Predicting Conditional Branch Directions From Previous Runs of a Program

Joseph A. Fisher and Stefan M. Freudenberger

Hewlett-Packard Laboratories
1501 Page Mill Rd. 3U-5
Palo Alto, CA 94304
jfisher@hpl.hp.com freuden@hpl.hp.com

Abstract

There are several important reasons for predicting which way the flow of control of a program is going to go: first, in instruction-level parallel architectures, code motions can produce more data-ready candidate instructions at once than there are resources to execute them. Some of these are speculative (executed ahead of a conditional branch that might otherwise have prevented their execution), so one must sensibly pick among them, and one must avoid issuing low probability speculative instructions when the system overhead associated with canceling them most of the time outweighs the gain of their infrequent success; second, important classes of compiler optimizations depend upon this information; and finally, branch prediction can help optimize pipelined fetch and execute, icache fill, etc. If substantial code motions are desired, it is probably impractical to expect the hardware to make them, and a compiler must instead. Thus, the compiler must have access to branch predictions made before the program runs. In this paper we consider the question of how predictable branches are when previous runs of a program are used to feed back information to the compiler. We propose new measures which we believe more clearly capture the predictability of branches in programs. We find that even code with a complex flow of control, including systems utilities and language processors written in C, are dominated by branches which go in one way, and that this direction usually varies little when one changes the data used as the predictor and target.

1 Introduction

Why Predict Conditional Branch Directions?

There are several important classes of reasons for attempting to predict which direction a conditional branch will go in before it is executed:

> *Speculative execution to enhance instruction-level parallelism.* Machines that do **speculative execution** start the execute phase of instructions before it is certain that they should be executed, possibly allowing their execution to use otherwise idle machine resources. Predictions can allow us to execute speculatively the instructions that are most likely to be profitable.
>
> *Compiler optimizations.* Important classes of compiler optimizations rely on dynamic information to decide among alternative code motions, transformations, etc.
>
> *Hardware reasons.* CPU fetch/execute pipelines can fetch and start to decode an instruction before it is certain that a conditional branch it follows will go in that direction. If a branch goes in the expected direction, a pipeline bubble can be avoided. In addition, elements in the memory hierarchy, such as instruction caches, can be instructed to prepare for the coming instruction stream, lowering the latency of memory instructions.

Here, we are primarily interested in **instruction-level parallelism** (or **ILP**), the parallelism found by overlapping the execute phase of several machine level instructions within a single CPU. Experiments (see, for example, [Wall 91a]) have indicated that the following are true:

> To get a lot of instruction-level parallelism, many instructions must be moved up past conditional branches that they followed in the source.
>
> If one blindly executes all instructions that are data-ready at all times, an enormous amount of hardware will be required, and most of it will be wasted doing instructions that were not in the path of the flow of control that eventually occurred.

Thus we must have some way of picking only the highest probability instructions to execute in a given cycle, and thus we must predict which way the branches in the program are likely to go. In addition, there is a cost associated with tracking speculative instructions that caused faults, since one does not know whether or not these faults should be serviced until the flow of control is resolved. One must also deal with the possibility of degraded performance due to page and cache misses caused by unnecessary speculative instructions. This is an active research area (see, for example, [Rogers and Li 92] and [Mahlke et al. 92], both in this volume). We must therefore have some method of gauging which speculative instructions are worth the trouble, and which aren't.

Are Branches Predictable?

In describing ILP code generation, the authors often say that conditional branch directions are very predictable from run to run. Some people accept that as obvious, others give strongly negative responses, saying it is true in vector-type codes, but absolutely not in systems, commercial, etc. codes; or they say it's true for FORTRAN programs, but false for C; or it's false for pointer oriented codes. Since we believe that the practicality of instruction-level parallelism in large quantities depends upon this predictability, the fact that there is such a difference of opinion is significant. These experiments were done to nail down this question by running a wide variety of programs, each over several different datasets, to find just how justifiable this claim is.

Static vs. Dynamic Branch Prediction

If one predicts conditional branch directions while a program is running, one is said to be doing **dynamic branch prediction.** If, instead, one tries to predict branch directions before the program runs, one is doing **static branch prediction.**

There has been a great deal of interest in dynamic branch prediction for the hardware benefits listed above. Dynamic methods usually involve attaching 1 or 2 bits to each branch and setting or incrementing those bits, as the program runs, to reflect the direction the branch most recently went in.

Static methods, by contrast, attach one direction to each conditional branch at compile time. The branch is then always predicted to go in that direction. This difference between static and dynamic branch predictions is typical of the many static/dynamic tradeoffs one finds in system design: static methods usually require little or no hardware and allow time to compute whatever quantities are required. But dynamic methods can use the added information present while a program is running, can adapt to changing conditions while the program runs, require less work at compile-time, and allow a greater degree of object code compatibility since changes are made after the object code is presented to the hardware.

In this paper, our interest is in static branch prediction. In particular, we are trying to find out whether one will get acceptable results using the branch behavior of the program during previous runs to predict branch directions for subsequent runs.

Using Static Conditional Branch Predictions to Increase Usable ILP

We believe that in both VLIWs and ambitious superscalars, compilers can benefit from a good job of static branch prediction.

VLIW systems. In a VLIW [Fisher 83] [Colwell et al. 87], or in any other machine in which compilers must arrange instructions into data-independent groups, the compiler must look at a large group of instructions in order to use the machine's resources well. (The group of instructions the compiler considers at once while scheduling is often called the **candidate set**. This is analogous to the instruction window of superscalar systems. Since the compiler is bundling several instructions into one long instruction at compile time, practitioners often avoid confusion by using the term **operation** for single RISC-level instructions, and **instruction** for the long instruction produced from several operations. In this paper, however, we refer to the smaller, individual operations as instructions.) The candidate set usually contains many potentially speculative instructions, so the compiler must be able to use branch prediction to schedule high probability and reject low probability speculative instructions. Techniques like software pipelining [Rau and Glaeser 81] [Lam 88] concentrate on generating code for tight inner loops. In those cases, the only required branch prediction is implicit in the assumption that loops are repeated. For code other than tight loops, code generation techniques like trace scheduling [Fisher 81] [Ellis 85], or others that deal with flow of control more general than tight loops, must rely on branch predictions to select candidate instructions. Obviously, since this work is being done by the compiler, static branch prediction must be used.

Superscalar systems. A superscalar [Johnson 91] does its instruction scheduling at run time, and thus may use dynamic branch prediction to pick which instructions to consider next. But what will happen when a superscalar is built which attempts to find the large quantity of ILP that VLIWs have attempted to find? The single worst limiting factor in such a machine is likely to be the fetch/decode/issue hardware (rather than the execute hardware), and especially the hardware that must determine whether instructions are data-independent; it must do this as the program is running. This hardware will grow approximately as the square of the number of instructions it must consider, since it must consider them pairwise. If instructions are considered in their natural source order, as presented by an ordinary optimizing compiler, only a small percentage will be data-independent of all prior instructions. Thus the hardware must consider many instructions to find enough that are data-independent, and is likely to grow unacceptably large. The natural solution to this problem is for the compiler to rearrange the code in such a way that instructions which are data-independent are much nearer to each other than they were in the original source. To carry out all of these code motions, the compiler must, as in the case of VLIWs, choose which instructions to move up past conditional branches and which not to, and again must rely upon static branch predictions.

Whether there exists enough ILP in ordinary systems and commercial programs to make this a significant factor for superscalars is a controversial question.

Three Methods of Static Branch Prediction

There are three ways we know of to predict which way branches will go before the program runs:

1. The programmer inserts directives.
2. The compiler examines the source and uses heuristics (which might have any degree of sophistication).
3. The program is run, statistics are gathered and fed back into the source code, and the program is recompiled using those statistics.

The trace scheduling compiler developed at Multiflow Computer (and used in the experiments reported upon here) offers all of these facilities:

1. The compiler understands directives, for example:

```
C!MF! IFPROB (32543, 20, 0)
```

This particular directive would be attached to a FORTRAN arithmetic if. The compiler is being told that when the program runs, a good guess is that it will go to the first branch target 32543 times, and so on.

2. The compiler by default uses very naive heuristics (the only ones it has).
3. The compiler has a tool called the IFPROBBER, which is central to the experiments reported on here and is described in more detail below. The IFPROBBER, invoked by a compiler switch, instruments the code with instruction counters before each conditional branch. Whenever the program runs, a database of branch counts is augmented. Later, a call to a utility feeds the branch counts back into the source in the form of the above directives.

2 This Experiment

General Methodology

In this experiment, we collected several different programs, and several different datasets for each, where practical. We then ran each program with each of its datasets, collecting a record of which way each branch went, and how often. We used these counts as predictors, one per dataset, and measured how well they performed predicting the other datasets. We then combined the results of runs to form new predictors. Sometimes we used the run we were trying to predict as its own predictor. This gave us an upper bound on how well another predictor can do predicting that dataset, since each branch is predicted to go in what will turn out to be the majority direction.

Previous Experiments

Several other experiments have been done to measure the effectiveness of branch prediction, especially dynamic branch prediction, since the majority of work has been done by CPU designers wanting to minimize pipeline flushing. In general, these experiments measured the effectiveness of hardware schemes, and found that simple schemes predicted correctly 80-90% of the branches in systems codes, and 95-100% of the branches in scientific FORTRAN [Smith 81] [Lee and Smith 84] [McFarling and Hennessy 86]. [Ponder and Shebanow 90] consider how well one could possibly do dynamically from an information-theoretic viewpoint.

Static branch prediction is sometimes mentioned in these experiments. [McFarling and Hennessy 86] reported that profiling allowed them to get prediction performance on 4,000 lines of Pascal to a degree comparable to much more expensive hardware schemes, and that when they changed datasets, 98% of the advantage of prediction was preserved, but no further details were given. Static prediction by heuristics has been tried: [Smith 81] reports poor results using a variety of opcode and branch direction heuristics. [Bandyopadhyay, et al. 87] reported tantalizing results: they said that heuristics that took into consideration data types and other source-level information were as effective as hardware prediction methods on systems code, but the detailed results were never published and are evidently unavailable.

Instead of considering what percentage of the branch predictions are correct, some experimenters (for example, [Wall 91b]) have considered the effect of prediction on the quantities that prediction is supposed to improve. [Conte and Hwu 92] [Chang, Mahlke and Hwu 92] report that trace selection and other optimizations are greatly improved by feedback methods.

Why Percent of Branches Correctly Predicted Is the Wrong Measure

When starting out, it was our intention to measure the traditional quantity that is reported is all of the above experiments: *percent conditional branches correctly predicted*. We soon realized that this was inappropriate. For example, two of the programs we used in our collection were *fpppp* and *li*, both programs in the SPEC suite. *fpppp* is a floating-point intensive chemistry application, its inner loop is a giant expression with no flow of control. *li* is a lisp interpreter, it is constantly looking at lisp instructions and deciding what to do. It seems obvious that *fpppp* has a more predictable flow of control for ILP purposes, but when we measured, we found that the branches in *fpppp* went in their more likely direction 83% of the time, while for *li* the same measure was 85%. This difference, which is small and in the wrong direction, did not capture the essence of predictability.

Obviously what is wrong is that this simple measure does not take into consideration the density of branches. In this case, *li* executes a conditional branch about every 10 instructions, *fpppp* does one about every 170 instructions. This difference dwarfs the small difference in per-branch predictability. Upon reflection, it becomes clear that a more appropriate measure is **instructions per mispredicted branch**, that is, how many instructions, including correctly predicted branches, one passes on average before encountering a mispredicted branch.

Instructions Per Mispredicted Branch

Instructions per mispredicted branch is a persuasive measure: it is as if a correctly predicted branch "goes away," that is, its condition is computed, but instructions move past it without break. The relevant question is the density of mispredicted branches. Those can be a barrier to ILP, and are an intrinsic property dependent only upon a program, predictor dataset, and target dataset. Even when one is doing branch prediction for hardware reasons, this seems like the right measure, though it never seems to be used in that context.

Once one begins to think in terms of instructions per mispredicted branch, it becomes evident that other forms of breaks of control are also relevant. Thus the thrust of what is being measured changes from the predictability of branches to a more general question. Given a program and a target dataset, we started out to ask: if one uses a predictor dataset, what percentage of the branches will be correctly predicted. Now we ask:

> Using a predictor dataset to anticipate the behavior of a program, how many instructions will we pass, on average, before either a mispredicted branch, or some other break in control, causes a barrier to instruction-level parallelism.

Other Breaks in Control

As described below, our sample base includes programs written in C and FORTRAN. In those languages there are other instructions, besides conditional branches, that can cause a transfer of control. These can be classified as:

1. Indirect calls and returns
   Assigned GOTOs
2. Direct calls and returns
   Jumps
   Switch statements
   Computed GOTOs
   Arithmetic IFs

Group 1 consists of instructions we will refer to as **unavoidable breaks in control**. These will almost always cause a break; there are few compiler or hardware tricks that could allow instruction-level parallelism to advance past them. (Though with a fair amount of work a compiler might identify all of the potential targets of an assigned GOTO and possibly produce a very predictable set of conditional branches. The compiler we used doesn't do this, and, in any case, none of the FORTRAN programs in our sample had assigned GOTOs.)

Group 2 are those we call **avoidable breaks in control**. These can be further broken down as follows:

Calls and returns. A compiler that is going to find large amounts of ILP must be able to inline the most commonly called procedures[1]. An executed call that is not inlined will cost two breaks in control—a deadly effect when a short routine is called in an inner loop. Below we show the instructions per break in control with calls and returns left in and with them ignored. The differences in our sample set are reasonably small.

Jumps. Compilers can eliminate many of these unconditional breaks in control by rearranging the static position of the code, and a compiler attempting to get substantial amounts of ILP should do that. Some jumps, however, represent rejoins in the code. Rejoins are an important topic for ILP compilers: a compiler would be most effective if it could totally unwind all rejoins and compile every possible path through the code. Given that that is impossible, many techniques have been embodied into code generation techniques to get as much of that benefit as possible without having an arbitrary barrier for ILP. It is our belief that a good ILP compiler will eliminate almost all of the negative effect of jumps, and the results presented below make that assumption.

Multiple destination branches. The other "avoidable" branches are those with multiple destinations. In this experiment, our compiler turns these into a set of linear or cascaded conditional branches. We believe this captures the information needed: if after that conversion the branch usually goes in one very predictable direction, then conditional branches are probably what the compiler should generate anyway, once it gets this feedback. If the branch is less predictable, then this will be reflected as unpredictable breaks in control. In this case the set of conditional branches that we generate may contribute more than one break in control most of the time—and then the numbers we are reporting overstate the number of breaks in control (since a compiler could use whatever single hardware mechanism closely resembles the source branch, probably causing only one break in control). Optimizing compilers often have a technique for deciding when to generate a "branch target table," and when to generate instead a series of conditional branches. We believe that a compiler for ILP with access to good branch predictions should be augmented to use a technique that mirrors the above argument.

[1]The Multiflow compiler used some simple heuristics to do this automatically when a compiler switch was set. Source control systems must account for this inlining, in that a file can change indirectly when an inlined routine changes.

Methods and Tools

Instruction counts. These experiments were done on a Multiflow Trace 14/300, a CPU with 512-bit VLIW-instructions, each composed of up to 14 independent RISC instructions (or operations, in VLIW terminology). When we counted instructions per mispredicted branch, these RISC instructions were what we counted.

The Trace has typical RISC-like instructions: fixed 32-bit length, fixed-format, three register operations with memory accessed only through explicit loads and stores. Memory can only be accessed in 32- and 64-bit quantities, so 8- and 16-bit memory accesses must extract and merge data using explicit operations. To factor out the VLIW-ness of our results, we prevented our instruction scheduler from executing operations speculatively, and we subtracted counts for data motions between the multiple register banks of the Trace. Our experience with the Trace has led us to believe that measures of its RISC instructions are generally comparable to RISC instructions for more popular CPUs [Freudenberger and Ruttenberg 92], and we found that our measures of correctly predicted branches were in line with measures reported in the literature on similar programs.

The compiler used was the highly optimizing, trace scheduling compiler shipped by Multiflow, modified to suit these and other experiments. In doing these experiments, we allowed most of the typical classical intraprocedural optimizations (common subexpression elimination, copy propagation, strength reduction, induction variable simplification, loop-invariant code motion, promotion of scalar variables to registers, renaming of scalar variables, etc.) but suppressed some more advanced optimizations that would have changed the flow of control, such as loop unrolling and if-conversion[2].

IFPROBBER. Each program in our dataset was compiled twice, resulting in one image that would collect branch prediction information and one that would collect instruction counts. For the first compilation, we set a compiler switch which invoked a tool called the IFPROBBER, causing the compiler to generate code containing counters before each branch. After each run, these counters contained a record of how many times the branch was encountered and how often the associated condition was true. Upon the completion of each run, the generated code collected the value of each counter and added that value to the amount that had been accumulated in a database for that counter during previous runs. Finally, an associated utility could read the database of accumulated counter values and insert these

[2] The Trace compiler front ends convert some simple if statements into a special select instruction that evaluates both operands and selects one of them depending on a tested condition. We didn't turn this off, and so supressed a few branches that would otherwise have been generated. This was a very small effect, since selects were typically less than 0.2% (sometimes up to 0.3%, and in one case 0.7%) of all instructions executed.

values back into the source code in the form of compiler directives. Although the actual workings are more oriented towards the intermediate code, the user sees everything occurring at the source level. Thus the IFPROBBER results are independent of compiler optimizations, and reflect the probabilities associated with the static source branches. This tool and an earlier version were used in-house for competitive benchmarking. The version used in this experiment was intended for release to customers, but the release containing it was in testing when Multiflow closed.

MFPixie. The second compilation invoked the **Multiflow Pixie-like Tool**, an internal Multiflow development tool modeled after the MIPS Pixie utility. Being an internal tool it is somewhat more flexible than the MIPS product (if you are willing and able to write code that interacts with the compiler). It works by inserting frequency counters into the output of the compiler, but now counting accurately how often each RISC-level instruction is executed. It is capable of giving very detailed dynamic information about instruction execution.

Somewhat unsatisfying was the fact that we had to turn off the compiler's global dead code elimination in order to synchronize the branch counts between the IFPROBBER and MFPixie runs. The problem is that dead code elimination removes conditional branches with constant outcome, hence changes the total number and order of conditional branches in the program. Since these dead branches always go in one direction, the branch count is unaffected by our having left them in, but the presence of dead code (including the branches) slightly inflates our results. We approximated that effect by measuring the amount of dead code that the compiler would have eliminated for each of the SPEC benchmarks, with the result shown in table 1.

PROGRAM	*DEAD CODE*	*PROGRAM*	*DEAD CODE*
li	0%	eqntott	4%
fpppp	1%	tomcatv	14%
spice2g6	1%	espresso	18%
gcc	2%	nasa7	20%
doduc	2%	matrix300	29%

Table 1. The amount of dead code (measured dynamically) that would have been eliminated from each of the SPEC benchmarks if the compiler had been permitted to do so.

Of the four programs with large enough factors to make a significant difference, *tomcatv*, *espresso*, *nasa7*, and *matrix300*, all but *espresso* are so predictable that this difference seems unimportant. Were one to eliminate dead code from *espresso*, however, *espresso*, already one of the less predictable programs, would be significantly more unpredictable.

Program Sample Base

We relied heavily on the SPEC benchmark suite for our experiments, and used all of the programs found in SPEC v1.1. In addition to the datasets that come with SPEC, we added several of our own, where practical. We also added several other programs and datasets not in SPEC. Table 2 lists the programs and a brief description of datasets.

Dataset Selection

There was no practical way for us to choose our datasets "at random." Instead, we took the datasets provided with the SPEC suite and, where practical, augmented them with datasets that we thought were both realistic uses of the program being tested, and as different in spirit as possible from the datasets already there. So, for example, to the search-oriented 9queens dataset in *li*, we added a scientific calculation (the SPEC program *tomcatv*, which we rewrote in xlisp), and a lisp program produced as the output of a machine language simulator that works by generating lisp from pseudo assembly code. The pseudo assembly code being simulated computes primes using the sieve method. Similarly, *mfcom* was given programs that compiled, in one case, typical systems-oriented C programs, and in the other typical scientific FORTRAN routines.

Nonetheless, bias enters into this selection process, and thus these results. Since only two people were selecting, we might be more prone to picking sets of similar data than different random users. In addition, several of the datasets that came with the SPEC suite were similar to each other.

On the other hand, a code developer might not be able to anticipate every use a program will be put to, but should be able to provide sample data that exercises all of the code in a natural way. We couldn't do that in providing our experimental sample data. We believe that the situation presented here is probably not a bad reflection of the real-life use of programs.

3 Results and Discussion

Results

We set out to examine two subjective claims: that branches go in one direction most of the time, and that by using previous runs of a program, one can predict that direction well. We believe that our results show both statements to be true.

Figures 1a&b show the number of instructions between breaks when branches are not predicted, both with and without subroutine call/return breaks. *fpppp*, with a huge basic block in its inner loop, is very uncharacteristic in having 150-170 instructions per break. Otherwise, the FORTRAN programs in our sample have between about 15-25 instructions per break; the C programs have between 5-17. (A compiler trying to extract ILP from blocks this size might have a difficult time, especially since basic blocks tend to have data-dependent sequences of instructions leading up to a test.)

Figures 2a&b, and Table 3 show the best possible prediction for each dataset and how well we were able to predict that dataset using the scaled sum of the other datasets. In each case, we are measuring instructions per mispredicted branch, with all indirect jumps and calls (and their returns) considered mispredicted branches. Table 3 lists the programs with only one meaningful dataset. We believe that any reasonable method will predict those programs' branch directions almost perfectly.

In Figure 2a, *spice2g6* is broken out. Although predicting *spice2g6*'s branch behavior is much less successful, the number of instructions per break is always large. Indeed, Circuit2 is the

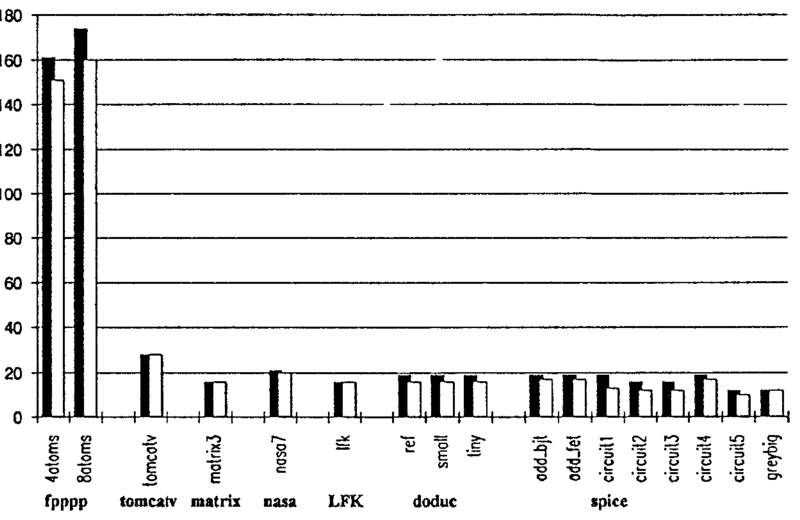
FORTRAN/FLOATING POINT PROGRAMS AND THEIR DATASETS

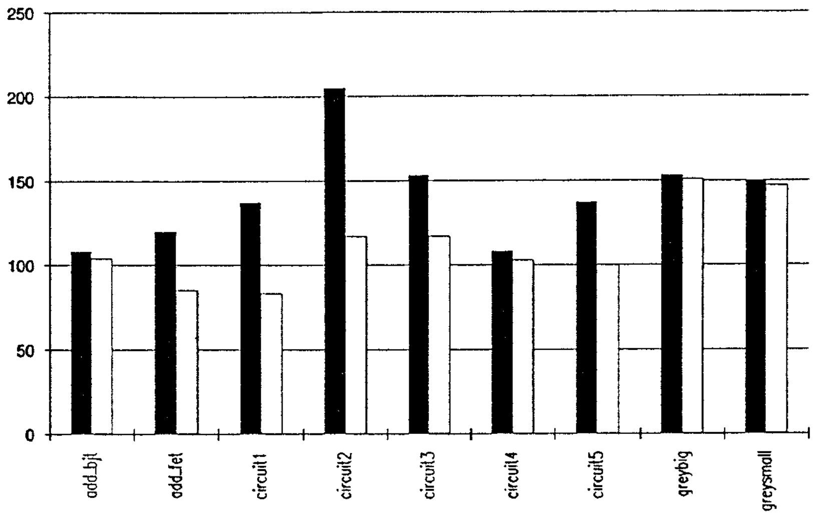
PROGRAM NAME	*PROGRAM DESCRIPTION*	*DATASET NAMES*	*DATASET DESCRIPTIONS*
013.spice2g6	Electronic design simulator	circuit1, circuit2, circuit3, circuit4, circuit5 add_bjt, add_fet greysmall greybig	Examples from Spice version 2G User's Guide, appendix A 4-bit all nand adders based on circuit4 (ttl and mosfet gates) Greycode counter, smaller SPEC input Greycode counter, larger SPEC input
015.doduc	Nuclear reactor modeling	tiny, small, ref	All similar datasets from SPEC
020.nasa7	7 synthetic kernels		Program does not read a dataset
030.matrix300	300x300 linear matrix solver		Program does not read a dataset
042.fpppp	Quantum chemistry	4atoms, 8atoms	Different settings of parameter, both from SPEC
047.tomcatv	Mesh generation and solver		Program does not read a dataset
LFK	Livermore FORTRAN Kernels		Program does not read a dataset, only subr KERNEL measured

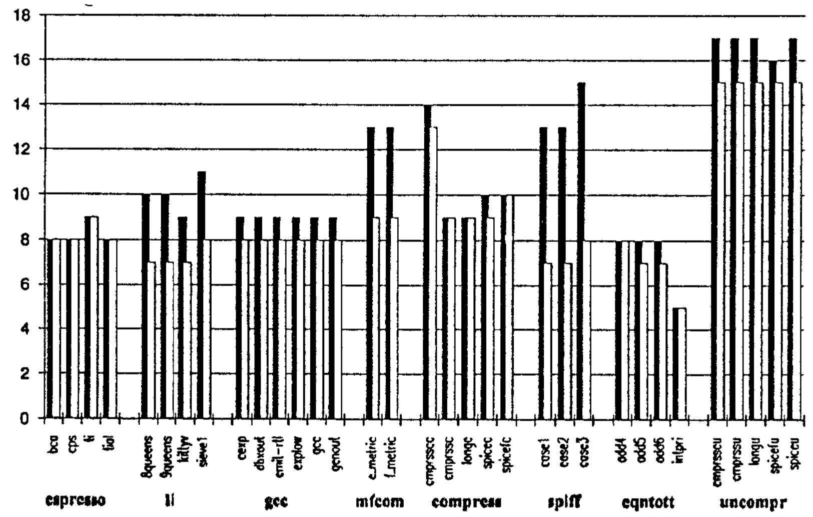
C/INTEGER PROGRAMS AND THEIR DATASETS

PROGRAM NAME	*PROGRAM DESCRIPTION*	*DATASET NAMES*	*DATASET DESCRIPTIONS*
001.gcc1.35	GNU C compiler	19 modules	19 compiler modules from SPEC. We ran all, report on only 6
008.espresso	PLA optimizer	bca, cps, ti, tial	SPEC reference datasets
022.li	XLISP 1.6 public domain lisp interpreter	8queens, 9queens kittyv sieve1	SPEC input, placing 8 or 9 queens on a chessboard SPEC tomcatv rewritten in XLISP Prime number sieve, output of machine lang to lisp simulator
023.eqntott	Converts boolean equations to truth tables	add4, add5, add6 intpri	Naive sum and carry equations for 4, 5, and 6 bit adders, respectively Priority circuit, from SPEC
compress	UNIX file compression, SPEC 3.0	cmprssc, cmprss long spicef, spice	C source, & Multiflow compiled image for SPEC 3.0 compress The SPEC 3.0 reference data FORTRAN source and compiled image for spice
uncompress	compress, above, with switch set for decompression instead	cmprssc, cmprss, long, spice, spicef	Same datasets used for compress
mfcom	The Multiflow C & FORTRAN compiler with common optimizer and backend (measured code common to both langs)	c_metric fortran_metric	5047 lines of source from cat, cpp, diff, make, maze, whetstone 5855 lines of scientific subroutine source (These datasets were originally used to profile the compiler)
spiff	File comparison tool included in SPEC	case1, case2 case3	Pairs of files of floating point numbers , some differences 26/28 line directory listings with the last few lines different

Table 2. The programs tested and their datasets.







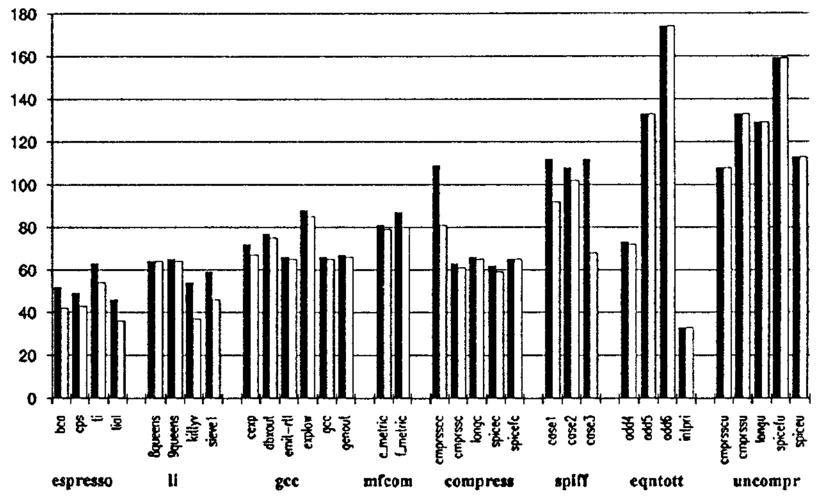


Figure 1a and 1b. Instructions per break in control when branches are not predicted. The number of instructions executed by the given program running the given dataset, divided by the number of breaks in control. Here, breaks in control are counted two different ways, and both results are shown. The black bars count as breaks all conditional branches (with no attempt to predict their direction), as well as indirect jumps, indirect calls and their associated returns. The white bars add direct subroutine calls and returns to the number of breaks. The programs are broken into FORTRAN/Floating Point (figure 1a) and C/Integer (figure 1b).

Figure 2a and 2b. Instructions per break in control when branches are predicted. Again, the number of instructions is divided by the number of breaks in control, but now branches are predicted to go in one direction and only those which are mispredicted, or are indirect jumps or calls, are counted as breaks in control. The black bars show the result of using the best possible prediction: each dataset is used to predict itself. The white bars show the result of using the sum of all the other datasets, weighed by dataset size, to predict the given dataset. In Figure 2a, the *spice2g6* datasets are shown. In Figure 2b, the C/Integer programs are shown.

hardest to predict, but still has 117 instructions between breaks because its branches are so unidirectional to start with[3]. Figure 2b shows the C/Integer programs. There are isolated instances where the prediction method doesn't do very well, but in general using the other datasets to predict a given dataset is almost always effective at predicting branch directions in these programs. Here instructions per break range from about 40 to about 160.

PROGRAM	*DATASET*	*INSTRS/ BREAK*
tomcatv		7461
matrix300		4853
nasa7		3400
fpppp	4atoms	951
	8atoms	1028
LFK		399
doduc	tiny	257
	small	269
	ref	275

Table 3. Instructions/break. (FORTRAN programs with little variability in datasets.)

Figures 3a&b show how well or poorly we can do using one single dataset to predict another. Considering the best possible prediction (using a dataset to predict itself) to be 100%, we show how close to that we come with the best other dataset, and how close we come with the worst. In most of the programs (*espresso, li, compress, spiff, eqntott, spice2g6*) the worst tended to hover around 50-70% of what was possible. The most dramatically bad worst cases were in *spice2g6* and *compress*. In *spice2g6*, the worst cases came about when a dataset was used to predict another that ran over 20,000 times as long. *compress* was generally very predictable (note how well it does in Figure 2b), except that one dataset, *cmprssc*, was very different from the others. This demonstrates that one has to pick datasets carefully (for example, by picking several and accumulating them, as in Figure 2)

ILP Compilers Will Get Larger Candidate Sets Than This.

There are several reasons that an ILP compiler will tend to get larger candidate sets than the numbers of instructions between breaks shown here:

Dominator parallelism. Code often contains **hammocks**, or places where the code splits off and then quickly rejoins. The splitting test may be very unpredictable, causing a break in control, but if the compiler is sophisticated it might be able to pull instructions up past the hammock despite the break in control (see, for example, [Bernstein and Rodeh 91]).

[3]Circuit2 is also very short. It runs 1/10,000 as long as greybig, for example.

Considering instructions from both sides of a branch. An indirect subroutine call, for example, might represent an absolute end of candidate selection for an ILP compiler. But an unpredictable branch need not be. Several techniques of ILP code generation allow instructions from both sides of a branch to be executed speculatively.

The distribution of runs of instructions between mispredicted branches will not be constant. If one is trying to avoid flushing a pipeline, then a mispredicted branch is a mispredicted branch. It has its cost and the system pays it. But for ILP purposes, the actual distribution of branches is significant. For example, far more ILP will be available if one has 80 instructions followed by two mispredicted branches than if one has 40 instructions, a mispredicted branch, 40 more instructions, and another mispredicted branch. Branches in real programs are not evenly spaced.

The compiler turned some single breaks into multiple breaks. For example, *switch* statements that might cause a single break were turned into linear or cascaded *ifs* with many breaks. In a discussion above we suggest a compiler methodology for handling these to cause only a single break, but allow the candidate set to go beyond a very predictable multiple destination branch.

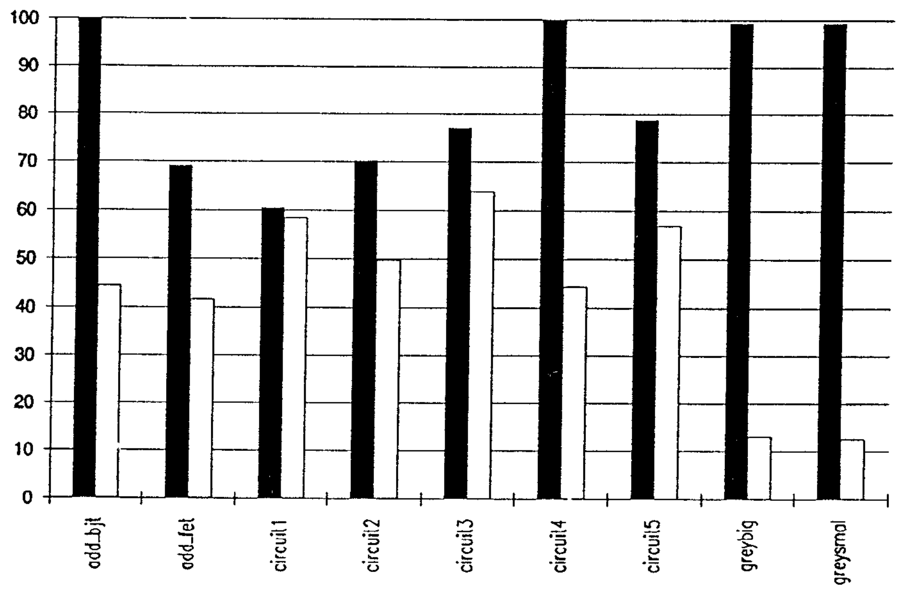
Informal Observations

In the course of this experiment several things seemed clear when we measured them but didn't seem to merit inclusion in the formal results, or were perceptions we had along the way, but didn't measure or carry out carefully.

"Coverage." We felt that when a dataset predictor did poorly, it was usually because it emphasized a different part of the program than the target dataset, rather than that the branches changed direction. We tried many schemes to capture this concept in some measurable quantity: we tried measuring the percentage covered by screening out predictor branches that were below some threshold, both absolute and relative to program size; we tried measuring the overlap between predictors and targets in various ways; and so on. Nothing we tried seemed to correlate well with the results. Either the intuition we had is wrong (the problem is that the branches were changing direction) or we simply haven't looked hard enough, perhaps because the relationship is subtle. *spice2g6* in particular was very difficult to predict, which matches its reputation. We believe that this is a result of different datasets using entirely different modules of the simulator, but, again, were not able to meaningfully quantify this.

Although *compress* is really two distinct programs (compression and uncompression) controlled by a command line switch, it is one program as seen by our tools, so we were able to compare runs of the two different modes. Unsurprisingly, there seemed to be no correlation between them. Using the data from one to predict the other is a very bad idea.

Scaled vs. unscaled summary predictors. We reported above on the strategy of using the sum of all of the datasets except the one being predicted as a predictor. We tried this three different ways: we simply added the counts for each test for each dataset (called *unscaled*), we divided each count by the total branches executed by the predictor dataset to give each dataset equal total weight (*scaled*), and we allowed each dataset to get one vote to predict in which direction the branch would go, no matter how large or small its execution count was (*polling*). Polling seemed,



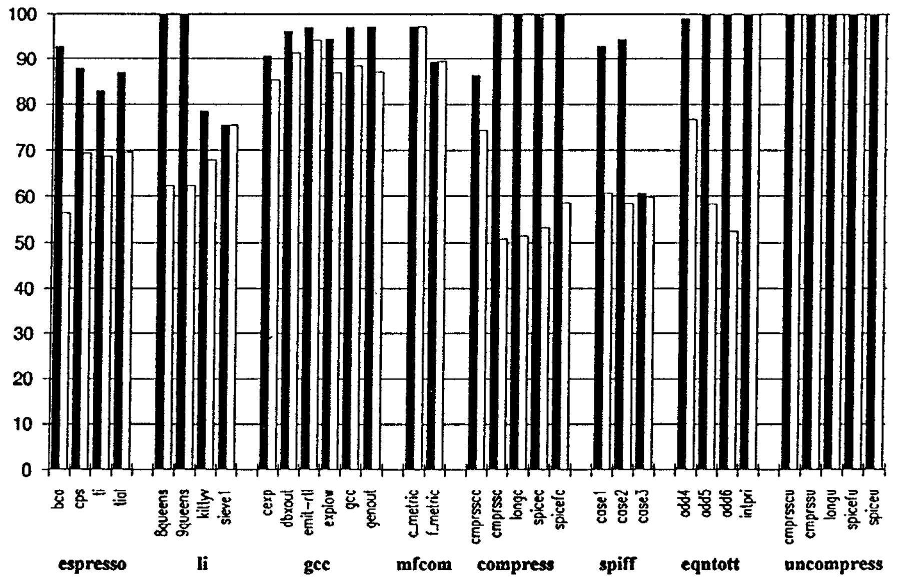


Figure 3a and 3b. The best and worst predictions of a dataset using another dataset. The charts show how well or poorly one can do in this sample when one dataset is used to predict another. The best possible result, when a dataset is used to predict itself, is considered 100%. The black bar shows the percentage of that best possible prediction obtained by the other single dataset that came the closest to that, and the white bar shows the same for the worst other dataset. Figure 3a shows these ratios for the *spice2g6* datasets, and Figure 3b shows it for the C/Integer datasets.

unsurprisingly, to perform poorly and it was discarded before it was measured carefully. The other two appeared to perform as well as each other. Sometimes one was significantly better than the other, but on average they were indistinguishably close. We chose scaled in our reports, but only because it was intuitively more satisfying, though it requires slightly more computation.

Simple opcode heuristics. The conventional wisdom is that looking at the opcode and using simple heuristics to predict branch directions does not work as well as feedback-based methods, although we mentioned above one study that gives hope that more elaborate heuristics might. We tried using very simple heuristics, distinguishing between loops and nonloops, and our results were, unsurprisingly, terrible. Except for some very easily predictable vectorizable codes, this usually gave up about a factor of two in instructions per break.

Branch percent taken as a "program constant." We were surprised at how constant the measure of *percent taken* was from dataset to dataset within a given program, except for *spice2g6*. *spice2g6* had one dataset, greysmall, that took its branches 21% of the time, and another, circuit5, at 76%. Remarkably, greysmall was able to predict circuit5 far better than was circuit2, which took its branches 72% of the time! Again, this seems to show that the datasets are emphasizing different parts of the program.

Except for *spice2g6*, the maximum difference in branch percent taken for the datasets of a single program was 9% (and most were much closer), which seems remarkably constant.

4 Summary and Discussion

Branch prediction is an important capability for high performance CPUs, especially for those that offer instruction-level parallelism. Static branch prediction offers advantages over dynamic prediction, and for some uses is the only practical alternative. The experiments reported upon here show that static prediction can be done almost as well as is possible by taking previous runs of a program, and using those runs to make decisions about which way branches will go in future runs. Although it is possible to pick previous runs that are not effective, it appears that a variety of runs with their branch direction counts added together make a good predictor. In this fashion a compiler can consider large candidate sets without a likely break in control. This result is not unexpected in FORTRAN/Floating point codes, but holds for systems codes as well. Inlining is an important capability for compilers seeking instruction-level parallelism, though evidently the loss in not inlining is small in terms of the candidate sets available.

An important issue not covered here is the user interface to a system that provides this feedback. We know of no work published in this area, nor do we know of any commercial compilers that have offered branch direction prediction feedback as an option. The effectiveness of any future system that offers this capability rests not only on the predictability demonstrated here, but on the existence of an interface that users find palatable.

Acknowledgments.

Brian Siritzky developed the Multiflow IFPROBBER tool used in these experiments, Kevin Dowd wrote an earlier version. Richard Lethin originally developed Multiflow Pixie. Woody Lichtenstein and Rajiv Gupta contributed valuable suggestions about breaks in control.

References

[Bandyopadhyay, et al. 87] S. Bandyopadhyay, V. S. Begwani and R. B. Murray. "Compiling for the CRISP microprocessor," *Spring Compcom 87*, pp 96-100, IEEE Computer Society, September 1987.

[Bernstein and Rodeh 91] D. Bernstein and M. Rodeh. "Global instruction scheduling for superscalar machines," *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 241-255, ACM, June 1991.

[Colwell et al. 87] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth and P. K. Rodman. "A VLIW architecture for a trace scheduling compiler," *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 180-192, Palo Alto, California, ACM and IEEE Computer Society, October 1987.

[Conte and Hwu 92] T. M. Conte and W. M. Hwu. "The validity of optimizations based on profile information," (personal communication of working draft), 1992.

[Chang, Mahlke and Hwu 92] P. P. Chang, S. A. Mahlke and W. M. Hwu. "Using profile information to assist classic code optimizations," (personal communication of paper to be published), 1992.

[Ellis 85] J. R. Ellis. *Bulldog: A Compiler For VLIW Architectures*, The MIT Press, Cambridge, MA, 1985.

[Fisher 81] J. A. Fisher. "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, 30(7), pp. 478-490, July 1981.

[Fisher 83] J. A. Fisher. "Very long instruction word architectures and the ELI-512," *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pp. 140-150, Stockholm, Sweden, ACM and IEEE Computer Society, June 1983.

[Freudenberger and Ruttenberg 92] S. M. Freudenberger and J. C. Ruttenberg. "Phase ordering of register allocation and instruction scheduling," to appear in *Proceedings of the International Workshop on Code Generation—Concepts, Tools, Techniques*, Springer-Verlag, London, UK, 1992.

[Johnson 91] W. Johnson. *Superscalar Microprocessor Design*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Lam 88] M. Lam. "Software pipelining: an effective scheduling technique for VLIW machines," *Proceedings of the SIGPLAN '88 Conference on Programming Language*

Design and Implementation, pp. 318-327, ACM, June 1988.

[Lee and Smith 84] J. K. F. Lee and A. J. Smith. "Branch prediction strategies and branch target buffer design," *IEEE Computer 17, 1*, pp 6-22, January 1984.

[McFarling and Hennessy 86] 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.

[Mahlke et al. 92] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau and M. S. Schlansker. "Sentinel scheduling for VLIW and superscalar processors," to appear in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (this volume),* Boston, Massachusetts, ACM and IEEE Computer Society, October 1992.

[Ponder and Shebanow 90] C. G. Ponder and M. C. Shebanow. "An information-theoretic look at branch-prediction," in Carl Ponder, *Studies in Branch Prediction (preprint),* #UCRL-ID-106077, Technical Information Department, Lawrence Livermore National Laboratory, Livermore, CA, September 1990.

[Rau and Glaeser 81] B. R. Rau and C. D. Glaeser. "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," *Proceedings of the 14th Annual Workshop on Microprogramming*, pp. 183-198, ACM and IEEE Computer Society, October 1981.

[Rogers and Li 92] A. Rogers and K. Li. "Software support for speculative loads," to appear in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (this volume),* Boston, Massachusetts, ACM and IEEE Computer Society, October 1992.

[Smith 81] J. E. Smith. "A study of branch prediction strategies," *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pp. 135-148, ACM and IEEE Computer Society, May 1981.

[Wall 91a] D. W. Wall. "Limits of instruction-level parallelism," *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176-188, ACM, April 1991.

[Wall 91b] D. W. Wall. "Predicting program behavior using real or estimated profiles," *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pp. 59-70, ACM, June 1991.