An Analysis of Hard to Predict Branches

Celal Ozturk and Resit Sendag
Department of Electrical, Computer and Biomedical Engineering
University of Rhode Island
Kingston, RI
(cozturk, sendag)@ele.uri.edu

Abstract

Branch prediction accuracy remains to be critical for high performance and low power. Prior work has studied causes of branch mispredictions in order to provide insights into how better branch predictors can be designed. However, most of the previous works have only considered run-time classification of branch mispredictions, leaving a large number of mispredictions in an unknown category. For more comprehensive analysis, in this paper, we present a detailed source code analysis of branch mispredictions for SPEC CPU 2000 and Mibench benchmarks. Our analysis show that constant loop exits, insufficient history lengths, wrong-type history, array access/pointer references, complex linked list data structures, changing function inputs, and varying loop counts are the major causes for most of the branch mispredictions. We further show that most mispredictions have repetitive patterns that suggest different design strategies for future branch predictors.

1. Introduction

Branch prediction has been studied extensively for the last two decades. There have even been two championship branch prediction (CBP) competitions [1, 2], which have introduced very sophisticated and accurate branch predictor designs. Yet, there has not been a comprehensive study on branch misprediction classification through detailed source-code analysis of hard-to-predict branches. Most comprehensive branch misprediction classification was done by Skadron et al [3]. In their paper, Skadron proposed a taxonomy for run-time classification of branch mispredictions. This taxonomy classifies mispredictions as being from one of the following categories: *training*, *interference* in tables, *wrong-history*, *need both histories* simultaneously, and *other*. *Other* category includes insufficient history length or inherently difficult to predict branch mispredictions that can not be classified by this taxonomy. Many of the prior work target one or more of the above problems to improve the accuracy of branch prediction. [4-11] studied eliminating interference (aliasing) in predictors' pattern history table/s (PHT), which occurs when two unrelated branches destructively interfere by using the same prediction resources. To address wrong-history mispredictions, a number of hybrid predictors were proposed [6, 12-14]. Because these predictors include multiple tables with global and local histories, they also required a mechanism (a *meta-predictor*) to choose which predictor to believe at any moment (later methods used *adder trees* instead of meta-predictors [15, 16]). An improvement to this, which could also reduce mispredictions that need both types of history simultaneously, was a pseudo-hybrid organization that alloys global and local history in the same predictor index [17]. Other predictors try to also reduce mispredictions that need longer histories [18, 19]. These predictors employ multiple prediction tables indexed with different-length folded histories. Another effort in reducing mispredictions that need longer histories has been prediction based on neural networks [16, 20-24]. To mitigate longer prediction and training times of complex/large predictors, they are usually cascaded with simple ones [12] or ahead-pipelined [25]. Finally, to reduce constant loop exit mispredictions, loop count predictor was proposed [26].

In this paper, we extend prior work on branch misprediction classification through detailed source code analysis of hard-to-predict branches. We focus mainly on branches, which can not be classified by Skadron's taxonomy. This paper makes the following contributions:

1. It confirms already known facts about the causes of branch mispredictions, and extends prior work on branch misprediction classification by introducing new classes of mispredictions.
2. It shows that array element accesses or pointer references, linked list traversals, varying loop counts, and changing function inputs make harder to predict branches that depend on them and may require different ways to improve prediction accuracy.
3. It introduces new type of correlations other than branch outcome histories.

The remainder of this paper is organized as follows: Section 2 describes the experimental setup. Section 3 presents run-time classification of branch mispredictions. Section 4 presents misprediction coverage by top 10 hot branches. In Section 5, we present a detailed source-code analysis of branch mispredictions. Section 6 summarizes our findings on source-code analysis. Section 7 describes some related work, and Section 8 concludes.

2. Experimental Setup

To collect the results presented in this paper, we have used `sim-bpred` simulator from the Simplescalar tool suite [27], version 3.0d. We also extend this simulator to include Alpha 21264 [12] and Piecewise Linear (PWL) [21] branch predictors. We used benchmarks from the SPEC CPU 2000 [28] and MiBench [29] benchmark suites. We present the results for all 26 SPEC CPU 2000 benchmarks when using the reference input set, but, for the benchmarks with more than one input set, to reduce the simulation time, we randomly selected one input set. The input set is listed in parenthesis for the following list of benchmarks that had more than one input set: `gzip` (graphic), `vpr` (route), `gcc` (166), `art` (110), `eon` (cook), `vortex` (ref1), and `bzip2` (graphic). For the MiBench benchmarks, due to compilation problems, we used a

subset of the benchmarks (basicmath, bitcount, dijkstra, fft, ghostscript, jpeg, patricia, qsort, rsynth, sha, stringsearch, and susan). These benchmarks were compiled using gcc 4.1.2 at optimization level O3. For all MiBench benchmarks, we used the large input set. Overall, we evaluated a total of 38 benchmarks.

To reduce the simulation time of the SPEC CPU 2000 benchmarks, we used multiple 100M instruction simulation points that we generated using SimPoint tool [30]. Due to their relatively short simulation time, we ran the MiBench benchmarks to completion.

3. Run-time Classification of Branch Mispredictions

We repeat Skadron's run-time branch misprediction classification for SPEC CPU 2000 and Mibench benchmarks with a 4kB (i.e., 16K entries) `gshare` [6] predictor. Mispredictions are classified into five groups: *conflict*, *training*, *wrong-history*, *needs both history*, and *other*. To classify a branch's misprediction type, we perform a sequence of tests as described in [3]. Each branch flows down this sequence of tests until it is categorized or falls through as a misprediction that could not be categorized.

1) The prediction starts with a `gshare` predictor. If the prediction is incorrect, misprediction classification starts.

2) The first step is to test if a `gshare` predictor with no aliasing could predict the branch. When the `gshare` predictor that is free of aliasing is implemented, the number of table entries is kept the same (i.e., same history size is used). However, each table entry remembers all branch references to that entry by updating their corresponding 2-bit counters. Therefore, the predictor is free of destructive interference. If this predictor was able to provide correct prediction, the misprediction falls into the *conflict* category. That is, the predictor under test would predict the branch correctly, but a destructive interference prevented the predictor from doing so, and as a result, a *conflict* misprediction has occurred.

3) The second step uses a 2-bit predictor to predict the branch. If this prediction is correct, it suggests that the branch has not been predicted correctly before because the branch predictor under test has long training time. This is a misprediction due to *training* (as mentioned in [3], this is an approximation.)

4) If the branch misprediction has not been classified in the previous steps, it may have happened because the branch needs local history. If a local predictor of the same size, but free of interference (logically infinite sized predictor), predicts this branch correctly, it suggests that global history is not appropriate for this branch because it needs local history, i.e., it is a *wrong type history* misprediction.

5) If still not classified, an interference-free predictor that uses both global and local histories is tested if it can provide correct prediction for this branch. A correct prediction in this case suggests the branch needs both types of history, and the misprediction is classified as "*needs both types of history*". However, if the branch mispredicts with this predictor also, it falls into the group of *other* mispredictions as it cannot be classified by this taxonomy.

By running several predictor organizations of increasing sophistication simultaneously, our simulator performs the abovementioned cascade of tests until the branch either predicts correctly, or the misprediction fails all tests. Remaining branches are either inherently difficult to predict, or fall into a category not included in this scheme (e.g., need longer history). This process categorizes each dynamic branch's behavior for `gshare` branch predictor.

Figure 1 shows the breakdown of the branch misprediction categories for SPECint, SPECfp, and Mibench benchmarks, respectively. An interesting observation is that, for most of the benchmarks, the "*other"* is the largest category. This is more pronounced for the following benchmarks: for bzip2, vpr, mcf, parser, perl, and twolf, about 50% of the mispredictions fall into the "other" category; And for art, swim, mgrid, lucas, sixtrack, dijkstra, susan, sha, and bitcount, more than 75% of the mispredictions fall into the "other" category.

Table 1 summarizes the results by showing average percentages of each class of mispredictions per benchmark suite. These results show the importance of wrong type history along with well known problems of conflicts and training times. However, we also see that a large percentage of mispredictions (about 40% on average) can not be categorized as being from one of the abovementioned misprediction types using this taxonomy. It must also be noted that, with this

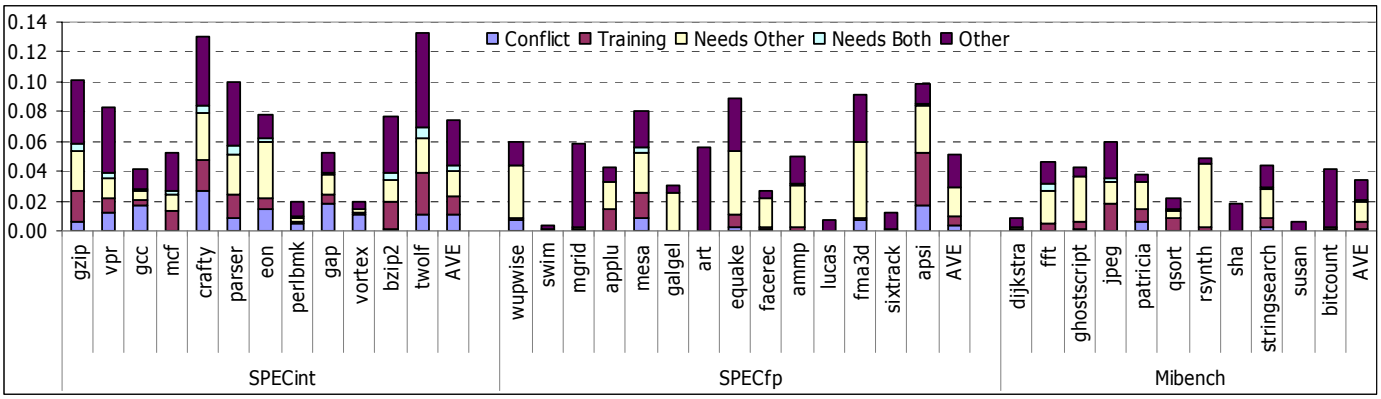

Figure 1. Breakdown of misprediction types for 4kB `gshare` predictor for SPECint, SPECfp and Mibench benchmarks. Our experiments also show that larger global history decreases mispredictions in conflict, need other and other categories while increasing mispredictions in training category, which is expected.

taxonomy, a branch's mispredictions may fall into different categories for different dynamic instances of the branch. Therefore, this taxonomy can not provide detailed information about a specific branch. This suggests a further investigation for important branch instructions. In this paper, after identifying hot branches through run-time profiling, we perform source-code analysis in order to provide more insights into why specific branches mispredict often. This also identifies branches, which cause mispredictions that go under the "other" category.

Table 1. Average contribution of misprediction types (%)

	Conflict	Training	Need Other	Need Both	Other
SPECint	14.88	16.51	23.79	4.40	40.42
SPECfp	6.65	11.69	39.52	0.85	41.28
Mibench	2.96	14.94	41.18	3.26	37.65

4. Mispredictions by Hot Branches

Figure 2 shows how hot branch PCs contribute to the overall mispredictions for SPECint, SPECfp, and Mibench benchmarks, respectively, when a 4kB `gshare` branch predictor is used. On average, for SPECint, top 5, top 10, top 20 static branches cause 39%, 53%, 65% of all mispredictions, respectively. For SPECfp, top 5, top 10, top 20 static branches cause 71%, 83%, 92% of all mispredictions, respectively. Finally, for Mibench, top 5, top 10, top 20 static branches cause 67%, 79%, 87% of all mispredictions, respectively. Majority of mispredictions are caused by few hot branches.

5. Source Code Analysis for Branch Misprediction Classification of Hot Branches

This section presents a detailed source code analysis on top ten hot branch PCs that mispredict most in order to evaluate why they mispredict, how and if they can be corrected (either with current methods or others), and if they go under a new category for mispredictions.

In the analysis, we have used 4kB `gshare` branch predictor. When branches are analyzed, however, we have also tried to see if they still remain important when different (and/or better) predictors, such as Alpha or PWL, are used to predict these branches. We have analyzed all SPEC and Mibench benchmarks written in C.

The rest of this section presents a detailed discussion for various different branch misprediction types.

5.1. Array Element Access or Pointer References

When a particular branch depend on array access or pointer references, irregular values loaded from array element accesses make it difficult to predict this branch. However, in some cases, the value that each array element stores remain unchanged for a long time. Due to this address-value correlation, branch outcome is consistent with the address values, i.e., the array indices. A branch predictor with local history length size greater than the number of loop iterations will be able to capture this type of behavior. However, this is often impractical.

Examples of branch mispredictions that are caused by array element accesses or pointer references are found in almost all benchmarks.

Figures 3a and 3b show assembly and C source codes, respectively from `gcc` benchmark that include the most frequently mispredicted branch (`BR1`). This branch mispredicts 29.3% of the time and corresponds to 44% of all mispredictions in gcc. Using longer history for gshare predictor does not help reduce mispredictions for this branch. A 32KB Piecewise Linear branch predictor can only slightly help improve the prediction rate. This is obviously a hard-to-predict branch that must go under the "other" category with the taxonomy used earlier. However, we noticed that the "other" category for gcc was only 31% as shown in Figure 1. This confirms that some dynamic instances of this branch's mispredictions were classified as conflict, training, wrong-history or need both.

Figure 3 shows that `BR1` checks if the value loaded into register `$3` by the load `LD1` is less than or equal to zero. Register `$3` contains the value of `regno_first_uid[j]` (value of jth element of the array) in Figure 3b. Register `$2` is used for calculating and storing the address for `regno_first_uid[j]` (see `LD1` in Figure 3a). This branch is executed for 198614 times. There are 41112 different addresses (i.e., register `$2` values) that are used by `LD1` and each of these addresses are accessed for five times. The values in these addresses (i.e., values (`$3`) of `regno_first_uid` array elements) are very stable or do not change frequently. The predictor mispredicts at 12166 different addresses (12166

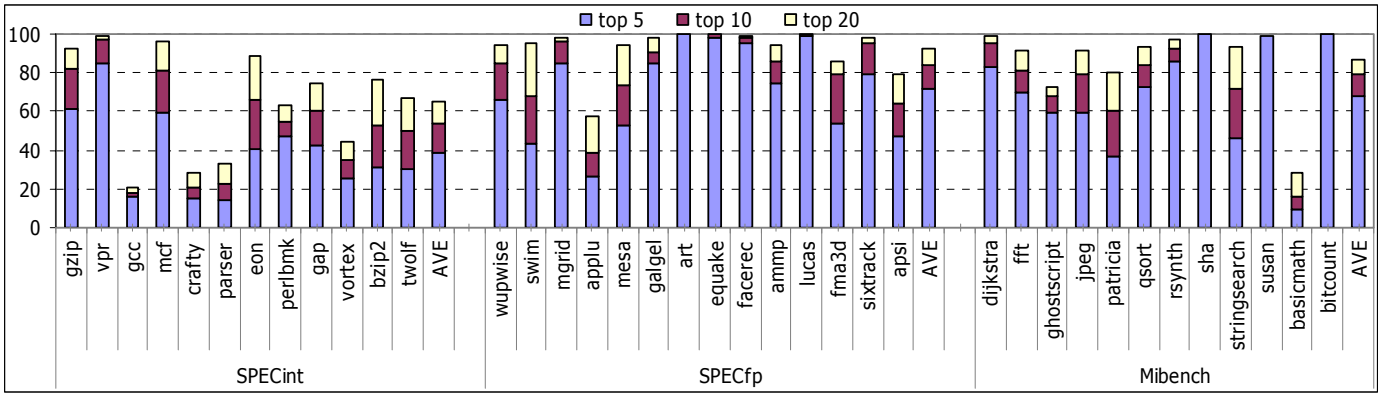

Figure 2. Contribution of the top 5, top 10, and top 20 branch PCs to overall mispredictions for (a) SPECint, (b) SPECfp, (c) Mibench. 4kB gshare branch predictor.

different values of $2). If there is a misprediction on one address, misprediction occurs again next time the same address is loaded (i.e., the same array element is accessed). While very hard to predict with current techniques, a correlation between addresses and mispredictions and the stability of data in the array suggest different methodology for prediction. For example, mispredictions could be predicted by using register $2 values, that is, the addresses of the array elements. In fact, the misprediction can be detected in the earlier iterations of the loop before it actually happens since this is an array access. However, naïve implementation could require large storage to detect this correlation. Further research needs to be done to see if there is an efficient way for reducing this type of mispredictions.

```
unroll.c:746
 5358c8:    28 00 00 00    lw $2,-17468($28)
 5358d0:    43 00 00 00    addiu $17,$0,64
 5358d8:    5b 00 00 00    slt $2,$17,$2
 5358e0:    05 00 00 00    beq $2,$0,5359f8
 5358e8:    28 00 00 00    lw $11,120($30)
 5358f0:    43 00 00 00    addiu $10,$0,1
 5358f8:    43 00 00 00    addiu $7,$11,64
unroll.c:747
 535900:    28 00 00 00    lw $2,-17360($28)
 535908:    55 00 00 00    sll $6,$17,0x2
 535910:    42 00 00 00    addu $2,$6,$2
 535918:    28 00 00 00    lw $3,0($2) → LD1
 535920:    07 00 00 00    blez $3,5359d0 → BR1
```

(a)

```
// unroll.c
746    for (j = FIRST_PSEUDO_REGISTER; j < max_reg_before_loop;
++j)
747      if (regno_first_uid[j] > 0 && regno_first_uid[j] <=
max_uid_for_loop → LD1, BR1
748          && uid_luid[regno_first_uid[j]] >= copy_start_luid
749          && regno_last_uid[j] > 0 && regno_last_uid[j] <=
max_uid_for_loop
750          && uid_luid[regno_last_uid[j]] <= copy_end_luid)
751        local_regno[j] = 1;
752   }
```

(b)

Figure 3. Code snippet for one of the hot PCs in gcc, (a) assembly, (b) C code

In Figure 4, we show another example of this type of mispredictions for the art benchmark. Misprediction rate for this branch (BR1) is 8.62%. This corresponds to 8% of all mispredictions in art when using a 4KB gshare predictor. Increasing the history size (predictor size) does not reduce the mispredictions significantly for this branch (i.e., a 256KB gshare reduces misprediction rate from 8.62% to 8.1%). Even when using a 256KB PWL predictor misprediction rate for this branch is 8.2%. However, when a very large and/or better predictor is used, total mispredictions for art decreases significantly, which makes this branch far more important because it then corresponds to 95% of all mispredictions.

Figure 4 shows the source code (assembly and C code) from art that includes this hot branch (BR1). BR1 checks if value stored in register $f0 is less than the value stored in register $f4. $f0 holds the value of f1_layer[ti].X and $f4 holds the value of theta as shown in Figure 4. Register $3 holds the address of f1_layer[ti] (LD1). BR1 outcomes depend on the values returned by LD1. BR1 is executed for 340000 times (the loop in figure iterates 34 times, and there is an outer loop, which is executed 10000 times (not shown in figure). Register $3 takes 34 different addresses and at all those addresses, the value in that address (f1_layer[ti].X) remains unchanged throughout the simulation. The branch outcome is consistent with the load addresses (values in register $3). This is another example of address-value correlation, which suggests that the branch can be predicted very well by using the value in register $3 (the address of f1_layer[ti]). This branch also does not require large storage to remember correlations since there are only 34 addresses.

```
scanner.c:419
 401af0:    28 00 00 00    lw $2,-31344($28)
 401af8:    42 00 00 00    addu $6,$0,$0
 401b00:    07 00 00 00    blez $2,401bc8
 401b08:    2b 00 00 00    l.d $f4,-31400($28)
 401b10:    2b 00 00 00    l.d $f6,-31448($28)
 401b18:    42 00 00 00    addu $4,$0,$2
 401b20:    28 00 00 00    lw $3,-31360($28)
scanner.c:421
 401b28:    2b 00 00 00    l.d $f0,16($3) → LD1
 401b30:    93 00 00 00    c.lt.d $f0,$f4
scanner.c:424
 401b38:    7b 00 00 00    mov.d $f2,$f0
scanner.c:421
 401b40:    0b 00 00 00    bc1f 401b58 → BR1
```

(a)

```
// scanner.c
419     for (ti=0;ti<numf1s;ti++)
420     {
421       if (f1_layer[ti].X < theta) → LD1, BR1
422           xr = 0;
423       else
424           xr = f1_layer[ti].X;
...       ...
429       f1_layer[ti].V = xr + b*qr;
430       tnorm += f1_layer[ti].V * f1_layer[ti].V;
431     }1485   return 0;
1486 }
```

(b)

Figure 4. Code snippet for second top hot PC in art, (a) assembly, (b) C code

The two examples given from gcc and art in this section show a new class of branch mispredictions, which we call array element access or pointer references. When data values are stable, there is chance to reduce mispredictions of this type. However, if data values change frequently and they are random, it is not possible to find correlation of any kind for the mispredictions. We found few examples of this kind in parser and bzip2.

5.2. Linked List Traversals

A pointer-chasing load, such as node=node→next, that determines the end of a linked list makes it hard to predict the branch that depend on it. If the linked list has n nodes, the loop iterates n times and the branch outcomes would be n-1 times "taken" followed by a "not taken". Branch predictors that exploit correlation in branch outcome histories often fail to predict these branches accurately. Our analysis shows that mcf, parser, and dijkstra have significant amount of

hard-to-predict branches of this type. However, at a closer look, these hard-to-predict branches may be predicted correctly because, although they do not have regular correlation in branch histories, they exhibit a type of locality that can be exploited with different mechanisms. Most components of data structures in SPEC CPU 2000 and Mibench benchmarks tend to remain stable. For example, after a linked list is initialized, the address of the end node remains the same until a new node is added to the end. In fact, even the order of the node addresses that is traversed remain the same until there is insertion or deletion. Therefore, if there are `n` nodes and if last `m` nodes of the linked list remain stable, once node `n-m` is accessed, one can predict that branch outcome that depends on this linked list traversal should be not taken when node `n` is reached. If a branch depends on such stable data, address of the data is sufficient to determine the branch outcome.

Figures 5 and 6 show examples of linked-list-traversal-caused branch mispredictions for `parser` and `mcf`, respectively. Similar examples are also found in `gcc`, `art`, `ammp`, `jpeg`, `bzip2`, `basicmath`, `dijkstra`.

Figure 5 shows the source code (assembly and C code) from `parser` that includes a tree structure. In this pointer-chasing code, the loaded values that determine the branch outcome are irregular, which makes this branch hard to predict. Conventional branch predictors fail to provide very accurate predictions for this type of branches. However, because `BR1` in Figure 5 is mostly taken (92% of the time), a 4KB `gshare` predictor is still doing well. The misprediction rate is 8.67%. A 32KB `gshare` further reduces the misprediction rate to 7.4%. A 32KB PWL can achieve 6% misprediction rate.

```
post-process.c:746
 419d70:    28 00 00 00    lw $16,0($18)
 419d78:    05 00 00 00    beq $16,$0,419e80
post-process.c:747
 419d80:    28 00 00 00    lw $3,4($16)
 419d88:    55 00 00 00    sll $2,$3,0x2
 419d90:    42 00 00 00    addu $2,$2,$3
 419d98:    55 00 00 00    sll $2,$2,0x2
   ...        ...            ...
 419dc0:    02 00 00 00    jal 416ca0
 419dc8:    06 00 00 00    bne $2,$0,419de0
post-process.c:746
 419dd0:    28 00 00 00    lw $16, 8($16) → LD1
 419dd8:    06 00 00 00    bne $16,$0,419d80 → BR1
```

(a)

```
// post-process.c
743   D_tree_leaf * dtl;
744   int d, count;
745   for (d=0; d<N_domains; d++) {
746     for (dtl = domain_array[d].child;
              dtl != NULL;   → BR1
              dtl = dtl->next) {  → LD1
747       if (ppmatch(selector, pp_link_array[dtl>link].name))
            break;
748     }
```

(b)

Figure 5. Code snippet for top second hot PC in parser (for one of the simpoints), (a) assembly, (b) C code.

In Figure 5, branch `BR1` checks if the value in register `$16` is not equal to NULL. `$16` holds the address of `dtl`. `BR1` is dependent on the pointer-chasing load, `LD1` (`dtl=dtl→next`). `LD1` often misses in cache, which means `BR1` resolves late. `BR1` is accessed 476293 times. There are 1172 different values for register `$16` (`dtl` addresses). Each address is accessed about 400 times on average. The values in these addresses do not change frequently. There are only a few changes throughout the simulation. Thus, address values (`$16`) instead of data loaded from these addresses are sufficient to know the branch outcome. There is also a pattern in which addresses follow each other, i.e., few node insertions or deletions for a long time. Since the data structures are very stable, register values that hold node address values in previous iterations of the loop can be used to predict the outcome of the branch instance that is dependent on the end node in the linked list.

Another example of linked-list-traversal-caused branch mispredictions is shown in Figure 6 for the `mcf` benchmark. Misprediction rate for this branch (`BR1`) is 24.3%. This corresponds 22% of all mispredictions in `mcf` when using a 4KB `gshare` predictor. Increasing the history size (predictor size) does not reduce the mispredictions. A 256KB `PWL` predictor also fails to reduce misprediction rate for this branch. There are total of three branches of this type in `mcf`, which correspond to 58% of all mispredictions.

```
mcfutil.c:96
 4007d8:    42 00 00 00    addu $4,$0,$3
mcfutil.c:98
 4007e0:    28 00 00 00    lw $2,8($4)
 4007e8:    05 00 00 00    beq $2,$0,400828
mcfutil.c:100
 4007f0:    28 00 00 00    lw $3,16($4)  → LD1
mcfutil.c:101
 4007f8:    05 00 00 00    beq $3,$0,400810 → BR1
mcfutil.c:84
 400748:    28 00 00 00    lw $2,28($4)
 400750:    06 00 00 00    bne $2,$7,400788
```

(a)

```
// mcfutil.c
79   tmp = node = root->child;
80   while( node != root )
81   {
82     while( node )
83     {
         …
94     }
95
96     node = tmp;
97
98     while( node->pred )
99     {
100      tmp = node->sibling;  → LD1
101      if( tmp )  → BR1
102      {
103        node = tmp;
104        break;
105      }
106      else
107        node = node->pred;
108    }
109  }
```

(b)

Figure 6. Code snippet for one of the hot PCs in `mcf`, (a) assembly, (b) C code

Figure 6 shows the source code from mcf that includes this hot branch (BR1). BR1 checks if the value in register $3 is equal to zero. Register $3 holds the address of node→sibling (value returned by a load, LD1, which is temp in Figure 6.b), and $4 holds the address of node.

BR1 is accessed for 2579273 times. There are 33111 different values for $4 (LD1 addresses). Every address is accessed about 78 times. The values at each address change rarely throughout the simulation. There are 35855 address-value pairs. Therefore, this branch's outcome could be predicted by using the address value in register $4 instead of the loaded data value in register $3. The top three hot branches in mcf have the same behavior.

Branch mispredictions that are caused by linked list traversals as described in this section are very hard to predict using conventional branch predictors. However, due to address-value correlations, it is possible to make correct predictions if corresponding register values (addresses) are used. More research is needed on how such mechanisms can be built efficiently with limited hardware budgets. Perhaps, since the data structures are very stable, a mechanism could be designed to correct mispredictions before they take place by using register values that hold node address values in previous iterations of the loop.

5.3 Varying Loop Counts

This type of mispredictions occurs when a loop is executed for different number of iterations every time it is accessed. Almost all of the benchmarks we have tested contain loops with varying loop counts.

Figure 7 shows a source code example from patricia with a loop that has varying loop count. Register $6 holds the value of __nbytes. Initial value of __nbytes is input to the BYTE_COPY_FWD () function. BR1 checks if the value in register $6 is greater than zero, and is executed 758584 times. The input __nbytes has two different values only; 23538 times 4 and 83054 times 8. There is no pattern for the loop count but it is 8 most of the time. The misprediction rate for this branch with a 4KB gshare predictor is 14.1%. This branch could be predicted better by a complex loop predictor that would use the most common loop count. Because, the branch is either taken 8 times followed by a not taken or taken 4 times followed by a not taken, a local predictor would also do well. In fact, a 4KB local predictor is able to reduce to misprediction rate to 5%. If there was also a pattern of varying loop counts, a local predictor could even do better. Another observation we have about this type of mispredictions is that if there is a pattern, future mispredictions can usually be predicted.

Varying loop count mispredictions are usually classified as conflict- or training-caused mispredictions when run-time classification taxonomy is applied.

5.4 Changing Function Inputs

This type of mispredictions occurs because a branch depends on an input to a function and this input often changes next time the function is called and causes unless otherwise easy to predict branch very hard to predict. We have given two

```
../sysdeps/generic/memcpy.c:52
 404658:    42 00 00 00    addu $6,$0,$18
 404660:    05 00 00 00    beq $6,$0,404698
 404668:    22 00 00 00    lbu $2,0($16)
 404670:    43 00 00 00    addiu $6,$6,-1
 404678:    43 00 00 00    addiu $16,$16,1
 404680:    30 00 00 00    sb $2,0($17)
 404688:    43 00 00 00    addiu $17,$17,1
 404690:    06 00 00 00    bne $6,$0,404668 → BR1
```

(a)

```
//memcpy.c
52  BYTE_COPY_FWD (dstp, srcp, len); → BR1
//memcopy.h
76 #define BYTE_COPY_FWD(dst_bp, src_bp, nbytes)
77  do
78    {
79      size_t __nbytes = (nbytes);
80      while (__nbytes > 0)  →  BR1
81        {
              …
87        }
88    } while (0)
```

(b)

Figure 7. Code snippet for the second top hot PC in patricia, (a) assembly, (b) C code

examples of this type in this section because they show very different behavior. The first example is from patricia where changing inputs cause varying loop counts, but easy to predict with local predictor because there is a pattern for inputs. The other example is from parser, where there is no pattern – a truly hard to predict branch. Most benchmarks contain this type of branch mispredictions.

Figure 8 presents a simple while loop from patricia where a variable (j) that is a function input checked against zero. Initial value of j is input to the function. Register $6 holds the value of j. BR1 checks if the value in register $6 is not equal to zero. BR1 is executed 405708 times. The input j is182596 times -1 and 40516 times 0. There is a pattern for the branch behavior. The behavior is 19 times "1 taken, 1 not taken" and 1 time "1 taken, 3 not taken".

Misprediction rate for this branch (BR1) with 4KB gshare is 45% ! However, the branch follows a simple consistent pattern and therefore could be predicted almost perfectly with a local predictor. Branch predictors that can exploit very long global histories, such as PWL and LTAGE could also eliminate the mispredictions for BR1. We also observe that this branch has also global and local patterns for mispredictions, which suggests that a mechanism that targets predictable misprediction patterns can also eliminate these mispredictions when the branch predictor is gshare.

This branch is predictable because function inputs have a nice pattern. However, if the inputs were random, this would have been a hard to predict branch. Such an example is given from parser in Figure 9. BR1 in Figure 9 has an outcome that is 45% taken and 55% not taken. Misprediction rate for this branch (BR1) is 24.4% with a 4KB gshare. A 256KB gshare predictor, a 256KB local predictor or a 256KB PWL does not reduce the misprediction rate for BR1 at all. This is an inherently very hard to predict branch because of the random data.

```
../sysdeps/generic/mul_1.c:55
 413be8:    43 00 00 00    addiu $6,$6,1
../sysdeps/generic/mul_1.c:53
 413bf0:    34 00 00 00    sw $3,0($8)
../sysdeps/generic/mul_1.c:55
 413bf8:    43 00 00 00    addiu $8,$8,4
 413c00:    43 00 00 00    addiu $5,$5,4
 413c08:    06 00 00 00    bne $6,$0,413bb0  → BR1
```

(a)

```
//mul_1.c
 55  while (++j != 0); → BR1
```

(b)

Figure 8. Code snippet for top hot PC in `patricia`, (a) assembly, (b) C

```
parse.c:186
 414718:    43 00 00 00    addiu $29,$29,-48
 414720:    34 00 00 00    sw $16,24($29)
 414728:    42 00 00 00    addu $16,$0,$4
 414730:    34 00 00 00    sw $17,28($29)
 414738:    42 00 00 00    addu $17,$0,$5
 414740:    34 00 00 00    sw $18,32($29)
 414748:    42 00 00 00    addu $18,$0,$6
   …          …              …
parse.c:193
 4147e0:    28 00 00 00    lw $2,-29260($28)
parse.c:194
 4147e8:    24 00 00 00    lh $3,0($4) → LD1
parse.c:193
 4147f0:    43 00 00 00    addiu $2,$2,1
 4147f8:    34 00 00 00    sw $2,-29260($28)
parse.c:194
 414800:    06 00 00 00    bne $3,$16,414858 → BR1
  …           …                 …
 414850:    01 00 00 00    j 414870
```

(a)

```
186 Table_connector * table_pointer(int lw, int rw,
Connector *le,  Connector *re, int cost) {
188    Table_connector *t;
189    N_hash_lookups++;
190    work_in_hash_lookups++;
191    t = table[hash(lw, rw, le, re, cost)];
192    for (; t != NULL; t = t->next) {
193       work_in_hash_lookups++;
194       if ((t->lw == lw) &&  → BR1, LD1
                 . . . )  return t;
195    }
196    return NULL;
197 }
```

(b)

Figure 9. Code snippet for one of the hot PCs in `parser`, (a) assembly, (b) C

`BR1` checks if the value in register `$3` is equal to the value in register `$16`. Register `$3` holds the value of `t->lw` (value returned by `LD1`) and `$16` holds the value of `lw`. Register `$4` holds the address of `t` (see `LD1`). `BR1` is accessed 481215 times. There are 46370 different values for `$4`, which corresponds to the different addresses of `t`. Values that are stored in those addresses remain unchanged. Each different address is accessed 10 times, on average. Although values in these addresses do not change, value they are compared to, `lw`, changes frequently. Due to this randomness, this branch can not be predicted well. However, if sufficient hardware resource (>256KB, i.e., very impractical) is given to store history for this particular branch, mispredictions can be eliminated because the branch outcome at any address of `t` and `lw` pair is consistent.

5.5 Wrong Type History

Mispredictions can also occur because the predictor tracks the wrong type of history for the branch in question: global instead of local, or vice versa. Run-time branch classification taxonomy describes wrong-history misses as mispredictions that occur when a nonconflict-miss/non-training-miss is correctly predicted using the other type of history. We see this type of branch mispredictions in all benchmarks that we studied. This class of mispredictions can be identified by run-time classification taxonomy, however, we have included it in our source-code analysis for completeness.

Figure 10 shows an example of wrong-history type mispredictions from `basicmath`. Misprediction rate for `BR1` is 22.2% (corresponds to 40% of all mispredictions) with a 4KB `gshare` predictor. A 4KB local predictor eliminates mispredictions for this branch almost completely (1% misprediction rate).

In Figure 10, registers `$4` and `$2` hold the address of and the value stored in `char_ptr`, respectively. Register `$5` holds the value of `c`. `LD1` loads the value stored in address `$4` and stores it into `$2`. BR1 checks if the value in `$2` is equal to the value in `$5`.

There are 8 different address values for `$4` and the values in these addresses never change. Also, there is a pattern of length 8 for values in `$5` and always repeats itself. Because branch follows a consistent pattern, it can be predicted very well with a local predictor. One important observation is that there is also a misprediction pattern, 7 hits and 2 misses, which always repeats itself. Our experiments show that wrong-history type mispredictions often follow patterns, which suggests that efficient separate mechanisms can be designed to help the branch predictor by targeting its mispredictions.

```
../sysdeps/generic/strchr.c:43
 40b9a0:    22 00 00 00    lbu $2,0($4)  → LD1
 40b9a8:    05 00 00 00    beq $2,$5,40baf8 → BR1
../sysdeps/generic/strchr.c:45
 40b9b0:    05 00 00 00    beq $2,$0,40bb08
../sysdeps/generic/strchr.c:42
 40b9b8:    43 00 00 00    addiu $4,$4,1
```

(a)

```
//strchr.c
 36  c = (unsigned char) c;
 37
 38  /* Handle the first few characters by reading one character at a time.
 39     Do this until CHAR_PTR is aligned on a longword boundary.  */
 40  for (char_ptr = s; ((unsigned long int) char_ptr
 41                      & (sizeof (longword) - 1)) != 0;
 42       ++char_ptr)
 43    if (*char_ptr == c)  → LD1, BR1
```

(b)

Figure 10. Code snippet for top hot PC in `basicmath`, (a) assembly, (b) C

5.6 Constant Loop Exits

A loop with a loop count of `n`, is often taken for `n` times followed by a not-taken at the loop exit. For cases when the loop counts are larger than what branch predictor can remember, prediction fails at the loop exits. Often, it is difficult for a global, local, or combined history predictor to keep

sufficient history for this type of branches. Therefore, to target loop-exit mispredictions, loop predictor [26] was proposed. Constant loop exit mispredictions can also be put into insufficient history length or wrong-history type mispredictions categories.

All benchmarks have constant loop branches, though they may not mispredict often enough to make the top 10 hot PCs list that mispredict most. A source-code example for a constant loop branch from `art` is shown in Figure 11.

`BR1` in Figure 11 is a constant loop branch with a 9.1% misprediction rate. The loop always iterates for 10 times and executed 340000 times. `gshare` predicts `BR1` correctly for 10 times and mispredicts the loop exit condition. `BR1` checks if value in register `$2` is not equal to zero. Register `$2` is reset to zero if value in register `$4` is greater than value in register `$8`. `$8` corresponds to `num2fs` and `$4` corresponds to `tj` (see line 449 in Figure 11). `numf2s` has a constant value, which is 10. This is not a hard to predict branch. It can be eliminated by a loop predictor or an 11 bit local history predictor. Since there are three more branches in the loop, `gshare` requires a history length greater than 40 bits in order to eliminate mispredictions of this branch. Predictors that can exploit long histories, such as `PWL` and `LTAGE`, eliminate this branch's mispredictions.

```
scanner.c:449
 401cc8:    42 00 00 00    addu $4,$0,$16
 401cd0:    05 00 00 00    beq $13,$0,401d68
 401cd8:    42 00 00 00    addu $9,$0,$10
 401ce0:    28 00 00 00    lw $8,-31320($28)
 401ce8:    55 00 00 00    sll $2,$16,0x4
 401cf0:    42 00 00 00    addu $5,$2,$14
scanner.c:451
 401cf8:    06 00 00 00    bne $4,$12,401d48
            …
 401d10:    0b 00 00 00    bc1f 401d48
scanner.c:452
 401d18:    28 00 00 00    lw $3,0($9)
            …
 401d40:    71 00 00 00    add.d $f4,$f4,$f0
scanner.c:449
 401d48:    43 00 00 00    addiu $5,$5,16
 401d50:    43 00 00 00    addiu $4,$4,1
 401d58:    5b 00 00 00    slt $2,$4,$8
 401d60:    06 00 00 00    bne $2,$0,401cf8 → BR1
```

(a)

```
// scanner.c
441     tnorm =0;
442     tsum=0;
443     tresult = 1;
444     for (ti=0;ti<numf1s;ti++)
445     {
446       tsum = 0;
447       ttemp = f1_layer[ti].P;
448
449       for (tj=spot;tj<numf2s;tj++)  → BR1
450       {
451         if ((tj == winner)&&(Y[tj].y > 0))
452             tsum += tds[ti][tj] * d;
453       }
454
455       f1_layer[ti].P = f1_layer[ti].U + tsum;
456
              …
```

(b)

Figure 11. Code snippet for top hot PC in `art`, (a) assembly, (b) C code

5.7 Insufficient History Lengths

This type of mispredictions can be eliminated if same type of predictor is given more history bits. This is not very easy for a `gshare` predictor because each additional bit doubles the size of the predictor. A predictor like `PWL` or `LTAGE` will do much better for this type of branches. Often wrong-history and constant loop exit mispredictions could be eliminated if the predictor can exploit longer branch histories. Therefore, examples in Figures 10 and 11 can also be considered as insufficient history length mispredictions. However, this does not mean longer history eliminates only wrong-history or constant loop exit mispredictions. An example of this is shown in Figure 12 from `gcc`. `BR1` checks if the loop count (`i` in Figure 12.b) that is stored in register `$19` (see Figure 12.a) is greater than or equal to zero. The loop's start value is the value returned by `GET_RTX_LENGTH(code)` as shown in Figure 12.b. The reason that this branch mispredicts is because the iteration count (i.e., the value returned by the function) varies, and there is no observed pattern for the loop counts. However, according to our simulations, these counts are 1, 2, or 3, which are small values. Therefore, the number of mispredictions can be reduced by a larger predictor. The misprediction rate for `BR1` is 17.4% with a 4KB `gshare` predictor. As the size of the predictor increase the misprediction rate decreases. For example, 32KB and 256KB `gshare` predictors reduce the misprediction rate for this branch to 9.6% and 6%, respectively.

```
rtlanal.c:1469
 4ceb88:    43 00 00 00    addiu $17,$17,-4
 4ceb90:    43 00 00 00    addiu $18,$18,-1
 4ceb98:    43 00 00 00    addiu $19,$19,-1
 4ceba0:    0a 00 00 00    bgez $19,4cead8 → BR1
```

(a)

```
// rtlanal.c
1469    for (i = GET_RTX_LENGTH (code) - 1; i >= 0; i--) → BR1
1470      {
1471        if (fmt[i] == 'e')
1472          {
1473            if (volatile_refs_p (XEXP (x, i)))
1474              return 1;
1475          }
1476        if (fmt[i] == 'E')
1477          {
1478            register int j;
1479            for (j = 0; j < XVECLEN (x, i); j++)
1480              if (volatile_refs_p (XVECEXP (x, i, j)))
1481                return 1;
1482          }
1483      }
1484  }
```

(b)

Figure 12. Code snippet for second top hot PC in `gcc`, (a) assembly, (b) C

6. Summary of Findings and Discussion

Our source code analysis confirms that wrong history type, insufficient history length, and constant loop counts (e.g., >10) are some of the major causes of branch mispredictions. It also extends the previous work by introducing new class of mispredictions. We have found that array element accesses or pointer references, linked list traversals, varying loop counts, and changing function inputs make harder to predict branches

that depend on them and may require different ways to improve prediction accuracy.

Table 2 partially summarizes the source-code analysis results discussed in Section 5. The table shows the misprediction classification for top 10 hot branches in SPEC CPU 2000 and Mibench benchmarks. Each row in the table shows the number of branches in the top 10 that fall into each misprediction category and their contributions to mispredictions (shown as percentages in parentheses). We can see that `mcf`, `parser`, and `dijkstra` have significant amount of linked-list-traversal caused mispredictions. Especially this is most pronounced for `mcf` (68%). `bzip2`, `ammp` and `jpeg`'s mispredictions are almost all caused by array access or pointer references. All benchmarks except `qsort` and `bitcount` has significant amount of mispredictions caused by array accesses or pointer reference. `basicmath` and `fft` have significant amount of wrong-history branch mispredictions. Changing function inputs is one of the most important causes for mispredictions for most of the `Mibench` benchmarks. `gzip`, `twolf`, `dijkstra`, `patricia`, `qsort` and `fft` contain branches that mispredict because of varying loop counts. Constant loop exit mispredictions is important for `art` and `sha`. Finally, `gcc` has significant amount of insufficient history length mispredictions. Note that all benchmarks have branch mispredictions that fall into each of the categories in the table, however, they may not be in the top 10 list that Table 2 covers.

Based on our source-code analysis, we can summarize our findings as follows:

1. Since few branches correspond to a disproportional amount of mispredictions, it may be worth performing detailed source-code analysis on these hot branches.
2. In many cases, misprediction patterns exist. Most misprediction classes that we observed exhibit some sort of repeating patterns.
3. Some mispredictions are harder to correct than others. Mispredictions due to linked list traversals, randomly varying loop counts, changing inputs to functions are harder (will require more (and different type) history) than other types of mispredictions (conflicts, wrong-type history, constant loops, insufficient history length).
4. Address-value correlation provides some opportunity to correct mispredictions otherwise not possible, especially for linked list traversals and array accesses/pointer references. Data often do not frequently change in addresses. Many examples in benchmark programs.
5. Constant loop exit, insufficient history length, and wrong-type history mispredictions can usually be eliminated with relatively small size of misprediction histories because they often have regular misprediction patterns.
6. Given a constant hardware budget for branch prediction, it may be better to have a combination of branch predictor and a predictor that tracks and reduces mispredictions than having one complicated branch predictor.

7. Related Work

Skadron et al [3] proposed a unified run-time branch misprediction classification taxonomy, which classified mispredictions as being from one of the following categories: training, conflict, wrong-history, need both histories simultaneously, and other. Other category includes insufficient history length or inherently difficult to predict branch mispredictions. In this paper, we extend Skadron's work by performing a detailed source-code analysis of branch mispredictions and introducing new classes of mispredictions. Prior to Skadron's work a number of studies characterized mispredictions although they were not as comprehensive as [3]. Most of the effort has been on characterizing conflicts (PHT interference) [4-11]. Some others studied wrong-history [6, 13, 14, 31] and training times [32, 33]. To reduce constant loop exit mispredictions, loop count predictor was proposed [26].

Table 2. Source-code classification of branch mispredictions

	Array Access Pointer Reference	Linked List Traversal	Wrong Type History	Changing Function Inputs	Varying Loop Counts	Constant Loop Exits	Insufficient History Length	Other
GCC	1 / 52%			2 / 16%			1 / 20%	12%
ART	1 / 7%					1 / 91%		2%
PARSER	2 / 19%	5 / 31%		3 / 50%				0%
MCF	5 / 19%	3 / 68%		2 / 13%				0%
JPEG	9 / 96%					1 / 4%		0%
BZIP2	10 / 100%							0%
BASICMATH	1 / 19%		1 / 40%	2 / 40%				1%
DIJKSTRA	1 / 31%	1 / 33%			1 / 33%			3%
PATRICIA	2 / 22%			6 / 52%	2 / 26%			0%
SHA	1 / 14%					6 / 84%		2%
QSORT				7 / 76%	1 / 23%			1%
BITCOUNT				1 / 99%				1%
SUSAN	5 / 58%			5 / 42%				0%
FFT	2 / 12%		2 / 49%	4 / 29%	2 / 10%			0%
GZIP	4 / 57%			5 / 26%	1 / 17%			0%
VORTEX	8 / 52%			2 / 48%				0%
AMMP	1 / 99%							1%
TWOLF	2 / 5%			1 / 3%	2 / 92%			

Recent works, such as O-GEHL [18] and TAGE [19], and neural-based predictors perceptron [16], PWL [21], Frankenpredictor [23] and others [20, 22, 24] studied ways of exploiting longer history in order to further improve branch prediction accuracy.

8. Conclusion

A great deal of prior branch prediction work has focused on ways to improve prediction accuracy by using local history, global history, a combination of both with a mechanism of how to choose which component to believe, multiple predictors with different histories and sizes, and a number of neural predictors. Most of the improvements to predictor design have come from ad-hoc run-time analysis, e.g, running various configurations of or extensions to a branch predictor.

This paper presents a detailed source-code analysis of important misprediction types to better understand how to further improve branch prediction design. Our analysis show that in addition to the problems – training time, history table interference, wrong-history mispredictions – found through run-time classification of branch mispredictions and well recognized problems such as insufficient history length and constant loops, array/pointer references, linked list traversals, varying loop counts and changing function inputs are also substantial problems for branch prediction. We also show that some mispredictions depend on loads that exhibit address-value correlations, which suggests designing mechanisms that exploits this type of locality. Finally, we show that mispredictions to a branch often have repetitive patterns, which also can be exploited possibly by considering a secondary predictor that targets mispredictions only.

Acknowledgments

This work was supported in part by US National Science Foundation Grant CCF-0541162.

References

[1] http://www.jilp.org/cbp/ First Championship Branch Prediction Competition website, 2004.

[2] http://cava.cs.utsa.edu/camino/cbp2/ Second Championship Branch Prediction Competition website, 2006.

[3] Kevin Skadron, Margaret Martonosi, Douglas W. Clark: A Taxonomy of Branch Mispredictions, and Alloyed Prediction as a Robust Solution to Wrong-History Mispredictions. IEEE PACT 2000: 199-206.

[4] P.-Y. Chang, M. Evers, and Y. N. Patt. Improving branch prediction accuracy by reducing pattern history table interference. IEEE PACT 1996, 48-57.

[5] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge. The bi-mode branch predictor. 30th International Symp. on Microarchitecture, 4-13, 1997.

[6] S. McFarling. Combining branch predictors. Tech. Note TN-36, DEC WRL, June 1993.

[7] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. 24th International Symposium on Computer Architecture, pages 292-303, June 1997.

[8] S. Sechrest, C.-C. Lee, and T. Mudge. Correlation and aliasing in dynamic branch predictors. ISCA 23, 22-32, 1995.

[9] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt. The agree predictor: A mechanism for reducing negative branch history interference. ISCA-24, pages 284-91, June 1997.

[10] A. R. Talcott, M. Nemirovsky, and R. C. Wood. The influence of branch prediction table interference on branch prediction scheme performance. PACT 1995, 89-96, 1995.

[11] C. Young, N. Gloy, and M. D. Smith. A comparative analysis of schemes for correlated branch prediction. ISCA 22, 276-86, 1995.

[12] R. Kessler, "The Alpha 21264 Microprocessor," IEEE Micro, Vol. 19, No. 2, March-April 1999, pp. 24-36.

[13] P.-Y. Chang, E. Hao, and Y. N. Patt. Alternative implementations of hybrid branch predictors. 28th Annual International Symposium on Microarchitecture, pages 252-57, Dec. 1995.

[14] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. ISCA-25, pages 52-61, June 1998.

[15] L. N. Vintan and M. Iridon. Towards a high performance neural branch predictor. In IJCNN'99. International Joint Conference on Neural Networks. Proceedings., 1999.

[16] D. Jiménez and C. Lin, "Dynamic Branch Prediction with Perceptrons," HPCA 2001.

[17] Zhijian Lu, John Lach, Mircea R. Stan, Kevin Skadron: Alloyed Branch History: Combining Global and Local Branch History for Robust Performance. Int. Jrnl. of Parallel Programming 31(2): 137-177 (2003).

[18] André Seznec: Analysis of the O-GEometric History Length Branch Predictor. ISCA 2005: 394-405.

[19] A. Seznec "The L-TAGE predictor", Journal of Instruction Level Parallelism, May 2007

[20] D. Jiménez, "Fast Path-Based Neural Branch Prediction," Int. Symposium on Microarchitecture, 2004.

[21] D. Jiménez, "Piecewise Linear Branch Prediction," Int. Symposium on Computer Architecture, 2005.

[22] A. Seznec, Redundant History Skewed Perceptron Predictors: pushing limits on global history branch predictors, IRISA Report No 1554, 2003

[23] Gabriel H. Loh, "Deconstructing the Frankenpredictor for Implementable Branch Predictors," JILP, vol. 7, pp. 1-10, April, 2005.

[24] Gabriel H. Loh, Daniel A. Jimenez, "Reducing the Power and Complexity of Path-Based Neural Branch Prediction," 5th Workshop on Complexity Effective Design (WCED), pp. 1-8, June 5, 2005.

[25] A. Seznec and A. Fraboulet "Effective ahead pipelining of instruction block address generation," ISCA 30, 2003..

[26] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. Valentine, "The Intel Pentium M processor: Microarchitecture and performance," Intel Technology Journal, 7(2), pp. 21-33, May 2003.

[27] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0," University of Wisconsin-Madison CS Dept. Tech. Report #1342, 1997.

[28] J. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millenium," IEEE Computer, Vol. 33, No. 7, July 2000, pp. 28-35.

[29] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," Workshop on Workload Characterization, 2001.

[30] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," ASPLOS, 2002.

[31] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. ISCA 25, 52-61, 1998.

[32] T. Juan, S. Sanjeevan, and J. J. Navarro. Dynamic history-length fitting: A third level of adaptivity for branch prediction. ISCA 25, 156-66, 1998.

[33] D. Jiménez, "Reconsidering Complex Branch Predictors," HPCA 2003.