2D-Profiling: Detecting Input-Dependent Branches with a Single Input Data Set

Hyesoon Kim M. Aater Suleman Onur Mutlu Yale N. Patt

Department of Electrical and Computer Engineering
University of Texas at Austin
{hyesoon,suleman,onur,patt}@ece.utexas.edu

Abstract

Static compilers use profiling to predict run-time program behavior. Generally, this requires multiple input sets to capture wide variations in run-time behavior. This is expensive in terms of resources and compilation time. We introduce a new mechanism, 2D-profiling, which profiles with only one input set and predicts whether the result of the profile would change significantly across multiple input sets.

We use 2D-profiling to predict whether a branch's prediction accuracy varies across input sets. The key insight is that if the prediction accuracy of an individual branch varies significantly over a profiling run with one input set, then it is more likely that the prediction accuracy of that branch varies across input sets. We evaluate 2D-profiling with the SPEC CPU 2000 integer benchmarks and show that it can identify input-dependent branches accurately.

1. Introduction

Profile-guided code optimization has become essential for achieving good performance in programs. Profiling discovers new optimization opportunities in a program that are not identified by static analysis. However, profile-guided optimization is beneficial only if the program's run-time behavior is similar to its profile-time behavior. A good profiling compiler should therefore profile with representative input data which shows a wide range of program behavior [28]. To do so, compilers usually need to profile with multiple different input data sets. How a compiler should find and select input sets that can represent a wide range of program behavior still remains a very hard problem.

The predictability and bias of a branch instruction are two examples of input-dependent program characteristics. As Fisher and Freudenberger showed in [5], some branches behave similarly across input sets but others do not. We observe that, for some applications, even highly-biased (or easy-to-predict) branches for one input set can behave very differently for another input set. If the compiler aggressively optimizes the code assuming that a branch is highly-biased, the optimization could hurt performance if that branch behaves differently with another input set. Hence, input-dependent branches can significantly affect the performance of profile-guided code optimizations. Identifying such branches at compile-time would aid in increasing the quality and performance of profile-guided optimizations.

To identify input-dependent branches, a compiler needs to either profile with multiple input sets or find one representative input set. Unfortunately, profiling with multiple input sets is costly in terms of resources and compilation time, and finding a single representative input set is a very hard problem. Furthermore, there is a very large number of possible different input sets for many applications, which always leaves the possibility that the compiler may *miss* an important input set while profiling.

This paper proposes a new profiling mechanism, called *2D-profiling*, to identify input-dependent branches without having to profile with multiple input sets. A 2D-profiling compiler profiles with only *one input set* and predicts whether the result of the profile will remain similar or change significantly across input sets. In order to predict whether or not a program property is input-dependent, 2D-profiling measures *time-varying phase characteristics* of the program during the profiling run in addition to the *average program characteristics over the whole profiling run*. Previously-proposed branch profiling techniques usually use only average (aggregate) values [26] such as the average misprediction rate for a branch instruction. We find that measuring the outputs of a profiling run over time and calculating even simple statistics such as standard deviation can provide valuable information about the input-dependence characteristics of a program property.

We use 2D-profiling to profile the prediction accuracy of individual branch instructions. Our 2D-profiling algorithm predicts whether the prediction accuracy of a branch instruction will remain similar or change significantly across different input sets. The basic insight of the mechanism is that *if the prediction accuracy of a static branch changes significantly over time during the profiling run (with a single input data set), then it is more likely that the prediction accuracy of that branch is dependent on the input data set*. Similarly, 2D-profiling can also be used with edge profiling to determine whether or not the bias (taken/not-taken rate) of a branch is input-dependent.

This paper presents the 2D-profiling algorithm that is used to identify input-dependent branches. The major contribution of our paper is an *efficient profiling mechanism that identifies input-dependent branches in order to aid compiler-based branch-related optimizations, such as predicated execution*. No previous work we are aware of proposed a way of identifying input-dependent branches at compile-time using a single input data set. We show that 2D-profiling can identify the input-dependent branches in the SPEC CPU 2000 integer benchmarks by profiling with only one input set, even if the profiler uses a different and less accurate branch predictor than the target processor.

2. Input-Dependent Branches

We classify a conditional branch as input-dependent if its prediction accuracy changes by a certain threshold value across two input sets. We set this threshold to be 5% in our analysis.[1]

2.1. Importance of Input-Dependent Branches

We first explain why input-dependent branches are important in compiler optimizations. We use predicated execution via if-conversion [1] as our compiler optimization example. We describe why even a small absolute change like 5% in branch prediction accuracy can impact the performance of predicated code.

2.1.1. Importance of Input-Dependent Branches in Predicated Execution Figure 1 shows an example source code and the corresponding assembly code with branches (normal branch code - 1a) and with predication (predicated code - 1b). With normal branch code, the processor speculatively fetches and executes block B or C

[1] For example, if the prediction accuracy of a branch instruction is 80% with one input set and 85.1% with another, we consider this branch to be input-dependent since the delta, 5.1%, is greater than the threshold, 5%.

Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)
0-7695-2499-0/06 $20.00 © 2006 IEEE

based on the predicted direction of the branch in block A. When the branch is mispredicted, the processor flushes its pipeline, rolls back to the end of block A, and fetches the alternate block. With predicated code, the processor fetches both block B and block C. Instructions in blocks B and C are not executed until the predicate value (p1 in Figure 1) is resolved. Since there is no branch misprediction, there is no pipeline flush penalty. However, the processor always fetches and executes instructions from both control-flow paths.

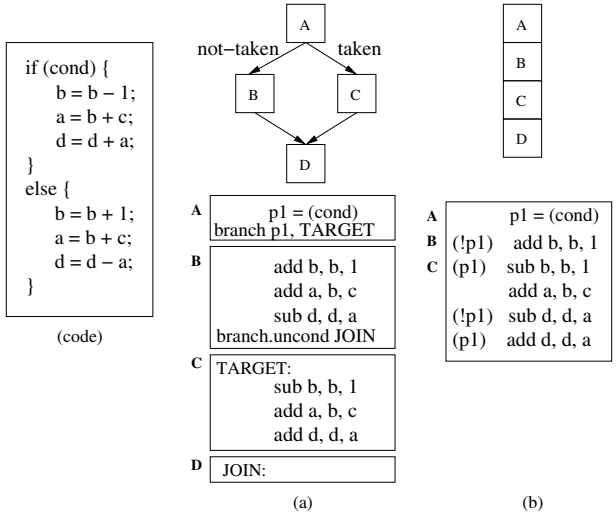

Figure 1. Source code and the corresponding assembly code for (a) normal branch code (b) predicated code

Equations (1) and (2) show the cost of normal branch code and the cost of predicated code respectively. The compiler decides whether a branch is converted into predicated code or stays as a branch based on equation (3) [21, 16]. A branch is converted into predicated code if it satisfies equation (3), otherwise it stays as a normal branch instruction.

(1) $Exec\_cost(normal\ branch) = exec\_T * P(T) + exec\_N * P(N) + misp\_penalty * P(misp)$

(2) $Exec\_cost(predicated\ code) = exec\_pred$

(3) $Exec\_cost(normal\ branch) > Exec\_cost(predicated\ code)$

$exec\_T$: Exec. time of the code when the branch under consideration is taken
$exec\_N$: Exec. time of the code when the branch under consideration is not taken
$P(case)$: The probability of the case; e.g., P(T) is the prob. that the branch is taken
$misp\_penalty$: Machine-specific branch misprediction penalty
$exec\_pred$: Execution time of the predicated code

To demonstrate how sensitive equation (3) is to the branch misprediction rate, we apply the equation to the code example shown in Figure 1. We set misp_penalty to 30 cycles, exec_T to 3 cycles, exec_N to 3 cycles, exec_pred to 5 cycles. Figure 2 displays the two equations, (1) and (2), as the branch misprediction rate is varied in the X-axis. With the given parameters, if the branch misprediction rate is less than 7%, normal branch code takes fewer cycles to execute than predicated code. If the branch misprediction rate is greater than 7%, predicated code takes fewer cycles than normal branch code. For example, if the branch misprediction rate is 9%, predicated code performs better than normal branch code. But if the misprediction rate becomes 4%, then normal branch code performs better. Hence, even a 5% absolute change in prediction accuracy can change the decision of whether or not to predicate a branch.

In this example, if the compiler can identify whether or not the branch is input-dependent, then it can make a better optimization decision, especially for those branches with misprediction rates close to

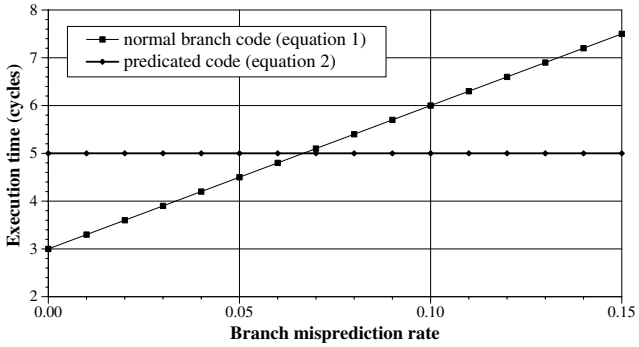

Figure 2. Execution time of predicated code and non-predicated code vs. branch misprediction rate

7%. If the branch misprediction rate is greater than 7% and the compiler correctly identifies that the branch is not input-dependent, then the compiler can safely convert the branch into predicated code to obtain better performance. However, if the compiler correctly identifies that the branch is input-dependent, it would know that predicating the branch may cause performance loss with different input sets.[2] Therefore, in the latter case, the compiler is more apt to leave the decision of how to handle the branch to a dynamic optimizer [6] or to convert the branch into a *wish branch* so that the hardware decides whether or not to use predicated execution for the branch [10].

2.1.2. Other Compiler Optimizations that Can Benefit from the Identification of Input-Dependent Branches

Aside from predicated execution, identifying input-dependent branches can be useful for other branch-based compiler optimizations. For example trace/superblock identification/scheduling and code-reordering based optimizations rely on the same program path to be frequently executed (i.e., hot) across different input sets [4, 22, 8, 18]. If a branch on a hot path identified with the profiling input set(s) is input-dependent, that path may not be a hot path with another input set. Hence, aggressive optimizations performed on that path may hurt performance with a different input set. Identifying the input-dependent branches can aid a compiler in deciding whether or not to perform aggressive code optimizations on a given hot path.

2.2. Characteristics of Input-Dependent Branches

Figure 3 shows the dynamic and static fraction of conditional branches that show input-dependent behavior. Train and reference input sets for the SPEC CPU 2000 integer benchmarks were used to identify the input-dependent branches. Our baseline branch predictor is a 4KB gshare branch predictor. The dynamic fraction is obtained by dividing the number of dynamic instances of all input-dependent branches by the number of dynamic instances of all branch instructions, using the reference input set.[3] The benchmarks are sorted by the dynamic fraction of input-dependent branches, in descending order from left to right. The data shows that there are many branches

[2]Previous work [10] showed that the same predicated code binary, when executed on an Itanium-II processor, can lead to performance improvement or performance loss depending on the input set.

[3]Note that input-dependence is a property of a *static* branch. Input-dependence cannot be defined for a dynamic instance of a branch, since the dynamic instance of a branch is executed only once. We show the dynamic fraction of input-dependent branches in Figure 3 to provide insight into the execution frequency of input-dependent branches. All other results in this paper are based on *static branches*.

that show more than 5% absolute change in prediction accuracy between the train and reference input sets. More than 10% of the static branches in bzip2, gzip, twolf, gap, crafty, and gcc are input-dependent branches. Note that this data is obtained using only two input sets to define the set of input-dependent branches. In Section 5.2, we will discuss how increasing the number of input sets affects the set of input-dependent branches.

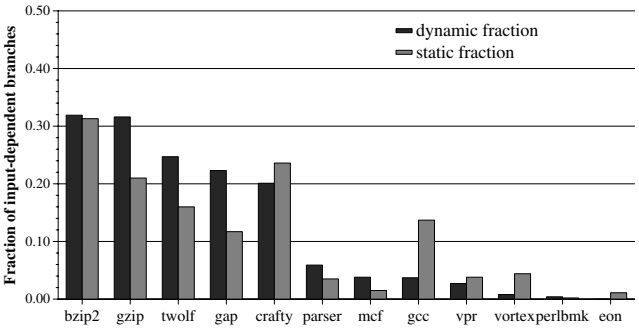

Figure 3. The fraction of input-dependent branches (using train and ref input sets)

One might think that (1) all input-dependent branches are hard-to-predict, (2) all hard-to-predict branches are input-dependent, or (3) a benchmark has input-dependent branches only if the overall program branch prediction accuracy changes across input sets. The following results show that these conjectures are not true. Therefore, we cannot identify input-dependent branches by only identifying hard-to-predict branches or observing the change in the overall branch prediction accuracy across input sets.

Figure 4 shows whether or not all input-dependent branches are hard-to-predict. This figure displays the distribution of all input-dependent branches based on their prediction accuracy. Input-dependent branches are classified into six categories based on their prediction accuracy using the reference input set. The data shows that a sizeable fraction of input-dependent branches are actually relatively easy-to-predict (i.e., have a prediction accuracy of greater than 95%) in many of the benchmarks. Even the fraction of input-dependent branches with a prediction accuracy greater than 99% -which is a very strict accuracy threshold- is significant for gap (19%), vortex (8%), gcc (7%), crafty (6%), twolf (4%), and parser (4%). Hence, not all input-dependent branches are hard-to-predict. There are many input-dependent branches that are relatively easy-to-predict.

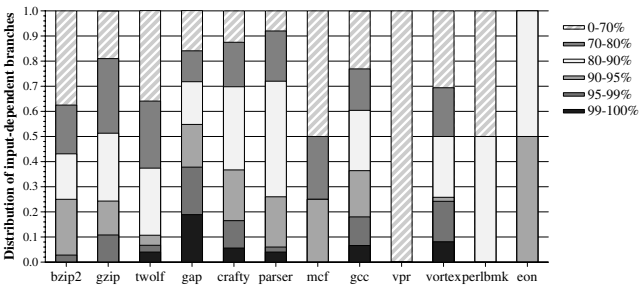

Figure 4. The distribution of input-dependent branches based on their branch prediction accuracy

Figure 5 shows whether or not all hard-to-predict branches are input-dependent. We classify all branches into six categories based on their prediction accuracy. The figure presents the fraction of input-dependent branches in each category. For example, in bzip2, 75% of branches with a prediction accuracy lower than 70% are input-dependent and only 10% of branches with a prediction accuracy between 95-99% are input-dependent. In general, the fraction of input-dependent branches increases as the prediction accuracy decreases. Thus, branches with a low prediction accuracy are more likely to be input-dependent. However, many branches with a low prediction accuracy are actually not input-dependent. For example, in gzip only half of the branches with a prediction accuracy lower than 70% are input-dependent. Therefore, we cannot identify input-dependent branches by only identifying hard-to-predict branches.

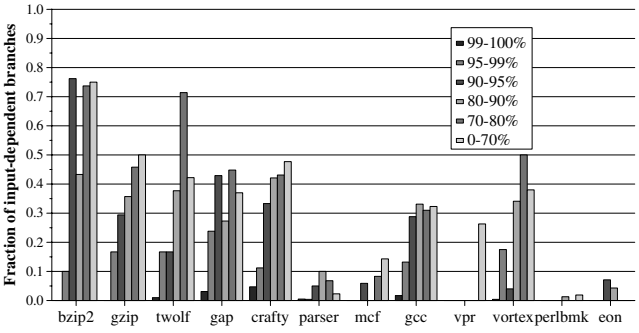

Figure 5. The fraction of input-dependent branches in different prediction accuracy categories

We also measure the overall branch misprediction rate to examine the correlation between the overall branch misprediction rate difference across input sets and the fraction of input-dependent branches. Table 1 shows the average branch misprediction rate for each input set. Some benchmarks that have a small difference in the overall branch misprediction rate between the two input sets, such as eon and perlbmk, also have a small fraction of input-dependent branches (as can be seen in Figure 3). For these benchmarks, profiling with multiple input sets and computing the average branch prediction accuracy would correctly indicate that there are not many input-dependent branches. In contrast, even though twolf and crafty have a small difference in overall branch prediction accuracy across the two input sets, they have a high number of input-dependent branches. So, just comparing the overall branch misprediction rate across input sets does not provide enough information to judge whether or not a benchmark has many input-dependent branches. Only comparing the prediction accuracy of each individual branch across input sets would identify input-dependent branches. However, comparing the branch prediction accuracies for each individual branch for multiple input sets would significantly increase the profiling overhead. To reduce the profiling overhead of identifying input-dependent branches, we propose 2D-profiling to discover input-dependent branches *without requiring multiple input sets for profiling*.

2.3. Examples of Input-Dependent Branches

What kind of branches are sometimes easy to predict and sometimes hard to predict? We provide two examples to show the code structures causing input-dependent branch behavior.

One example of an input-dependent branch is a branch that checks data types. A branch in the gap benchmark, which is shown on line 5 in Figure 6, checks whether or not the data type of a variable (hd) is an integer. The program executes different functions depending on

Table 1. Average branch misprediction rates of the evaluated programs (%)

Input Data Set	bzip2	gzip	twolf	gap	crafty	parser	mcf	gcc	vpr	vortex	perlbmk	eon
train	1.9	7.5	16.4	5.7	12.4	9.1	7.8	7.3	11.2	0.8	5.1	12.2
reference	8.3	6.5	15.7	3.9	11.8	8.9	6.6	2.4	11.1	0.4	5.1	12.1

the data type of the variable. The misprediction rate of this branch is 10% with the train input set, but it is 42% with the reference input set. With the train input set, the variable is an integer for 90% of the time, so the taken rate of the branch is 90%. Hence, even a simple predictor achieves 90% accuracy for that branch. In contrast, with the reference input set, approximately half of the time the variable is of non-integer type and therefore the branch misprediction rate increases to 42%. Gap is a math program that can compute using different types of data. It uses a non-integer data type to store values greater than 2^{30}. The reference input set contains a large fraction of values that are greater than 2^{30}, which are stored in variables of a non-integer data type. In contrast, most input data values in the train input set are smaller than 2^{30} and they are stored as integers. This results in very different behavior across input sets for the branch that checks the type of the input data.

```
1 :TypHandle Sum ( TypHandle hd ) {
2 :   // initialize hdL and hdR using hd
3 :   // ...
4 :   // input-dependent br. checks the type of hd (line 5)
5 :   if ( (long)hdL & (long)hdR & T_INT ) {
6 :        // use integer sum function for integer type
7 :        result = (long)hdL + (long)hdR - T_INT;
8 :        ov = (int)result;
9 :        if ( ((ov << 1) >> 1) == ov )
10:           return (TypHandle) ov; // return integer sum
11:   }
12:
13:  // call a special SUM function for non-integer type
14:  return SUM( hdL, hdR );
15:}
```

Figure 6. An input-dependent branch from gap

The prediction behavior of a loop branch is strongly dependent on what determines the number of loop iterations. If the loop iteration count is determined by input data, the prediction behavior of the loop branch is dependent on the input set. If the iteration count is a large number, then the branch is easy to predict, whereas if the iteration count is small, the branch can be hard to predict. For example, some loops in the gzip benchmark execute for different number of iterations depending on the compression level, which is specified as a parameter to the program. Figure 7 shows an example. The branch on line 25 is a loop exit branch. The exit condition is defined on line 18 using *pack_level* and *max_chain*. *pack_level* is the compression level and *max_chain* is the value that determines the number of loop iterations. *max_chain* has a higher value at higher compression levels, as shown on lines 9-13. At compression level 1, the loop iterates 4 times and the prediction accuracy of the branch is 75% (3/4) without a specialized loop predictor. But, at compression level 9, the loop iterates 4096 times, so the prediction accuracy of the branch is very close to 100% (4095/4096). Therefore, the branch is input-dependent on the input parameter that specifies the compression level.

3. 2D-Profiling

3.1. The Basic Idea

The purpose of 2D-profiling is to identify the input dependence behavior of a program by profiling only one input set. Most current profiling compilers collect only average data, such as the average bias (taken rate) of a branch, for the complete profiling run of a program and use it for optimization. We propose the use of information about

```
1: typedef struct config {
2:    int good_length;
3:    int max_lazy;
4:    int nice_length;
5:    int max_chain;
6: } config;
7:
8: local config config_table[10] = {
9:   /* 1 */ {4,    4,   8,    4}, // min compression level
10:  // ...
11:  /* 4 */ {4,    4,  16,   16},
12:  // ...
13:  /* 9 */ {32, 258, 258, 4096} // max compression level
14: };
15:
16: /*** Initialization code begin ***/
17:   // max chain length is read from the config table
18:   max_chain_length = config_table[pack_level].max_chain;
19:   unsigned chain_length = max_chain_length;
19: /*** Initialization code end ***/
20:
21: do {
22:    // ...
23:    // input-dependent loop exit branch (line 25)
24: } while ((cur_match = prev[cur_match & WMASK]) > limit
25:            && --chain_length != 0);
```

Figure 7. An input-dependent loop exit branch from gzip

the time-varying phase behavior of the program during the profiling run to predict the input-dependence behavior of the program.

In this paper, we demonstrate the use of 2D-profiling to identify input-dependent branches. The same concept can also be applied to other profiling mechanisms such as edge profiling. The insight behind our idea is that if the prediction accuracy of a branch changes significantly during the profiling run, then its prediction accuracy is also likely to change across input sets. We call our mechanism 2D-profiling because it collects profile information in two dimensions: *prediction accuracy for a branch* over *time*.

Figure 8 shows the prediction accuracy of two branches over the execution time of the gap benchmark. The prediction accuracy for the input-independent branch on the right is stable throughout the execution of the program, whereas the prediction accuracy for the input-dependent branch on the left varies significantly over time. 2D-profiling will detect this variation and classify the branch on the left as an input-dependent branch. The branch on the right will be classified as an input-independent branch even though its prediction accuracy is very low (around 58%).

3.2. Implementation

We describe a simple and efficient implementation of 2D-profiling. Figure 9 shows the pseudo-code of the implementation.

3.2.1. Overview In order to detect the time-varying phase behavior of a branch, the profiler needs to record the prediction accuracy for all static branches at fixed intervals during a profiling run. We refer to each interval as a slice. A function executed at the end of each slice (Figure 9b) collects data needed for input-dependence tests that are performed at the end of the profiling run. The data that is needed for each branch is shown in Figure 9a and described in Section 3.2.3.

Three input-dependence tests are performed for each branch at the end of the profiling run (Figure 9c). The purpose of these tests is to determine whether the branch has shown phase behavior during the

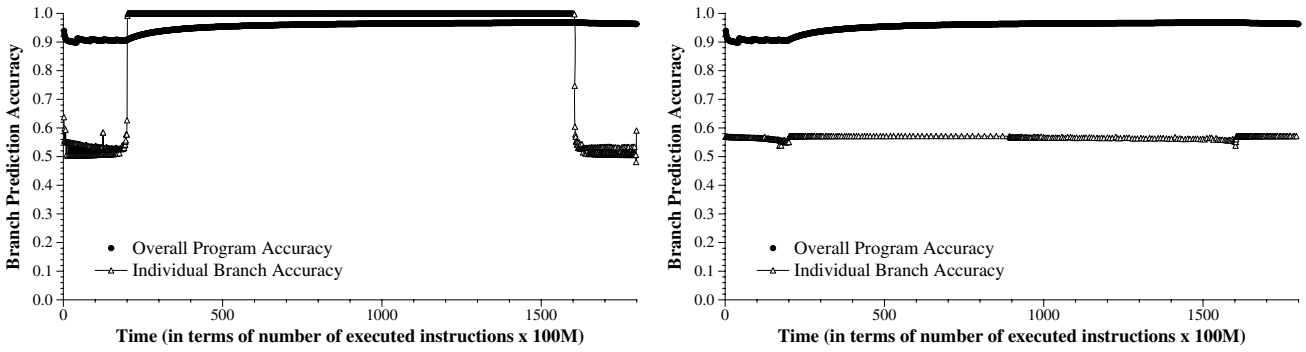

Figure 8. The time-varying phase behavior of the prediction accuracy of an input-dependent branch (left) vs. an input-independent branch (right) from the gap benchmark

profiling run. If the branch passes the tests, it is identified as input-dependent.

3.2.2. Input-dependence Tests Three tests are used to decide input-dependence: MEAN-test, STD-test, and PAM-test. The MEAN-test and the STD-test are used to identify branches that are good candidates for input-dependent branches. The PAM-test is then applied on these candidate branches as a filter. A branch is identified as input-dependent if it passes either the MEAN-test or the STD-test, along with the PAM-test (lines 26-28 in Figure 9c).

MEAN signifies the arithmetic mean of the branch prediction accuracies of a branch over all slices. The purpose of the MEAN-test is to find branches with a low average prediction accuracy because highly-mispredicted branches are more likely to be input-dependent, as shown in Figure 5. This test computes the mean of the prediction accuracy data collected for all the slices for a branch. If the mean prediction accuracy of the branch is less than a threshold (*MEAN_th*), the branch passes the MEAN-test.

STD stands for standard deviation. The motivation for the STD-test is to find branches that show a significant variation in prediction accuracy over time. This test helps especially to identify branches that have a high average prediction accuracy but show large variations in the prediction accuracy. STD-test computes the standard deviation of the prediction accuracy data collected for all the slices. If the standard deviation is greater than a threshold (*STD_th*), the branch passes the STD-test.

PAM stands for Points Above Mean. The purpose of the PAM-test is to filter out the branches that do not show a stable phase behavior or that show phase-like behavior due to outlier slices. The test is most useful when applied in combination with the MEAN-test or STD-test or both. When used with the MEAN-test, the PAM-test can filter out those branches that pass the MEAN-test with only a low mean prediction accuracy but no significant variation in prediction accuracy (e.g., the branch shown in the right graph of Figure 8). When applied with the STD-test, the PAM-test filters out those branches that pass the STD-test only due to contributions from outlier points. The test computes the fraction of slices for which the prediction accuracy of the branch is greater than the mean prediction accuracy of the branch. If this fraction is between *PAM_th* and *(1 - PAM_th)* thresholds, indicating that there is enough variation in prediction accuracy, then the branch passes the PAM-test.[4]

[4]In statistical terminology, the PAM-test is a *two tailed test* [19] that eliminates the influence of outliers in the prediction accuracy distribution; outliers that are both above and below the mean (See Figure 9c, lines 21-25).

3.2.3. Implementation Details and Storage Overhead Each of the three tests, when performed on a branch, need to compute a statistic of the data for all slices. The overhead of 2D-profiling would be very high in terms of the memory requirements, if the profiler stored the prediction accuracy for each branch for all slices. Therefore, we maintain just enough information that the three tests can use to compute their respective statistics. In our implementation, only seven variables need to be stored for each branch. The list of the variables and their descriptions are provided in Figure 9a.

For the MEAN-test, the profiler needs to maintain only the *Sum of Prediction Accuracies over all slices* (SPA) and the *total Number of slices* (N). The number of slices needs to be stored separately for each branch (instead of using just one global count for all branches) because the profiler disregards the slice data for a branch if the number of times the branch was executed in the slice is less than a threshold value ($exec\_threshold$). This check is performed to avoid noise in the branch prediction accuracy data due to infrequent occurrences and to reduce the effects of predictor warm-up.

For the STD-test, to compute the standard deviation of all prediction accuracies of the branch over all slices, the profiler maintains the *Sum of Squares of Prediction Accuracies over all slices* (SSPA), in addition to SPA and N.

For the PAM-test, the profiler maintains the *Number of slices for which the Prediction Accuracy is above the Mean accuracy* (NPAM), in addition to N, to compute the fraction of points above mean.[5]

The profiler also needs two variables to calculate the prediction accuracy of the branch in a slice: the number of times the branch is executed in the slice ($exec\_counter$) and the number of times the branch is predicted correctly in the slice ($predict\_counter$). These variables are updated every time the branch is executed. At the end of each slice, they are first used to update N, SPA, SSPA, and NPAM and then reset.

Another variable is needed to implement a simple 2-TAP FIR filter [20]. We added this filter to reduce the high-frequency sampling noise in the data sampled during execution. The filter is a simple averaging low-pass filter, which replaces the prediction accuracy of the branch in the current slice with the average of the prediction ac-

[5]In order for the profiler to compute the *exact* number of points above mean, it needs to know the mean prediction accuracy for a branch. Since the mean accuracy can only be computed at the end of the program, the profiler needs to store the prediction accuracy for all slices for the branch and compute NPAM at the end of the profiling run by comparing each slice's prediction accuracy to the mean accuracy, which is an expensive solution. A cheaper approximation is to utilize a running mean accuracy value for each branch, which can be computed at the end of every slice (See Figure 9b, lines 7-9).

Name	Description	Purpose	Update frequency
N	Number of slices	All tests	At the end of every slice
SPA	Sum of Prediction Accuracies across all slices	MEAN- and STD-test	At the end of every slice
SSPA	Sum of Squares of Prediction Accuracies across all slices	STD-test	At the end of every slice
NPAM	Number of Prediction accuracies Above the running Mean	PAM-test	At the end of every slice
exec_counter	Number of times the branch was executed in a slice	Prediction Accuracy in Slice	Every time the branch is executed
predict_counter	Number of times the branch was predicted correctly in a slice	Prediction Accuracy in Slice	Every time the branch is predicted correctly
LPA	Last Prediction Accuracy for the branch (from previous slice)	FIR filter to remove noise	At the end of every slice

(a) Variables stored for each branch instruction

```
At the end of every slice, for each branch:

1:  if ( exec_counter > exec_threshold ) {      // Collect data if the branch was executed enough times in the slice
2:     N++;                                     // Increment slice count by one
3:     pred_acc = predict_counter / exec_counter;        // Compute prediction accuracy in this slice
4:     filtered_pred_acc = (pred_acc + LPA) / 2;         // FIR filter averages current accuracy with last accuracy
5:     SPA += filtered_pred_acc;                         // Update SPA (Sum of Prediction Accuracies)
6:     SSPA += (filtered_pred_acc * filtered_pred_acc);  // Update SSPA (Sum of Squares of Prediction Accuracies)
7:     running_avg_pred_acc = SPA / N;                   // Compute a running average of prediction accuracy (for NPAM)
8:     if (filtered_pred_acc > running_avg_pred_acc)
9:       NPAM++;                                 // Update NPAM (Number of Points Above Mean)
10:    LPA = filtered_pred_acc;                  // Update LPA (Last Prediction Accuracy)
11: }
12: exec_counter = predict_counter = 0;          // Reset temporary counters used during each slice
```

(b) Method executed for each branch at the end of every slice

```
13:   MEAN-test(SPA, N)
14:      Compute mean (i.e., the average prediction accuracy for the branch)
15:      if (mean is less than MEAN_th) return Pass
16:      else return Fail

17:   STD-test(SPA, SSPA, N)
18:      Compute standard deviation (of the prediction accuracies for all slices)
19:      if (standard deviation is greater than STD_th) return Pass
20:      else return Fail

21:   PAM-test(NPAM, N)
22:      Compute the fraction of points above mean
23:      if (the fraction of points above mean is less than PAM_th) return Fail
24:      else if (the fraction of points above mean is greater than (1 - PAM_th)) return Fail
25:      else return Pass

At the end of the profiling run, for each branch:

26:   if ( MEAN-test Passed OR STD-test Passed )
27:       if ( PAM-test Passed )
28:           Add branch to the set of input-dependent branches
```

(c) Methods executed for each branch at the end of the profiling run

Figure 9. Pseudo-code for the 2D-profiling algorithm to detect input-dependent branches

curacies of the branch in the current slice and in the previous slice (Figure 9b, line 4). To implement the filter, the prediction accuracy of the branch in the previous slice is stored separately in a variable called LPA (*Last Prediction Accuracy*).

3.2.4. Computation Overhead When a branch is executed, the profiler needs to increment the execution count and update the correct prediction count for that branch. The branch predictor outcome for the branch can be obtained with hardware support [3] or by implementing the branch predictor in software in the profiler. In addition, a simple method is executed for every branch at the end of every slice (Figure 9b, lines 1-12). Last, the pseudo-code shown in lines 13-28 of Figure 9c is executed only once for each branch, at the end of the profiling run, to compute the outcome of the input-dependence tests. We quantify the execution time overhead of 2D-profiling in Section 5.4.

4. Experimental Methodology

We used the SPEC CPU 2000 integer benchmarks to evaluate 2D-profiling. The programs were compiled for the x86 ISA using the gcc-3.3.3 compiler with the `-O3` optimization flag. Table 2 shows the baseline input sets and relevant information for each benchmark. In our baseline experiments, we define input-dependent branches using the train and reference input sets. We use only the train input set in the profiling runs.

We used Pin [13], a binary instrumentation tool for x86 binaries, to instrument the programs and implement the 2D-profiling algorithm. All programs with different input sets were run until the end using Pin. Our baseline branch predictor is a 4KB (14-bit history) gshare branch predictor [17]. We also show results with a 16KB (457-entry, 36-bit history) perceptron predictor [9] in Section 5.3.

4.1. 2D-Profiling Setup

In our experiments, the slice size was fixed at 15 million branches. $exec\_threshold$ was set to 1000 to avoid extraneous data points. We experimented with different threshold values for $MEAN\_th$, STD_th, and PAM_th. $MEAN\_th$ is set to the *average overall branch prediction accuracy of the benchmark*, which is determined at the end of the profiling run for each benchmark. STD_th is set to 0.04 and PAM_th is set to 0.1.

We evaluated the sensitivity of 2D-profiling to the threshold value

Table 2. Evaluated benchmarks and input sets and their characteristics (dynamic instruction and branch counts)

Benchmark	Reference input set			Train input set			Static cond. branch counts	
	input	inst.count	cond. br. count	input	inst.count	cond br. count	input-dependent	total branches
bzip2	input.source	317B	43B	input.compressed	170B	28B	72	230
gzip	input.source	54B	10B	input.combined	38B	6.5B	37	176
twolf	ref	287B	40B	train	10B	1.5B	75	468
gap	ref.in	226B	21B	train.in	6.9B	1B	106	907
crafty	ref/crafty.in	215B	21B	train/crafty.in	32B	3B	248	1050
parser	ref/ref.in	330B	44B	train/train.in	8.2B	1.1B	50	1420
mcf	ref/inp.in	49B	11B	train/inp.in	7.1B	1.5B	4	273
gcc	166.i	19B	4B	cp-decl.i	3.1B	550M	678	4964
vpr	ref/net.in	110B	14B	train/net.in	14B	1.8B	5	130
vortex	ref/lendian1.raw	99B	15B	train/lendian.raw	14B	1.7B	62	1421
perlbmk	diffmail.pl param1	33B	4B	diffmail.pl param2	29B	4B	2	1260
eon	ref/chair.control.cook	110B	8B	train/chair.control.cook	2.4B	164M	2	181

Table 3. Metrics used to evaluate 2D-profiling

Metric	Definition
COV-dep	(the number of branches correctly identified to be input-dependent) / (the number of input-dependent branches)
ACC-dep	(the number of branches correctly identified to be input-dependent) / (the number of branches identified to be input-dependent)
COV-indep	(the number of branches correctly identified to be input-independent) / (the number of input-independent branches)
ACC-indep	(the number of branches correctly identified to be input-independent) / (the number of branches identified to be input-independent)

used to define input-dependent branches and the threshold values used in the 2D-profiling algorithm. The results of these experiments, along with more detailed per-benchmark results, are provided in an extended version of this paper [11].

4.2. Evaluation Metrics

We use four different metrics to evaluate the effectiveness of 2D-profiling: coverage of input-dependent branches (COV-dep), accuracy for input-dependent branches (ACC-dep), coverage of input-independent branches (COV-indep), accuracy for input-independent branches (ACC-indep). We define these metrics in Table 3.

Input-independent branches are those branches that are not input-dependent (i.e., input-independent branches have less than 5% absolute difference in branch prediction accuracy between the train and reference input sets). Table 2 shows the number of input-dependent branches and the total number of branches for each benchmark. We use coverage and accuracy metrics for input-independent branches in addition to metrics for input-dependent branches because identifying input-independent branches could also be important, depending on the optimization performed by the compiler. For example, if the compiler accurately predicts that a branch is input-independent, then it can safely decide to predicate it if other constraints for predicating the branch are satisfied.

4.3. Evaluations with More Than Two Input Sets

We also evaluated our mechanism using more than two input sets in Section 5.2. The extra input sets used in these experiments are described in Table 4. The reduced input sets were obtained from the MinneSPEC suite [12]. The additional inputs for the crafty benchmark, a chess program that computes the set of possible moves given an initial chess board layout, were constructed by modifying the initial layout of the chess board. Two inputs for the gap benchmark were provided by the programmers of gap.

Table 4 also shows the number of branches that are identified as input-dependent when each input set is considered together with the train input set. Note the difference in input-dependent branches when different input sets are considered. For example, for gcc only 9 branches are classified as input-dependent when we consider input sets train and ext-6. In contrast, 821 branches are classified as input-dependent when we consider input sets train and ext-5. Some input sets are similar to each other, so a branch may not show a significant difference between its prediction accuracies for those two similar input sets. Considering multiple different input sets would therefore increase the likelihood of exercising the input-dependent behavior of branches and accurately defining the target set of input-dependent branches for the 2D-profiling algorithm.

5. Results

5.1. Results with Two Input Sets (ref and train)

Figure 10 shows the coverage and accuracy of 2D-profiling for input-dependent branches and input-independent branches when two input sets (ref and train) are used to define the target set of input-dependent branches. 2D-profiling has low accuracy for input-dependent branches in benchmarks where the number of input-dependent branches is very small, such as mcf, vpr, perlbmk and eon.[6] However, 2D-profiling accurately identifies input-independent branches for these benchmarks. This result shows that 2D-profiling can distinguish between input-dependent branches and input-independent branches. In general, 2D-profiling has high (more than 65%) accuracy and coverage in identifying input-independent branches.

The accuracy of 2D-profiling for input-dependent branches (ACC-dep) is between 28%-54% for the five benchmarks (bzip2, gzip, twolf, gap, crafty, gcc) that have the highest static fraction of input-dependent branches. While these numbers may seem low, they are actually an artifact of using only two input sets to define the set of input-dependent branches. In the next section, we show that ACC-dep significantly increases (beyond 70%) as we consider more and more input sets.

5.2. Results with More Than Two Input Sets

So far, we have defined input-dependent branches using only the train and reference input sets. However, using two input sets is not enough to pinpoint *all* input-dependent branches. An actually input-dependent branch may not show a more than 5% absolute change in

[6] Note that, if the number of input-dependent branches is very small, ACC-dep and COV-dep are not reliable metrics since they can be inflated or deflated. For example, if there is only one input-dependent branch and 2D-profiling identifies 4 (including that one), ACC-dep is only 25% and COV-dep is 100%.

Table 4. Extra input sets and their characteristics (used in experiments with more than two input sets)

benchmark	input name	input description	inst. count	branch count	branch mispred rate (%)		input-dep. branches (w.r.t. train)	
					4KB-gshare	16KB-percep.	4KB-gshare	16KB-percep.
bzip2	ext-1	input.graphic (spec)	431B	62B	4.7	4.5	47	47
bzip2	ext-2	166.s out (spec gcc)	380B	50B	6.0	5.0	54	55
bzip2	ext-3	an 11MB text file	405B	45B	6.2	5.2	40	34
bzip2	ext-4	a 3.8MB video file	679B	47B	7.8	6.6	42	30
gzip	ext-1	input.log (spec)	28B	4.9B	4.9	3.9	63	51
gzip	ext-2	input.graphic (spec)	72B	10.5B	10.0	7.5	38	28
gzip	ext-3	input.random (spec)	58B	9.6B	6.7	6.1	22	18
gzip	ext-4	input.program (spec)	104B	9.5B	13.4	11.3	43	38
gzip	ext-5	166.i (spec gcc)	35B	6.7B	4.9	4.1	53	43
gzip	ext-6	an 11MB text file	69B	15B	3.8	2.9	62	42
twolf	ext-1	large reduced input	755M	97M	21.2	16.0	53	53
twolf	ext-2	medium reduced input	217M	27M	21.4	14.4	100	78
twolf	ext-3	modified ref input	361M	44M	20.9	14.1	64	38
twolf	ext-4	small reduced input	77M	10M	20.3	11.5	128	82
gap	ext-1	Smith Normal Form	977M	137M	4.2	0.8	106	50
gap	ext-2	groups	173M	23M	5.8	4.1	81	51
gap	ext-3	medium reduced input	303M	37M	6.8	1.1	112	48
gap	ext-4	modified ref input	532M	82M	6.9	4.5	117	155
crafty	ext-1	modified ref input	628M	57M	10.7	6.3	240	261
crafty	ext-2	modified ref input	932M	86M	12.6	6.1	294	270
crafty	ext-3	modified ref input	1.1B	99M	11.8	6.3	235	261
crafty	ext-4	modified ref input	2.5B	237M	14.8	7.8	325	250
crafty	ext-5	modified ref input	148M	13M	11.9	7.0	245	233
crafty	ext-6	modified ref input	594M	46M	13.3	6.6	291	282
gcc	ext-1	small reduced input	70M	10M	9.3	7.7	727	304
gcc	ext-2	jump.i	180M	27M	9.8	9.2	804	402
gcc	ext-3	emit-rtl.i	309M	53M	10.1	8.8	747	370
gcc	ext-4	dbxout.i	361M	61M	8.6	5.5	484	43
gcc	ext-5	medium reduced input	382M	64M	9.4	8.3	821	288
gcc	ext-6	large reduced input	31B	585M	7.4	4.6	9	1

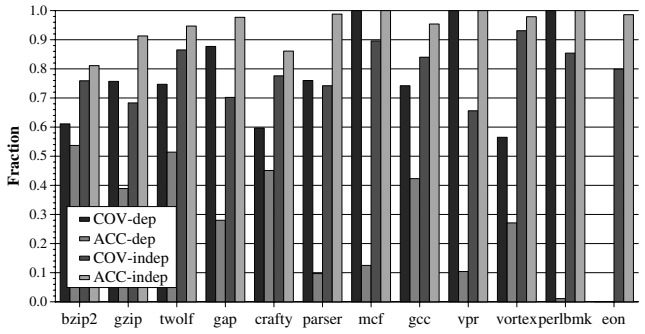

Figure 10. 2D-profiling coverage and accuracy with two input sets

overall prediction accuracy between the two input sets because the condition the branch is dependent on may be similar in both input sets. Such a branch may start showing significant changes in prediction accuracy if more (and different) input sets are considered.

We study how the set of input-dependent branches changes when more than two input sets are considered. Table 4 shows the input sets used in this section. We tested six benchmarks, bzip2, gzip, twolf, gap, crafty, and gcc, where more than 10% of the static branches are input-dependent (using train and ref input sets). We define a branch to be input-dependent if its prediction accuracy changes by more than 5% with another input set when compared to its prediction accuracy with the train input set. As we consider more and more input sets, we would expect the set of input-dependent branches to grow.

Figure 11 shows the fraction of input-dependent branches with more than two input sets. Base is the set of input-dependent branches obtained using only the train and reference input sets. Ext1 is the set of input-dependent branches obtained using the ext-1 input set and the train input set. Base-ext1 is the union of the two sets, base and ext1. Similarly, base-ext1-3 is the union of four sets (base, ext1, ext2, ext3) of input-dependent branches. The fraction of input-dependent branches monotonically increases as more and more input sets are used. For example, when we only use train and reference input sets (base) for gcc, only 14% of the branches are input-dependent. However, if we use all evaluated input sets (base-ext1-6), 33% of all branches in gcc are input-dependent. Hence, the number of input sets used significantly changes the set of branches that are defined to be input-dependent.

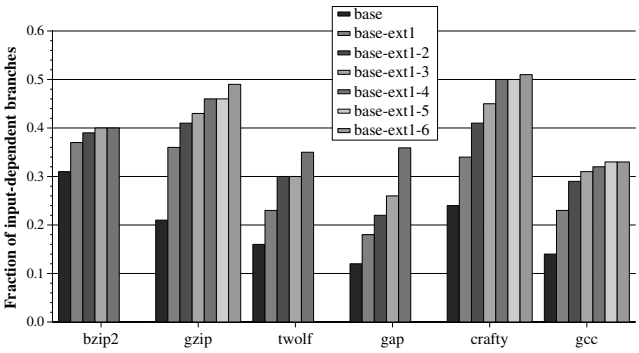

Figure 11. Fraction of input-dependent branches when more than two input sets are used

Considering a larger number of input sets would also change the evaluated performance of 2D-profiling because it changes the target set of input-dependent branches that 2D-profiling aims to identify. In fact, considering a larger number of input sets is a more fair way of evaluating 2D-profiling. For example, suppose that 2D-profiling identifies a branch as input-dependent. That branch may not actually be defined as input-dependent when only two input sets are used to define the set of input-dependent branches. In contrast, when more input sets are used, the branch may be classified as input-dependent.

Hence, the fact that we use only two input sets to define the target set of input-dependent branches may artificially penalize the accuracy of 2D-profiling: even though 2D-profiling *correctly* predicts that a branch is input-dependent, the branch may *not* show input-dependent behavior when only two input sets are used.

Figure 12 shows the coverage and accuracy, averaged over the six benchmarks,[7] of the 2D-profiling mechanism when more than two input sets are used. As the number of input sets increases, the coverage of input-dependent branches drops by a small amount, but the accuracy for input-dependent branches increases significantly. Figure 13 shows the coverage and accuracy for each benchmark when the maximum number of available input sets are used. ACC-dep is higher than 70% for every benchmark. Thus, 2D-profiling is actually accurate at identifying input-dependent branches; it just takes many input sets to trigger the input-dependent behavior. We would expect the accuracy of 2D-profiling to increase further when more input sets are taken into account. However, there is no limit to the number of different input sets that can be used for many benchmarks, which makes the *exact* determination of input-dependent branches and an *exact* evaluation of 2D-profiling challenging tasks.

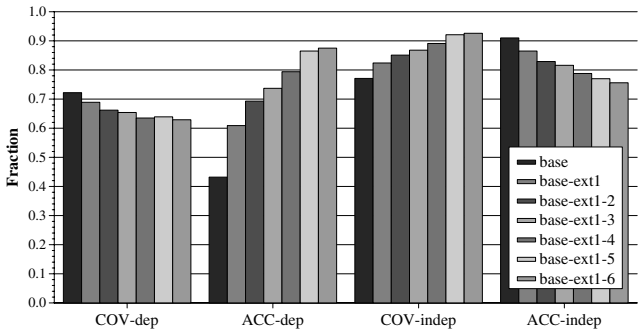

Figure 12. 2D-profiling coverage and accuracy when more than two input sets are used

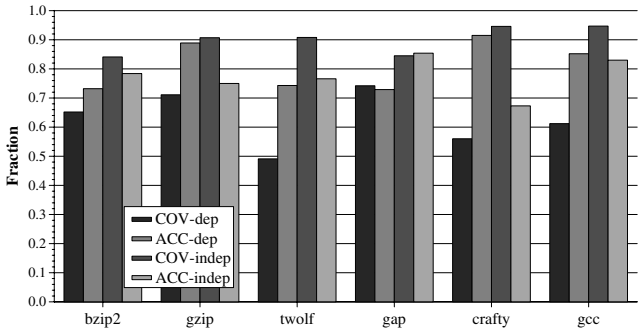

Figure 13. 2D-profiling coverage and accuracy when the maximum number of input sets are used for each benchmark

5.3. Using a Different Branch Predictor for Profiling

We also evaluate 2D-profiling when the profiler and the target processor use different branch predictors. The purpose of this experiment is to examine whether the 2D-profiling mechanism can identify input-dependent branches when the profiling mechanism uses a different branch predictor than the target machine. This is useful because (1) the 2D-profiling mechanism might not have information about the branch predictor on which the profiled application will eventually be run and (2) the 2D-profiling mechanism might want to use a simple, small branch predictor to reduce the overhead of simulating a large and complicated branch predictor in software (if hardware support does not exist for the profiler).

We use a 16KB perceptron branch predictor [9] for the target machine's branch predictor. 2D-profiling uses a 4KB gshare predictor as in previous sections. Note that the target machine's branch predictor defines the set of input-dependent branches. Table 4 shows the number of input-dependent branches and the branch misprediction rate for both predictors. We evaluate the effect of using a different branch predictor using more than two input sets to define the set of input-dependent branches, similar to Section 5.2. Figure 14 shows the fraction of input-dependent branches with the perceptron predictor. Similar to the results for the gshare predictor (Figure 11), as we consider more input sets, the number of input-dependent branches increases with the perceptron predictor.

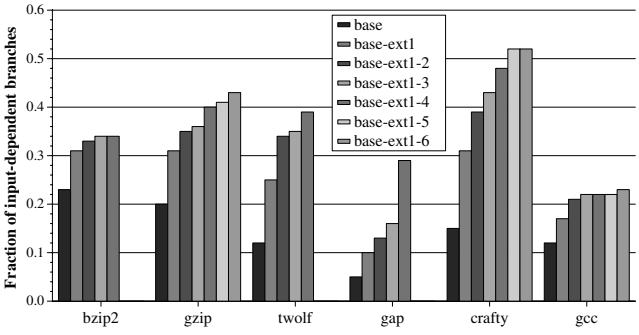

Figure 14. Fraction of input-dependent branches when the target machine uses a perceptron branch predictor

Figure 15 shows the coverage and accuracy of 2D-profiling when the profiler uses a different branch predictor than the target machine. Remember that the 2D-profiling mechanism profiles only with the train input set to identify input-dependent branches. Comparing Figure 15 to Figure 13, ACC-dep drops when the target machine has a different predictor than the profiler. However, 2D-profiling still achieves relatively high coverage and accuracy for both input-dependent and input-independent branches in most of the benchmarks except gap, even when it uses a smaller and less accurate branch predictor than the target machine's predictor.

5.4. Overhead of 2D-Profiling

To estimate the overhead of 2D-profiling, we compared the execution times of benchmarks with instrumentation required for 2D-profiling to the execution times with instrumentation needed for edge profiling. Figure 16 shows the normalized execution times of six branch-intensive benchmarks without any instrumentation (*Binary*) and with four different types of instrumentation using the Pin tool: [8] *Pin-base* is the execution time of the benchmark with Pin but without any user instrumentation; *Edge* is the execution time with instrumentation for edge profiling; *Gshare* is the execution time with instrumentation that models a gshare branch predictor in order to compute the branch prediction outcome of each branch; *2D+Gshare* is the execution time with instrumentation needed for 2D-profiling using the

[7] For individual benchmark results, please refer to [11].

[8] Each experiment was run five times and the results were averaged.

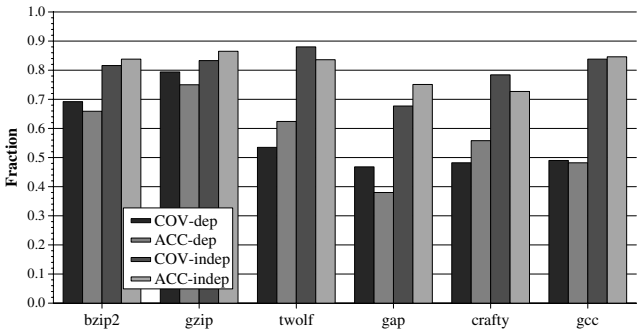

Figure 15. 2D-profiling coverage and accuracy (with max number of input sets) when the target machine uses a perceptron predictor

software gshare branch predictor. On average, the execution time of binaries with instrumentation modeling a gshare predictor is 54% higher than that of binaries with edge profiling. 2D-profiling adds only 1% overhead on top of the software gshare branch predictor. Hence, if the profiler uses the outcomes a hardware branch predictor instead of implementing the branch predictor in software [3], 2D-profiling would not add much overhead over edge profiling.

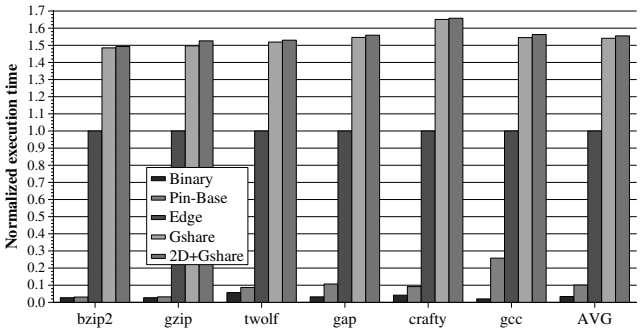

Figure 16. Overhead of 2D-profiling compared to edge profiling

6. Related Work

We discuss research related to 2D-profiling in the areas of profiling and predicated execution. The major contribution of our paper to the body of research in these areas is an *efficient profiling mechanism that identifies input-dependent branches in order to aid compiler-based branch-related optimizations, such as predicated execution, without requiring multiple profiling runs*. No previous work we are aware of proposed a way of identifying input-dependent branches at compile-time using a single input set. 2D-profiling provides a new way of using time-varying phase information obtained via a single profiling run to detect input-dependent branches.

6.1. Related Work in Profiling

Much research has been done on evaluating and improving the accuracy of compile-time profiles. Wall [28] defined the first metrics to measure the representativeness of the profile data across different input sets. His work focused on identifying the similarities and differences in basic blocks, procedure calls, and referenced global variables between different profiling runs.

Fisher and Freudenberger [5] studied branch behavior across input sets. They concluded that most C-language SPEC 89 benchmarks are dominated by biased branches whose directions usually vary little across input sets. However, they found that the behavior of branches in two programs, spice and compress, varied significantly across input sets. Their study focused on quantifying the occurrence of input-independent branches, not identifying them. Building on their work, we provide a way to efficiently identify input-dependent and input-independent branches at compile-time.

Sherwood and Calder studied the time-varying phase behavior of programs [24] and proposed using this behavior to automatically find representative program points for microarchitecture simulation [25]. They showed that IPC, branch/value/address prediction accuracy, and cache miss rates have phase behavior. Smith [26] gave examples of the phase behavior of individual branch and load instructions and discussed how dynamic optimization systems may be able to exploit such behavior. Our paper uses the phase behavior of an individual branch instruction as a compile-time predictor for its input-dependence behavior.

6.2. Related Work in Predicated Execution

Predicated execution has been studied extensively in order to reduce the branch misprediction penalty [14, 23, 27, 2] and to increase the instruction-level parallelism [1, 7]. Our motivation for 2D-profiling is to identify input-dependent or input-independent branches in order to aid the compiler in making predication decisions. We only discuss the relevant mechanisms that proposed ways to find candidate branches for predication.

The IMPACT compiler [15] uses path execution frequencies to decide which blocks can be combined to generate superblocks or hyperblocks. In the IMPACT compiler, predicated code is mainly used for increasing the basic block sizes (via hyperblocks) to provide more optimization opportunities to the compiler and to increase the instruction-level parallelism, as well as for eliminating branch mispredictions.

Less aggressive predicated code generation algorithms convert only short forward branches [23, 27] or only highly mispredicted branches to predicated code [2]. Chang et al. [2] showed that the set of branches that cause most of the mispredictions remains similar across input sets. With their criteria, they were able to convert only a small subset of all static branches to predicated code. Recent compilers [16, 21] use edge profile information and equation (3) to generate predicated code. However, these compilers do not identify input-dependent branches when generating predicated code. As we have described, input-dependent branches may significantly impact the performance of predicated code and identifying them via 2D-profiling can increase the effectiveness of predicated execution (and other compile-time optimizations).

Hazelwood and Conte [6] discussed the performance problems associated with predicated code when the input set of the program changes. They used dynamic profiling to identify hard-to-predict branches at run-time to solve this problem. Their mechanism dynamically converted the identified hard-to-predict branches to predicated code via a dynamic optimization framework. They sampled the individual branch misprediction rates at the beginning of the program to identify most of the hard-to-predict branches for a given input set. In contrast to their mechanism, 2D-profiling identifies input-dependent branches at compile-time and therefore does not require a dynamic optimization framework.

7. Conclusion and Future Work

We have described 2D-profiling, a new compile-time profiling algorithm to detect input-dependent branches without requiring profiling runs with multiple input data sets. 2D-profiling tracks the time-varying prediction accuracy behavior of branch instructions during a profiling run. If the prediction accuracy of a branch varies significantly over time during the profiling run, that branch is identified as input-dependent. Our evaluation of 2D-profiling using the SPEC CPU 2000 integer benchmarks shows that 2D-profiling can identify the input-dependent and input-independent branches accurately.

The use of 2D-profiling can help increase the quality of many branch-based compiler optimizations, such as predicated execution and hot-path/trace/superblock-based optimizations. 2D-profiling can eliminate the performance loss of predicated execution due to input-dependent behavior of branches. For example, if a 2D-profiling compiler identifies a branch is input-independent, it can aggressively generate predicated code for it. In contrast, if a branch is input-dependent, the compiler can convert it into a wish branch [10] instead of predicating it. In future work, we intend to develop better if-conversion heuristics that utilize 2D-profiling. We also intend to investigate the application of 2D-profiling to the detection of other kinds of input-dependent behavior, such as load values or branch taken rates.

Acknowledgments

We thank Veynu Narasiman, other members of the HPS research group, and the anonymous reviewers for their comments and suggestions. We gratefully acknowledge the support of the Cockrell Foundation, Intel Corporation and the Advanced Technology Program of the Texas Higher Education Coordinating Board.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *POPL-10*, 1983.

[2] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. N. Patt. Using predicated execution to improve the performance of a dynamically-scheduled machine with speculative execution. In *PACT*, 1995.

[3] T. M. Conte, B. A. Patel, K. N. Menezes, and J. S. Cox. Hardware-based profiling: an effective technique for profile-driven optimization. *International Journal of Parallel Programming*, 24(2):187–206, Apr. 1996.

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

[5] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *ASPLOS-V*, 1992.

[6] K. Hazelwood and T. Conte. A lightweight algorithm for dynamic if-conversion during dynamic optimization. In *PACT*, 2000.

[7] P. Hsu and E. Davidson. Highly concurrent scalar processing. In *ISCA-13*, 1986.

[8] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(9–50), 1993.

[9] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *HPCA-7*, 2001.

[10] H. Kim, O. Mutlu, J. Stark, and Y. N. Patt. Wish branches: Combining conditional branching and predication for adaptive predicated execution. In *MICRO-38*, 2005.

[11] H. Kim, M. A. Suleman, O. Mutlu, and Y. N. Patt. 2D-Profiling: Detecting input-dependent branches with a single input data set. Technical Report TR-HPS-2006-001, The University of Texas at Austin, Jan. 2006.

[12] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[14] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *MICRO-27*, 1994.

[15] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *MICRO-25*, 1992.

[16] S. Mantripragada and A. Nicolau. Using profiling to reduce branch misprediction costs on a dynamically scheduled processor. In *ICS*, 2000.

[17] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[18] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *ISCA-26*, 1999.

[19] D. C. Montgomery and G. C. Runger. *Applied Statistics and Probability for Engineers*. John Wiley and Sons Inc., third edition, 2002.

[20] A. V. Oppenheim, R. Schafer, and J. R. Buck. *Discrete-Time Signal Processing*. Prentice-Hall, second edition, 1999.

[21] ORC. Open research compiler for Itanium processor family. http://ipf-orc.sourceforge.net/.

[22] K. Pettis and R. C. Hansen. Profile guided code positioning. In *PLDI*, 1990.

[23] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and dynamic branch prediction in dynamic ILP processors. In *ISCA-21*, 1994.

[24] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, 1999.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X*, pages 45–57, 2002.

[26] M. D. Smith. Overcoming the challenges to feedback-directed optimization. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 1–11, 2000.

[27] G. S. Tyson. The effects of predication on branch prediction. In *MICRO-27*, 1994.

[28] D. W. Wall. Predicting program behavior using real or estimated profiles. In *PLDI*, 1991.