Exploiting Intra-function Correlation with the Global History Stack

Fei Gao and Suleyman Sair

Department of Electrical and Computer Engineering,
NC State University, Raleigh, NC 27695, USA
{fgao, ssair}@ece.ncsu.edu

Abstract. The demand for more computation power in high-end embedded systems has put embedded processors on parallel evolution track as the RISC processors. Caches and deeper pipelines are standard features on recent embedded microprocessors. As a result of this, some of the performance penalties associated with branch instructions in RISC processors are becoming more prevalent in these processors. As is the case in RISC architectures, designers have turned to dynamic branch prediction to alleviate this problem. Global correlating branch predictors take advantage of the influence past branches have on future ones. The conditional branch outcomes are recorded in a global history register (*GHR*). Based on the hypothesis that most correlation is among intra-function branches, we provide a detailed analysis of the *Global History Stack (GHS)* in this paper. The GHS saves the global history in the return address stack when a call instruction is executed. Following the subsequent return, the history is restored from the stack. In addition, to preserve the correlation between the callee branches and the caller branches following the call instruction, we save a few of the history bits coming from the end of the callee's execution. We also investigate saving the GHR of a function in the Branch Target Buffer (BTB) when it returns so that it can be restored when that function is called again. Our results show that these techniques improve the accuracy of several global history based prediction schemes by 4% on average. Consequently, performance improvements as high as 13% are attained.

1 Introduction

With an ever growing number of uses and applications, the computation demand on embedded systems has reached new heights. To meet these challenges, embedded microprocessor designers started introducing microarchitectural features such as caches and pipelining which are not common in the cost-conscious embedded domain. Newly released high-end processors routinely feature these techniques [1, 2, 3]. One of the major performance bottlenecks due to pipelining is the branch misprediction penalty. When considering the fact that these high end processors can execute multiple instructions every cycle, branch mispredictions can potentially induce a considerable waste of execution resources (both cycles and power). Furthermore, if the branch depends on a long latency instruction such as a divide or a load that misses in the data cache, the branch resolution time can grow even longer. As a result, accurate branch prediction is the *key* factor in eliminating this penalty.

Global predictors exploit the influence of past branch instructions on future ones. In a typical global predictor, a *Global History Register (GHR)* establishes the correlation between branches. The GHR records conditional branch outcomes and becomes part of the index into the branch prediction table. Consequently the number of history bits held in the GHR is of critical importance in the accuracy of a correlating branch predictor [4]. This is especially true for global predictors that have special features to eliminate negative interference in the prediction tables [4, 5, 6, 7, 8, 9].

One aspect of applications that reduce the effective GHR width is function calls. When a parent function calls one of its children functions, the GHR of the parent is overwritten by the branches in the child. By the time we return back to the parent, the branch outcomes that the ensuing parent branches are most likely correlated with have been wiped out of the GHR. Calder et al. found that on average, C functions execute 133.6 instructions over a wide range of applications [10]. Furthermore, they report that one out of every 9.3 instructions in these programs is a conditional branch. From these numbers we can deduce that approximately 15 conditional branches execute each time a function call takes place. Considering the fact that most branch predictor implementations have GHRs with 16 or fewer bits, a function call overwrites all but one bit of the branch outcomes belonging to the parent. This problem is exacerbated in embedded microprocessors. First, embedded applications feature many more function calls than typical desktop applications. Additionally, embedded processors have relatively shorter (8-bits or less) GHRs.

This paper proposes saving the GHR of the parent function in the return address stack when a function call occurs. Subsequently we restore the GHR using the value in the return address stack upon returning from the callee. This ensures that the conditional branch outcomes generated before the call are still available in the GHR after returning from the call. In case a few of the trailing branches in the callee function influence branches in the caller, we also investigate saving their values instead of clobbering them. Additionally, we evaluate storing the GHR of a function in the Branch Target Buffer (BTB) when it returns and restoring the GHR when the function is called again to look for branch correlation between its consecutive instances.

The contributions of this paper are:

– Analyze the sources of correlation among branches separated by function calls
– Examine correlation between branches in different instances of a function
– Investigate several low cost, low overhead techniques to exploit these two types of correlation
– Provide a detailed performance analysis of these designs and quantify their impact on branch prediction accuracy

The rest of this paper is organized as follows. We show a couple of code examples illustrating why intra-function correlation is hindered by function calls and how correlation across function instances exists in Section 2. Section 3 presents some background into global branch prediction. Section 4 introduces the different mechanisms we employ to preserve and improve intra-function correlation. Next, we describe our simulation methodology and the details of the chosen benchmarks in Section 5. We then discuss the impact of the proposed schemes on prediction accuracy and performance in Section 6. Finally, Section 7 summarizes our findings and concludes.

2 Motivation

There are many techniques that take advantage of correlation among branches [11, 12, 13, 14, 15]. In these schemes, the predictor stores past branch outcomes either in a single register or in a separate entry of a PC-indexed table depending on whether the predictor exploits local (i.e. among instances of the same branch) or global (i.e. among different branches) correlation. Function calls do not pose a problem for local correlation because each branch gets its own entry in the table and aside from interference, the history is not perturbed by other branches. Global prediction accuracy however suffers when the history register contents is lost across function calls. Figure 1 lists a code segment from the SPEC'95 program `gcc`. This loop is within the function `copy_rtx_if_shared` in the source file `emit-rtl.c`. The code includes two loops with many function calls (including recursive ones) inside. The callee functions overwrite the GHR bits from `copy_rtx_if_shared` and makes it very hard to correctly predict the `for` loops which are normally very predictable. A means of recovering the GHR upon returning from a call addresses this problem.

```
for (i = 0; i < GET_RTX_LENGTH(code); i++)
  {
    switch (*format_ptr++)
     {
      case 'e':
        XEXP(x, i) = copy_rtx_if_shared (XEXP(x, i));
        break;
      case 'E':
        if (XVEC(x, i) != NULL)
         {
          register int j;
          int len = XVECLEN(x, i);

          if (copied && len > 0)
            XVEC(x, i) = gen_rtvec_v(len, &XVECEXP(x, i, 0));
          for (j = 0; j < len; j++)
            XVECEXP(x, i, j) = copy_rtx_if_shared(XVECEXP (x, i, j));
         }
        break;
     }
  }
```

Fig. 1. Example code segment from the SPEC'95 `gcc` benchmark. This code exemplifies one problem our paper aims to solve. There are two loops executing many function calls which completely overwrite the GHR bits of the caller

```
while (count>0)
 {
        c2=getranchar(c1,ran2());
        text_buffer[bufindex++]=c2 ;
        ...
      c1=c2;
        count--;
 }
```

Fig. 2. Loop from SPEC'95 benchmark `compress`. `getranchar` function benefits from correlation among branches from different invocations of a function

Meanwhile, recursive calls also exhibit an interesting behavior. Even though there may be correlation among instances of the function, in this particular case recursion hinders the prediction process. Each recursive call results in a different loop trip count depending on the length of the subexpressions being analyzed. This means there are many GHR patterns at the end of these recursions and they each train completely different entries in the predictor table, potentially causing destructive aliasing. Another source of distant correlation is displayed in the `getranchar` function in Fig. 2. `getranchar` takes a character and a random number as arguments and returns another character. This loop is inside the function `fill_text_buffer` in source file `harness.c`. The `while` loop forces the character generated in the previous iteration to be used as the starting point in the current iteration. This results in branch outcomes of the previous iteration affecting the branches in the current one. We can preserve this correlation if we can remember the pattern in the GHR at the end of the previous iteration.

3 Related Work

Recognizing the fact that the outcome of some branches depends heavily on other recent branches, Yeh and Patt proposed the *GAg scheme* which uses a global history register to index into the predictor table instead of the PC [16]. The global history consists of a shift register updated with the outcome of each committed branch instruction. The global history may not provide enough information to distinguish the current branch however. For those branch instructions that do not benefit from global correlation, this results in a worse performance than the bimodal predictor. To overcome this, McFarling proposed the *Gshare predictor* hashing the branch PC with the global history to form the index [12]. He found that the exclusive OR of the branch address with the global history gives more information than either component alone when used as an index.

In general, branch predictor entries are not tagged. Consequently this results in entries being shared by multiple branches. This is referred to as *aliasing* or *interference*. When two branches with opposing biases alias, this results in poor branch prediction accuracy for both branches. To eliminate this *negative interference* a *Skew predictor* was proposed [5, 6]. This scheme uses three two-bit counter tables indexed with different hash functions. The intuition is that even if two branches alias in one table, they will hopefully map to separate entries in the other two tables. Other “de-aliased” branch predictors include the *Bi-mode* predictor [9], *Agree* predictor [7] and the *YAGS* predictor [8]. In principal, the idea is to separate the predictor tables for mostly taken and mostly not-taken branches so that any aliasing will result in neutral interference. The Alpha EV8 processor implements an aggressive branch predictor in 2Bc-gskew [6, 4]. As most recent commercially implemented predictors, 2Bc-gskew is a hybrid predictor. It combines bimodal prediction with a “de-aliased” skew predictor to further improve its accuracy.

In addition to designing new global predictors, another way to improve prediction accuracy is to enhance the correlation among branches. Nair proposed dynamic path-based branch correlation [17] which forms a history of past branches addresses instead of their outcomes. This information is able to represent the execution path resulting in more accurate prediction. In [18], Jacobson et al. proposed a path-based next trace predictor to form sequences of traces to index a trace cache with. They also introduce the return history stack (RHS). The operation of a return address stack (RAS) requires information on an instruction-level granularity. Since traces do not entail this much detail, they can not utilize a RAS. The primary focus of RHS is compensating for the absence of the RAS by improving the predictability of branches after a return. It uses a similar stack like architecture to our GHS to restore global history contents following a return. In this paper, we provide an in depth analysis of the effects of intra-function correlation in the context of superscalar branch prediction in much more detail. Furthermore, we propose an architectural extension to the BTB to store GHR values at the end of functions and we evaluate the implications of preserving correlation across instances of the same function.

Another scheme that potentially increases the effective correlation distance is [15]. In this technique Thomas et al. analyze the dynamic data-flow between instructions to find the producers (direct and indirect) of the values used in branch instructions. Next, they determine the branches that these producer instructions are control dependent on.

This process yields a set of branches called *affectors* that directly or indirectly *affect* the computation of values consumed by branch instructions. Hence a particular branch is most likely to be correlated with its affector branches.

4 Proposed Architectural Extensions

There are two aspects of intra-function that we can preserve: 1) the GHR of the parent function through function calls, 2) the GHR of any function across different invocations. We will now detail the mechanisms that we utilize to achieve these goals.

4.1 Global History Stack

In order to preserve the branch outcomes before a function call, we need a temporary storage to save the GHR. The return address stack (RAS) fits this purpose perfectly as it saves the return address under the exact conditions that we want the GHR maintained. We can simply extend each RAS entry with a field to hold the pre-call GHR. This field would be populated together with the return address field on a call and freed when the RAS entry is popped following a return.

Let us illustrate the operation of the GHS with an example. Consider the code segment shown in Fig. 3, which shows three functions. The instruction marked as 1 causes the call instruction at the return address ($0x12004CA8$) and the current GHR (the 10-bit binary value 1000111101) to be pushed on to the next available entry in the RAS. This operation is marked with circle 1 in Fig. 3. Subsequently when the second call instruction is executed, a similar series of events take place and the return address, history pair of ($0x12005F50$,0100101110) is inserted into the RAS (operation 2 in Fig. 3). In the function starting at address $0x12006684$, there are two conditional branches of which the first one is taken and the second one is not taken. With this assumption, the GHR has the value 0010111010 by the time we reach the return instruction as shown in Fig. 3. When the return instruction executes, we pop the RAS, obtaining the predicted target of the return as well as the new value of the GHR (operation 3 in Fig. 3).

For functions that are called from multiple call sites different paths lead to the call site. This results in a different GHR pattern each time the function is called from a different call site. Especially when functions are relatively short, such as in C++ programs where on average there are 5 conditional branches per function invocation [10], this hinders the predictability of these branches because separate two-bit counters need to be trained for each different path. One approach to resolve this issue is clearing the GHR each time a function is called. We call this *zeroing*.

Note that restoring the GHR to its pre-call state can actually hurt prediction accuracy if branches subsequent to returning from a function are dependent on some of the branches in the callee. In this scenario, conditional branch instructions executed prior to returning from the callee influence the return value of the function. Hence upon returning from the callee, when the caller uses the return value as a predicate to conditional branch instructions, it would find it beneficial to correlate with the branches that the return value is control dependent on. These branches are called the *affector* branches in [15]. For this purpose, we evaluate preserving a few of the most recent branch outcomes of the callee function when we restore the GHR using the value in the RAS.

```
1  0x12004CA4    call  0x12005F48
   0x12004CA8    ...

   0x12005F48    ...
   ...
2  0x12006F4C    call  0x12006684
   0x12006F50    ...

   0x12006684    ldq_u   t1, 0(a0)
   0x12016688    bne     t1, 0x12016690 // taken
   0x1201668c    addq    v0, 0x4, t0
   0x12016690    ldq     t0, 8(v0)
   0x12016694    addq    v0, 0x8, v0
   0x12016698    cmpbge  zero, t0, t1
   0x1201669c    beq     t1, 0x12016680 // not taken
   0x120166A4    subq    zero, t1, t2
3  0x120166A4    ret
```

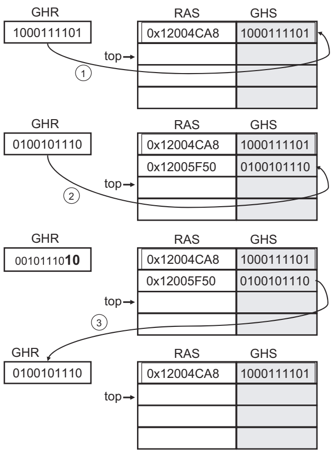

Fig. 3. Operation of GHS: Code example and the corresponding GHR and GHS values when the code executed

4.2 Preserving GHR Values Across Function Invocations

Zeroing solves the problem of having different GHR values when a function is called from different call sites. However, by clearing out the GHR on each call, it prevents taking advantage of any correlation. We can improve the predictability of these branches if we *remember* the GHR from the past invocation of that function.

As shown in Section 2 (recall Fig. 2), there are various examples of functions that have loops and other conditional nests that can benefit from the history of previous instances of these branches. The intermittent branch instructions between two invocations of a function clobber the GHR however. This results in lost correlation opportunities. To mitigate this problem we propose saving the GHR value in the BTB before returning from a function. The next time the same function is called, we can restore the value from the BTB and utilize the past history of branches in the function.

Recall the two `for` loops in Fig. 1. Remembering the histories from one instance of the function to the next can improve the predictability of these loops by training separate predictor entries to reflect the steady state behavior as well as the termination of these loops. To this end, we add an *old history* field to the BTB. When we return from a child function, we update the BTB entry corresponding to the parent's call instruction with the current GHR value. We can simply obtain the PC of the call instruction by subtracting one (actually the size of one instruction) from the return address. The next time that call instruction is fetched, we check the BTB. On a tag match, if the call bit in the BTB is set, we will fill the GHR using the value in the old history field of the BTB. Note that this technique associates the previous history with a particular call site, not a function. In other words, if a function is called from multiple places, the history that will be reloaded into the GHR will be coming from the previous instance of the function when called from the same particular call site. We can store the beginning PC of a function when we enter it in a temporary register and use that PC to store the GHR before returning. But that would potentially introduce non-branch instructions into the BTB and reduce its effective size. Instead of creating a separate table for saving function GHR instances, we chose the simpler approach of adding a field to the BTB and maintaining regular BTB semantics.

5 Methodology

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [19], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction. Latency values for the caches and register files were obtained using CACTI [20] for a *130nm* process technology.

Table 1. Baseline misprediction rates

Benchmarks	com	gcc95	go	ijpeg	li	vor95	crafty	gcc2K	twolf	vor2K
% Mispred	18.9	10.6	27.6	13.2	6.1	5.4	10	12.2	15.2	5.4

To perform our evaluation, results were collected for 10 of the SPEC95 and SPEC2000 integer benchmarks that were similar to typical embedded programs and had higher than a 5% branch misprediction rate with a 4K entry gshare predictor (see Table 1. These were `compress`, `gcc`, `go`, `ijpeg`, `li`, and `vortex` from SPEC95, and `gcc`, `twolf`, and `crafty` and `vortex` from SPEC2000 suites respectively. The programs were compiled on a DEC Alpha AXP-21164 processor using the DEC C and C++ compilers under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`). We skipped the initialization of each program by skipping 500 million instructions (except for `gcc` from SPEC'95 which executes for fewer than 500 million instructions) and simulated for 100 million committed instructions. All benchmarks were simulated using the *ref* inputs. We evaluate the effectiveness of the proposed intra-function correlation enhancements on a gshare predictor. The predictor has 4K entries and uses an 8-bit global history. The performance analysis models a next generation embedded processor similar to the configuration of ARM11 [1]. The processor can execute 4 instructions every clock cycle. It includes a 10 entry return address stack and a 512 entry BTB.

6 Results

We present the results of our experiments on the efficacy of the proposed techniques in this section. All the results except for the baseline configuration utilize zeroing. In these results GHS refers to adding a history field to the RAS to restore the GHR across function calls. `GHS+r6` is overwriting 6 most significant bits of the GHR (i.e. retaining the last 2 bits) in addition to GHS. Results for the combination of GHS and adding an old history field to the BTB in order to remember the GHR value from the previous

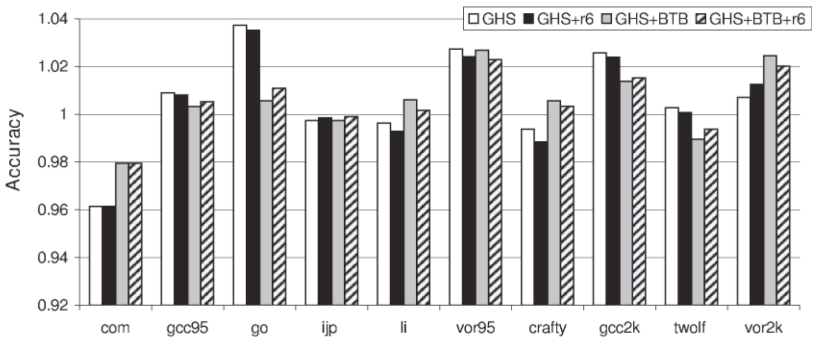

Fig. 4. Normalized branch prediction accuracy of different intra-function correlation techniques used in conjunction with a gshare predictor. The gshare predictor has a 4K entry table and 8 bit global history. The results are normalized to the baseline case

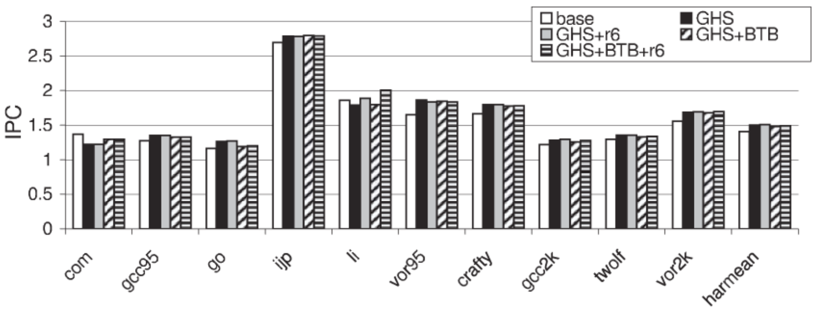

Fig. 5. Performance results for a gshare predictor with 4K entries and a 8-bit global history for different intra-function correlation schemes

iteration of a function are listed as `GHS+BTB`. And finally adding retaining to this last combination is referred to as `GHS+BTB+r6`. For the BTB cases if the beginning PC of the function is not found in the BTB, the GHR is initialized to 0.

We show the prediction accuracy for each benchmark normalized to the baseline predictor without the intra-function correlation enhancements. Figure 4 quantifies how effective we are in exploiting intra-function correlation for a 4K entry gshare predictor with 8 bits of history. We can observe several trends in this figure. Applications such as `gcc`, `go` and `twolf` perform better without the addition of the history field to the BTB. In these benchmarks the GHR patterns show great variation from one invocation of a function to another, rendering restoration of the GHR from the BTB ineffective. In this case zeroing proves better as in each instance of the function the history will start from 0. Contrast this behavior to those of `compress`, `li`, `crafty` and `vortex`. In these applications, the `GHS+BTB` combinations outperform GHS alone. Even though `compress` gets a big boost from the addition of BTB extensions, it still performs worse than the baseline gshare predictor. This is primarily attributable to situations where a conditional branch is directly data dependent on the return value of a function. Since we restore the GHR to its pre-call state, the opportunity to exploit this correlation is lost. We experimented with longer retaining values for `compress` and found that prediction accuracy is restored back to the same level as the baseline predictor. Despite the techniques working well on some programs and worse on others, the average misprediction rate is reduced by 4%. To attain best possible results, we are currently investigating adaptive application of GHS techniques where the compiler determines which functions have access to the GHS.

We also measured the performance of these different schemes on a next generation embedded processor with a gshare predictor. Figure 5 displays the number of committed instructions per cycle (IPC) for a 4K entry gshare predictor with 8 bits of history. The last set of bars represent the harmonic mean of the IPC for all benchmarks. On average the `GHS+BTB+r6` configuration improves performance by 5%. Individually, `vortex` enjoys a performance boost of 13% from the GHS while `compress` suffers a slowdown of 11% in the worst case.

7 Conclusions

Global branch prediction is a powerful tool in tolerating control related stalls in a pipelined processor. Whether as a stand-alone predictor or as part of a hybrid predictor, it is extensively used in current processor designs. In global prediction, correlation is established through a Global History Register (GHR). The limited size of the GHR causes callee functions to overwrite the GHR of the parent function, thrashing correlation. Furthermore, branches from previous invocations of a function can influence the direction branches in the current instance will take. Remembering the branch history when the function was executed the last time can improve branch prediction accuracy by cutting down on training time through providing a steady starting point for each invocation of a function.

In this paper, we proposed several intra-function correlation preservation mechanisms. The first of these, the Global History Stack (GHS), saves the history information of the parent when it calls another function into the return address stack (RAS). When the callee finishes and returns, we pop the history value from the RAS. We introduced *zeroing* and *retaining* as means of providing stable starting points for functions and preserving callee-to-caller correlation. Finally, to promote the inherent control dependence across branches in consecutive instances of a function, we proposed saving the global history register (GHR) at the time of the return in the BTB, only to be restored when that function is called again.

The proposed techniques provide, on average, a 4% reduction in misprediction rate. Absolute reduction in misprediction rates is as high as 3% in the case of `go`. Performance improvements as high as 13% (for `vortex95`) are observed. In general, the fact that some programs benefit from our techniques while others do not encourages future research in application specific use of the GHS and the BTB extensions. One compiler solution is identifying branches that benefit from intra-function correlation and apply these techniques only to those functions. This can be done via profiling as was done in [21]. Runtime techniques similar to the methods used in [15] can also help, where we analyze the register and control dependencies to determine correlating branches.

References

1. ARM Ltd.: ARM1136 Technical Reference Manual, Version r0p2. (2004) http://www.arm.com.
2. Intel Corp.: The Intel(R) XScale(TM) Microarchitecture Technical Summary. (2000) http://www.intel.com/design/intelxscale/.
3. Analog Devices Inc.: Analog Devices Blackfin Processor Data Sheet. (2005) http://www.analog.com/processors/processors/blackfin/.
4. Seznec, A., Felix, S., Krishnan, V., Sazeides, Y.: Design tradeoffs for the Alpha EV8 conditional branch predictor. In: Proc. Ann. Int. Symp. Comput. Architecture. (2002) 295–306
5. Michaud, P., Seznec, A., Uhlig, R.: Trading conflict and capacity aliasing in conditional branch predictors. In: Proc. Ann. Int. Symp. Comput. Architecture. (1997)
6. Seznec, A., Michaud, P.: De-aliased hybrid branch predictors. Technical Report RR-3618, Inria (1999)
7. Sprangle, E., Chappell, R., Alsup, M., Patt, Y.: The agree predictor: A mechanism for reducing negative branch history interference. In: Proc. Ann. Int. Symp. Comput. Architecture. (1997) 284–291
8. Eden, A.N., Mudge, T.: The YAGS branch prediction scheme. In: Proc. Ann. ACM/IEEE Int. Symp. Microarchitecture. (1998) 69–77
9. Lee, C.C., Chen, I.C., Mudge, T.N.: The bi-mode branch predictor. In: Proc. Ann. ACM/IEEE Int. Symp. Microarchitecture, Research Triangle Park, NC (1997) 4–13
10. Calder, B., Grunwald, D., Zorn, B.: Quantifying behavioral differences between C and C++ programs. Journal of Programming Languages **2** (1994)
11. Yeh, T.Y., Patt, Y.: Two-level adaptive branch prediction. In: Proc. Ann. Int. Symp. Microarchitecture. (1991)
12. McFarling, S.: Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab (1993)
13. Sechrest, S., Lee, C.C., Mudge, T.: Correlation and aliasing in dynamic branch predictors. In: Proc. Ann. Int. Symp. Comput. Architecture. (1996) 22–32
14. Evers, M., Patel, S.J., Chappell, R.S., Patt, Y.N.: An analysis of correlation and predictability: What makes two-level branch predictors work. In: Proc. Ann. Int. Symp. Comput. Architecture, Barcelona, Spain (1998) 52–61
15. Thomas, R., Franklin, M., Wilkerson, C., Stark, J.: Improving branch prediction by dynamic dataflow-based identification of correlated branches from a large global history. In: Ann. Int. Symp. Comput. Architecture, San Diego, CA (2003) 314–323
16. Yeh, T.Y., Patt, Y.: Alternative implementations of two-level adaptive branch prediction. In: Proc. Ann. Int. Symp. Comput. Architecture. (1992)
17. Nair, R.: Dynamic path-based branch correlation. In: Proc. Ann. Int. Symp. Microarchitecture. (1995) 15–23
18. Jacobson, Q., Rotenberg, E., Smith, J.E.: Path-based next trace prediction. In: Proc. Int. Symp. Microarchitecture. (1997) 14–23
19. Burger, D.C., Austin, T.M.: The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, WI (1997)
20. Shivakumar, P., Jouppi, N.P.: Cacti 3.0: An integrated cache timing, power, and area model. Technical Report (2001)
21. Stark, J., Evers, M., Patt, Y.N.: Variable length path branch prediction. In: Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems. (1998) 170–179