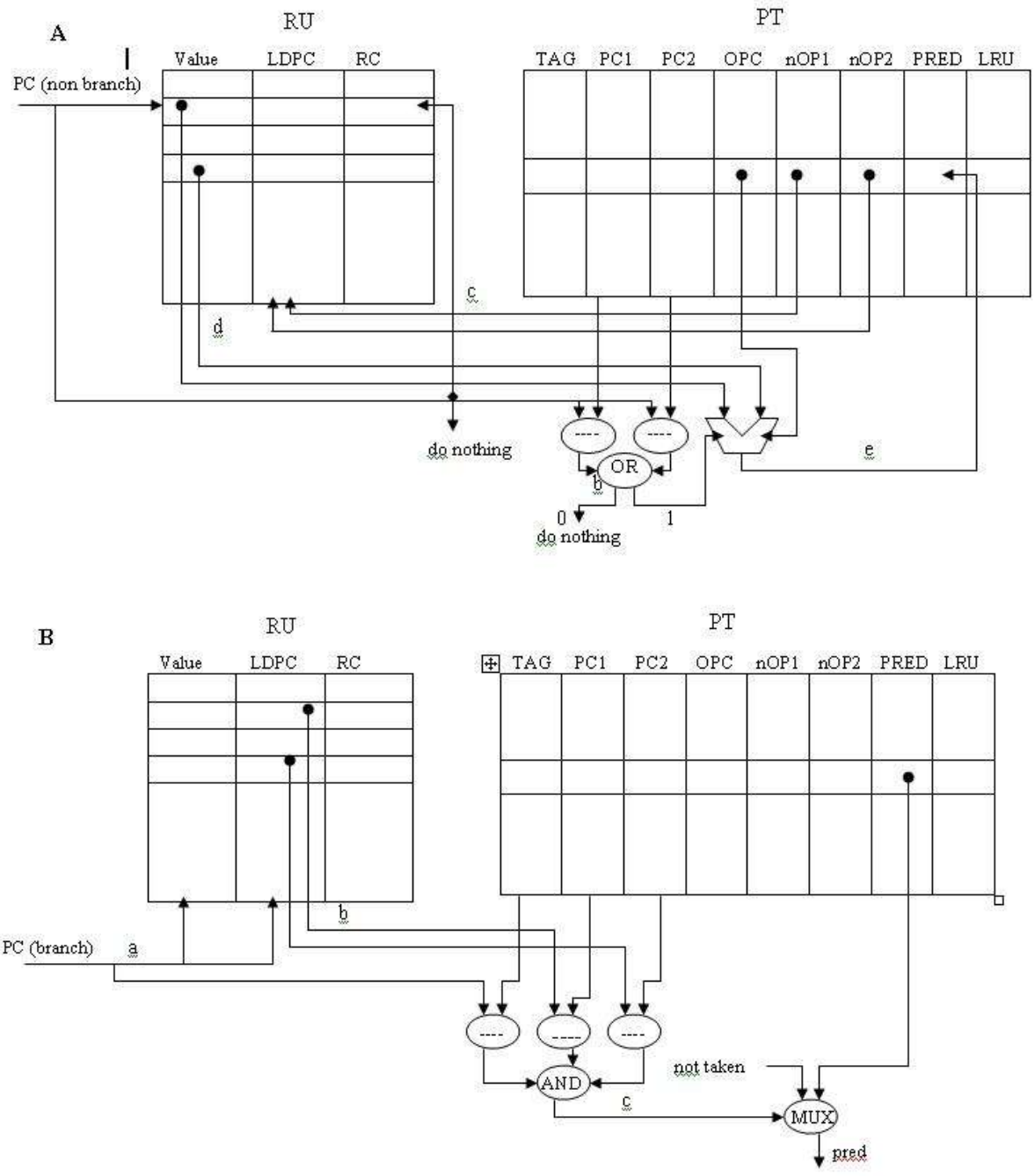

Figure 1: The new proposed prediction scheme.
A) when a non-branch instruction is encountered; **B)** when a branch instruction is encountered

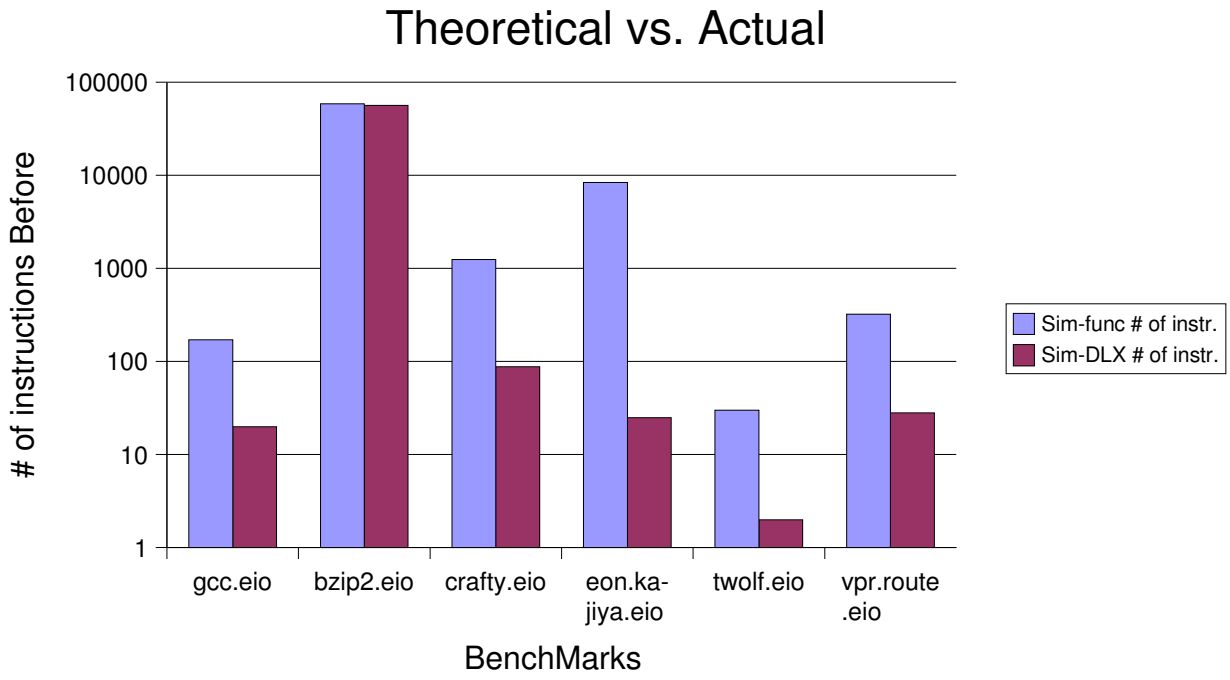

Figure 2

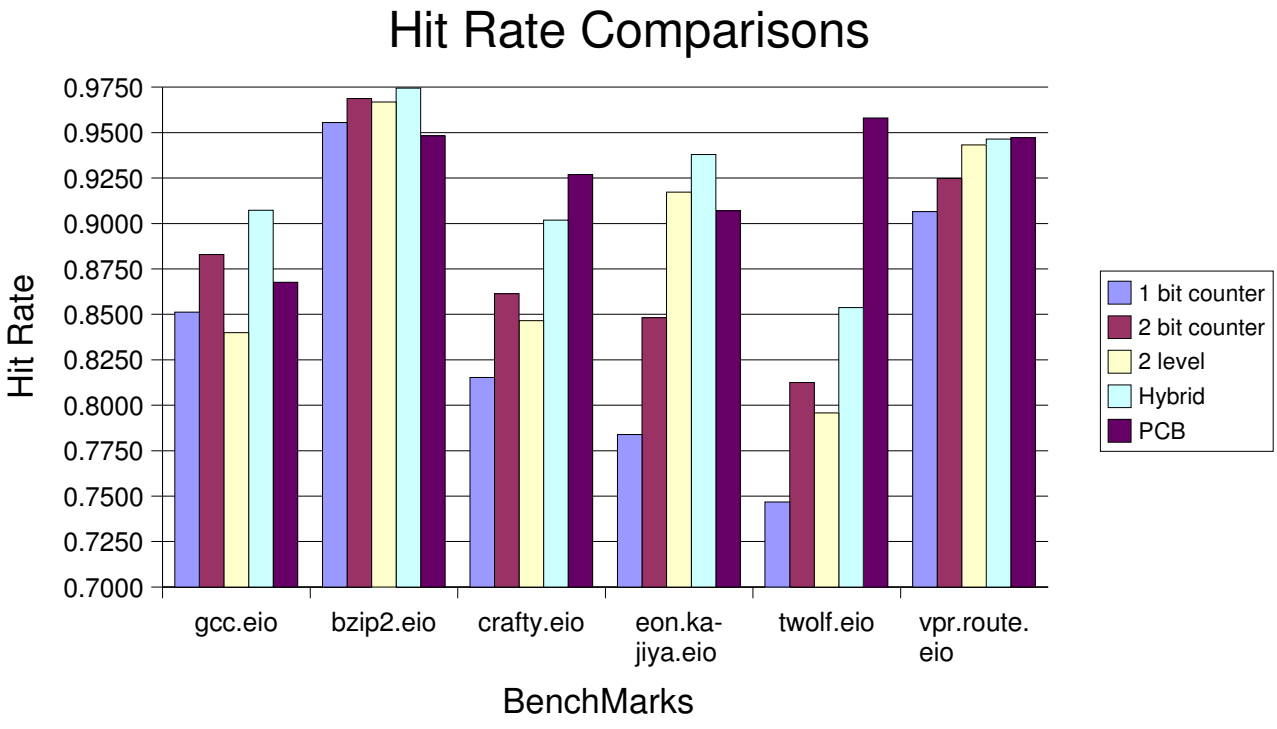

Figure 3

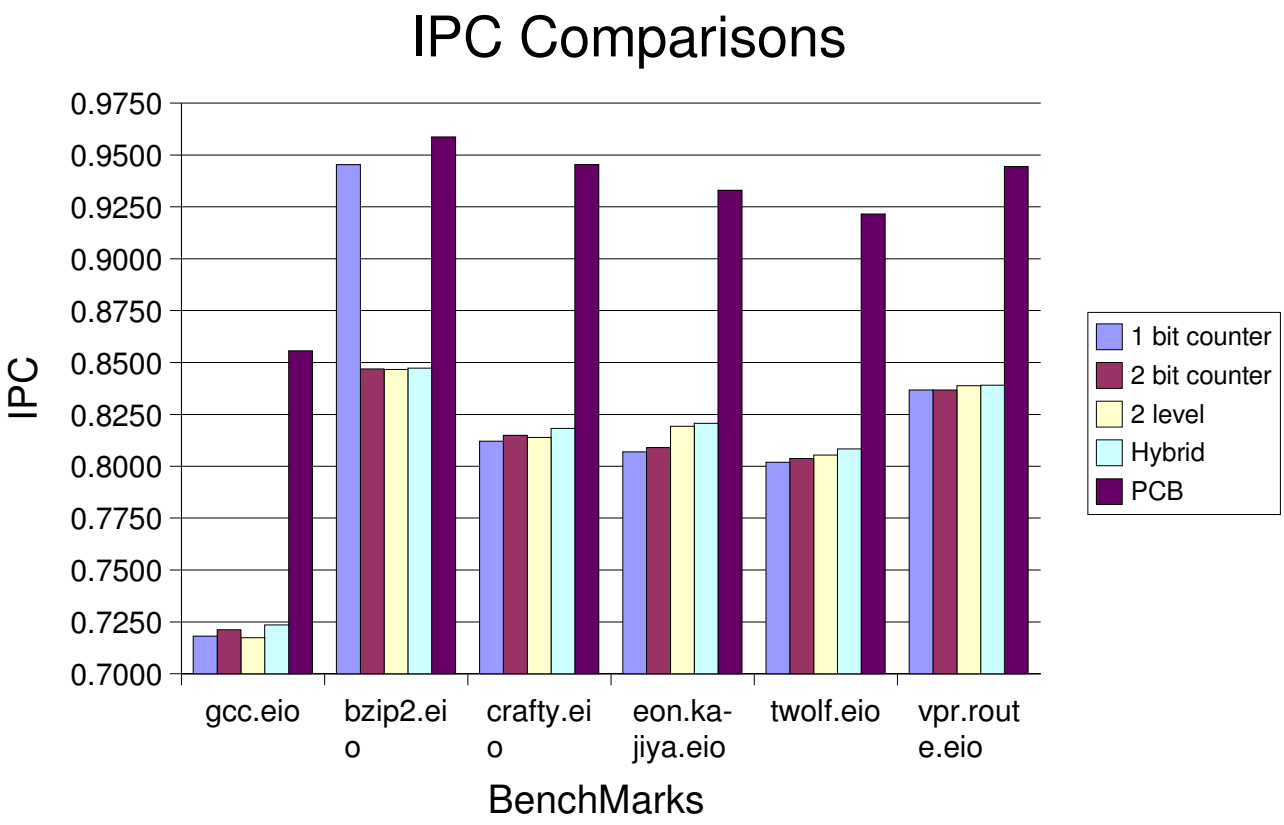
IPC Comparisons
IPC
0.9750
0.9500
0.9250
0.9000
0.8750
0.8500
0.8250
0.8000
0.7750
0.7500
0.7250
0.7000
gcc.eio
bzip2.eio
crafty.eio
eon.ka-jiya.eio
twolf.eio
vpr.route.eio
BenchMarks
1 bit counter
2 bit counter
2 level
Hybrid
PCB

Figure 4