Analysis of Branch Prediction Strategies and Branch Target Buffer Design*

Johnny K. F. Lee**

and

Alan Jay Smith***

University of California, Berkeley

Keywords: pipeline, branches, branch prediction, instruction statistics.

C.R. Categories: B.12, C.11, C.4

ABSTRACT

CPU pipelines need a steady flow of instructions in order to function with maximum effectiveness. Branches which change the expected order of instruction execution require that the pipeline be reloaded, resulting in several lost machine cycles per such branch. By examining the type of branch and the past execution behavior of that branch (taken/not taken) it is possible to predict with high accuracy whether the branch will be taken or not taken, and by remembering the previous branch target (destination), to predict the current branch target. In this paper we use a systematic approach to selecting good prediction strategies. Our studies are based on 26 program address traces grouped into four IBM 370 workloads (scientific, commercial, compiler, supervisor) and CDC 6400 and DEC PDP-11 workloads. Our 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 from 5% to 20% can be expected in CPU performance when a branch target buffer is installed. We also consider issues relating to the implementation of real branch target buffers.

---

*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.

**Present Address - Hewlett-Packard Company, GSD, 978 E. Arques Ave., Sunnyvale, Ca. 94086.

***Computer Science Division, EECS Department, University of California, Berkeley, Ca. 94720.

I. Introduction

Modern high speed computer systems achieve their high performance by two means: fast logic and parallelism. One of the most important aspects of that parallelism is the pipelining of instruction execution. That is, to execute each instruction, a number of operations must be performed sequentially. A typical sequence might consist of instruction fetch (IF), instruction decode, operand address generation, operand fetch, execution and operand write. This sequence is illustrated in figure 1. Pipelining consists of executing several instructions concurrently, with each instruction in a different stage of the processing sequence (see figure 2.) The pipeline (pipe) shown in figure 1 could contain up to 7 instructions, each in a different stage. We refer the reader to [Ram77] or [Kogg81] for a comprehensive survey of pipelining.

Pipelines are very cost effective as a means to achieve parallelism because although several instructions may be processed in parallel, each is in a different phase of execution, and thus there need exist only one set of hardware for each stage. Of course, the logic required to implement a totally non-pipelined machine would be less, since there would be less control logic, but a significant savings is still obtained.

For a number of reasons, the N-fold parallelism promised by an N stage pipeline is seldom realized. First, we note that the pipeline is constrained to run no faster than its slowest stage. Even if all of the stages have the same average processing time, random variations in one or more of them will disrupt the smooth flow of instructions [Pine79]. For example, the execution stage of the pipeline will generally run more slowly for "decimal divide" than for "integer add". We will refer to the minimum time for an instruction to advance one stage in the pipe as the pipeline stage time (PST). Other delays encountered in pipelines occur because of attempts to use busy resources (e.g. conflicts in accessing a single port register file), or failure to have an input available. (E.g. one instruction uses the output of the immediately previous one as part of

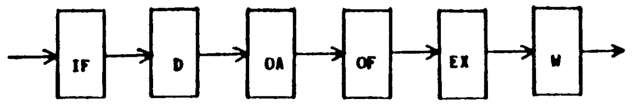


Figure 1
Typical
Pipeline

IF - Instruction Fetch
D - Decode
OA - Operand Address Generation
OF - Operand Fetch
EX - Execution
W - Operand Write

Figure 2
Typical Pipeline
Time Sequence

Time	Stage: IF	D	OA	OF	EX	W
1	1					
2	2	1				
3	3	2	1			
4	4	3	2	1		
5		4	3	2	1	
6			4	3	2	1
7			4	3	2	
8				4	3	2

Instruction Number

Figure 3

Branch Target Buffer

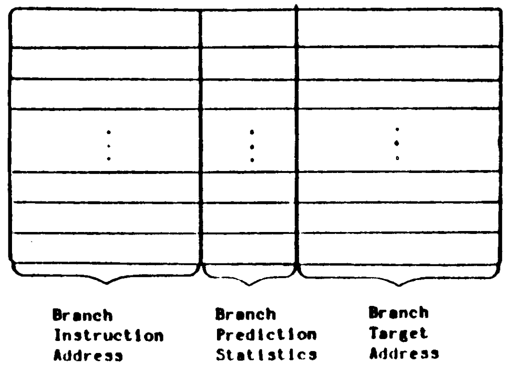
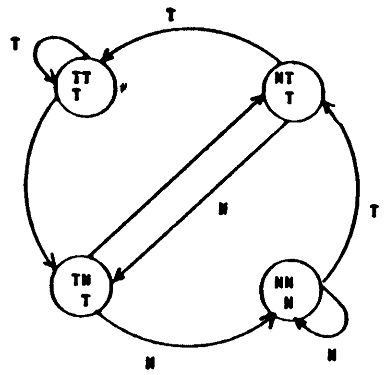


Figure 4



State = Last 2 Branches
Followed by Prediction

Figure 5

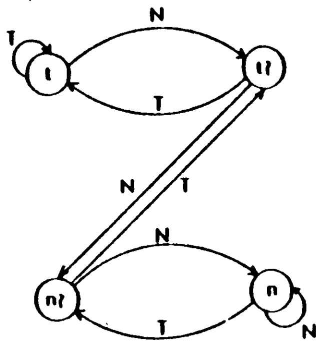
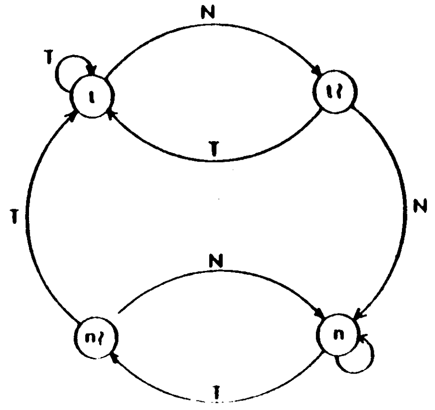


Figure 6



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 [Smit79].

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: (1) The memory access time is long enough that a request by the instruction fetch (IF) stage of the pipe for another instruction will not be satisfied in one PST; or (2) a change in the expected sequence of instructions, due, for example, to a branch, will cause the contents of part of the pipeline to be discarded, and the pipeline to be reloaded. We call this latter reason the "branch problem". The branch problem is intimately 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 more fully as follows: The "execution" of a branch instruction consists of causing the instruction fetch unit to select a different instruction as the next instruction to execute. Thus, considering the pipeline shown in figure 1, all of the partially executed instructions in the stages of the pipeline preceding the execution unit must be discarded; an additional delay is also encountered in fetching the new, out of sequence instruction. Each of these two delays can seriously impact the CPU performance.

There are a number of ways in which the performance degradation due to branches in the instruction stream can be reduced. We list those methods here; each is discussed in more detail in the next section: loop buffer(s), prepare to branch, delayed branch, multiple instruction streams, prefetch branch target, data fetch target, taken/not taken switch, and the branch target buffer. The branch target buffer (BTB), the principal topic of this paper, is a small associative memory which retains the addresses of recently executed branches and their targets (destinations). This buffer is used to predict whether the branch will be taken this time, and if so, what the target of the branch will be. The instruction fetch stage then continues by fetching instructions from the predicted target address.

In this paper we present a thorough and systematic study of the design and effectiveness of branch target buffers. In the next section we review the various existing solutions to the branch problem and also discuss previous studies of the BTB. Our research is based on extensive analysis of program instruction traces, which is explained in section III. Section IV considers instruction behavior and tabulates a large variety of measurements on those traces. After that, we consider branch prediction mechanisms of various complexity. There are a number of implementation considerations which are discussed in section VI. **Finally**, we estimate the overall effectiveness of the BTB.

II. Existing Approaches to the Branch Problem

Modern computers implement a variety of mechanisms to minimize the branch problem. In this section, we briefly discuss each.

A. Loop Buffer(s)

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. 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 (this may include backward branches) and will deliver those instructions without accessing memory. It is possible to fetch all of the instructions for a loop entirely from this buffer; thus the name "loop buffer". Among the machines using a loop buffer include the CDC Star-100 [CDC73] with a buffer of 256 bytes, the CDC-6600 with 60 bytes, [Thor64,CDC74] and the CDC 7600 with twelve 60 bit words [CDC75].

The Cray-I maintains four loop buffers [Cray76,Russ78], and replaces their contents in a FIFO manner. The idea here is that a loop may consist of several non-contiguous instruction sequences and thus may be better captured this way than by a mechanism that permits only one sequence.

B. 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 either fetch the next sequential instruction or it must fetch 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. There are three problems with this approach: (1) The branch target cannot be fetched until its address is determined. That may require a computation, 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, which occurs if an operand is the result of a not yet completed instruction or when a memory fetch is required. There may also be contention delays, as, for example, in accessing the register file. Also, additional memory traffic is generated, further creating resource contention [Garc80]. (2) If instruction I is a branch instruction, then there may be additional branch instructions to enter the pipeline (either part) before I is resolved as to taken/not taken and target. Riseman and Foster [Rise72] found that for a typical length pipeline, more than two branches would have to be so processed for there to be a significant improvement. The net amount of hardware is likely to be impractical. (3) Finally, the cost of replicating significant parts of the pipeline (including instruction fetch, instruction decode, operand address generate) is substantial; thus this mechanism is of questionable cost effectiveness.

Despite the problems with following multiple instruction streams, a number of machines do so: The IBM 370/168 [IBM73] can fetch one alternate instruction path; the IBM 3033 [IBM78] can pursue two alternate instruction streams. The 3033 only fetches an alternate instruction stream when it is predicted to be taken; this prediction depends on the branch condition mask in the instruction, the operation code and the target address operand register. (See also [Hugh81b].)

C. Prefetch Branch Target

Rather than replicate several initial stages of the pipeline, it is helpful to 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 [Yamo80]. 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.

IBM 360/91 uses the simple prefetch mechanism described, whereby it prefetches a double word target [Ande67, IBM70].

D. 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 [Amda76] 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.)

E. Prepare to Branch

The Texas Instruments ASC computer [TI76] 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.

F. Delayed Branch

The problem with a branch is that if instruction I is a taken (successful) branch, then instruction I+1 will be out of sequence, with the consequences described above. The instruction set architecture can be specified so that a branch is defined to affect the address not of instruction I+1 but of instruction I+k, for some k. If k is equal to or larger than the number of pipeline stages preceding the stage at which the branch is resolved (executed) then the instruction fetch can (almost) always be given the correct address from which to fetch. (The "almost" refers to the occurence of asynchronous events such as interrupts, which cannot be predicted from the instruction stream.)

There are several problems with designing a machine to use a delayed branch. The most significant is that it is likely that human programmers will find it very difficult to write code in which some instructions (branches) have 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. We also note that the delayed branch requires a new architecture; it cannot be used as a technique to speed up an existing one. In addition, not all of the potential speed up of the delayed branch may be realized; there may not be k-1 instructions that can be usefully performed once the branch is resolved.

Despite the problems noted above, there are two existing experimental computers which use the delayed branch: the IBM 801, an experimental minicomputer constructed at IBM Research, Yorktown Heights [Radi82], and a dedicated microprogrammed machine constructed by E. R. Berlekamp [Berl79] to insert and remove error correcting codes from signal transmissions. It has been proposed for the RISC computer [Patt81].

G. Taken/Not Taken Switch

As we will show in the remainder of this paper, whether or not a branch will be taken can be predicted with good accuracy. A prediction mechanism which specifies whether a branch will or won't likely be taken is called the taken/ not taken switch. The idea is that with every instruction in the cache memory, one or more bits are associated. 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.

The taken - not taken switch has been proposed for the S-1 computer [Widd77; see also Hail79,Wood79]. 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 wrong (the Wrong bit). Two wrong predictions in a row cause the Jump bit to be changed. The effectiveness of this prediction algorithm is discussed in [Widd77] and below in section V.B. We note that this mechanism still encounters delays due to target address computation and the out of sequence fetch. A version of this scheme is proposed in [Lile79].

H. Look Ahead Resolution

Another proposed solution to the branch problem [Hugh81a] is to place extra logic in the pipeline so that an early stage of the pipeline can resolve a branch whenever possible. It is possible if the condition code affecting a conditional branch has already been determined. See also [Losq82b] and [Rao82].

I. Branch Target Buffer

The Branch Target Buffer is a small cache memory associated with the instruction fetch (IF) stage of the pipeline. The BTB retains three-tuples, each of which contains: the address of a previously executed instruction, information which permits a prediction as to whether the instruction branch will be taken, and the most recent target address for that branch (see figure 3). It 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 of the pipe, the BTB is updated, if necessary, with the corrected prediction information and target address. We note that the BTB can be used for every instruction fetch, and thus it can have as many predictions in force as there are instructions uncompleted 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. There are also the implementation issues of how large the BTB will/should be, and how it should be organized (e.g. set associative, hashed, etc.). There have been two previous papers which study these optimizations. Holgate and Ibbett [Holg80] study its effectiveness in the context of the MU-5, which actually implements a branch target buffer, roughly of the type described. (See also [Losq82c] which proposes this idea.) J. Smith [Smit81] examines a number of BTB designs using traces for the CDC Cyber 170 computer. In both cases, the results are similar to our own, but in this paper we study three different machine architectures (IBM 370, DEC PDP-11, CDC 6400) and we consider prediction strategies much more systemmatically. We compare their results to our own as appropriate in the remainder of this paper.

III. Methodology and Data

There currently exists no statistically acceptable model to characterize any aspect of program behavior (although with respect to paging and memory management, there has been much research [Smit78c], [Spir77]). With respect to the design and evaluation of branch target buffers, there is the need for a model of when branches occur, whether they will be taken or not, and whether the branch target will change. Because there is no model which can accurately predict these things, our research will be based on the thorough analysis and use for trace driven simulation of program address traces.

A. The Data

We have available 26 program address traces (see Appendix I) which we have grouped into six workloads. Four of these 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 a supervisor state set of traces (MVS operating system). Six traces form the Digital Equipment Corporation PDP-11 workload, and six more traces comprise the Control Data Corporation 6400 workload. In Appendix I we list each of the programs in each workload and also give their combined total length in instructions.

From each program trace, the branch instructions were extracted, along with their targets, addresses, number in sequence and operation codes. All of the 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, serves to show the variation to be expected from the different job mixes experienced on different computer centers, different machines and at different times of day. It is well known that certain workloads have different instruction mixes; e.g. business programs use many more SS type operations on the IBM 370 than scientific programs. Conversely, the scientific programs have far more floating point operations. If such differences affect the effectiveness of a branch target buffer, it will be apparent in our studies. Similarly, the use of traces from three very (!) different machine architectures will ve some indication of whether the results are sensitive to the instruction set architecture.

Some of our studies will show results for various specific machine instructions. For reference purposes, we show the branch instructions for each machine in Appendix II. Some of our studies will be limited to conditional branches only; the instructions considered to be conditional branches are also listed in Appendix II.

B. 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, in which various parameters or features of interest can be varied. If the variation in parameters does not affect the validity of the trace, then the trace driven simulation can accurately predict the effect of changes in the system. An early example of trace driven simulation for the evaluation of paging algorithms is given in [Bela66], and for the evaluation of CPU scheduling, in [Sher72].

We use our 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 branch target buffer designs.

IV. Branch Behavior

A. Taken/Not Taken and Branch Frequency by Opcode

For each trace, in table 1 we show the overall probability of a branch being taken or not, and the ratio (r) of branch instructions to all instructions in the trace. We note two important features here: first, branches are taken twice as often as not; thus by just guessing that branches are always taken, we are right 60 to 70% of the time. (In [Smit81] the range over six traces is 57% to 99%, with an average of 76.7%.) Variation between workloads is moderate, and for all workloads, branches are taken a majority of the time.

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

Fraction of Branches, taken, not taken, and fraction of branches overall(r).
Table 1

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				

Frequency of Branch Types
Table 2

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				

Probabilities of Branch Taken by Branch Type
(- indicates instruction not in that trace)
Table 3

The probability that a branch is of a specific operation code is shown in table 2 for each workload. In the case of the 370 workloads, we note significant variation in the frequency of the various operation types.

Table 3 shows the probability that a branch is taken for each operation code. It is worth noting, of course, that 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 usually not taken because it is often used as a decrement instruction).

B. Dynamic Branch Behavior

It is important to note that not all branches are executed equally frequently. A good deal of our ability to predict branches will depend on the fact that some branches are executed large numbers of times and therefore from past behavior we can make a good guess as to what will happen next. To provide a clear vocabulary for discussing this, we first define two terms:

Static Branch Instructions - This 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.

Dynamic Branch Instructions - This 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; every instance of the execution of a static branch instruction results in a new dynamic branch.

In table 4 we show the probability distribution for each workload for the number of times a static branch occurs as a dynamic branch. Table 5 shows the probability that a dynamic branch is due to a static branch which is executed n times. As may be evident, the large bulk of dynamic branches occur for frequently executed static branches; for example, the 23.4% of the static branches of the IBM/CPL mix get executed only once, but they account for only .5% of the dynamic branches. On the other hand,

n	IBM CPL	IBM BUS	IBM SCI	IBM SUP	DEC PDP11	CDC 6400
1	23.4	25.2	19.4	64.7	13.8	16.2
2	9.1	10.7	6.0	10.6	5.8	3.0
3	4.7	8.6	8.1	5.0	2.6	7.8
4	5.0	7.3	3.1	3.3	5.0	2.7
5	2.8	2.9	2.2	4.6	1.5	1.1
6	2.8	4.4	3.4	1.7	3.7	1.0
7	3.0	1.6	2.1	0.9	1.3	.7
8	2.2	2.0	2.1	1.6	.5	2.0
9	2.7	.5	2.6	.8	.3	.9
10	1.1	4.2	4.2	.5	.7	.7
>200	10.4	9.3	10.9	.1	40.1	32.0

Percentage of branch instructions executed n times
Table 4

n	IBM CPL	IBM BUS	IBM SCI	IBM SUP	DEC PDP11	CDC 6400
1	.5	.3	.1	18.5	.0	.0
2	.3	.4	.1	6.0	.0	.0
3	.2	.5	.1	4.2	.0	.0
4	.4	.7	.1	3.8	.0	.0
5	.3	.4	.0	6.5	.0	.0
6	.3	.7	.2	2.9	.0	.0
7	.5	.3	.1	1.8	.0	.0
8	.4	.4	.1	3.6	.0	.0
9	.4	.1	.3	2.0	.0	.0
10	.3	.2	.5	1.5	.0	.0
>200	48.2	74.5	80.4	16.5	98.4	95.6

Percentage of branch instructions executed n times weighted by n
i.e. Probabiility that Dynamic Branch Results from
Static Branch Executed n times.
Table 5

1	0000000000000000000000000000000
2	1111111111111111111111111111111
3	110100110100111
4	110110110111110
5	00000000000000100000000000001000000000000001
6	111111111111111111111111111111111111111111
7	1111111111111111101111111111111111111110111111111111111110
8	1111111111110111111111111001111111111110001111111111110000
9	1111111111111111111111111111111111111111111111111111111111
10	0000000000000000000000000000000000000000000000000000000000
11	0100011001010000011100110101000101100010010000000010001100
12	0000000000001000000000000001100000000001110000000000001111
13	0000111110000010000011111000000000011111000000000011111000
14	1111111100000011111111110000011111111000000011111110000000
15	1111111111111101111111111111101111111111111101111111111111
16	010101010101010101010101010101010101010101010101010101010
17	110110110110110110110110110110110110110110
18	11110111101111011110111101111011110111101111011110111101
19	001001001001001001001001001001001001001001001001001
20	0000000011111111111110000000000000011111111111000000000000

Some Sample Sequences of Takens and Not Takens
(0 = not taken;   1 = taken)
Table 6

the 10.4% of the static branches that were executed over 200 times comprise 48.2% of the dynamic branches.

Many of our predictions of whether a branch will be taken will be contingent on the past behavior (taken/not taken) of that branch. To illustrate such branch behavior, we show in table 6 some sequences of taken/not taken for a number of branches. We observe that for many branches, there are long sequences of either taken or not taken; it is less common to see an alternation. Such a sequence of taken or not taken, we call a "run"; a run is defined as 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. Tables 7 and 8 show the distributions of run lengths for the conditional branches and all branches respectively. The same data is shown weighted by the run length in tables 9 and 10. (That is, tables 9 and 10 show the probability that a given dynamic branch is an element of a run of length n.) As can be seen, most branches occur as parts of long runs.

C. Branch Clustering

Earlier, in section II.B, we described one method of coping with the branch problem called "multiple instruction streams." That 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 the potential occurence of a large number of closely clustered branches, so that it would be impossible to follow all 2**k paths possible from k branches. A measure of the size of k appears in tables 11 and 12. Those tables 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 there is more than one branch unresolved at any one time.

n	IBM CPL	IBM BUS	IBM SCI	IBM SUP	DEC PDP11	CDC 6400
1	0.542	0.448	0.512	0.670	0.504	0.611
2	0.118	0.096	0.174	0.114	0.081	0.192
3	0.063	0.058	0.047	0.048	0.076	0.028
4	0.047	0.047	0.020	0.035	0.046	0.031
5	0.034	0.023	0.050	0.039	0.023	0.015
6	0.021	0.017	0.014	0.015	0.019	0.017
7	0.020	0.017	0.013	0.009	0.018	0.006
8	0.015	0.011	0.011	0.012	0.016	0.005
9	0.011	0.008	0.012	0.007	0.014	0.003
10	0.009	0.012	0.009	0.004	0.009	0.008
11	0.011	0.003	0.013	0.023	0.010	0.045
12	0.007	0.004	0.030	0.003	0.007	0.002
13	0.007	0.002	0.021	0.003	0.006	0.004
14	0.005	0.003	0.009	0.001	0.011	0.004
15	0.006	0.002	0.003	0.001	0.002	0.001
16	0.004	0.003	0.017	0.001	0.007	0.002
>16	0.080	0.226	0.044	0.013	0.151	0.026

Table 7
Distribution of Run Lengths (Conditional Branches)

n	IBM CPL	IBM BUS	IBM SCI	IBM SUP	DEC PDP11	CDC 6400
1	0.517	0.417	0.500	0.674	0.487	0.601
2	0.132	0.099	0.148	0.112	0.142	0.189
3	0.066	0.060	0.040	0.050	0.067	0.033
4	0.049	0.047	0.019	0.034	0.041	0.032
5	0.033	0.022	0.070	0.036	0.028	0.014
6	0.022	0.025	0.016	0.014	0.028	0.017
7	0.024	0.015	0.014	0.009	0.014	0.005
8	0.014	0.017	0.012	0.012	0.016	0.006
9	0.012	0.007	0.014	0.006	0.010	0.003
10	0.009	0.019	0.011	0.004	0.010	0.009
11	0.011	0.030	0.018	0.026	0.009	0.045
12	0.007	0.005	0.025	0.003	0.009	0.002
13	0.007	0.002	0.018	0.003	0.004	0.006
14	0.005	0.002	0.009	0.001	0.005	0.004
15	0.007	0.002	0.013	0.002	0.004	0.001
16	0.004	0.003	0.008	0.001	0.005	0.003
>16	0.081	0.229	0.064	0.013	0.121	0.030

Distribution of Run Lengths (All Types)
Table 8

n	IBM CPL	IBM BUS	IBM SCI	IBM SUP	DEC PDP11	CDC 6400
1	0.069	0.030	0.110	0.218	0.058	0.215
2	0.032	0.014	0.083	0.074	0.019	0.123
3	0.026	0.013	0.025	0.046	0.039	0.021
4	0.024	0.015	0.014	0.045	0.032	0.025
5	0.021	0.010	0.086	0.064	0.014	0.011
6	0.016	0.007	0.014	0.030	0.011	0.024
7	0.018	0.009	0.014	0.022	0.018	0.010
8	0.015	0.006	0.014	0.031	0.015	0.010
9	0.014	0.005	0.013	0.018	0.013	0.005
10	0.013	0.004	0.011	0.014	0.008	0.008
11	0.016	0.002	0.011	0.083	0.009	0.105
12	0.010	0.004	0.021	0.011	0.007	0.003
13	0.012	0.002	0.016	0.010	0.008	0.005
14	0.010	0.003	0.012	0.004	0.015	0.008
15	0.010	0.002	0.008	0.006	0.003	0.001
16	0.008	0.003	0.047	0.007	0.007	0.005
>16	0.685	0.871	0.501	0.317	0.724	0.421

Distribution of Run Lengths Weighted by n (Conditional Branches)
Table 9

n	IBM CPL	IBM BUS	IBM SCI	IBM SUP	DEC PDP11	CDC 6400
1	0.052	0.025	0.074	0.228	0.032	0.130
2	0.027	0.013	0.046	0.075	0.016	0.091
3	0.021	0.013	0.015	0.050	0.017	0.008
4	0.020	0.014	0.011	0.046	0.014	0.018
5	0.016	0.008	0.064	0.061	0.011	0.007
6	0.014	0.009	0.013	0.039	0.014	0.014
7	0.017	0.007	0.013	0.020	0.007	0.008
8	0.012	0.006	0.014	0.034	0.011	0.006
9	0.011	0.004	0.016	0.019	0.006	0.002
10	0.009	0.011	0.017	0.014	0.008	0.008
11	0.013	0.004	0.024	0.096	0.006	0.077
12	0.009	0.003	0.018	0.012	0.011	0.002
13	0.010	0.002	0.016	0.011	0.003	0.004
14	0.008	0.003	0.014	0.005	0.006	0.005
15	0.013	0.002	0.032	0.008	0.004	0.001
16	0.009	0.003	0.015	0.007	0.003	0.004
>16	0.739	0.872	0.598	0.284	0.831	0.614

Distribution of Run Lengths Weighted by n (All Types)
Table 10

n	IBM CPL	IBM BUS	IBM SCI	IBM SUP	DEC PDP11	CDC 6400
0	0.025	0.182	0.405	0.028	0.005	0.509
1	0.119	0.495	0.734	0.140	0.088	0.726
2	0.345	0.722	0.878	0.402	0.347	0.985
3	0.613	0.855	0.959	0.739	0.792	0.994
4	0.840	0.911	0.985	0.940	1.000	0.998
5	0.976	0.983	0.996	0.995	1.000	1.000
6	0.999	0.999	0.999	1.000	1.000	1.000
7	1.000	1.000	1.000	1.000	1.000	1.000

Table 11
Probability of n or fewer Branches in Window of Size 10

n	IBM CPL	IBM BUS	IBM SCI	IBM SUP	DEC PDP11	CDC 6400
0	0.087	0.341	0.553	0.104	0.065	0.603
1	0.370	0.689	0.857	0.430	0.337	0.931
2	0.724	0.875	0.969	0.827	0.937	0.994
3	0.969	0.984	0.995	0.985	1.000	1.000
4	0.999	1.000	1.000	1.000	1.000	1.000
5	1.000	1.000	1.000	1.000	1.000	1.000

Probability of n or fewer Branches in Window of Size 6
Table 12

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

Probability of Correct Branch Prediction, Given Only Op-Code, and Assuming Branch Is Always Either Taken or Not Taken, Based on Op-Code.

Table 13.

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	0.003	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	0.002	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	0.007	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

Distribution of 5 Consecutive Executions (Conditional Branches)
Table 14

V. Branch Prediction

A number of the solutions to the branch problem, as discussed in section II, attempt to predict whether a branch will be taken or not. The general problem can be stated as: what is the value of F(x1,x2,...) where F is the probability that a branch is taken, and x1, x2, ... are parameters on which F may be reasonably conditioned. If F(x1,x2,...) > .5, then we predict that a branch will occur, and conversely. (We note that if the costs of errors of commission are not equal to errors of omission, then the best figure for deciding to predict a branch may not be equal to .5. We discuss this issue below in section VI.A). Of particular interest is x1= operation code, and x2= past execution history of this branch. It is possible to continue with other factors (for x3, x4, etc.) such as other dynamic branches that precede the current dynamic branch (and their execution behavior) [Angi82], other dynamic instructions that precede the current dynamic instruction, the source language of the program, the direction of the branch (e.g. forward/back [Smit81]), etc. For example, certain instruction sequences will generally indicate a taken branch; others will almost always fall through.

Any solution to the branch problem must be implemented in hardware, since it is part of the pipeline and must execute at machine cycle speeds. For that reason, the complexity of the schemes that are practical is very limited, and we will consider primarily only two cases: predictions that depend only on the operation code (F(x1)) and predictions that depend only on the past history of the branch (F(x2)).

The other aspect of branch prediction concerns knowledge of the target address, since delays are encountered even for a correctly predicted taken branch when the target address is not immediately known. We discuss that aspect of the problem toward the end of this (V) section.

A. Prediction Based on Operation Code

Earlier, in tables 2 and 3, we showed the probability that a branch was of a specific op (operation) code, and the probability that for that opcode, whether the branch was taken. These two tables can be easily combined to yield the probability that a branch can be correctly predicted as to taken/not taken, given only the op code. That result is shown in table 13. For example, we note that for the IBM CPL mix, the prediction accuracy rises from 64% (assume all branches are taken) to 66.2% (assume that only BR, B, BAL, BALR, BCT, BXLE, BCR, EX and SVC are taken; all others never taken). While this 2.2% improvement is helpful, we shall see that it is considerably less than can be obtained by predictions based on branch history. (In [Smit81] a range of accuracy for opcode based prediction of 65.7% to 99.4% is obtained, with a mean of 86.7%.)

B. Prediction Based on Past Branch History

Prediction based on past branch history uses the previous sequence of taken/not takens for each branch to predict the taken/not taken behavior of the next occurence of that branch. The most powerful such predictor, of course, is one in which the entire past history of the branch is used to predict the next choice. Such a predictor is infeasible due to the large possible number of such past sequences, so the problem reduces to the following two questions: for a given amount of past history, what prediction accuracy can be obtained, and what is the most desirable amount of past history to retain, given all cost and performance tradeoffs? The basic data for this evaluation is given in tables 14 and 15, where we show the observed probability of all possible sequences of five taken/not taken events (referred to as y1,y2,y3,y4,y5) for conditional and for all branches respectively.

The data in tables 14 and 15 may be used for prediction as follows: 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 conversely (where y1,y2,y3,y4 is the sequence of the four previous dynamic occurences of this static branch). Predictions based on the previous 3 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

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

Distribution of 5 Consecutive Execution (All Types)
Table 15

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

Percentage Correct Guess Using Past n Branches
Using Conditional Probabilities Drawn From Only Given Trace
Table 16

previous 2, 1 or 0 branches can be similarly derived. The accuracy of such predictions are shown in table 16, where in each case, the prediction is based only on the values of F(yi) for that workload. (For one previous branch, the success rate in [Smit81] was from 76.2% to 98.9%, with a mean of 90.4%.)

It is possible to create a composite predictive strategy; that is, a prediction that is based on F(yi), where F(yi) is computed over all six of the workloads used, rather than for just the workload in question. This latter strategy is much more valid, since it is unlikely to be cost effective to vary the predictive strategy on a real computer depending on the program running. In any case, as can be seen in table 17, the predictive accuracy is almost identical to that shown in table 16.

There are a number of interesting observations to be made from tables 16 and 17. First, we note that the predictive accuracy approaches very closely to its maximum with 1, 2 or 3 preceding branches used for prediction. Increasing the amount of history to 4 or 5 branches doesn't seem to add accuracy.

Second, the predictive accuracy for as few as two preceding branches ranges from 83.4% to 97.5%, which is much higher than the accuracy using only the branch type, and no branch history, as in table 13. Finally, we note a very significant variation in the effectiveness of prediction between the various workloads. Most striking is the variation of 83.9% to 97% between the IBM/SUP and IBM/BUS workloads, both of which are for the same architecture. (The authors speculate that the low prediction success rate for the IBM/SUP workload is due to the low probability that a branch is executed repeatedly (see table 4). This is to be expected in supervisor code, in which loops are relatively less frequent.)

C. Prediction Based on Non-Uniform History Retention

Tables 16 and 17, as noted, give the effectiveness of branch prediction when that prediction is based on exactly the n preceding executions of the branch in question, and whether that branch was taken or

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

Percentage Correct Guess Using Past n branches
Using Conditional Probabilities Drawn From Average of All Traces
Table 17

Workload	Figure 5	Figure 6
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 18
Prediction Success of State Diagrams
from Figures 5 and 6.

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%

Fraction of Branch targets
found to have changed from previous
execution of that branch.
Table 19

Lost Cycles

Actual \ Predicted	No Branch	Branch
No Branch	0	k
Branch	m	j

Table 20

..ot taken. Those n preceding executions may be remembered in the branch target buffer with n bits, those n bits representing the possible 2**n possible sequences of preceding taken/not taken.

Given that there are n bits available to use in predicting the next branch, they need not be allocated to show the past n executions, but can record a state that does not map into the precise past history. That is, given a state S(i) (for the branch in question) at time i, we have a function G(S(i)) which yields the prediction T or NT, and a mapping E(S(i),T/NT) -> S(i+1) which maps the current state S(i) and whether the branch is actually taken (T/NT) 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 4 shows the algorithm which uses the past two executions to predict the next, the effectiveness for which is shown in the line labeled "2" in table 17. In figure 4, the states are labeled with the their past history (as a name) and the prediction in force (e.g. NT/T), and each edge shows the transition (mapping E) from state to state depending on whether the branch was taken or not taken.

It is possible to suggest other mappings E and functions G than those based on the last n executions of the branch. Two such are shown in figures 5 and 6.

In figure 5, we show an algorithm in which two errors are required to change the prediction. That is, assume that the current prediction is N and the last two 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% wrong predictions; when started in either of the other two states, the predictions will be 50% wrong.

Another algorithm in shown in figure 6. In this case, two wrong guesses are again required to change the prediction, but two are also required to return to the previous prediction. (In figure 5, one step could return to the previous prediction after two èrrors.) (This algorithm was proposed for the S-1 [Widd77]; see section II.G and VI.G). It can be seen that the sequence NNTTNNTTNNTT... can cause every prediction to be incorrect.

In both figures 5 and 6, close examination will show that the states indicated do not correspond exactly with the previous two branches. For example, state n of figure 5 implies a past history of NN, whereas state n? implies a past history of NNT or TNN.

The success of the algorithms represented in figures 5 and 6 are shown in table 18. Comparing the two, we see that their results are almost identical. Further comparison with the line labeled "2" of table 17 shows that in most cases, (5 workloads), the algorithms of figures 5 and 6 are very slightly better. In the one remaining case, the IBM Supervisor workload, the earlier results are 3% better. This latter effect is likely due to the fact that branches in supervisor code are much less frequently used for loop control than in user programs.

It is possible to consider all possible functions G and mappings E for n bits of state in order to derive the optimal algorithm. We have not done so, and the results in tables 17 and 18, and the comparison between them suggests that such an exercise would yield very little, if any, improvement.

D. 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. It should be clear that the branch target can be obtained only by either computing it directly from the instruction, or remembering it from the past execution and assuming that it will be the same. Since the purpose of the BTB is to immediately predict the target, the previous target must be remembered. While target changes are likely to be infrequent, they will sometimes occur. In particular, a target will change if the source (higher level language program) contains a computed goto or a case statement. Execute instructions (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 found to be 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 not occur. (This suggests that perhaps a branch whose target has been found to change previously should not predict a branch. We believe that predicting a branch is better, if the cost of an incorrect prediction is the same as the cost of an incorrect fall through. That is because a fall through is very unlikely, whereas the target need not always change.)

Table 19 shows the fraction of all dynamic branches executed for which the branch is taken with a target address different than that previous. Some of 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, predict no branch and none occurs) are not affected.

E. Writes Into The Instruction Stream

The branch target buffer is designed to search for 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. There are two ways to deal with this problem. First, and most correct, the instruction in question, identified by the BTB, can be tagged (as it moves down the pipeline) with a bit specifying "branch". If the instruction decode stage finds that the instruction is not a branch, then the pipeline can be flushed of the following instructions and reloaded, and the BTB can either be flushed or just the 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 modify the instruction stream.

F. Extensions and Alternatives

Above, we defined a general mechanism for predicting branches and showed some results for the more important cases. There exist some cases we haven't considered, and some improvements have been suggested in the literature. We note some of those in this section.

In [Pome80a], it is suggested that a machine be built so that both the taken and not taken directions can be followed (as in section II.B above, "multiple instruction streams"). Then, if a change in locality is detected (when there are instruction misses in the CPU cache), the multiple 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. See also [Pome80b].

[Smit81] proposes a strategy (his 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. He reports poor performance. (Smith reports on the effectiveness of a number of his other "strategies", but in many cases, those 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. Some of his other ideas are: Keep a table of recently used not taken branch instructions. (This, of course, fails to retain branch targets for successful branches, and so can be of only limited use). Keep a taken/not taken bit in the cache (as in section II.G). Use a hashed BTB with one bit prediction. Use the same design, but with a two bit predictor (as in [Widd77].)

An interesting use of the branch target buffer is described in [Dris81]. An address-generate interlock (AGI) in a pipeline is a logical ependency 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 due to 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. See also [Losq82a].

One additional use to which a branch target buffer or similar buffer can be put 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) which could be indirect (either by tag in instruction or by tag in target) are matched against an "indirect buffer", then the ultimate target of an indirect could be fetched in one step. The branch target buffer could serve double duty for this purpose, 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.

VI. Branch Target Buffer Implementation Issues

A. Performance Costs and Optimal Predictions

Thus far in this paper, we have assumed that the performance impact of the branch target buffer is as follows: 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 need be true.

Specifically, it is quite possible that a prediction of a taken branch will always cost a small number of machine cycles. The reason is that a taken branch is out of sequence and storage access time (cache or _ain memory) may be long enough that the target cannot be fetched before the instruction decode stage of the pipe is ready for it. Assume that j such cycles are lost for every predicted branch. (See table 20).

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 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 (i.e. the fall through instruction) may already be on hand. Assume that the cost of an incorrect positive (predict taken) prediction is k cycles and an incorrect negative (not taken) prediction costs m cycles.

There are four events of interest, as noted above: predict no branch and no branch occurs, predict no branch and branch occurs, predict branch and none occurs, predict branch correctly. (We are omitting the target change case here for simplicity.) The costs for these events are respectively 0, m, k, j. Previously, we assumed m=k and j=0. In that previous 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 it is possible that the optimal strategy does not have the highest prediction accuracy.

Because the values of m, k and j are very implementation dependent, we have not developed strategies for the case of this (VI.A) section. Such strategies can easily (but tediously) be generated, given the costs m, k and j, from tables 14 and 15. This is done as follows: 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 taken. 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, then the cost is $p*m$. The correct prediction to make is the one with the lower expected cost.

_. Branch Target Buffer Size and Hit Ratio

The branch target buffer, like the CPU cache or the translation lookaside buffer (TLB), is a small high speed memory, and for both cost and performance reasons, must be of limited size. Our analysis thus far has always assumed that the BTB was unboundedly large, and could hold all previously executed branches; of course, this cannot be true. In this section, we examine the effect of the BTB being only a finite size.

Define the hit ratio of the BTB to be the probability that a branch is found to be in the BTB at the time it is fetched. Then the hit ratio of the BTB depends on two algorithms: 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 to not enter branches in the BTB if they are not taken, given that the BTB is now of finite size.

In this paper we have used a "fetch all" algorithm; that is, whenever a branch is recognized, it is entered in the BTB if it is not already there. For replacement, we have used global LRU [Matt71] (remove the least recently executed branch resident in the BTB). (The replacement algorithm can 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. I.e. if replacement is LRU, then the branch entry is not moved to the top of the LRU stack.)

The hit ratios for various BTB sizes, given "fetch all" and global LRU replacement, are shown for each workload in table 21. As may be seen, the miss ratio varies widely. For example, for a 256 entry BTB, the hit ratio varies from a low of 61.5% (for the IBM supervisor workload) to 99.7% hits for the CDC-6400 programs. These results are qualitatively similar to the relative cache hit ratios [Smit79] for the various types of programs, as one would expect. (In [Widd77] it is reported that 16 to 32 entries in a BTB yield over 50% misses for S-1 traces.)

n entries	IBM CPL	IBM BUS	IBM SCI	IBM SUP	DEC PDP11	CDC 6400
1	0.031	0.121	0.158	0.012	0.012	0.221
2	0.075	0.150	0.223	0.066	0.168	0.338
3	0.150	0.192	0.253	0.124	0.235	0.370
4	0.185	0.212	0.272	0.154	0.292	0.420
6	0.267	0.241	0.342	0.252	0.326	0.511
8	0.298	0.247	0.402	0.259	0.377	0.522
11	0.324	0.260	0.575	0.265	0.428	0.582
16	0.369	0.315	0.636	0.275	0.599	0.682
23	0.455	0.325	0.677	0.283	0.683	0.763
32	0.514	0.412	0.693	0.301	0.847	0.818
45	0.568	0.450	0.775	0.374	0.898	0.870
64	0.634	0.624	0.812	0.435	0.928	0.878
91	0.701	0.752	0.888	0.486	0.942	0.966
128	0.769	0.832	0.953	0.545	0.960	0.996
181	0.835	0.881	0.964	0.589	0.978	0.996
256	0.888	0.929	0.970	0.615	0.989	0.997
362	0.934	0.960	0.976	0.650	0.994	0.997
512	0.967	0.968	0.993	0.672	0.998	0.997

Table 21
Branch Target Address Buffer Hit Ratio
(Organized as a LRU Stack)

Buffer Size	Set Size								
	1	2	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 22
Branch Target Buffer Hit Ratios
(IBM/CPL Mix)

	IBM CPL	IBM BUS	IBM SCI	IBM SUP	DEC PDP11	CDC 6400
NoFlush	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

Table 23
Comparison of Percentage of Correct Guesses
in a Multiprogramming Environment

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

A major effect of the limited size of the branch target buffer is to decrease or remove its advantages over other "branch problem" solutions, as discussed in section II. 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 of the fact that the branch target address is not immediately available, the higher hit ratio may be sufficient to compensate.

C. Buffer Addressing and Organization

The branch target buffer is accessed associatively; the address of the instruction fetch is matched with the instruction address fields in the BTB. If there is a match, then 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 [Cont69], [Smit78a] or hashed [Knut73] as is done for most TLBs [Smit79]. In the former case, some middle bits of the address of the instruction are used to select a "set", and then 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 (see [Smit79] for a discussion of ways to do this), 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 experiments in [Smit79] showed the two methods to be about equally effective, we select the standard set associative mapping as simpler, cheaper and faster. ([Smit81] uses hashing in one of his strategies.)

The effect of the set size is shown for the IBM/CPL mix in table 22. (For the other mixes, the effects are available in [Lee82].) It can be seen there and in [Smit79], [Smit78a] that set sizes of 4 or 8 are sufficiently large and closely approach the hit ratio of the fully associative design.

D. The Effect of Multiprogramming

Multiprogramming is important both to the design and the 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 and incorrect predictions will take place. Each such incorrect prediction will have to be corrected. Since many of these 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 would suggest. As a very worst case example, we show the data in table 23. There we compare the fraction of correct predictions using an infinite BTB with those from an infinitely large BTB which is flushed every 1000 instructions. As may be seen, these frequent flushes significantly impact performance. It is the impression of the authors, however, that address space switches will occur at intervals closer to 5000 to 25000 instructions than to 1000. Therefore, the BTB flushes may have less of an effect on the miss ratio than the finite size of the BTB.

It should be kept in mind that 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. (Fast methods for flushing TLBs are discussed in [Smit79].)

E. 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 limited the range of alternatives considered to those that are simple enough and cheap enough to implement. Further, we have looked at the effect of BTB size and organization for the same reason. Anyone proposing to either design a BTB or to study them further should keep in mind this important constraint.

F. MU-5 Implementation and Results

The MU-5 Computer System [Morr79] uses a branch target buffer; its effectiveness is discussed in [Holg80]. The BTB retains up to 8 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% of the instructions from Algol and Fortran executions respectively. The BTB correctly predicts from 40% (Algol compilation) to 65% (Algol execution) of the correct sequences after a branch (including fall throughs) as compared to 15% to 25% without the BTB.

G. S-1 Trace Experiments

In [Widd77] some branch target buffer experiments on S-1 traces are reported. Success rates range from 91% to 95% with 1 to 5 bit predictors, and using the method of figure 6 above. The effectiveness of the figure 6 scheme varies from worse than the 1 bit predictor to almost as good as the four bit predictor. These experiments were run on two traces of about 100,000 instructions.

H. Use For Tracing

In some computers, circular buffers are maintained of the last n instructions (or branches) executed. The contents of these buffers are useful for debugging both hardware and software. It is worth noting that the branch target buffer can be combined in function with the circular branch buffer [Brow79,Boni81].

VII. Overall Branch Target Buffer Effectiveness

The reason to build a branch target buffer is to improve CPU performance. The results above 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. We consider a few typical cases in this section.

One example is to consider the IBM/CPL mix, for which we can predict the branch path with an accuracy of 93.8%, using the predictor of figure 5. A hit ratio of 86.5% is obtained with a BTB consisting of 128 sets of 4 entries each. Up to 4.2% of our predictions will be incorrect due to target changes. This gives an overall minimum prediction accuracy of (93.8-4.2)(.87) = 78%.

The prediction accuracy can be used to estimate the performance impact by considering a real machine. As an example, we use the Amdahl 470V/6 [Amda76], which has a machine cycle time of 32.5 nsec, and runs at about 4 MIPS [Peut77]. Excluding memory access delays, 5 MIPS is closer and that is the figure we use. That yields a mean of 6 cycles per instruction. Each branch taken results in a delay of 4 machine cycles. Assume that the branches are 30% of the instructions, and 65% of the branches are taken. Excluding the branch penalty, then, the mean execution time for an instruction would be $t = 6-(0.3)(0.65)(4) = 5.22$ 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, this gives a performance improvement of 9.5%.

The above computation, using the same basic figures, has been replicated, varying each parameter of interest, one case per table, and the results appear in tables 24, 25, 26 and 27. They show respectively the percentage improvement for different basic instruction execution times, the percentage improvement for different time penalties when the wrong stream is processed after an unresolved branch, and percentage improvement for different hit ratios in the BTAB with basic instruction times of 5.22 and 2 cycles.

We observe, from table 24, that the BTAB is most effective when the cost of an incorrect guess is large relative to the mean instruction time. That result confirmed in table 25, which varies the other parameter of that pair. Table 26 shows that the hit ratio to the BTAB is important. Its importance rises, as seen in table 27, when the machine cycle time is short.

VIII. Summary and Conclusions

Taken branches have long been known to be one of the major obstacles to high efficiency in a pipelined computer system [Scho71,Flyn66]. 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.). In this paper, we have discussed the branch target buffer, which we believe is the most effective way to minimize branch penalties.

Our study of the branch target buffer 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 past history and opcode, and have measured the effectiveness of the important variants of this predictor. Our results show that 2 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 machine.

Consideration has also been given to various implementation issues, such as the design of the BTB addressing (set associative), the effect of

Basic Instruction Time	Instruction Time without BTB	Instruction Time with BTB	Percentage Improvement
2	2.78	2.26	22.8
3	3.78	3.26	15.8
4	4.78	4.26	12.1
5	5.78	5.26	9.8
6	6.78	6.26	8.2

Table 24
Percentage Performance Improvement with BTB
(Correct Guess Probability = 0.78
Incorrect Guess Penalty = 4 machine cycles)

Incorrect Guess Penalty	Instruction Time without BTB	Instruction Time with BTB	Percentage Improvement
0	5.22	5.22	0.0
2	5.61	5.35	4.8
4	6.00	5.48	9.5
6	6.39	5.61	13.9
8	6.78	5.74	18.1

Percentage Performance Improvement with BTB
(Correct Guess Probability = 0.78
Basic Instruction Time = 5.22 machine cycles)
Table 25

BTB hit Probability	Percentage Correct Guess	Instruction Time with BTB	Percentage Improvement
1.00	89.9	5.34	12.3
0.95	85.4	5.40	11.2
0.90	80.9	5.45	10.1
0.85	76.4	5.50	9.0
0.80	71.9	5.56	8.0
0.75	67.4	5.61	6.9
0.70	62.9	5.67	5.9

Percentage Performance Improvement with BTB
(Basic Instruction Time = 5.22 machine cycles
Incorrect Guess Penalty = 4 machine cycles)
Table 26

BTB Hit Probability	.ercentage Correct Guess	Instruction Time with BTB	Percentage Improvement
1.00	89.9	2.12	31.0
0.95	85.4	2.18	27.8
0.90	80.9	2.23	24.7
0.85	76.4	2.28	21.8
0.80	71.9	2.34	18.9
0.75	67.4	2.39	16.3
0.70	62.9	2.45	13.7

Percentage Performance Improvement with BTB
(Basic Instruction Time = 2 machine cycles
Incorrect Guess Penalty = 4 machine cycles)
Table 27

multiprogramming on the hit ratio, the need to flush the BTAB when the address space changes, and the problems of branch target changes and writes into the instruction stream.

The use of six different workloads, taken from three different machines, gives us reason to think that our results are representative of the results to be generally expected and we believe that this paper will have direct application to high speed computer system design. A number of extensions to the basic branch target buffer design were mentioned, including the use of the BTB or another similar buffer to avoid penalties from indirect addressing. Improvements in CPU performance of from 5% to 20% can be expected when comparing a BTB design to a similar CPU design without one.

Acknowledgements

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

BIBLIOGRAPHY

[Ager82] T. K. M. Agerwala, "Simple Algorithm for Locking Short Loops in an Instruction Buffer", IBM Tech. Disc. Bull., 25, 1, June, 1982, pp. 56-58.

[Amda76] Amdahl 470 V/6 Machine Reference Manual, Amdahl Corp., Sunnyvale, CA, 1976.

[Ande67] D.W. Anderson, F.J. Sparacio, R.M. Tomasulo - "The Model 91: Machine Philosophy and Instruction Handling" - IBM J. Research and Development 11, pp. 8-24, January, 1967.

[Angi80] J. M. Angiulli, D. J. Friedman Jr., L. C. Garcia, and S. G. Tucker, "Branch Direction Prediction Mechanism", IBM Tech. Disc. Bull., 23, 1, June, 1980, pp. 268-269.

[Bela66] L. A. Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer", IBM Sys. J., 5, 2, 1966, pp. 78-101.

[Berl79] E. R. Berlekamp, presentation at CS Division Seminar, UC Berkeley, 1979.

[Boni81] D. Boniface, J. C. Caudrillier, P. Thery and G. Toubol, "Central Control Unit Branch Trace Mechanism", IBM Tech. Disc. Bull., 24, 7A, December, 1981, pp. 3503-3505.

[Brow79] W. J. Brown, H. Ovies, and J. L. Pooler, "Circuit for Tracing Branch Instructions", IBM Tech. Disc. Bull., 22, 7, December, 1979, pp. 2651-2654.

[Cont69] C. J. Conti, "Concepts for Buffer Storage", IEEE Computer Group News, March, 1969, pp. 9-13.

[CDC73] STAR-100 Hardware Reference Manual 60256000, Control Data Corporation, Arden Hills, Minn. 1973.

[CDC74] Control Data 6000 Series Computer System Reference Manual 60100000, Control Data Corporation, Arden Hills, Minn. 1974.

[CDC75] Control Data 7600 Hardware Reference Manual 60367200, Control Data Corp., Arden Hills, Minn. 1975.

[Cray76] Cray-1 Computer System Reference Manual 2240004, Cray Research Inc, Bloomington MN, 1976.

[Dris81] G. C. Driscoll, J. J. Losq, T. R. Puzak and J. J. Shedletsky, "Address Generate Interlock Avoidance for Branch Instructions in a Branch-History-Table Processor", IBM Technical Disclosure Bulletin, 24, 1A, June, 1981, pp. 350-354.

[Flyn66] M.J. Flynn - "Very High-Speed Computing Systems" - Proc IEEE, 54, pp. 1901-1909, Dec 1966.

[Garc80] L. C. Garcia and T. Huynh, "Storage Fetch Contention Reduction by Using Instruction Branch Prediction", IBM Technical Disclosure Bulletin, 23, 6, November, 1980, pp. 2404-2405.

[Hail79] Brent T. Hailpern and Bruce L. Hitson, "S-1 Architecture Manual", Computer Systems Laboratory Report STAN-CS-79-715, January, 1979, Stanford University, Stanford, Ca.

[Holg80] R. W. Holgate and R. N. Ibbett, "An Analysis of Instruction Fetching Strategies in Pipelined Computers", IEEETC, C-29, 4, April, 1980, pp. 325-329.

[Hugh81a] J. F. Hughes, "Branch on Condition Decoding with Instruction Queues Empty", IBM Technical Disclosure Bulletin, 24, 4, September, 1981, pp. 1857-1858.

[Hugh81b] J. F. Hughes, J. J. Losq, G. Rao, R. N. Rechtschaffen, H. E. Sachar and D. Slawitschek, "Instruction Fetch", IBM Tech. Disc. Bull., 24, 4, September, 1981, pp. 1859-1860.

[Ibbe78] R.N. Ibbett, P.C. Capon - "The Development of the MU5 Computer System" - CACM, 21, 1, pp. 13-24, January, 1978.

[Ibbe72] R.N. Ibbett - "The MU5 Instruction Pipeline" - Computer Journal 15, 1, pp. 42-50, February, 1972.

[IBM78] IBM Maintenance Library 3033 Processor Complex Theory of Operation Diagrams Manual, Volume 2, Instruction Preprocessing Function (SY22-7002), Volume 3, E-Function Devices and Interruptions (SY22-7003), Volume 1, Introduction and Instruction Execution (SY22-7001), January, 1978, IBM Corp., Poughkeepsie, N. Y.

[IBM70] IBM Field Engineeering Theory of Operation System/360 Model 195, System Introduction, Instruction Processor, SY22-6855, August, 1970, IBM Corp., Poughkeepsie, New York.

[IBM73] IBM Maintenance LIbrary System/370 Model 168 Theory of Operation/Diagrams Manual, Volume 2, I-unit, SY22-6932, IBM Corp., Poughkeepsie, New York, 1973.

[Knut73] D. Knuth, "The Art of Computer Programming, Volume 3, Sorting and Searching", Addison Wesley, Reading Massachusetts, 1973.

[Kogg81] Peter M. Kogge, "The Architecture of Pipelined Computers", Mc-Graw Hill, Washington D. C., 1981.

[Lee82] J. K. Lee, "Performance Improvement of CPU Pipelines", Ph.D. Dissertation, to appear, 1982.

[Lile79] A. G. Liles Jr., and B. E. Willner, "Branch Prediction Mechanism", IBM Technical Disclosure Bulletin, 22, 7, December, 1979, pp. 3013-3016.

[Losq82a] J. J. Losq, "Address Generate Interlock Memory Buffer", IBM Tech. Disc. Bull., 25, 1, June, 1982, pp. 114-120.

[Losq82b] J. J. Losq and G. S. Rao, "Zero Condition Code Detection for Early Resolution of BCS and BCRS", IBM Tech. Disc. Bull., 25, 1, June, 1982, pp. 130-133.

[Losq82c] J. J. Losq, "Generalized History Table for Branch Prediction", IBM Tech. Disc. Bull., 25, 1, June, 1982, pp. 99-101.

[Matt70] R.L. Mattson, J. Gecsei, D.R. Slutz, I.L. Traiger - "Evaluation Techniques for Storage Hierarchies" - IBM Sys J, 9, 2, pp. 78-117.

[Morr79] D. Morris and R. N. Ibbett, "The MU5 Computer System", Springer-Verlag, New York, 1979.

[Patt81] David A. Patterson, and Carlo H. Sequin, "RICS-1: A Reduced Instruction Set VLSI Computer", Proc. Eighth Annual Symposium on Computer Architecture, May, 1981, pp. 443-458.

[Peut77] B.L. Peuto, L.J. Shustek - "An Instruction Timing Model of CPU Performance" - Proc. Fourth Annual Sumposium on Computer Architecture, College Park, Md, pp. 165-178, March, 1977.

[Pine79] Michael Lawrence Pinedo, "Some Models in Scheduling and Queueing", Ph.D. Dissertation, IEOR Dept., University of California, Berkeley, Ca., 1979.

[Pome80a] J. H. Pomerene and R. N. Rechtschaffen, "Dynamic Branch Prediction Using Branch History Table", IBM Technical Disclosure Bulletin, 22, 8A, January, 1980, p. 3437.

[Pome80b] J. Pomerene and R. Rechtschaffen, "Reducing Cache Misses in a Branch History Table Machine", IBM Technical Disclosure Bulletin, 23, 2, July, 1980, p. 853.

[Radi82] George Radin, "The 801 Minicomputer", Proc. Symp. on Arch. Support for Programming Languages and Operating Systems", March, 1982, Palo Alto, Ca., pp. 39-47. (Available as Sigarch Computer Architecture News, 10, 2, March, 1982, and as IBM Research Technical Report RC 9125, November, 1981.)

[Rama77] C.V. Ramamoorthy, H.F. Li - "Pipeline Architecture". Computing Surveys, 9, 1, March, 1977, pp. 61-102.

[Rao82] G. S. Rao, "Technique for Minimizing Branch Delay Due to Incorrect Branch History Table Predictions", IBM Tech. Disc. Bull., 25, 1, June, 1982, pp. 97-98.

[Rau77] B.R. Rau, G.E. Rossman - "The Effect of Instruction Fetch Strategies Upon The Performance of Pipelined Instruction Units" - Proc 4th Annual Symp Comp Arch, 5, 7, pp. 80-89, March, 1977.

[Rise72] E.M. Riseman, C.C. Foster - "The Inhibition of Potential Parallelism by Conditional Jumps" - IEEETC, C-21, pp. 1405- 1411, December, 1972.

[Russ78] R.M. Russell - "The Cray-1 Computer System" - CACM 21, 1, pp. 63-72, January, 1978.

[Scho71] H. Schorr, "Design Principles for a High Performance System", Proc. Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, April 13-15, 1971, pp. 165-192.

[Sher72] S. Sherman, F. Baskett and J. Browne, "Trace Driven Modeling and Analysis of CPU Scheduling in a Multiprogramming System", CACM, 15, 12, December, 1972, pp. 1063-1069.

[Smit78a] A.J. Smith - "A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory" - IEEETSE SE-4, 2, pp. 121-130, March, 1978.

[Smit78b] A.J. Smith - "Sequential Program Prefetching in Memory Hierarchies" - IEEE Computer, 11, 12, pp. 7-21, December, 1978.

[Smit78c] Alan Jay Smith, "Bibliography on Paging and Related Topics", Operating Systems Review, 12, 4, October, 1978, pp. 39-56.

[Smit79] Alan Jay Smith, "Cache Memories", 1979, to appear, Computing Surveys, 1982.

[Smit81] James E. Smith, "A Study of Branch Prediction Strategies", Proc. 8'th Annual Symposium on Computer Architecture, May, 1981, Minneapolis, Minn., (Sigarch Newsletter, 9, 3, 1981), pp. 135-148.

[Spir77] Jeffrey R. Spirn, "Program Behavior: Models and Measurements", Elsevier, New York, 1977.

[TI76] The ASC System Central Processor, Texas Instruments Corp., December, 1976, 929982-11.

[Thor64] J.E. Thornton - "Parallel Operation in the Control Data 6600" - AFIPS Proc FJCC 26 pp. 33-40, 1964.

[Widd77] Lawrence C. Widdoes, Jr., "Jump Prediction", (draft) - February, 1977, unpublished.

[Wood79] Lowell Wood, L. Curtis Widdoes Jr., Erik Gilbert, Steven Correll, Richard Kovalcik, "1979 Annual Report, The S-1 Project, Volume I, Architecture", Lawrence Livermore Laboratory, Livermore, Ca. 1979.

[Yamo80] J. Y. Yamour, "Instruction Scan for an Early Resolution of a Branch Instruction", IBM Technical Disclosure Bulletin, 23, 6, November, 1980, pp. 2600-2604.

[Yann77] N.A. Yannacopoulos, R.N. Ibbett, R.W. Holgate "Performance Measures of the MU5 Primary Instruction Pipeline" - Information Processing 77, Proc IFIP Congress, Toronto, pp. 471-476, August, 1977.

Appendix I

IBM System/370 Instruction Traces

1. Compiler Mix (IBM/CPL) (9,719,849 instructions)

PLIC.BILL	PL1 compile of a report and billing program
COBC.SUM	COBOL compile of a job step summary program
PLIC.SAM	PL1 compile of a simulation program
FORC.	FORTRAN-H compile

2. Business Mix (IBM/BUS) (11,930,003 instructions)

COBG.UPD	COBOL go step of a master file update program
COBG.TVA	COBOL go step of comparative analysis of power plant alternatvies
PLIG.SMF	PL1 go step of SMF billing program

3. Scientific Mix (IBM/SCI) (25,487,392 instructions)

FORG.STA	FORTRAN go of single precision stress analysis
FORG.EM	FORTRAN go of an EM wave computation program
FORG.FFT	FORTRAN go of a FFT computation program
FORG.NUM	FORTRAN go of double precision numerical analysis
FORG.SAT	FORTRAN go of double precision analysis of satellite information

4. Supervisor State Mix (IBM/SUP) (13,974,553 instructions)

NL0466	All three traces are of the IBM MVS operating
NL0309	system with a commercial type workload.
NL0582	

PDP-11/70 traces (8,995,386 instructions)

ED.C	Execution of the line editor, ed, in UNIX. ed is written in the language C and compiled.
ROFF.AS	Execution of the text formatter, roff, in UNIX. This program is written in 11/70 assembler code.
PLOT.F	Execution of a printer plotter program. The program was written in FORTRAN and compiled with the fc compiler of UNIX. The compiler generates intermediate codes which are interpreted by a runtime program.
OS.C	Execution of an operating system used by an undergraduate course in operating system design. The program is written in C and compiled by the cc compiler of UNIX
SIMPIPE.FOR	Execution of a pipeline simulation program written in FORTRAN and compiled with the FTN compiler.

CDC 6400 traces (23,818,580 instructions)

CMOTS.FORTG	FORTRAN go of a MOS circuit analysis program. The program was compiled with the RUN compiler.
TRACE.ASM	Execution of the tracer tracing the go step of a FORTRAN program which does curve fitting.
TWOD.FORTG	FORTRAN go of a program that solves the two dimensional scattering problem of an infinite circular cylinder.
CAPPA.S.FORTG	FORTRAN go of a phase plane analysis program solving a set of two simultaneous differential equations. The trace was collected to include the program startup portion.
CAPPA.L.FORTG	Same as CAPPA.S.FORTG execpt than tracing began after the program has gone into the iteration loop.
DIPOLE.FORTG	FORTRAN go of a program that solves the three dimensional scattering problem of a cube using the dipole approximation technique.

Appendix II

System 370 Branch Instructions

BAL	Branch and Link. Used to make subroutine calls.
BALR	Branch and Link Register. Similar to BAL. Sometimes used only to load base registers and as such the branch will not be taken.
BCT	Branch on Count equal to zero. Used for loops.
BCTR	Branch on Count Register. Similar to BCT. Often used as decrement instruction and as such the branch will not be taken.
BXH	Branch on Index High. For loop control.
BXLE	Branch on Index Low or Equal. For loop control.
BC,BCR	Branch on Condition. A 4 bit mask determines the branch condition.
B,BR	Unconditional Branch. These are actually BC and BCR respectively, with the condition mask set equal to 15.
EX	The single instruction at the operand address is executed. Subsequent instructions following the EX instruction are processed.
SVC	Generate a Supervisor Call Interrupt to pass control to
LPSW	The operand data is loaded as the new Program Status Word.
MC	Monitor Call Interrupt. This instruction is similar to the SVC interrupt instruction except that the interrupt is vectored through a different location in memory. This instruction is used by system software to pass control to various modules.

The branch instructions of the PDP-11/70 are grouped as follows:

JSR	Subroutine call
SOB	Subtract one and branch if zero
BGET	Signed conditional branch
BVCS	Carry, Overflow conditional branch
BHSL	Unsigned conditional branch
BNEQ	Equal/Not Equal to zero conditional branch
RTS	Return from subroutine
JMP	Unconditional absolute address branch
BR	Unconditional relative address branch
TRAP	System call

The branch instructions of the CDC 6400 are grouped as follows:

RJ	Return Jump, subroutine call
JP	Unconditional jump
XJ	Register Xj conditional jump
EQ	Jump on register Bi=Bj
NE	Jump on register Bi~=Bj
GE	Jump on register Bi>=Bj
LT	Jump on register Bi<Bj
SYS	Call to system

Branch instructions can be divided into functional types:

type	I	unconditional branch - always taken or always no
	II	subroutine call - always taken
	III	loop control - usually taken (loop back)
	IV	decisions - either way
	V	computed goto - always taken (changing target)
	VI	supervisor calls - always taken
	VII	execute - always taken

Some of our analysis makes reference to "Conditional Branch Instructions". Conditional branch instructions for the IBM traces consist of Op-codes BC and BCR; the PDP traces consist of BGET, BVCS, BHSL, and BNEQ; the CDC traces consist of XJ, EQ, NE, GE, and LT.