...branch be taken, and if so, where will it go? These problems ... have ... CPU pipeline designers, may be minimized ... a system for predicting branch behavior.

B...ch Prediction Str...gies ... Branch Target ...uffer Design

...ny K. F. L... Hewlet...
...n Jay Smith ...niver... Camorga, Berkeley

...dern h... com...er systems achieve their ...igh p...orm... ...st ... and parallelism. In parallelism, o... ...portant aspects is the pipelining of inst... ...cessing, in which a number of operations are ...equentially. A typical sequence (...re 1) ...st of instruction fetch, instruction ...cod... ...s generation, operand fetch, execution ... Pipelining consists of executing ...structions con...rently, with each instruction ...different stage of processing (Figure 2). ...hown in Fig... ...contain up to seven ...ch ... differe... Ra... worthy ...omprehens... ...ining of ...more deta...

...ost-effe... means to ac... parallel... ...gh ... instructions ... the ... ...ifferent phas... ...et of hardware is nee... ...required ...elined machi... the ... decrease, bu... ...ea... wi... ...l reason... ...sm pro... ...age ... is seldom realize... ...st, the ...s... ...to run no faster than its ...st ...e stages have the same averag... ...dom variations in one or more of the ...he ...ooth flow of instructions. For e... ...ution ...ge of the pipeline will generally run ...for "decimal divide" than for "integer ...minimum time for an instruction to advance ...the p...e is called the pipeline stage time, or ...the machine cycle time in most machines. ...ys encountered in pipelines occur because of ...o use busy resources (e.g., conflicts in access...-port register file), or failure to have an input ...ion uses the output of the immediately previous one as part of an address calculation, but that previous instruction has not passed the execute stage of the pipeline). Cache memory misses are another major source of pipeline delay.[2]

One of the major problems in designing a CPU pipeline is to ensure a steady flow of instructions to the initial stage of the pipeline. Such a flow can be either impeded or interrupted for two reasons. The first is that the memory access time is so long that a request by the instruction fetch stage for another instruction will not be satisfied in one PST. The second is that a change in the expected sequence of instructions, due to a branch for example, will cause the contents of part of the pipeline to be discarded, and the pipeline to be reloaded. This "branch problem" is closely related to the timely fetch of instructions, since the penalty for a branch will depend on the time required to fetch the branch target.

The branch problem can be explained as the "execution" of a branch instruction, which consists of causing the instruction fetch unit to select a different instruction as the next instruction to execute. Thus, considering the pipeline in Figure 1, all partially executed instructions in stages preceding execution must be discarded; an additional delay is also encountered in fetching the new, out-of-sequence instruction. Each of these two delays can seriously degrade CPU performance.

...an ... dation from ...hes in the instruction ... ...m a number of ways, which are discussed in detail later: ...o buffer(s), multiple instruction stre... prefetch branch target, data fetch targe... prepare t... anch, delayed branch, taken/not taken switch, and t... branch target buffer. The *branch targ... buffer* is a sma... sociative memory that retains the add...esses of recently ...ted branches and their targets (dest...ations). The buffer ...ses to predict whether the branch will be taken this time, ..., what its target will be. The instruction fetch stage ...

continues by fetching instructions from the predicted target address.

This article presents a systematic approach to selecting good prediction strategies, which is based on 26 program address traces grouped into four IBM 370 workloads (scientific, commercial, compiler, supervisor) and CDC 6400 and DEC PDP-11 workloads. Results show the effectiveness of various prediction strategies, the number of past branches that should be remembered, the amount of state required for each, and the effect of workload and branch type. Improvements of 5 to 20 percent can be expected in CPU performance when a branch target buffer is installed. Issues relating to the implementation of real branch target buffers are also considered, as are alternative approaches.

Existing approaches to the branch problem

Loop buffers. A loop buffer is a small, very high speed buffer maintained by the instruction fetch stage of the pipeline. A single loop buffer contains one set of sequential instructions, while multiple-loop buffers contain *n* sequences, one per buffer, but the contents of the various buffers need not be contiguous with each other. The loop buffer functions in two ways. First, it contains instructions sequentially ahead of the current instruction fetch address; thus, instructions fetched in sequence will be available without the usual memory access time. Second, it will recognize when the target of a branch falls within its contents (including backward branches) and will deliver those instructions without accessing memory. All instructions for a loop could be fetched entirely from this buffer; hence, the name "loop buffer." Among the machines using a loop buffer are the CDC Star-100 with a buffer of 256 bytes,[3] the CDC-6600 with 60 bytes,[4] and the CDC-7600 with 12 60-bit words.[5]

The Cray-1 maintains four loop buffers,[6] and replaces their contents in a FIFO manner. (This structure can also be considered to be a four-block, associative instruction cache.) The idea here is that a loop may consist of several noncontiguous instruction sequences and may be better captured this way than by a mechanism that permits only one sequence.

Multiple instruction streams. A normal pipeline suffers a branch penalty because for a conditional branch it must make a choice—the instruction fetch unit must fetch either the next sequential instruction or the branch target. A brute force approach to this problem is to replicate the initial stages of the pipeline so that both the sequential instruction and the potential branch target can be fetched, decoded, and processed. However, this approach gives rise to three problems. The first is that the branch target cannot be fetched until its address is determined, which may require a computation, such as when a displacement is added to both a base and index register. This computation requires time even when all operands are available. Further delays may occur when operands are not available, such as when an operand is the result of an uncompleted instruction or when a memory fetch is required. Contention delays are also a problem, for example, in accessing the register file. Also, additional memory traffic is generated, further creating resource contention.[7]

The second problem in replicating the initial stages of the pipeline is that if instruction *I* is a branch instruction, then additional branch instructions may need to enter the pipeline (either part) before *I* can be resolved as taken/not taken and its target determined. Riseman and Foster[8] found that for a pipeline of typical length, more than two branches would have to be processed this way to yield a significant improvement, and the net amount of hardware required would be impractical.

The third problem is that the cost of replicating significant parts of the pipeline (including instruction fetch, instruction decode, operand address generate) is substantial, making this mechanism of questionable cost-effectiveness.

Despite these problems, a number of machines follow multiple instruction streams, including the IBM 370/168,[9] which can fetch one alternative instruction path and the IBM 3033,[10] which can pursue two alternative instruction streams. The 3033 fetches an alternative instruction stream only when the stream is predicted to be taken; the prediction depends on the branch condition mask in the instruction, the operation code, and the target address operand register. These machines do not decode the alternative instruction paths. Hughes[11] proposes that fetching alternative instruction streams be combined with predictive information from a branch target buffer so that the most likely instruction stream is decoded.

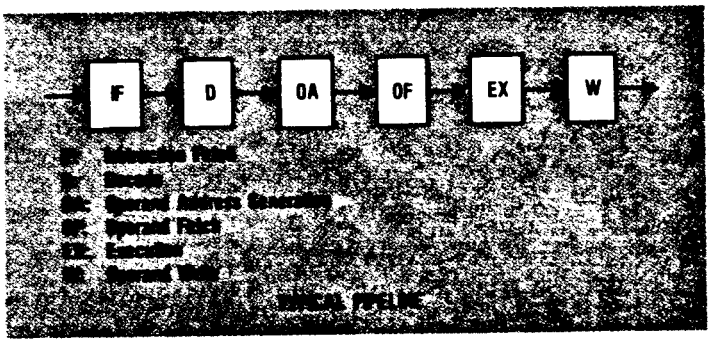


Figure 1. Typical pipeline stages in a 370-like architecture.

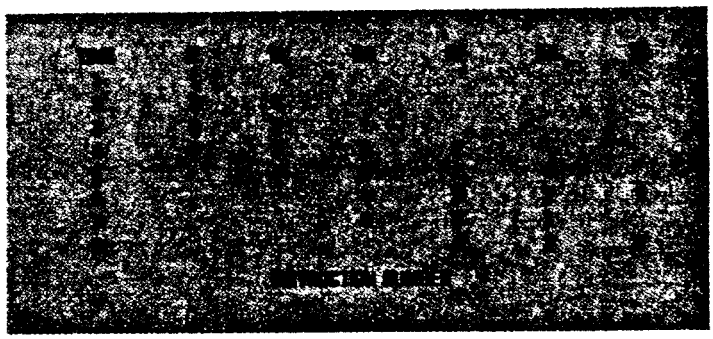


Figure 2. Typical pipeline time sequence showing the instructions executed per stage.

Prefetch branch target. Rather than replicate several initial stages of the pipeline, we can duplicate only enough logic to prefetch the branch target. That is, when a branch is recognized, a special mechanism calculates and prefetches the target of the branch; thus, if the branch is found to be taken, the target is loaded immediately into the instruction decode stage of the pipe, with no additional delay for instruction fetch.[12] Several such prefetches can be accumulated along the main instruction sequence, but since the secondary (prefetched) sequences are not decoded, no additional prefetches can be generated there.

The IBM 360/91 uses this mechanism to prefetch a double-word target.[13]

Data fetch target. In the IBM 370 architecture, the "branch conditional" instruction has the same form as the "load" or "add" (from memory) instruction; that is, the target of the branch is computed in just the same way as the memory-based operand of the load or add. The Amdahl 470 computers[14] use this feature to produce an effect very much like the target prefetch mechanism of the 360/91: the branch target is accessed as if it were an ordinary operand; if the branch is taken, the target is loaded into the instruction decode stage of the pipeline, rather than being placed in a register, as for load, or being sent to the adder, as for add.

Prepare to branch. The Texas Instruments ASC computer[15] uses two buffers into which it alternately prefetches instructions from memory. The "prepare to branch" and "load look-ahead" instructions can cause the machine to prefetch from the branch target rather than to prefetch sequentially. The effectiveness of this scheme depends on the programmer or the compiler correctly inserting these instructions.

Delayed branch. The problem with a branch is that if instruction I is a taken branch, then instruction $I + 1$ will be out of sequence, with the consequences just described. The instruction set architecture can be specified such that a branch is defined to affect the address not of instruction $I + 1$ but of instruction $I + k$. That is, consider a sequence of instructions $I1 \ldots I10$, where $I4$ is a conditional branch whose target is $I9$. Assume that branches are delayed two instruction times, making $k = 2$, and that $I4$ is a taken branch. Then the actual sequence of instructions would be: $I1$, $I2$, $I3$, $I4$, $I5$, $I6$, $I9$, $I10$.

If k, the branch delay, is equal to or larger than the number of pipeline stages preceding the stage in which the branch is executed, then the instruction fetch can almost always be given the correct address from which to fetch. (The "almost" refers to the occurrence of asynchronous events such as interrupts, which cannot be predicted from the instruction stream.)

In designing a machine to use a delayed branch, we encounter several problems. The most significant is that human programmers will find it very difficult to write code containing instructions (branches) with delayed effects. Thus, code for such a machine must be almost entirely compiler-generated, with the consequent need for a bug-free and very efficient compiler. The delayed branch, requiring a new architecture, cannot be used as a technique to speed up an existing one. In addition, not all the potential speedup of the delayed branch can be realized; it may not be possible to schedule $k - 1$ instructions after the branch.

Despite these problems, two experimental computers are actually using the delayed branch: the IBM 801, an experimental minicomputer constructed at IBM T. J. Watson Research Center, Yorktown Heights,[16] and a dedicated microprogrammed machine constructed by E. R. Berlekamp[17] to insert and remove error-correcting codes from signal transmissions. It has been proposed for the RISC computer.[18]

Taken/not taken switch. As we will show later, we can predict with good accuracy whether or not a branch will be taken. A prediction mechanism that specifies whether a branch is or is not likely to be taken is called the taken/not taken switch. The idea is that one or more bits are associated with every instruction in the cache memory. The setting of these bits determines whether the branch is predicted to be taken or not. After the branch is resolved, the values of the bits may be reset in the cache to reflect the prediction for the next time.

In the taken/not taken switch proposed for the S-1 computer,[19,20] two bits are stored with each instruction. One bit specifies whether a jump should be predicted (the Jump bit) and the other tells whether the last prediction was wrong (the Wrong bit). Two wrong predictions in a row cause the Jump bit to be changed. As we note later, this mechanism still encounters delays due to target address computation and the out-of-sequence fetch. Widdoes[19] discusses the effectiveness of the prediction algorithm in more detail, and Liles and Willner[21] propose a version of this scheme.

Look-ahead resolution. Another proposed solution to the branch problem is to place extra logic in the pipeline so that an early stage of the pipeline can resolve a branch whenever the condition code affecting a conditional branch has already been determined.[11] Rao provides further detail on this method.[22]

Branch target buffer. The branch target buffer (Figure 3) is a small cache memory associated with the instruction fetch stage of the pipeline. The BTB retains three tuples, each of which contains the address of a previously executed instruction, information that permits a prediction as to whether or not the instruction branch will be taken, and the most recent target address for that branch. The BTB functions as follows: the instruction fetch stage compares the instruction address against the instruction addresses in the BTB. If there is a match, then a prediction is made as to whether the branch is likely to be taken. If the prediction is that the branch will occur, then the target address field is used to select the next instruction fetch address. When the branch is actually resolved, at the execute stage, the BTB can be updated with the corrected prediction information and target address. Since the BTB can be used for every instruction

fetch, it can have as many predictions as there are uncompleted instructions in the pipeline.

The major optimization problem in the design of a BTB is the selection of the algorithm that predicts whether or not the branch will be taken. How large the BTB will or should be and how it should be organized (e.g., set associative or hashed) are also issues. Holgate and Ibbett[23] have studied the BTB design effectiveness for the MU-5, which actually implements a branch target buffer, roughly of the type described. Losq[24] proposes the use of the BTB, and Smith[25] examines a number of BTB designs using traces for the CDC Cyber 170 computer. Results from these studies are similar to our own, but here we consider three different machine architectures (IBM 370, DEC PDP-11, CDC 6400), and prediction strategies are examined much more systematically.

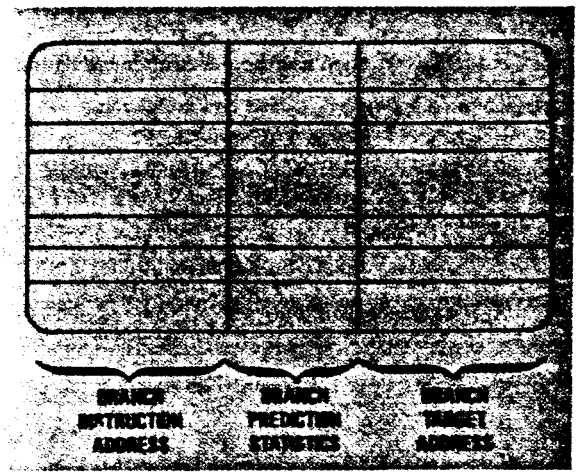
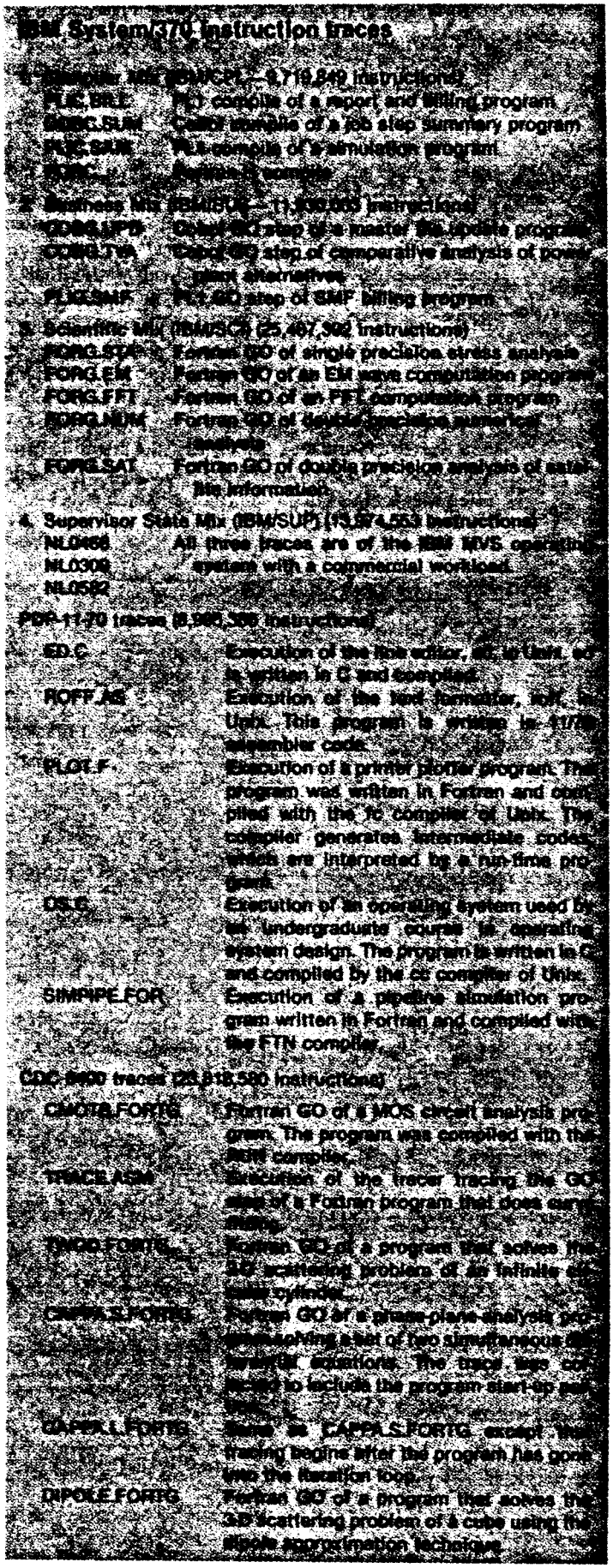


Figure 3. Branch target buffer organization.

Methodology and data

There is now no statistically acceptable model to characterize any aspect of program behavior (although much research has been done in paging and memory management).[26,27] For the design and evaluation of branch target buffers, we still need a model of when branches occur, whether or not they will be taken, and whether or not the branch target will change. Because no existing model can now predict these things accurately, our research is based on the thorough analysis and use for trace-driven simulation of program address traces.

Data. We have 26 program address traces (see box at right), grouped into six workloads. Four workloads are for the IBM 370 architecture and consist of compiler executions (PL/I, Cobol, Fortran-H), business programs (Cobol, PL/I), a scientific mix (Fortran), and supervisor state set of traces (MVS operating system). Six traces form the DEC PDP-11 workload, and six more make up the CDC 6400 workload.



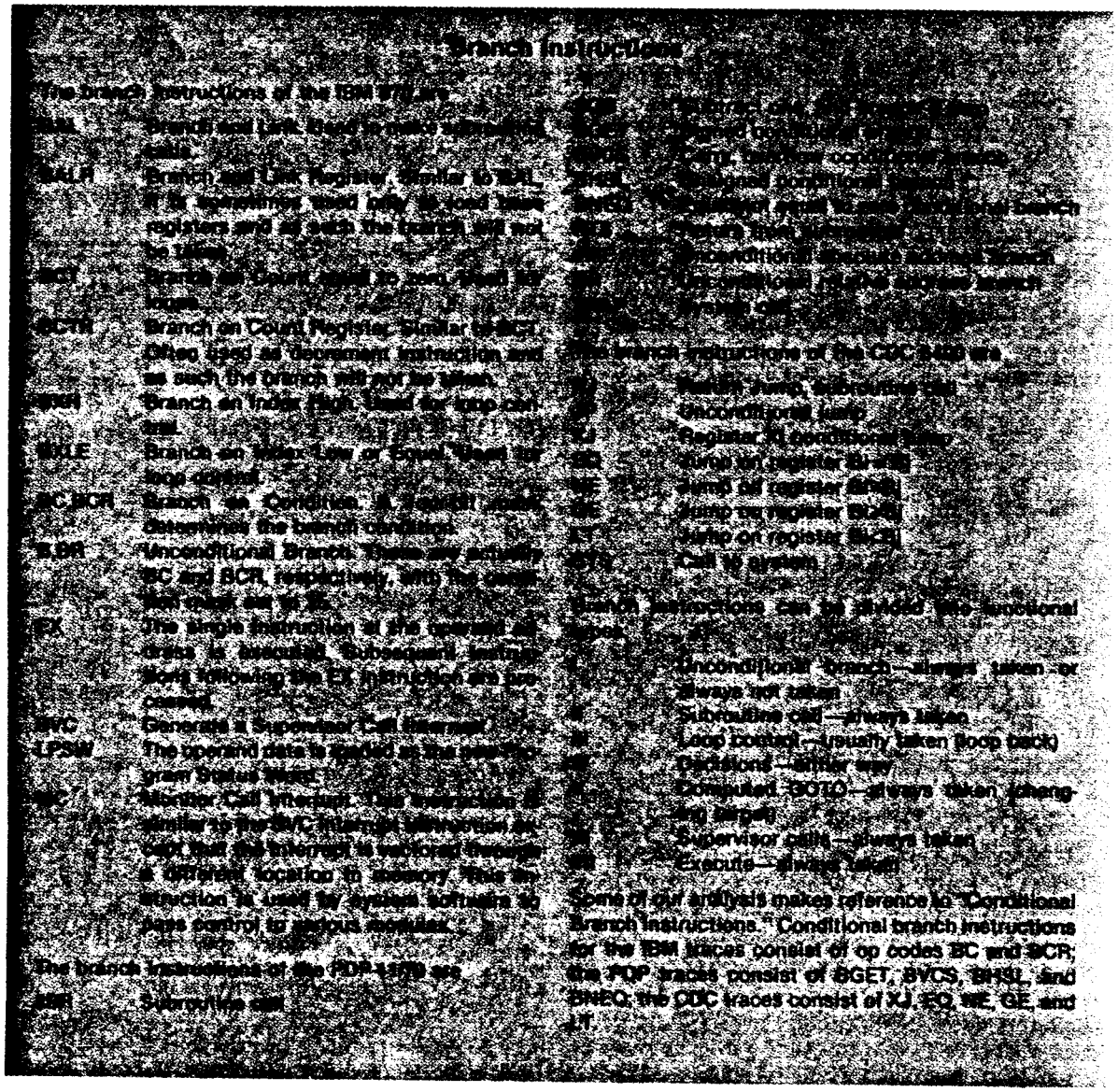
From each program trace, we extracted the branch instructions, along with their targets, addresses, sequence numbers, and operation codes. All analysis was based on this extraction.

The large number of traces used in this research and the grouping of them into workloads serves several purposes. First, the large number of individual traces and the use of several of them in each workload should give representative behavior; no individual trace, no matter how peculiar, can significantly throw off the overall results. Conversely, the use of workloads, rather than a grand average, shows the variation to be expected from the different job mixes experienced at different computer centers, on different machines, and at different times of the day. Certain workloads are known to have different instruction mixes; business programs use many more storage-to-storage, or string, operations on the IBM 370 than scientific programs. Conversely, the scientific programs have far more floating point operations. If such differences impact the efficiency of a branch target buffer, our study will show these effects. Similarly, the use of traces from three very different machine architectures will indicate whether the results are sensitive to the instruction set architecture.

Some of our studies show results for various specific machine instructions, and branch instructions for each machine are given in the box below. Some studies are limited to conditional branches only; the instructions considered to be conditional branches are also listed.

Methodology. Trace-driven simulation is a technique by which a trace is recorded of the operation of some system. That trace is then used to drive a model of the system that allows us to vary different parameters or features of interest. If the variation does not affect the validity of the trace, then the trace-driven simulation can accurately predict the effect of changes in the system.

We use program address traces in two different ways. First, we examine them and measure various features of interest; for example, the frequency of taken and not taken branches. We then use these measurements as one



basis from which we can formulate branch-buffering strategies. The traces are then used to evaluate designs for a branch target buffer.

Branch behavior

Before presenting actual measurements of branch behavior, we need to consider what we can expect. There will be several types of branches: loop-control branches, which are usually taken and go backward; branches used as part of IF/THEN/ELSE logical constructs, which always go forward and may or may not have a consistent behavior pattern; branches used for subroutine calling, which will always be taken; branches used to load registers, which are never taken; and branches used as "no-ops," which are never taken. While for most of these, we can predict likely behavior, the relative frequency of each makes reasoning out overall average behavior extremely difficult. Thus, we rely almost exclusively on data analysis and empirically derived prediction algorithms.

Taken/not taken and branch frequency by opcode. For each trace, we show the overall probability of a branch being taken or not taken and the ratio *r* of branch instructions to all instructions in the trace (Table 1). Two features are important: first, branches are taken twice as often as not; thus by just guessing that branches are always taken, we are right 60 to 70 percent of the time. (In Smith's study,[25] the range over six traces was 57 to 99 percent, with an average of 76.7 percent.) Variation among workloads is moderate, and for all workloads, branches are taken most of the time.

The probability that a branch is of a specific operation code is shown in Table 2 for each workload. For IBM 370 workloads, note the significant variation in the frequencies of the various operation types.

Table 3 shows the probability that a branch is taken for each operation code. Unconditional branches are always either taken or not taken, but BALR is sometimes used to set up the base registers, and so is not taken. Those used for indexing are usually taken, but BCTR is generally not taken because it is often used as a decrement instruction.

Dynamic branch behavior. Not all branches are executed with the same frequency, so much of our ability to predict branches relies on the fact that because some branches are executed many times, we can make a good guess as to what will happen next. Before examining this approach further, we need to define *static branch instructions* and *dynamic branch instructions.*

The first type refers to the individual branch instructions found in a program. For a given program, the number of these branches is fixed and can be counted by looking at the program. The second type refers to the branch instructions found in the trace of a program. A static branch instruction can occur more than once as a dynamic branch instruction, and every time a static branch instruction is executed, a new dynamic branch is formed.

In Figure 4, we show the probability distribution for each workload for the number of times a static branch

Table 1.
Fraction of branches, taken *T* and not taken *N* and fraction of branches overall *r*.

	IBM CPL	IBM BUS	IBM SCI	IBM SUP	DEC PDP11	CDC 6400	AVERAGE
T	0.640	0.657	0.704	0.540	0.738	0.778	0.676
N	0.360	0.343	0.296	0.460	0.262	0.222	0.324
r	0.317	0.189	0.105	0.376	0.388	0.079	0.242

Table 2.
Frequency of branch types.

OP CODE	IBM CPL	IBM BUS	IBM SCI	IBM SUP	OP CODE	DEC PDP11	OP CODE	CDC 6400
BR,B	0.222	0.243	0.254	0.138	JSR	0.111	RJ	0.049
BAL	0.056	0.036	0.013	0.036	SOB	0.008	JP	0.017
BALR	0.036	0.050	0.079	0.065	BGET	0.113	XJ	0.560
BCT	0.024	0.013	0.027	0.016	BVCS	0.030	EQ	0.157
BCTR	0.022	0.050	0.006	0.019	BHSL	0.031	NE	0.199
BXH	0.004	0.000	0.000	0.000	BNEQ	0.278	GE	0.000
BXLE	0.032	0.000	0.188	0.003	RTS	0.074	LT	0.003
BC	0.544	0.521	0.318	0.674	JMP	0.190	SYS	0.015
BCR	0.051	0.081	0.112	0.034	BR	0.162		
EX	0.009	0.005	0.003	0.005	TRAP	0.002		
SVC	0.000	0.001	0.000	0.001				
LPSW	0.000	0.000	0.000	0.005				
MC	0.000	0.000	0.000	0.005				

Table 3.
Probabilities of branch taken by branch type (blanks mean instruction is not in that trace).

OP CODE	IBM CPL	IBM BUS	IBM SCI	IBM SUP	OP CODE	DEC PDP11	OP CODE	CDC 6400
BR,B	1.000	1.000	1.000	1.000	JSR	1.000	RJ	1.000
BAL	1.000	1.000	1.000	1.000	SOB	0.448	JP	1.000
BALR	0.659	0.555	0.850	0.531	BGET	0.330	XJ	0.604
BCT	0.584	0.899	0.857	0.713	BVCS	0.155	EQ	1.000
BCTR	0.007	0.173	0.000	0.207	BHSL	0.496	NE	1.000
BXH	0.404				BNEQ	0.495	GE	0.848
BXLE	0.865	0.994	0.865	0.522	RTS	1.000	LT	0.000
BC	0.462	0.571	0.342	0.415	JMP	1.000	SYS	1.000
BCR	0.539	0.348	0.647	0.584	BR	1.000		
EX	1.000	1.000	1.000	1.000	TRAP	1.000		
SVC	1.000	1.000	1.000	1.000				
LPSW				1.000				
MC				1.000				

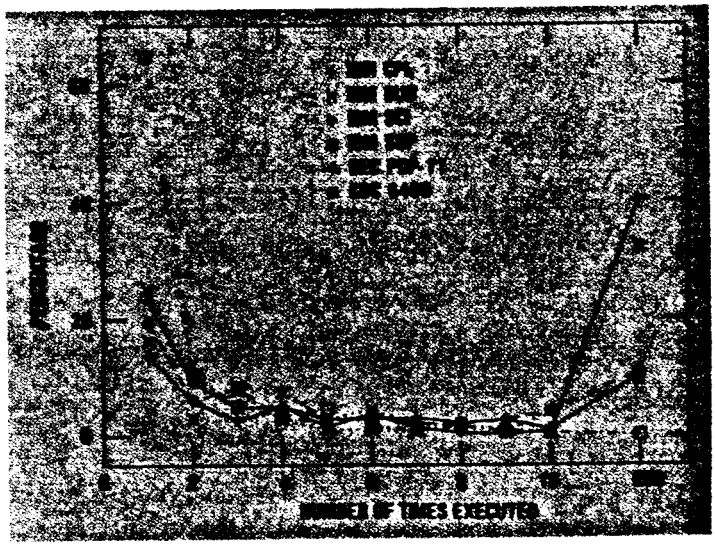


Figure 4. Percentage of branch instructions executed *N* times for each of six workloads.

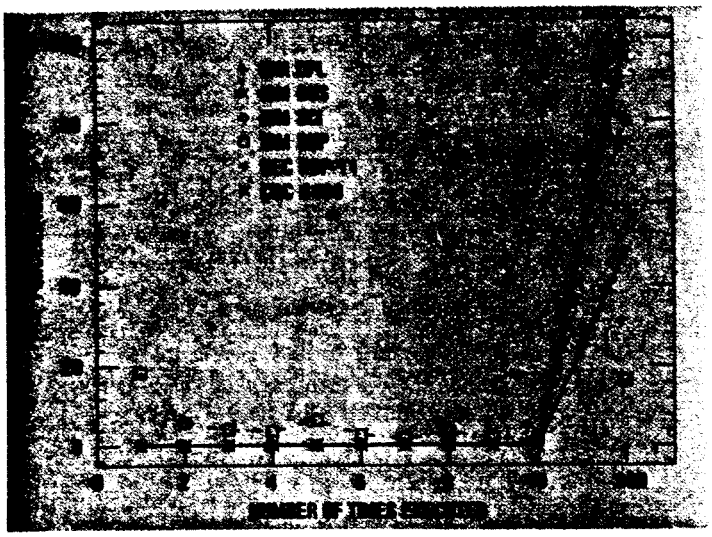


Figure 5. Percentage of branch instructions executed *N* times weighted by *N*.

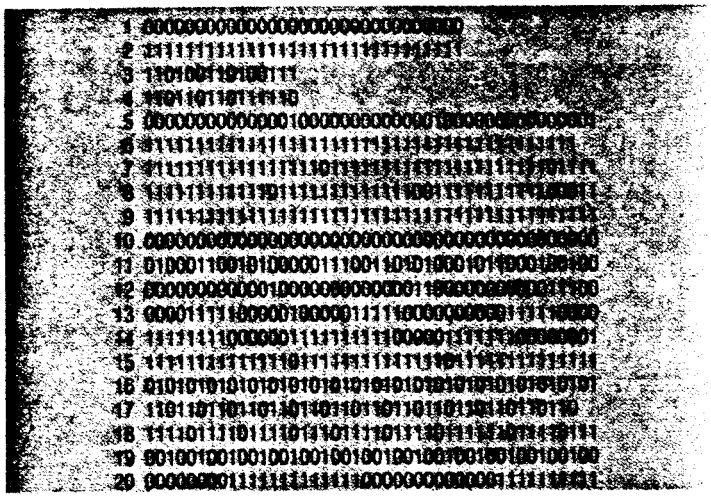


Figure 6. Sample sequences of taken and not taken branches (0 = not taken, 1 = taken).

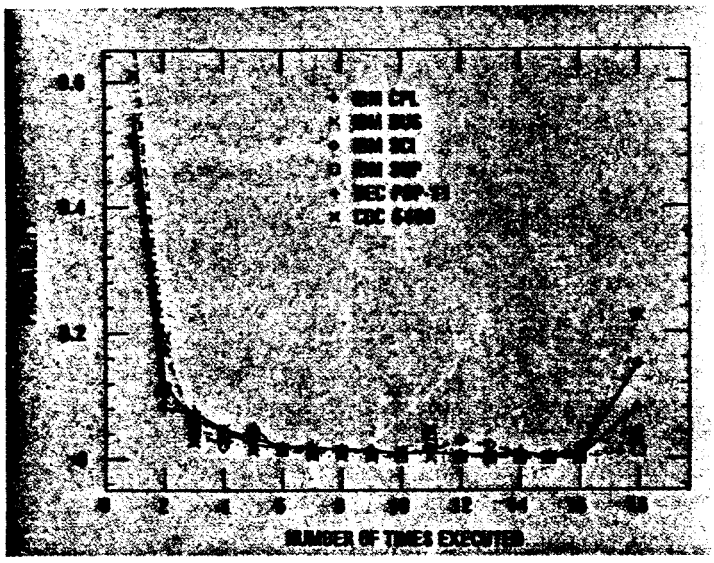


Figure 7. Distribution of the number of times that a conditional branch has the same result.

occurs as a dynamic branch. Figure 5 shows the probability that a dynamic branch is due to a static branch executed N times. The large bulk of dynamic branches occur for frequently executed static branches; for example, 23.4 percent of the static branches in the IBM CPL mix get executed only once, but they account for only 0.5 percent of the dynamic branches. On the other hand, 10.4 percent of the static branches executed over 200 times make up 48.2 percent of the dynamic branches.

Many of our predictions as to whether a branch will be taken are contingent on the branch's past behavior (taken/not taken). To illustrate such branch behavior, Figure 6 shows some sequences of taken/not taken for a number of branches. For many branches, there are long sequences of either taken or not taken; it is less common to see an alternation. We call such a sequence a *run*, or a sequence of identical behavior (taken, not taken, taken with a changed target) of a static branch as it gets executed many times. For example, the sequence of takens T and not takens N, $TTTTTNNTTTTNTNNN$, consists of run lengths of 5, 2, 4, 1, 1, etc. Figures 7 and 8 show the distributions of run lengths for conditional branches only and all branches, respectively. The same data are shown weighted by the run length in Figures 9 and 10. (That is, Figures 9 and 10 show the probability that a given dynamic branch is an element of a run N branches long.) As the figures show, most branches occur as parts of long runs.

Branch clustering. We have described one method of coping with the branch problem, called multiple instruction streams, which involved recognizing branches at the instruction decode step of the pipeline, and then fetching and decoding both the taken and not taken outcomes of the branch. As noted, one difficulty with that solution was that a large number of closely clustered branches could occur, making it impossible to follow all 2^k paths possible from k branches. A measure of the size of k appears in Figures 11 and 12. The figures show the probability that in H sequential instructions ($H = 10$ and $H = 6$, respectively), there are k branches. If the pipeline is long enough (and 6 and 10 are typical numbers for high-speed machines), then there is a significant probability that more than one branch is unresolved at any one time.

Branch prediction

A number of the solutions to the branch problem attempt to predict whether or not a branch will be taken. The general problem can be stated as what is the value of $F(x1,x2,\ldots)$, where F is the probability that a branch is taken, and $x1, x2,\ldots$ are parameters on which F may be reasonably conditioned. If $F(x1,x2,\ldots) > 0.5$, then we predict that a branch will occur; if less than 0.5 we predict that it will not. (If the cost of commission errors is not equal to that of omission errors, the best figure for deciding to predict a branch may not be equal to 0.5. We discuss this issue later.) Of particular interest is $x1 =$ operaton code, and $x2 =$ execution history of this branch. We can continue with other factors (for $x3$, $x4$, etc.) such as other dynamic branches that precede the current

...namic branch (and their execution behavior),[28] other ...namic instructions that precede the current dynamic ...struction, the source language of the program, and the ...rection of the branch (e.g., forward/ back[25]). For ex...ple, certain instruction sequences will generally in...ce a taken branch; others will almost always fall ...ough.

...solution to the branch problem must be imple...ented in hardware, since it is part of the pipeline and ...st execute at machine cycle speeds. For that reason, ...complexity of practical schemes is very limited, and ...consider only predictions that depend solely on the ...ration code $F(x1)$ and those that depend only on the ...ory of the branch $F(x2)$.

The other aspect of branch prediction concerns ...wledge of the target address, since delays are en...ntered even for a correctly predicted taken branch ...n the target address is not immediately known.

Prediction based on operation code. In Tables 2 and 3, ...now the probability that a branch was of a specific ...de, and the probability that the branch with that op ...de would be taken. These two tables can be easily com...ed (Table 4) to yield the probability of whether or not ...nch will be taken given only the op code. Note that ...the IBM CPL mix, the prediction accuracy rises from ...percent (assume all branches are taken) to 66.2 per... (assume that only BR, B, BAL, BALR, BCT, BX...E, BCR, EX, and SVC are taken; all others never ...en). While this 2.2-percent improvement is helpful, ...shall see that it is considerably less than what can be ...tained by predictions based on branch history. ...ith[25] gives a range of accuracy for op-code-based ...dictions of 65.7 to 99.4 percent, with a mean of 86.7 ...cent.)

Prediction based on branch history. Prediction based ...branch history uses the previous sequence of taken/ ...t taken for each branch to predict whether or not the ...nch will be taken next time it occurs. The most ...erful predictor, of course, uses the entire history of ...e branch to predict the next choice, but such a predic...or is infeasible because of the large possible number of ...ch past sequences. Consequently the problem becomes ...for a given amount of history, what prediction accuracy ...can be otained, and what is the most desirable amount of ...history to retain, given all cost and performance trade...offs? The basic data for this evaluation are presented in Tables 5 and 6, where we show the observed probability ...of all possible sequences of five taken/not taken events ...($y1,y2,y3,y4,y5$) for conditional branches and all ...branches, respectively.

Table 4.
Probability of correct branch prediction given only op code, and assuming branch is always either taken or not taken, based on op code.

IBM CPL	IBM BUS	IBM SCI	IBM SUP	DEC PDP11	CDC 6400
0.662	0.692	0.710	0.552	0.798	0.778

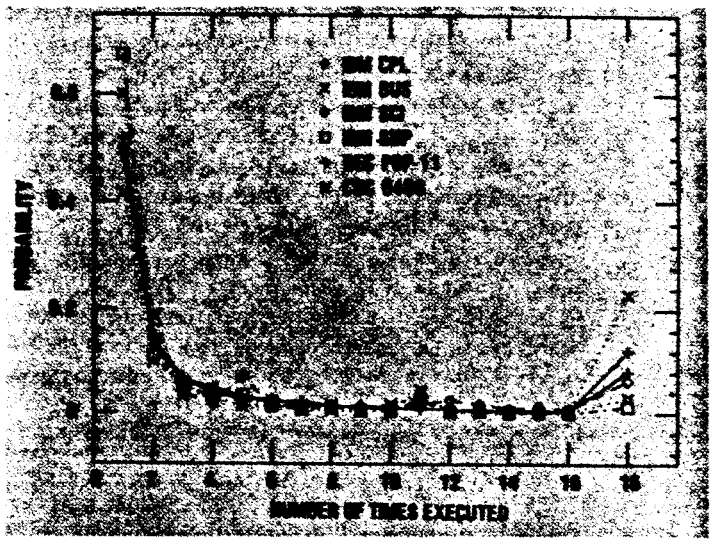


Figure 8. Distribution of the number of times that any type branch has the same result.

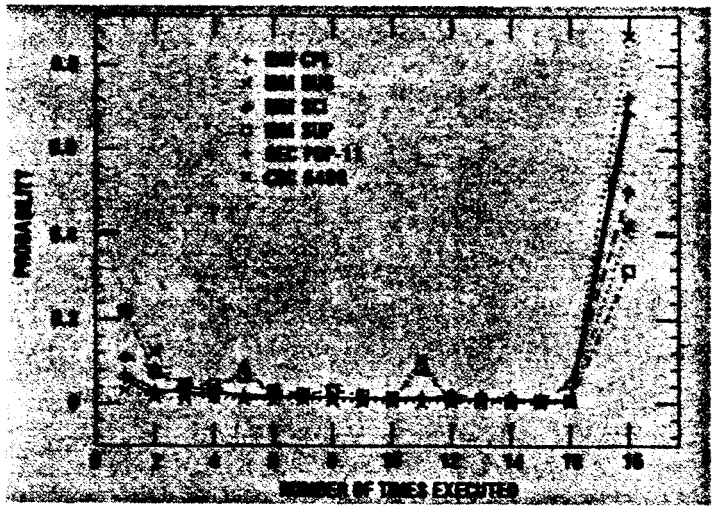


Figure 9. Distribution of the number of times that a conditional branch has the same result, weighted by run length.

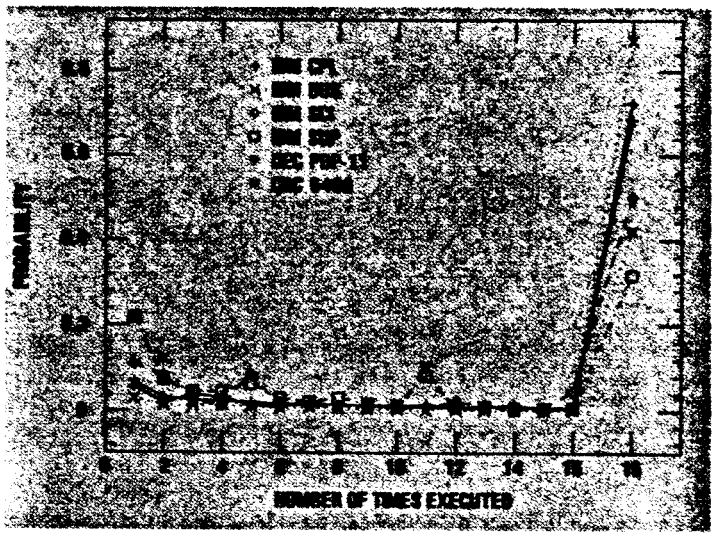


Figure 10. Distribution of the number of times that any type branch has the same result, weighted by run length.

The data in Tables 5 and 6 may be used for prediction in the following manner: whenever the probability $F(y1,y2,y3,y4,T)$ is greater than $F(y1,y2,y3,y4,N)$, the branch should be predicted as taken and when less than, the prediction should not be taken (where $y1,y2,y3,y4$ is the sequence of the four previous dynamic occurrences of this static branch). Predictions based on the previous three events, $F(y2,y3,y4,T)$ and $F(y2,y3,y4,N)$, can be computed by noting that $F(y2,y3,y4,N) = F(T,y2,y3,y4,N) + F(N,y2,y3,y4,N)$. Predictions based on the previous two, one, or zero branches can be similarly derived. Table 7 shows the accuracy of such predictions, where each is based only on the values of $F(yi)$ for that workload. (For one previous branch, Smith's success rate[24] was from 76.2 to 98.9 percent with a mean of 90.4 percent.)

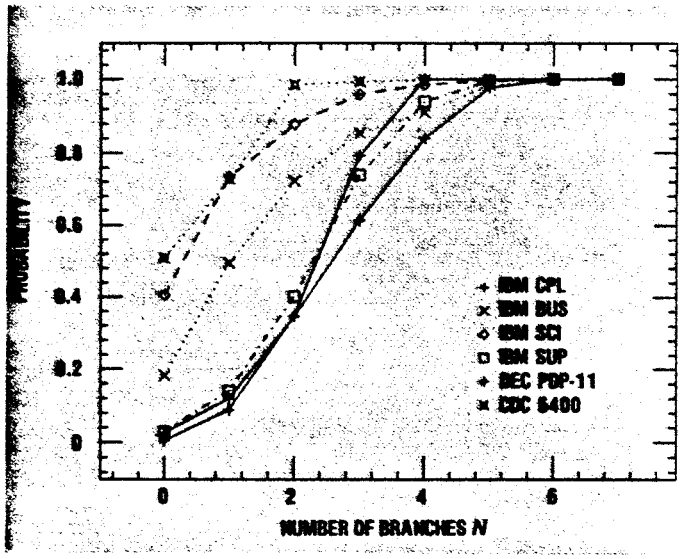


Figure 11. Probability of *N* or fewer branches in 10 consecutive instructions.

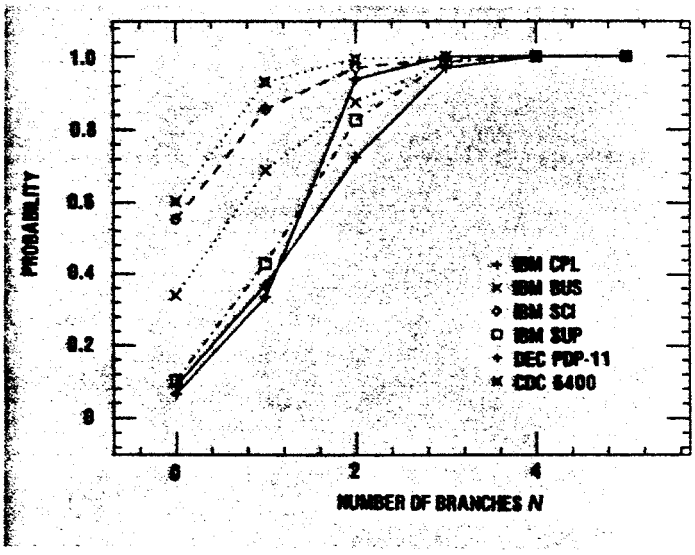


Figure 12. Probability of *N* or fewer branches in six consecutive instructions.

We can create a composite predictive strategy; that is, a prediction based on $F(yi)$, where $F(yi)$ is computed over all six workloads used, rather than for just the workload in question. This strategy is much more valid, since varying the predictive strategy on a real computer is not likely to be cost-effective (depending on the program running). In any case, as Table 8 shows, the predictive accuracy is almost identical to that shown in Table 7.

A number of interesting observations can be made from Tables 7 and 8. First, the predictive accuracy approaches very closely to its maximum with one, two, or three preceding branches used for prediction. Increasing the amount of history to four or five branches does not seem to add accuracy.

Second, the predictive accuracy for as few as two preceding branches is from 83.4 to 97.5 percent, which is much higher than the accuracy using only the branch type, and no branch history (Table 4). Finally, the effectiveness of prediction varies significantly among the workloads. Most striking is the variation of 83.9 to 97 percent between the IBM SUP and the IBM BUS workloads, both of which are for the same architecture. [We believe that the lower prediction success rate for the IBM/SUP workload is due to the low probability that a branch is executed repeatedly (see Figure 4). This low

Table 5.
Distribution of five consecutive executions (conditional branches).

HISTORY	IBM CPL	IBM BUS	IBM SCI	IBM SUP	DEC PDP11	CDC 6400
NNNNN	0.407	0.414	0.437	0.422	0.491	0.170
NNNNT	0.013	0.006	0.014	0.005	0.011	0.008
NNNTN	0.012	0.004	0.014	0.005	0.012	0.006
NNNTT	0.004	0.003	0.005	0.003	0.002	0.002
NNTNN	0.013	0.005	0.019	0.005	0.012	0.008
NNTNT	0.003	0.001	0.005	0.003	0.001	0.003
NNTTN	0.002	0.001	0.004	0.002	0.001	0.003
NNTTT	0.004	0.002	0.004	0.004	0.003	0.003
NTNNN	0.018	0.008	0.019	0.021	0.014	0.006
NTNNT	0.005	0.002	0.010	0.004	0.001	0.006
NTNTN	0.029	0.017	0.026	0.005	0.005	0.025
NTNTT	0.008	0.005	0.006	0.026	0.004	0.002
NTTNN	0.003	0.001	0.004	0.003	0.001	0.003
NTTNT	0.003	0.001	0.014	0.003	0.003	0.044
NTTTN	0.004	0.001	0.002	0.002	0.001	0.003
NTTTT	0.015	0.013	0.020	0.020	0.012	0.020
TNNNN	0.018	0.009	0.017	0.034	0.012	0.008
TNNNT	0.003	0.002	0.005	0.003	0.004	0.001
TNNTN	0.004	0.002	0.010	0.003	0.001	0.005
TNNTT	0.003	0.001	0.003	0.003	0.001	0.003
TNTNN	0.011	0.006	0.010	0.029	0.003	0.003
TNTNT	0.017	0.010	0.016	0.021	0.007	0.025
TNTTN	0.003	0.001	0.014	0.004	0.002	0.044
TNTTT	0.015	0.012	0.018	0.021	0.016	0.019
TTNNN	0.003	0.002	0.004	0.002	0.001	0.003
TTNNT	0.003	0.001	0.004	0.003	0.001	0.002
TTNTN	0.003	0.000	0.003	0.002	0.005	0.002
TTNTT	0.011	0.009	0.027	0.004	0.014	0.061
TTTNN	0.004	0.002	0.004	0.002	0.002	0.003
TTTNT	0.011	0.008	0.016	0.003	0.017	0.019
TTTTN	0.011	0.009	0.018	0.004	0.012	0.019
TTTTT	0.338	0.442	0.228	0.341	0.320	0.471
NNNNN + TTTTT	0.745	0.856	0.665	0.763	0.811	0.641

Table 6.
Distribution of five consecutive executions (all types).

HISTORY	IBM CPL	IBM BUS	IBM SCI	IBM SUP	DEC PDP11	CDC 6400
NNNNN	0.275	0.310	0.196	0.378	0.230	0.129
NNNNT	0.008	0.004	0.005	0.004	0.004	0.007
NNNTN	0.008	0.003	0.005	0.004	0.004	0.006
NNNTT	0.003	0.002	0.002	0.003	0.001	0.002
NNTNN	0.008	0.003	0.008	0.003	0.005	0.007
NNTNT	0.002	0.001	0.002	0.004	0.000	0.003
NNTTN	0.002	0.015	0.002	0.002	0.000	0.003
NNTTT	0.003	0.002	0.002	0.002	0.001	0.003
NTNNN	0.012	0.006	0.008	0.017	0.005	0.005
NTNNT	0.003	0.001	0.005	0.003	0.001	0.004
NTNTN	0.027	0.020	0.017	0.005	0.005	0.048
NTNTT	0.009	0.008	0.007	0.036	0.002	0.003
NTTNN	0.001	0.000	0.002	0.002	0.000	0.003
NTTNT	0.002	0.001	0.012	0.002	0.001	0.040
NTTTN	0.002	0.001	0.002	0.002	0.001	0.002
NTTTT	0.014	0.012	0.030	0.024	0.004	0.017
TNNNN	0.011	0.006	0.007	0.028	0.004	0.007
TNNNT	0.002	0.001	0.002	0.003	0.001	0.001
TNNTN	0.003	0.001	0.005	0.003	0.001	0.004
TNNTT	0.001	0.001	0.002	0.003	0.000	0.003
TNTNN	0.007	0.005	0.005	0.024	0.001	0.003
TNTNT	0.016	0.012	0.013	0.028	0.005	0.046
TNTTN	0.002	0.001	0.012	0.003	0.001	0.040
TNTTT	0.014	0.013	0.030	0.029	0.005	0.018
TTNNN	0.002	0.002	0.002	0.002	0.001	0.003
TTNNT	0.001	0.000	0.002	0.002	0.001	0.002
TTNTN	0.002	0.001	0.002	0.002	0.001	0.001
TTNTT	0.008	0.007	0.036	0.004	0.004	0.055
TTTNN	0.002	0.001	0.002	0.002	0.001	0.002
TTTNT	0.008	0.007	0.027	0.003	0.004	0.016
TTTTN	0.008	0.007	0.027	0.003	0.004	0.017
TTTTT	0.534	0.561	0.521	0.384	0.702	0.500
NNNNN + TTTTT	0.809	0.871	0.717	0.762	0.932	0.629

probability is to be expected in supervisor code, in which loops are relatively less frequent.]

Prediction based on nonuniform history retention. Tables 7 and 8 give the effectiveness of branch prediction when prediction is based on exactly the n preceding executions of the branch in question, and whether that branch was taken or not taken. These n preceding executions may be remembered in the branch target buffer with n bits, those n bits representing the 2^n possible sequences of taken/not taken.

Given that n bits are available to use in predicting the next branch, the bits need not be allocated to show the past n executions, but can be used to record a state that does not map into the precise history. That is, given a state $S(i)$ (for the branch in question) at time i, we have a function $G(S(i))$ that yields the prediction T or N, and a mapping $E(S(i), T/N) \rightarrow S(i+1)$ that maps the current state $S(i)$ and whether the branch is actually taken into the next state $S(i+1)$. Thus, the prediction algorithm can be specified by giving n (2^n states), the function G and the mapping E. For example, Figure 13 shows the algorithm that uses the past two executions to predict the next; the effectiveness of this method is shown in Table 8 in the line labeled "2." In Figure 13, the states are labeled with the their history (as a name) and the prediction in

Table 7.
Percentage of correct guesses, using n past branches and conditional probabilities drawn from only given trace.

n	IBM CPL	IBM BUS	IBM SCI	IBM SUP	DEC PDP11	CDC 6400
0	64.1	64.4	70.4	54.0	73.8	77.8
1	91.9	95.2	86.6	79.7	96.5	82.3
2	93.3	96.5	90.8	83.4	97.5	90.6
3	93.7	96.7	91.2	83.5	97.7	93.5
4	94.5	97.0	92.0	83.7	98.1	95.3
5	94.7	97.1	92.2	83.9	98.2	95.7

force, and each edge shows the transition (mapping E) from state to state depending on whether the branch was taken or not taken.

We can suggest mappings E and functions G other than those based on the last n executions of the branch. Figure 14, for example shows an algorithm in which two errors are required to change the prediction. That is, when the current prediction is N and the last two

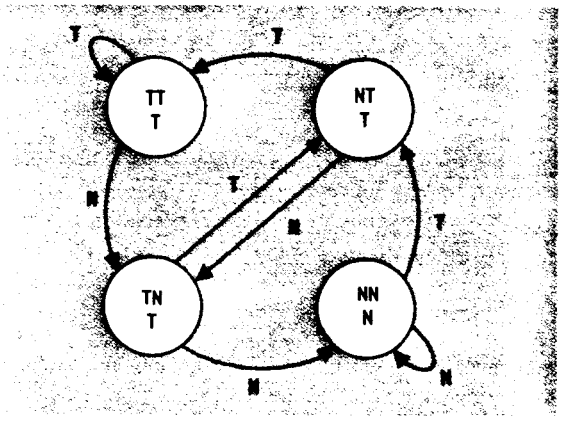


Figure 13. State diagram for branch predictor. The state name (top line) is the history of the last two dynamic occurrences of this branch followed by the prediction (bottom line). *TT* means both were taken, and *T* implies predict taken. The label on each arrow is the result of the branch.

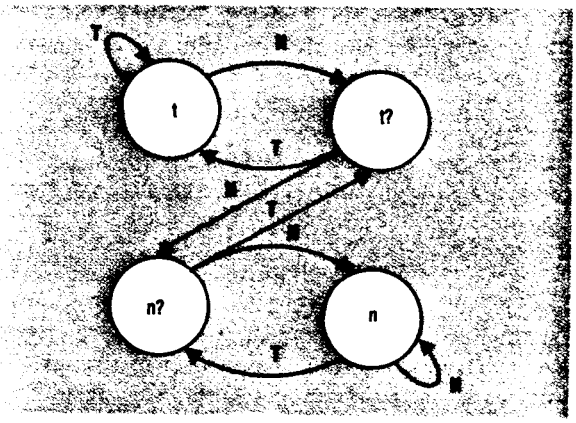


Figure 14. State diagram for branch predictor. The name of the state gives the prediction. For *t* and *t*?, prediction is taken. For *n* and *n*?, prediction is not taken. The label on the arrow is the result of the branch.

Table 8.
Percentage of correct guesses using *n* past branches and conditional probabilities drawn from average of all traces.

n	IBM CPL	IBM BUS	IBM SCI	IBM SUP	DEC PDP11	CDC 6400
0	64.1	64.4	70.4	54.0	73.8	77.8
1	91.9	95.2	86.6	79.7	96.5	82.3
2	93.3	96.5	90.8	83.4	97.5	90.2
3	93.7	96.6	91.0	83.5	97.7	93.4
4	94.5	96.8	91.8	83.7	98.1	94.8
5	94.7	97.0	92.0	83.9	98.2	95.1

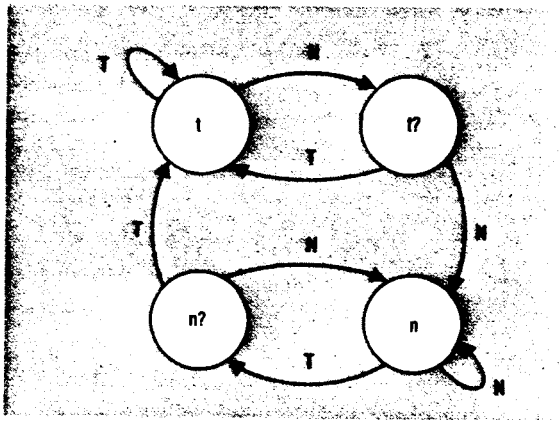


Figure 15. State diagram for branch predictor. The name of the state gives the prediction. The label on the arrow is the result of the branch.

branches were *N*, then two *T*s are required to change the prediction to *T*. The idea here is that a loop exit will not serve to change the prediction. We note, however, that the sequence *NTNTNTNT*..., when started in the wrong state (either *n*? or *t*?) will yield 100 percent wrong predictions; when started in either of the other two states, the predictions will be 50 percent wrong.

In another algorithm, proposed for the S-1,[19] (Figure 15), two wrong guesses are again required to change the prediction, but two are also required to return to the previous prediction. (In the previous algorithm we could return to the previous prediction in one step after two errors.) The sequence *NNTTNNTTNNTT*... can cause every prediction to be incorrect.

Close examination of both Figures 14 and 15 shows that the states indicated do not correspond exactly with the previous two branches. For example, state *n* in Figure 14 implies a history of *NN*, whereas state *n*? implies history of *NNT* or *TNN*.

The success of the algorithms represented in Figures 14 and 15 is shown in Table 9. Comparing the two, we see that their results are almost identical. For further comparison, Table 8's "2" line shows that in most cases (five workloads), the algorithms in Figures 14 and 15 are only slightly better. For the IBM Supervisor workload, the earlier results are three percent better, probably because supervisor code uses branches much less frequently for loop control than do user programs.

We can consider all possible functions *G* and mappings *E* for *n* bits of state to derive the optimal algorithm, but we have not done so, since the results in Tables 8 and 9 and the comparison between them suggest that such an exercise would yield very little, if any, improvement.

Branch target changes. As noted earlier, the branch target buffer contains a number of entries, each of which consists of a branch address, state information, and a target address. The branch target can be obtained only by computing it directly from the instruction or by remembering it from the past execution and assuming that it will be the same. Since the purpose of the BTB is to predict the target immediately, the previous target must be remembered. While target changes are likely to be infrequent, they will sometimes occur, particularly if the source (higher level language program) contains a computed GOTO or a case statement. Execute instructions, such as those from the IBM 370 architecture, also generally change targets.

The possibility of branch target changes implies that when a branch is resolved and found to be taken, the target address must be compared with the target predicted in the BTB. If it is different, the BTB entry must be changed. Also, if the BTB had predicted a branch, then the pipeline must be flushed, and the correct stream of instructions fetched, just as if the BTB had predicted that the branch would not occur. (With this requirement, perhaps a branch whose target has been found to change previously should not be used to predict a branch. We believe, however, that predicting a branch is better, if the cost of an incorrect prediction is the same as the cost of an incorrect fall-through—primarily because a fall-through is very unlikely, whereas the target need not always change.)

Table 10 shows the fraction of all dynamic branches executed for which a branch is taken whose target address differs from that of its previous target. Some of

Table 9.
Prediction success of state diagrams in Figures 14 and 15.

WORKLOAD	FIGURE 14	FIGURE 15
IBM/CPL	93.8	93.8
IBM/BUS	96.2	96.2
IBM/SCI	91.3	91.3
IBM/SUP	80.2	80.2
PDP-11	97.8	97.8
CDC6400	86.4	89.1

Table 10.
Fraction of branch targets found to have changed from previous execution of that branch.

WORKLOAD	PROBABILITY OF TARGET CHANGE (%)
IBM/CPL	4.2
IBM/BUS	2.1
IBM/SCI	4.4
IBM/SUP	1.4
PDP11	12
CDC6400	2.9

these target changes will cause predictions that were otherwise correct (predict branch) to be incorrect. The other cases (predict branch, but none occurs; predict no branch, but branch occurs; and predict no branch, and none occurs) are not affected.

WRITEs into the instruction stream. The branch target buffer is accessed using the address of a previously executed branch. If there has been a WRITE into the instruction stream, such that the bits at the given address no longer specify a branch, then the BTB will not operate correctly. We can deal with this problem in two ways. First, and more correct, is that the instruction in question, identified by the BTB, can be tagged as it moves down the pipeline with a bit specifying "branch." If in the instruction-decode stage, the instruction is found not to be a branch, then the pipeline can be flushed and reloaded, and the BTB can either be flushed or just that entry can be deleted. The alternative is to ignore the possibility of a WRITE into the instruction stream on the basis that the machine architecture forbids modifying instructions, and correct operation is not guaranteed. The latter solution is not acceptable for older architectures, for which existing programs do modify the instruction stream.

Extensions and alternatives. We have defined a general mechanism for predicting branches and shown some results for the more important cases. Some cases exist that we have not considered, and some improvements have been suggested.

Pomerene and Rechtschaffen[29] suggested that a machine be built so that both the taken and not taken directions can be followed (as in multiple instruction streams). Then, if a change in locality is detected, for example, when there are instruction misses in the CPU cache, the multple instruction stream mechanism should be used instead of the BTB predictions. More generally, such a scheme can be used whenever the BTB fails to contain the desired entry.[30]

Smith proposes a strategy (strategy number 3) in which all backward branches are predicted to be taken as loop closures and all forward branches are predicted to be not taken,[25] but the performance is poor. Smith reports on the effectiveness of a number of his other "strategies," but in many cases, the strategies combine the prediction algorithm with implementation issues such as the size of the BTB or its addressing. It is thus difficult to compare most of his results with ours. Another of his ideas is to keep a table of recently used not taken branch instructions, but this technique, of course, fails to retain branch tragets for successful branches, and so can be of only limited use for 370-like architectures. For CDC and Cray architectures,[31] however, the branch target address need not be in the branch target buffer. In those machines, the branch target address can be computed from the instruction itself well before the instruction branch condition is resolved.

Some other ideas Smith[25] has are to keep a taken/not taken bit in the cache, to use a hashed BTB with a one-bit predictor, and to use the same design but with a two-bit predictor. Smith also notes that the branch target buffer does not actually need to hold the address of the branch.[31] The buffer could, for example, have a direct mapping organization (using either bit selection or hashing[2] with a large number of sets. Thus, if a branch hashes into a specific set, the prediction contained therein would be assumed to be for that branch; if because of mapping conflicts, the branch prediction recorded was for the wrong branch, the penalty would at most be a wrong prediction.

An interesting use of the branch target buffer is described by Driscoll et al.[32] An address-generate interlock in a pipeline is a logical dependency between the address calculation function for operand addressing and the register update function in the execution unit. This AGI can delay the processing of a branch instruction because of the need to calculate the target address. Since the BTB predicts the target address, this interlock can be suppressed until the branch is resolved, and the target address can then be calculated only if necessary. An unnecessary pipeline interlock is thus avoided most of the time.[33]

An additional use of the branch target buffer or similar buffer is to speed up access to indirectly addressed operands or addresses. Indirect addressing is a major pipeline blocker, since indirect addressing requires a storage delay for each indirect step. If all fetches (operand, branch target) that could be indirect either by tag in instruction or by tag in target are matched against an "indirect buffer," the ultimate target of an indirect address could be fetched in one step. The BTB could serve double duty here, or a separate buffer could be used. We have not addressed this extension, since none of the three architectures for which we have traces permits indirect addressing.

Branch target buffer implementation

Performance costs and optimal prediction. Thus far, we have assumed that the branch target buffer impacts performance in the following way: A correct prediction by the BTB incurs no lost cycles (fall-through if no branch predicted or correct branch and target prediction), and all incorrect predictions (predict branch, and none occurs or predict fall-through, and branch occurs) result in the same number of lost machine cycles. In a real machine, neither of these assumptions is necessarily true.

Specifically, a prediction of a taken branch could always cost a small number of machine cycles because a taken branch is out of sequence, and storage access time (cache or main memory) may be long enough that the target cannot be fetched before the instruction decode stage of the pipe is ready for it. In Figure 16, we assume that j cycles are lost for every predicted branch.

The cost of a branch predicted to be taken and then not taken may be less than the cost of a branch not expected to be taken, but which is actually taken. This difference can occur because the fall-through sequence of instructions may be already available from a sequential fetch for more than one instruction, and thus when the branch is resolved, the correct target (the fall-through in-

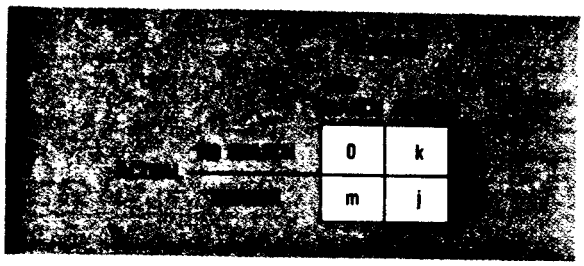


Figure 16. Diagram showing time penalties in lost machine cycles for correctly and incorrectly predicted branches.

struction) may already be on hand. In Figure 16, we assume that the cost of an incorrect positive (predict taken) prediction is k cycles and an incorrect negative (not taken) prediction costs m cycles.

The four events of interest are predict no branch, and no branch occurs; predict no branch, and branch occurs; predict branch, and none occurs; and predict branch correctly. (We omit the target change case here for simplicity.) The respective costs for these events are respectively 0, m, k, and j. Previously, we assumed that $m = k$ and $j = 0$. In that case, the optimal prediction is to maximize the probability of being right, i.e., predicting whether the branch occurs or not. In the latter, more complex case, the optimal prediction is the one that has the average minimum cost. Thus, the optimal strategy does not have to reflect the highest prediction accuracy.

Because m, k, and j are very implementation dependent, we have not developed strategies for cost-based performance predictions. Such strategies can easily (but tediously) be generated, given the costs m, k, and j, from Tables 5 and 6. For each sequence of preceding takens/not takens $\{yi\}$, there is some probability p that the branch is taken and probability $1 - p$ that it is not. If we decide to predict that the branch is taken, the cost is $(1 - p)*k + p*j$. If we decide to predict that the branch is not taken, the cost is $p*m$. The correct prediction is the one with the lower expected cost.

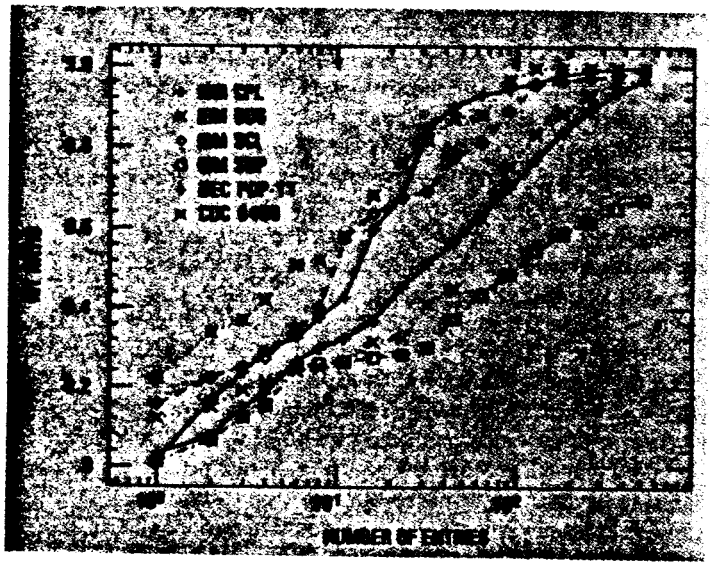


Figure 17. Hit ratio of the branch target buffer as function of the number of entries.

Branch target buffer size and hit ratio. The branch target buffer, like the CPU cache or the translation look-aside buffer, is a small, high-speed memory, and because of both cost and performance must be of limited size. In our analysis thus far, we have always assumed that the BTB had no boundaries and could hold all previously executed branches, which of course, cannot be true. Now we will examine the effect of a BTB with a finite size.

The *hit ratio* of the BTB is the probability that a branch is found to be in the BTB at the time it is fetched. As such, the hit ratio depends on the replacement algorithm and the BTB fetch algorithm. The former determines which item in the BTB to replace when a new entry is to be placed into the BTB. The latter determines when to place entries in the BTB. In particular, it may be better not to enter branches in the BTB if they are not taken, given that the BTB now has a finite size.

We have used a "fetch-all" algorithm here; that is, whenever a branch is recognized, it is entered in the BTB if it is not already there. For replacement, we use the global LRU algorithm, which removes the least recently used (executed) branch in the BTB. (The replacement algorithm could be modified to reflect the fetch algorithm. For example, if the fetch algorithm does not fetch a not taken branch, then when a branch is already in the BTB and is not taken, its replacement status is not altered. That is, if replacement is LRU, then the branch entry is not moved to the top of the LRU stack. Alternatively, to save space, a not taken branch could be deleted from the buffer entirely.)

The hit ratios for various BTB sizes, given fetch-all and global LRU replacement algorithms are shown for each workload in Figure 17. As the figure shows, the hit ratio varies widely. For example, for a 256-entry BTB, the hit ratio varies from a low of 61.5 percent (for the IBM Supervisor workload) to a high of 99.7 percent for the CDC 6400 programs. These results are qualitatively similar to the relative cache hit ratios[2] for the various types of programs, as we would expect. (Widdoes[19] reports that 16 to 32 entries in a BTB yield over 50 percent misses for S-1 traces.)

The branch target buffer is similar in cost and performance constraints to a translation look-aside buffer, or TLB, and the range of feasible sizes should be similar. Thus, the TLB sizes for the following machines are comparable: IBM 3033 (64), Amdahl 470V/6 (128), and Amdahl 470V/7 (256).

A major effect of the finite-size BTB is that it now has fewer advantages over the other "branch problem" solutions discussed earlier. For example, the taken/not taken bit stored in the cache will be more frequently available, if the cache is large, than the BTB entry. Although the taken/not taken bit method is less effective in improving performance, because the branch target address is not immediately available, the higher hit ratio may be sufficient to compensate.

Buffer addressing and organization. The branch target buffer is accessed associatively; that is, the address of the

instruction fetch is matched with the instruction address fields in the BTB. If they match, the appropriate prediction is made. Associative memories are slow and expensive if implemented in other than VLSI, so it is not always feasible to make the BTB fully associative. The two reasonable choices are to make it set associative[34] or hashed as is done for most TLBs.[2] In the former case, some middle bits of the instruction address are used to select a set, and the remaining bits are used for the associative match within the set. The replacement is within the set. Hashing is usually combined with set associative replacement as follows. The address of the instruction is hashed,[2] and a set of elements is selected. The search is then associative within this set (the set size may be one), and replacement is also within the set. Since A. Smith's experiments[2] showed the two methods to be about equally effective, we select the standard set associative mapping as simpler, cheaper, and faster. (J. Smith uses hashing as one of his strategies.[25])

Table 11 shows the effect of the set size is shown for the IBM/CPL mix. (The effects of other mixes are presented elsewhere.[35]) A. Smith shows that set sizes of four or eight are sufficiently large and closely approach the hit ratio of the fully associative design.[2,34]

The effect of multiprogramming. Multiprogramming is important to both the design and performance of the branch target buffer. Whenever the address space in control of the computer changes, the association between virtual memory addresses and memory contents changes. (Since virtual addresses are the ones generated by the program, the BTB must be accessed using virtual addresses. Otherwise all BTB accesses would require translation first.) Thus, the BTB should be purged when the address space changes; otherwise incorrect matches will occur as will incorrect predictions. Each such prediction will have to be corrected, and since many incorrect positive predictions will take place for non-branches, the number of errors will be high and the performance cost significant.

The effect of purging the BTB, or equivalently, in correcting it entry by entry, is that the BTB will usually contain far fewer valid entries than our previous discussions and simulations suggest. As a worst-case example, consider the data in Table 12. The table compares the fraction of correct predictions using an infinite BTB with those from an infinitely large BTB that is flushed every 1000 instructions. As the table shows, these frequent flushes significantly impact performance. We believe, however, that address space switches will occur at intervals closer to 3000 to 25,000 instructions than to 1000. Therefore, the BTB flushes may have less of an effect on the miss ratio than will the finite size of the BTB.

If the BTB is to be flushed when a task switch occurs, then the task switch must be detected. Further, some time may be lost as the flush takes place. Smith discusses fast methods for flushing TLBs.[2]

Restrictions on logic complexity. The branch target buffer, as noted earlier, is closely associated with the CPU pipeline and must therefore function very quickly. Cost and size limitations combine with the speed requirement to limit the feasible degree of complexity for the BTB. We have therefore narrowed the range of alternatives considered to those that are sufficiently simple and inexpensive to implement. Further, we have looked at the effect of BTB size and organization for the same reason. Anyone proposing either to design a BTB or to study BTBs further should keep in mind these important constraints.

MU-5 implementation and results. The MU-5 computer system uses a branch target buffer whose effectiveness is discussed by Holgate and Ibbett.[23] The BTB retains up to eight previously taken branches and their targets. Only branches with fixed (invariant) targets are placed in the BTB.

The effectiveness of the MU-5 BTB was studied using a hardware monitor; measurements were made for a mix of compilations and executions for both Fortran and Algol. Branches constitute 14 and 12.5 percent of the instructions from Algol and Fortran executions, respectively. The BTB correctly predicts from 40 (Algol compilation) to 65 percent (Algol execution) of the correct sequences after a branch (including fall-throughs), as compared with 15 to 25 percent without the BTB.

S-1 trace experiments. Some branch target buffer experiments on S-1 traces have been reported.[19] Success rates are from 91 to 95 percent with one- to five-bit predictors, using the method shown in Figure 15. The effectiveness of this scheme varies from worse than the one-bit predictor to almost as good as the four-bit predictor. These experiments were run on two traces of about 100,000 instructions.

Table 11.
Branch target buffer hit ratios (IBM/CPL mix).

BUFFER SIZE	SET SIZE 1	2	3	4	8	16	32	64	128	256
1	0.031									
2	0.057	0.075								
4	0.084	0.124	0.185							
8	0.161	0.174	0.228	0.298						
16	0.258	0.267	0.271	0.333	0.369					
32	0.353	0.359	0.355	0.369	0.441	0.514				
64	0.407	0.470	0.475	0.499	0.513	0.570	0.634			
128	0.562	0.602	0.617	0.623	0.623	0.626	0.702	0.769		
256	0.678	0.725	0.751	0.759	0.765	0.768	0.770	0.840	0.888	
512	0.784	0.835	0.865	0.879	0.886	0.886	0.880	0.911	0.952	
1K	0.864	0.919	0.944	0.956	0.961	0.964	0.965	0.966	0.966	
2K	0.917	0.961	0.974	0.979	0.981	0.981	0.981	0.981	0.981	
4K	0.946	0.976	0.981	0.981	0.981	0.981	0.981	0.981	0.981	

Table 12.
Comparative percentages of correct guesses in a multiprogramming environment.

	IBM CPL	IBM BUS	IBM SCI	IBM SUP	DEC PDP11	CDC 6400
No Flush	93.2	95.9	89.7	80.0	97.4	85.5
Flush Every 1000 Instructions	79.9	83.3	74.9		86.3	68.9

Use for tracing. In some computers, circular buffers are maintained of the last n instructions or branches executed, and their contents are useful in debugging both hardware and software. The branch target buffer can be combined in function with the circular branch buffer.[36]

Overall BTB effectiveness

The reason for building a branch target buffer is to improve CPU performance. Thus, the results on correct predictions and hit ratios must be integrated with the costs of hits and misses and correct and incorrect predictions to get an overall estimate of performance impact.

For example, in the IBM/CPL mix, we can predict the branch path with an accuracy of 93.8 percent, using the predictor depicted in Figure 14. A hit ratio of 86.5 percent is obtained with a BTB consisting of 128 sets of four entries each. Up to 4.2 percent of our predictions will be incorrect due to target changes, giving an overall minimum prediction accuracy of $(93.8 - 4.2)\ 0.87 = 78$ percent.

Prediction accuracy can be used to estimate the performance impact by considering a real machine. We used the Amdahl 470V/6,[14] which has a machine cycle time of 32.5 nsec and runs at about four MIPS.[37] Excluding memory access delays, five MIPS is closer (and the figure we used) and yields a mean of six cycles per instruction. Each branch taken causes a delay of four machine cycles. If the branches are 30 percent of the instructions, and 65 percent of the branches are taken. Excluding the branch penalty, the mean execution time t for an instruction would be $6 - (0.3)(0.65)(4) = 5.22$ cycles. Branch prediction using the BTB would then result in a mean execution time of $5.22 + (0.3)(1 - 0.78)(4) = 5.48$ machine cycles. Defining performance as the rate of instruction execution gives us a performance improvement of 9.5 percent.

This computation, using the same basic figures, has been replicated, varying each parameter of interest, one case per table, and the results appear in Figure 18. The figure shows (left to right, top to bottom) the mean instruction time for different basic instruction execution times, the mean instruction time for different time penalties when the wrong stream is processed after an unresolved branch, and the mean instruction time for different hit ratios in the BTB with basic instruction times of 5.22 and 2.2 cycles.

Figure 18a shows that the BTB is most effective when the cost of an incorrect guess is large relative to the mean instruction time. That result is confirmed in Figure 18c in which the other parameter of that pair is varied. Figure 18b shows that the hit ratio to the BTB is important and rises in importance, as seen in Figure 18d, when the basic instruction time is short.

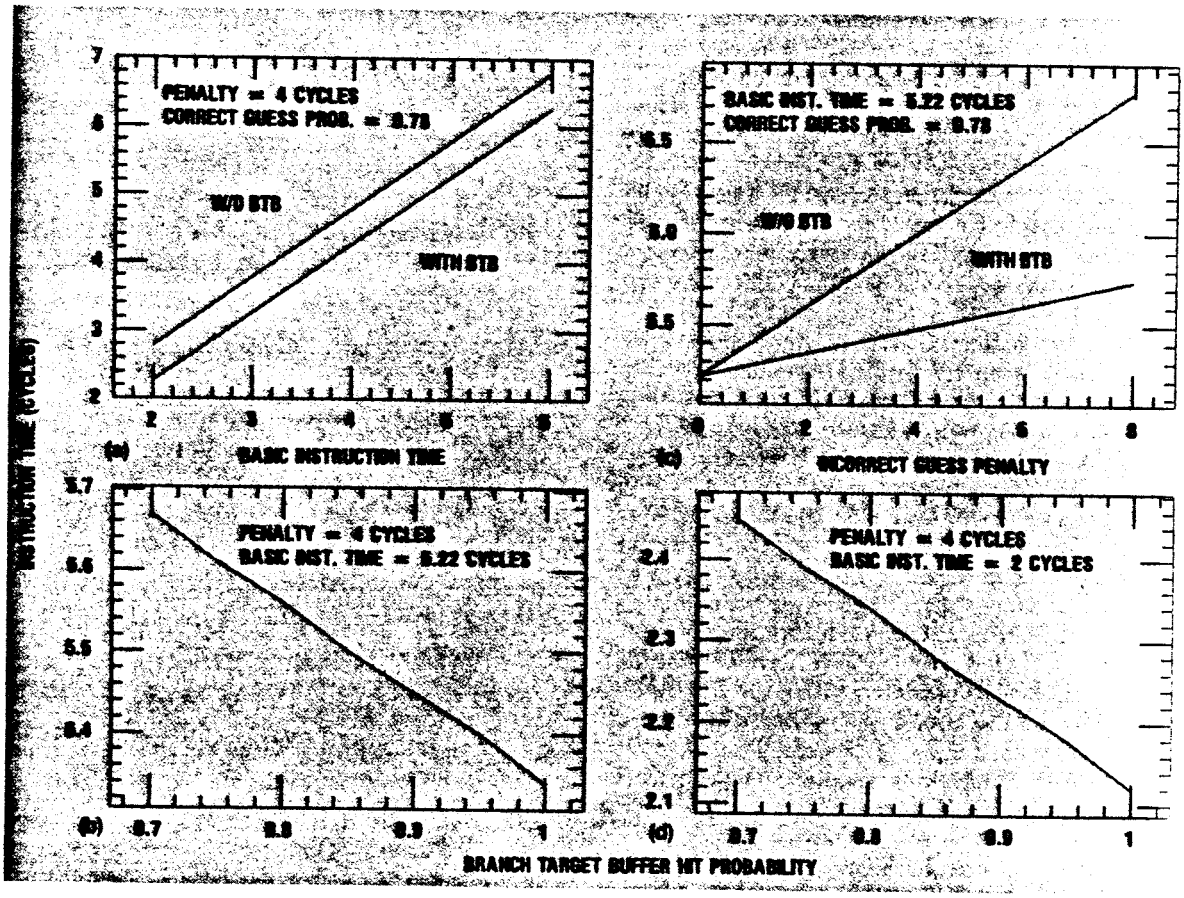


Figure 18. Mean instruction time in machine cycles as a function of variations in the basic instruction time (a), the incorrect guess penalty (c), and the probability of a branch target buffer hit (b), (d).

Taken branches have long been one of the major obstacles to high efficiency in a pipelined computer system. A great deal of effort has been invested in overcoming this problem, either by facilitating the access to instructions (loop buffers, target prefetch) or by directly attacking the branch problem (multiple instruction streams, delayed branch, etc.). We believe that the branch target buffer is the most effective way to minimize branch penalties.

Our study of the BTB has been based on a close examination of instruction traces and analysis of their behavior. We have developed a general prediction strategy, based on branch history and op code, and have measured the effectiveness of the important variants of this predictor. Our results show that two bits are sufficient to retain the necessary state information for effective prediction. We also found that on the order of 256 entries in the BTB are required for some workloads and represent a good design target for a large, high-performance machine.

We have also considered various implementation issues, such as the design of the BTB addressing (set associative), the effect of multiprogramming on the hit ratio, the need to flush the BTB when the address space changes, and the problems of branch target changes and WRITEs into the instruction stream.

The use of six workloads, taken from three machines, gives us reason to believe our results are representative of the those to be generally expected, and we believe our work has direct application to high-speed computer system design. A number of extensions to the basic BTB include the use of the BTB or another similar buffer to avoid penalties from indirect addressing. Improvements in CPU performance of from 5 to 20 percent can be expected when comparing a BTB design to a similar CPU design without a BTB. ■

Acknowledgments

Partial support for this research was provided by the National Science Foundation under grants MCS77-28429 and MCS-8202591 and by the Department of Energy under contract DE-AC03-76SF00515 to the Stanford Linear Accelerator Center.

The four IBM/370 program address trace workloads were created at Amdahl Corporation, and much of the analysis presented was also done while J. Lee was employed at Amdahl. We thank Amdahl and W. Harding for help and cooperation.

References

1. C. V. Ramamoorthy and H. F. Li, "Pipeline Architecture," *Computing Surveys*, Vol. 9, No. 1, Mar. 1977, pp. 61-102.
2. A. J. Smith, "Cache Memories," *Computing Surveys*, Vol. 14, No. 3, Sept., 1982, pp. 473-530.
3. *STAR-100 Hardware Reference Manual 60256000*, Control Data Corporation, Arden Hills, Minn., 1975.
4. J. E. Thornton, "Parallel Operation in the Control Data 6600," *AFIPS Conf. Proc.*, Vol. 26, part I, 1964 FJCC, pp. 33-40.
5. *Control Data 7600 Hardware Reference Manual 60367200*, Control Data, Arden Hills, Minn., 1975.
6. R. M. Russell, "The Cray-1 Computer System," *Comm. ACM*, Vol. 21, No. 1, Jan. 1978, pp. 63-72.
7. L. C. Garcia and T. Huynh, "Storage Fetch Contention Reduction by Using Instruction Branch Prediction," *IBM Technical Disclosure Bull.*, Vol. 23, No. 6, 1980, pp. 2404-2405.
8. E. M. Riseman and C. C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Trans. Computers*, Vol. C-21, No. 12, Dec. 1972, pp. 1405-1411.
9. *IBM Maintenance Library System/370 Model 168 Theory of Operation/Diagrams Manual*, Vol. 2, 1973, IBM, Poughkeepsie, N.Y.
10. *IBM Maintenance Library 3033 Processor Complex Theory of Operation/Diagrams Manual*, Vols. 1-3, Jan. 1978, IBM, Poughkeepsie, N.Y.
11. J. F. Hughes, "Branch on Condition Decoding With Instruction Queues Empty," *IBM Technical Disclosure Bull.*, Vol. 24, No. 4, 1981, pp. 1857-1858.
12. J. Y. Yamour, "Instruction Scan for an Early Resolution of a Branch Instruction," *IBM Technical Disclosure Bull.*, Vol. 23, No. 6, Nov. 1980, pp. 2600-2604.
13. D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The Model 91: Machine Philosophy and Instruction Handling," *IBM J. Research and Development*, Vol. 11, Jan. 1967, pp. 8-24.
14. *Amdahl 470 V/6 Machine Reference Manual*, Amdahl, Sunnyvale, Calif., 1976.
15. *The ACS System Central Processor*, manual 929982-11, Texas Instruments, Dec. 1976.
16. George Radin, "The 801 Minicomputer," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems*, Mar. 1982, pp. 39-47. (also available as *Sigarch Computer Architecture News*, Vol. 10, No. 2, Mar. 1982, and as IBM Research tech. report RC 9125, Nov. 1981).
17. E. R. Berlekamp, presentation at CS Division Seminar, UC Berkeley, 1979.
18. D. A. Patterson and H. Sequin, "RISC-1: A Reduced Instruction Set VLSI Computer," *Proc. Eighth Symp. Computer Architecture*, May 1981, pp. 443-458.
19. C. Widdoes, Jr., *Jump Prediction*, Feb. 1977, unpublished draft.
20. B. T. Hailpern and B. L. Hitson, *S-1 Architecture Manual*, Stanford University, Computer Systems Laboratory tech. report STAN-CS-79-715, Stanford, Calif., Jan. 1979.
21. A. G. Liles, Jr., and B. E. Willner, "Branch Prediction Mechanism," *IBM Technical Disclosure Bull.*, Vol. 22, No. 7, 1979, pp. 3013-3016.
22. G. S. Rao, "Technique for Minimizing Branch Delay Due to Incorrect Branch History Table Predictions," *IBM Technical Disclosure Bull.*, Vol. 25, No. 1, June 1982, pp. 97-98.
23. R. W. Holgate and R. N. Ibbett, "An Analysis of Instruction Fetching Strategies in Pipelined Computers," *IEEE Trans. Computers*, Vol. C-29, No. 4, Apr. 1980, pp. 325-329.
24. J. J. Losq, "Generalized History Table for Branch Prediction," *IBM Technical Disclosure Bull.*, Vol. 25, No. 1, June 1982, pp. 99-101.
25. J. E. Smith, "A Study of Branch Prediction Strategies," *Proc. Eighth Symp. Computer Architecture*, May 1981,

pp. 135-148 (also available as *Sigarch Newsletter*, Vol. 9, No. 3, 1981).

26. A. J. Smith, "Bibliography on Paging and Related Topics," *Operating Systems Review*, Vol. 12, No. 4, Oct. 1978, pp. 39-56.
27. J. R. Spirn, *Program Behavior: Models and Measurements*, Elsevier, New York, 1977.
28. J. M. Angiulli et al., "Branch Direction Prediction Mechanism," *IBM Technical Disclosure Bull.*, Vol. 23, No. 1, June 1980, pp. 268-269.
29. J. H. Pomerene and R. N. Rechteschaffen, "Dynamic Branch Prediction Using Branch History Table," *IBM Technical Disclosure Bull.*, Vol. 22, No. 8A, Jan. 1980, p. 3437.
30. J. Pomerene and R. Rechtschaffen, "Reducing Cache Misses in a Branch History Table Machine," *IBM Technical Disclosure Bull.*, Vol. 23, No. 2, July 1980, p. 853.
31. J. E. Smith, private letter, Jan. 3, 1983.
32. G. C. Driscoll et al., "Address Generate Interlock Avoidance for Branch Instructions in a Branch-History-Table Processor," *IBM Technical Disclosure Bull.*, Vol. 24, No. 1A, June 1981, pp. 350-354.
33. J. J. Losq, "Address Generate Interlock Memory Buffer," *IBM Technical Disclosure Bull.*, Vol. 25, No. 1, June 1982, pp. 114-120.
34. A. J. Smith, "A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory," *IEEE Trans. Software Engineering*, Vol. SE-4, No. 2, Mar. 1978, pp. 121-130.
35. J. K. Lee, "Performance Improvement of CPU Pipelines," PhD dissertation, University of California, Berkeley (to appear 1984).
36. D. Boniface et al., "Central Control Unit Branch Trace Mechanism," *IBM Technical Disclosure Bull.*, Vol. 24, No. 7A, 1981, pp. 3503-3505.
37. B. L. Peuto and L. J. Shustek, "An Instruction Timing Model of CPU Performance," *Proc. Fourth Symp. Computer Architecture*, Mar. 1977, pp. 165-178.

SOFTWARE PROFESSIONALS NEW ENGLAND/NATIONAL OPPORTUNITIES

Are you thinking of career advancement? Are you concerned that the person who represents you be as professional and informed in their field as you are in yours? Let Dan Meagher put his years of successful placement counselling to work for you. In New England and around the U.S. through our NPC network, E.P. Reardon Associates has been placing top professionals for over 20 years.

If the challenge is gone or your career path is blocked in your present position, we have requirements for experienced professionals in the following areas:

Operating Systems Design
Computer Architecture
Interactive Graphics
Peripherals Interfaces
CAD/CAM
Networking
Scientific Applications
Compiler Design
Office Automation
Artificial Intelligence

If you are interested in exploring these career openings, call Dan Meagher, (617) 273-5964, or forward a copy of your resume to him. All inquiries will be answered within 48 hours and will be treated with complete confidentiality.

E.P. Reardon Associates
P.O. Box 1038,
Burlington, MA 01803
Clients are EOE. Member of NPC.

Johnny K. F. Lee is a project manager at the Personal Office Computers Division of Hewlett-Packard Company, where he is involved in system software development for new personal computer products. He also worked at the company's Computer Research Laboratory. Prior to joining Hewelett-Packard, he worked at Amdahl Corporation.

Lee received a BS and MS from Cornell University and is now a PhD candidate at the University of California, Berkeley. His current research interests are operating systems, user interfaces, and VLSI design tools.

Alan Jay Smith is an associate professor in the Computer Sciences Division of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, where he has been on the faculty since 1974. He is also vice chairman of the EECS Department. His research interests include the analysis and modeling of computer systems and devices, operating systems, and computer architecture. He has published a large number of research papers, one of which won the 1979 IEEE Best Paper Award for the best article in the *IEEE Transactions on Computers*.

Smith is a senior member of the IEEE and a member of the ACM, the Society for Industrial and Applied Mathematics, Eta Kappa Nu, Tau Beta Pi, and Sigma Xi. He is also chairman of the ACM Special Interest Group on Operating Systems and an associate editor of the *ACM Transactions on Computer Systems*.

Smith received a BS in electrical engineering from the Massachusetts Institute of Technology and an MS and PhD in computer science from Stanford University.

Questions about this article can be directed to either author at Computer Science Div., EECS Dept., University of California, Berkeley, CA 94720.