BRANCH STRATEGIES: MODELING AND OPTIMIZATION

Pradeep K. Dubey
Michael J. Flynn

Technical Report: CSL-TR-90-411

FEBRUARY 1990

This work was supported in part by NASA under Contract NAGW-419 and used facilities supported under Contract NAG2-248.

Branch Strategies: Modeling and Optimization

by
Pradeep K. Dubey and Michael J. Flynn

Technical Report CSL-TR-90-411
February 1990

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford . California 94305-4055

Abstract

Instruction dependency introduced by conditional *branch* instructions, which is resolved only at run-time, can have severe performance impact on pipelined machines. A variety of strategies are in wide use to minimize this impact. Additional instruction-traffic generated by these branch-strategies can also have an adverse effect on the system performance. Therefore, in addition to the likely reduction a branch prediction strategy offers in average branch delay, resulting excess i-traffic can be an important parameter in evaluating its overall effectiveness. The Objective of this paper is two-fold : to develop a model for different approaches to the branch problem and to help select an optimal strategy after taking into account the additional i-traffic generated by the i-buffering.

The model presented provides a flexible tool for comparing different branch strategies in terms of the reduction it offers in average branch delay and also in terms of the associated cost of wasted instruction fetches. This additional criterion turns out to be a valuable consideration in choosing between two almost equally performing strategies. More importantly, it provides a better insight into the expected overall system performance. Simple compiler-support-based low implementation-cost strategies can be very effective under certain conditions. An active branch prediction scheme based on loop-buffer can be as competitive as a branch-target-buffer based strategy.

Key Words and Phrases: Branch prediction, conditional branches, degree of dependency, instruction dependency, instruction traffic, pipelining.

1 Introduction

This paper is an attempt to provide a common platform for modeling different schemes for reducing the branch-delay penalty in pipelined processors as well as evaluating the associated increased instruction bandwidth. The first section presents the details of the model. In the following section we summarize several existing branch-strategies and attempt to map them to our model. The next section derives certain inferences from the results obtained and leads us to some hybrid strategies.
Throughput in a pipeline environment is obtained by overlapping different instructions in different stages of execution. This implies predicting successive instructions before the completion of an instruction execution. *Instruction dependency,* i.e., the dependency of an instruction on the result of its predecessor limits this prediction ability. *Resource dependency* can be defined as the dependency arising out of system resources shared between independent instruction streams. Although normally these dependencies are mutually independent, sometimes they interact. As an example of possible interaction of this *instruction dependency* with the *resource dependency,* consider two pipelined processors in a shared-memory model. One possible approach to minimize performance degradation due to conditional branches (instruction dependency) would be to fetch both the streams, sequential as well as the branch-target instruction stream. As a direct side-effect of this, instruction traffic on the shared system bus goes up resulting in more conflicts. Thus, an attempt to reduce the impact of instruction dependency increases the performance impact due to resource dependency. As a result, overall system performance might even go down, instead of going up.

Previous Research

Tjaden and Flynn [8] provide an early framework in the area of formalizing the concept of instruction dependency. Impact of conditional branches on system performance was further substantiated by Riseman and Foster [5]. Interest in different branch-strategies for minimizing the performance impact has been renewed with the advent of new RISC machines. Most of the recent work in this area has mainly concentrated on specific branch-strategies and on improving prediction accuracy. Smith [6] discusses in

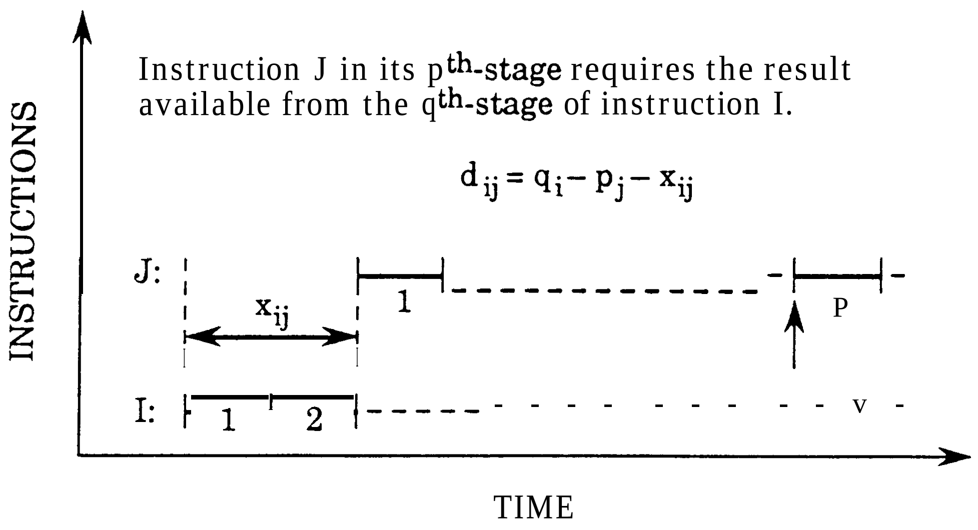

Instruction J in its pth-stage requires the result available from the qth-stage of instruction I.
$d_{ij} = q_i - p_j - x_{ij}$
INSTRUCTIONS
J:
x_{ij}
1
P
I:
1
2
v
TIME

Fig. 1: Instruction dependency in a pipeline.

detail different strategies for improving prediction accuracy. Lee/Smith [3] and McFarling/Hennessey [4] examine a range of schemes for reducing branch penalty. In comparing different branch strategies operational cost of increased instruction-traffic has so far been mostly ignored.

Consider a pipeline with S segments (Figure 1) executing an instruction *J*, which enters the pipeline the very next clock after instruction *I*. Assume a pipeline segment delay as equivalent to the system clock-period. Suppose the instruction *J* at the start of its p_jth stage of execution requires the result available at the completion of the q_ith stage of execution of instruction *I*. The *degree of dependency* in such a case is defined as $d_{ij} = (q_i - p_j)$, where we assume $q_i > p_j$. *I* and *J* above refer to successive instructions. Instead of entering the pipeline the very next clock after *I*, suppose *J* follows after an *additional* delay of x_{ij} clocks. The degree of dependency is reduced to

$$d_{ij} = (q_i - p_j - x_{ij}) \tag{1}$$

Any segment freeze possibility, i.e., the possibility that a data item may spend more than one clock in a certain pipeline segment, is ignored.

If $d_{ij} \leq 0$: I and J are considered to have *null* pipeline dependency, which means this dependency has no impact (null impact) on the pipeline throughput.

If $d_{ij} > 0$: I and J are considered to have *positive pipeline dependency,* which suggests this dependency has impact of d_{ij} clocks on the pipeline throughput. In other words, there is no pipeline output for d_{ij} clocks.

Degree of dependency is maximum when $p_j = p_{j(\min)} = 1$, $q_i = q_{i(\max)} = S$, and $x_{ij} = x_{ij(\min)} = 0$; i.e., when $d_{ij(\max)} = (S - 1)$ = maximum pipeline dependence.

We next consider instruction dependency due to *brunch* instructions. Let I represent a conditional branch instruction. In such a case, a following instruction J ($x = 0$) can not be fetched until the execution of I is complete. Assuming instruction fetch (IF)stage as the first stage of the pipeline (p_j = 1) and the execution (E) stage, which tests the condition-code, as the last pipeline-stage ($q_i = S$), this leads to maximum pipeline dependency of $(S - 1)$. Note that, although the condition code testing by the the branch instruction I can be typically done in a stage prior to the execution stage, normally it can only be done after the previous instruction $I - 1$ clears the execution stage and sets the condition-code. In other words, branch instructions can potentially result in the maximum possible slowdown of (S-1) clocks. In general, branch instructions may not need to wait until the last pipeline stage for their resolution, especially the unconditional branches. Let S_b refer to the pipeline stage that resolves the branch. In the case of unconditional branches, S_b would typically be the instruction-decode stage, whereas in the case of conditional branches it would refer to the execution stage of the pipeline that normally sets the condition-code, which may not be the last stage as is the case when the execution stage is followed by a write-back stage.

We have not considered so far any freeze situations. A pipeline stage is considered *frozen* if it is not ready to accept a new data item at the end of the current clock period. Such a situation typically results when some unexpected condition is encountered, such as a cache-miss on an instruction or an operand fetch, or a branch that unexpectedly disrupts the sequential instruction fetch. This freeze implies the addition of wait-cycles for the subsequent pipeline stages as they are forced to wait extra cycles for the frozen stage output. A successful *brunch* instruction involves the fetch and execution of an *out of sequence* instruction. Fetching the branch target instruction consists of (i) target address calculation and (ii) target fetch. Each of these steps has freeze-potential as compared to fetching a sequential instruction. In modeling the impact of branch instructions, other possible freeze conditions are deliberately ignored, including any instruction-fetch freeze in the sequential path.

Let

Y_a = average number of clocks spent during a target-address-calculation freeze

Y_f = average number of clocks spent in case of a page-fault during target-fetch

P_a = probability of a target-address-calculation freeze

P_f = probability of a target-fetch freeze

Therefore, the performance penalty k potentially associated with instructions I and J, where I is a branch instruction and J is dependent on I with a degree of dependence d_{ij}, can be written as:

$$k = d_{ij} + P_a * Y_a + P_f * Y_f \qquad (2)$$

1.1 Classification of Branch Strategies

The above equation (2) can be used for classifying different branch strategies on the basis of the approach taken to reduce the branch penalty, *k:*

a) *Reduce the dependency:* Branch strategies in this category are aimed at reducing the pipeline dependency, d_{ij}, between a branch instruction I and the following dependent instruction J. An attempt is made to make the positive pipeline dependency as close to the null pipeline dependency as possible. As expressed by equation (1), this attempt can be classified into the following sub-categories:

i) *Increase* p_j: A branch strategy which fetches both the sequential as well as the branch-target instruction without waiting for the branch outcome, increases p_j by removing the dependence of IF stage on the completion of execution stage and thereby reducing d_{ij}.

ii) Decrease q_i: Branch strategies may add hardware to evaluate the condition-codes for branches ahead of the execution stage, thereby reducing q_i for branch instructions to reduce the dependence, d_{ij}.

iii) *Increase* x_{ij}: Delayed-branch strategies fall into this category. Following a branch instruction, an attempt is made to insert instructions that need to be executed irrespective of the branch outcome. This reduces d_{ij} by increasing x_{ij}.

b) Other branch strategies are aimed at reducing the associated freeze-penalties:

i) *Reduce the address-calculation freeze:* Loop-buffers tend to reduce the freeze-penalty associated with target-fetch by capturing the loop environment.

ii) *Reduce the target-fetch* freeze: Branch-target buffers attempt to eliminate the target-address-calculation delay.

1.2 Branch Predict ion

Almost all the branch strategies do not absolutely eliminate branch delay but do so only with a certain probability. They make an implicit assumption about the most likely branch outcome and commit themselves to the sequential or the branch path to varying degrees. This commitment normally reduces the penalty associated with the chosen path but increases the penalty of taking the discarded path in case of incorrect prediction. As a result, overall performance improvement becomes critically dependent on the probability of correct prediction.

Let:

p_{t} = probability of *branch-to-be-taken* prediction
p_{c} = probability of *correct* prediction
$p_{\mathrm{n,n}}$ = probability of *predicted: no branch* and *actual: no branch*
$p_{\mathrm{n,b}}$ = probability of *predicted: no branch* and *actual: branch*
$p_{\mathrm{b,n}}$ = probability of *predicted: branch* and *actual: no branch*
$p_{\mathrm{b,b}}$ = probability of *predicted: branch* and *actual: branch*
$k_{\mathrm{n,n}}$ = performance penalty when *predicted: no branch* and *actual: no branch*
$k_{\mathrm{n,b}}$ = performance penalty when *predicted: no branch* and *actual: branch*
$k_{\mathrm{b,n}}$ = performance penalty when *predicted: branch* and *actual: no branch*
$k_{\mathrm{b,b}}$ = performance penalty when *predicted: branch* and *actual: branch*

Therefore:

$$\begin{aligned} p_{\mathrm{n,n}} &= (1-p_{\mathrm{t}}) * p_{\mathrm{c}} \\ p_{\mathrm{n,b}} &= (1-p_{\mathrm{t}}) * (1-p_{\mathrm{c}}) \\ p_{\mathrm{b,n}} &= p_{\mathrm{t}} * (1-p_{\mathrm{c}}) \\ p_{\mathrm{b,b}} &= p_{\mathrm{t}} * p_{\mathrm{c}} \end{aligned} \tag{3}$$

Consider a branch strategy which on an average predicts 6 out of every 10 branches as likely to be taken, and 2 out of every 10 predictions are incorrect. This results in $p_{\mathrm{t}} = 0.6$ and $p_{\mathrm{c}} = 0.8$, which leads to $p_{\mathrm{n,n}} = 0.32$, $p_{\mathrm{n,b}} = 0.08$, $p_{\mathrm{b,n}} = 0.12$, and $p_{\mathrm{b,b}} = 0.48$. This means, on an average out of every 100 branches, 32 are not taken as predicted, 8 are taken in spite of *not-to-be-taken* prediction, 12 are not taken though predicted as likely to be taken, and 48 are taken in accordance with the prediction. Therefore, 56 out of every 100 branches are taken. The number of branches actually taken, $(p_{\mathrm{n,b}} + p_{\mathrm{b,b}})$ is independent of the employed branch strategy.
The average branch penalty (in terms of additional cycles per branch) is given by:

$$K = p_{\mathrm{n,n}} * k_{\mathrm{n,n}} + p_{\mathrm{n,b}} * k_{\mathrm{n,b}} + p_{\mathrm{b,n}} * k_{\mathrm{b,n}} + p_{\mathrm{b,b}} * k_{\mathrm{b,b}} \tag{4}$$

Let the *branch frequency* in terms of number of branch instructions (conditional and unconditional) per instruction, be *b*. Branch performance throughput, G, can be expressed as:

$$G = 1/(1 + K * b) \tag{5}$$

Thus, for $K = 0$ or $b = 0$, performance throughput is assumed to be at its peak rate of 1 instruction per cycle. In other words, all the other pipeline overheads [1] are being ignored.

1.3 Cost of Branch Prediction

The discussion above has centered around assessing the performance of different branch strategies. Consider the two primary costs involved: (i) implementation cost and (ii) operational cost. *Implementation cost (H)* refers to the hardware/software costs involved in implementing the branch strategy. For example, *delayed-branch* strategy typically requires modification in the user program at compile-time and hence adds to the software cost. Since such costs are harder to quantify, this cost is ignored in the discussion to follow. On the other hand, the relatively less obvious *operational cost* refers to the adverse impact of implementing the strategy perceived at run-time; for example, the additional instruction traffic that results on the system bus with every incorrect branch-prediction. Although incorrect predictions are the primary source of extra i-traffic, even delayed correct prediction can cause wasted instruction fetch. For example, any sequential instruction fetch after the fetch of a branch instruction but before its decode would be a waste, even if the branch is successfully predicted. For architectures that allow machine-state update by instructions in the predicted path, there is an additional run-time overhead of *shadowing* the original machine-state to be able to recover in case of an incorrect prediction. For the sake of simplicity, this cost is not included in our calculations and we do not expect it to alter our conclusions about the relative merits of different strategies. Thus, the only operational cost emphasized in this study is that of the additional i-traffic.

Interestingly, freeze conditions, which tend to increase the branch delay, reduce the average additional i-traffic. When a certain path is predicted, freeze situations reduce the number of instructions that can be fetched, which reduces the number of wasted i-fetches in the case of incorrect prediction. Consider, for example, a branch-instruction fetch and decode followed by branch-target fetch, assuming a successful branch. Thus, target fetch starts after a delay of 1 clock (branch decode) and as a result, a maximum of $(S_b - 2)$ instructions from the branch-path can be in the pipeline

when branch instruction completes execution. If the branch turns out to be unsuccessful, these $(S_b - 2)$ fetches are a waste. In reality, this number is reduced due to possible freeze situations mentioned above and can be calculated as shown in the appendix.
Let:

$t_{n,n}$ = additional i-traffic when *predicted: no branch* and *actual: no branch*
$t_{n,b}$ = additional i-traffic when *predicted: no branch* and *actual: branch*
$t_{b,n}$ = additional i-traffic when *predicted: branch* and *actual: no branch*
$t_{b,b}$ = additional i-traffic when *predicted: branch* and *actual: branch*

The average cost of additional i-traffic (in terms of wasted i-traffic per branch instruction) is:

$$T = p_{n,n} * t_{n,n} + p_{n,b} * t_{n,b} + p_{b,n} * t_{b,n} + p_{b,b} * t_{b,b} \qquad (6)$$

The total cost, C (in terms of wasted i-traffic per instruction), can be expressed as:

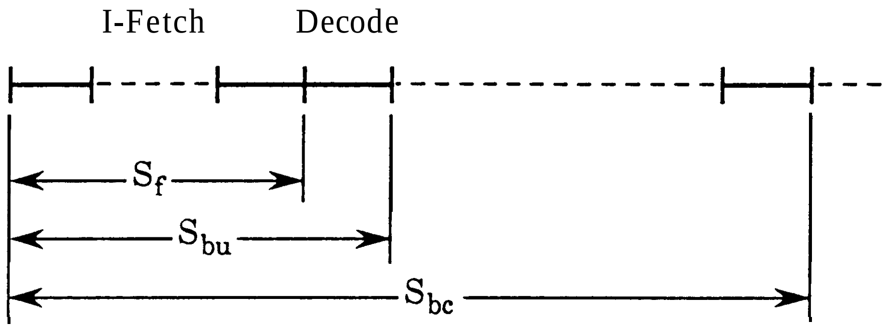
$$C = (1 + T * b) \qquad (7)$$

Combining equations (5) and (7), the cost-performance parameter MR (referred to as the *Merit Ratio* in the discussion below) can be expressed as:

$$MR = G/C = 1/((1 + T * b) * (1 + K * b)) \qquad (8)$$

Thus, for an ideal machine which can always correctly predict the branch outcome and, if needed, can start fetching the target path right after the branch-instruction fetch would have, $K = T = 0$ and hence, G = C = 1, resulting in unit merit ratio, MR, irrespective of the branch frequency, *b*.
The following simplifying assumptions (some of which have been referred to earlier) should be kept in mind (Figure 2):

a) Instruction fetch stage is assumed to consist of S_f slots (each containing a prefetched instruction) followed by the decode stage. No assumption is made about the remaining stages.



S_f: number of sub-stages in the fetch-stage
S_{bu}: pipeline stage that resolves unconditional branches
S_{bc} : pipeline stage that resolves conditional branches

Fig. 2: An instruction pipeline.

b) As mentioned before, S_b refers to the pipeline length only up to the stage that resolves the pending branch instruction. In the case of unconditional branches, we assume branches are resolved as soon as they are decoded, therefore, $S_b = S_{bu} = S_f + 1$; whereas in the case of conditional branches, $S_b = S_{bc}$ is dependent on the pipeline stage that sets the condition-code.

c) The operational cost of additional traffic refers only to the i-traffic and ignores any wasted data traffic.

d) Additional i-traffic during freeze-handling, e.g., in page-fault handling, is ignored.

e) Finally, for the sake of simplicity, the handling of multiple pending branches in the pipeline is restricted. If a branch is predicted as likely to be taken, it is assumed that we do not encounter additional branches with *to- be-taken* prediction before resolving the first branch. This assumption can be a source of some significant inaccuracy only for very long pipelines with prediction schemes which allow this possibility.

The following section describes different branch strategies. A typical system may use a combination of following strategies, in which case effective branch delay, K, and excess i-traffic, T, can be obtained by adding the components due to each strategy (weighted by the application probability of the strategy).

2 Branch Strategies

Some common strategies employed to reduce the delay caused by branches include the following. Different parameters described above involving performance penalty and increased i-traffic are calculated for each strategy in the appendix.

Predict Branch Never Taken *(PBNT)* This represents a simple strategy for branch-handling. Instruction-fetch and decode continue uninterrupted from the sequential path until the branch gets executed. Thus, implicitly a prediction is always made at the fetch of a branch-instruction that it will not be taken. If this turns out not to be the case, the $(S_b - 1)$ instructions fetched in the preceding stages of the pipeline are discarded and we start fetching from the target path.

Loop Buffers *(LB)* A *loop buffer* refers to a high-speed memory in the IF stage of the processor. The size of the loop buffer is a critical parameter and is normally not very large. Some CDC machines (6600, 7600 and Star-100) as well as CRAY-1 have used this idea. These buffers (Figure 3) can detect if the branch target (forward or backward) lies within the environment captured by the buffer and if so, the delays associated with both the IF stage memory-fetch as well as the possible freeze are eliminated. Since a hit in the loop-buffers avoids any external memory-access, it also reduces the average extra i-traffic in case of incorrect prediction. The size of a loop as well as the

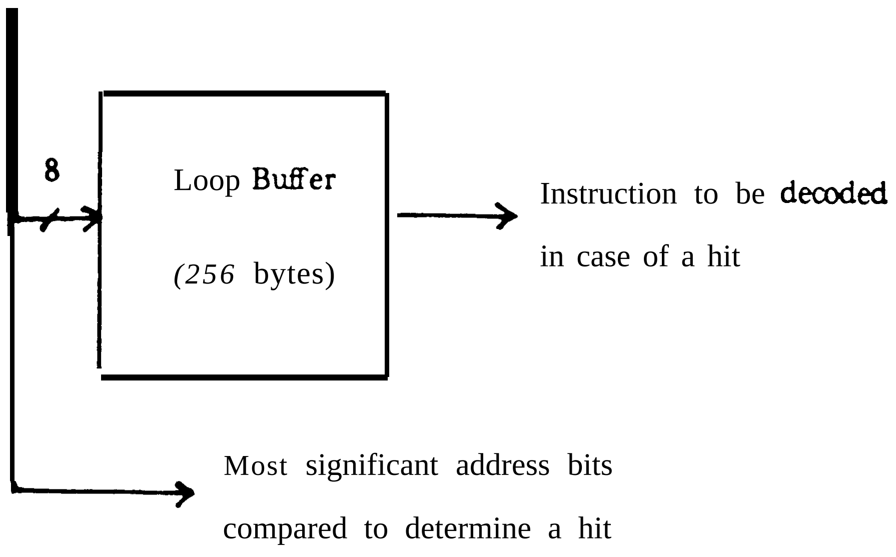

8
Loop Buffer
(256 bytes)
Instruction to be decoded
in case of a hit
Most significant address bits
compared to determine a hit

Fig. 3 Loop Buffer

average repetitiveness of a loop (average loop-count) determine the *hit* probability for a loop-buffer. High loop-count tends to minimize the impact of first-time misses and hence increases the hit-probability. Once a loop is captured in the loop buffer, branch target fetches are similar to sequential fetches (other than the target-address calculation). Although loop-buffers may appear to be similar to i-caches, they are relatively much smaller in size and lower in implementation cost. Branches are again always predicted as not likely to be taken, similar to the previous case.

Pre-calculate Target Address *(PTA)* This strategy pre-calculates branch-target address irrespective of the branch outcome. Thus, address calculation is assumed to start in parallel with the pipeline soon after the branch-fetch and decode. This significantly reduces any address calculation delay. In other respects, this strategy is similar to the *PBNT* (predict-branch-never-taken) scheme and branch is still always considered as less likely to be taken.

Fetch Target in OF-slot: ($FTOF$) Very often branch-instruction execution does not require any operand-fetch. Some IBM machines (370 series) use the operand-fetch (OF) slot of the pipeline for fetching the target. Thus, the branch can still be assumed as less likely to be taken, thereby choosing to fetch instructions from the in-line path, except in the OF-slot when one attempt is made at fetching from the branch target-path, assuming the target-address is available. In case the branch is unsuccessful, the target fetched in the OF-slot is wasted.

Predict Branch Always Taken ($PBAT$) In contrast to the $PBNT$ scheme, this strategy assumes that any branch is always more likely to be taken and hence, whenever a branch is decoded, instruction fetch switches to the branch-path. Sequential path fetch can be stopped and switch to the branch path can be made only after the decode of the branch instruction. As a result, one or more ($S_{\mathbf{f}}$) sequential instructions are fetched in any case. These instructions would be considered useful if the branch is finally not taken, and wasted if the branch is taken.

Predict Branch Always Taken with Target Copy ($PTTC$) This scheme is an effort to further reduce the branch penalty associated with the predict-branch-always-taken strategy, and modifies the instruction sequence at compile-time. A portion of target code (as dictated by the effective pipeline length for branch resolutions) is copied following the branch instruction (Figure 4). As a result, if the branch is actually taken all the pending instructions in the pipeline fetched sequentially are valid. In contrast to the previous strategy, target-fetch does not have to freeze for target-address calculation. Note that target-fetch freeze (due to page-faults, etc.) would still be encountered at a later point when the target fetch begins from the calculated target-address beyond the copied portion. In case of unsuccessful branch, all the pending target instructions fetched sequentially are wasted.

```
        CMP   R1, R2
        JZ    xx
       *ADD   R3, R4
       *SUB   R3, R5
       *INC   R4
       *ADD   R3, R4
        MOV   R6, R7
        ADD   R6, R2
        MOV   R1, mem
        ..    ..
        ..    ..

xx:     ADD   R3, R4
        SUB   R3, R5
        INC   R4
        ADD   R3, R4
xx+4:   MOV   R6, R3
        ..    ..
        ..    ..
```

Instructions marked with an asterisk (*) are the instructions copied from the target (xx) at compile-time. In case of a successful branch, control transfers to the label xx+4, after executing the marked (*) target instructions through sequential fetch. In case the branch is not taken, marked instructions are discarded without execution after fetch and decode.

Fig. 4 Predict Branch Always Taken with Target Copy ($PTTC$)

Fetch Both the Paths (*FBP*) Versions of this strategy are employed in some IBM machines (370/168, 3033). It uses the brute-force approach of fetching both the sequential and non-sequential instruction streams in case a branch is decoded. Again the branch-path fetch cannot begin until the branch instruction has been decoded and the target address formed. This is equivalent to predicting that the branch would either be taken or not taken. Since such a prediction is t autologically correct, $p_c = 1$. Therefore, in this case p_t should be interpreted as simply the probability that the branch would be taken (not as branch taken probability as per prediction).

Delayed Branch *(DB)* As mentioned earlier, this strategy aims at reducing the pipeline dependency, d, for branch instructions by increasing the delay, x, bet ween the branch instruction and its first dependent instruction. Instructions (if any) that are common to both the sequential and the non-sequential branch path are inserted between the branch instruction and its first dependent instruction. Let u represent the average number of such useful (non-NOP) common instructions. We assume that branch is always predicted as not likely to be taken.

Taken/Not-taken Switch in the Decode Stage *(TNTD)* This strategy assumes a branch *taken/not-taken* switch in the instruction decode stage. When a branch instruction is decoded a prediction is made as to whether the branch will be taken or not. This prediction information can be obtained and improved for better accuracy in many different ways [6]. In its simplest form, the prediction may be static and explicitly enoded as part of the specific branch opcode at compile-time based on some data collected regarding the past behavior of the branch. For example, backward branches are considered more likely to be taken than not due to their common association with loops. On the other hand, prediction strategy may be dynamic, evolving towards better accuracy. Still, as before, target-fetch can not begin until branch-fetch and decode.

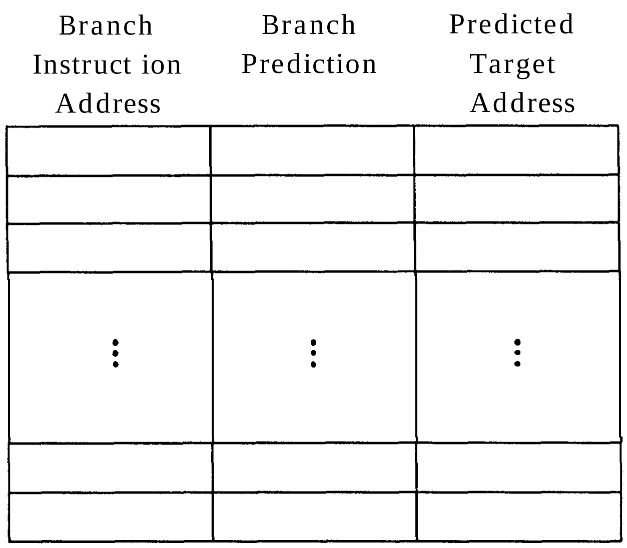


Fig. 5 Branch Target Buffer

Branch Target Buffer *(BTB)* Target *buffer* refers to a small associative memory in the IF stage of the processor [3]. Instruction-fetch addresses are associatively matched with the buffer contents and in case of a *hit* it predicts the most likely branch outcome as well as the most recent target address (Figure 5). As a result, target-fetch does not need to wait for the branch-decode and target-address calculation. If the branch is likely to be taken, first target-instruction fetch immediately follows the branch-instruction fetch. Following branch-decode, at the completion of actual target-address calculation, a comparison is made with the predicted target-address. A mismatch here flushes any fetches made from the incorrect target, aborts any freeze in the incorrect target-path and restarts target-fetch at the calculated address. We also assume that this comparison output is available along with the actual target, without any additional clock-overhead. Correct target prediction probability, P_{ct}, depends on the frequency of branch target changes. A *hit* in the *branch target buffer* (BTB) implicitly also assumes that the fetch-address contains a branch instruction.

In case of writable code segments, there is a (small) likelihood, P_w, that a non-branch instruction gets predicted as a branch instruction. To make things worse, if such an instruction is predicted as a branch likely to be taken then it has an impact on the system throughput even in the absence of any branch-instruction, as it blocks the sequential address fetch during the following cycle until the actual instruction-decode. This throughput deterioration is modeled using the following modified version of equation (5):

$$G = 1/(1 + K * b + S_f * P_w * P_t) \qquad (9)$$

Similarly, this probability of a BTB-hit with *branch-to-be-taken* prediction for a non-branch instruction also modifies equation (6), which so far included additional i-traffic only due to branch-instructions. The following equation reflects an additional wasted instruction-fetch in case a non-branch instruction is predicted as *to be taken branch* and there is no target-fetch freeze:

$$T = T_b * (1 - P_w) + (S_f \sim Y_f * P_f) * P_t * P_w \qquad (10)$$

where T_b refers to the excess i-traffic due to branch-instructions given by equation (6) and $\sim$ refers to the probability-based reduction explained in the appendix.

In case of a miss in BTB, branch instructions are handled in a manner similar to ***PBNT*** strategy. In other words, branch is assumed as not likely to be taken by default in case of a BTB-miss. The overhead involved in BTB-updates is ignored. Therefore, this strategy can be equivalently considered as a combination of two strategies, one as described above with BTB-hit probability P_{th} and the other the same as in the case of ***PBNT*** scheme with BTB-miss probability.

Calculation for different excess i-traffic parameters gets somewhat involved in this case and hence their derivation is described below in qualitative terms. Details of the calculation can be found in the appendix along with other parameters.

i) If a branch is predicted as likely to be taken, target-fetch begins immediately from the predicted address, and target-address calculation starts soon after the decode. If the calculated target-address mismatches with the predicted address, instructions fetched so far from the incorrect address are wasted and instruction fetches begin from the calculated (actual) target address.

ii) In the previous case, if the branch is not taken then in addition to the fetches made from the predicted target, instructions fetched from the actual target address are also wasted.

iii) If branch is predicted as not likely to be taken, we assume no attempt is made to calculate the target-address and instruction-fetch cont inues from the sequential path. If this prediction turns out to be false, sequential instructions fetched are discarded, target-fetch immediately begins at the predicted address and the actual target calculation starts simultaneously. If the calculated target mismatches with the predicted target, this fetched sequence is also wasted.

Finally, as mentioned before, the above set of parameters is used to calculate the average branch delay, K_{h}, and excess i-traffic, T_{h}, where the subscript refers to the BTB-hit case. In the case of BTB-miss, the corresponding parameters K_{m} and T_{m} are calculated from the components given for *PBNT* case. Combining these cases, we get:

$$K = K_{\mathrm{h}} * P_{\mathrm{th}} + K_{\mathrm{m}} * (1 - P_{\mathrm{th}}) \tag{11}$$

$$T_{\mathrm{b}} = T_{\mathrm{h}} * P_{\mathrm{th}} + T_{\mathrm{m}} * (1 - P_{\mathrm{th}}) \tag{12}$$

3 Results

This section deals with our model environment and some results. The model described above can be used to obtain the average branch delay (K), the average number of wasted i-fetches per branch *(T),* and the overall merit ratio *(MR)* once the variables outlining the system-environment are defined. For the purposes of our model, we assume certain nominal values for some of these variables. We assume a nominal operating environment with branch frequency b = 0.25, where 80% of the branches are conditional. Probability of a freeze during target address calculation is assumed to be 0.5 with a freeze duration of 2 cycles. Thus, on average, we freeze for a clock for target address calculation. Probability of freeze during target-fetch is ignored. For delayed-branch approach, an average of one useful common instruction (i.e., u = 1) is assumed. One such machine employing delayed branch approach is MIPS [4], which could use only a single delay slot about 70% of the time. There may be special cases where a significant number of delayed branch slots may be utilized. For example, Hsu/Davidson [11] suggest a scheme whereby on machines such as CRAY-1, where conditional branch resolution may take 14 clocks, a large number of these time-slots may be filled with *guarded* instructions. These instructions are considered 'guarded' because, in case the branch resolution is not as expected, they are effectively turned into NOPs. Only the last two strategies described in the previous section provide support for active branch prediction; the rest rely on their respective default prediction of *branch always taken* or *branch never taken*. Based on Smith [6], we assume a correct prediction probability of 0.85 for conditional branches, resulting in an overall correct prediction probability around 0.9, assuming a unconditional branches are always predicted correctly. In the case of Branch Target Buffer, probability of correct target address prediction is optimistically set at 0.9, assuming stable branch targets [6]. Probability of BTB-hit for non branch instructions in case of writable code segments is assumed very low at 0.05. We assume nominal P_{lh} = 0.6 and nominal P_{th} = 0.8. Peuto/Shustek [9] report a hit ratio of 0.6 for a loop buffer of ±256 entries, whereas Lee/Smith [6] report a hit ratio of around 0.8 for a target buffer with 256 entries and a set size of 4 or 8. *Set-size* refers to the degree of associativity in contrast to fully-associative BTB search.

We concentrate initially on three system parameters in our discussion below: successful branch probability (conditional and unconditional combined) P_{sb}, varying the number of sub-stages in the instruction-fetch stage (S_f) followed by a fixed number of remaining pipeline-stages, and finally we vary the total number of pipeline stages required for conditional branch resolution (S_{bc}) assuming a fixed number of IF sub-stages. Results are obtained for the three performance parameters, average branch delay (K), average number of wasted i-fetches (T), and the cost-performance merit-ratio (MR), as a function of these three system parameters. While one of these system parameter is varied, others are kept at their nominal values. We assume nominal $P_{sb} = 0.6$, nominal $S_{bc} = 5$, and nominal $S_f = 1$. It should be noted that nominal value of 0.6 for P_{sb}, which represents successful branch probability of conditional and unconditional branches combined, implies that conditional branches are assumed equally likely to be taken or not taken.

3.1 Inferences

The following inferences can be made regarding the three performance parameters as a function of the successful branch probability (Figs. 6–11, Appendix):

a) As one might expect, BTB-strategy outperforms others for the entire typical operating range ($0.55 < P_{sb} < 0.7$).

b) Predict-branch-always-taken scheme with target copy ($PTTC$) emerges as a good second choice around P_{sb} of 0.65 or more. Interestingly, even without any active-branch-prediction support, it exhibits better performance potential than BTB around $P_{sb} \geq 0.75$. This advantage stems primarily from the fact that this scheme does not have to pay the delay penalty of incorrect target address prediction. BTB-strategy has a cost of incorrect target address prediction even with correct branch prediction. On the other hand, as a cautionary note, the $PTTC$ scheme also exhibits the steepest slope in terms of all three performance parameters as opposed to the relatively stable performance curves of active-prediction schemes (BTB-strategy and the taken/not-taken switch strategy).

c) In terms of excess i-fetches, the loop-buffer scheme (LB) performs almost as well as the best performing BTB-strategy. As explained earlier, loop-buffers can significantly cut down the cost of excess i-traffic resulting from incorrect predictions.

d) At nominal P_{sb} (0.6), both strategies *PBNT* and *PBAT* have the same branch delay. Which of the two should be the preferred scheme? A look at the additional i-traffic cost can help resolve the issue. The *PBAT* scheme has lower cost of wasted i-fetches and hence has better merit ratio (MR). In the absence of any possible address-calculation-freeze (or target-fetch-freeze) , on average the *PBNT* scheme wastes more instructions during misprediction than *PBAT* for any S_{bc}. A similar dilemma between *PTTC* strategy and strategy *DB* can be resolved in favor of the *DB* scheme, due to its lower added i-traffic cost. Also note that in both the scenarios just described, implementation costs are almost identical for the two strategies in question, hence operational cost in terms of excess i-traffic *T* provides an important decisive input. Interestingly, at P_{sb} = 0.5, three different strategies: predict branch never-taken, target-fetch in the OF-slot, as well as the scheme to fetch both the paths, show the identical merit-ratio. Implementation-cost can probably be the only decisive input in such a scenario.

e) Not only does excess i-traffic cost, *T*, help us choose between two almost equally performing strategies (as shown above), it can also caution us about otherwise very well performing strategies. The *FBP* strategy (fetch both the paths) provides an interesting study in this regard. In terms of average branch delay (K), it performs almost as well as the best performing *BTB* strategy. But after considering the cost of wasted i-fetches, in terms of the overall merit ratio (MR), it is not much better than the worst-performing *PBNT* strategy. Thus, a conclusion based solely on average branch delay K may be an elusive one as far as the overall system-performance is considered. Garcia/Huynh [2] discuss the efforts made to reduce the resulting high contention on the system bus in an early IBM 370 implementation using this strategy.

The following conclusions can be drawn regarding the three performance parameters as a function of S_f, i.e., number of buffer stages in the IF-stage (Figs. 12–14):

a) Again BTB outperforms every other strategy in terms of average branch delay (K) for any amount of buffering in the IF-stage.

b) Similar to our earlier observation, the $PTTC$ scheme stays as the second-best strategy in terms of average branch-delay.

c) Although the $TNTD$ scheme provides lower branch-delay than the loop-buffer scheme, due to its significantly lower cost of wasted i-fetches, loop-buffer comes out as the second choice in terms of the merit-ratio (MR).

d) Finally, all the strategies show an almost identical performance-slope on the merit-ratio curve. In other words, all the strategies show identical sensitivity with respect to S_f.

As a function of the total number of pipeline segments, we have the following inferences (Figs. 15-17):

a) BTB-strategy continues to be the first choice for any number of segments in terms of overall merit-ratio. But for long pipelines ($S_{bc} > 6$) it slips, and instead the fetch-both-paths scheme *(FBP)* wins with its constant branch delay with respect to S_{bc}.

b) For small number of segments (< 5) there is no obvious second choice in terms of the average branch delay, whereas for large number of segments (> 5), active-prediction schemes clearly start outperforming others (except $PTTC$). This emphasizes the need for active branch prediction for long pipelines.

c) It is interesting to note that just a branch taken/not-taken switch in the decode stage (*TNTD* scheme) significantly reduces the branch delay. In other words, the incremental reduction in branch delay obtainable through BTB rapidly goes down with larger S_{bc}.

Therefore, in the typical operating range ($0.6 < P_{sb} < 0.75$), we have three competing strategies: Loop-Buffer *(LB),* Predict Branch Always Taken with Target Copy (*PTTC*) and Branch Target Buffer *(BTB).* It is also interesting to note that our branch-delay numbers for *PBNT, DB* and *BTB* strategies under nominal conditions come quite close (within 30%) to those reported by McFarling/Hennessey [4]. Note that in addition to a difference in technique (analytical vs. simulation), our nominal conditions, although close, are not exactly the same as theirs. In the following section we discuss some hybrid strategies based primarily on these three strategies. Delayed branch *(DB)* strategy and *TNTD* strategy also show good performance potential in possible combinations with the above strategies.

4 Hybrid Strategies

The following hybrid strategies are considered:

a) Predict Branch Always taken with target-copy and delayed branch *(TTCDB):* This strategy is a combination of *PTTC* (predict branch always taken with target-copy) and *DB* (delayed branch) schemes. This is the only hybrid strategy considered with almost no additional implementation cost and only some software (compiler) cost.

b) Predict Branch Always taken with target-copy, delayed branch and Loop buffer *(TTDLB):* This strategy adds loop buffer (as described in *LB* scheme) with the previous strategy *(TTCDB).*

c) Taken/Not-taken Switch in the Decode Stage with Loop buffer *(TNTLB):* This strategy represents a combination of the *LB* and *TNTD* schemes given above. At the start of branch-instruction decode, the following sequential instruction is checked for a hit in the loop-buffer. In case of a miss, it is fetched from outside with regular memory-access delay associated with the fetch-stage. If the branch is predicted as likely to be taken, right after the target address calculation, loop-buffer access is attempted. In case of a hit, again we save the usual delay associated with the out-of-sequence fetch.

d) Taken/Not-taken Switch in the Decode Stage with Branch target buffer *(TNBTB):* Finally, we consider a combination of *TNTD* and *BTB* strategies above. In case of a miss in the BTB, instead of falling back on the default *PBNT* (predict-branch-never-taken) case, this strategy assumes a branch taken/not-taken switch in the decode stage similar to *TNTD* scheme. Model parameters in this case would be same as in the case of *BTB,* except that average branch-delay and excess i-traffic parameters K_m and T_m in case of BTB-miss are calculated using *TNTD* case instead.

4.1 Inferences

Sensitivity-plots of the performance parameters K, *T,* and MR are obtained with respect to the same three system parameters, P_{sb}, S_f, and S_{bc} (Figs. 18-26). The following observations can be made:

a) Around the nominal values of system-parameters, out of the hybrid-strategies, the minimum implementation cost *TTCDB* strategy performs better than every other non-hybrid strategy of the previous section except *B TB*. For P_{sb} around 0.7, it even outperforms *BTB* strategy in terms of average branch delay (K) as well as merit-ratio *(MR).*

b) Around our nominal conditions, the last three hybrid strategies: *TTDLB, TNTLB,* and *TNB TB* are almost equally competitive. For shorter pipelines ($S_{bc} < 5$), *T TDLB* has a slight edge over the other two, whereas for longer pipelines, active branch prediction becomes more important and *TNTLB* and *TNBTB* schemes perform better than the rest and continue to follow each other closely. Therefore, on a system with a branch-taken/not-taken prediction switch in the ID stage, if one were to choose between the addition of either the loop-buffer or the branch target buffer, careful consideration should be given to implementation-cost issues which may tilt the balance slightly in favor of the loop-buffer based *TNTLB* scheme.

c) Around P_{sb} = 0.7 or more, at nominal S_{bc}, *TTDLB* strategy starts outperforming the others and emerges as the first choice in terms of all the three performance parameters. Around $P_{sb} =$ *0.7, TTDLB* reduces the branch delay to less than one-third as compared to the 'don't do anything'-type *PBNT* strategy.

d) Similar to the non-hybrid strategies in last section, all the hybrid strategies show identical sensitivity with respect to the number of sub-stages in the IF-stage, S_f.

Effect of Loop/Target-buffer hit probability: Target-buffer based strategies show more sensitivity to the hit-ratio, P_{th}, than the loop-buffer based strategies in terms of average branch delay (Figure 27). On the other hand, loop-buffer based strategies are more sensitive in terms of the excess i-traffic cost with respect to the corresponding hit-ratio P_{lh} (Figure 28). As a result, both classes of strategies exhibit almost identical slope on the merit-ratio performance curve (Figure 29).

Effect of Target-fetch freeze probability: In our discussion so far we have ignored any potential for a freeze (due to, say, page-fault) while attempting to fetch the branch-target. Assuming a fetch-freeze duration (Y_f) of 10 clocks, we take a look at the performance sensitivity with respect to the fetch-freeze probability, P_f. Loop-buffer based strategies are at an advantage in such a case because a hit in the loop-buffer also eliminates any page-fault type potential associated with external memory access. As a result, loop-buffer based strategies show more performance stability with respect to P_f than the BTB-based strategies. For example, if P_f increases from 0 to 0.1, the average branch delay for loop-buffer based *TNTLB* strategy increases by 20%, whereas that in the case of BTB-based *TNBTB* strategy increases by more than 75% (Figs. 30-32).

5 Conclusions

This paper studies the effectiveness of different branch-strategies in terms not only of the reduction in branch delay but also the associated cost of excess i-fetches. The model presented here for comparing different strategies is flexible, enabling sensitivity analysis to estimate the incremental performance impact. We have ignored any on-chip cache in our discussion in order to highlight the increase in i-traffic due to mispredictions. This informat ion about additional i-traffic helps to bring out the overall performance difference between some apparently equally well performing strategies if one were to only consider the improvement in average branch delay. All the branch strategies analyzed show almost identical sensitivity with respect to buffering in the instruction-fetch stage. A branch strategy using a branch-taken/not-taken switch in the decode stage is found to be almost as effective in combination with the loop-buffer as with the branch target buffer. In a typical microprocessor environment with less than 5 segments with a successful branch probability around 0.6, a branch strategy based on default prediction of branch always taken, along with compiler support for target-copy and delayed branch, is shown to provide performance potential comparable to a branch strategy based on Branch Target Buffer.

6 References

[1] P. Dubey and M. J. Flynn, “Optimal Pipelining,” *Journal of Parallel and Distributed Computing,* pp. 10-19, Jan., 1990.

[2] L. C. Garcia and T. Huynh, “Storage Fetch Contention Reduction by Using Instruction Branch Prediction,” *IBM Technical Disclosure Bulletin,* Vol. 23, No. 6, 1980.

[3] J. K. Lee and A. J. Smith, “Branch Prediction Strategies and Branch Target Buffer Design,” *Computer,* Vol. 17, No. 1, January, 1984.

[4] S. McFarling and J. Hennessey, “Reducing the Cost of Branches,” *Proc. 13th Symp. on Computer Architecture,* 1986.

[5] E. Riseman and C. Foster, “The Inhibition of Potential Parallelism by Conditional Jumps,” *IEEE Trans. on Computers,* Dec., 1972.

[6] J. E. Smith, “A Study of Branch Prediction Strategies,” *Proc. Eighth Symp. on Computer Architecture,* May, 1981.

[7] G. S. Tjaden and M. J. Flynn, “Represent at ion of Concurrency with Ordering Matrices,” *IEEE Trans. on Computers,* Aug., 1973.

[8] G. S. Tjaden and M. J. Flynn, “Detection and Parallel Execution of Independent Instructions,” *IEEE Trans. Computers,* Vol. C-19, No. 10, pp. 889-895, Oct., 1970.

[9] B. L. Peuto and L. J. Shustek, “Current issues in the architecture of microprocessors ,” *IEEE Computer,* February 1977, pp. 20–25.

[10] T. R. Gross and J. Hennessey, “Optimizing delayed branches,” *Proc. 15th Workshop on Microprogramming,* 1986.

[11] P. Y. T. Hsu and E. S. Davidson, “Highly concurrent scalar processing,” *Proc. 13th Symp. on Computer Architecture,* pp. 386-395, June, 1986.

Appendix

Additional i-traffic calculation under freeze conditions:

Additional i-traffic

$$T = m_1 + m_2 + m_3 + m_4$$

where

m_1	=	wasted i-fetches, assuming address-calculation as well as target-fetch freezes
	$=$	pos $(\mathrm{N} - Y_a - Y_f) * P_a * P_f$
m_2	=	wasted i-fetches, assuming address-calculation freeze but no target-fetch freeze
	$=$	pos $(\mathrm{N} - Y_a - Y_f) * P_a * (1 - P_f)$
m_3	=	wasted i-fetches, assuming no address-calculation freeze but target-fetch freeze
	$=$	pos $(\mathrm{N} - Y_a - Y_f) * (1 - P_a) * P_f$
m_4	=	wasted i-fetches, assuming no address-calculation freeze and no target-fetch freeze
	$=$	pos $(\mathrm{N} - Y_a - Y_f) * (1 - P_a) * (1 - P_f)$
N	**=**	maximum possible i-fetches assuming no freeze (N $= (S_b - 2)$ in the above example)
pos (l)	=	l for $l > 0$
	$=$	0 for $l \leq 0$

For the sake of brevity, in following sections the above calculation would be expressed as:

$$T = N \sim P_a * Y_a \sim P_f * Y_a$$

where $\sim$ refers to the probability-based reduction in N, as explained above.

The above calculation is based on an import ant assumption that there is no additional i-traffic generated during the freeze conditions. Excess i-traffic would be generated if the freeze conditions require any i-fetches (for example, during page-fault handling). Incorrect predictions not only result in wasted instruction fetches but may also result in unnecessary operand fetches. For our analysis, this increase in d-traffic or any interference of operand fetches with the instruction fetches is ignored.

Given below are the details of calculation for the different branch-delay and excess i-traffic parameters associated with each strategy:

Predict Branch Never Taken *(PBNT):*

$$\begin{aligned}
p_{\mathrm{t}} &= 0 \\
k_{\mathrm{n,n}} &= 0 \\
k_{\mathrm{n,b}} &= (S_{\mathrm{b}} - 1) + P_{\mathrm{a}} * Y_{\mathrm{a}} + P_{\mathrm{f}} * Y_{\mathrm{f}} \\
t_{\mathrm{n,n}} &= 0 \\
t_{\mathrm{n,b}} &= (S_{\mathrm{b}} - 1)
\end{aligned}$$

Loop Buffers *(LB):* Considering p_{lh} as the hit-probability for the loop-buffers, different parameters for this case are:

$$\begin{aligned}
p_{\mathrm{t}} &= 0 \\
k_{\mathrm{n,n}} &= 0 \\
k_{\mathrm{n,b}} &= ((S_{\mathrm{b}} - 1) + P_{\mathrm{a}} * Y_{\mathrm{a}} + P_{\mathrm{f}} * Y_{\mathrm{f}}) * (1 - P_{\mathrm{lh}}) + ((S_{\mathrm{b}} - 2) \\
&\quad + P_{\mathrm{a}} * Y_{\mathrm{a}}) * P_{\mathrm{lh}} \\
t_{\mathrm{n,n}} &= 0 \\
t_{\mathrm{n,b}} &= (S_{\mathrm{b}} - 1) * (1 - P_{\mathrm{lh}})
\end{aligned}$$

Pre-calculate Target Address *(PTA):*

$$\begin{aligned}
p_{\mathrm{t}} &= 0 \\
k_{\mathrm{n,n}} &= 0
\end{aligned}$$

$$
\begin{aligned}
k_{\mathrm{n,b}} &= (S_{\mathrm{b}} - 1) + P_{\mathrm{a}} * pos(Y_{\mathrm{a}} - (S_{\mathrm{b}} - 1 - S_{\mathrm{f}})) + P_{\mathrm{f}} * Y_{\mathrm{f}} \\
t_{\mathrm{n,n}} &= 0 \\
t_{\mathrm{n,b}} &= (S_{\mathrm{b}} - 1)
\end{aligned}
$$

Target-fetch in the OF-slot *(FTOF):*

$$
\begin{aligned}
p_{\mathrm{t}} &= 0 \\
k_{\mathrm{n,n}} &= (1 - p_{\mathrm{a}}) * (1 - p_{\mathrm{f}}) + p_{\mathrm{a}} * p_{\mathrm{f}} \\
k_{\mathrm{n,b}} &= \alpha_1 * P_{\mathrm{a}} * P_{\mathrm{f}} + \alpha_2 * P_{\mathrm{a}} * (1 - P_{\mathrm{f}}) \\
&\quad + \alpha_3 * (1 - P_{\mathrm{a}}) * P_{\mathrm{f}} + \alpha_4 * (1 - P_{\mathrm{a}}) * (1 - P_{\mathrm{f}})
\end{aligned}
$$

where

$$
\begin{aligned}
\alpha_1 &= (S_{\mathrm{b}} - 1 + Y_{\mathrm{a}} + Y_{\mathrm{f}}) \\
\alpha_2 &= (S_{\mathrm{b}} - 1 + Y_{\mathrm{a}}) \\
\alpha_3 &= (S_{\mathrm{b}} - 1 + Y_{\mathrm{f}}) \\
\alpha_1 &= (S_{\mathrm{b}} - 2) \\
t_{\mathrm{n,n}} &= (1 - p_{\mathrm{a}}) * (1 - p_{\mathrm{f}}) \\
t_{\mathrm{n,b}} &= \beta_1 * P_{\mathrm{a}} * P_{\mathrm{f}} + \beta_1 * P_{\mathrm{a}} * (1 - P_{\mathrm{f}}) \\
&\quad + \beta_2 * (1 - P_{\mathrm{a}}) * P_{\mathrm{f}} + \beta_2 * (1 - P_{\mathrm{a}}) * (1 - P_{\mathrm{f}})
\end{aligned}
$$

where

$$
\begin{aligned}
\beta_1 &= (S_{\mathrm{b}} - 1) \\
\beta_2 &= (S_{\mathrm{b}} - 2)
\end{aligned}
$$

Predict Branch Always Taken *(PBAT):*

$$
\begin{aligned}
p_{\mathrm{t}} &= 1 \\
k_{\mathrm{b,n}} &= (S_{\mathrm{b}} - 1 - S_{\mathrm{f}}) \\
k_{\mathrm{b,b}} &= S_{\mathrm{f}} + P_{\mathrm{a}} * Y_{\mathrm{a}} + P_{\mathrm{f}} * Y_{\mathrm{f}} \\
t_{\mathrm{b,n}} &= (S_{\mathrm{b}} - 1 - S_{\mathrm{f}}) \sim P_{\mathrm{a}} * Y_{\mathrm{a}} \sim P_{\mathrm{f}} * Y_{\mathrm{f}} \\
t_{\mathrm{b,b}} &= S_{\mathrm{f}}
\end{aligned}
$$

Predict Branch Always Taken with Target Copy $(PTTC)$**:**

$$\begin{aligned}
p_t &= 1\\
k_{b,n} &= (S_b - 1)\\
k_{b,b} &= P_a * pos(Y_a - (S_b - 1 - S_f)) + P_f * Y_f\\
t_{b,n} &= (S_b - 1)\\
t_{b,b} &= 0
\end{aligned}$$

Fetch Both the Paths *(FBP):*

$$\begin{aligned}
p_c &= 1\\
k_{n,n} &= 0 \text{ (branch not taken)}\\
k_{b,b} &= S_f + P_a * Y_a + P_f * Y_f \quad \text{(branch taken)}\\
t_{n,n} &= (S_b - 1 - S_f) \sim P_a * Y_a \sim P_f * Y_f \quad \text{(branch not taken)}\\
t_{b,b} &= (S_b - 1) \quad \text{(branch taken)}
\end{aligned}$$

Delayed Branch *(DB):*

$$\begin{aligned}
p_t &= 0\\
k_{n,b} &= \text{pos}\,(S_b - 1 - u) + P_a * Y_a + P_f * Y_f\\
k_{n,n} &= 0\\
t_{n,b} &= \text{pos}\,(S_b - 1 - u)\\
t_{n,n} &= 0
\end{aligned}$$

Taken/Not-taken Switch in the Decode Stage *(TNTD):*

$$\begin{aligned}
k_{b,n} &= (S_b - 1 - S_f)\\
k_{b,b} &= S_f + P_a * Y_a + P_f * Y_f\\
k_{n,b} &= (S_b - 1) + P_a * Y_a + P_f * Y_f\\
k_{n,n} &= \mathbf{0}\\
t_{b,n} &= (S_b - 1 - S_f) \sim P_a * Y_a \sim P_f * Y_f
\end{aligned}$$

$$\begin{aligned} t_{b,b} &= S_f \\ t_{n,b} &= (S_b - 1) \\ t_{n,n} &= 0 \end{aligned}$$

Branch Target Buffer *(BTB):* Model parameters in case of BTB-hit are given below:

$$\begin{aligned} k_{b,n} &= (S_b - 1) \\ k_{b,b} &= (P_f * Y_f) * P_{ct} + (S_f + P_a * Y_a + P_f * Y_f) * (1 - P_{ct}) \\ k_{n,b} &= ((S_b - 1) + P_f * Y_f) * P_{ct} \\ &\quad +((S_b - 1) + P_a * Y_a + P_f * Y_f) * (1 - P_{ct}) \\ k_{n,n} &= 0 \\ t_{n,n} &= 0 \end{aligned}$$

Additional i-traffic, when branch is predicted as *to-be-taken* and is actually taken:

$$t_{b,b} = (\sigma_1 + \sigma_2 + \sigma_3 + \sigma_4) * (1 - P_{ct})$$

Assuming min *(a, b)* stands for *a* or *b*, whichever is minimum, we have:

$$\begin{aligned} \sigma_1 &= \min(\text{pos}(S_f + Y_a - Y_f)(S_b - 1)) * P_a * P_f \\ \sigma_2 &= \min((S_f + Y_a)(S_b - 1)) * P_a * (1 - P_f) \\ \sigma_3 &= \text{pos}(S_f - Y_f) * (1 - P_a) * P_f \\ \sigma_4 &= S_f * (1 - P_a) * (1 - Pf) \end{aligned}$$

Additional i-traffic, when branch is predicted as *to-be-taken* but it turns out to be an incorrect prediction:

Let P_{fp} and Y_{fp} refer to the freeze potential and freeze duration, respectively, at the predicted target and P_{fc} and Y_{fc} refer to the same at the actual calculated target. Note that this distinction is made only for better understanding of the following details; numerically we assume $P_{\mathrm{fp}} = P_{\mathrm{fc}} = P_{\mathrm{f}}$ and $Y_{\mathrm{fp}} = Y_{\mathrm{fc}} = Y_{\mathrm{f}}$. Discarded i-fetches in this case would be:

$$t_{\mathrm{b,n}} = ((S_{\mathrm{b}} - 1) \sim P_{\mathrm{f}} * Y_{\mathrm{f}}) * P_{\mathrm{ct}} + \delta * (1 - P_{\mathrm{ct}})$$

where

$$\delta = \delta_1 + \delta_2 + \delta_3 + \delta_4 + \delta_5 + \delta_6 + \delta_7 + \delta_8$$

and

$$\begin{aligned}
\delta_1 &= (\mathrm{pos}\,(S_{\mathrm{f}} - Y_{\mathrm{fp}}) + S_{\mathrm{b}} - 1 - S_{\mathrm{f}}) * P_{\mathrm{fp}} * (1 - P_{\mathrm{fc}}) * (1 - P_{\mathrm{a}}) \\
\delta_2 &= (\mathrm{pos}\,(S_{\mathrm{f}} - Y_{\mathrm{fp}}) + \mathrm{pos}\,(S_{\mathrm{b}} - 1 - S_{\mathrm{f}} - Y_{\mathrm{fc}})) * P_{\mathrm{fp}} * P_{\mathrm{fc}} * (1 - P_{\mathrm{a}}) \\
\delta_3 &= (S_{\mathrm{b}} - 1) * (1 - P_{\mathrm{fp}}) * (1 - P_{\mathrm{fc}}) * (1 - P_{\mathrm{a}}) \\
\delta_4 &= (S_{\mathrm{f}} + \mathrm{pos}\,(S_{\mathrm{b}} - 1 - S_{\mathrm{f}} - Y_{\mathrm{fc}})) * (1 - P_{\mathrm{fp}}) * P_{\mathrm{fc}} * (1 - P_{\mathrm{a}}) \\
\delta_5 &= (\min\,(\mathrm{pos}\,(S_{\mathrm{f}} + Y_{\mathrm{a}} - Y_{\mathrm{fp}}), (S_{\mathrm{b}} - 1)) + \mathrm{pos}\,(S_{\mathrm{b}} - 1 - S_{\mathrm{f}} - Y_{\mathrm{a}})) \\
&\quad * P_{\mathrm{fp}} * (1 - P_{\mathrm{fc}}) * P_{\mathrm{a}} \\
\delta_6 &= (\min\,(\mathrm{pos}\,(S_{\mathrm{f}} + Y_{\mathrm{a}} - Y_{\mathrm{fp}}), (S_{\mathrm{b}} - 1)) + \mathrm{pos}\,(S_{\mathrm{b}} - 1 - S_{\mathrm{f}} - Y_{\mathrm{a}} - Y_{\mathrm{fc}})) \\
&\quad * P_{\mathrm{fp}} * P_{\mathrm{fc}} * P_{\mathrm{a}} \\
\delta_7 &= (S_{\mathrm{b}} - 1) * (1 - P_{\mathrm{fp}}) * (1 - P_{\mathrm{fc}}) * P_{\mathrm{a}} \\
\delta_8 &= (\min\,((S_{\mathrm{f}} + Y_{\mathrm{a}}), (S_{\mathrm{b}} - 1)) + \mathrm{pos}\,(S_{\mathrm{b}} - 1 - S_{\mathrm{f}} - Y_{\mathrm{a}} - Y_{\mathrm{fc}})) \\
&\quad * (1 - P_{\mathrm{fp}}) * P_{\mathrm{fc}} * P_{\mathrm{a}}
\end{aligned}$$

Additional i-traffic, when branch is predicted as *not-to-be-taken* but is actually taken:

$$t_{\mathrm{n,b}} = (S_{\mathrm{b}} - 1) + \gamma * (1 - P_{\mathrm{ct}})$$

where

$$\begin{aligned}
\gamma &= \gamma_1 * P_{\mathrm{a}} * P_{\mathrm{f}} + \gamma_2 * P_{\mathrm{a}} * (1 - P_{\mathrm{f}}) + \gamma_3 * (1 - P_{\mathrm{a}}) * P_{\mathrm{f}} \\
&\quad + \gamma_4 * (1 - P_{\mathrm{a}}) * (1 - P_{\mathrm{f}})
\end{aligned}$$

and

$$\begin{aligned}
\gamma_1 &= \min\,(\mathrm{pos}\,(Y_{\mathrm{a}} - Y_{\mathrm{f}}), S_{\mathrm{b}} - 1) \\
\gamma_2 &= \min\,(Y_{\mathrm{a}}, S_{\mathrm{b}} - 1) \\
\gamma_3 &= \gamma_4 = 0
\end{aligned}$$

Predict Branch Always taken with target-copy and delayed branch *(TTCDB):*

$$\begin{aligned}
p_{\mathrm{t}} &= 1 \\
k_{\mathrm{b,n}} &= \mathrm{pos}\,(S_{\mathrm{b}} - 1 - u) \\
k_{\mathrm{b,b}} &= P_{\mathrm{a}} * \mathrm{pos}\,(Y_{\mathrm{a}} - (S_{\mathrm{b}} - 1 - S_{\mathrm{f}})) + P_{\mathrm{f}} * Y_{\mathrm{f}} \\
t_{\mathrm{b,n}} &= \mathrm{pos}\,(S_{\mathrm{b}} - 1 - u) \\
t_{\mathrm{b,b}} &= 0
\end{aligned}$$

Predict Branch Always taken with target-copy, delayed branch and Loop buffer *(TTDLB):*

$$\begin{aligned}
p_{\mathrm{t}} &= 1 \\
k_{\mathrm{b,n}} &= \mathrm{pos}\,(S_{\mathrm{b}} - 1 - u) * (1 - P_{\mathrm{lh}}) + \mathrm{pos}\,(S_{\mathrm{b}} - 2 - u) * P_{\mathrm{lh}} \\
k_{\mathrm{b,b}} &= P_{\mathrm{a}} * \mathrm{pos}\,(Y_{\mathrm{a}} - (S_{\mathrm{b}} - 1 - S_{\mathrm{f}})) * P_{\mathrm{lh}} + (P_{\mathrm{a}} * \mathrm{pos}\,(\,Y_{\mathrm{a}} - (S_{\mathrm{b}} - 1 - S_{\mathrm{f}})) \\
&\quad + P_{\mathrm{f}} * Y_{\mathrm{f}}) * (1 - P_{\mathrm{lh}}) \\
t_{\mathrm{b,n}} &= \mathrm{pos}\,(S_{\mathrm{b}} - 1 - u) * (1 - P_{\mathrm{lh}}) \\
t_{\mathrm{b,b}} &= 0
\end{aligned}$$

Taken/Not-taken Switch in the Decode Stage with Loop buffer *(TNTLB):*

$$\begin{aligned}
k_{\mathrm{b,n}} &= (S_{\mathrm{b}} - 1 - S_{\mathrm{f}}) \\
k_{\mathrm{b,b}} &= ((\,S_{\mathrm{f}} - 1) + P_{\mathrm{a}} * Y_{\mathrm{a}}) * P_{\mathrm{lh}} + (S_{\mathrm{f}} + P_{\mathrm{a}} * Y_{\mathrm{a}} + P_{\mathrm{f}} * Y_{\mathrm{f}}) * (1 - P_{\mathrm{lh}}) \\
k_{\mathrm{n,b}} &= ((S_{\mathrm{b}} - 1) + P_{\mathrm{a}} * Y_{\mathrm{a}} + P_{\mathrm{f}} * Y_{\mathrm{f}}) * (1 - P_{\mathrm{lh}}) + ((S_{\mathrm{b}} - 2) + P_{\mathrm{a}} * Y_{\mathrm{a}}) * P_{\mathrm{lh}} \\
k_{\mathrm{n,n}} &= \mathbf{0} \\
t_{\mathrm{b,n}} &= ((S_{\mathrm{b}} - 1 - S_{\mathrm{f}}) \sim P_{\mathrm{a}} * Y_{\mathrm{a}} \sim P_{\mathrm{f}} * Y_{\mathrm{f}}) * (1 - P_{\mathrm{lh}}) \\
t_{\mathrm{b,b}} &= S_{\mathrm{f}} * (1 - P_{\mathrm{lh}}) \\
t_{\mathrm{n,b}} &= (S_{\mathrm{b}} - 1) * (1 - P_{\mathrm{lh}}) \\
t_{\mathrm{n,n}} &= \mathbf{0}
\end{aligned}$$

Non-hybrid strategies:

PBNT:	Predict branch never taken
LB:	*PBNT* + Loop Buffer
PTA:	*PBNT* + Pre-calculate target address
FTOF:	*PBNT* + Target-fetch in the OF-slot
PBAT:	Predict branch always taken
PTTC:	*PBAT* + Target-copy
FBP:	Fetch both the paths
DB:	*PBNT* + Delayed branch
TNTD:	Branch taken/not-taken prediction switch in the ID stage
BTB:	Branch target buffer

Hybrid strategies:

TTCDB:	Predict branch always taken with target-copy and delayed branch *(PTTC + DB)*
TTDLB:	Predict branch always taken with target-copy and delayed branch and loop buffer *(TTCDB + LB)*
TNTLB:	Branch taken/not-taken prediction switch in the ID stage and loop buffer *(TNTD + BTB)*
TNBTB:	Branch taken/not-taken prediction switch in the ID stage and branch target buffer *(TNTD + BTB)*

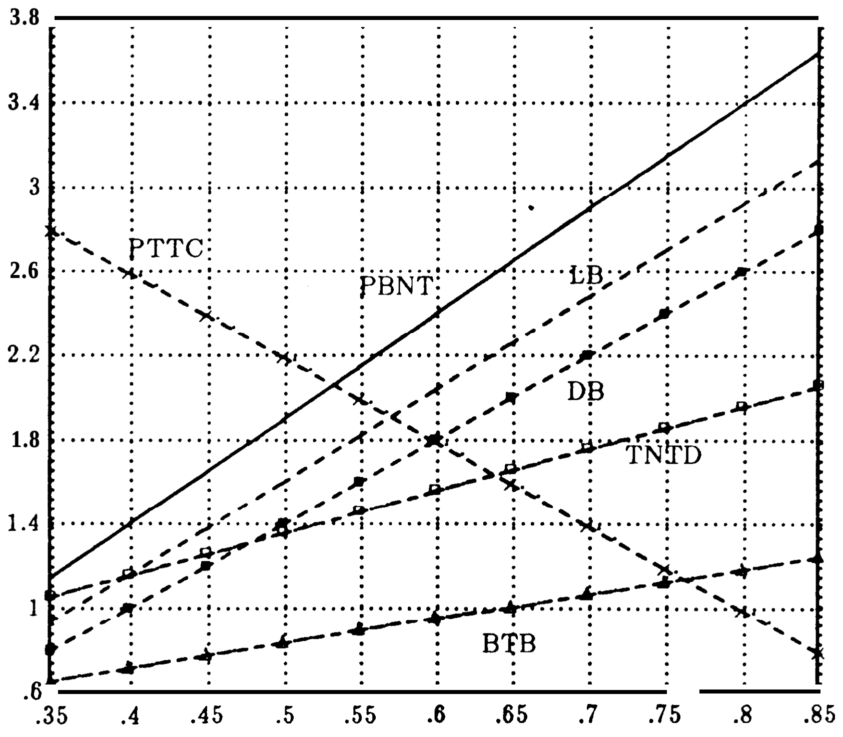
Nominal values:

Average branch frequency:	0.25
Average fraction of conditional branches:	0.8
Overall fraction of successful branches: (conditional/unconditional combined)	0.6
# of pipeline stages until unconditional branch resolution:	2
# of buffer sub-stages in the instruction-fetch stage:	1
# of pipeline stages until conditional branch resolution:	5
Prob. of freeze during target address formation:	0.5
Duration of target-address-calculation freeze:	2 cycles
Prob. of freeze during target-fetch:	0
Duration of target-fetch freeze:	10 cycles
Prob. of loop-buffer hit:	0.6
Prob. of BTB hit:	0.8
Prob. of correct address prediction from BTB:	0.9
Prob. of BTB-hit for non-branch instruction:	0.05
Average # of delay-slots filled in delayed branch approach:	1

For cases with active prediction schemes: $(TNTD,\ BTB,\ TNTLB,\ TNBTB)$

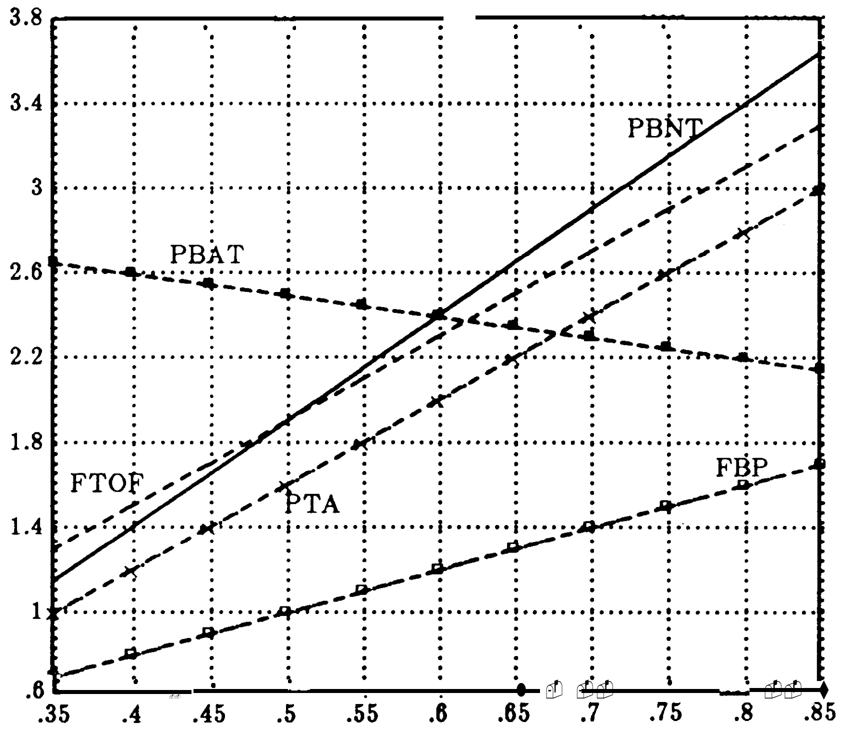
Correct prediction prob. for unconditional branches:	1.0
Correct prediction prob. for conditional branches:	0.85

Average
Delay
(per branch)

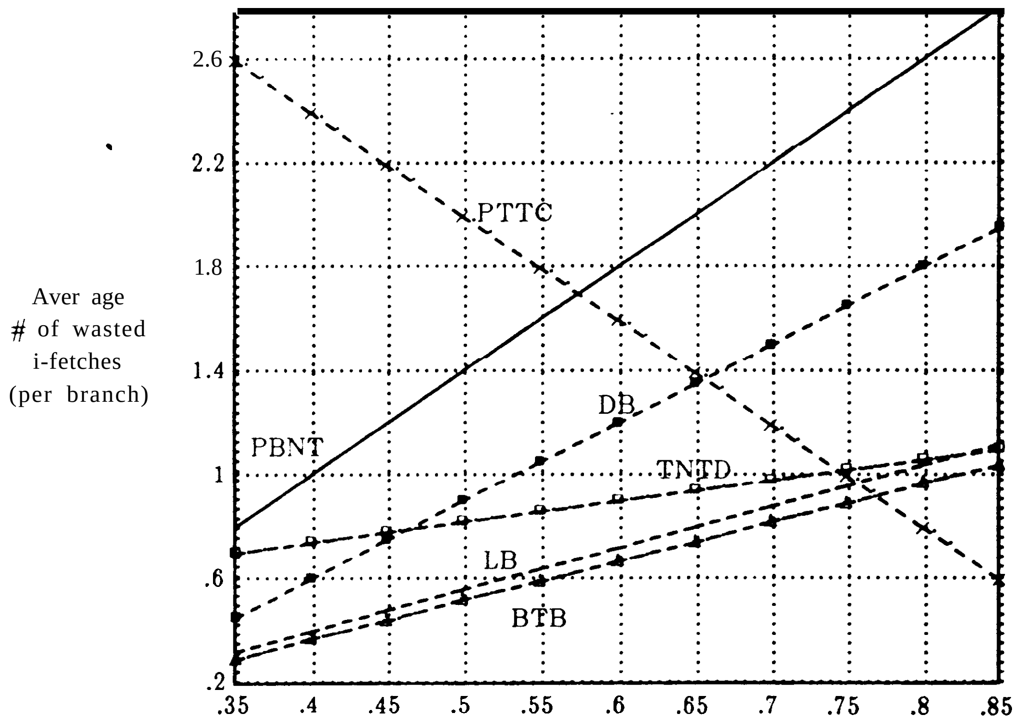


Successful Branch Probability
Figure 6

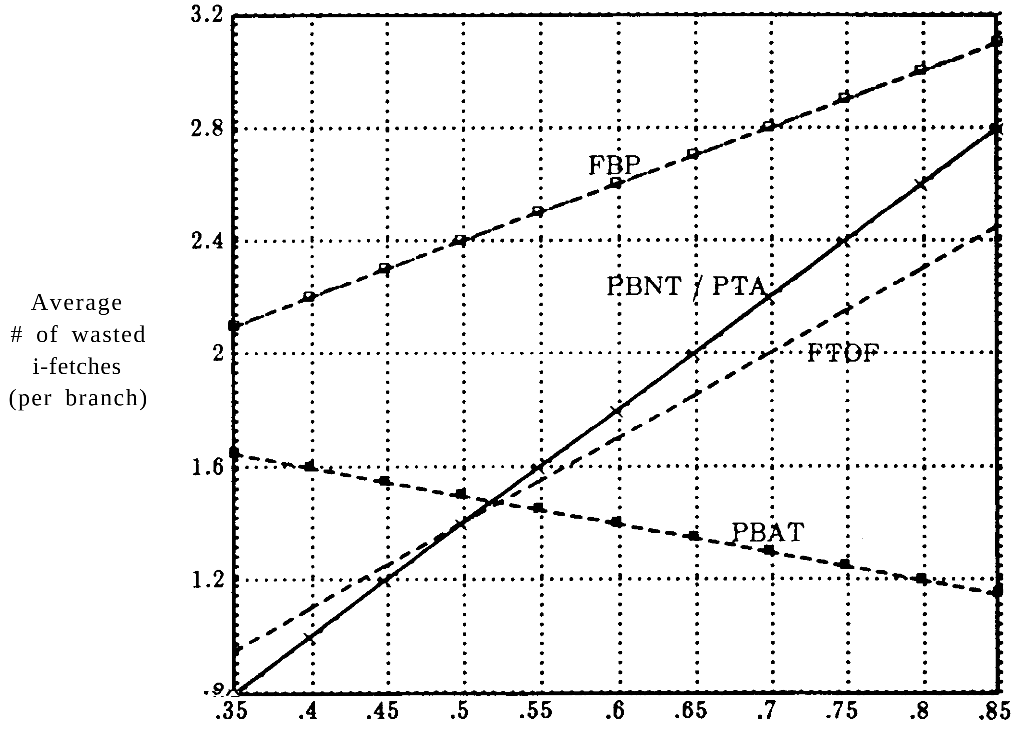
Average
Delay
(per branch)



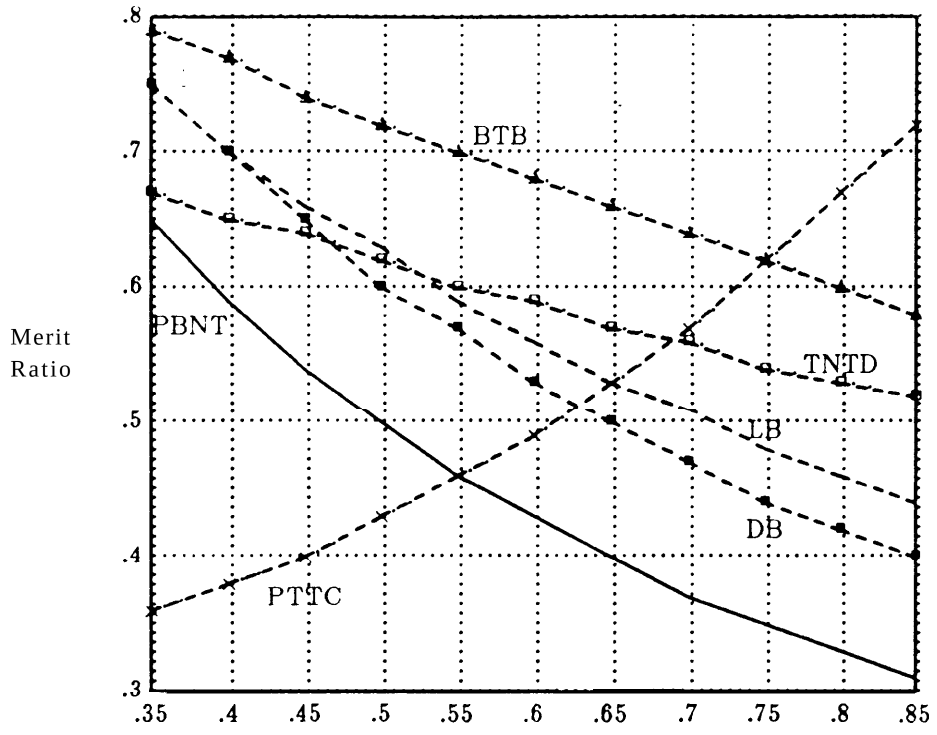
Successful Branch Probability
Figure 7



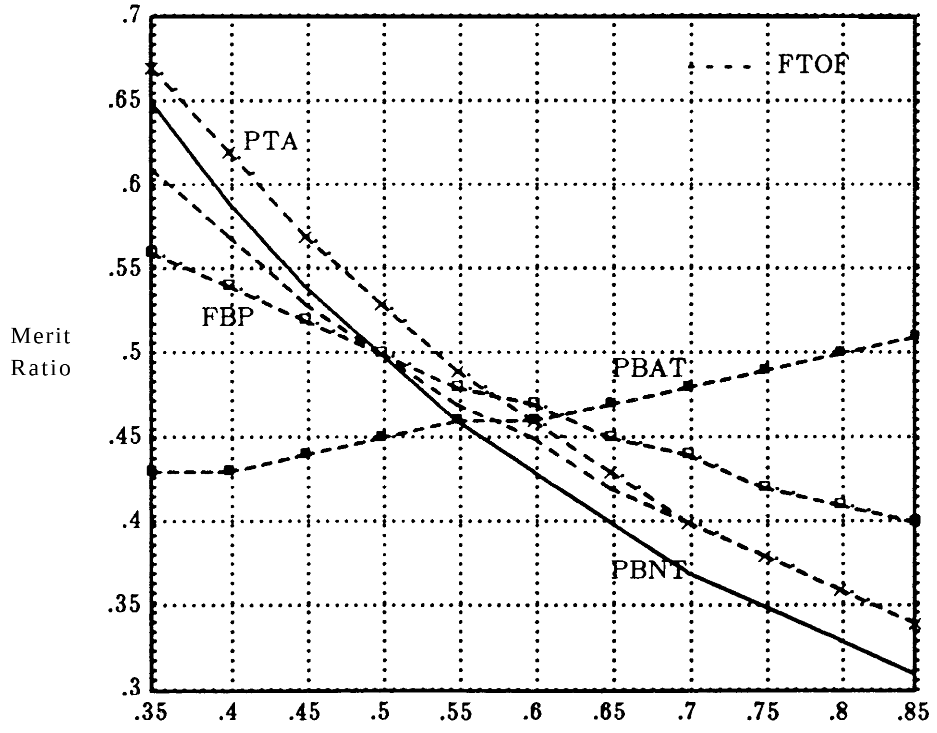
Successful Branch Probability
Figure 8



Successful Branch Probability
Figure 9

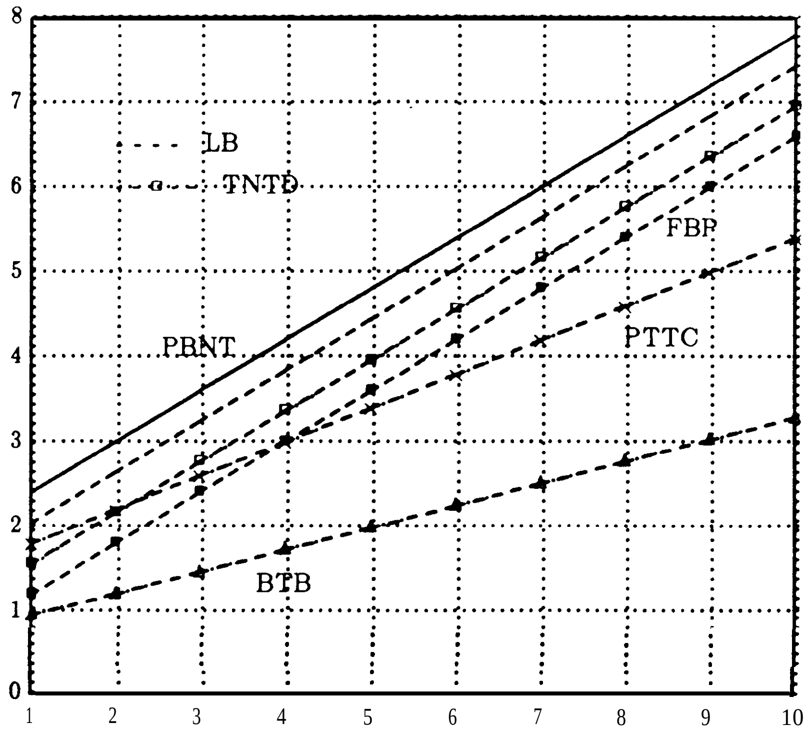


Successful Branch Probability
Figure 10



Successful **Branch Probability**
Figure 11

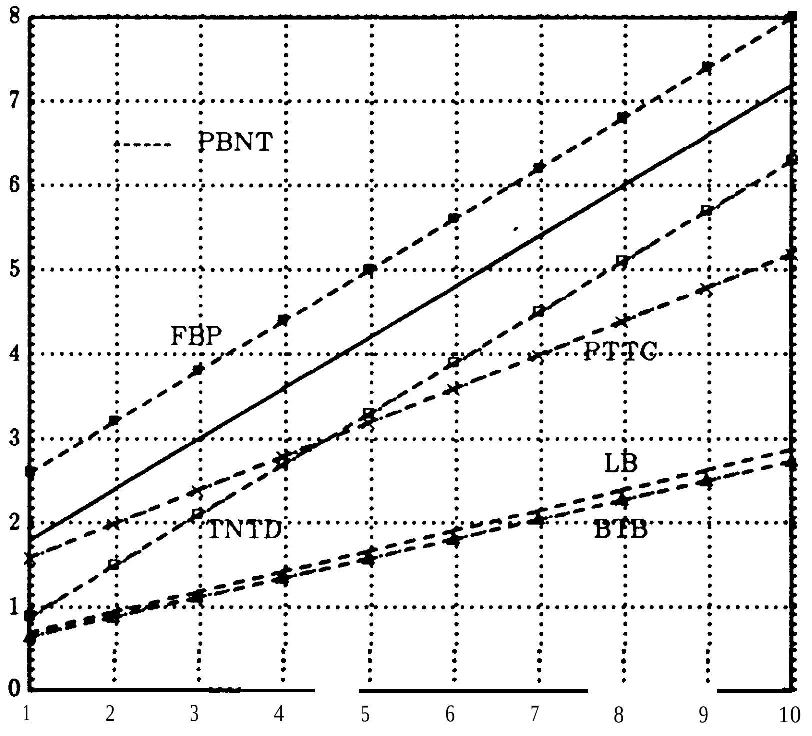
Average
Delay
(per branch)



# of sub-stages in the fetch-stage

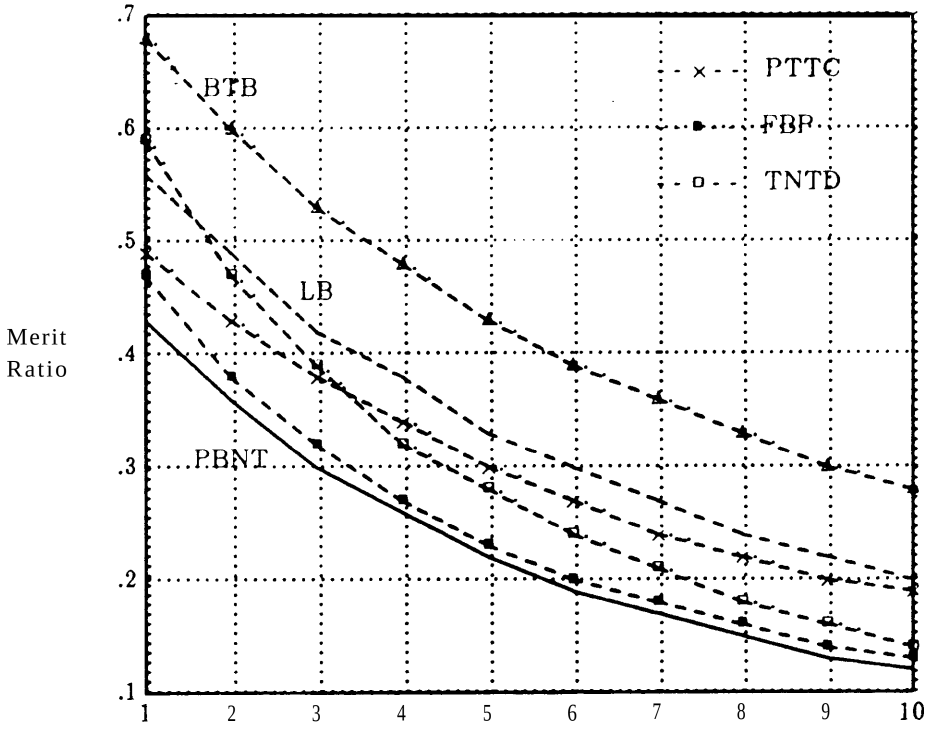
Figure 12

Aver age
# of wasted
i-fetches
(per branch)

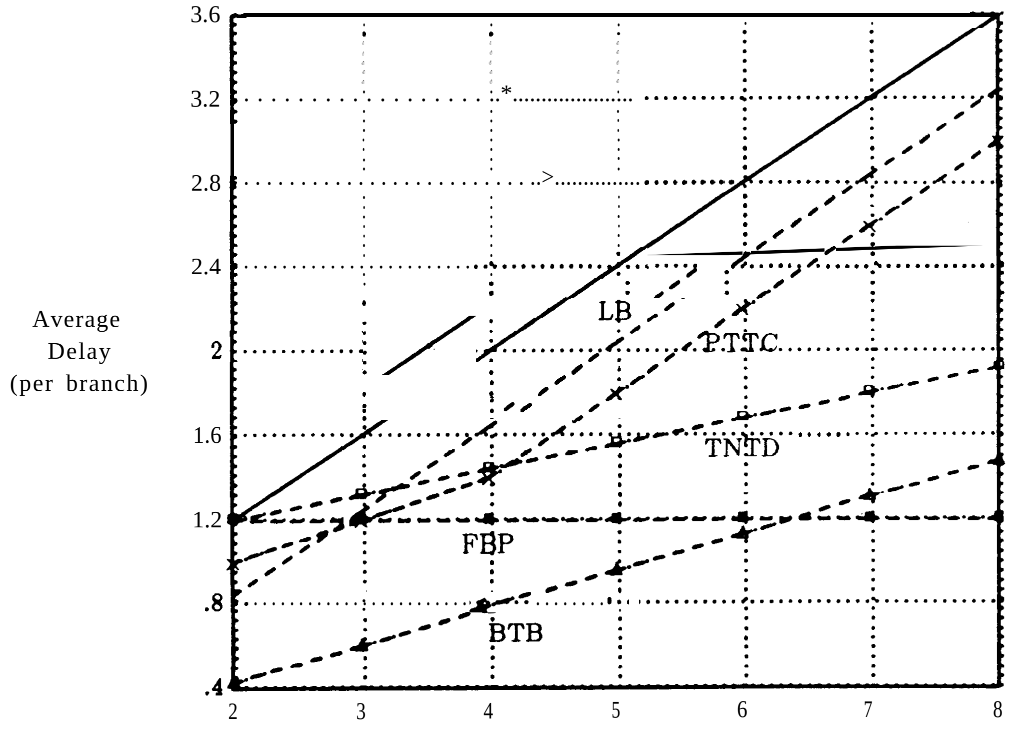


# of sub-stages in the fetch-stage

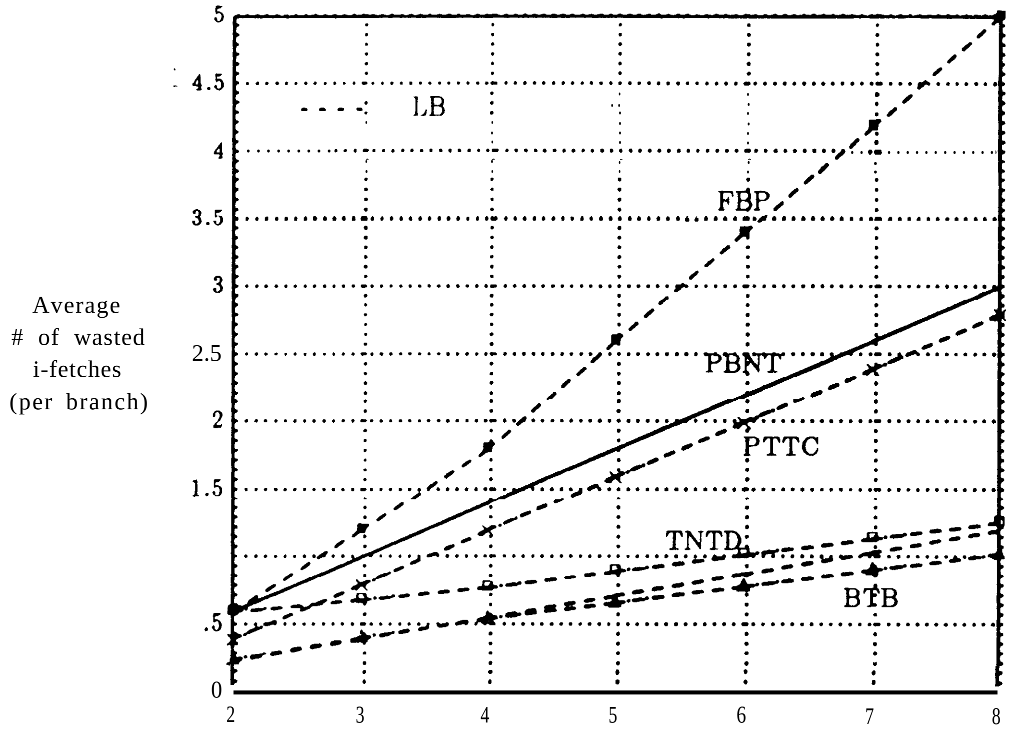
Figure 13



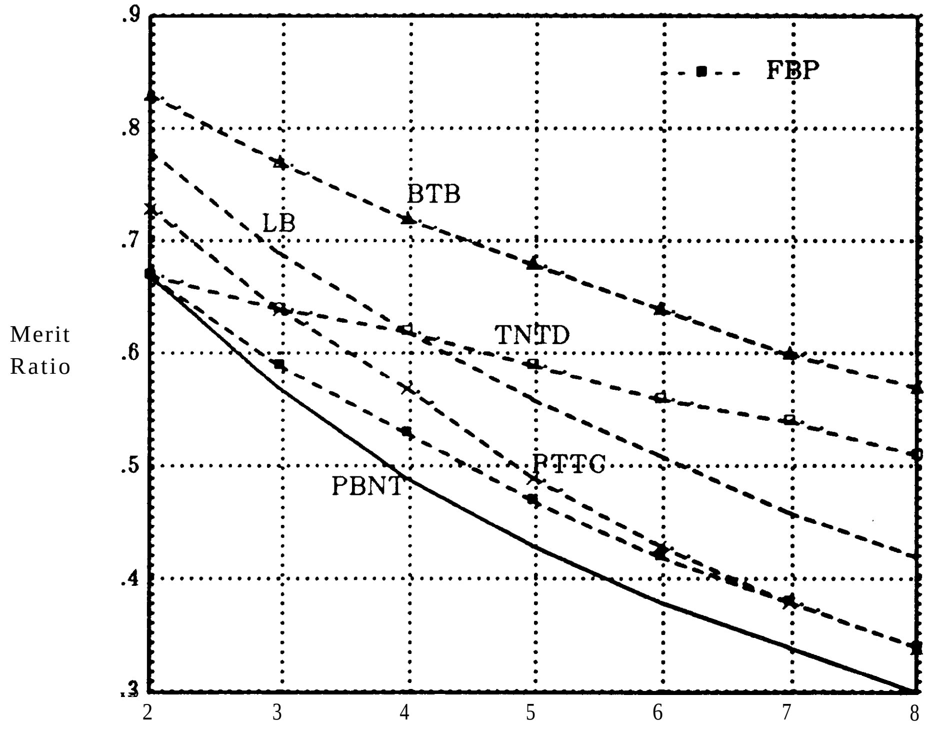
# of sub-stages in the fetch-stage
Figure 14



# of Stages for Conditional Branch Resolution
Figure 15

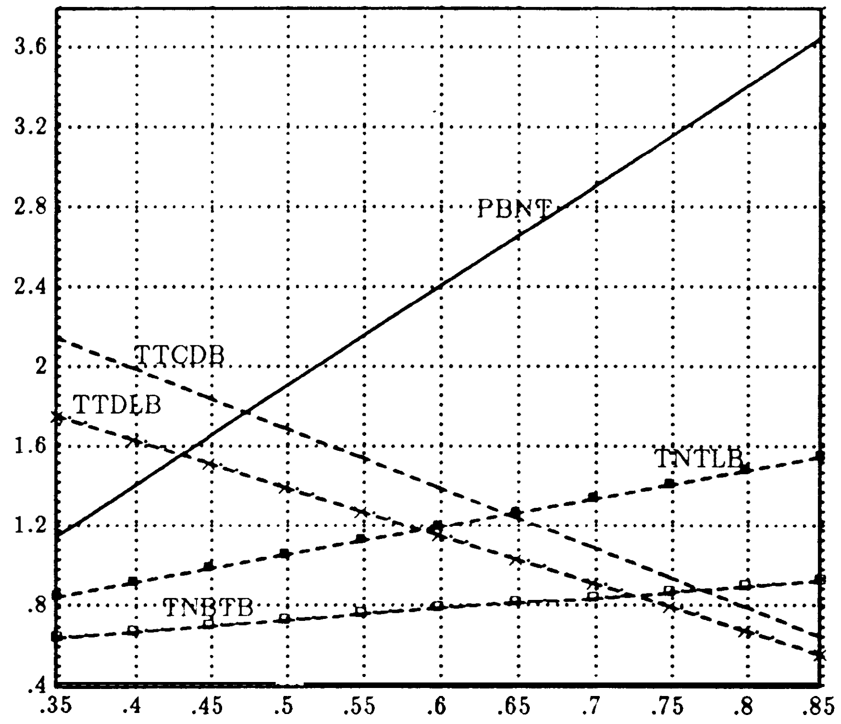


# of Stages for Conditional Branch Resolution
Figure 16



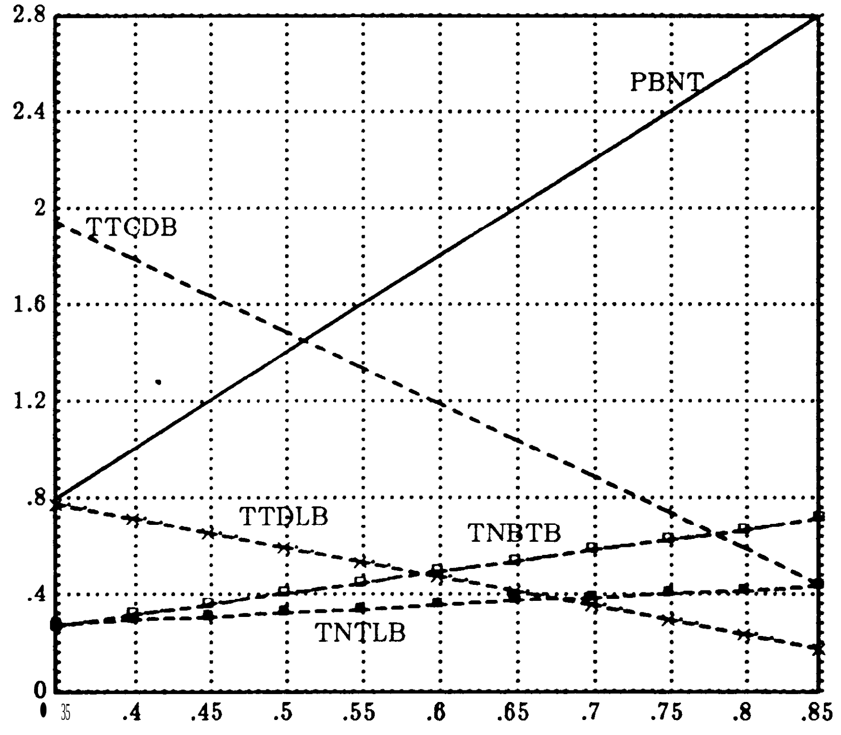
# of Stages for Conditional **Branch Resolution**
Figure **17**

Average
Delay
(per branch)

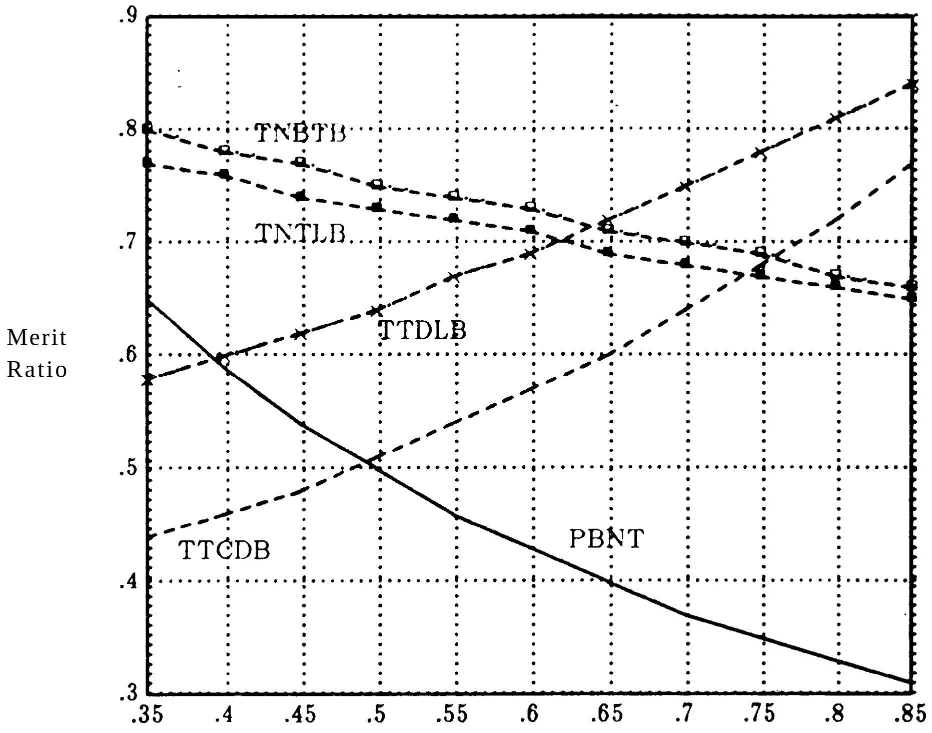


Successful Branch Probability
Figure 18

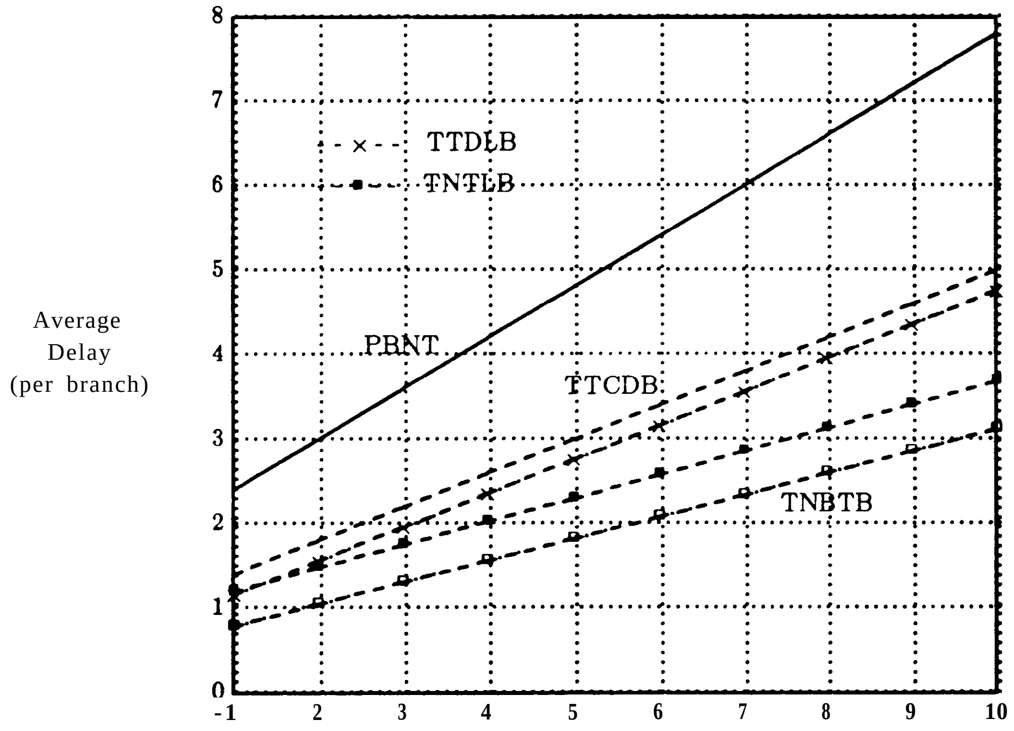
Average
# of wasted
i-fetches
(per branch)



Successful Branch Probability
Figure 19

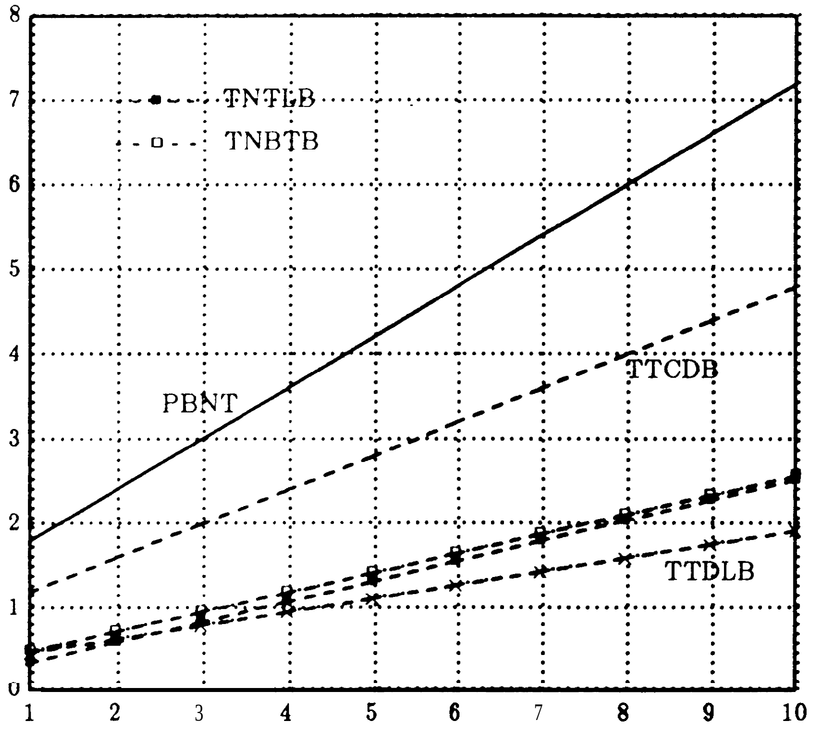


Successful Branch Probability
Figure 20



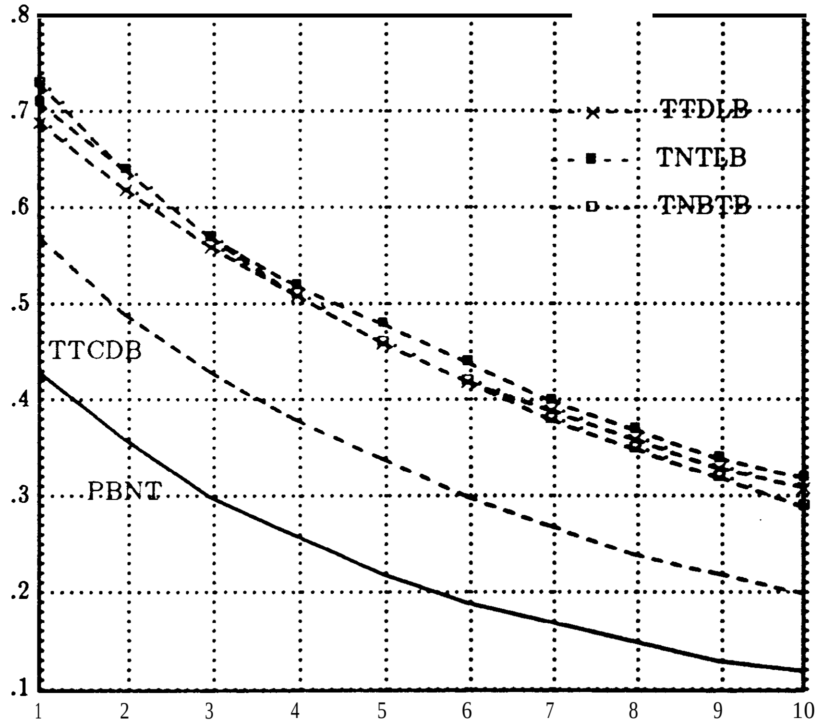
# of sub-stages in the fetch-stage
Figure 21

Average
# of wasted
i-fetches
(per branch)



# of sub-stages in the fetch-stage
Figure 22

Merit
Ratio



# of sub-stages in the fetch-stage
Figure 23

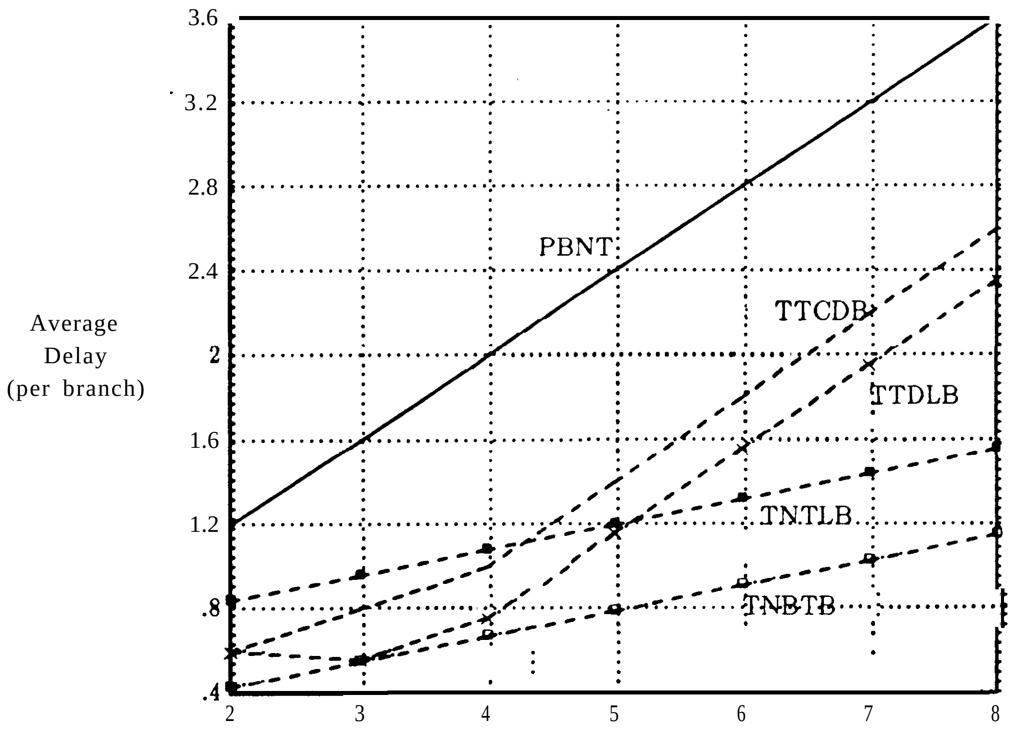


Figure 24

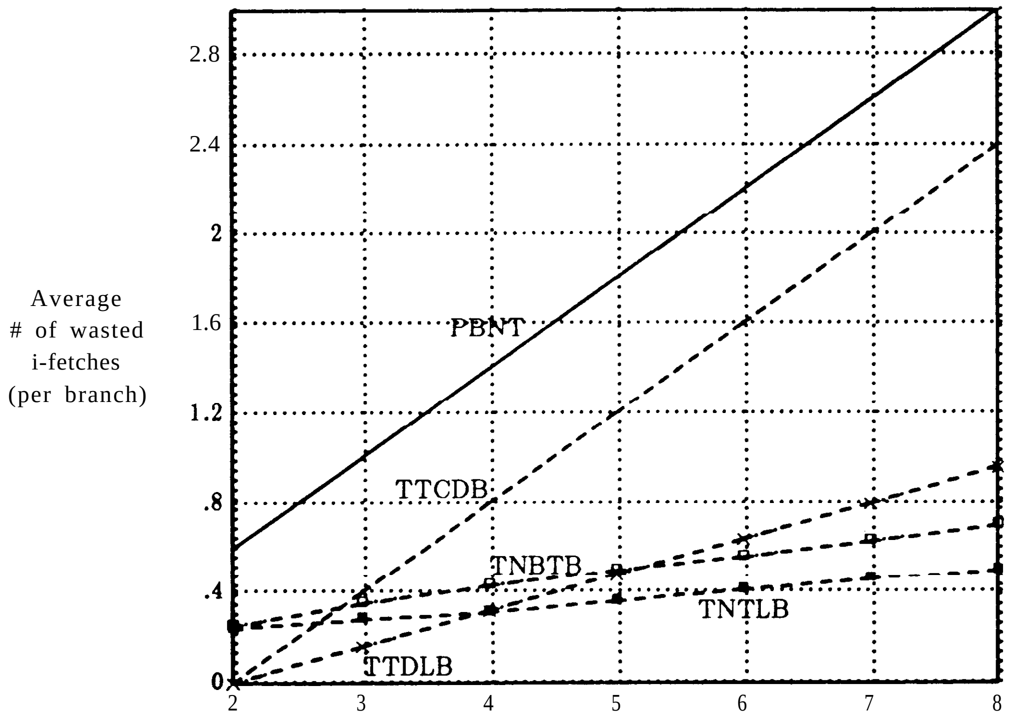


Figure 25

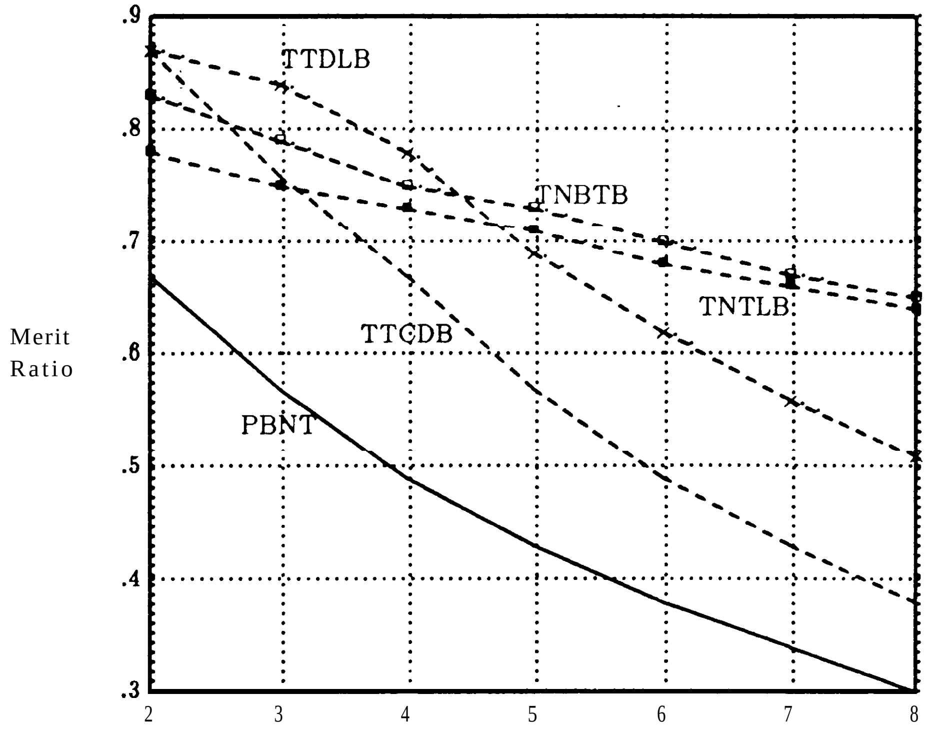


Figure 26

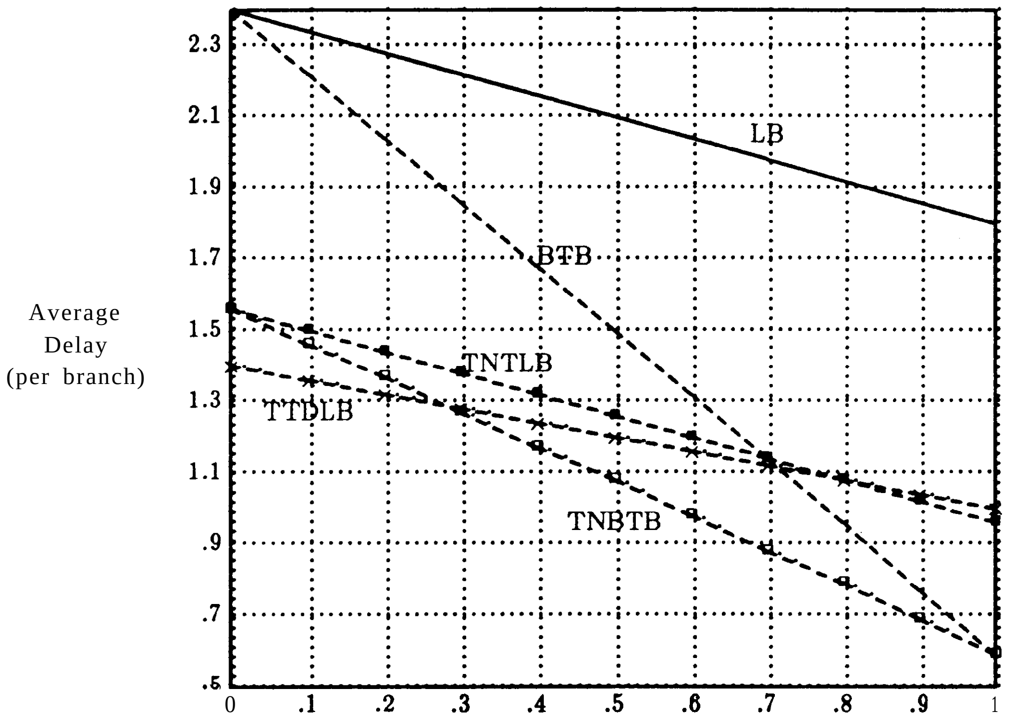
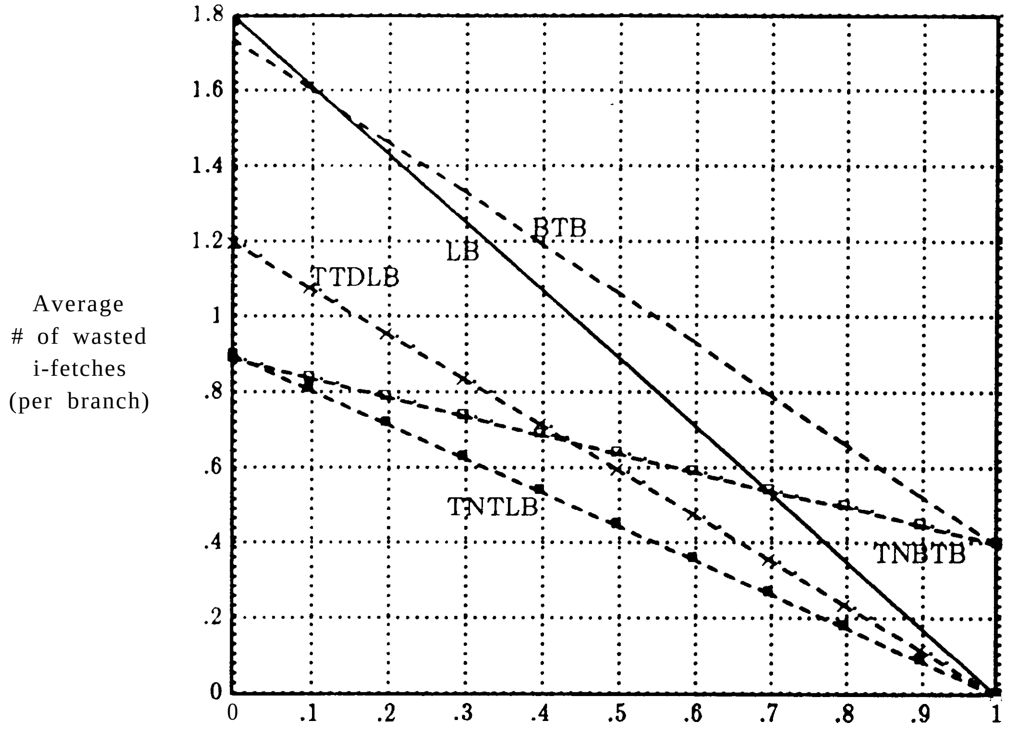
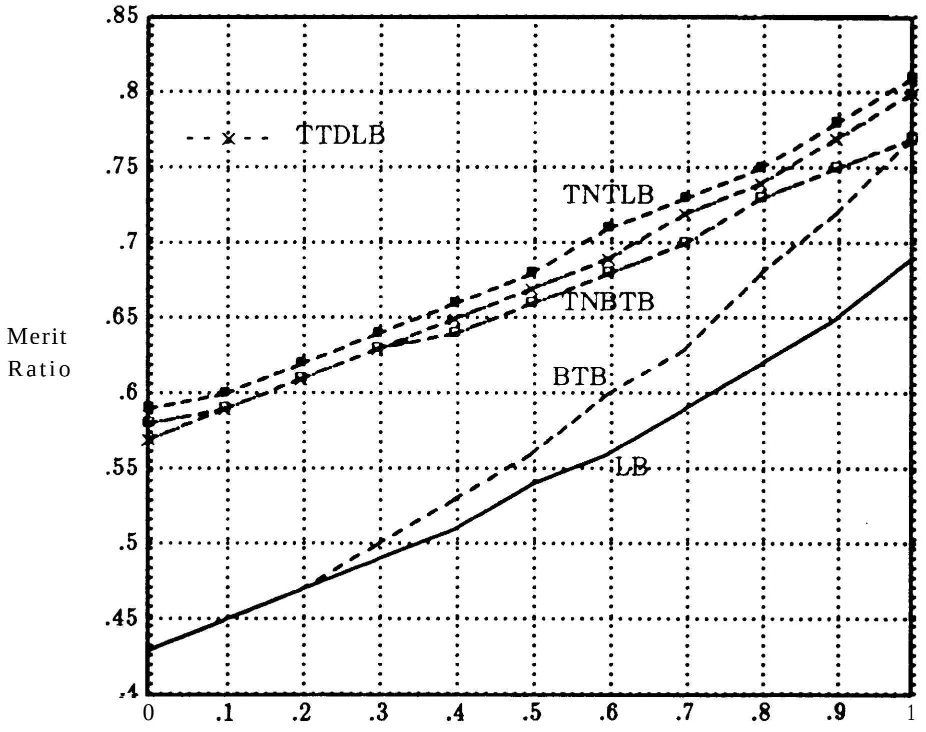


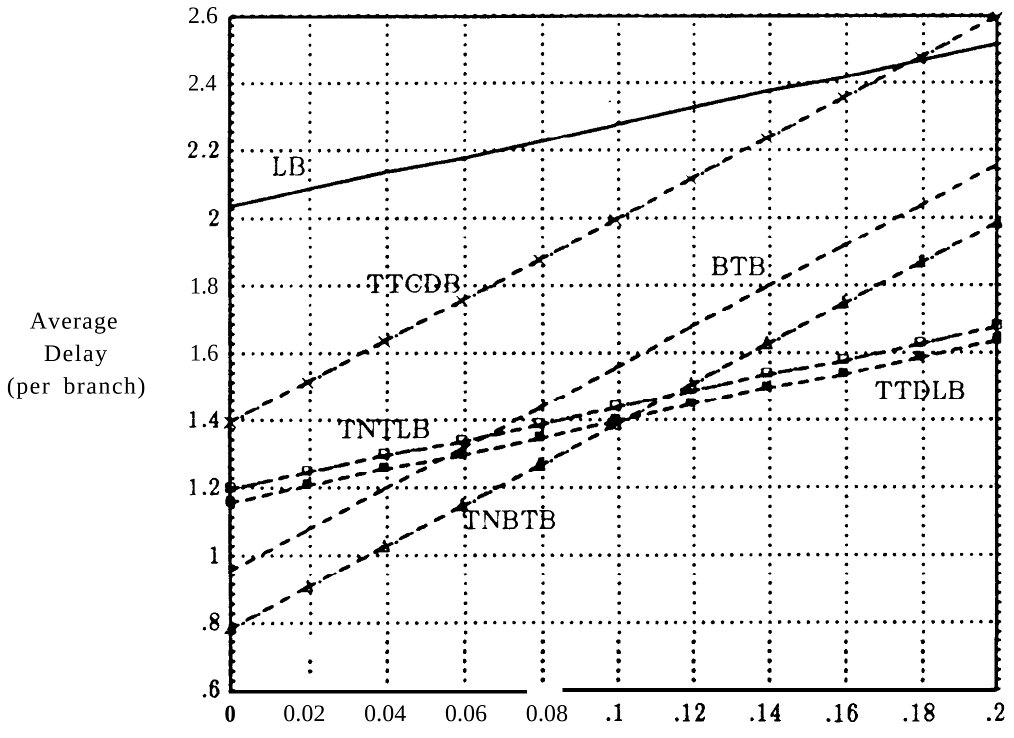
Figure 27



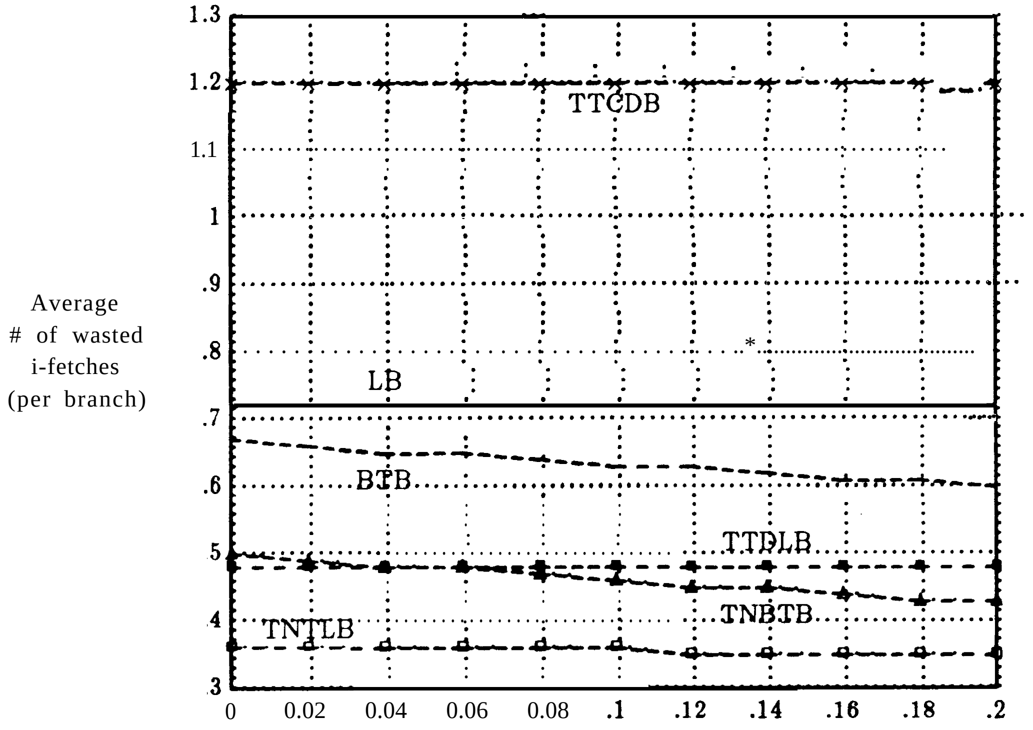
Loop/Target Buffer Hit Probability
Figure 28



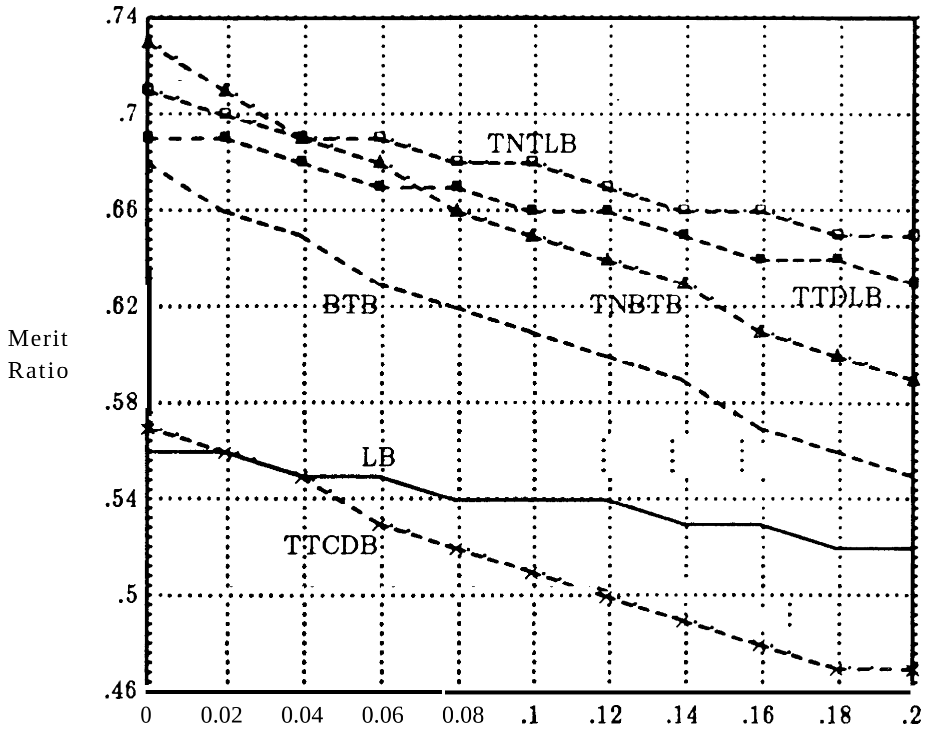
Loop/Target Buffer Hit Probability
Figure 29



Target-Fetch Freeze Probability
Figure 30



Target-Fetch Freeze Probability
Figure 31



Target-Fetch Freeze Probability
Figure 32