Source Level Static Branch Prediction

W. F. Wong

Department of Computer Science

School of Computing

National University of Singapore

Lower Kent Ridge Road

Singapore 119260

Tel: (+65) 874-6902 Fax: (+65) 779-4580

Email: wongwf@comp.nus.edu.sg

March 24, 1999

Abstract

The ability to predict the directions of branches, especially conditional branches, is an important problem in modern computer architecture and advanced compilers. Many static and dynamic techniques have been proposed. Today, all state-of-the-art microprocessors have some form of hardware support for dynamic branch prediction. Static techniques, on the other hand, have not been widely studied because of the belief that they give poorer results. However, good static branch predictions are invaluable information for (static) compiler optimization or performance estimation. In this paper, we propose performing static branch prediction at the *source* code level. The assumption is that the source code contains information unavailable at the assembly or machine code level that may be used for branch prediction. Empirical studies on 14 integer Spec benchmarks indicate that the simple heuristics proposed can be effective in practice.

1 Introduction

Branch prediction is the attempt to foretell the directions of branch instructions in a

program. It is a particularly important problem in computer architecture especially with pipelined superscalar processors. In a pipelined processor, the final branch decision cannot be made until the instruction has passed a number of pipeline stages. In a deep pipeline, this is often somewhere in the middle of the pipeline. For example, for the DEC Alpha 21164 microprocessor [8], for integer branches, the final resolution takes place in the 4th stage of its 7 stage integer pipeline. If the branch prediction was incorrect, then the instructions that followed the branch in entering the pipeline must be *flushed* (cancelled and undone). The situation is made worse in superscalar processors which allow several (typically four in today's technology) instructions to be issued in a single machine cycle. So in the example of the Alpha 21164, some 12 instructions may have to be flushed on a misprediction. This translates to a severe performance penalty. For this reason, almost all processors today have some form of dynamic hardware branch prediction facility.

In this paper, we investigate the efficacy of branch prediction based on inferences done by compile-time inspection of the source code. This is refered to as *program-based* branch prediction [3] as opposed to *profile-based* prediction [17]. Previous approaches have been done mainly at the machine code level. This work is concerned only with conditional, non-loop branches, i.e. all conditional branches outside of loops and all conditional branches within a loop but excluding final conditional branch back to the top of a loop. We believe loop-based branches are effectively handled by existing schemes [16]. The contribution of this paper is two-folded:

- it is a quantitative study of source level program-based branch prediction based on the 14 C programs from the Spec 92 and Spec 95 benchmarks;
- a new set of heuristics based on naming conventions is proposed and evaluated.

Evidence shows that using a set of simple heuristics, static branch prediction at the source code level can achieve good results.

In section 2, we shall survey the various existing methods and proposals on the problem of branch prediction. In section 3, a set of source level program-based branch prediction heuristics is proposed. Section 4 outlines the experimental setup used in the study while section 5 reports on the results. This is followed by a discussion and a conclusion.

2 Previous Works

There are three main families of branch prediction strategies [7]. One of the earliest approach is to fix a branch prediction strategy in the hardware[1]. An important example of this prediction strategy that has worked very well in practice is that of predicting backward branches (i.e. branches to instructions that precede the branch instructions themselves in the program order) as taken because it was found that backward branches are more likely to be taken than forward ones [16]. While such schemes are effective especially in loops, they fair less well in general conditional branches.

A dynamic scheme is one that performs branch prediction based on the specific behavior of the program in execution. They are based on the assumption that the historical behavior of branches can serve to predict future branches. Proposals include the branch history bits [16], the branch target buffer [12], and the *gshare* predictor [15]. For example, the DEC Alpha 21164 microprocessor has a 2 bit history counter associated with each branch instruction in the instruction cache [8]. This is incremented on taken branches and decremented on non-taken ones. A counter value greater than 0 is cause the branch to be predicted as taken and vice versa. When the branch is finally resolved, the counter is updated. The problem with these approaches is that a significant amount of hardware is required to support them. Furthermore, the extra hardware may lengthen the critical path of instruction processing.

Both of the above approaches cannot involve the compiler as they are done only at runtime. This is undesirable because advanced compilers, especially those that attempt optimizations including some form of code motion and scheduling, global register allocation, inlining etc., need branch prediction information to achieve good results (see for example Lowney et. al. [13]).

The CRISP compiler [1] was among the first compiler to perform static branch prediction based on the source code. It detected loop-based branches and by using previously gathered data, predicted the direction of branches based on the comparison operator and

[1]This family of strategies is refered to as 'static' by Cragon [7] but in this paper, we will reserve this word for compile or link-time branch prediction strategies (which is refered to as 'semi-static' strategies by Cragon).

the types of operand in C source programs.

The case for static branch prediction was made by Ball and Larus [3] where static branch prediction done on executable codes was studied. Static branch prediction based on analyzing the abstract syntax tree of a program was described by Wagner et. al. [18]. Both approaches are heuristics driven. Attempts to improve the performance of heuristics based on studying a corpus of existing programs was described by Calder et. al. [4] where neural networks were employed. The work of Ball and Larus was also extended in Wu and Larus [21] by the use of the Dempster-Shafer theory of evidence. However, it was shown by Calder et. al. in [5] that this method is susceptible to differences in compilers and architectures as it was based on a prior prediction of object code. Recent works such as that of Grunwald, Lindsay, and Zorn [11] propose the use of static predictors to aid dynamic predictors.

This paper proposes a set of source-code level heuristics for static branch prediction. Some of the heuristics described are extensions of earlier reported ones. An important innovation is the use of names (macro, function and variable) as part of the heuristics. The entire prediction is strictly program-based and do not use any profiling means directly. Specifically, they are based on analyzing C source programs during and just after macro expansion. It is therefore also less vulnerable to influences of architectures and compilations.

3 Source Level Branch Prediction Heuristics

The heuristics studied in this paper are as follows:

- **Baseline**. In this 'heuristic', the branches are assumed 'as-is'. In other words, all (C) 'if' branches are predicted as 'taken'.

- **Random**. Here, the 'if' and the 'else' branches are given a 50-50 chance of being predicted as 'taken'. The standard Unix random number generator `drand48` was used for the generation of the prediction probability. This is precisely the strategy used in a trace scheduling compiler [13] which requires branch prediction at compile-time to perform interprocedural code optimizations. However, there is no data on

the effectiveness of this heuristic.

- **Heuristic S.** This is based on a scoring system. Both the 'if' and the 'else' branches are examined as follows:
  - if the if-condition is an equality comparison, then a score of -1 is recorded against the 'if' branch. Rationale: assuming a uniform distribution of data values, the probability of two data values being equal is low.
  - for each logical 'and' found in the if-condition, a score of -1 is recorded against the 'if' branch. Rationale: satisfying two predicates simultaneously is generally harder than satisfying just one.
  - for each logical 'or' found in the if-condition, a score of 1 is recorded against the 'if' branch. Rationale: 'or' is used to relax constraints.
  - the branch which contains '`fprintf(stderr, ...);`' has a score of -1 recorded against it for every such statements. Rationale: writing to `stderr` is generally used (in production programs) for error reporting or debugging purposes. Therefore, such a branch is less likely to be executed.
  - in each branch, for every call to functions which has the word '`exit`', '`warn`' or '`err`' as part of its name will have a score of -1 recorded against it. Rationale: such an arm of a branch probably perform some form of error handling and is therefore unlikely to be executed under normal circumstances.
  - in each branch, for every occurence of the '`return`' statement, a score of -1 would be recorded against it. Rationale: here the bet is that on entering a procedure, one would do a certain amount of work before returning, and therefore the arm containing the return is less likely to be resorted to than the other.
  - if the if-condition is a greater than check against a variable (or expression) in which the string '`max`' occurs, then a score of -1 would be recorded against it. Rationale: the check is most probably to see if certain limits have been exceeded, and most of the time under normal circumstances, this should not happen.

– if the if-condition is a lesser than check against a variable (or expression) in which the string '`min`' occurs, then a score of -1 would be recorded against it. Rationale: as above.

For each conditional branch, the branch with a more positive score will be predicted as 'taken'. For this study, we did not investigate the use of different weights, i.e. scores other than 1 and -1, for the different rules. Such an investigation would take us beyond the basic aim of this paper which is to demonstrate that the heuristics work well in practice. For those branches where no inference can be made, a random choice is made as to whether the 'if' or the 'else' branch (with a probability of 0.5 each) is to be predicted as 'taken'.

- **Heuristic SF**. This heuristic is an extension of the S heuristic. A table of integer Unix system functions and the values they are likely to return as well as error codes are kept (with the implicit assumption that error is unlikely). Should these functions participate in an 'if' conditional, an analysis would be performed to see which branch is the likely one. For example, in

```
if (fopen(...)) {
    ...
}
else {
    ...
}
```

it is likely that the 'if' branch is taken. The integer values kept in the table are encoded in the form of ranges. So for example, for the `fopen` system call, the likely value is 'greater than zero' while the error value kept is 'equal to zero'. The implementation of this technique has been extended to handle simple propagations. For example it is able to analyze the following:

```
FILE *f;
  ...
```

```
f = fopen(...);
  ...    /* No redefinition of f */
if (f == NULL) {
   ...
}
else {
   ...
}
```

For this purpose, def-use chains of variables are maintained.

- **Heuristic SFM**. A number of system calls are implemented as macros and expanded during macro expansion via inclusion of system header files. A good example is the `getchar()` "function". This heuristic extends Heuristics S and SF by the inclusion of macros into consideration. Heuristics S and SF, on the other hand, is applied after macro expansion. The reader may notice that another way of doing this is to perform the Heuristic SF processing *prior* to the macro expansion of the C compiler. However, this was not done in the hope that macro expansion may yield further information.

- **Dynamic Prediction**. For comparison purposes, the combined bimodal-*gshare* predictor proposed by McFarling [15] was also implemented. In our implementation (which closely mirrors the original proposal of McFarling), three tables of 256-entry, 2-bit saturating counters are maintained (see Fig. 1). One table is used for a bimodal predictor. The lowest 8 bits of the line number of the 'if' statement is used to index this table. In the original proposal, the lowest 8 bits of the program counter pointing to the branch instruction is used. Since aliasing is expected in the scheme, we do not believe this violates the predictor's workings in any serious way. For a taken branch, the 2-bit counter is incremented. Otherwise, it is decremented. However, the counter is saturating and cannot be incremented beyond 3 or decremented below 0. A value greater than 1 is equivalent to predicting the branch as taken, and vice-versa. The second predictor is *gshare*. Here, the counter works in the same way.

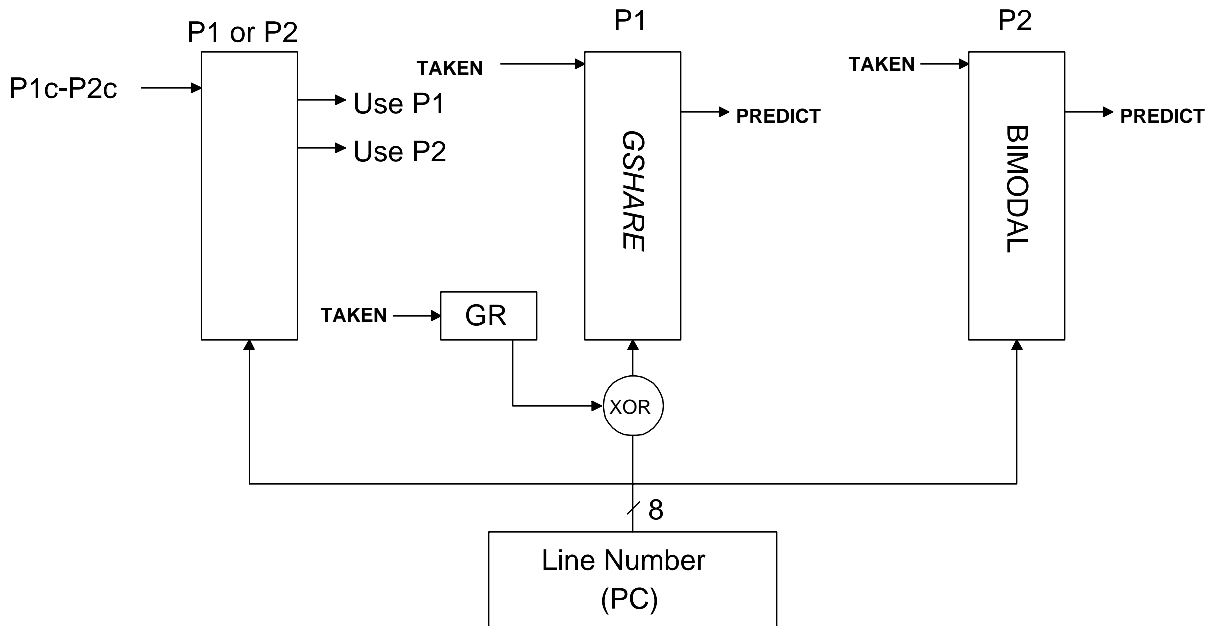

Figure 1: Combined Dynamic Branch Predictor.

What is different is the way the table is indexed. The lowest 8 bits of a special shift register, the `GR` register is exclusive-OR'ed with the 8 bits of the line number ('PC') to obtain the index. This is termed '*gshare* 8/8' by McFarling. `GR` is maintained as follows: if a branch is taken, a bit 1 is shifted into the right-end of the register while a not taken branch causes a 0 bit to be shifted in. The main assumption in this scheme is that the behaviour of neighbouring branches are correlated. By hashing on the past branches' behaviour (captured by `GR`), for each branch one can maintain a number of counters for each predecessor branching pattern. A third table indexed by the lowest 8 bits of the line number checks if for a particular 'if', the bimodal or *gshare* was accurate in the past and use the more accurate one [15]. The reader should note that this is a far more elaborate scheme than those which exist in practice. By comparison, for example, the DEC Alpha 21164 scheme maintains a 2 bit local counter for only those instructions that are in the instruction cache.

Our heuristics make the assumption that programmers name their variables in a predictable way. It is certainly possible to violate this assumption but in practice, for various software engineering reasons, we expect that the naming of variables to be fairly pre-

dictable. Our initial results with tests on the Spec benchmark suite also seem to validate this assumption.

4 Experimental Setup

The heuristics were tested on the 14 C programs from the Spec 92 and Spec 95 benchmark suites. The GNU C version 2.7.2 compiler was extended for this purpose. The modifications include:

- The C macro preprocessor (`cccp.c`) was extended to insert markers at macro expansion sites. These markers maintain pointers in such a way that at a later stage the names of the macros prior to expansion can be retrieved. Essentially, the pointers are line indices into a simple ASCII file containing, in sequential order, the names of the macros encountered in processing the file.

- A new pass that follows immediately the macro preprocessor was added to the compiler. This is where the actual branch prediction processing is performed. It is essentially a strip-downed version of the C parser (`c-lex.c` and `c-parse.y`) and outputs an augmented version of the macro-preprocessed C program. The main addition is a data structure which captures all the prediction information. The data structure also contains the space allocated for the counters that will be used during the execution of the program to capture runtime statistics.

- The original C parser is modified to insert statistic gathering function calls at the certain locations in the code, namely the start of each branch in a conditional branch.

- The statistic gathering functions were added to the main C runtime library. In addition, the C startup routine is made to do some additional initializations which are mainly related to patching pointers so that counters can be properly located during runtime.

Program	Lines of code	Total no. of cond. executed (excl. loops)	Dyn. Pred. (Wt. Ave.)	Perfect Static Pred. (Wt. Ave.)
Spec CINT 92				
008.espresso	18,901	2.737×10^8	0.8987	0.7486
022.li	19,413	3.783×10^8	0.9145	0.8193
023.eqntott	3,901	3.535×10^8	0.9169	0.6754
026.compress	1,504	3.591×10^7	0.8836	0.7508
072.sc	8,818	8.205×10^7	0.9659	0.9188
085.gcc	152,531	3.049×10^8	0.8862	0.8583
Spec CINT 95				
099.go	33,172	8.691×10^9	0.7616	0.7567
124.m88ksim	22,035	4.959×10^9	0.9454	0.9243
126.gcc	214,090	2.665×10^9	0.8680	0.8433
129.compress	4,461	4.732×10^9	0.8903	0.7685
130.li	8,986	4.250×10^9	0.9012	0.8165
132.ijpeg	34,269	2.455×10^9	0.8411	0.8270
134.perl	32,064	3.048×10^9	0.9284	0.8389
147.vortex	71,389	8.091×10^9	0.8956	0.7157
		Ave. over 14 benchmarks	0.8926	0.8044

Table 1: Characteristics of C benchmarks used.

5 Results

In this section, we shall present the main results of our investigations into the efficacy of the proposed heuristics. The 14 benchmarks were compiled with the modified compiler under the "-O" option of the compiler and executed over the 'reference' data set as defined by Spec. The outputs were validated as per Spec requirements. For most of the benchmarks the reference data set consists of several independent data sets and therefore runs. Furthermore, the length of each runs differ. We therefore need to introduce a fair means of reporting the results.

For a program P_j executing over a data set D_k, let $N^H_{i,I/I}[P_j(D_k)]$ be the number of times the 'if' arm of the ith branch was predicted as 'taken' under heuristic H and indeed was taken. Similarly, we can define $N^H_{i,I/E}[P_j(D_k)]$, $N^H_{i,E/I}[P_j(D_k)]$ and $N^H_{i,E/E}[P_j(D_k)]$ as, for the ith branch, the number of times the 'if' arm was taken given that the 'else' arm was predicted, the 'else' arm was taken when the 'if' arm was predicted and the 'else' arm was predicted and was indeed taken, respectively, under heuristic H. For brevity, when the context is clear, we shall drop the '$[P_j(D_k)]$' portion of the formula. The total number of branches covered by a heuristic H for a program P_j executing over a data set D_k is therefore

$$N^H[P_j(D_k)] = \sum_i (N^H_{i,I/I} + N^H_{i,I/E} + N^H_{i,E/I} + N^H_{i,E/E})$$

We define the *accuracy* of heuristic H for a program P_j executing over a data set D_k as

$$\text{Accuracy}^H[P_j(D_k)] = \frac{\sum_i (N^H_{i,I/I} + N^H_{i,E/E})}{N^H[P_j(D_k)]}$$

The *miss rate* of heuristic H for a program P_j executing over a data set D_k is defined as $1 - \text{Accuracy}^H[P_j(D_k)]$. Due to the fact that the Spec benchmarks are executed over a set of data, we used a *weighted average accuracy* measure which is defined as follows:

$$\text{Weighted-Average-Accuracy}^H[P_j] = \frac{\sum_k \text{Accuracy}^H[P_j(D_k)] \times N^H[P_j(D_k)]}{\sum_k N^H[P_j(D_k)]}$$

Table 1 shows some of the basic characteristics of the 14 benchmarks we used in this study. Table 1 also reports the weighted average of the dynamic predictor described in the above section. The *perfect static predictor* statistics for program P_j executing over data set D_k is obtained by the following formula:

$$\text{Accuracy}^{\text{PSP}}[P_j(D_k)] = \frac{\sum_i \max(\text{if_true}_i[P_j(D_k)], \text{if_false}_i[P_j(D_k)])}{\sum_i (\text{if_true}_i[P_j(D_k)] + \text{if_false}_i[P_j(D_k)])}$$

where $\text{if_true}_i[P_j(D_k)]$ and $\text{if_false}_i[P_j(D_k)]$ is the number of times the ith if conditional of program P_j evaluated to 'true' and 'false', respectively, while executing over data set D_k.

The perfect static predictor is the upper bound for any static branch prediction strategy. Dynamic predictors are not bounded by this. Contrary to what was reported by Ball and Larus [3], our dynamic predictor performed consistently better than the perfect static predictor. We attribute this to the better dynamic predictor used in our experiments. On the average, the dynamic predictor is some 11% better than the perfect static predictor. However, as stated earlier, the domain of the static and dynamic predictors are quite different. Therefore, we argue that there is still good potential for static prediction strategies.

Program	**Heuristic S**		**Heuristic SF**		**Heuristic SFM**	
	Appl. (Stat)	Appl. (Dyn)	Appl. (Stat)	Appl. (Dyn)	Appl. (Stat)	Appl. (Dyn)
Spec CINT 92						
008.espresso	31.74	55.59	31.74	55.59	40.19	65.87
022.li	65.25	71.32	65.25	71.32	82.38	84.59
023.eqntott	91.84	62.47	91.84	63.13	91.84	64.44
026.compress	60.99	80.21	60.99	79.17	74.48	88.54
072.sc	67.35	56.12	67.35	56.12	62.60	66.33
085.gcc	87.26	70.81	88.47	74.45	93.89	79.33
Spec CINT 95						
099.go	80.40	74.61	80.40	74.62	82.28	79.47
124.m88ksim	82.97	69.95	82.97	70.44	82.96	75.92
126.gcc	78.27	76.71	78.28	77.26	87.67	83.37
129.compress	77.16	74.55	77.16	74.55	83.43	80.00
130.li	59.20	72.82	59.20	72.82	81.94	85.44
132.ijpeg	45.29	70.49	45.29	74.93	47.58	81.54
134.perl	25.91	57.88	25.91	59.44	39.48	69.65
147.vortex	50.51	42.21	50.63	42.35	56.36	51.63
Average	64.58	66.84	64.68	67.59	71.93	75.44

Table 2: Applicability of Heuristics in Percentages.

Table 2 shows the *applicability* of the heuristics, i.e. the percentage of non-loop branches for which a heuristic was able to perform a prediction. Recall that for all those (non-loop) conditionals about which it was not possible to apply any of a heuristic's rule, random predictions were made. Applicability measures the percentage of conditionals for which definite predictions were possible and therefore random predictions were not resorted to. There are two values to each, a static and a dynamic ratio. The static ratio is the percentage of conditionals in the source code for which prediction was possible. The dynamic ratio is the percentage of executed branches for which prediction was possible. The best average value was that achieved by SFM which was 75%. This compares favourably with the 61% applicability for the corresponding Spec integer code (the overall figure reported for their 23 benchmarks was 79%) reported by Ball and Larus [3] for their non-loop predictors.

Program	Baseline	Random	Heuristic S		Heuristic SF		Heuristic SFM	
			Accu.	Accu. (+Ran)	Accu.	Accu. (+Ran)	Accu.	Accu. (+Ran)
CINT 92								
008.espresso	0.4268	0.5105	0.4631	0.4843	0.4631	0.4843	0.5253	0.5121
022.li	0.2570	0.5350	0.8142	0.6629	0.8142	0.6629	0.7337	0.6642
023.eqntott	0.3329	0.3315	0.6697	0.6705	0.6697	0.6705	0.6697	0.6705
026.compress	0.4342	0.4783	0.7078	0.6596	0.7078	0.6685	0.6865	0.6753
072.sc	0.0872	0.4309	0.7428	0.7281	0.7428	0.7281	0.9057	0.6548
085.gcc	0.2352	0.7711	0.7466	0.7160	0.7466	0.7159	0.7641	0.7424
CINT 95								
099.go	0.2653	0.4931	0.7318	0.7101	0.7318	0.7101	0.7301	0.7065
124.m88ksim	0.1640	0.6796	0.8556	0.7652	0.8556	0.7652	0.8672	0.7849
126.gcc	0.2822	0.5281	0.7220	0.6895	0.7220	0.6882	0.7384	0.7086
129.compress	0.4493	0.5024	0.7322	0.7091	0.7322	0.7091	0.7049	0.6765
130.li	0.2933	0.5443	0.8079	0.6560	0.8079	0.6559	0.6998	0.6411
132.ijpeg	0.4487	0.5489	0.6664	0.3991	0.6664	0.3991	0.6558	0.5517
134.perl	0.2634	0.6969	0.7390	0.5089	0.7390	0.5075	0.7383	0.5821
147.vortex	0.3582	0.5189	0.6472	0.5654	0.6481	0.5623	0.6664	0.6206
Average	0.3070	0.5407	0.7176	0.6375	0.7177	0.6377	0.7204	0.6565

Table 3: Accuracy of Heuristics.

The results for the various heuristics are shown in Table 3. Two set of numbers are given for each heuristics. The first set of numbers are the accuracies of a given heuristics computed over those branches for which it made a prediction. The second set of accuracies are the overall results in which random predictions were made to those branches for which

the heuristic were not able to make any prediction.

In all the 14 cases, at least one of the proposed heuristics performed better than the baseline and the random method. When compared with the dynamic predictor, the best of our predictors (SFM) achieved between 58% to 96% (average 80%) of the performance of the dynamic predictor. An important point to note is that the applicability of the dynamic predictor is 100%, i.e. it can be applied to all branches.

Heuristic	**Ave. Qual. Factor (Heur Only)**	**Ave. Qual. Factor (+ Rand Others)**
Baseline	0.3775	N.A.
Random	0.6368	N.A.
S	0.8934	0.7959
SF	0.8934	0.7962
SFM	0.8954	0.8201
Dynamic	1.1097	N.A.

Table 4: Average Quality Factors of the Heuristics.

Table 4 shows the quality of the different heuristics by means of a *Quality Factor* (of using heuristic H on program P_j) defined as

$$\text{Quality Factor}^H[P_j] = \frac{\text{Weighted-Average-Accuracy}^H[P_j]}{\text{Weighted-Average-Accuracy}^{\text{PSP}}[P_j]}$$

In comparing with results reported in the literature, we believe that the quality factor is a better means of comparison. Even if the benchmarks used are the same, differences in the data set used makes directly comparison of miss ratios meaningless.

From Table 3 and Table 4, we may draw the following observations:

1. Conditional branches are not symmetric. If a static strategy of predicting the 'else' branches as 'taken', then our data suggest that one can achieve an accuracy of $0.7045 (= 1.0 - 0.2955)$ on the average. We believe that this may be an artifact of the way these particular programs were written and do not provide any convincing proof for predicting all 'else' branch as 'taken'.

2. The random strategy pursued by Lowney et. al. [13] yields an accuracy better than an expected 0.5 accuracy due to the asymmetry in the branches. A small number

of branches tend to be the ones that are executed most of the time. A random predictor that happens to predict these correctly will get a higher overall score.

3. Heuristic SF adds little to the performance of Heuristic S. The reason can be seen as follows. Take the example of `fopen`. If its return value is tested, then it is unlikely to go to the branch in which the return value is zero as this indicates an error. However, since in this error handling branch, error handling or reporting procedures, such as `fprintf(stderr,...)` or `perror`, would be invoked. But these are already tested for by Heuristics S. Thus the effectiveness of Heuristic SF is reduced.

4. Heuristic SFM performs significantly better than Heuristic SF in 072.sc, a spreadsheet program. The boundary of the spreadsheet is defined by a number of macro variables prefixed by '`MAX`' and '`MIN`' which were readily made use of by Heuristic SFM.

5. When compared to the dynamic predictor, 008.espresso gives the worst performance. We attribute this mainly to inlined procedures found in the source code. We are not sure how these fragments were generated but they tend to use terse names for variables. If this benchmark is ignored from the suite, then for both Spec 92 and Spec 95, SFM (without randomizing those branches for which predictions were not made) achieved an average of about 82% of the accuracy of the dynamic predictor. The caveat is that the dynamic predictor is applicable to all branches.

6. Our best average quality factor of 0.89 (for SFM without randomizing those branches for which predictions were not made) is about 15% better than the value of 0.774 reported by Ball and Larus[3] for its 5 integer Spec benchmarks.

7. Wagner et. al. [18] reported their static non-loop branch predictor, which is similar to our Heuristic S, achieved a miss rate of about 2.2 times that of the perfect static predictor or a quality factor of 0.80. Our SFM predictor's quality factor is 10% higher.

In addition to being qualitatively better than the previous works, we should also point out that

- Ball and Larus' prediction is performed on executable codes. This makes the prediction unavailable at the compiler level.

- Wagner et. al.'s (non-loop) prediction, though a derivative of Ball and Larus' heuristics, is done at the abstract syntax tree level. Still, information available at prior to macro processing is not utilized. This, we showed, can help improve the accuracy of the prediction and the number of branches over which prediction can be performed.

6 Discussion

It must be stressed that the proposed scheme is *not* meant to replace runtime dynamic branch prediction. Instead, the aim is to get good approximation of branching behaviour during compilation. This information is useful is several ways. Here are some examples:

1. *Trace Selection.* In trace scheduling compilers [13], the selection of good traces is crucial to performance. The proposed schemes can aid in the selection process.

2. *Software Data Prefetching.*[14] Modern microprocessors have started to provide instructions for prefetching data into the cache. Aided by branching probabilities, a compiler can better evaluate if data accesses in a basic block is worth prefetching.

3. *Global Register Allocation.* In many global register allocation algorithms, such as the priority-based graph colouring approach [6], heuristics are used to weigh the importance of virtual registers so that those that contribute to better performance get allocated to physical registers. The availability of branching information would increase the accuracy of such estimates.

4. *Code Instrumentation.* A new and popular form of code profiling and runtime information gathering is code rewriting in which 'witness' code are inserted into strategic points of the target [2]. These witnesses capture the control flow of the program during execution. As an optimization, instead of inserting instrumentation code into every basic blocks, it is possible to insert witnesses into some less frequently executed basic blocks and then reconstruct the control flow later. Again, our proposal may be able to assist in identifying less frequently executed basic blocks.

5. *Assist Hardware Branch Prediction.* It may be possible that a software-hardware approach be taken so as to improve the accuracy of branch prediction during run-time. This remains a subject for further research [11]. In any case, we are now seeing architectures that recognize the importance of branch prediction and the possibility of software branch prediction. The SPARC version 9 [19] and the PowerPC [20] instruction sets are examples of modern superscalar architectures that have introduced branch instructions with prediction bits.

As described in Section 4, the implementation of the testbed for our ideas was a modification (of no more than an estimated 5%) of the GNU C compiler, a production quality compiler. The advantage of our approach is done very early in the compilation process making branch prediction information available to nearly all passes in the back end of the compiler. This makes it possible to use the information for the abovementioned areas. What is required is a modified macro preprocessor which basically notes down how macro expansion was done so that names can be related to their expanded phrases, and an additional pass before parsing. In this additional pass the entire source code file is scanned. Predictions are then made and the intermediate file is 'cleaned up' in such a way that the actual parser can accept the input. Arrangements must be made for the prediction information to be passed to the backend of the compiler. The degradation in the speed of compilation, though not measured, was observably negligible in our testbed.

The approach taken in this paper is quite closely coupled to the Unix/C environment. However, we believe that the general approach can be adapted to other operating systems and programming environments. A contribution of this paper is to show the potentials of source code level branch prediction.

7 Conclusion

In this paper, we proposed several heuristics for performing source level static branch prediction and evaluated their performance against large C benchmarks. A number of source level static branch prediction heuristics were proposed and studied in empirical experiments. Performing static branch prediction at the source code level has the advantage of

permitting the inspection of names under the assumption that programmers employ names (in macros and variables) that are indicative of the uses of the named data. Branch prediction can then be performed in the front end of a compiler assuming such uses. The best heuristic, i.e. SFM, is applicable 75% of the time and in the best case achieves an accuracy that is 96% that of the dynamic predictor. We argue that this is evidence that even with simple heuristics, good branch predictions can be obtained at compile time. It should be emphasized that we are not arguing that static branch prediction should replace dynamic ones. Rather, we believe that the heuristics proposed are straightforward to implement in any compiler and should be helpful to many compilers in their optimization.

Acknowledgements

I would like to thank the anonymous referees for their many useful and incisive comments.

References

[1] S. Bandyopadhyay, V. S. Begwani, and R. B. Murray. (1987) Compiling for the CRISP Microprocessor. Proc. of Compcon Spring '87. 96-100.

[2] T. Ball, and J. R. Larus. (1994) Optimally Profiling and Tracing Programs. ACM Trans. on Prog. Lang. and Systems. **16-4**. 1319-1360.

[3] T. Ball, and J. R. Larus. (1993) Branch Prediction for Free. Proc. of SIGPLAN '93 Conf. on Prog. Lang. Design and Impl. 300-313.

[4] B. Calder, D. Grunwald, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. (1995) Corpus-based Static Branch Prediction. Proc. of SIGPLAN '95 Conf. on Prog. Lang. Design and Impl. 79-92.

[5] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. (1997) Evidence-Based Static Branch Prediction using Machine Learning. ACM Trans. on Prog. Lang. and Systems. **19-1**. 188-222.

[6] F. Chow and J. Hennessy. (1990) The Priority-based Coloring Approach to Register Allocation. ACM Trans. on Prog. Lang. and Systems. **12-4**. 501-536.

[7] H. G. Cragon. (1992) *Branch Strategy Taxonomy and Performance Models.* IEEE Computer Society Press.

[8] Digital Equipment Corp. (1997) *Digital Semiconductor 21164 Alpha Microprocessor Hardware Reference Manual.*

[9] D. R. Ditzel, and H. R. McLellan. (1987) Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero. Proc. of 14th Int'l Symp. on Computer Arch. 2-9.

[10] J. A. Fisher, and S. M. Freudenberger. (1992) Predicting conditional branch directions from previous runs of a program. Proc. of 5th ASPLOS. 85-95.

[11] D. Grunwald, D. Lindsay, and B. Zorn. (1998) Static Methods in Hybrid Branch Prediction. Proc. of Int'l Conf. on Parallel Architectures and Compilation Techniques. `http://www.cs.colorado.edu/ grunwald/Papers/PACT98-StaticHybrid`.

[12] J. K. F. Lee, and A. J. Smith. (1984) Branch Prediction Strategies and Branch Target Buffer Design. IEEE Computer. **21-1**. 6-22.

[13] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donell, and J. C. Ruttenberg. (1993) The Multiflow Trace Scheduling Compiler. J. of Supercomputing. **7-1**. 51-142.

[14] C-K. Luk, and T.C. Mowry. (1996) Compiler-Based Prefetching for Recursive Data Structures. Proc. of 7th ASPLOS. 222-233.

[15] S. McFarling. (1993) Combining Branch Predictors. Digital WRL Technical Note TN-36.

[16] J. E. Smith. (1981) A Study of Branch Prediction Strategies. Proc. of 8th Int'l Symp. on Computer Architecture. 135-148.

[17] D. W. Wall. (1991) Predicting program behavior using real or estimated profiles. Proc. of SIGPLAN '91 Conf. on Prog. Lang. Design and Impl. 59-70.

[18] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison. (1994) Accurate Static Estimators for Program Optimization. Proc. of SIGPLAN '94 Conf. on Prog. Lang. Design and Impl. 85-106.

[19] D. L. Weaver, and T. Germond. (1994) *The SPARC Architecture Manual - Version 9.* Prentice-Hall.

[20] S. Weiss, and J. E. Smith. (1994) *Power and PowerPC.* Morgan Kaufmann Publishers, Inc.

[21] Y. Wu, and J. R. Larus. (1994) Static Branch Frequency and Program Profile Analysis. Proc. of MICRO-27. 1-11.