A Multi-component Branch Predictor Design for Low Resource Budget Processors

Moumita Das[1,2] Ansuman Banerjee[1] Bhaskar Sardar [2]
[1] Indian Statistical Institute, India [2] Jadavpur University, India

Abstract—**In this paper, we study the problem of designing branch predictors for an embedded processor with low resource budget. This is quite significant in the context of embedded systems or edge devices, which are being increasingly used today as compute nodes in the context of edge and fog computing. A branch predictor forms a crucial component of any modern processor, and we show that a multi-component predictor is the best for achieving high accuracy of branch prediction. We first examine a multi-component predictor design where the individual predictor components are made to share the predictor table structures. Our experiments reveal that this sharing often leads to a loss of prediction accuracy due to the extensive interference between the predictors on the shared data structures they operate on. We propose a simple modification to contemporary multi-component predictor designs to improve accuracy. We present our findings on the SPEC 2006 benchmarks.**

I. Introduction

In modern pipelined processors, a branch predictor is employed in the fetch stage to predict the direction of a branch instruction so that the normal flow of a pipeline can function and a new instruction can be fetched every cycle. However, a misprediction leads to wastage of effort and loss of compute cycles, since the entire pipeline has to be flushed and new instructions need to be brought in. Embedded processors with pipelines face a severe difficulty, since mispredictions have a toll on energy consumption, due to the extra overhead needed for restoring the execution to the correct path by flushing the pipeline and fetching new instructions from the alternate path on a misprediction. Evidently, efficient prediction policy design has always remained an important problem for researchers in computer architecture [1] [4] [6] [9] [15] [16]. In this paper, we study the predictor design problem for low resource budget processors in embedded computing. We show that it is possible to reach significant prediction efficiency while restricting to low storage. We believe our work has an important application in resource constrained embedded computing elements.

Modern pipelined processors today have quite efficient implementations of multiple predictors, with a reasonably high level of prediction accuracy through widely varying algorithms for learning branch direction patterns. These predictors typically use history tables to store the direction histories of the executing branches in a program. This history information is used to learn the branch direction patterns, and later used while making a prediction for a specific branch. An important observation [6] in predictor design literature has been the fact that different predictors perform differently on different branches in terms of the misprediction metric, in other words, branches which are ill-suited for a certain branch prediction policy often have better performance when run with another predictor. A single predictor component, therefore, may not be well suited for all branches in a program, thereby necessitating the idea of hybrid branch prediction. In this work, we study the problem of designing multicomponent hybrid prediction techniques for resource constrained environments. In particular, we explore if multiple predictors can be made to share the same prediction history table, thereby saving storage.

Our experimental findings on employing a multicomponent hybrid predictor with individual predictor components sharing the same predictor table on the SPEC 2006 benchmarks [7] revealed an interesting finding – the accuracy gain expected with the hybrid scheme is most often not achieved. This happens due to extensive interference on the shared predictor table, due to frequent switching between the predictors. The prediction information of one predictor is overwritten by another predictor during its prediction. This negative interference often leads to an incorrect direction for prediction, and therefore, accuracy degradation. In this paper, we propose a heuristic that attempts to improve a classical hybrid prediction mechanism by minimizing the number of context switches to different predictor components, with an objective of interference reduction. We show results of using our heuristic on top of the shared table implementation. Our heuristic attempts to control the amount of predictor interference by controlling the number of instances of new predictors being employed for prediction. We show an average prediction accuracy improvement of 3-4% on the SPEC 2006 benchmarks.

This paper is organized as follows. Section II presents a discussion on the background of the different predictor designs. Section III presents a motivating use-case to illustrate the problem of interference witnessed in multi-component predictor designs. Section IV presents a detailed analysis of the complete solution space of multi-component predictor designs, along with our proposal. Section V presents experimental results, while Section VII concludes this discussion.

II. Background

Classical research on branch prediction has led to two different classes of predictors based on the program life cycle they operate at, namely (a) *static* predictors [17] [26], which work before the program is in actual execution, and (b) *dynamic* predictors [5] [9] [14], which reside inside the processor and are employed at run-time. Static prediction techniques typically aim at designing improved prediction strategies that either use efficient program analysis [3] [17] to design branch prediction hints (often, using hint bits in the program executable) to be used at run time [24], or use execution profiling to learn branch directions [18]. Dynamic predictors, on the other hand, maintain and manipulate efficient prediction policies at run-time when the program is in actual execution. Evidently, dynamic policies are more effective, since they work when the program is in actual execution, something which is hard to mimic for their static counterparts. Indeed, modern pipelined processors today have quite efficient implementations of multiple dynamic predictors working in unison, with a reasonably high level of prediction accuracy and varying energy footprints. Some of the popular dynamic predictors include Bimodal, GAg, GShare, PAp [14], Perceptron [10], TAGE [23] its variants [20] [21]. Another orthogonal classification of dynamic predictors is based on the information that they use about other branches for predicting a specific branch at run-time. A *local* dynamic predictor (e.g. PAp) uses history information only about the branch under consideration for its current prediction, while a *global* history-based dynamic predictor (e.g. Perceptron) takes into account the direction histories of the preceding branches in addition to the present one while making a prediction for a specific branch. The history information has to be stored and manipulated upon inside a processor, thereby motivating researchers to look at energy and space efficient designs for these predictor tables.

Multicomponent hybrid predictors with multiple predictor components, with varying algorithms, have been well studied in the literature, with a number of design strategies, attempting to improve prediction accuracy and power consumption [1] [2] [4] [8] [9] [15] [16]. A single predictor design uses a single predictor for all branches in a program during execution. A hybrid predictor uses multiple components and for each branch, it uses the best predictor that provides maximum prediction accuracy and leads to minimum misprediction for that branch. Evidently, there is an increase in branch specific prediction accuracy and overall, prediction accuracy as well. This best predictor selection can be done statically (based on the static profiling step) [4] or dynamically (based on information recorded at runtime). The Tournament predictor [12], maintains multiple predictor components at run-time, queries each component for their prediction for each branch, and finally selects the best predictor to use, based on past performance of these. Internally, these predictors use a *choice predictor* component that maintains a running counter to keep track of each predictor's performance for each branch and overall, till that point, based on which, it decides whose prediction to go forward with. The Overriding predictor [11] [22] usually comprises of a combination of low response time (and usually, less accurate) and high response time (and possibly more accurate) predictors. These predictors typically start off with the prediction of a low response time predictor, and overrides the decision, in case the prediction from the high response time and expected to be more accurate component, differs from the former. All this is done at execution time.

In this work, we use a multi-component hybrid predictor design that includes GAg, GShare and Bimodal predictors. These predictors [14] use two main data structures - a Pattern

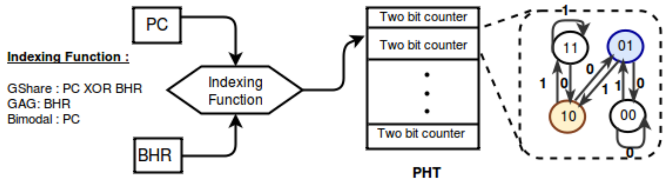

Fig. 1: Internal Architecture of a Branch Predictor

History Table (PHT) and Branch History Register (BHR) as shown in Figure 1. Here, BHR is a n bit shift register that contains the branch outcomes of the most recent n branches. In BHR, a 1 is recorded for a taken branch for a taken branch, and a 0 is recorded for the not taken branch. PHT stores the prediction information for all branches of a program. Each PHT entry contains a two bit saturating counter, the MSB of that counter gives the final prediction. When a branch condition gets resolved, the states of the two bit counter and the BHR content are updated with the actual branch outcome. The indexing functions used by the predictors to access the prediction corresponding to a branch as stored in the associated PHT entry are different for each predictor discussed here. GAg uses BHR to index the PHT, whereas GShare uses an exclusive-OR of the branch address and the BHR to index the PHT. The Bimodal predictor uses only the Program Counter (PC) value to access the PHT entry, as shown in Figure 1. Different predictors use different indexing functions, to access the PHT. The indexing function reflects the intrinsic dependence of a branch, which may be entirely on itself (local dependence), on preceding branches (global) or a mix of them. The indexing function is appropriately selected to include combinations of the required elements, usually, the PC, local history and global history [14].

III. Problem Overview

In this section, we illustrate an overview of the interference problem on a fragment of the *mcf* program of the SPEC2006 [7] benchmark, as shown in Figure 2. All figures and tables used here were generated by running *mcf* with different predictors on a small framework designed on top of the Tejas [19] architectural simulator. We first explain the performance of GAg and GShare [14].

Table I presents the misprediction rate obtained, averaged over multiple simulation runs on the standard simulation data provided as part of the benchmarks, for 7 branches of *mcf* shown in Figure 2 when a single dynamic predictor is used for branch prediction. GAg provides lowest misprediction rate for branches 1, 2, 4 and 6, whereas GShare provides the same for branches 3, 5 and 7. Quite evidently, the misprediction rate / prediction accuracy for a particular branch varies for different predictors which substantiates the fact that no single predictor performs best for all the branches. This motivates

```
 long read_min( network_t *net )
{
  ...
  if(( in = fopen( net->inputfile, "r")) == NULL )
                                         //branch 1
        return -1;
  ...             /*Non-branch statements*/
  if( sscanf( instring, "%ld %ld", &t, &h ) != 2 )
                                        //branch 2
        return -1;
  ...             /*Non-branch statements*/
  if( net->n_trips <= MAX_NB_TRIPS_FOR_SMALL_NET )
                                        //branch 3
    {
      net->max_m = net->m;
      net->max_new_m = MAX_NEW_ARCS_SMALL_NET;
    }
  ...            /*Non-branch statements*/
  if( !( net->nodes && net->arcs && net->dummy_arcs ) )
                                     //branch 4, 5,6
    {
      printf( "read_min(): not enough memory\n" );
      getfree( net );
      return -1;
   }
  ...           /*Non-branch statements*/
  if( sscanf( instring, "%ld %ld", &t, &h ) != 2 || t > h )
                                      //branch 7
            return -1;

  ...         /*Non-branch statements*/

}
```

Fig. 2: Program P: A fragment of *mcf*

use of a hybrid predictor (HP). For each branch, HP selects the predictor with the lowest misprediction rate / maximum prediction accuracy (we call it the best predictor) for that branch, as obtained from Table I. The last column of this table presents the predictor selected for each branch. HP switches to a different predictor when the current active predictor is not the best for that branch. As an example, HP selects GAg for branch 6 and switches to GShare for branch 7. Figure 4 presents the prediction accuracy degradation for shared PHT table implementation with respect to GAg, GShare and Bimodal predictors.

Branch	Branch Address	BHR Pattern	Misprediction rate(%)			Predictor Invoked (HP)
			GShare	GAg	Hybrid	
Branch1	401999	110101010	12.0	10.0	10.0	GAg
Branch2	4019d6	101010101	7.0	6.0	6.0	GAg
Branch3	401a17	010101010	3.6	4.0	10.0	GShare
Branch4	401a91	101010101	11.0	10.0	15.0	GAg
Branch5	401a9a	010101011	0.10	3.0	0.10	GShare
Branch6	401aa3	101010111	10.3	10.0	10.0	GAg
Branch7	401cd9	010101110	2.0	5.0	7.0	GShare

TABLE I: Branch profile for *mcf* program branches

We then experimented with a single PHT to be used by both GShare and GAg, as summarized in Figure 3. Table I Column 6 shows the result for the 7 branches run with a hybrid predictor that combined GShare and GAg together. As observed, there is a loss in prediction accuracy in some cases due to the interference between predictors.

We present interference statistics on some SPEC 2006 benchmarks in Table II. For a hybrid predictor with a shared 32KB PHT, Column 2 shows the percentage of entries suffering from interference (access and modification by the different components) with respect to all PHT accesses. Figure 4 shows the difference in average prediction accuracy between three single stream predictors (GAg, GShare and Bimodal) and the shared table hybrid predictor implementation. It is seen, for most of the cases the bars are in the positive direction, i.e. there is an accuracy fall with the shared implementation. These observations motivated us to address this interference.

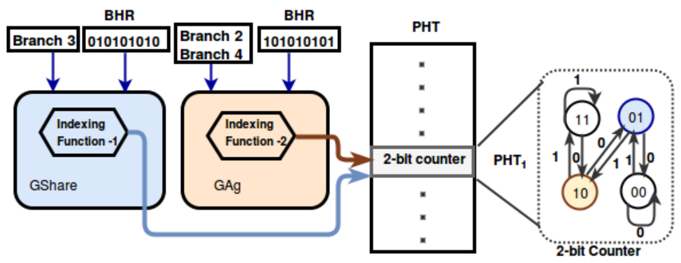

Fig. 3: Predictor table interference between two predictors

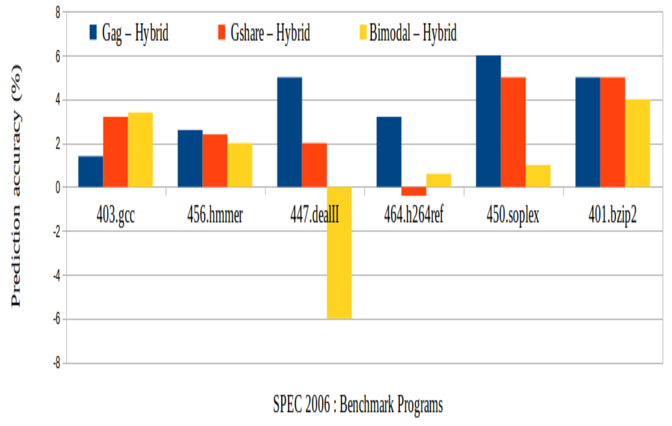

Fig. 4: Accuracy comparison: hybrid and single predictors

IV. Proposed Solution

In this section, we present our proposal in greater detail. We begin by illustrating the solution complexity of the problem space, from the perspective of designing a hybrid predictor scheme that can maximize the overall end-to-end prediction accuracy for a given program. For ease of illustration and simplicity of explanation, we consider a hybrid predictor that combines 4 individual predictors: GShare (P1), GAg (P2), TAGE (P3) and PAp (P4) together and a program P with 4 branches: Branch$_1$, Branch$_2$, Branch$_3$ and Branch$_4$ (in this order of their occurrence in P), any of which can be predicted using any of the 4 predictors as shown in Figure 5.

Figure 5 shows a level-ordered 4-ary tree that shows all possible predictor combinations for all the four branches in the way the branches are ordered in the original program. In this case, this tree has 4 levels, where each level represents an individual branch of this program (and the different predictor

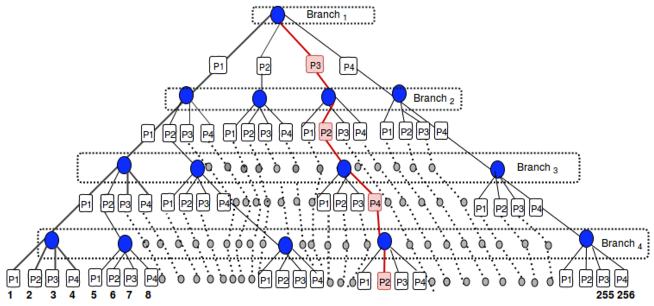

Fig. 5: Predictor Sequence Tree

ALGORITHM 1: Selective Switching

```
begin
    Find the predictor $\mathcal{R}$ with maximum prediction accuracy for a program $\mathcal{Q}$
    Initialize currentPredictor $\mathcal{C}$ = $\mathcal{R}$
    for each branch $i$ in $\mathcal{Q}$ do
        Let $P_i$ be its best predictor if available, or else $P_i$ = $\mathcal{R}$.
        if $|A_{\mathcal{C}} - A_{P_i}| < \theta$ then
            Select $\mathcal{C}$ for prediction of $i$
        else
            Select $P_i$ for prediction of $i$
            Update $\mathcal{C}$ = $P_i$
end
```

choices for that branch) and 256 different paths to represent 256 different predictor sequences. Intuitively, this tree captures all the possible ways a hybrid predictor scheme can be designed using individual predictors. The path marked in red is the predictor sequence generated by the traditional hybrid predictor which works by selecting the best predictor for every branch based on prediction accuracy for that branch.

As illustrated in the previous section, a hybrid scheme with shared PHT table, that picks the best predictor for each branch is not necessarily the best choice in terms of overall prediction accuracy for the entire program, and hence, the need for this exhaustive enumeration. A hybrid predictor should ideally explore all the paths of this tree to get all possible predictor sequences, compute the average prediction accuracy of each path and choose the best that maximizes it. Thus, all possible interleaving of the predictors (and the resulting interferences) would be automatically considered, and a choice can be made that minimizes the negative interference as much as possible, while also maximizing prediction accuracy.The exhaustive enumeration described above is however, beyond scope for any realistic implementation. The following subsection presents our proposal for interference reduction.

Selective Switching based Interference Control

We propose to reduce predictor interference using selective switching as described in Algorithm 1 below. The philosophy of Algorithm1 is to find the best predictor $\mathcal{R}$ that has the best overall prediction accuracy for a program and the best predictor P_i for each branch i, obtained from previous simulation runs or earlier phase of execution. We start with $\mathcal{R}$ as the current predictor $\mathcal{C}$. Let $A_{\mathcal{C}}$ denote the prediction accuracy of $\mathcal{C}$, and A_{P_i} likewise denote the prediction accuracy of predictor P_i. We elaborate on our choice of these in the following section. For each branch, if the best predictor is not $\mathcal{C}$, it checks whether the prediction accuracies of the current predictor and P_i differ beyond a threshold value (say θ). If this condition is true, it selects that best predictor for prediction and updates the current predictor accordingly. Otherwise, it continues with the predictor $\mathcal{C}$ for prediction.

Consider the *mcf* program (Figure 2) and a threshold of 0.5. Let us assume GAg is the overall best predictor to start with. From the information given in Table I, for branch 1, the best predictor is the same, and we continue with GAg as the current best predictor and use it for prediction of this branch. A similar thing happens for branch 2 as well. For branch 3, the best predictor is GShare, however the gain in prediction accuracy is less than our threshold. Hence, we continue with GAg for prediction. For branch 4, the best predictor is GAg as well. For branch 5, the best predictor is GShare and the difference in accuracy gain is beyond 0.5, hence we change the current best predictor and switch to GShare. For branch 6, the best predictor is GAg, however, we do not switch to GAg since the accuracy gain is less than θ. We continue using the same predictor for branch 7 as well, since GShare is the best for it. The number of predictor switches (GAg→GAg→GAg→GAg→GShare→GShare→GShare) in our proposal is 1, while in a classical hybrid scheme (as shown in Column 7 of TableI), it is 5 (GAg→GAg→GShare→GAg→GShare→GAg→GShare). Thus the total number of interfering predictors is less than the classical scheme. This also facilitates in reducing the number of inter-predictor interferences on a shared-table implementation, which was the main motivation behind this work. It may be noted that the worst case number of predictor switches in our case is upper bounded by the number of switches in a classical scheme.

V. Implementation and Results

We now present details of our experiments. All simulations were run on top of the Tejas [19] architectural simulator. Tejas is an open source, Java-based multicore architectural simulator, with support for both trace driven and execution driven simulation. After execution, it reports various statistics related to cache utilization, branch prediction accuracy, energy expenditure, etc. Tejas internally employs the McPAT power model (v1.0) [13] to compute the processor core energy that it reports after each simulation, assuming a 32nm technology. To mimic an embedded resource constrained environment, we took a pipeline depth of 5, with a PHT table size of 32 KB, and 3.4 GHz core frequency. Additionally, we disabled the out-of-order-execution, VLIW features. We modified the Tejas simulator source code to implement a hybrid predictor that includes a combination of three predictors (GShare, GAg and Bimodal). In this paper, execution driven simulation was done for the SPEC 2006 benchmarks [7]. To limit simulation time, the first 1 billion instructions from each benchmark were simulated. Tejas was used to record the branch behaviors for the benchmark programs across all test cases provided and we recorded the average performance numbers. Each program was run with different branch predictors and the prediction accuracy for all the branches of the program on every predictor was recorded. For each branch, we marked a best predictor and for the overall program in terms of prediction accuracy.

Our predictor selection algorithms were implemented on top of the hybrid prediction method that shares the PHT table among the predictors to improve the overall performance of a processor. As input, these methods take the branch profile information for all branches from the profile generation stage as discussed above. The simulation of the new selection mechanism was done using Tejas. For each execution, prediction accuracy, energy expenditure and latency were recorded for comparing against other hybrid prediction techniques (without selective switching) that either share the PHT table among interfering predictors or split the PHT table for them.

Benchmark	Interference (in shared implementation) (%)	Interference (using switching algorithm) (%)
403.gcc	3.7	2.2
400.perlbench	4	3.3
429.mcf	1.2	0.5
458.sjeng	1	0.2
456.hmmr	1.1	0.05
447.dealII	1	0.1
464.h264ref	2	0.7
450.soplex	2.5	1.2
401.bzip2	3.4	0.4

TABLE II: PHT interference statistics with our method

We now report the results of our methods obtained with a threshold of 0.5. Column 3 of Table II presents the percentage interference counts with our proposed switching method, with respect to the original shared table implementation. It can be seen that there is a reduction in % of interferences for almost every program used here. Figures 6, 7 and 8 respectively present the improvements in average prediction accuracy, energy and execution time for a shared 32KB PHT hybrid predictor implementation with selective switching, and ii) a split PHT table of size 16KB for every individual predictor, with respect to the original 32KB shared PHT table implementation. It can be observed from Figure 6 that the original shared PHT table implementation performs much worse than each of the above schemes, hence the differences are positive. Even for most of the programs, the prediction accuracy obtained from the shared PHT table implementation is even lower than the accuracy of any single predictor. Our experiments show that our proposed switching algorithm that works on a shared PHT table can improve the prediction accuracies as expected. Although the average accuracy improvement is not so high, around 2%-3%, however, a single misprediction can increase a significant amount of processor cycles as well as extra instruction fetches. Figure 7 shows that there is an increase in energy expenditure in the shared implementation and our switching method can reduce the energy expenditure for all the benchmark programs. Figure 8 presents the same detail with respect to execution time. It may be noted that a lower value of execution time is more desirable, and our heuristic indeed achieves comparable or marginally less at times. From the experiments, it can also be seen that our selective switching method when employed on top of a shared-table implementation produces better accuracy than the split table implementation for all the performance metrics discussed here.

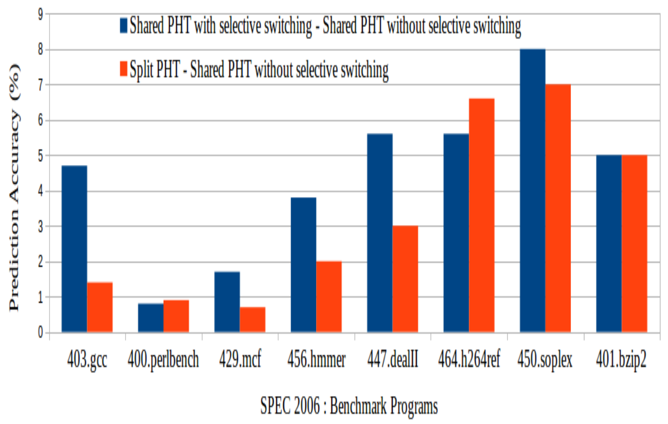

Fig. 6: Prediction accuracy comparison

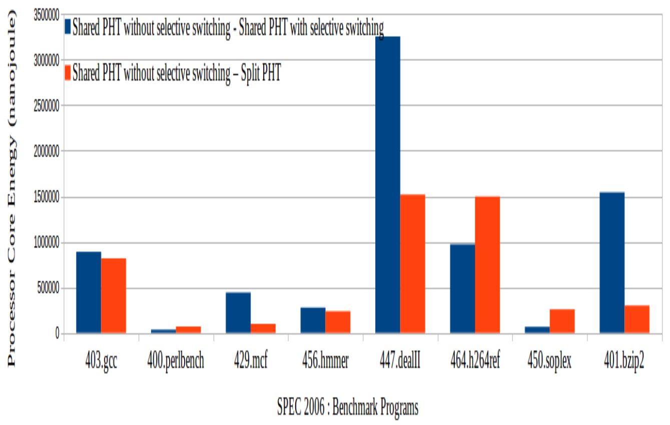

Fig. 7: Processor core energy comparison

VI. Discussion

In this paper, we utilize a static profile based selection scheme to select the overall best predictor with highest average prediction accuracy for a program and also for each branch. We use this profile information to control the number of predictor switches. This requires a good set of representative test-cases that can exercise the different branches with different conditions, and help us collect the required information about the predictors. Also, since this is an offline step, we need to

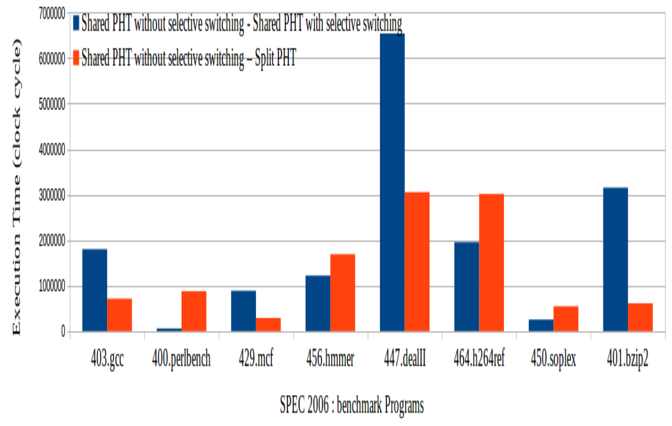

Fig. 8: Execution time comparison

somehow store this information for use at run-time, which can be considered as a significant overhead. In addition to this, we need to track during execution, whether there is a better choice than the current predictor, and whether that choice is better by a sufficiently high amount to cross the threshold. This extra overhead may lead to higher energy cost per prediction. Another obvious limitation of our proposed switching algorithm is the fact that along with the negative interferences, it also reduces the instances of positive interference (since we limit the number of uses of different predictors) and thereby, lose the benefit of it. However, as evident from our results, the gain in being able to reduce negative interference outweighs the benefit that we may have received from positive interference between predictors, since the number of instances of negative interference for a program is much more than their positive counterparts [25].

VII. Conclusion and Future work

In this paper, we examine the effect of predictor table interference on prediction accuracy of a hybrid predictor for resource constrained environments. We propose a predictor selection method to improve prediction accuracy by controlling the interference. Experimental results show the improvement.

Going ahead, we are planning to do away with the static profiling step. As a replacement, we are working on the phase-wise behavior of programs during execution. Our plan is to begin with a simple bimodal predictor and continue with it for prediction for an initial number of cycles (we have taken 20000) to observe the performance of each predictor for each branch and record the overall best predictor for that phase. We proceed to the next phase with these information, and use these predictors (overall best and best for each individual branch) for the entire next phase of the program. In the background, we continue to record the average prediction accuracy of each predictor and the branch specific accuracy, and update the information accordingly for the next phase. We are currently working on setting up this infrastructure. We believe that our design will open up new avenues for future research on architectural designs that can better balance the accuracy and energy tradeoff.

VIII. Acknowledgement

The authors would like to acknowledge Professor Mainak Chaudhuri, IIT Kanpur and Prof. Bei Yu, Chinese University of Hong Kong for their suggestions.

References

[1] M. I. Bielby. *Ultra low power cooperative branch prediction*. The University of Edinburgh, 2015.
[2] J. J. Bonanno et al. Hybrid branch prediction using a global selection counter and a prediction method comparison table, Aug. 30 2005. US Patent 6,938,151.
[3] M. Burrows. Dynamically determining instruction hint fields, Mar. 23 1999. US Patent 5,887,159.
[4] Chang et al. Branch classification: a new mechanism for improving branch predictor performance. In *MICRO*, pages 22–31. ACM, 1994.
[5] P.-Y. Chang et al. Branch classification: a new mechanism for improving branch predictor performance. *International Journal of Parallel Programming*, 24(2):133–158, 1996.
[6] Evers et al. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *ACM SIGARCH Computer Architecture News*, volume 24, pages 3–11. ACM, 1996.
[7] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.
[8] G. G. Henry et al. Hybrid branch predictor with improved selector table update mechanism, Apr. 15 2003. US Patent 6,550,004.
[9] Hicks et al. Towards an energy efficient branch prediction scheme using profiling, adaptive bias measurement and delay region scheduling. In *DTIS*, pages 19–24. IEEE, 2007.
[10] Jiménez et al. Dynamic branch prediction with perceptrons. In *HPCA*, pages 197–206. IEEE, 2001.
[11] D. A. Jiménez et al. The impact of delay on the design of branch predictors. In *MICRO-33*, pages 67–76. IEEE, 2000.
[12] R. E. Kessler et al. The alpha 21264 microprocessor architecture. In *ICCD'98*, pages 90–95. IEEE, 1998.
[13] S. Li et al. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, pages 469–480. IEEE, 2009.
[14] S. McFarling. Combining branch predictors. Technical report, Technical Report TN-36, Digital Western Research Laboratory, 1993.
[15] Mohammadi et al. On-demand dynamic branch prediction. *Computer Architecture Letters*, 14(1):50–53, 2015.
[16] Patil et al. Combining static and dynamic branch prediction to reduce destructive aliasing. In *HPCA-6*, pages 251–262. IEEE, 2000.
[17] J. R. Patterson. Accurate static branch prediction by value range propagation. 30(6):67–78, 1995.
[18] A. Ramirez et al. Branch prediction using profile data. In *Euro-Par 2001 Parallel Processing*, pages 386–394. Springer, 2001.
[19] Sarangi et al. Tejas: A java based versatile micro-architectural simulator. In *PATMOS*, 2015.
[20] A. Seznec. A 64 kbytes isl-tage branch predictor. *JWAC-2: Championship Branch Prediction*, 2011.
[21] A. Seznec. A new case for the tage branch predictor. In *MICRO*, pages 117–127. ACM, 2011.
[22] A. Seznec et al. Design tradeoffs for the alpha ev8 conditional branch predictor. In *ACM SIGARCH Computer Architecture News*, volume 30, pages 295–306. IEEE Computer Society, 2002.
[23] A. Seznec et al. A case for (partially) tagged geometric history length branch prediction. *Journal of ILP*, 8:1–23, 2006.
[24] Shah et al. Method and apparatus for using static branch predictions hints with dynamically translated code traces to improve performance, Mar. 20 2001. US Patent 6,205,545.
[25] E. Sprangle et al. The agree predictor: A mechanism for reducing negative branch history interference. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 284–291. ACM, 1997.
[26] C. Young et al. Improving the accuracy of static branch prediction using branch correlation. In *ACM Sigplan Notices*, volume 29, pages 232–241. ACM, 1994.