Branch Prediction For Free

THOMAS BALL
tom@cs.wisc.edu

JAMES R. LARUS
larus@cs.wisc.edu

Computer Sciences Department
University of Wisconsin – Madison
1210 W. Dayton St.
Madison, WI 53706 USA

Technical Report #1137

February 9, 1993

An abridged version of this paper will appear in the 1993 SIGPLAN Conference on Programming Language Design and Implementation (June, 1993).

Branch Prediction For Free

THOMAS BALL[*] JAMES R. LARUS[*]
tom@cs.wisc.edu larus@cs.wisc.edu

Computer Sciences Department
University of Wisconsin – Madison
1210 W. Dayton St.
Madison, WI 53706 USA

Abstract

Many compilers rely on branch prediction to improve program performance by identifying frequently executed regions and by aiding in scheduling instructions. *Profile-based* predictors require a time-consuming and inconvenient compile-profile-compile cycle in order to make predictions. We present a *program-based* branch predictor that performs well for a large and diverse set of programs written in C and Fortran. In addition to using natural loop analysis to predict branches that control the iteration of loops, we focus on heuristics for predicting non-loop branches, which dominate the dynamic branch count of many programs. The heuristics are simple and require little program analysis, yet they are effective in terms of coverage and miss rate. Although program-based prediction does not equal the accuracy of profile-based prediction, we believe it reaches a sufficiently high level to be useful. Additional type and semantic information available to a compiler would enhance our heuristics.

1. INTRODUCTION

In this paper we study the behavior of branches in programs and show that simple, static *program-based* heuristics can predict branch directions with surprisingly high accuracy. Our heuristics go beyond simply identifying loop branches, because in many programs, non-loop branches occur more frequently than loop branches. These heuristics are inexpensive to employ and simple to implement, yet accurately predict a high percentage of loop and non-loop branches for a large and diverse set of programs, including many programs with complex conditional control flow.

Our measurements show that a perfect static predictor has the potential to predict dynamic loop *and* non-loop branches with a miss rate of approximately 10%. Naive strategies that always predict the target or fall-thru successor of a non-loop branch have a miss rate of approximately 50%. Our heuristic has an average miss rate of 26% for

[*] This work was supported by the National Science Foundation under grants CCR-8958530 and CCR-9101035.

An abridged version of this paper will appear in the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation.

non-loop branches, with good performance on benchmarks that appear, at first, difficult to predict. Taking into account loop branches—for which we employ a more accurate heuristic than the common technique of simply identifying backwards branches—our heuristics have an average miss rate of 20%.

Many compiler optimizations rely on branch prediction to identify heavily-executed paths [6, 12, 14]. In addition, recently introduced architectures, such as the DEC Alpha [15] and MIPS R4000 [9], exact a heavy pipeline penalty for a mispredicting a branch (up to 10 cycles [15]). To help alleviate this problem, some architectures predict that forward conditional branches are not taken and backward conditional branches are taken, thereby relying on a compiler to arrange code to conform to these expectations. Run-time profile information [2, 8] from a program execution typically is used to statically predict branch directions. Fisher and Freudenberger observed that profiled-based static branch prediction works well because most branches take one direction with high probability and the highly probable direction is the same across different program executions [7].

Profile-based branch prediction can be quite accurate, but it is inconvenient and time-consuming to use. First, a program is compiled. To be profiled, a program must be instrumented with counting code, which may be done by the compiler or another tool. The instrumented program executes (possibly several times), which produces a profile. Finally, the program can be recompiled with the aid of the profile. This process requires two compilations and an execution. Furthermore, when the program changes, the entire process must be repeated. On the other hand, program-based prediction can be employed during the original compilation to make branch predictions. Although program-based prediction is a factor of two worse, on the average, than profile-based prediction, we believe it reaches a sufficiently high level to be useful.

This paper is organized as follows. Section 2 contains background material and describes the benchmark programs. Section 3 classifies loop and non-loop branches and compares their behavior. Section 4 presents several simple heuristics for non-loop branches and measures their effectiveness in isolation. Section 5 considers combining these simple heuristics into a complete heuristic and contains the results for this heuristic. Section 6 presents results on how our heuristic performs at finding sequences of instructions without a mispredicted branch. We compare profile-based methods for measuring this quantity with trace-based methods and show why trace-based methods are preferable. Section 7 examines the performance of our heuristic on different datasets. Section 8 reviews related work and Section 9 concludes the paper.

2. BACKGROUND

We restrict our heuristics to predicting two-way conditional branches with fixed targets. Throughout the paper, the word *branch* refers to such branches. We do not consider branches whose target is dynamically determined (by lookup in a jump table, for example). Associated with each conditional branch instruction is its *target* successor—the instruction to which control passes if the branch condition evaluates to true—and its *fall-thru* successor—the instruction to which control passes if the branch condition evaluates to false.

We used our profiling and tracing tool QPT [2] both as a platform for studying branch behavior and for making branch predictions. QPT takes as input a MIPS executable file and produces an instrumented program that generates a branch profile (*i.e.*, for each branch, a count of how many times control passes to the target successor and fall-thru successor) when run. QPT can also instrument a program to produce an instruction and address trace. Since QPT operates on an executable file, all program procedures are analyzed. The numbers in this paper include DEC Ultrix 4.2 library procedures as well as application procedures.

In order to instrument an executable file, QPT builds a control flow graph for each procedure in the executable file. Each vertex in the control flow graph represents a basic block of instructions. A basic block ending with a conditional branch corresponds to a vertex in the control flow graph with two outgoing edges. The root vertex of the control flow graph is the entry point of the procedure. A basic block containing a return (procedure exit) has no successors in the control flow graph.

Some of our heuristics make use of the control flow graph's domination and postdomination relations [1]. A vertex *v* dominates *w* if every path from the entry point of the procedure to *w* includes *v*. A vertex *w* postdominates *v* if every path from *v* to any exit vertex includes *w*. If the successor of a branch postdominates the branch, then no matter which direction the branch takes, the successor eventually executes.

We analyzed the programs in the SPEC89 benchmark suite [4], along with a number of other programs. These benchmarks (23 of them) are listed in Table 1, along with a description of their function. We have broken the benchmarks into two major groups: programs that perform little to no floating point computation and programs that perform many floating point operations. Within each group, programs are sorted by the size of their object code. All of the benchmarks were compiled and analyzed on a DECstation (a MIPS R2000/R3000 processor) with -O optimization.

The results presented in Sections 3 to 6 are for a single execution of each benchmark. Previous work has shown that most branches behave similarly over different executions [7]. That is, if a branch takes one direction with high probability during one execution of a program, it most likely takes the same direction with high probability in other executions. The goal of this work is to show that static prediction can accurately determine these branch directions, rather than confirm previous results. However, we also tested our predictor on multiple datasets per benchmark and found similar results to those of [7]. Section 7 summarizes the results of these experiments.

We are concerned with *static* branch prediction. That is, for each branch either the target successor or fall-thru successor is predicted, and this prediction does not change during the execution of the program. Predicting a branch corresponds to choosing one of the two outgoing edges from the vertex containing the branch in the control flow graph. The standard for how well static branch prediction potentially performs is the *perfect static predictor*, which predicts the more frequently executed outgoing edge of each branch in a program. If the perfect predictor has a low miss rate, then most branches follow one direction with high probability. If most branches take both directions with approximately equal probability, a perfect static predictor would do no better than a 50% miss rate.

Program	Description	Language	Code Size (X 1000 bytes)
congress	Interpreter for Prolog-like language	C++	856
ghostview	X postscript previewer	C	831
gcc *	GNU C compiler	C	688
lcc	Fraser and Hanson’s C compiler	C	254
rn	Net news reader	C	221
espresso *	PLA minimization	C	188
qpt	Profiling and tracing tool	C	143
awk	Pattern scanning and processing	C	102
xlisp *	Lisp interpreter	C	78
eqntott *	Boolean equations to truth table	C	45
addalg	Integer program solver	C	33
compress	File compression utility	C	25
grep	Search file for regular expressions	C	20
poly	Polydominoes game	C	16
spice2g6 *	Circuit simulation	F	385
doduc *	Monte Carlo hydrocode simulation	F	184
fpppp *	Two-electron integral derivative	F	168
dnasa7 *	Floating point kernels	F	90
tomcatv *	Vectorized mesh generation	F	66
matrix300 *	Matrix multiply	F	61
costScale	Solve minimum cost flow	C	41
dcg	Conjugate gradient	C	41
sgefat	Gaussian elimination	C	33

Table 1. Benchmarks, sorted by code size. SPEC benchmarks are marked with *. Fortran benchmarks are marked with an F.

The perfect predictor provides an upper bound on the performance of any static predictor. Branches for which the perfect predictor performs poorly will not be predicted well by any static predictor. To measure how well a predictor *P* performs (for a given set of branches) we use two percentages (notated *C*/*D*), where *C* is the percentage of the dynamic branches that are mispredicted by *P* (*i.e.,* miss rate), and *D* is the miss rate for the perfect predictor.

3. LOOP BRANCHES AND NON-LOOP BRANCHES

In this section we show that predicting non-loop branches is key to good branch prediction for many programs. In addition, we show that a static predictor can potentially do very well at predicting non-loop branches.

First, we precisely classify branches as loop or non-loop. Backwards branches in code (a backwards branch passes control to an address that is before the address of the branch instruction) usually control the iteration of loops. However, many non-backwards branches can also control the iteration of loops either by exiting the loop or continuing the iteration [13]. For many of the benchmarks, loop branches that are not backwards branches account for a very high percentage of loop branches (for example, 40% of dynamic loop branches in *xlisp* were not backwards branches and 45% of loop branches in *doduc* were not backwards branches). Such branches can be easily identified by natural loop analysis [1] of the control flow graph, as we now review.

Each vertex that is the target of one (or more) loop backedges (as identified by a depth-first search of the control flow graph from the root vertex) is a *loop head*. Removing the backedges from a control flow graph eliminates all directed cycles. The natural loop of a loop head y is:

$$\text{nat-loop}(y) = \{y\} \cup \{\, w \mid \text{there exists a backedge } x \rightarrow y \text{ and a } y\text{-free path from } w \text{ to } x \,\}$$

An edge $v \rightarrow w$ is an exit edge if there is loop nat-loop(y) such that $v \in$ nat-loop(y) and $w \notin$ nat-loop(y). It is clear from the definition of natural loop that for any vertex v in nat-loop(y), at least one of v's successors must be in nat-loop(y). Therefore, for any vertex, either none of its outgoing edges are exit edges, or exactly one of its outgoing edges is an exit edge. We classify branches as follows:

- a branch is a *loop* branch if either of its outgoing edges is an exit edge or a loop backedge.
- a branch is a *non-loop* branch if neither of its outgoing edges is an exit edge or a backedge.

Loop branches can be very accurately predicted, as follows: if either of the outgoing edges is a backedge, it is predicted.[1] Otherwise, the non-exit edge is predicted. The intuition is that loops iterate many times and only exit once. The loop prediction chooses iterating over exiting. Figure 1 illustrates loop and non-loop branches. Edges $D \rightarrow B$ and $E \rightarrow B$ are backedges. There is one natural loop (with loop head B) which contains the vertices B, C, D, and E. The exit edges are $C \rightarrow F$ and $E \rightarrow F$. Vertices A and B are non-loop branches, while C, D and E are loop branches. The predictions for the three loop branches are $C \rightarrow D$, $D \rightarrow B$, and $E \rightarrow B$, respectively.

Table 2 shows the breakdown of dynamic branches in each benchmark according to this loop classification scheme. Within each group, programs are ordered by the percentage of dynamic branches that are non-loop branches ("% of All Branches"). Many programs' executions are dominated by non-loop branches. We first consider how our loop branch predictor performs. Under "Loop Branches", the column "Pred./Perf." contains the miss rates for the loop predictor and the perfect predictor, applied to loop branches. The results are not too surprising:

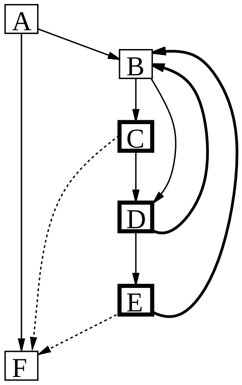


Figure 1. Control flow graph with loop. Bold edges are backedges and dashed edges are exit edges. Vertices *C*, *D*, and *E* are loop branches.

[1] Although it is theoretically possible for both the outgoing edges from a branch to be loop backedges, this never occurred in our analysis of the benchmarks. If it did occur, one could predict the edge that leads to the innermost loop.

the loop predictor does very well, and in some cases approaches the perfect predictor (*compress, addalg, eqntott*). The mean miss rate for the loop predictor is 12% ± 10%.

We now consider non-loop branches. The perfect predictor (“Perf.”) performs very well for all benchmarks, implying that most non-loop branches take one direction with high probability. For some benchmarks, non-loop branches are “better behaved” than loop branches! For instance, for *gcc* and *xlisp*, a perfect static predictor does better on non-loop branches than on loop branches.

“Target” shows the performance of a simple strategy that always predicts the target successor. Not surprisingly, this heuristic does not fare well. Sometimes it pays to choose the target (*grep*), sometimes the fall-thru (*compress*), and sometimes neither does well (*qpt, rn*). The mean miss rate is 51% ± 19%. In fact, for many benchmarks, random prediction (“Random”) performs as well as or better than predicting the target (mean of 49% ± 13%). These numbers show that simply predicting the target or fall-thru produces results of varying quality with mediocre overall

Program	**Loop Branches**	**Non-Loop Branches**				
	Pred./Perf. (%/%)	**% of All** **Branches**	Target/Perf. (%/%)	Random/Perf. (%/%)	Big (num	%)
gcc	22/15	**73**	46/11	50/11	0	0
lcc	18/14	**71**	47/12	52/12	1	13
qpt	19/14	**70**	56/9	52/9	0	0
compress	12/12	**66**	72/18	66/18	6	69
xlisp	28/19	**62**	67/7	50/7	0	0
addalg	7/7	**52**	43/30	43/30	7	67
ghostview	8/6	**52**	45/4	47/4	4	53
eqntott	3/2	**49**	73/25	50/25	2	92
rn	7/3	**48**	51/1	51/1	3	25
grep	26/2	**44**	34/0	3/0	3	96
congress	21/12	**40**	37/3	57/3	2	10
espresso	18/12	**37**	59/13	42/13	3	24
awk	4/3	**29**	51/3	57/3	4	29
poly	11/10	**20**	50/3	31/3	3	54
fpppp	34/34	**86**	41/9	41/9	0	0
costScale	7/6	**71**	48/21	49/21	6	52
doduc	8/7	**52**	62/3	49/3	0	0
tomcatv	1/1	**38**	2/0	50/0	2	98
dcg	2/2	**21**	40/4	46/4	4	51
spice2g6	9/8	**21**	53/8	52/8	2	27
sgefat	2/2	**18**	28/8	61/8	8	73
dnasa7	1/1	**10**	68/4	55/4	4	58
matrix300	1/1	**4**	99/0	66/0	3	99
MEAN	12/8		51/10	49/10		
Std.Dev.	10/8		19/8	13/8		

Table 2. Dynamic breakdown of loop branches vs. non-loop branches. For each class of branch, “Perf.” is the miss rate for the perfect predictor. “Pred.” shows the miss rate for the loop predictor on loop branches. “% of All Branches” is the percentage of all branches that are non-loop branches. “Target” shows the results for predicting the target successor of each non-loop branch and “Random” shows the results for predicting each non-loop branch randomly. Finally, “Big” shows how many non-loop branches in the program contributed more than 5 percent of all dynamic non-loop branches, and what percentage these “big” branches account for.

performance. For a compiler to predict non-loop branches well for an architecture such as the DEC Alpha, a more sophisticated strategy is necessary.

Correctly predicting a frequently executed branch has a high payoff. The column “Big” shows how many distinct non-loop branches in each program generate more than 5 percent of the dynamic non-loop branch executions and the percentage of dynamic non-loop branches accounted for by these branches. For some benchmarks (*eqntott, grep, tomcatv, matrix300*) a handful of non-loop branches in the program produce most of the dynamic non-loop branches. For such programs, the performance of a predictor depends crucially on predicting these branches correctly. Other programs (*gcc, lcc, qpt, xlisp, congress, doduc*) execute many different branches, each of which contributes a small percentage of the dynamic non-loop branches.

To summarize the main points of this section:

- Branches that control the iteration of loops can be identified and predicted accurately with the aid of natural loop analysis. Any branch not identified as a loop-branch by natural loop analysis cannot directly control the iteration of a loop.
- For many programs, non-loop branches dominate the loop branches and must be predicted accurately to get good overall branch prediction. Naively predicting the target or fall-thru successor for non-loop branches produces middling results.
- Static prediction has the potential to accurately predict non-loop branches since most non-loop branches choose one direction with high probability.

4. SIMPLE HEURISTICS FOR PREDICTING NON-LOOP BRANCHES

This section examines a number of heuristics for predicting non-loop branches. The heuristics are completely automatic and only make use of information available from an executable file. Some heuristics could clearly be refined and made more accurate with source-level information available to a compiler. This section examines the performance of each heuristic in isolation. The next section discusses how we combined these simple heuristics.

Table 3 summarizes the results for each benchmark and heuristic. Each entry in the table presents the percentage of (dynamic) non-loop branches to which the heuristic applies (bold number) and the miss rates (for the heuristic and perfect predictors). A table entry is left blank if the percentage of branches covered is less than one percent. For reference, the second column of the table (“NL”) repeats the percentage of all branches that are non-loop branches. It is useful to keep this percentage in mind when examining the effectiveness of these heuristics.

4.1. Opcode Heuristic

We predict some branches based on the branch instruction opcode. The MIPS R2000 has integer branch instructions that branch if an operand register is less than, less than or equal, greater than, or greater than or equal to zero (`bltz, blez, bgtz, bgez`). Because many programs use negative integers to denote error values, the heuristic predicts that `bltz` and `blez` are not taken and that `bgtz` and `bgez` are taken. The heuristic also identifies floating point comparisons that check if two floating point numbers are equal, predicting that such tests usually evaluate false. As Table 3 shows, the Opcode heuristic performs very well for most benchmarks, although its coverage varies widely. The heuristic performs poorly on *spice2g6* because of a high number of integer branches that compare against a negative value.

4.2. Loop, Call, Return, Guard, and Store Heuristics

The following heuristics are based on properties of the basic block successors of a branch. Each heuristic consists of two pieces of fixed information, a *selection property* and a *predictor*. If neither successor to the block containing the conditional branch has the selection property or both have the property, no prediction is made. If exactly one successor has the property, the predictor chooses either the successor with the property, or the successor without the property, depending on the heuristic.

Program	NL	Opcode		Loop		Call		Return		Guard		Store		Point	
gcc	73	**12**	26/5	**8**	34/8	**18**	28/8	**10**	32/10	**41**	36/12	**27**	33/7	**9**	46/16
lcc	71	**2**	33/2	**10**	36/12	**23**	14/5	**9**	33/6	**48**	29/14	**47**	47/13	**32**	37/19
qpt	70	**7**	31/14	**10**	30/10	**31**	18/6	**22**	28/5	**46**	16/5	**24**	39/8	**10**	15/10
compress	66	**56**	29/13	**4**	87/12	**32**	6/6			**53**	41/26	**58**	78/9		
xlisp	62	**3**	1/1	**20**	12/2	**44**	25/7	**35**	20/1	**41**	13/3	**29**	65/16	**20**	14/0
addalg	52	**19**	19/15	**8**	16/5	**8**	42/35	**2**	43/1	**57**	58/29	**30**	39/21	**64**	51/38
ghostview	52	**47**	1/1	**17**	73/11	**22**	18/1	**7**	64/5	**51**	60/5	**29**	64/1	**18**	19/10
eqntott	49	**2**	2/1	**3**	10/1			**2**	0/0			**1**	63/13	**2**	88/12
rn	48	**37**	8/1	**6**	11/6	**45**	9/1	**6**	36/0	**35**	33/1	**47**	55/1	**6**	77/1
grep	44					**68**	0/0			**33**	2/1				
congress	40	**3**	12/4	**4**	16/5	**46**	29/3	**31**	47/3	**42**	38/4	**26**	42/6	**34**	15/6
espresso	37			**21**	22/3	**3**	27/4	**5**	33/28	**27**	69/16	**16**	34/21		
awk	29	**11**	0/0	**9**	4/1	**53**	4/1	**16**	17/5	**38**	9/2	**25**	58/1	**21**	10/2
poly	20	**25**	0/0	**34**	0/0	**32**	36/4	**46**	36/3	**2**	100/0	**31**	29/6		
fpppp	86	**22**	20/6	**5**	9/0	**11**	54/30	**15**	29/4	**52**	27/8	**25**	47/6	**13**	91/2
costScale	71	**8**	2/0	**3**	13/3	**5**	9/2	**2**	8/0	**39**	45/26	**39**	40/17	**1**	23/16
doduc	52	**42**	31/6	**7**	4/0	**10**	58/2	**17**	27/1	**44**	25/3	**27**	30/4		
tomcatv	38									**99**	99/0	**99**	1/0		
dcg	21	**13**	7/3	**6**	14/5	**7**	5/1	**51**	4/1	**28**	9/3	**9**	68/4	**1**	51/2
spice2g6	21	**40**	81/2	**3**	36/1	**23**	9/4	**15**	20/5	**33**	29/9	**29**	15/3		
sgefat	18	**8**	12/4	**10**	1/1	**5**	26/1	**3**	42/1	**49**	34/5	**44**	16/9		
dnasa7	10	**23**	11/2	**7**	1/1			**15**	32/7	**78**	33/4	**2**	73/9		
matrix300	4	**67**	0/0	**33**	100/0			**66**	0/0	**99**	33/0				
MEAN			16/4		25/4		22/6		28/4		38/8		45/8		41/10
Std.Dev.			19/5		28/4		17/10		16/6		26/9		20/6		29/11

Table 3. The effectiveness of each heuristic for predicting non-loop branches, applied individually. For each heuristic, the table shows the percentage of dynamic non-loop branches to which the heuristic applies (in bold) and the miss rates for those branches. A table entry is left blank if the coverage is less than one percent of all non-loop branches. Blank entries are not counted in the mean and standard deviation.

Loop Heuristic

> The successor does not postdominate the branch and is either a loop head or a loop preheader (*i.e.*, passes control unconditionally to a loop head which it dominates). If the heuristic applies, predict the successor *with* the property.

The loop heuristic determines if a branch chooses between executing or avoiding a loop and predicts that loops are executed rather than avoided. Many compilers generate code for **while** loops and **for** loops by generating an **if-then** around a **do-until** loop, replicating the loop test in the **if-then** condition (this strategy avoids generating an extra unconditional branch). The heuristic catches these cases as well as branches around loops explicitly specified in the program. The performance of this heuristic is quite good except on *compress*, *ghostview*, and *matrix300*. It has excellent coverage and/or performance on *xlisp* and *espresso* and *doduc*.

Call Heuristic

> The successor block contains a call or unconditionally passes control to a block with a call that it dominates, and the successor block does not postdominate the branch. If the heuristic applies, predict the successor *without* the property.

This heuristic surprised us. Initially, we had believed that a branch that decided between executing or avoiding a call would execute the call, as programs typically make calls to perform useful work. However, the numbers strongly show the exact opposite, especially for the first set of programs. In examining the programs we found that many conditional calls are to handle exceptional situations. Just one example of this is printing. For many programs, printing is an exceptional occurrence. Even in applications that print to standard output or to a file, most printing is done unconditionally rather than conditionally.

Return Heuristic

> The successor block contains a return or unconditionally passes control to a block that contains a return. If the heuristic applies, predict the successor *without* the property.

There are several justifications for this predictor, the most compelling of which is recursion. Because programs must loop or recurse to do useful work, we expect that loops iterate and that recursive procedures recurse. A return is the base case, which is the exception in recursion, just as a loop exit is the exception in iteration. In addition, many returns from procedures handle cases which occur infrequently (*i.e.*, error and boundary conditions). The performance of the return heuristic is good over most of the benchmarks.

Guard Heuristic

> Register *r* is an operand of the branch instruction, register *r* is used in the successor block before it is defined, and the successor block does not postdominate the branch. If the heuristic applies, predict the successor *with* the property.[2]

This heuristic analyzes both integer and floating point branches. It attempts to find instances in which a branch on a value guards a later use of that value. The intuition is that the function of many guards is to catch exceptional conditions and that the common case is for a guard to allow the value to flow to its use.

The coverage for this heuristic is quite high over most of the benchmarks (we remind that reader that the heuristic applies only if exactly one of the successors has the property). The performance is fairly good over most of the benchmarks, with stronger performance on the second set of programs. The heuristic performs well on most of the pointer-chasing programs (*gcc, lcc, qpt, xlisp, congress*) because the common case for a null pointer test guarding the use of the same pointer is that the pointer is not null. The heuristic performs very poorly on *tomcatv*, mispredicting the two branches that account for 99% of all non-loop branches. These branches are inside a loop that determines the maximum value in an array of values (*i.e.*, **if** (a[i,j] > max) **then** max := a[i,j] **fi**). In this program the common case is to avoid updating the maximum, but the guard heuristic predicts the opposite.

We note that global register allocation can greatly affect the coverage of this heuristic. Without performing global register allocation on the benchmarks, the coverage for this heuristic would be much lower due to reloads of values, which the heuristic does not detect.

Store Heuristic

> The successor block contains a store instruction and does not postdominate the branch. If the heuristic applies, predict the successor *without* the property.

We tried this heuristic more because of curiosity than intuition about how it might perform. On the first set of benchmarks, the store heuristic has very poor performance and coverage is very high. However, the performance improves on the floating-point intensive benchmarks. On *tomcatv*, the heuristic performs perfectly, correctly predicting the two branches that the guard heuristic mispredicted.

4.3. Pointer comparisons

Pointer comparisons either compare a pointer to null or compare two pointers. As mentioned before, in pointer-manipulating programs, most pointers are non-null. Furthermore, we expect equality comparison of two pointers to rarely be true. To distinguish pointer comparisons from other comparisons requires type information that we did not have. But because the MIPS is a RISC architecture, there are very few possible code sequences for pointer

[2] The heuristic does not analyze past calls in the succesor blocks since no interprocedural register use or definition information is computed.

comparisons. Two of the code sequences are shown below:

```
load rM, ...                   load rM, ...
 ...                           load rN, ...
beq r0, rM, ...                 ...
                               beq rM,rN, ...
```

The pointer heuristic looks for these two cases in the basic block containing the branch and predicts that the fall-thru is taken. It also looks for the same patterns with a `bne` branch and predicts that the branch is taken.[3] Of course, similar code sequences may be generated for comparisons that do not involve pointers. Our heuristic does not distinguish these cases from those involving pointers. However, a compiler could easily make the distinction. We made one small optimization to the heuristics, noting that many pointers are either local variables and addressed off the `SP` register (stack pointer), or are in the heap and addressed off a register other than `SP` or `GP` (pointer to global storage). If either load instruction loads off of `GP`, the branch is not considered. If a local pointer variable is allocated a register, then the heuristic will miss comparisons involving that pointer.

The results for some pointer-chasing programs such as *lcc*, *xlisp*, *qpt*, and *congress* are fairly good. For some other pointer-manipulating programs, such as *gcc*, the heuristic does not perform as well. This is because the heuristic picks up comparisons of variables that are not pointer types. The heuristic could certainly be improved by incorporating type information. As expected, the pointer heuristics performs poorly on the floating point benchmarks, which contain little to no pointer manipulation.

4.4. Discussion

Unsuccessful Heuristics

We tried many heuristics that were unsuccessful. These included heuristics that were based on the number of instructions between a branch and its target, and the domination and postdomination relations between a branch and its successors.

Generalizations

All of the heuristics discussed above are very local in nature. Excluding the information made available from natural loop, domination, and postdomination analysis, they examine only information from the basic block containing the conditional branch or in successors of the block (at most two steps away). Some of the heuristics could clearly be generalized to consider more basic blocks. For example, the guard heuristic could look farther away from the branch to see if the branch value is reused by an instruction whose execution is controlled by the branch. Other heuristics could be similarly generalized. It remains to be seen how such generalizations affect the coverage and performance of the heuristics.

[3] In both cases, the heuristic does not apply if there is a call instruction between the load and the branch.

5. A BRANCH PREDICTION HEURISTIC AND ITS EFFECTIVENESS

This section describes how we combined the heuristics from the previous section into a single heuristic procedure for predicting non-loop branches. It is clear that more than one heuristic can apply to a branch. We chose to combine the heuristics by totally ordering them. To predict a branch, the combined heuristic simply marches through the heuristics until one applies and uses it to predict the branch. We will discuss later what to do for branches for which no heuristic applies. Many other approaches for combining the heuristics are possible, such as a voting protocol with weightings. However, for any such approach there is the central problem of prioritizing the heuristics.

As Graph 1 shows (see the Appendix for graphs), the ordering of the heuristics can have quite an impact on miss rate. The graph shows the average miss rate (for all benchmarks except *matrix300*) for non-loop branches for every possible order (there are 7! = 5040 possible orderings), where the orders have been sorted by miss rate.

How does one choose an order for the heuristics? The best one can do is to use analysis of available benchmarks to select a good order and hope that the order works well in the future when additional benchmarks are encountered. We performed the following experiment to see if it is reasonable to expect that orders picked for a subset of the benchmarks will perform well over all benchmarks. To get an even number of benchmarks we eliminated *matrix300*, the least interesting of the benchmarks in terms of non-loop branch prediction. For each subset of cardinality 11 of the remaining 22 benchmarks we computed the order that minimized the average miss rate (for non-loop branches) for the benchmarks in the subset.[4] These 11 benchmarks represent the "known" benchmarks. Using the chosen order, we computed the average miss rate for all 22 benchmarks. The experiment consisted of ($\binom{22}{11} = 705,432$) trials, one for each subset. Of the possible 5040 orders, only 622 appeared in the trials. Graph 2 shows the 101 most frequently occurring orders (ordered by frequency) versus the cumulative percentage of all trials that these orders appeared in. As can be seen, almost 90% of the trials are accounted for by the 40 most

% of Trials	Miss Rate	Order						
8.92	26.00	Opcode	Call	Return	Store	Point	Loop	Guard
8.50	25.52	Call	Opcode	Return	Store	Point	Loop	Guard
7.52	25.50	Point	Call	Opcode	Return	Store	Loop	Guard
5.82	25.59	Point	Loop	Call	Opcode	Return	Store	Guard
5.22	27.12	Opcode	Call	Return	Store	Point	Guard	Loop
4.99	25.99	Point	Opcode	Call	Return	Store	Loop	Guard
4.86	26.64	Call	Opcode	Return	Store	Point	Guard	Loop
4.82	26.04	Loop	Call	Opcode	Return	Store	Point	Guard
4.58	25.55	Call	Opcode	Return	Point	Store	Loop	Guard
4.40	26.02	Opcode	Call	Return	Point	Store	Loop	Guard

Table 4. The 10 most common orders from the $\binom{22}{11}$ experiment. Shown with each order are the percent of all trials in which the order appeared and the average miss rate (all 22 benchmarks) for that order.

[4] Each benchmark gets equal weight in this average. It would also be interesting to use a weighted average that accounts for the percentage of dynamic non-loop branches in and number of predictions made for each benchmark.

frequently occurring orders. Graph 3 shows the average miss rate (for all 22 benchmarks) for each of the 101 most frequently occurring orders.

The results of our analysis are encouraging. The 40 most common orders account for approximately 90% of all trials and the average miss rate for most of these orders is below 27%. The order that occurred third most frequently was also the order that minimized the average miss rate for all benchmarks. Table 4 shows the 10 most common orders from the experiment and the percentage of the trials that each one accounts for. The *Opcode*, *Call*, and *Return* heuristics are consistently among the top 3 heuristics in these orders.

Computing the order that minimizes the miss rate for a set of benchmarks is not an inexpensive proposition, especially as the number of heuristics grows. A less expensive approach that we explored was pair-wise analysis: we examined pairs of heuristics and for the set of branches in the intersection, compared the performance of the two heuristics to determine a pair-wise ordering. The orders we found with this analysis were generally inferior to those found by the previous experiment, but were in the top quarter of performers.

Program	Point		Call		Opcode		Return		Store		Loop		Guard		Default	
gcc	**9**	46/16	**17**	28/7	**10**	28/5	**6**	30/10	**15**	28/8	**3**	32/5	**19**	33/14	**21**	56/14
lcc	**32**	37/19	**21**	12/3	**2**	29/1	**4**	45/7	**7**	31/7	**2**	65/20	**11**	32/11	**20**	41/11
qpt	**10**	15/10	**26**	14/6	**6**	30/14	**13**	15/6	**5**	47/7	**3**	13/12	**11**	10/3	**24**	55/14
compress			**32**	6/6	**24**	60/22			**18**	62/19			**16**	51/49	**10**	43/0
xlisp	**20**	14/0	**37**	23/9			**15**	18/2	**11**	71/21			**7**	7/1	**10**	59/12
addalg	**64**	51/38	**1**	36/5	**19**	19/15			**4**	14/1	**2**	35/9			**10**	60/27
ghostview	**18**	19/10	**21**	15/1	**33**	1/1	**2**	33/9	**9**	8/1			**4**	40/13	**12**	45/7
eqntott	**2**	88/12			**2**	1/1									**95**	50/26
rn	**6**	77/1	**42**	10/1	**6**	1/1	**5**	31/0	**13**	50/3			**5**	83/0	**21**	57/1
grep			**68**	0/0											**32**	0/0
congress	**34**	15/6	**32**	22/3	**2**	23/7	**13**	50/1	**1**	25/2	**3**	1/0	**4**	20/8	**11**	68/1
espresso			**2**	26/4			**4**	31/30	**14**	35/22	**16**	6/3	**6**	48/26	**56**	26/12
awk	**21**	10/2	**38**	3/2			**8**	11/5	**3**	33/0	**1**	29/3	**4**	17/2	**24**	34/7
poly			**32**	36/4	**21**	0/0	**24**	69/6	**12**	65/4					**11**	39/0
fpppp	**13**	91/2	**11**	54/30	**18**	14/1	**12**	28/5	**16**	54/6			**12**	8/3	**18**	50/19
costScale	**1**	23/16	**5**	8/1	**4**	4/0	**2**	3/0	**33**	32/19			**25**	38/36	**29**	26/18
doduc			**10**	58/2	**39**	33/6	**14**	25/0	**14**	14/3			**15**	31/1	**8**	55/1
tomcatv									**99**	1/0						
dcg	**1**	51/2	**7**	3/0	**8**	11/4	**48**	3/2	**4**	82/4	**1**	38/11	**3**	32/5	**27**	24/8
spice2g6			**23**	9/4	**20**	76/1	**14**	20/5	**5**	12/3	**1**	97/0	**11**	27/16	**25**	43/19
sgefat			**5**	27/1	**5**	19/5	**2**	46/1	**40**	11/10	**9**	1/0	**35**	46/6	**4**	53/29
dnasa7					**22**	11/1	**13**	22/7			**3**	0/0	**56**	40/4	**5**	70/3
matrix300					**67**	0/0					**33**	100/0				
MEAN		41/10		21/5		20/5		28/6		36/7		35/5		33/12		45/11
Std.Dev.		29/11		17/7		21/6		17/7		23/7		36/6		19/14		17/9

Table 5. The performance of the simple heuristics, when applied in a prioritized ordering (left to right). If no heuristic applies to a branch, it is covered by the "Default" heuristic, which predicts randomly (the same prediction is made as in Table 2). A table entry is left blank if the coverage is less than one percent of all non-loop branches. Blank entries are not counted in the mean and standard deviation.

Table 5 presents the results for the simple heuristics applied in the order *Point*→*Call*→*Opcode*→*Return*→*Store*→*Loop*→*Guard*. Recall that as soon as a heuristic applies, the prediction is made and the next branch is considered. If no heuristic applies to a branch, then a *Default* prediction is made, which is simply a random prediction. For these branches, the same prediction is made as the random prediction for Table 2.

Table 6 presents the results of the combined heuristic for non-loop branches. The column “Heuristics” shows the percentage of dynamic non-loop branches covered by the heuristics excluding the default, and the miss rates for those branches. As this column shows, the combined heuristic is effective in terms of coverage and miss rate, even for programs with much conditional control flow such as *gcc*, *xlisp*, and *doduc*. The column “+Default” adds in

Program	Heuristics		+Default	All	Loop+Rand
gcc	**79**	32/10	37/11	33/12	43/12
lcc	**80**	30/12	32/12	28/12	42/12
qpt	**76**	17/7	26/9	24/10	42/10
compress	**90**	39/20	40/18	30/16	48/16
xlisp	**90**	25/7	28/7	28/12	42/12
addalg	**90**	42/30	43/30	26/19	26/19
ghostview	**88**	12/4	16/4	12/5	28/5
eqntott	**5**	37/5	50/25	26/13	26/13
rn	**79**	27/1	34/1	20/2	28/2
grep	**68**	1/1	1/0	15/1	16/1
congress	**89**	23/4	28/3	24/9	35/9
espresso	**44**	25/15	26/13	21/13	27/13
awk	**76**	8/2	14/3	7/3	19/3
poly	**89**	40/4	40/3	17/9	15/9
fpppp	**82**	40/7	42/9	41/13	40/13
costScale	**71**	30/22	29/21	22/17	37/17
doduc	**92**	31/3	33/3	21/5	30/5
tomcatv	**100**	1/0	2/0	1/1	20/1
dcg	**73**	11/2	15/4	5/2	12/2
spice2g6	**75**	33/5	36/8	14/8	18/8
sgefat	**96**	25/7	26/8	7/3	13/3
dnasa7	**95**	29/4	32/4	4/1	6/1
matrix300	**100**	33/0	33/0	3/1	4/1

Table 6. Final results. “Heuristics” shows the percent of non-loop branches covered by the heuristics (bold) and the miss rates. “+Default” adds in the predictions for non-loop branches not covered, and “All” adds in predictions for loop branches. For comparison, “Loop+Rand” is the miss rate for loop prediction on loop branches and random prediction on non-loop branches.

	Heuristics		+Default	Target	Random	All	Loop+Rand
MEAN (all)	**79**	26/8	29/10	51/10	49/10	19/8	27/8
Std.Dev. (all)	**20**	12/8	12/8	19/8	13/8	11/6	13/6
MEAN (most)	**82**	27/9	30/9	51/9	50/9	20/9	29/9
Std.Dev. (most)	**12**	10/8	9/7	11/7	8/7	10/5	12/5

Table 7. Means and standard deviations of results from Table 6, for two sets of programs. (all) is for all the benchmarks. (most) excludes the programs *eqntott, grep, tomcatv, and matrix300*. Results for target and random prediction of non-loop branches are included for comparison.

predictions for branches covered by the default prediction. "All" adds in predictions for loop branches, as discussed in Section 3. For comparison, the column "Loop+Rand" shows the miss rate for loop prediction on loop-branches and random prediction on non-loop branches.

Table 7 contains the means and standard deviations of the above results for all benchmarks and for the set of benchmarks excluding *eqntott*, *grep*, *tomcatv*, and *matrix300* (the programs for which over 90% of the non-loop branches are accounted for by a few branch instructions). We also include the results for target and random prediction of non-loop branches (from Table 2), for comparison. On average, our heuristics provide a miss rate of 26% on non-loop branches.

6. INSTRUCTIONS PER MISPREDICTED BRANCH

The previous sections measured the performance of branch prediction by miss rate. Such a metric is useful because most modern architectures exact a performance penalty for mispredicting a branch. However, such a metric does not identify the performance benefit that can be realized when the percent of mispredicted branches decreases. For example, with good branch prediction instruction-level parallel architectures can find more data-independent threads to execute in parallel [5] and compilers can globally schedule code to improve program performance [14].

This section measures the performance of branch prediction based on its ability to find sequences of instructions without a mispredicted branch. Fisher and Freudenberger have proposed a metric [7]: instructions executed per break in control (a break in control is a mispredicted branch instruction, an indirect jump other than procedure return, or an indirect call; correctly predicted branch instructions are not breaks in control). As they argue, the ability of branch prediction to find long sequences of instructions without a break in control depends not just on the branch predictor's miss rate, but also on the density of mispredicted branches in the program's instruction stream.

Fisher and Freudenberger computed instructions per break in control (IPBC) based on execution profiles (*i.e.*, total number of instructions executed / number of breaks in control in the execution). We used instruction traces of program executions to collect more detailed data on IPBC than is available from an execution profile. With instruction traces, we were able to measure the number of instructions executed between each pair of consecutive breaks in control. This information is simply not available from an execution profile. Our data shows that the profile-based IPBC average underestimates the length of available sequences and fails to accurately distinguish different branch prediction strategies.

We collected instruction traces for the following benchmarks: *compress, gcc, lcc, qpt, xlisp, doduc, fpppp,* and *spice2g6*. For the most part, we chose benchmarks that contain complex control flow and are hard to predict. The instruction traces were generated by the same datasets as in the previous sections. We used three branch predictors in this experiment:

- the perfect predictor (*Perfect*);
- loop prediction on loop branches and the ordering *Point*→*Call*→*Opcode*→*Return*→*Store*→*Loop*→*Guard* on non-loop branches (*Heuristic*).

- loop prediction on loop branches and random prediction on non-loop branches (*Loop+Rand*).

Each branch predictor defines a set of breaks in control in a program's execution. Each break in control *B* defines a *sequence* of instructions from (but not including) the break in control preceding *B* up to and including *B*. These sequences partition the instruction trace of an execution. For each predictor, we recorded the following information: $\forall\ j,\ 0 \le j \le 999$, the number of sequences whose length is in the interval $(10j,\ 10j+9)$. The last bucket ($j = 999$) records all sequences of length greater than or equal to 9990. For each bucket, we also recorded the sum of the length of the sequences associated with that bucket.

We graph the distribution of sequence lengths by plotting sequence length (x) versus the percentage of the execution accounted for by sequences of length less than x. The graphs (4-12) can be found in the Appendix. Each branch predictor contributes a plot to each graph. The slower the growth rate of a plot, the better. For each predictor, the graph also shows the miss rate (for all branches) and the IPBC average. In many cases, the *Heuristic* plot is closer to *Loop+Rand* than to *Perfect* because very high accuracy is necessary to obtain long sequences, especially in programs in which there is conditional control flow in most loops and the basic block size is small (*i.e., compress, gcc, lcc, qpt, xlisp* and *doduc*). In these programs non-loop branches are distributed fairly regularly over the entire execution, so the miss rate (on non-loop branches) must be very low to get long sequences. A very simple model captures this behavior. The model assumes that every basic block is of unit length and ends with a conditional branch, branches are independent, and every branch has a miss rate of m. If s is a sequence length ($s \ge 1$) then the function

$$f(m,s) \ = \ m \sum_{i=0}^{s-1} (1-m)^i \ = \ 1-(1-m)^s$$

represents the percentage of the execution accounted for by sequences of length less than or equal to s. Graph 13 shows plots of this function for miss rates between 2.5% and 30% by increments of 2.5%. The payoff in sequence length comes not from moving from 30% to 15%, but from reducing the miss rate to less than 15%. Similar behavior can be found in *gcc* (Graph 7), *lcc* (Graph 8), and *qpt* (Graph 9). While the growth rates of the plots and accompanying absolute miss rates are different than the model (due to the simplifying assumptions of the model), the relationship between miss rate and plot roughly follows that of the model.

We now turn our attention to the IPBC average, the profile-based metric defined previously. Because the IPBC average evenly distributes mispredicted branches over the entire execution, it tends to underestimate the available sequence length. This can lead to underestimation or overestimation of the difference between predictors, depending on the control flow complexity of the program.

We use the benchmark *spice2g6* to illustrate some of these points. As Graph 4 shows, the IPBC average for the perfect predictor is 183 (instructions per break in control). However, for the perfect predictor, sequences of length less than or equal to 183 account for approximately 30% of the execution. If we examine Graph 5, the reason for this disparity becomes clear. This graph shows sequence length (x) versus the percentage of breaks in control accounted for by sequences of length less than x. Sequences of length less than or equal to 183 account for approximately 80% of the breaks in control. Because the profile-based IPBC average distributes the breaks in control evenly over the entire execution, the highly skewed distribution of sequence lengths causes the average to underestimate the available sequence length. This skew occurred (to varying degrees) among all the benchmarks. For many of the benchmarks we examined, it was more informative to look at the sequence length at which 50% of the execution was accounted for (we refer to this length as the *dividing length*). For the perfect predictor for *spice2g6*, the dividing length is approximately 800 instructions.

When a high percentage of the sequences (*i.e.*, breaks in control) account for a low percentage of the execution, substantial differences in the IPBC average may not accurately reflect the differences between predictors. In the case of *spice2g6*, the IPBC average overestimates the difference between the perfect predictor and other predictors. As Graph 4 shows, The IPBC averages are 87, 108, and 183 while the dividing lengths are approximately 550, 650, and 800 instructions for the *Loop+Rand*, *Heuristic* and *Perfect* predictors, respectively. The reason for this disparity has to do with the control flow complexity of the benchmark. In *spice2g6* predicting loop branches gives a big payoff. Correctly predicting non-loop branches is not crucial to finding sequences of long length. However, differences in miss rates for non-loop branches can have a large impact on the IPBC average because the average distributes mispredicted branches evenly over the entire execution.

The IPBC average can also *underestimate* the difference between predictors. This is especially true for benchmarks in which there is conditional control flow inside most loops. In these cases, the non-loop branches are truly more evenly distributed in the execution and the IPBC average tends to underestimate the dividing length for the perfect predictor. For example, in the *lcc* benchmark, the perfect predictor has an IPBC average of 58 and a dividing length of 100, while the *Heuristic* predictor has an IPBC average of 28 and a dividing length of 40.

To summarize the results of this section:

- High accuracy on non-loop branches is crucial to getting long sequences for programs with conditional control flow in loops.
- Instruction traces provide a much more accurate view than execution profiles of the impact of branch prediction on instructions executed per mispredicted branch. The skewed distribution of sequence lengths causes the profile-based IPBC average to underestimate the length of available sequences. Depending on the control flow complexity of a program, this may cause the IPBC average to overestimate or underestimate the difference between predictors.

7. OTHER DATASETS

If a program-based predictor is to be useful it must have good performance over different executions of the same program, as well as over executions of different programs. We ran a number of the benchmarks on different datasets to examine how well the *Heuristic* predictor performs. Graph 14 presents the results. For each benchmark and dataset, the graph shows the miss rate for the perfect predictor and for the *Heuristic* predictor. We emphasize that the heuristic predictor makes the same predictions no matter which dataset is used, while the perfect predictor is dataset dependent, making the best possible static prediction per dataset.

For many of the benchmarks, such as *gcc, lcc, qpt, compress, xlisp, ghostview, grep, espresso, costScale*, and *doduc*, the miss rates do not vary too widely. The consistent results for benchmarks with large amounts of conditional control flow is encouraging. It is often the case that a difference in miss rates for the heuristic predictor is accompanied by a similar difference in the miss rates for the perfect predictor. For example, in the first two datasets for *spice2g6* the miss rate for both the heuristic and perfect predictor approximately double.

8. RELATED WORK

Related work on static branch prediction falls into two categories: profile-based and program-based. McFarling and Hennessy reported that profile-based static prediction yields results comparable to dynamic hardware-based methods [11]. As mentioned before, Fisher and Freudenberger examined profile-based static prediction in detail, showing that most branches behave similarly over different executions of the same program and that profiles can be used to effectively predict branch directions in other executions [7].

J. E. Smith discusses several static prediction strategies based on instruction opcodes, applied to six FORTRAN programs with success [16]. Program-based static prediction was used by Bandyopadhyay, et. al. in a C compiler for the CRISP microprocessor [3]. They identified loop tests as those in the boolean expression associated with a loop construct. Branch prediction for tests associated with *if* statements was accomplished by a table lookup based on the comparison operator and operand types. The authors reported an extremely high success rate but gave no numbers or details on this table lookup strategy. Wall used program-based heuristics to estimate various program profiles (the number of times a particular program component executes) rather than to predict individual branches [17]. He reported poor results for his estimator, compared to a randomly generated profile.

Lee and A. J. Smith's paper on branch prediction strategies reported that for the workloads they considered (IBM 370, DEC PDP-11, and CDC 6400) branches were taken twice as often as they fell through [10]. Lee and Smith considered branch prediction based on instruction opcodes and dynamic branch history. They found that the miss rates for opcode prediction ranged from 20.2% to 44.8% with an average of 30.1%.

9. CONCLUSIONS

We have presented a simple set of program-based heuristics for statically predicting branches and combined these into a single branch prediction heuristic that performs well for a large and diverse set of programs. In addition to using natural loop analysis to predict branches that control the iteration of loops, we focus on heuristics for predicting non-loop branches, which dominate the dynamic branch count in many programs. These heuristics are local in nature, requiring little program analysis, yet are effective in terms of coverage and miss rate. We believe that many of these heuristics could be generalized and refined with information available in a compiler to produce even better results.

ACKNOWLEDGEMENTS

We would like to thank Susan Horwitz for her support of this work. Thanks to Paul Adams for providing the *congress* benchmark, to Stefan Freudenberger for his extra SPEC datasets, and to Steve Kurlander for help with some benchmarks. Josh Fisher, Mark Hill, and Guri Sohi provided helpful comments on earlier drafts of this work. Thanks also to Todd Austin for his input.

REFERENCES

1. A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA (1986).

2. T. Ball and J. R. Larus, “Optimally Profiling and Tracing Programs,” *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages,* (Albuquerque, NM, January 19-22, 1992), pp. 59-70 ACM, (1992).

3. S. Bandyopadhyay, V. S. Begwani, and R. B. Murray, “Compiling for the CRISP Microprocessor,” *Spring Compcon 87*, pp. 96-100 IEEE Computer Society, (February 1987).

4. Systems Performance Evaluation Cooperative, *SPEC Newsletter (K. Mendoza, editor)* **1**(1)(1989).

5. J. A. Fisher, “Trace Scheduling: A Technique for Global Microcode Compaction,” *IEEE Transactions on Computers* **C-30**(7) pp. 478-490 (July 1981).

6. J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau, “Parallel Processing: A Smart Compiler and a Dumb Machine,” *Proc. of the ACM SIGPLAN 1984 Symposium on Compiler Construction (SIGPLAN Notices)* **19**(6) pp. 37-47 (June 1984).

7. J. A. Fisher and S. M. Freudenberger, “Predicting Conditional Branch Directions From Previous Runs of a Program,” *Proceedings of the 5th International Conference on Architectural Support for Programmming Languages and Operating Systems (ACM SIGPLAN Notices)* **27**(9) pp. 85-95 (October 1992).

8. S. L. Graham, P. B. Kessler, and M. K. McKusick, “An Execution Profiler for Modular Programs,” *Software—Practice and Experience* **13** pp. 671-685 (1983).

9. G. Kane and J. Heinrich, *MIPS RISC Architecture,* Prentice Hall (1992).

10. J. K. F. Lee and A. J. Smith, “Branch Prediction Strategies and Branch Target Buffer Design,” *Computer* **17**(1) pp. 6 - 22 (January 1984).

11. S. McFarling and J. Hennessy, “Reducing the Cost of Branches,” *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 396-403 ACM and IEEE Computer Society, (June 1986).

12. W. G. Morris, “CCG: A Prototype Coagulating Code Generator,” *Proceedings of the SIGPLAN 91 Conference on Programming Language Design and Implementation,* (Toronto, June 26-28, 1991), *ACM SIGPLAN Notices* **26**(6) pp. 45-58 (June, 1991).

13. D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach,* Morgan, Kaufmann Publishers Inc. (1990).

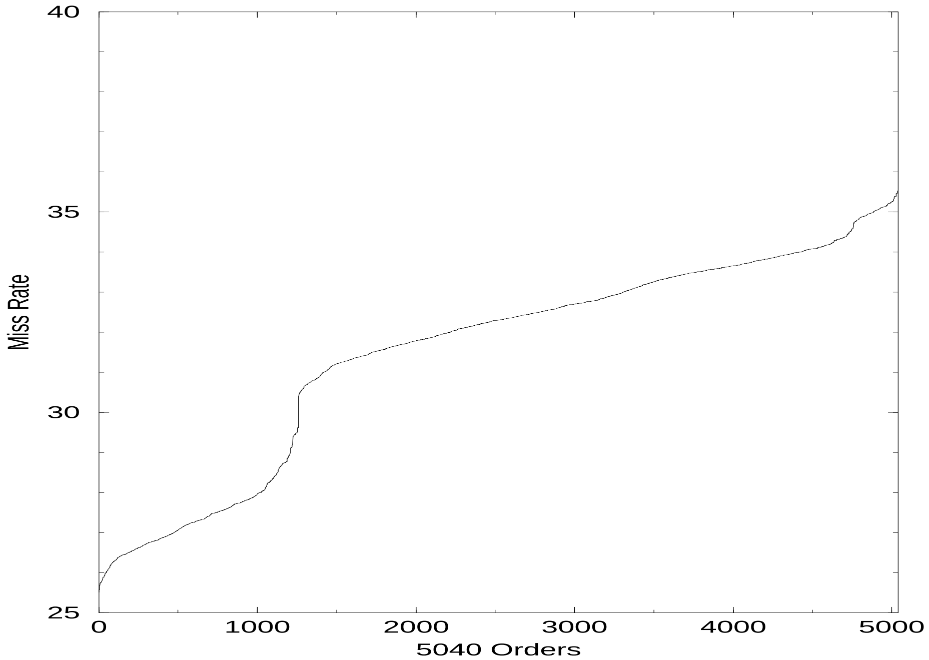
14. K. Pettis and R. C. Hanson, “Profile Guided Code Positioning,” *Proceedings of the ACM SIGPLAN ’90 Conference on Programming Language Design and Implementation (SIGPLAN Notices)* **25**(6) pp. 16-27 ACM, (June, 1990).

15. R. L. Sites, *Alpha Architecture Reference Manual,* Digital Press, Burlington, MA (1992).

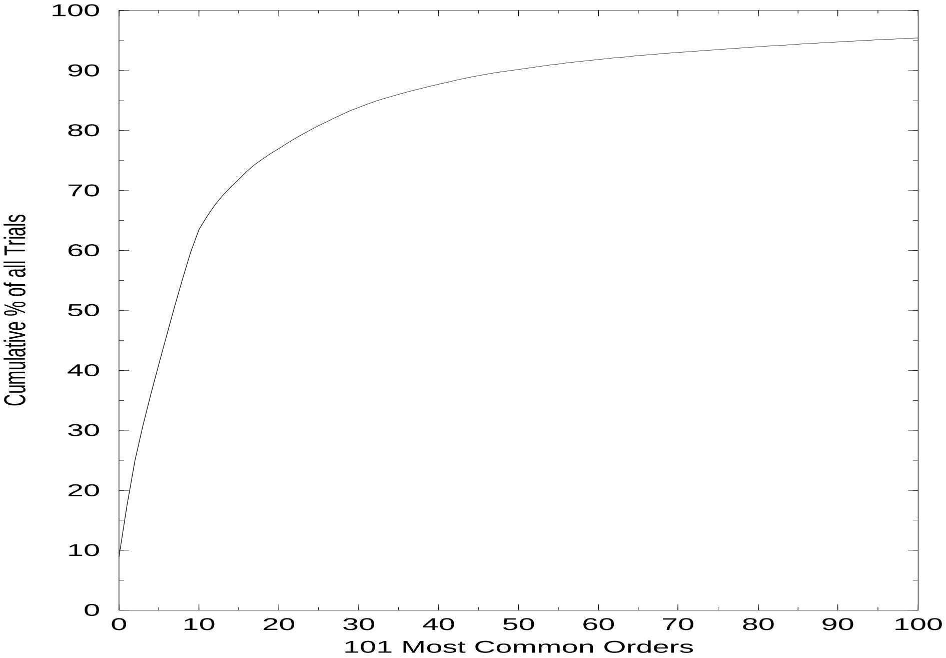
16. J. E. Smith, “A Study of Branch Prediction Strategies,” *Proceedings of the 4th Annual International Symposium on Computer Architecture (SIGARCH Newsletter)* **9**(3) pp. 135-148 ACM and IEEE Computer Society, (May 1981).

17. D. W. Wall, “Predicting Program Behavior Using Real or Estimated Profiles,” *Proceedings of the SIGPLAN 91 Conference on Programming Language Design and Implementation,* (Toronto, June 26-28, 1991), *ACM SIGPLAN Notices* **26**(6) pp. 59-70 (June, 1991).

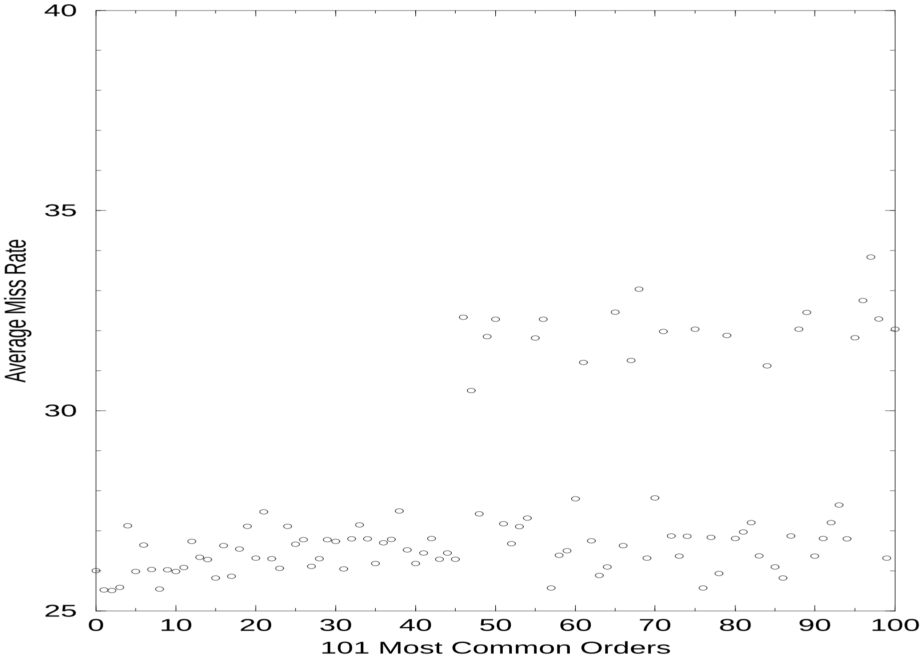
APPENDIX - Graphs



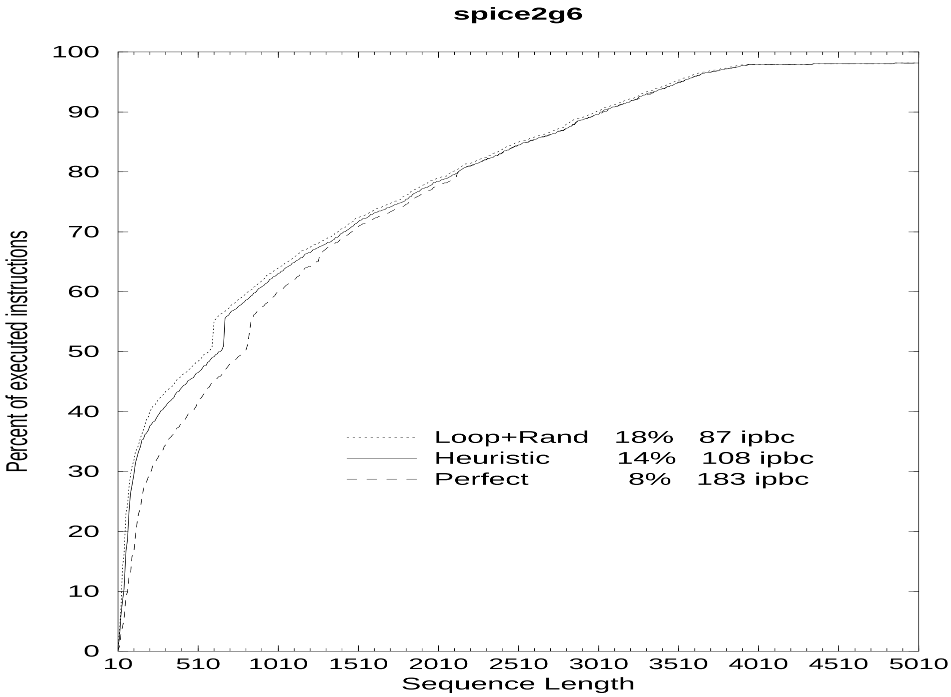
Graph 1. Average miss rates for each of the 7! = 5040 possible orderings, sorted by miss rate.



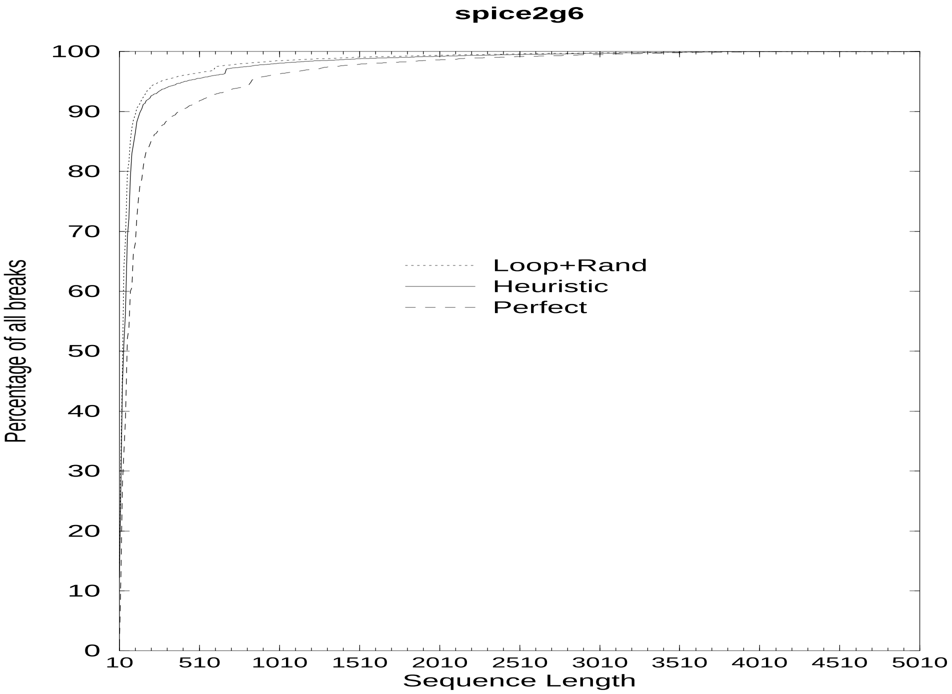
Graph 2. The most common 101 orders from the $\binom{22}{11}$ experiment and their cumulative distribution in the trials.



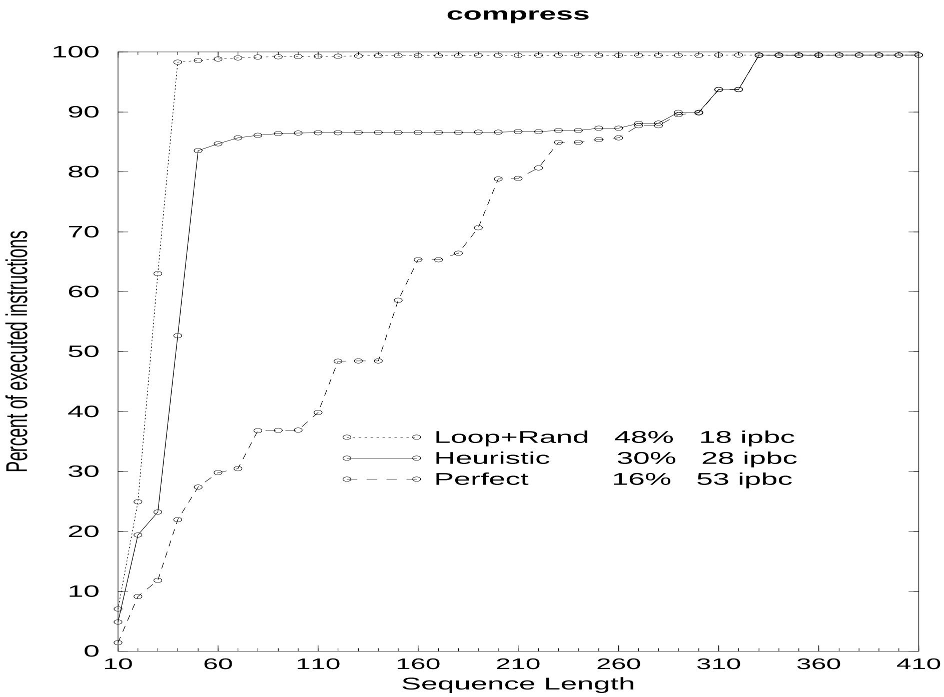
Graph 3. Average miss rates (all 22 benchmarks) for the most common 101 orders from the $\binom{22}{11}$ experiment.



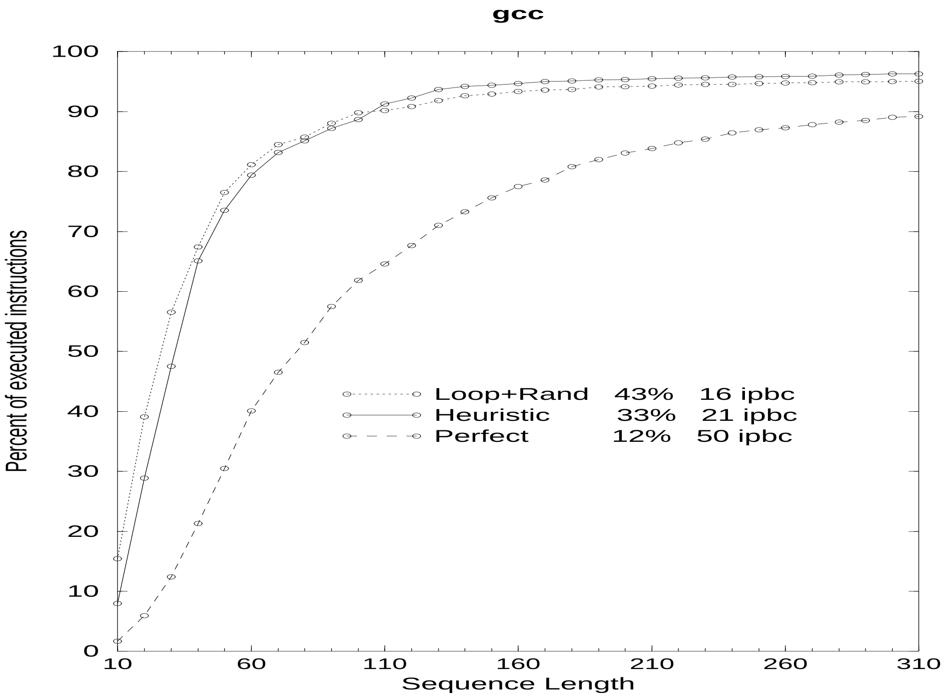
Graph 4. *spice2g6*: cumulative distribution of sequence lengths.



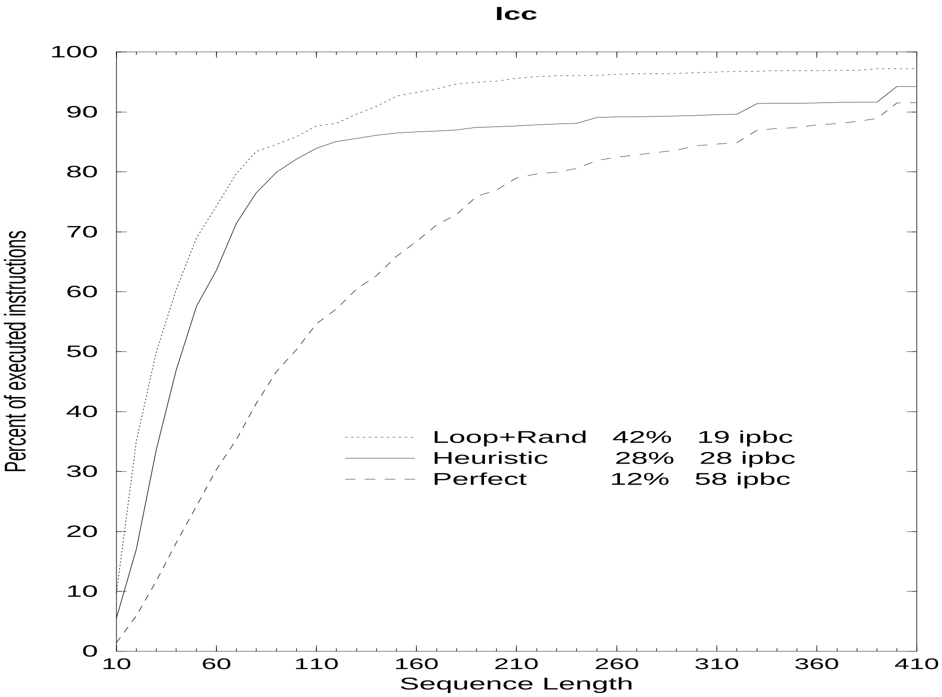
Graph 5. *spice2g6*: cumulative distribution of breaks.



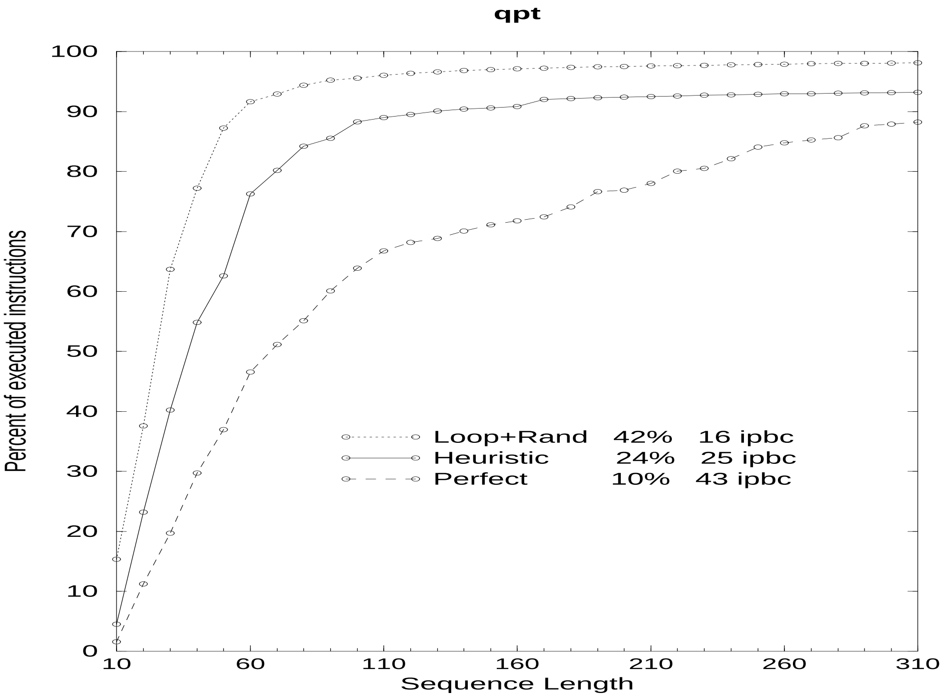
Graph 6. *compress*: cumulative distribution of sequence lengths.



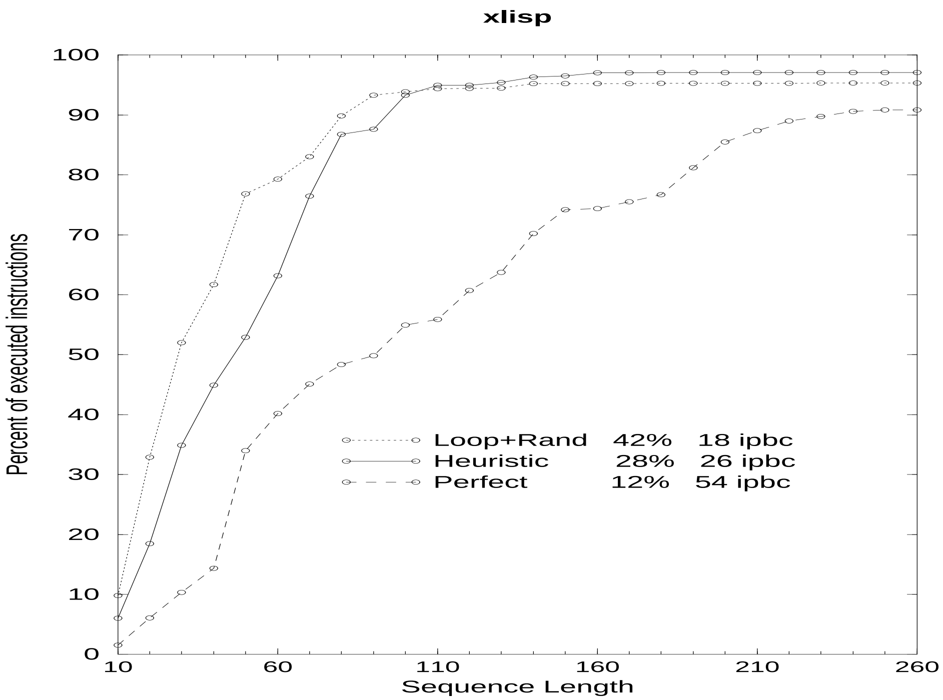
Graph 7. *gcc*: cumulative distribution of sequence lengths.



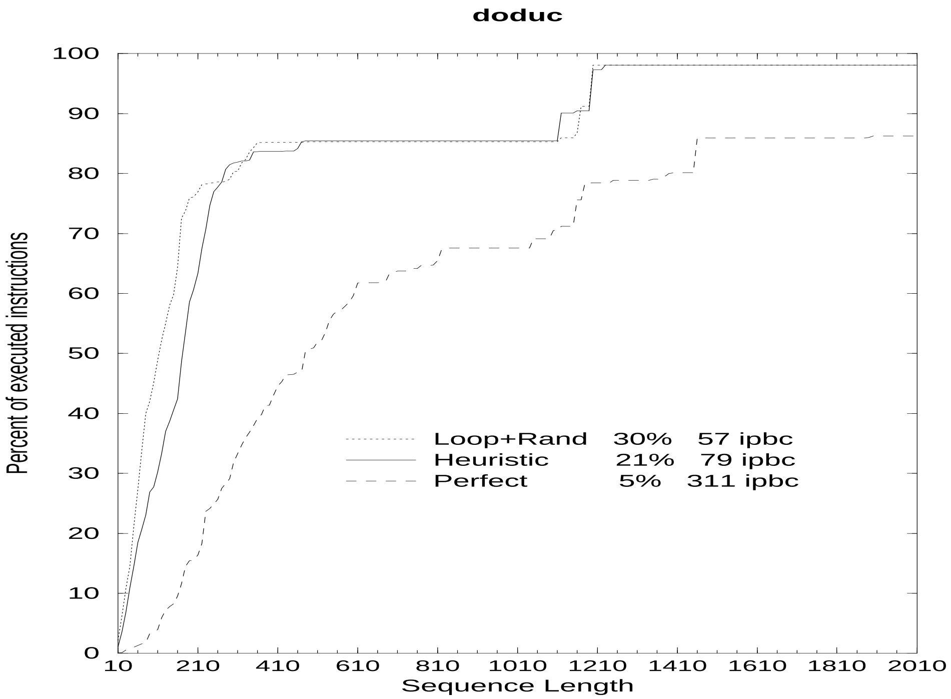
Graph 8. *lcc*: cumulative distribution of sequence lengths.



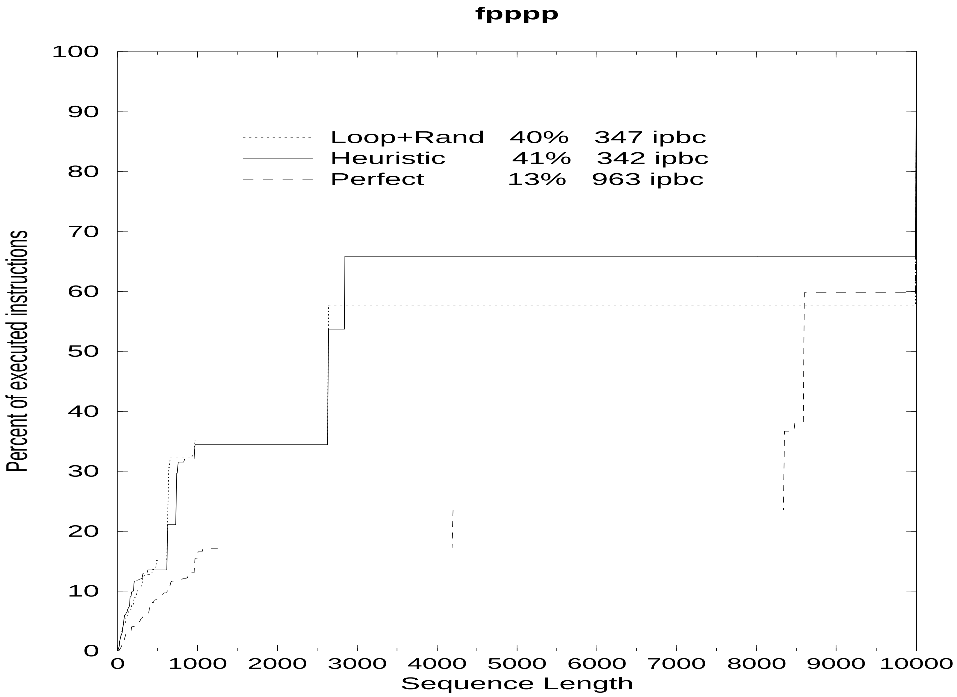
Graph 9. *qpt*: cumulative distribution of sequence lengths.



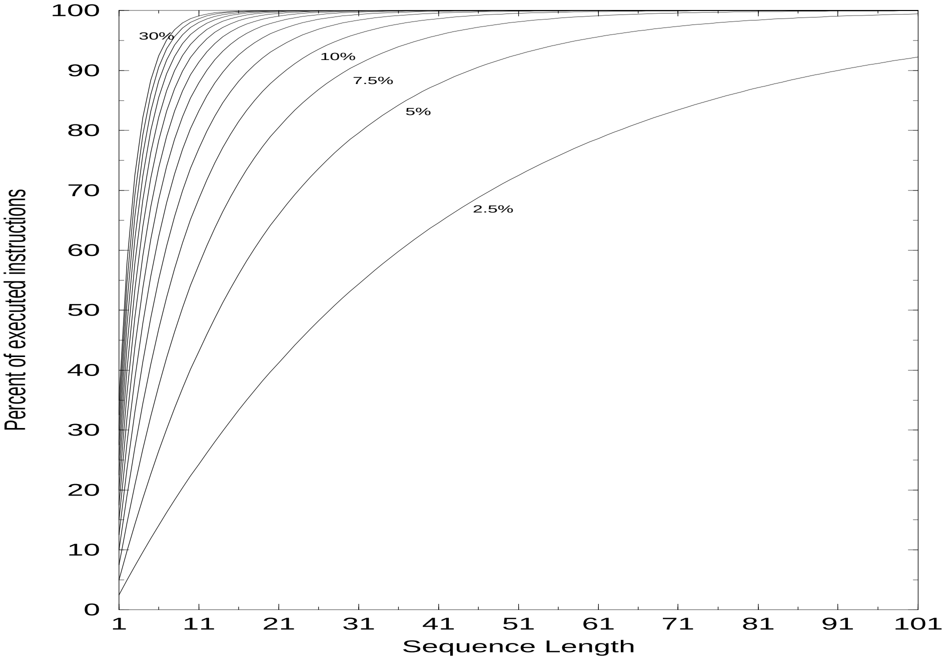
Graph 10. *xlisp*: cumulative distribution of sequence lengths.



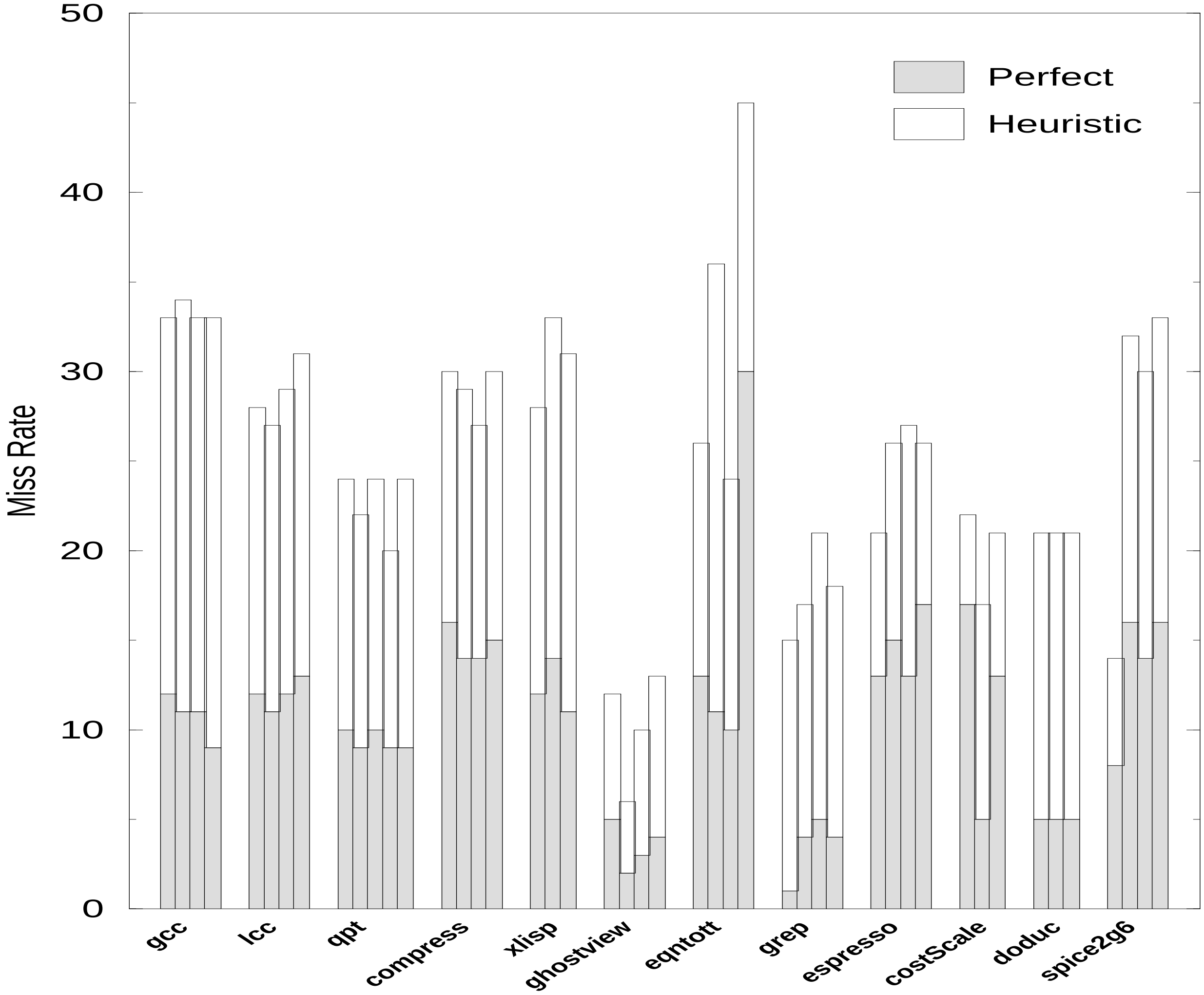
Graph 11. *doduc*: cumulative distribution of sequence lengths.



Graph 12. *fpppp*: cumulative distribution of sequence lengths.



Graph 13. A simple model for cumulative distribution of sequence lengths. The function $y = 1-(1-m)^x$ is plotted for miss rates (m) between 0.025 and 0.3 by increments of 0.025.



Graph 14. Miss rates for different runs of various benchmarks.