2

CONF-900564--1

UCRL- 101868
PREPRINT

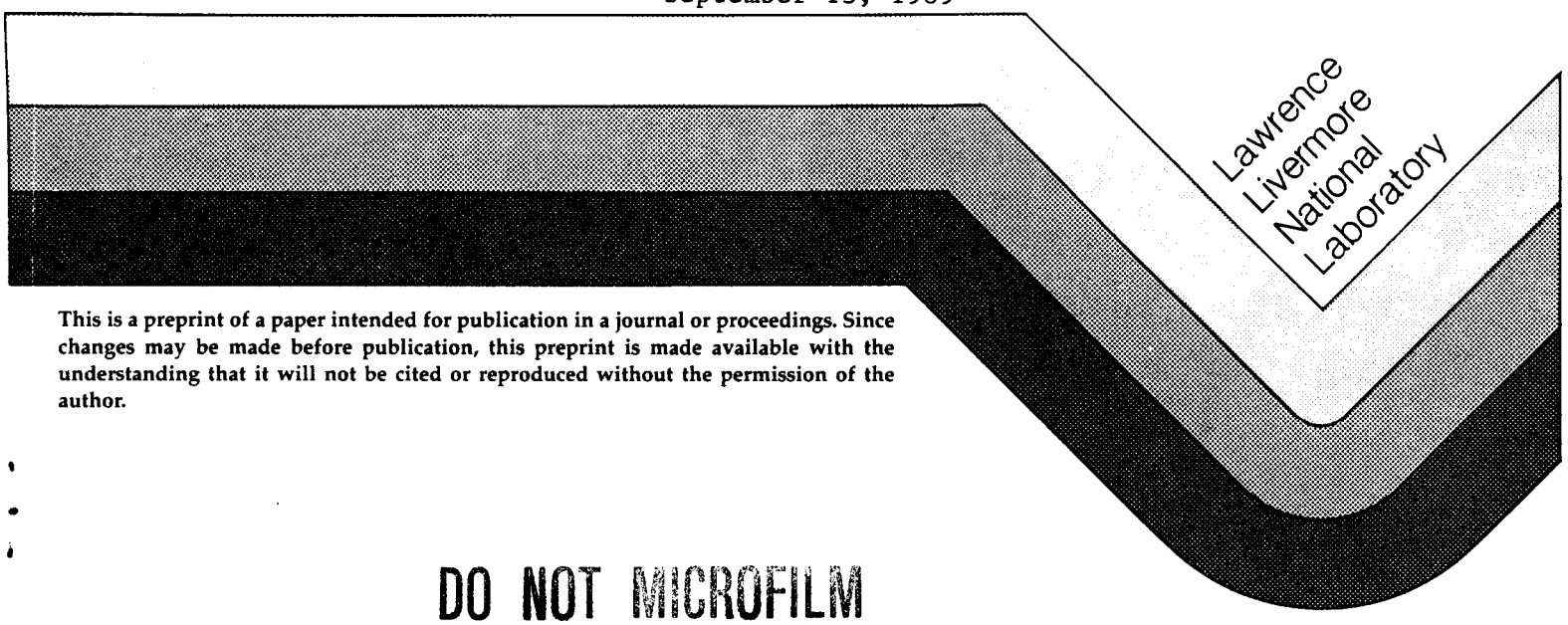


# An Information-Theoretic Look at Branch-Prediction

Carl Ponder
Lawrence Livermore National Laboratory
Michael C. Shebanow
Motorola, Inc.
Austin, TX

This paper was prepared for the ACM Sigmetrics '90 Conference in Boulder, Colorado, May 22-25, 1990

September 13, 1989



---

UCRL--101868

DE90 001170

# An Information-Theoretic Look at Branch-Prediction (Extended Abstract)

Carl G. Ponder

Computing Research Group
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

September 14, 1989

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.

**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.

# 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 is referred to as a *bubble* and 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 ("branch-prediction") are reasonably accurate. Treatments of pipelining and pipelined machines are found in [6] [10]. The specific problem of branch-prediction is treated in [3] [4] [5].

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. The 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] [8]. 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 their upper limits 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:

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

The quantity $N$ represents the size of the bubble introduced in the pipeline upon an incorrect branch prediction. This assumes the penalty does not depend upon *which* branch is being predicted. This model also ignores the 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 $\mu$ 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 $N/\mu$ and $p$. $N/\mu$ is used as a normalized penalty value; the architect may treat $N$ as a variable parameter, but $\mu$ is determined by the instruction set, the workload, and the compiler.

The speedup $S$ due to pipelining is expressed as follows:

$$S = \text{speedup} = (\text{pipeline depth})(\text{pipeline utilization}) = DU$$

$$= \frac{D}{1 + \frac{(1-p)N}{\mu}} \tag{2}$$

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 $p$; this relationship is shown in figure 3.

# 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: 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 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 the 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: 1] 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. 2] 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.

These two forms of information can be coupled. Here is an example of a predictor using static+dynamic information: we apply the strategy 2] for dynamic information, using 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 [9]. 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*, 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 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.

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. 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 $k$th activation, the predictor simply finds the branch-history string associated with the address and returns the $k$th 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. 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 optimization procedure was only effective for up to 2 bits. The optimal Moore-machines for the UNIX composite case are shown in appendix I, which we dub *superpredictors* for outperforming any other predictor of the same size.

| Table 4 – Bounds for Finite-State Predictors | | | | | | |
|---|---|---|---|---|---|---|
| Test Case | Designated Initial State | | | Nondeterministic Initial State | | |
| | 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, 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 | | | | |
|---|---|---|---|---|
| Test Case | Same Initial State | | Nondeterministic Initial State | |
| | Max Accuracy | Necessary # Bits | Max Accuracy | Necessary # Bits |
| ccom1 | 99.46% | $\leq 17$ | 100% | $\leq 18$ |
| ccom2 | 99.77% | $\leq 17$ | 100% | $\leq 18$ |
| cpp1 | 99.58% | $\leq 16$ | 100% | $\leq 17$ |
| cpp2 | 99.90% | $\leq 18$ | 100% | $\leq 19$ |
| fgrep | 99.98% | $\leq 19$ | 100% | $\leq 19$ |
| find | 99.94% | $\leq 18$ | 100% | $\leq 18$ |
| ls | 99.24% | $\leq 17$ | 100% | $\leq 17$ |
| (UNIX composite) | 99.79% | $\leq 20$ | 100% | $\leq 21$ |

# 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%. Also 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 [3], 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 start-states were not specified in the reference, for each test case we chose the one start-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 in 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 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. 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% |

In bad cases the superpredictors and the Lee/Smith predictors require complicated mappings, and hence significant amounts of logic. For 16 or more bits the cost in logic 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 an efficient 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 a start-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 start-state for the same branch address each time. To test this, then, we derive the predictor and associate the start-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 start-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<br>ccom2: 511<br>cpp1: 326<br>cpp2: 297 |
| # of Instructions Reversing Behavior | ccom1+ccom2: 18<br>cpp1+cpp2: 2 |
| # Branches Performed | ccom1+ccom2: 463,133<br>cpp1+cpp2: 402,781 |
| # Branches Lost | ccom1+ccom2: 205<br>cpp1+cpp2: 276 |

| Table 10 – Consistency in Start-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 | | | | |
|---|---|---|---|---|
| | | | % Correct Upper Bound | |
| Design Set | Test Set | # Branches Performed | 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× under static prediction and 3.8× 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× under static prediction and 7.2× 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 – these results look almost too good, and may be biased by the nature of the test cases and/or the way the UNIX C compiler code-generator manages control-flow. Extending these results to a greater number of test cases is important if "superpredictors" are to be shown practical. Furthermore, the information-theoretic upper bound on dynamic predictors might be reduced enough to indicate that small superpredictors are near-optimal in practice.

The problem of finding a superpredictor [7] appears to be intractable since it is similar to an $\mathcal{NP}$-complete problem for finite-state transducers [11]. Brute force case-by-case analysis was used to find the optimal predictor, 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 amount of logic 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 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.

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. 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 [6] with Wen-Mei Hwu and Yale Patt. Umesh Vazirani, Chris Perleberg, and Corina Lee provided useful feedback.

## References

[1] Colwell, R.P., Nix, R.P., O'Donnell, J.J., Papworth, D.P., Rodman, P.K. A VLIW Architecture for a Trace Scheduling Compiler. *IEEE Trans. on Computers 37,8* (Aug. 1988), 967-979.

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

[3] Lee, J.K., Smith, A.J. Branch Prediction Strategies and Branch Target Buffer Designs. *IEEE Computer 17,1* (Jan. 1984), 6-22.

[4] Lilja, D.J. Reducing the Branch Penalty in Pipelined Processors. *IEEE Computer 21,7* (July 1988), 47-55.

[5] McFarling, S., Hennessy, J. Reducing the Cost of Branches. *Proc. 13th International Symposium on Computer Architecture,* ACM/IEEE (June 1986), 396-403.

[6] Patt, Y.N., Melvin, S.W., Hwu, W., Shebanow, M.C. Critical Issues Regarding HPS, a High-Performance Microarchitecture. *Proc. 18th Microprogramming Workshop* (Dec. 1985), 109-116.

[7] Ponder, C. String Prediction by a Small Machine. Submitted to *American Mathematical Monthly* (May 1988).

[8] Rau, B.R. CYDRA$^{TM}$ 5 Directed Dataflow Architecture. *Proc. 1988 IEEE Spring Compcon,* IEEE (Feb. 1988), 106-113.

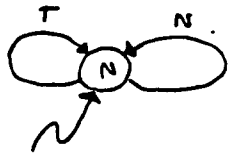
[9] Shebanow, M.C., Patt, Y.N. Autocorrelation, a New Method of Branch Prediction. Submitted to *IEEE Transactions on Computers* (Nov. 1987).

[10] Stiles, D.R., McFarland, H.L. Pipeline Control for a Single Cycle VLSI Implementation of a Complex Instruction Set Computer. *Proc. 1989 IEEE Spring Compcon,* IEEE (Feb. 1989), 504-508.

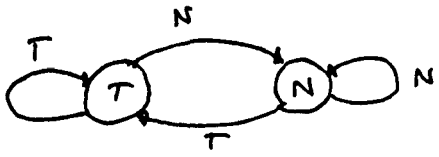
[11] Vazirani, U.V., Vazirani, V.V. A Natural Encoding Scheme Proved Probabilistic Polynomial Complete. *Theoretical Computer Science 24* (1983), 291-300.

# Appendix I – Finite-State Superpredictors

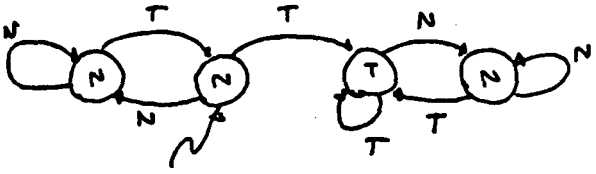
## Designated Start-State




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



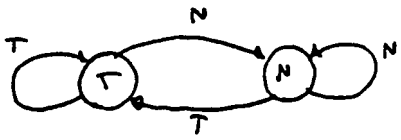

Accuracy: 95.00%
Remarks: Always predict previous input.



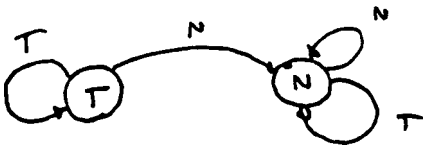

Accuracy: 95.05%
Remarks: Kludge on top of 2-state superpredictor.

## Nondeterministic Start-states


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

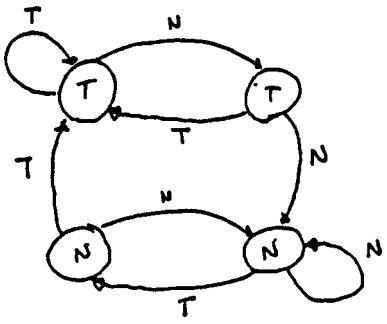



Accuracy: 95.10%
Remarks: Always predict previous input.

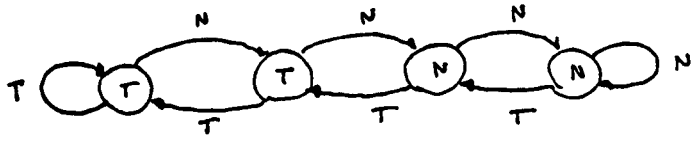



Accuracy: 95.28%
Remarks: 3 disconnected components treat 3 kinds of behavior.

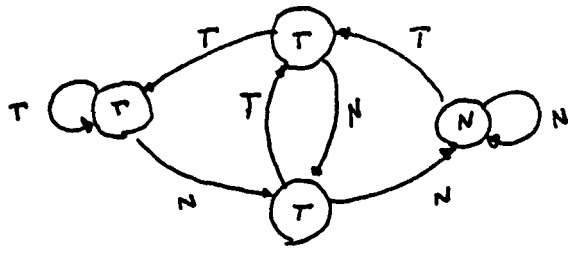
# Appendix II – Miscellaneous Predictors




**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.

Figure 1: Distribution of instruction-block sizes

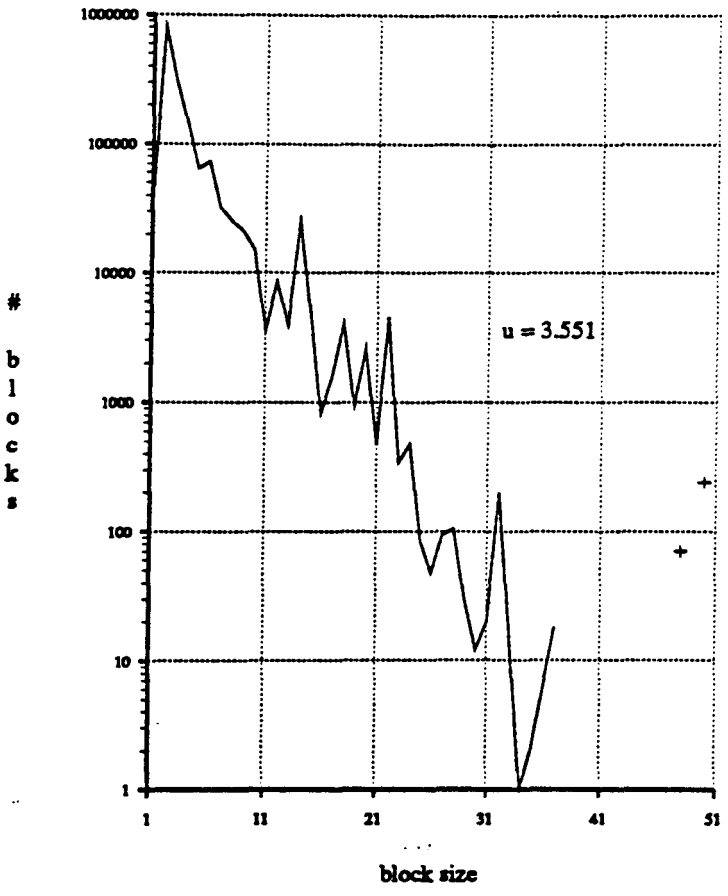


Figure 2: Pipeline utilization contours

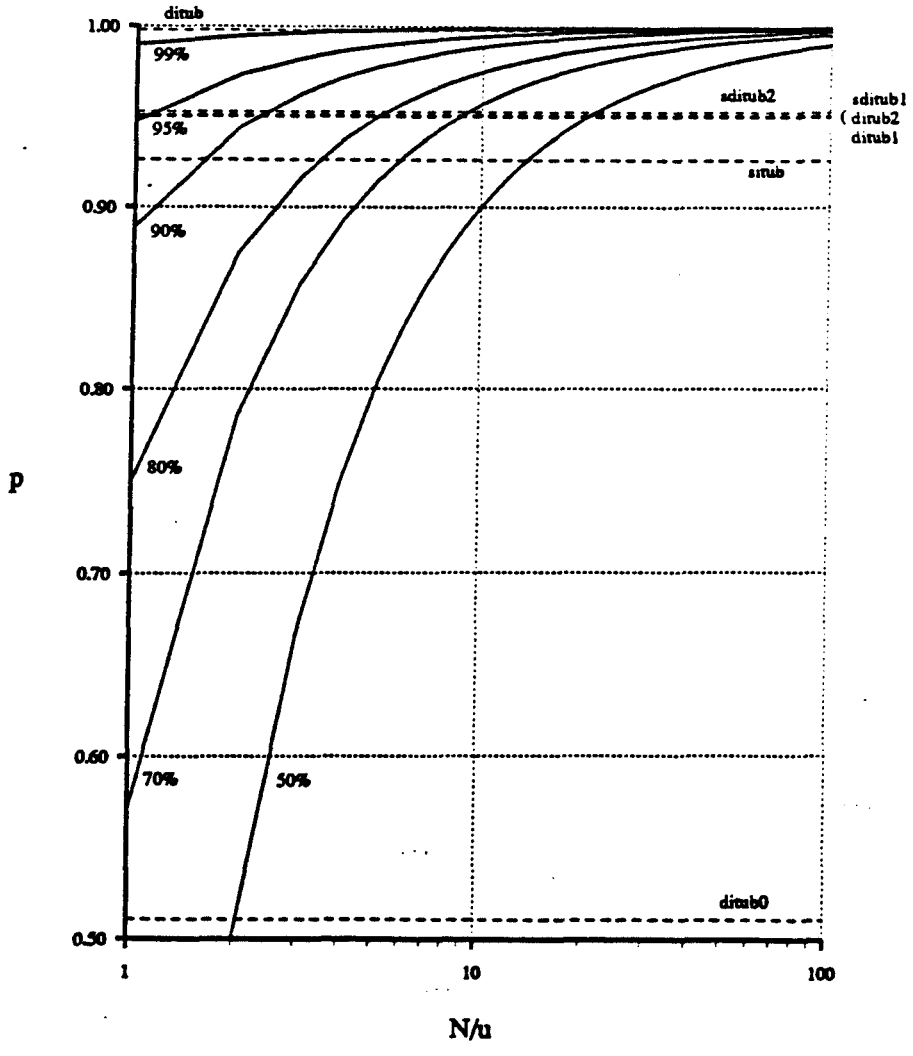


Figure 3: Speedup for infinitely long pipeline

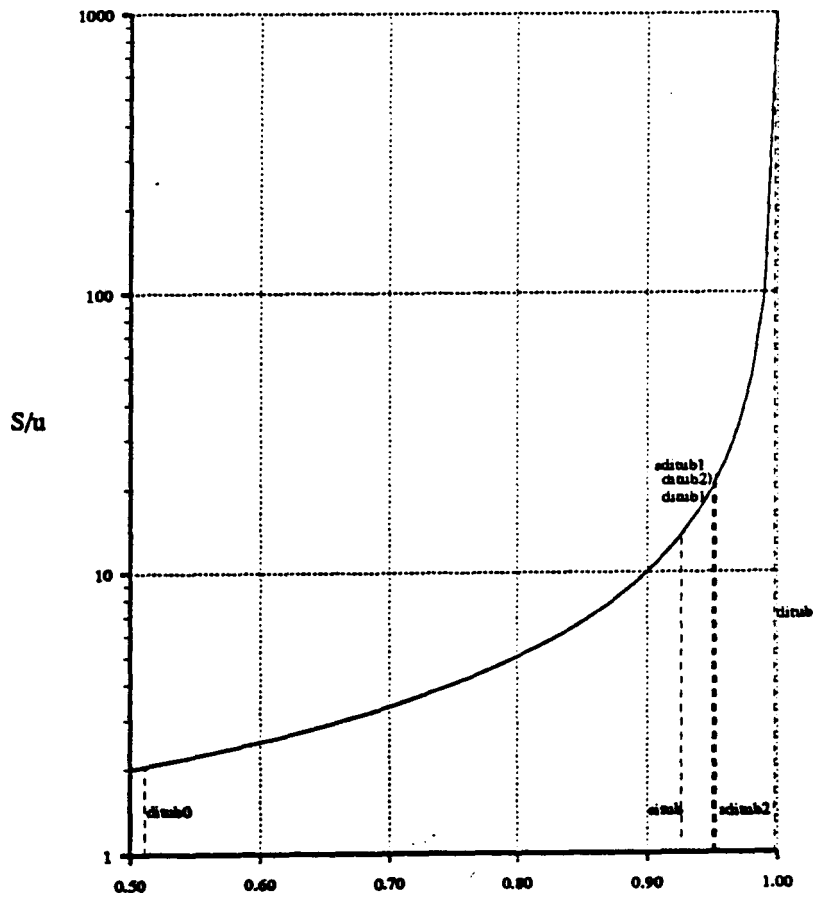


Figure 4: Distribution of history-string lengths

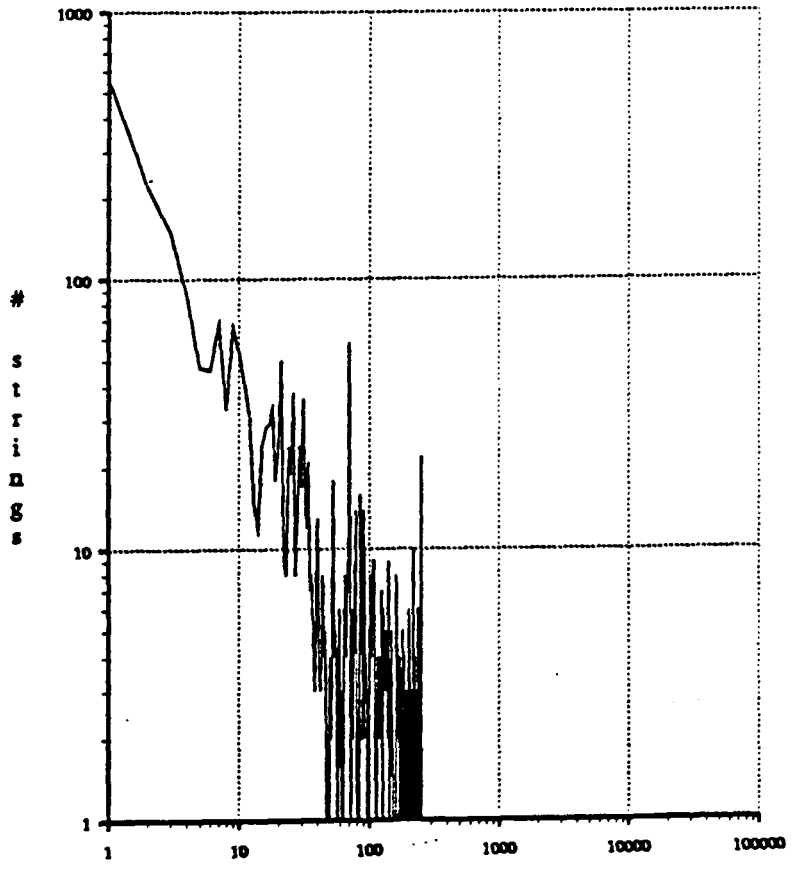


Figure 5: Distribution of taken-branch densities