10 / 16 [illegible] 95 ①

UCRL-ID-106077
PREPRINT

Studies in Branch-Prediction

Carl Ponder
Lawrence Livermore National Laboratory
Livermore, CA

September 11, 1990

This report has been reproduced
directly from the best available copy.

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information
P.O. Box 62, Oak Ridge, TN 37831
Prices available from (615) 576-8401, FTS 626-8401.

Available to the public from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd.,
Springfield, VA 22161

Price Code	Page Range
A01	Microfiche
Papercopy Prices	
A02	1- 10
A03	11- 50
A04	51- 75
A05	76-100
A06	101-125
A07	126-150
A08	151-175
A09	176-200
A10	201-225
A11	226-250
A12	251-275
A13	276-300
A14	301-325
A15	326-350
A16	351-375
A17	376-400
A18	401-425
A19	426-450
A20	451-475
A21	476-500
A22	501-525
A23	526-550
A24	551-575
A25	576-600
A99	601 & UP

UCRL-ID--106077

DE91 000499

Forward

September 11, 1990

The following four papers describe work I performed on the *branch-prediction* problem. Originally Mike Shebanow brought the problem to my attention, while developing an improved branch-predictor for the HPS design at UC Berkeley. It occurred to me to derive upper-bounds on the predictability of branches, based on the characteristics of the predictor; the bounds provide a way to judge the quality of a given branch-predictor, and of knowing when certain design constraints must be violated to achieve a requisite level of prediction accuracy. Many of the ideas in the first and second papers came from discussions with Mike Shebanow, who also provided the trace data.

Previous work in this area concentrates on presenting specific branch-predictors and evaluating them on trace data. This *trace-driven simulation* approach has made outstanding progress in improving methods of branch-prediction. *Trace-driven analysis* or *trace-driven optimization* is a refinement applied here, to study the problem of improving the predictions as much as possible. I expect to apply methods of trace-driven analysis to other problems in the future. The results are limited by the trace data available; although the comprehensive data from the Lee & Smith study is on magtape, it is unreadable, and no one else was willing to provide anything beyond more UNIX traces.

Deriving bounds is not always easy. The Moore-machine prediction problem described in the third paper must be solved; the brute-force solution described in the fourth paper constructs optimal machines as a by-product. These optimal machines, or *finite-state superpredictors,* appear in the first paper, which suggests that an architect should design a superpredictor from program traces and incorporate it in improved CPU designs. An efficient solution to the Moore-machine prediction problem would help considerably.

-- Carl Ponder,
Lawrence Livermore National Laboratory

This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. Indirect support was provided by the National Science Foundation under grant number CCR-8812843; Army Research Office, grant DAAG29-85-K-0070, through the Center for Pure and Applied Mathematics, University of California at Berkeley; the Defense Advanced Research Projects Agency (DoD) ARPA order #4871, monitored by Space & Naval Warfare Systems Command under contract N00039-84-C-0089, through the Electronics Research Laboratory, University of California at Berkeley; the IBM Corporation; and a matching grant from the State of California MICRO program.

ℓb

Table of Contents

List of Figures

List of Tables

List of Formulas

An Information-Theoretic Look at Branch-Prediction

Carl G. Ponder

Computing Research Group, L-419
Lawrence Livermore National Laboratory
P.O. Box 808
Livermore, CA 94550
(415) 423-7034

Michael C. Shebanow

88000 Advanced Processor Design Group
Motorola, Inc. - OE318
6501 Wm. Cannon Drive West
Austin, TX 78735

Abstract

Accurate branch-prediction is necessary to utilize deeply pipelined and Very Long Instruction-Word (VLIW) architectures. For a set of program traces we show the upper limits on branch predictability, and hence machine utilization, for important classes of branch-predictors using static (compiletime) and dynamic (runtime) program information. A set of optimal "superpredictors" is derived from these program traces. These optimal predictors compare favorably with other proposed methods of branch-prediction.

Index Terms: branch prediction, CPU performance, information-theoretic bounds, Moore machine, pipeline optimization.

1 Motivation

The majority of modern high-speed computer architectures employ pipelining as a speedup mechanism. Pipelining subdivides the work of an individual instruction into a sequence of stages, and overlaps the execution of successive instructions by executing different stages of different instructions simultaneously. Future systems will use pipelining to larger and larger degrees.

Conditional branch instructions potentially interrupt the smooth execution of a pipeline – the pipeline may be ready to process instructions from the destination of the conditional branch *before* the condition has been evaluated. Null instructions may be passed through the pipeline until the branch-condition is resolved and instructions from the correct destination are ready to be fetched. This sequence of null instructions, referred to as a *bubble*, inhibits pipeline utilization by filling pipeline stages but performing no work.

An alternative to passing pipeline bubbles is *branch-prediction*, where the result of a branch condition is *guessed* before it is fully evaluated. Instructions from the assumed branch destination are processed immediately. Some repair work is necessary if the guess was incorrect, to erase the effect of executing the wrong sequence of instructions. It is interesting to note that fairly simple schemes of guessing are reasonably accurate. Treatments of pipelining and pipelined machines are found in [3] [8] [16]. The specific problem of branch-prediction is treated in [4] [6] [7] [15].

The *Very Long Instruction-Word (VLIW)* architectures perform simultaneous instruction execution, and also benefit from branch-prediction. A program is compiled into a number of instruction streams which execute in lock-step. Analysis of the program at compiletime and in sample executions is used to detect parallelism. Some program transformations are employed to improve parallelism, notably the movement of instructions across conditional branches. If a branch condition is deemed likely to hold, or likely to not hold, instructions from the favored destination may be moved to execute *before* the condition is evaluated. Instructions must be introduced at the alternate destination to erase the effects when the condition did not behave as expected. Utilization of the VLIW processor is inhibited if the condition tends to behave contrary to expectation, since operations are done and undone. Again, accurate branch-prediction is necessary to achieve high utilization. Treatments of VLIW processors are found in [1] [13]. The specific problem of analyzing and compiling programs for VLIW architectures is treated in [2]

In this study we examine general classes of proposed branch-predictors, and show upper limits on their accuracy with respect to a set of program traces. The relationship between prediction accuracy and machine utilization and speedup is studied. An architect requiring a certain level of utilization will require a corresponding level of prediction accuracy, and may need to devise a new class of branch-predictor to achieve this.

2 A Simplified Model of Pipeline Utilization

Consider a simple linear pipelined machine model, where instructions are issued and retired in order at a rate of one per clock cycle. The pipeline is D stages long. We define a *block* as a sequence of instructions executed between conditional branches. For an idealized pipelined machine we have the following relationship:

$$\begin{aligned}
\mu &= \text{average blocksize} = \frac{\#\text{ instructions}}{\#\text{ branches}} \\
N &= \text{penalty for a wrong guess} \\
p &= \text{proportion of correct guesses} \\
U &= \text{utilization} = \frac{\#\text{ instructions}}{(\#\text{ instructions}) + N(\#\text{ wrong guesses})} \\
&= \frac{1}{1+\frac{(1-p)N}{\mu}} \qquad (1)
\end{aligned}$$

Expression (1) appears to be independent of the actual pipeline length; dependent upon pipeline length is the quantity N, which represents the size of the pipeline bubble introduced upon an incorrect prediction. This assumes the penalty does not depend upon *which* branch is being predicted.[1] This model ignores the

[1]Strictly speaking, the assumption is that the likelihood of correctly guessing a branch *does not correlate* with the cost of

initial pipeline-fill and final pipeline-empty when the process is started and ended; these are not significant if the length of the instruction stream is long with respect to N and D. Similar effects are present in VLIW architectures, but are not so easily modeled.

Average blocksize values μ are presented in table 1, for a number of real programs described in section 4. Figure 1 shows the distribution of blocksizes across all cases (the last two values are represented as a scatter-plot). Figure 2 shows contours for fixed values of U, as a function of N/μ and p. N/μ is used as a normalized penalty value; the architect may treat N as a variable parameter, but μ is determined by the instruction set, the workload, and the compiler.

The speedup S due to pipelining is expressed as follows:

$$\begin{aligned} S &= \text{speedup} = (\text{pipeline depth})(\text{pipeline utilization}) = DU \\ &= \frac{D}{1+\frac{(1-p)N}{\mu}} \end{aligned} \tag{2}$$

From figure 2 the speedup can be determined by reading the contours in units of D. If we assume $D = N$, and let the pipeline depth go to infinity, we find a strict upper bound on speedup which depends only on the proportion of correct guesses p; this relationship is shown in figure 3. The dotted lines labelled ***situb, ditub, sditub***, etc. correspond to upper bounds on the predictability of our program traces, derived in sections 5 & 6. As we study various kinds of predictors we will see how accurate they can possibly be, and what speedups and pipeline utilizations can be achieved.

The model ignores pipeline stalls due to other effects such as cache misses. We can adjust it to accommodate other causes as follows:

$$\begin{aligned} N_1 &= \text{penalty for a wrong guess} \\ N_2 &= \text{penalty for other causes} \\ p &= \text{proportion of correct guesses} \\ q &= \text{frequency of other causes} \\ U &= \frac{1}{1+\frac{(1-p)N_1+qN_2}{\mu}} \end{aligned} \tag{3}$$

It assumes the penalties are additive when an incorrect guess occurs in conjunction with some other cause. The penalty for an incorrect guess should be insignificant w.r.t. a cache miss, so the assumption should not significantly perturb the results.

Formula (3) rearranges into

$$U = \frac{1}{1+\frac{N_1}{\mu}(1-(p-q\frac{N_2}{N_1})} \tag{4}$$

which amounts to deducting some quantity from the proportion p. If we assume that adjustments to the branch-prediction strategy (and thus p) do not significantly affect N_1, N_2, and q, it appears that we can still treat U and S as being functions of p. Figures 2-3 would remain valid by shifting the p values to deduct the appropriate quantity. For example, if the penalties for a cache miss and an incorrect guess are the same, and the table hit rate is 95%, the proportion of correct guesses must be reduced by 0.05 to give the correct utilization and speedup relationships.

3 Branch Predictors

The CPU can predict branches using information collected as the program executes, or information provided by the compiler, or both. We will call these forms of information *dynamic information, static information,* and *static+dynamic information*, respectively.

Here are three notable examples of branch-prediction using static information:

incorrectly guessing that branch. An example of how they might correlate is that "expensive" branches would tend to behave unpredictably.

Figure 1: Distribution of instruction-block sizes

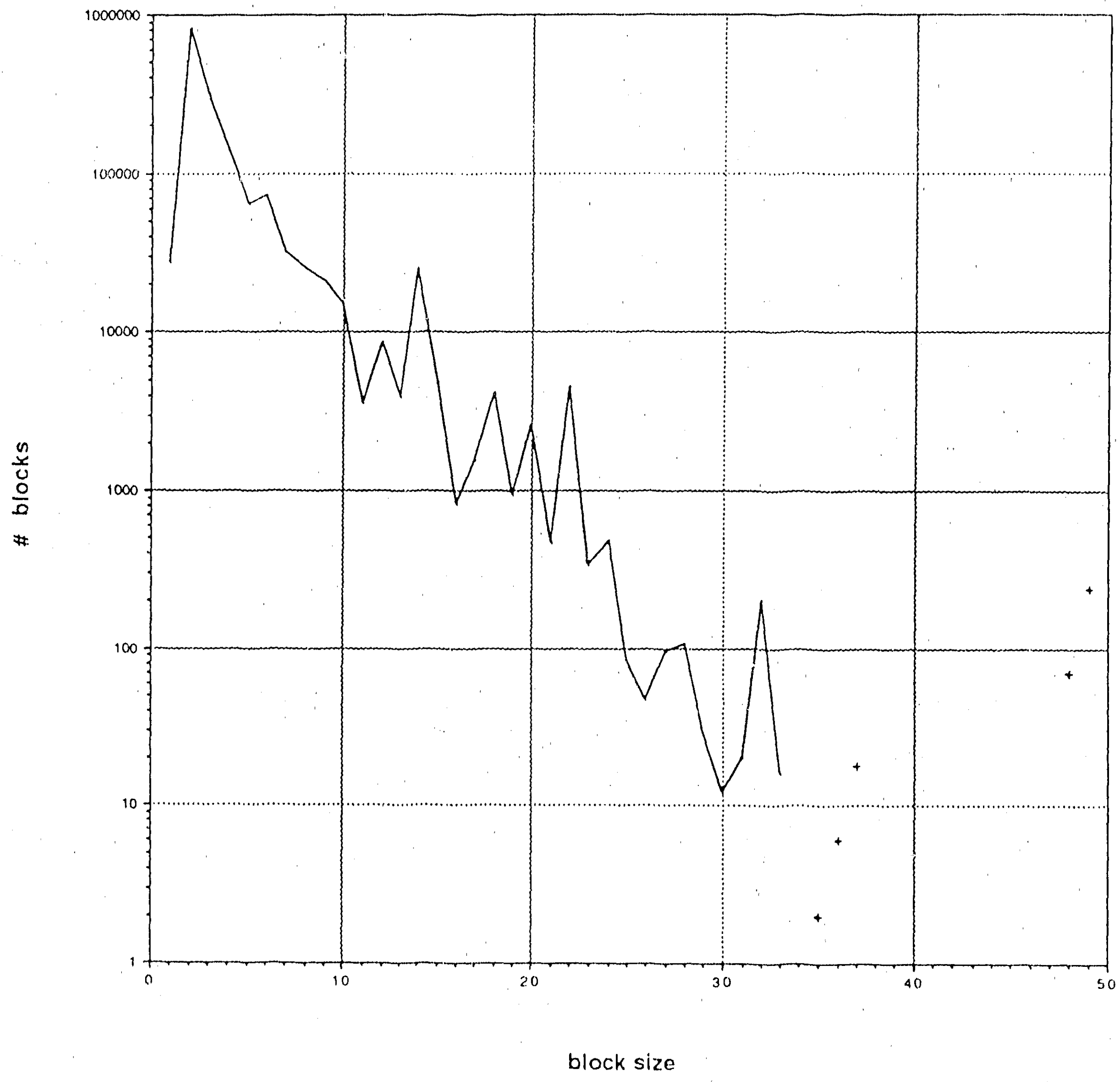


Figure 2: Pipeline utilization contours

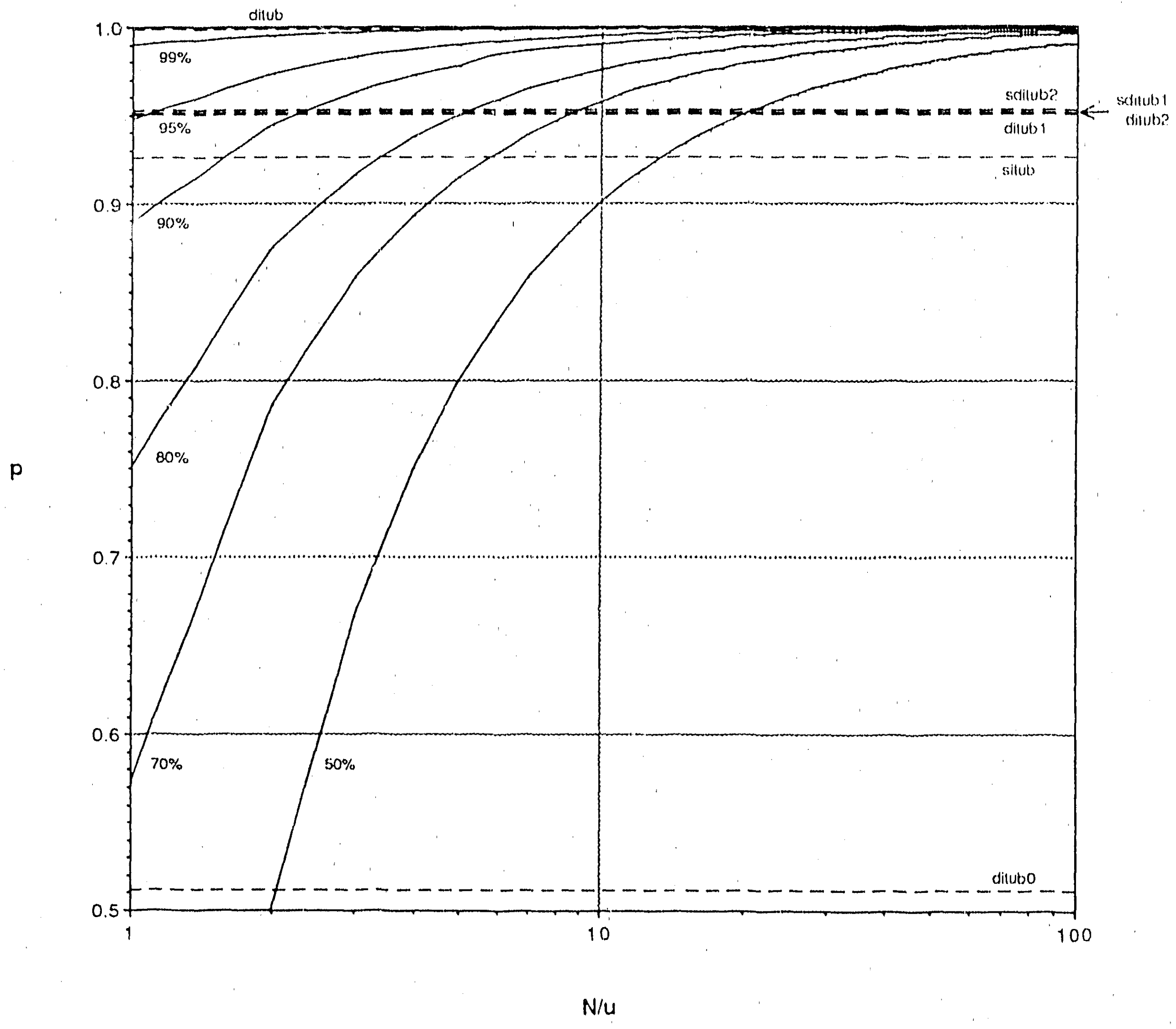
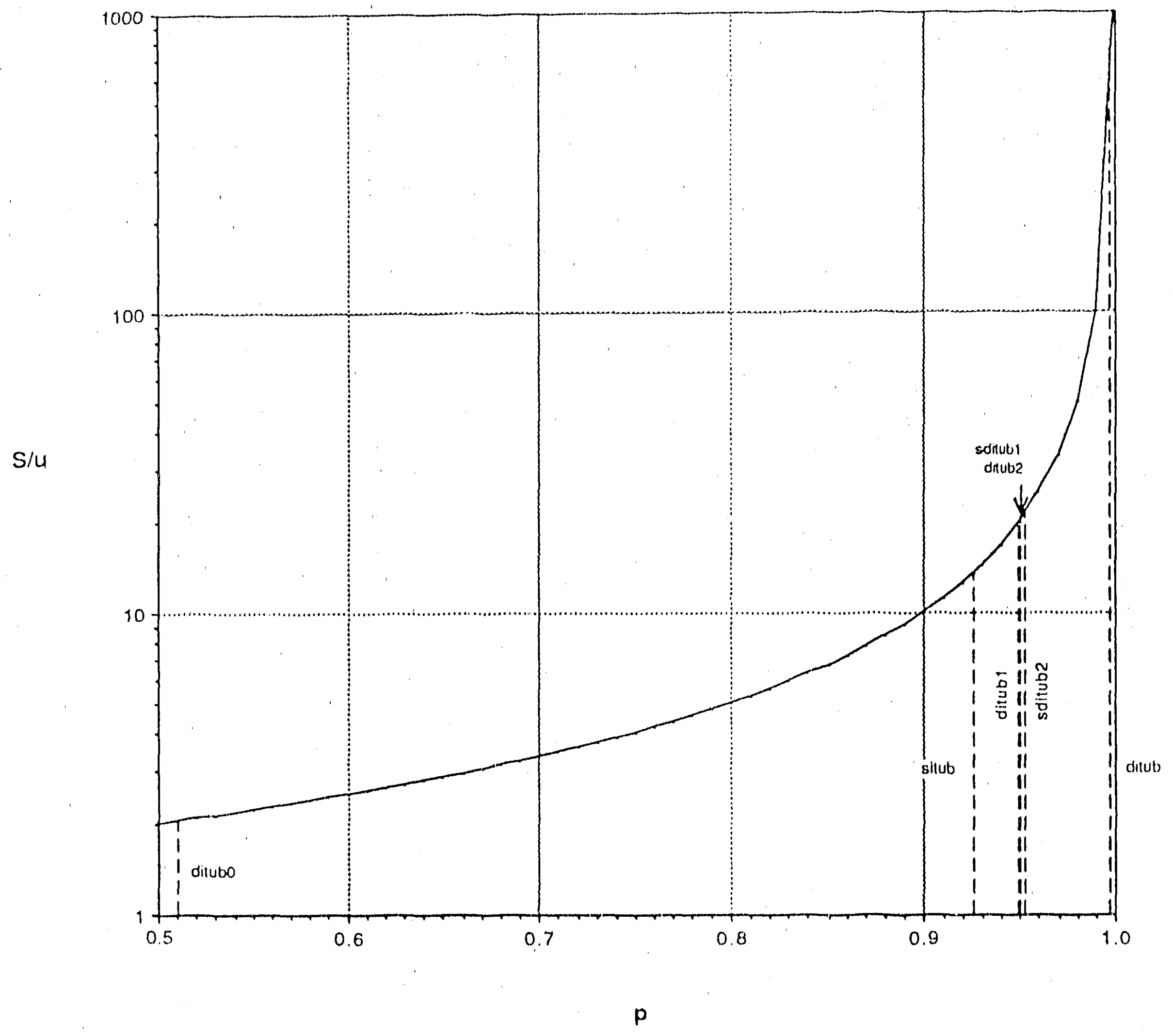


Figure 3: Speedup for an infinitely long pipeline



1. The CPU assumes that a branch-condition always holds true.
2. The CPU assumes that a branch-condition holds true if the destination is a previous address ("backward branch"); this would be accurate for repeatedly-executed loops.
3. Two conditional branch instructions are defined, "branch-probable" and "branch-improbable". The compiler issues one or the other depending upon the context. The CPU guesses the branch-condition always holds true in the first case, and always fails in the second.

The static information associated with a conditional-branch instruction *does not change* as the program executes.

There is no initial dynamic information before a program begins executing. The CPU must guess each branch-condition using information it accumulates as the program executes. In this study we restrict dynamic information to a *per address* basis – the prediction of a conditional-branch instruction depends on the past behavior of that instruction *and no other.* Here are two examples of branch-predictors using dynamic information:

4. The CPU maintains a table of addresses containing conditional-branch instructions. A bit indicates whether or not the condition held on the last activation of the corresponding instruction; on the next activation the CPU will guess that the branch-condition holds if and only if it did previously.
5. Instead of associating a bit with each address, a k-bit counter is associated with each conditional-branch address. Each time the branch-condition holds, the associated counter is incremented, otherwise it is decremented. The CPU guesses that the branch-condition holds if and only if the counter has a 1 in the highest bit-position, indicating that the branch-condition held in the majority of its recent activations.

Most of these methods have been studied in detail [4] [6] [7] [15].

These two forms of information can be coupled. For example, we could apply strategy 5, with initial k-bit counters accumulated in a test run of the program. The same initial information is used each time the program begins executing.

4 The Test Data

We use 7 Vax Unix traces from Mike Shebanow's original study [14]. Each is a frequently used utility program. The larger traces were truncated to 1 million instructions. *Unconditional* branch instructions are not considered here, since they require no real guessing. Unfortunately there were no counts of context-switch or other control-flow instructions available. Likewise, instructions with multiple destinations or computed destinations were ignored. This may skew the results somewhat, but the effective blocksize was small enough that we suspect that few "exotic" branch operations occurred. The test cases are described as follows, with statistics reported in table 1.

ccom1, ccom2: Executions of the Unix portable C compiler.

cpp1, cpp2: Executions of the C preprocessor.

fgrep: A search in a dictionary for words stored in a small text file.

find: A file-search program using the command "find / -name '*.o' -print".

ls: A directory listing using the command "ls -alsg /bin".

Unix composite is the combination of all the other traces.

Table 1 – Characteristics of the Test Data

Program Name	# Instructions Executed	# Active Branch Locations	# Branches Executed	Mean Block Size
ccom1	1,000,000	1384	247,262	4.044
ccom2	1,000,000	511	215,871	4.632
cpp1	249,708	326	75,657	3.300
cpp2	1,000,000	297	327,124	3.057
fgrep	1,000,000	131	394,546	2.535
find	1,000,000	164	220,167	4.542
ls	440,722	402	121,811	3.618
(UNIX composite)	5,690,430	3215	1,602,438	3.551

5 Upper Bounds Assuming Unbounded Information

Now we will construct upper bounds on the predictability of the program traces. We can draw some general bounds by restricting the *type*, but not the *quantity* of information (deferred to section 6) the predictor is allowed to use.

Central to our discussion is the notion of a *branch-history string* [4], which is associated with the address of a branch instruction in the program. For example,

```
program address:        0010100
opcode:                 BRC
branch-history string:  NTTNNNNTN
```

For the given execution, the branch-history string associated with the BRC instruction at address 0010100 indicates that, on the first activation of this instruction, the branch-condition failed to hold ("N" for *not-taken*). On the second activation, the branch-condition held ("T" for *taken*), and so on. Figure 4 shows the distribution on the lengths of the branch-history strings accumulated from our set of program traces. Figure 5 shows how they are distributed in terms of the fraction of T's they contain.

The purpose of a branch-predictor is to try to guess whether the *k*th position of the branch-history string will be an N or a T, using static information or dynamic information accumulated up to the *k*th activation. We will explore combinatorial properties of branch-history strings in order to make statements about branch-predictors in general.

For predictors using static information, for each branch-history string the predictor must make the same guess N or T throughout. The best the predictor can do, then, is to have always predicted N if the string is densest with N's, and T if it is densest with T's. For example, for the branch instruction with associated history-string

```
TTNTTTTNTTT
```

the optimal static predictor would have predicted

```
TTTTTTTTTTT
```

For our instruction traces, then, the optimal branch-predictor based on static information will always guess T for branches with an associated branch-history string densest with T's, and N for branches with an associated branch-history string densest with N's. The results for the optimal assignment are in table 2 under *Optimal Static Predictor*. This value for the UNIX composite appears as line *situb* (for *static information-theoretic*

Figure 4: Distribution of history-string lengths

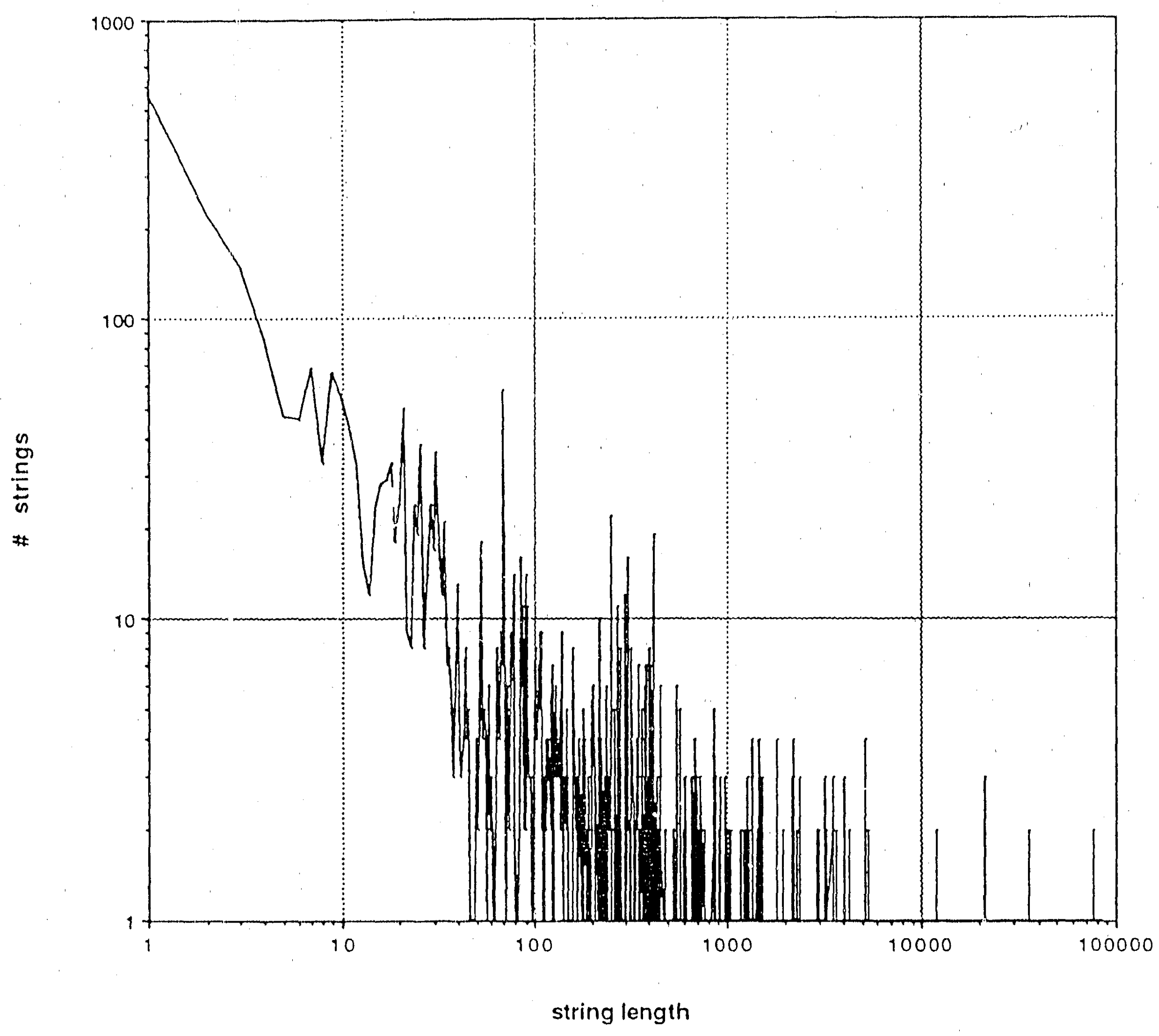
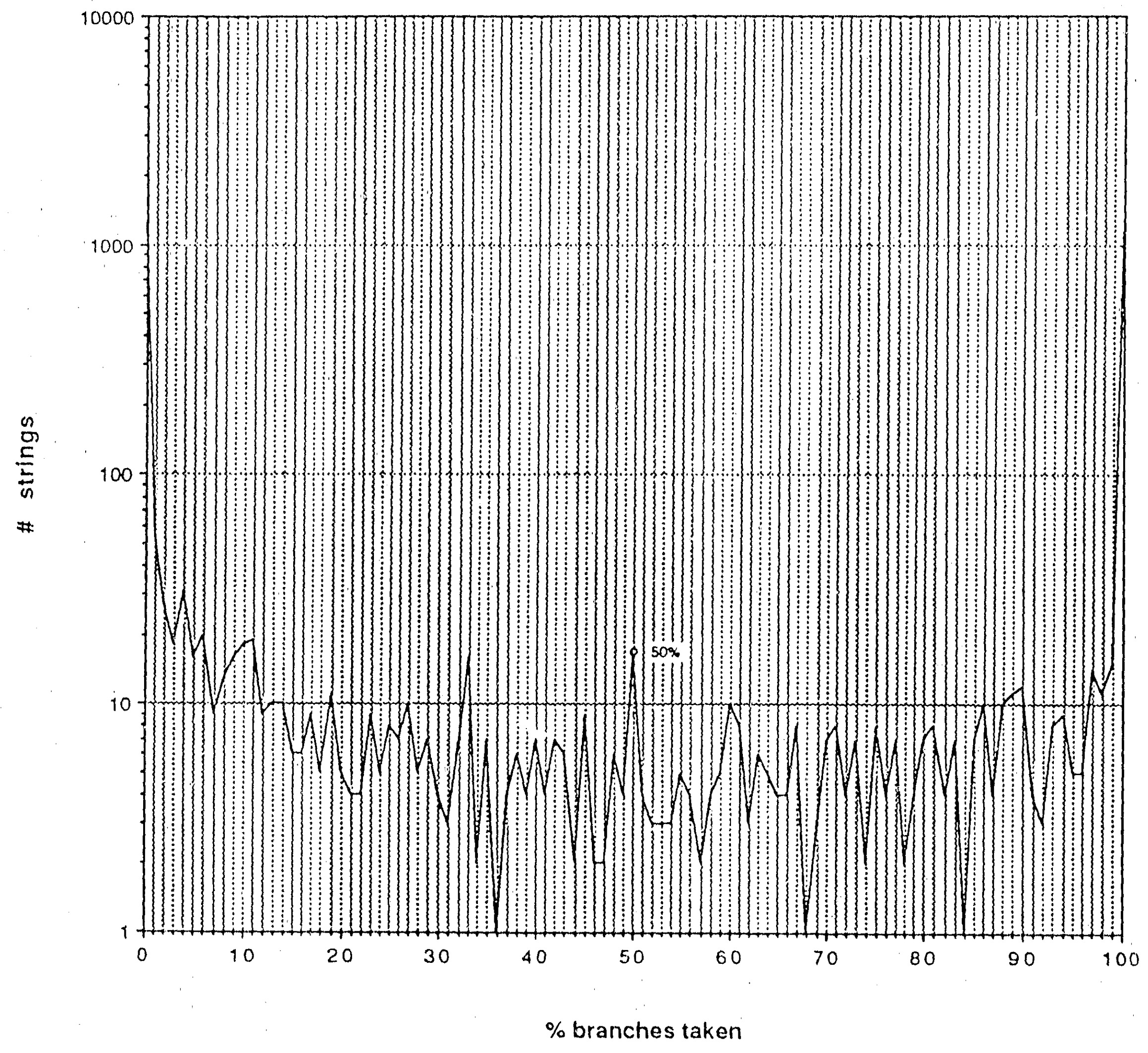


Figure 5: Distribution of taken-branch densities



upper bound) in figures 2 & 3, showing how this upper bound restricts the potential speedup and utilization under static prediction.

Table 2 – Upper Bounds on Prediction Accuracy

Program Name	Optimal Static Predictor	Optimal Dynamic Predictor	
		Own Execution	Group Execution
ccom1	90.75%	99.46%	99.42%
ccom2	90.28%	99.77%	99.74%
cpp1	90.49%	99.58%	99.50%
cpp2	93.49%	99.90%	99.88%
fgrep	93.85%	99.98%	99.97%
find	95.24%	99.94%	99.93%
ls	90.68%	99.74%	99.69%
(UNIX composite)	92.61%	99.79%	99.79%

Suppose two branch locations have these associated branch-history strings:

```
TTNNTTNNT
TTNNTTNNN
```

Under our definitions, a predictor using dynamic information will base its prediction solely on the past behavior of the given branch. The prefixes of these two strings are identical; thus the predictor will make the same guess for the last branch in each string. Any predictor based on dynamic information will guess incorrectly for the last branch in one of these strings. By identifying all the distinct prefixes of the branch-history strings from our traces, we can weigh the number of cases branching each way after having generated a given prefix. No dynamic predictor can do better than to guess whichever direction is observed most frequently. From this "interference" property we can establish an upper bound on the accuracy of dynamic predictors, for these test cases.

Such upper bounds are given in table 2 under *Optimal Dynamic Predictor*. *Own Execution* is where we consider only the interference between the strings from the given trace. *Group Execution* is where we consider interference between the strings of the given trace and the combination of the remaining traces. Ties occur when an equal number of cases branch N and T from a given prefix; ties were broken to evenly divide the incorrect guesses between the test case and the remaining cases. Since few ties occurred this had little effect on the results, roughly 0.15% for the most significant case *(cpp1)*.

The dynamic upper bounds are quite high, decreasing only slightly as we increase the set of test cases and thus the interference between strings. This upper bound for the UNIX composite is shown as *ditub* (for *dynamic information-theoretic upper bound*) in figures 2 & 3. If a predictor could be constructed this accurately, pipeline utilization would be determined more significantly by other effects such as memory stalls or branch-target buffer misses. Perleberg & Smith [9] study this in detail.

We can only speculate where this bound should be for a real system workload; it may possibly be even lower than the static upper bound [12]. Table 3 shows the interference of short prefixes of the history-strings. A significant number of branches are accounted for by short prefixes, indicating that many branch instructions fired few times. There is a strong interference between these short strings. The upper bound grows fairly steadily as the lengths increase. This occurs because the set of strings observed is relatively small compared to the (exponential) number of strings possible for the given length. There is less interference because there are few strings to interfere with. Figure 4 illustrates this.

Table 3 – Dynamic Bounds Using Truncated History-Strings

Maximum String Length	# Branches Accounted For	# Distinct Strings Possible	# Distinct Strings Observed	Dynamic Predictor Upper Bound
1	3215	2	2	51.51%
2	5867	6	5	71.48%
3	8298	14	9	79.08%
4	10,580	30	14	83.29%
5	12,776	62	20	85.93%
6	14,925	126	27	87.83%
7	17,028	254	35	89.18%
8	19,063	510	44	90.24%
9	21,065	1022	62	89.82%
10	23,001	2046	88	90.18%
20	40,607	2,097,150	846	93.36%
30	55,812	2,147,483,646	2626	94.88%
40	69,189	$2^{41} - 2$	5004	95.67%
50	81,880	$2^{51} - 2$	7721	96.23%
60	94,158	$2^{61} - 2$	10,629	96.67%
70	106,041	$2^{71} - 2$	13,680	97.01%
80	116,757	$2^{81} - 2$	16,704	97.27%
90	126,918	$2^{91} - 2$	19,691	97.47%
100	136,455	$2^{101} - 2$	22,656	97.64%
84049	1,602,438	$2^{84050} - 2$	975,465	99.79%

For branch-predictors using static+dynamic information, the only upper bound for unbounded information is exactly 100%. The "static information" would be a table of addresses and associated branch-history strings; the "dynamic information" would be the number of times the branch at that address was executed. To predict the branch-condition on the kth activation, the predictor simply finds the branch-history string associated with the address and returns the kth entry:

```
address      branch-history string
-------      ---------------------

0000001      NNNNNNTNNNNNNTTTTTNNNN
0000010      TTTNTTTNTTTTTTTTNNNNTT
   :                  :
   :                  :
  etc.               etc.
```

Thus there is no "interference", static *or* dynamic, to reduce the upper bound from 100% correct. Such a table could not be realistically constructed; not only is it large, but it would have to be the same across all program runs (this is discussed in more detail in section 7.3). In order to make for more realistic bounds, we now shift from a pure information-theory to an information-based complexity-theory using restricted *quantities* of information.

6 Upper-Bounds Given Bounded Information

The three previous upper bounds depended upon the type of information used by a branch-predictor; for dynamic and static+dynamic information these upper bounds were too high to significantly bound pipeline utilization or speedup. Furthermore, for dynamic and static+dynamic information the optimal predictor would have to encode large tables of program trace information, which should not be practical. By bounding the *quantity* of information used by the predictor, we can reduce these upper bounds to more interesting ranges.

If a branch-predictor associates k bits of information with each conditional-branch instruction, and predicts each branch based only on these k bits, we can model the predictor as a Moore-machine with 2^k states. Each state represents a configuration of the k bits. The output from each state represents the guess made from those k bits. The input represents the actual N or T result of the branch-condition. The state-transitions represent transformations on the k bits as the branch is executed. There is a designated initial-state if static information is not used, so all branch instructions start with the same initial k bits. If static information is used, different branch instructions can start at different initial states. In the presence of static information we refer to the machine as having a *nondeterministic initial state* (in the automata-theoretic sense), since the initial state is chosen to minimize the prediction error. Examples of such Moore-machines are given in the appendices.

Now we can draw information-theoretic upper bounds on the quality of any predictor using k bits, by deriving the optimal Moore-machine predictor with 2^k states. This is done in table 4, for $0, 1$, and 2 bits corresponding to $1, 2$ and 4 states. Unfortunately our (brute-force) optimization procedure [11] was only effective for up to 2 bits. The optimal Moore-machines for the UNIX composite case are shown in appendix 1, which we dub *superpredictors* for outperforming any other predictor of the same size.

Table 4 – Bounds for Finite-State Predictors

Test	Designated Initial State			Nondeterministic Initial State		
Case	0 bits	1 bit	2 bits	0 bits	1 bit	2 bits
ccom1	55.16%	95.27%	95.32%	55.16%	95.54%	95.86%
ccom2	64.52%	95.14%	95.21%	64.52%	95.25%	95.57%
cpp1	61.65%	95.02%	95.10%	61.65%	95.23%	95.41%
cpp2	72.06%	96.11%	96.19%	72.06%	96.16%	96.35%
fgrep	55.98%	93.90%	93.94%	55.98%	93.91%	93.97%
find	55.95%	95.19%	95.22%	55.95%	95.25%	95.43%
ls	64.34%	94.39%	94.43%	64.34%	94.55%	94.83%
(UNIX composite)	51.08%	95.00%	95.05%	51.08%	95.10%	95.28%

For 0 bits of information there is no information, static or dynamic; the guess must be uniformly N or T. The values for the UNIX composite case are shown in figures 2 & 3 as *ditub0, ditub1, ditub2* for the 0, 1, and 2-bit predictor using dynamic information (designated initial state) and *sditub1, sditub2* for the 1 and 2-bit predictor using static+dynamic information (nondeterministic initial state). There was little difference between them.

The accuracy of the finite-state predictors increases as we add states. In fact we could encode the program execution traces directly into a machine of sufficient size (one state for each distinct substring), achieving the information-theoretic upper bound for unbounded information for these traces. The results are shown in table 5; inequalities are used because the traces might be compressible into smaller machines.

Table 5 – Optimal Large Machines Directly Encoding History Traces

	Same Initial State		Nondeterministic Initial State	
Test Case	Max Accuracy	Necessary # Bits	Max Accuracy	Necessary # Bits
ccom1	99.46%	≤ 17	100%	≤ 18
ccom2	99.77%	≤ 17	100%	≤ 18
cpp1	99.58%	≤ 16	100%	≤ 17
cpp2	99.90%	≤ 18	100%	≤ 19
fgrep	99.98%	≤ 19	100%	≤ 19
find	99.94%	≤ 18	100%	≤ 18
ls	99.24%	≤ 17	100%	≤ 17
(UNIX composite)	99.79%	≤ 20	100%	≤ 21

7 How General are the Results?

So far we have shown concrete upper bounds on the predictability of branches in a collection of traces, for various classes of predictors. So long as a given predictor falls into one of these classes, it will predict the traces no more accurately than the upper bound dictates.

There are a number of side results, however, which are worth pursuing. We constructed optimal static and dynamic predictors for the 7 traces; if these traces are good indicators of general program behavior, then the optimized predictors may be accurate for most programs. In particular, the finite-state "superpredictors" constructed in section 6, and the technique of static prediction based on one program run, seem quite practical.

7.1 Superpredictor Sensitivity Analysis

In section 6 we provided upper bounds on dynamic predictors using 2 bits of information. This was done by deriving optimal predictors for the given trace. If we were to use one of these superpredictors in a real machine, it would have to demonstrate high prediction accuracy beyond the one design trace. To study this, we will analyze the *sensitivity* of the predictor to the case it was designed for, by comparing its accuracy across all cases. Interestingly enough, the test cases *ccom1, ccom2, cpp1, cpp2, fgrep, ls,* and the UNIX composite all designed the same 4-state superpredictor; *find* generated another. Table 6 compares the two superpredictors across all test cases.

Note that the difference between them was at most 0.12%. The superpredictor derived from *find* was not significantly better than the composite superpredictor, which was uniformly better for all the other cases.

Table 6 – Sensitivity of 4-Superpredictor Construction

Test Case	Design Case: ccom1, ccom2, cpp1, cpp2, fgrep, ls, (UNIX composite)	Design Case: find
ccom1	95.32%	95.30%
ccom2	95.21%	95.12%
cpp1	95.10%	94.98%
cpp2	96.19%	96.11%
fgrep	93.94%	93.89%
find	95.21%	95.22%
ls	94.43%	94.35%
(UNIX composite)	95.05%	95.00%

7.2 Comparison with Other Predictors

Three 4-state predictors were presented in Lee & Smith [4], which we show in appendix II. Each exhibits an interesting symmetry, and is designed to capture an intuitively plausible form of branch behavior. Table 7 compares them against the composite 4-superpredictor. Since the initial states were not specified in the reference, for each test case we chose the one initial state that minimized the error for the trace.

Note that the 4-superpredictor was strictly superior to the other predictors by a significant amount. Furthermore it was more stable, in that the range of performance was narrower. The behavior of the other predictors is fairly consistent with the results for the workloads used by Lee & Smith; unfortunately their traces were not available for our study.

Table 7 – Comparison of 4-State Predictors

Test Case	4-Superpredictor	S-1 Proposal	Majority	2-Branch History
ccom1	95.31%	93.11%	93.52%	93.33%
ccom2	95.21%	92.79%	93.21%	92.82%
cpp1	95.10%	92.59%	93.04%	92.82%
cpp2	96.19%	94.14%	94.54%	94.39%
fgrep	93.94%	89.82%	90.33%	91.32%
find	95.21%	92.17%	92.64%	93.18%
ls	94.43%	91.16%	91.76%	91.93%
(UNIX composite)	95.05%	92.17%	92.62%	92.85%

The gap between the dynamic information-theoretic upper bound for 2 bits and for unbounded information (=20 bits) was significant. Unfortunately we could not generate superpredictors for 3 or more bits to see how quickly they approach the upper bound. Other methods, however, may be able to utilize more bits to achieve higher accuracy. There are two proposed ways of constructing predictors to use any number of bits; we will see how these work w.r.t. our bounds.

One obvious approach [4] [15] is to simply count the frequency of taken branches vs. not-taken branches, and guess with whichever is higher. In table 8 we do this one step better (for the UNIX composite case), which is to guess whichever direction is more frequent upon observing that density. The counter is restricted to stay at fixed maximum or minimum values instead of overflowing or underflowing. Note that the performance is generally *worse* for more bits; this is consistent with the previous studies. The likely explanation is that the behavior at a given activation is a good indicator of the behavior at the next activation, and this pattern is obscured by the behavior of earlier activations.

Lee & Smith used a method of identifying all branch-history substrings of length k, and assigning the most frequently encountered next branch as the guess for each. This constructs a predictor that encodes the results of the last k activations, and branches with the most frequently encountered next result. Upper-bounds on this approach are presented in table 8, for up to 16 bits. Note that this upper bound is less than the 4-superpredictor performance all the way up through 8 bits. The 16-bit performance is rather low considering the fact that a 20-bit superpredictor can achieve the dynamic information-theoretic upper bound of 99.79% for this trace.

Table 8 – Accuracy Upper Bounds for Two Families of Predictors

# bits	Counter Method	History Substring Method
1	94.99%	95.00%
2	92.73%	95.00%
3	90.80%	95.00%
4	89.97%	95.00%
5	89.58%	95.00%
6	89.28%	95.00%
7	89.15%	95.00%
8	89.06%	95.00%
9	88.99%	95.85%
10	88.93%	95.85%
11	88.88%	95.85%
12	88.85%	95.85%
13	88.85%	95.85%
14	88.85%	95.86%
15	88.85%	95.87%
16	88.85%	95.94%

Extensions of the superpredictors and the Lee/Smith predictors may be complicated, and hence require significant amounts of logic to implement. For 16 or more bits the cost in gates and gate-delays could be prohibitive. Other *ad hoc* prediction schemes might be easily designed to use large numbers of bits, but the structure must be simple enough to allow a compact implementation.

7.3 Consistency Between 2 Program Runs

Static information does not change during the execution of a program, or across multiple executions of the same program. The usefulness of static information or static information coupled with dynamic information depends on some uniform behavior between program runs. Since we have traces for two runs each of the C compiler and the C preprocessor, we can look for evidence of this uniformity.

For example, our optimal static predictor in section 5 assigned a guess T to each conditional branch with an associated branch-history string densest with T's, or N otherwise. Such a prediction scheme might be useful in practice, using a test run of the program to determine the static prediction. Such a technique is used in trace-scheduling compilers for VLIW architectures [2]. For this technique to work, between multiple runs of the same program the branch-history strings associated with the location of a conditional-branch instruction should be consistently denser with T's or N's. In table 9 we find this pattern holds for our test cases; the few branch instructions reversing the density relationship happened to perform few branches.

In table 10 we extend this to 2 bits of static+dynamic information: not only is a predictor designed from the trace, but initial static information associates an initial state with each branch address. For two runs of the same program we use the same predictor (note that our optimization procedure independently derived the same predictor for each), as well as the same initial state for the same branch address each time. To test this, then, we derive the predictor and associate the initial states using the design case, and evaluate it on the test case. The performance for the test cases was always quite good. Note, though, that if a branch instruction was never activated in the design case, we use the test case to select the optimal initial state, so these results are actually upper bounds.

In table 11 we go back to studying unbounded dynamic information. In section 5 we studied the interference between history-string prefixes, to see how predicting for the benefit of one branch would hurt another. The upper bound was reduced slightly if we measured the interference of one case with all the remaining cases, rather than just itself. In table 11 we study the interference between multiple runs of *cpp* and *ccom*, to see if there was enough self-consistency between the two runs that no additional dynamic interference occurred. Comparing against columns 3 and 4 of table 2 shows that there was significant additional interference. In some cases the test case interfered more with its other runs than it did with the UNIX composite case.

Table 9 – (Static) Consistency Between 2 Runs of the Same Program

# Active Branch Instructions	ccom1: 1384 ccom2: 511 cpp1: 326 cpp2: 297
# of Instructions Reversing Behavior	ccom1+ccom2: 18 cpp1+cpp2: 2
# Branches Performed	ccom1+ccom2: 463,133 cpp1+cpp2: 402,781
# Branches Lost	ccom1+ccom2: 205 cpp1+cpp2: 276

Table 10 – Consistency in Initial State Selection for 4-Superpredictor

Design Set	Test Set		
	ccom1	ccom2	ccom1 + ccom2
ccom1	95.86%	95.50%	95.69%
ccom2	95.47%	95.57%	95.51%
ccom1 + ccom2	95.85%	95.55%	95.71%
	cpp1	cpp2	cpp1 + cpp2
cpp1	95.41%	96.31%	96.14%
cpp2	95.34%	96.35%	96.16%
cpp1 + cpp2	95.39%	96.35%	96.17%

Table 11 – (Dynamic) Consistency Between 2 Runs of the Same Program

Design Set	Test Set	# Branches Performed	% Correct Upper Bound Individual	Composite
ccom1 + ccom2	ccom1	247,262	99.38%	99.36%
"	ccom2	215,871	99.75%	99.72%
"	ccom1 + ccom2	463,133	99.53%	99.53%
cpp1 + cpp2	cpp1	75,657	99.53%	99.48%
"	cpp2	327,124	99.90%	99.89%
"	cpp1 + cpp2	402,781	99.81%	99.81%

8 Conclusions & Directions for Further Work

For our set of UNIX traces, the bounds on the accuracy of the ***best possible*** branch-predictor using static information or $\leq$ 2 bits of dynamic or static+dynamic information are enough to limit pipeline speedup and utilization by a significant degree. For example, in a machine with a pipeline depth (and associated misprediction penalty) of 4, the utilization will be no better than 92% under static prediction, and 95% under dynamic or static+dynamic prediction with 2 bits. The speedup will be no better than 3.7$\times$ under static prediction and 3.8$\times$ under dynamic or static+dynamic prediction with 2 bits. For a machine with a pipeline depth (and associated misprediction penalty) of 8, the utilization will be no better than 86% under static prediction, and 90% under dynamic or static+dynamic prediction with 2 bits. The speedup will be no better than 6.9$\times$ under static prediction and 7.2$\times$ under dynamic or static+dynamic prediction with 2 bits. To achieve higher degrees of speedup and utilization for the same pipeline depth and workload, the architect will need to design a branch-predictor with more bits of dynamic or static+dynamic information, or devise a new class of branch-prediction strategy.

The 4-superpredictor, generated by an optimization procedure, appears to be superior to any proposed method of branch-prediction using up to 8 bits of information for each branch instruction. The results of such a study are only as strong as the data used – they look almost too good – and may be biased by the nature of the test cases. For example, the 2-superpredictor outperformed the miscellaneous predictors shown in Appendix II; this contradicts earlier findings [4] [15], suggesting different control-flow patterns between the test cases. The observed consistency between two program runs in 7.3 is based on too few cases to be convincing. All the results are based on VAX UNIX utilities; they should be extended to numeric Fortran codes, system programs, and Cobol business codes (as is done in [4]) if "superpredictors" are to be shown practical. Very likely the structure of the 4-superpredictor will change using such design sets, though by definition it will have to perform at least as well as any other 4-state predictor. With more data the information-theoretic upper bound on dynamic predictors may also be reduced enough to indicate that small superpredictors are near-optimal in practice. We consider our methods more interesting than our results at this stage; an architect needing an improved branch-predictor can use the information-theoretic bounds as reference in deciding how much improvement is possible.

The problem of finding a superpredictor [10] appears to be intractable since it is similar to an $\mathcal{NP}$-complete problem for finite-state transducers [17]. Brute force case-by-case analysis was used to find the optimal predictor [11], and was effective for only 1 to 4 states. There may have been a large amount of redundancy in the enumeration of cases. A more efficient enumeration scheme might provide results for 8 or 16 state predictors, though a polynomial-time optimization algorithm should allow solutions to arbitrary sizes. In the meantime, more mediocre prediction schemes might outperform the 4-superpredictor simply by designing predictors on larger numbers of states.

The number of gates required to implement a superpredictor on 16 or more bits would be prohibitive if it requires a complex boolean mapping. Again a more mediocre prediction scheme may be able to perform well with less logic. It might be more realistic to use a theory of circuit-complexity rather than information-complexity with which to classify and optimize the predictors.

Some statistical methods might be used to study the information-theoretic upper bound for unbounded dynamic information. The bound drawn had depended on there being a small but significant number of branch-history strings of long length. We might assume these strings were drawn randomly from a distribution; this underlying distribution would determine the actual information-theoretic upper bound. The set of strings used was small enough that the underlying distribution could be almost anything, with any resulting bound between 50-100%. Textbook distributions, such as Gaussian or Poisson, are unlikely to be realistic, judging from the distribution in figure 5. An interesting hypothesis is that strings of higher Kolmogorov-complexity [5] occur with lower frequency, since the Kolmogorov complexity measures the computational "difficulty" of generating the string. The range of Kolmogorov-complexities increases among longer strings – this implies that sparse sets of long branch-history strings would tend to be observed. Further, the cumulative Kolmogorov-complexity across strings with a given ratio of 0's to 1's should show an irregular pattern. Sparsity of long strings is illustrated in table 3 and figure 4; an irregular distribution of 0/1 densities is demonstrated in figure 5. The resulting upper bounds on prediction could be impossible to determine.

The model of pipeline utilization made some gross assumptions regarding the costs of operations and the penalty of a bad prediction. The model might be extended, and considered in a revised optimization procedure; alternately the sensitivity of the model to these assumptions could be studied empirically. The model of finite-state predictors assumes that a predictor is associated with a branch operation for the duration of the execution; in reality the state may be stored in a cache which is periodically erased, and prediction is restarted further along the branch-history string. Our model would assume the predictor states are stored and reloaded whenever cache entries are displaced; in fact examples can be constructed where periodically reinitializing a Moore-machine improves the prediction [12]. A Markov-chain may be a more appropriate predictor model under these conditions. The penalty of a prediction cache miss can be considered in our extended model in section 2.

Our definition of predictors based on dynamic information was quite narrow – prediction could only depend on the prefix of the branch-history string. More exotic methods are possible, such as cross-correlating the behavior of different conditional branches in the program. Such methods are not necessarily as limited as the ones treated here; they may be practical for VLIW systems, where the program can be modified to analyze and predict its own behavior. We did not pursue these possibilities because a) we have found no proposed methods falling outside our classification scheme, and b) our analytic approaches did not allow us to make general statements about other classes of predictors.

Acknowledgments

Our interest in the problem of branch-prediction arose from our work on the HPS architecture [8] with Wen-Mei Hwu and Yale Patt. Umesh Vazirani, Chris Perleberg, and Corina Lee provided useful feedback.

References

[1] R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.P. Papworth, P.K. Rodman, "A VLIW architecture for a trace scheduling compiler," *IEEE Trans. on Computers,* vol. 37, no. 8, pp. 967-979, Aug. 1988.

[2] J.R. Ellis, *Bulldog: a Compiler for VLIW Architectures.* MIT Press, Cambridge MA, 1986.

[3] R.W. Holgate, R.N. Ibbett, "An analysis of instruction-fetching strategies in pipelined computers," *IEEE Trans. on Computers,* vol. 29, no. 4, pp. 325-329, April 1980.

[4] J.K. Lee, A.J. Smith, "Branch prediction strategies and branch target buffer designs," *IEEE Computer,* vol. 17, no. 1. pp. 6-22, Jan. 1984.

[5] M. Li, P.M.B. Vitanyi, "Two decades of applied Kolmogorov complexity," *Proc. IEEE Structure in Complexity (3rd annual conference),* June 1988, pp. 80-101.

[6] D.J. Lilja, "Reducing the branch penalty in pipelined processors," *IEEE Computer,* vol. 21, no. 7, pp. 47-55, July 1988.

[7] S. McFarling, J. Hennessy, "Reducing the cost of branches," *Proc. 13th ACM/IEEE International Symposium on Computer Architecture,* June 1986, pp. 396-403.

[8] Y.N. Patt, S.W. Melvin, W. Hwu, W., M.C. Shebanow, "Critical issues regarding HPS, a high-performance microarchitecture," *Proc. 18th Microprogramming Workshop,* Dec. 1985, pp. 109-116.

[9] C.H. Perleberg, A.J. Smith, "Branch target buffer design and optimization," Univ. of California, Berkeley, Tech. Rep. UCB/CSD 89/552, Dec. 1989.

[10] C.G. Ponder, "String prediction by a small machine," in *Studies in Branch-Prediction* (this report).

[11] C.G. Ponder, "Solving Moore-machine prediction by brute force," in *Studies in Branch-Prediction* (this report).

[12] C.G. Ponder, "Questions fundamental to the theory of branch-prediction," in *Studies in Branch-Prediction* (this report).

[13] B.R. Rau, "CYDRA^{TM} 5 directed dataflow architecture," *Proc. 1988 IEEE Spring Compcon,* Feb. 1988, pp. 106-113.

[14] M.C. Shebanow, Y.N. Patt, "Autocorrelation, a new method of branch prediction," Submitted to *IEEE Transactions on Computers* (Nov. 1987).

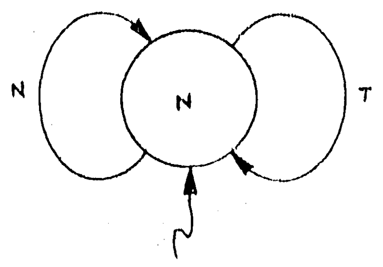
[15] J.E. Smith, "A study of branch prediction strategies," *Proc. 8th IEEE Symposium on Computer Architecture,* May 1981, pp. 135-148.

[16] D.R. Stiles, H.L. McFarland, "Pipeline control for a single cycle VLSI implementation of a complex instruction set computer," *Proc. 1989 IEEE Spring Compcon,* Feb. 1989, pp. 504-508.

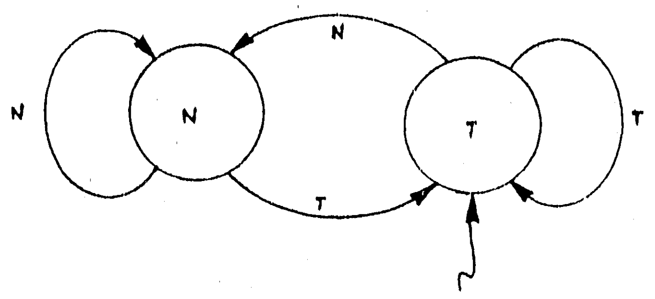
[17] U.V. Vazirani, V.V. Vazirani, "A natural encoding scheme proved probabilistic polynomial complete," *Theoretical Computer Science,* vol. 24, pp. 291-300, 1983.

Appendix I – Finite-State Superpredictors

Designated Initial State

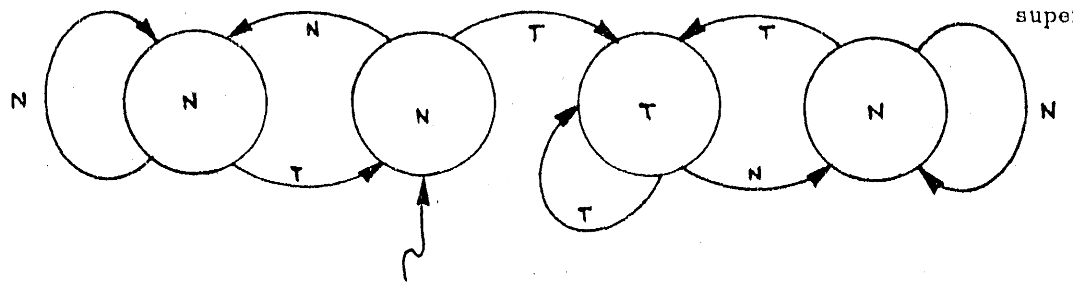


Accuracy: 51.08%
Remarks: Always predict "not taken."

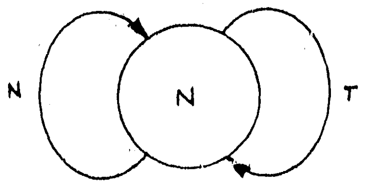


Accuracy: 95.00%
Remarks: Always predict previous input.

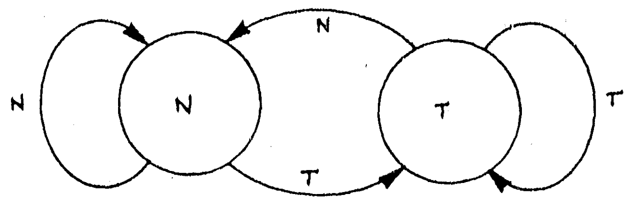
Accuracy: 95.05%
Remarks: Kludge on top of 2-state superpredictor.



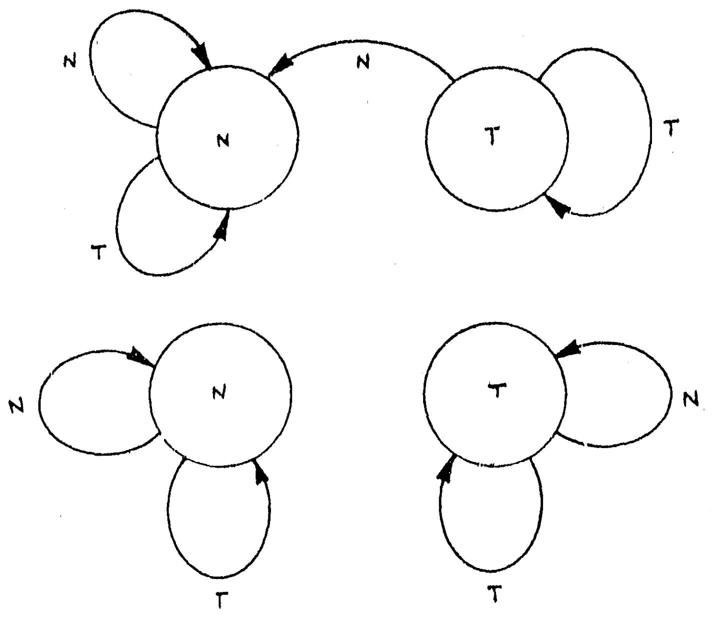
Nondeterministic Initial State



```
Accuracy:  51.08%
Remarks:   Always predict
            "not taken."
```

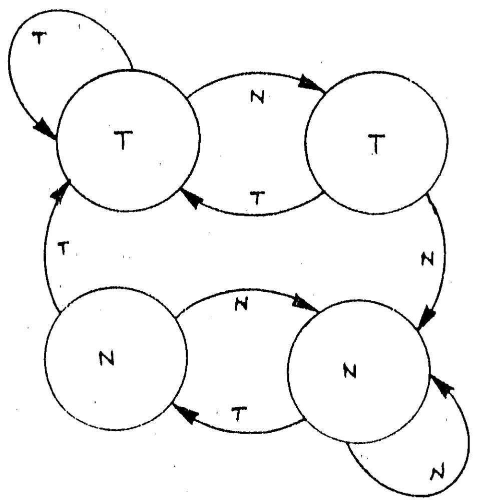


```
Accuracy:  95.10%
Remarks:   Always predict
            previous input.
```

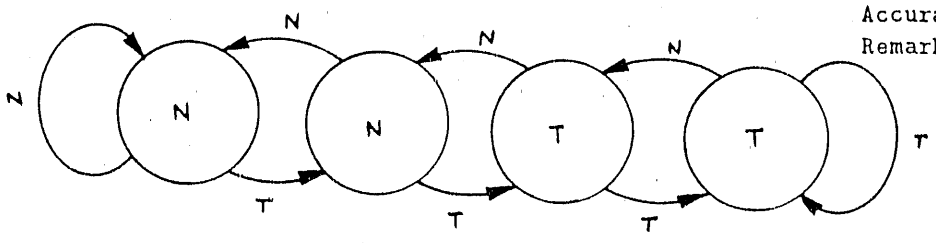


```
Accuracy:  95.28%
Remarks:   3 disconnected
            components treat
            3 kinds of
            behavior.
```

Appendix II – Miscellaneous Predictors

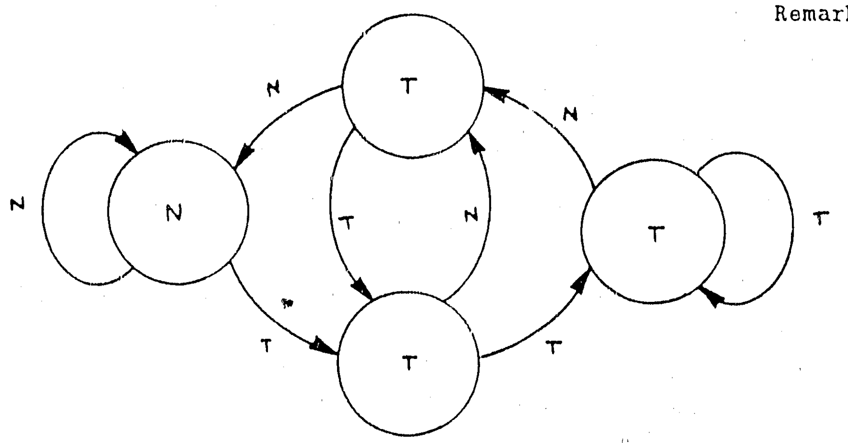


Description:	S-1 proposal
Accuracy:	92.16%
Remarks:	Requires 2 "takens" or "not takens" in a row to change guess.



Description:	Majority
Accuracy:	92.62%
Remarks:	Predicts more frequent result so far.

Description:	2-Branch History
Accuracy:	92.85%
Remarks:	State encodes last 2 branch results.



Questions Fundamental to the Theory of Branch-Prediction *

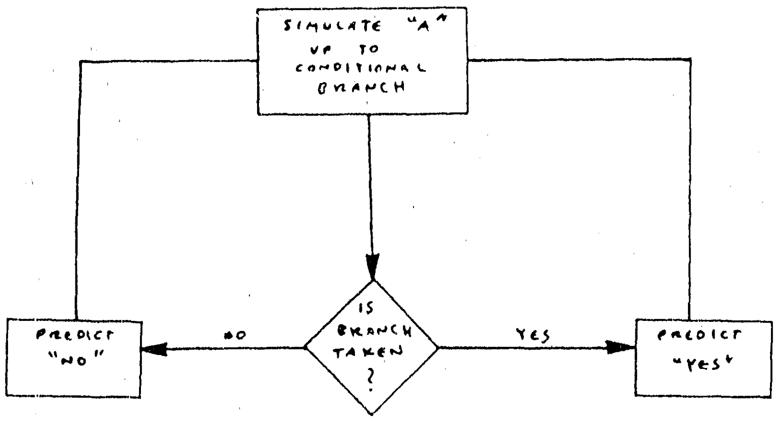
Carl G. Ponder

Computing Research Group, L-419
Lawrence Livermore National Laboratory
P.O. Box 808
Livermore, CA 94550
(415) 423-7034

There are a number of issues to be resolved in designing a good branch-predictor. How good is a given predictor? When is it optimal? How does one predictor perform with respect to another, across some range of cases? How is the performance conditioned by other considerations, such as cache misses? Some of these questions can be posed in a general way, and answered by constructing (degenerate) examples.

1. Can branches be predicted 100% correct?

A branch-predictor can be constructed to be 100% correct for a given program A:



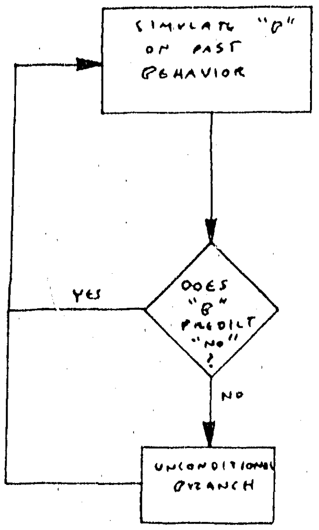
By simulating the program A, the predictor determines the direction of the next branch correctly each time. No program is bad for all predictors; furthermore, for any predictor there is a program for which another predictor is at least as good.

2. Can a branch predictor be 100% correct across all programs?

On the flipside, a program can be constructed to render a given branch-predictor B 100% wrong. By simulating the predictor B on its own execution, the program simply does the opposite of what the predictor

*This material was omitted from [1] for the sake of compactness, but is necessary for a complete treatment of the subject.

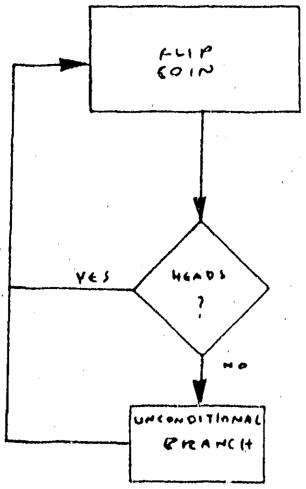
will guess:



No branch-predictor is, or can be proven, infallible.

3. How good can an oblivious predictor be?

The above cases depend upon the predictor encoding the program, or the program encoding the predictor. If the predictor is limited to monitoring the branch behavior of the program, we can still find limits on its quality. Let P perform independent flips of a fair coin, and branch accordingly:



Any predictor will have an expected correctness rate of 50%. Using a pseudo-random number generator (along the lines of Yao [2]) we can in fact construct a family of programs approaching this behavior, so long as the predictor does not encode P.

4. Is dynamic information better than static information?

So far we have placed few restrictions on the nature of the predictors. In reality the prediction for a given branch instruction may be based only on its own behavior. We distinguish two types of prediction information: *static information*, which is associated with a branch instruction *before* the program executes, and does not change; and *dynamic information*, which is initially the same across all branch instructions, but changes with the behavior of the instruction. The quality of a static predictor is sensitive to the interference between elements of a given branch-history string. The quality of a dynamic predictor is sensitive to the interference between strings.

The following branch-history string will be predicted 50% correct by a static predictor:

```
NTNTNTNTNT
```

A dynamic predictor need only reverse its prediction each time to predict 100% correct. The following set of branch-history strings will be predicted 50% by a dynamic predictor, since the second element is essentially independent of the first element:

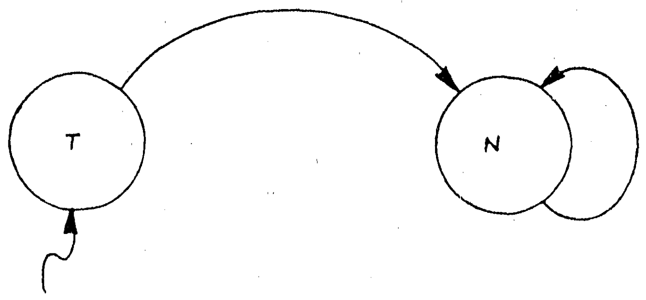
```
NN
NT
TN
TT
```

A static predictor will be 75% correct by predicting "NN" for the first two cases, and "TT" for the second two. How static and dynamic predictors compare in practice is another matter.

5. What is the effect of re-initializing a predictor?

This question is concerned with the hardware implementation of a predictor. Prediction is usually done by maintaining a table of information associated with each conditional branch instruction in the program; if this table is stored as a small cache, entries are periodically displaced. If a displaced entry is not saved and restored, prediction for the associated branch instruction is re-initialized when it next executes.

Intuitively this re-initialization should degrade the prediction, since information is lost. We can construct an example where this is *not* the case: let $\{TN^k\}^*$ be the string we want to predict. No finite-state predictor will be 100% correct unless it has $> k$ states; however, the following two-state predictor will be 100% correct if it is re-initialized every $(k+1)$ steps:



Whether an erasing-cache scheme will outperform a replacing-cache scheme in practice is another question.

References

[1] C.G. Ponder, M.C. Shebanow, "An information-theoretic look at branch-prediction," in *Studies in Branch-Prediction* (this report).

[2] A.C. Yao, "Theory and applications of trapdoor functions," *Proc. 23rd IEEE Symposium on Foundations of Computer Science,* Nov. 1982, pp. 80-91.

String Prediction by a Small Machine

Carl G. Ponder

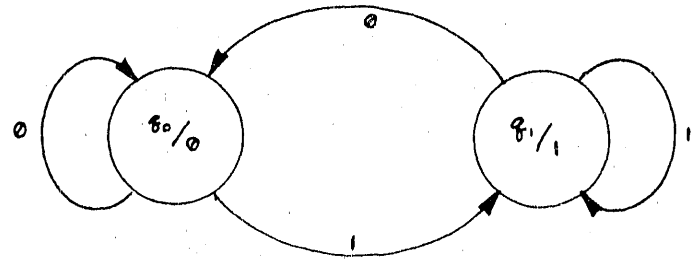
Computing Research Group, L-419
Lawrence Livermore National Laboratory
P.O. Box 808
Livermore, CA 94550
(415) 423-7034

There are circumstances in online computation where we wish to predict the value of an input before it is actually known. An example is in branch-prediction [2]: the microcontroller guesses whether a branch condition will hold before the conditional expression is evaluated. It then prefetches instructions from the assumed branch destination. The instruction-stream can be processed faster by prefetching than by repeatedly waiting for conditions to be evaluated. The more correct guesses, the lower the penalty due to incorrect prefetching.

Here we formalize the prediction problem as a prediction of the *i*th element of a string, based on the previous $(i-1)$ elements. The predictor is only allowed to maintain a fixed amount of information. The predictor is modeled as a *Moore-machine* [1], where the states of the machine encode the information states of the predictor. We can then determine how good the "best" predictor of this type can be.

Definition: Let x and y be two strings over alphabet Σ, of length m and n respectively. We define error(x,y) as the number of positions i, $1 \leq i \leq \min(m,n)$ such that $x[i] \neq y[i]$. For example, error$(0001110, 00001110) = 2$.

Definition: a *rootless Moore-machine M over* Σ [1] is a four-tuple $(Q, \Sigma, \delta, \lambda)$. Σ is the input and output alphabet. Q is the set of states. δ is the state-transition function $\delta : Q \times \Sigma \longrightarrow Q$, determining a new state based on the old state and the input symbol. λ is the output function $\lambda : Q \longrightarrow \Sigma$ determining an output symbol for each state. For example:



Given input string $a_1 a_2 ... a_n$ and initial state q_{i_1}, M computes a mapping $M(q_{i_1}, a_1 a_2 ... a_m) = b_1 b_2 ... b_n b_{m+1}$, where $\delta(q_{i_j}, a_j) = q_{i_{j+1}}$ and $\lambda(q_{i_j}) = b_j$. The machine shown above maps the string 0001110 into 00001110, when started from state q_0.

We can now combine these two definitions, to measure how well M predicts a given string x: let $\Phi(q,x) =$ error$(x, M(q,x))$. Φ is precisely the number of positions where the input x fails to match the output of M, starting from state q. In the above example, $\Phi(q_0, 0001110) = 2$.

[1]Note that this definition differs from standard Moore-machines [1]; under the standard definition the input and output alphabets may be different, and there is a preassigned start state q_0 for every computation. This variation better fits our needs.

Here are four forms of the prediction problem. We may in some cases want to predict collections of strings, with or without the restriction that we must start in the same initial state each time.

`Problem 1:` (predicting a single string). Given k and a string x, what is the smallest $E = \Phi(q, x)$ for any k-state machine M and choice of initial state q?

`Problem 2:` (predicting a set of strings, from the same start state). Given k and a set of strings $\{x_j\}$, what is the smallest $E = \sum_j \Phi(q, x_j)$ for any k-state machine M and choice of initial state q?

`Problem 3:` (predicting a set of strings, from arbitrary start states). Given k and a set of strings $\{x_j\}$, what is the smallest $E = \sum_j \Phi(q_{i_j}, x_j)$ for any k-state machine M and assignment of initial states q_{i_j}?

`Problem 4:` (predicting a set of strings, where designated subsets must start from same initial state). Given k, and a set of strings $\{x_{jl}\}$, what is the smallest $E = \sum_{j,l} \Phi(q_{i_j}, x_{jl})$ for any k-state machine M and assignment of initial states q_{i_j}?

Is there an efficient solution to these problems? An ideal solution would be polynomial in k and linear in the lengths of the strings.

`Remarks:` These problems are clearly in $\mathcal{NP}$, if we ask whether there exists a k-state machine M such that the error $E \leq E'$, for given k and E'. A nondeterministic Turing machine need only generate a k-state machine M (picking an assignment of initial states as necessary), simulate it on the input, and answer "yes" if M predicts the input within the margin of error. A related problem for Mealy-machines [1] has been shown $\mathcal{NP}$-complete [3].

Problem 1 is trivial if k is as large as the input string, since we can encode x directly into M. Problems 2-4 are similarly trivialized if k is as large as the sums of the lengths of the input strings.

There is a variation on these questions which is not of direct interest, but seems natural to ask. For each machine M there is a string s which M will predict 100% correct, and another s' which M will predict 0% correct. But for fixed k, there should be "long" strings which any k-state machine will show only limited success.

`Problem 5:` (the worst possible string for any k-state machine). Given k and n, what is

$$\max_s(\min_{M,q}(\Phi(q, s)))$$

where M ranges over all k-state machines, q ranges over the states of M, and s ranges over all strings of length n? Is there any pattern to these "worst" strings?

Clearly a lower bound is $n/|\Sigma|$, independent of k, since at least $1/|\Sigma|$ of the positions of any string will contain the same symbol. A 1-state machine predicting sequences of the same symbol will be right for that many positions.

References

[1] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, Mass., 1979.

[2] J.K. Lee, A.J. Smith, Branch Prediction Strategies and Branch Target Buffer Designs. *IEEE Computer* 17 (1984) 6-21.

[3] U.V. Vazirani, V.V. Vazirani, A Natural Encoding Scheme Proved Probabilistic Polynomial Complete, *Theoretical Computer Science*, 24 (1983), 291-300.

Solving Moore-Machine Prediction by Brute Force

Carl G. Ponder

Computing Research Group, L-419
Lawrence Livermore National Laboratory
P.O. Box 808
Livermore, CA 94550
(415) 423-7034

Abstract

A *Moore-machine* is a form of finite-state transducer, mapping string x into string x'. Given k and a string x over $\{0,1\}$, the *Moore-machine prediction* problem is as follows: find a k-state Moore-machine such that $x_i = x'_i$ for as many i as possible. A brute force solution enumerates all k-state Moore-machines, simulates each on x, counts the number of positions where each machine produces a mismatch, and identifies the machine minimizing this count. Some theoretical refinements allow us to remove redundant Moore-machines from consideration.

Introduction

Variations of the Moore-machine prediction problem are discussed in [2]. The purpose of this report is to explain how the problem was solved in [3], for readers intending to reproduce or repeat the study on additional data sets. The problem was solved using a brute-force algorithm. Two details of this algorithm are sketched: how the Moore-machines are enumerated, which derives from Graph Theory, and how we eliminate redundant machines, which derives from Automata Theory. Familiarity with Moore-machines and the string prediction problem are assumed. Details of representing, simulating, and choosing the optimal machines are straightforward; the code is somewhat readable if one is interested.

The following table shows the number of distinct k-state machines used in the simulation. The number depends on whether an initial state is designated, or can be chosen to best predict the string (called *nondeterministic* initial state). A gross upper bound of $k^{2k}2^k$ is shown for comparison; there are 2^k ways of assigning 0 or 1 as the output of each state, and there are k possible destinations for each of the $2k$ state-transitions. Many of the possibilities compute the same mapping, or form improperly configured machines; the algorithms in the following sections limit the enumeration to nonredundant and correctly-formed cases.

Counts of distinct Moore-machines

# states	number of distinct machines		upper bound
	designated initial-state	nondeterministic initial-state	
1	1	2	2
2	12	16	64
3	216		5832
4	5428	14279	1048576
5	160675		312500000

For machines with designated initial states, the states are not assigned outputs. This should reduce the machine count considerably. The simulation is simplified by assigning outputs to states *after* completing, as described in the next section.

Machine sizes of $k = 2^n$ are considered in our study [3], where each state corresponds to a configuration of n bits. The number of k-state machines grew fast enough that the problem could not be solved for $2^3 = 8$

states: storing the machine descriptions takes too much space, and simulating them takes too much time. For $k = 4$, generating the machines takes about a day on a lightly-loaded VAX 8600 or MIPS/1000.

Enumerating Machines with Designated Initial State

The following nondeterministic algorithm generates all possible k-state machines, such that a) every state is reachable from the initial state, b) every state has two exiting transitions (0 and 1). No two machine descriptions will be isomorphic.

Three sets of states are maintained: U is the set of *unused* states with no incoming or outgoing transitions. W is the set of *working* states with incoming, but not outgoing, transitions. F is the set of *finished* states with incoming and outgoing transitions. At each iteration $|U| + |W| + |F| = k$.

1. Initially $W = \{1\}$, the initial state, with an implicit incoming transition. $U = \{2, ...k\}$ contains the remaining states.
2. Draw a state S from W; set the transition on 0 to either
   - any one of the $|W|$ working states,
   - any of the $|F|$ final states, or
   - a new state if $|U| > 0$, in which case this new state is moved from U to W.
3. Set the transition on 1 to either
   - any one of the $|W|$ working states,
   - any of the $|F|$ final states, if $|W| > 0$ or $|U| = 0$ (this prevents the machine from being completed with fewer than k states), or
   - a new state if $|U| > 0$, in which case this new state is moved from U to W.
4. Move S into F. Return to step 2 if $|F| < k$.

This will take exactly k iterations.

To actually *program* the nondeterminism, a recursive procedure keeps copies of U, W, and F. Whenever alternatives are presented, recursive calls are made which create appropriately modified copies of U, W, and F. Once a machine description is completed, it is written into an array used in the later simulation.

Note that outputs are never assigned to the states. This saves work in the simulation. Rather than comparing the input and output of each machine, the simulation records the number of times a 1 or 0 is read as input to each state. After the simulation completes, the state is defined to output whichever input was more frequent; this minimizes the prediction error across all 2^k assignments of outputs, without having to enumerate them.

Redundancy still exists: states will be equivalent under many assignments of outputs. The machine will be non-minimal under such assignments, and hence equivalent to a smaller machine. Likewise, some minimal machines may compute identical mappings. An alternative approach is to assign outputs to the states, which increases the number of configurations, and remove redundant machines, which decreases the number of configurations. This approach had to be used in the following section. The number of machines grew by less than a factor of 2^k, as shown in the table, so a significant number of possibilities must have been eliminated. Since the number of configurations is larger with nondeterministic initial states, it is still not clear whether or not this approach would produce more machines with designated initial states.

Enumerating Machines with Nondeterministic Initial States

A harder version of the prediction problem allows each string in a set to be predicted from a different initial state. We must determine both the best machine and the best assignment of initial states to strings. The Moore-machines have a different character: there is no designated initial state, and the states need not all be connected. The "machine" may actually be divided into disjoint connected components; and different strings are predicted by the different parts. The previous enumeration prevented itself from creating machines that were too small or had unreachable states. These are no longer considerations: any state may be a start state, and hence is reachable.

Given a machine with k states, let c be the number of connected components. Let s_i be the size of the ith connected component, nondecreasing in i, $k = \sum_i s_i$. If two machines are isomorphic, then a) they consist of the same number of components, and b) corresponding components are isomorphic. The machines fall into classes according to the arrangement of s; we generate distinct machines as combinations of distinct s_i-sized components. The only problem occurs when two components have equal size: they may be isomorphic to each other, in which case the machine is non-minimal and can be reduced, or the machine may be shown isomorphic to another by reordering the components. The next section shows how to eliminate such cases after they are constructed.

The following nondeterministic algorithm generates all possible n-state connected components: U, W, and F are defined as before, except the states of F need not have incoming transitions.

1. Initially $W = \{1\}$ and $U = \{2, ...n\}$.
2. Draw a state S from W and go to step 3, or from U and go to step 4.
3. Set the transition on 0 to either
   - any one of the $|W|$ working states,
   - any of the $|F|$ final states, or
   - a new state if $|U| > 0$, in which case this new state is moved from U to W.

   Repeat for the transition on 1, and go to step 6.
4. Set one of the transitions to either
   - any one of the $|W|$ working states, or
   - any of the $|F|$ final states, if $|W| > 0$ or $|U| = 0$.

   This prevents a new connected component from forming.
5. Set the remaining transition to either
   - any one of the $|W|$ working states,
   - any of the $|F|$ final states, if $|W| > 0$ or $|U| = 0$, or
   - a new state if $|U| > 0$, in which case this new state is moved from U to W.
6. Move S into F. Return to step 2 if $|F| < n$.

Again, this process takes n iterations.

The simulation processes each string using each state as the initial state. State outputs must be assigned: the balance of 0's and 1's predicted by each state depends upon the initial state assignment of each string, which is not fixed. All 2^k output assignments must be considered. Non-minimal machines are eliminated, since they are implicitly considered with smaller k. Machines are discarded if an equivalent machine has already been produced. Testing minimality and equivalence is explained in the next section.

Identification of Non-Minimal and Equivalent Machines

The Myhill-Nerode theorem [1] provides a way to test the minimality of a machine, or the equivalence of two machines, using the following fact:

> Two states are inequivalent if, for some input string x, the machine produces different output depending upon which is used as the initial state.

Textbooks define the theorem in terms of finite-state machines; the extension to transducers is identical in all ways.

A $k \times k$ table marks pairs of states known to be inequivalent. The minimality test is as follows:

1. If two states have different output, mark them as inequivalent.

2. On a given input, if two states transfer to inequivalent states then mark them inequivalent.

For each pair of inequivalent states, then, we follow each transition *backwards* to find the states they transfer from. This requires only k^2 tests, since each pair of states is considered at most once, and only *after* the two have been shown inequivalent. Once every pair of inequivalent states is processed, and no new ones appear, any pair not marked inequivalent are now equivalent. If two equivalent states exist, the machine is non-minimal.

Testing equivalence between machines is identical. Instead of comparing states of the same machine, we compare the states of one machine with the states of the other. Repeat the above procedure; the machines are equivalent if a correspondence can be found between the states of the two machines, such that none of the pairs of corresponding states are shown inequivalent.

References

[1] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, Mass., 1979.

[2] C.G. Ponder, "String prediction by a small machine," in *Studies in Branch-Prediction* (this report).

[3] C.G. Ponder, M.C. Shebanow, "An information-theoretic look at branch-prediction," in *Studies in Branch-Prediction* (this report).

END

DATE FILMED

11 / 07 / 90