Hybridization of TAGE and Linear Branch Prediction

Reid O'Boyle
Department of Computer Science
Texas A&M University
College Station, Texas
631009610
oboylere@tamu.edu

Furkan Sahin
Department of Computer Science
Texas A&M University
College Station, Texas
930005601
furkansahin@tamu.edu

Satya Sreenadh Meesala
Department of Computer Science
Texas A&M University
College Station, Texas
232003265
mssreenadh@tamu.edu

***Abstract*—Branch prediction is an approach in computer architecture that attempts to mitigate the cost of branching. The main purpose of this document is to create a hybrid branch prediction model by taking two existing popular models. This is done to achieve superior performances from the existing efficiency of both the predictors. Pipeline stalling due to conditional branches is one of the most encountered problems in solving the performance potential of deep pipeline superscalar processors. In non-ideal conditions, such as in a time-shared system, code of interest involves context switches. Even for fairly large time-intervals, context switching can degrade the performance of many accurate branch predictors available today. The purpose here is to improve performance statistics and accuracy than the available branch predictors especially during the presence of context switching.**

***Index Terms*—Speculative Execution, Context Switching, Branch Prediction, Hybridization**

I. Introduction

In superscalar processor design, branch prediction accuracy is a major performance factor. There are several branch prediction strategies researched upon to to improve branch prediction. Branch history is used by these complicated branch predictors to obtain higher performance.

Several hybrid branch predictors have been proposed to achieve the desired performance and accuracy. These hybrid predictors combine several branch prediction strategies into a single predictor. A selection mechanism is used by these hybrid branch predictors to determine suitable branch predictor for each branch thereby achieving higher prediction accuracy than the single scheme branch predictor by exploiting the strength of each of their component predictors.

The hybrid branch predictor must apply the most appropriate component branch predictor for each branch in order to achieve higher prediction and accuracy. The most effective dynamic selection is obtained when using two components of branch predictors. For larger number of components, a static selection mechanism, branch classification has been proposed. The effectiveness of this is limited due to less ability to adapt to dynamic changes during program execution. Past research shows that performance of single scheme branch predictions decreases when branch prediction history is periodically destroyed. Context switches may happen due to various reasons, either due to I/O, page faults, end of time quantum, etc during program execution. Context switching destroys the branch history associated with a particular process and therefore affects the performance of branch predictors.

This paper proposes hybrid branch prediction with two branch predictors in inclusion to selection mechanism allowing branch predictor to maintain high prediction accuracy even in the presence of context switching.

This paper is divided into six sections. Section II describes the background information of the branch prediction schemes. Section III describes the hybrid branch prediction and the methodology used and the difficulties encountered. Section IV describes the results and performance comparisons of different branch predictors. Section V represents concluding remarks. Section VI describes Acknowledgements.

II. Background Information

A. TAGE

The TAgged GEometric predictor, TAGE, is one of the best conditional branch predictors available and is used in many hybrid predictors because of its high accuracy. This predictor gains its performative advantage over other competitive predictors by maintaining multiple predictions with varying history lengths. It partially tags predictions in order to avoid false positives, only provides a prediction result on a tag match, and is even rumored to be incorporated into the Intel microarchitecture. TAGE uses the idea of a geometric history length prediction. This involves using multiple prediction tables and hashing using the branch address and global prediction history to map to prediction tables. The history length is then used to get the index of the table. TAGE uses a geometric series to index the prediction table so that history lengths follow a geometric series. This shortens the amount of indexes while still being able to use very long history lengths. TAGE uses a base predictor along with the prediction tables which are based on tag addresses. The use of tagged addresses avoids collisions when hashing long branch histories. The

base predictor is used as a default prediction, and if there are tag matches the prediction tables will be used. If there are multiple tag matches, the one with the match with the longest history will be used. The update policy for TAGE is very important in its accuracy. A useful counter u is used as a counter for correct vs. incorrect predictions. predictions. When the alternate prediction from a tag match table is different from the overall prediction u must be updated. When the overall prediction is wrong u is decremented and incremented when correct.

We are planning on going with a common implementation of TAGE, in which the base predictor will be a program counter indexed 2-bit counter. The 1-bit counter yielding 2 states of "Yes" vs "No" will not suffice. The 2-bits will allow us to represent the necessary 4 states of "Yes", "Yes Maybe", "No Maybe", and "No". Entries of the tagged component utilize a signed counter for prediction. When the algorithm needs to decide on a prediction, the tagged components and the base predictor are both accessed to provide both a prediction on tag match and a default prediction. The final prediction is decided based on the length of the tagged predictor's hit. It will choose the one with the longest history. If the tagged predictor yields no matches, then it will default to the base predictor's choice. According to (Seznec), their experiments found on several applications that the alternate prediction for freshly allocated entries tends to be more efficient. Entries are classified as freshly allocated if their predictor counter is weak, usually monitored through a 4-bit counter. The algorithm can be summed up to

1) Choose the matching component that contains the longest history.
2) If the signed `ALT` counter is negative AND the prediction counter is not in a `WEAK` state, then use the prediction counter's most significant bit (sign) as the prediction. If this is not the case, then choose the alternate prediction.

If the alternate prediction does not match the final prediction, then the referenced useful counter u, belonging to the provider component, is updated. The counter u is also used to maintain age. We will use an aging algorithm to reset the bits as well. Once u is reset, as Seznec recommends, we will flip the two significant bits u_0 and u_1 until the next reset occurs. If the useful counter u is null, then we must also update the alternate prediction counter.

As stated before, we utilize a hash function, global branch history, and a 16-bit path history to index the predictor components. Since traces often contain a mixture of both user level and kernel level instructions, we must utilize some sort of distinct property (probably the address range) to distinguish between kernel and user instructions. Kernel level instructions can be expected to pollute the history with unnecessary entries, and so we choose to use 2 different histories. We will maintain both a kernel history that gets updated indiscriminately of branch type, along with a user history that only gets updates on user type branches. Another wasteful component that we should optimize are tags. Allocating too many bits to tags leads to storage waste while using an insufficient amount of bits leads to an increase in hash function collisions, which consequently lead to invalid false positives by the tag match detector. Seznec suggests that tag length should follow the history lengths, as smaller history lengths allow us to use a lower amount of tag bits, based on experimentation.

We will have to consider issues such as the number of predictor components, predictor update implementation, management, and the response time. To first tackle the response time, we can utilize a method known as "ahead pipelining". We can take inspiration from the OGEHL predictor to implement ahead pipelining. Utilize the n-branch ahead program counter, the n-branch ahead path history, and the n-branch ahead global history in order to read prediction tables. From each table, we will read $2^{(n-1)}$ adjacent entries. Each entry is a possible prediction, and they are computed in parallel. The most distant entries, index as the $n-1$ branches are used to select the final prediction. Although ahead pipelining usually tends to result in less accuracy on medium sized predictors due to an extra layer of indirection through aliasing the base predictor, it still improves overall efficiency. To handle transitioning between user and kernel mode, Seznec hypothesizes that

1) The first two branches after a return in user code should be predicted with ahead information available before the trap or exception.
2) The first two branches after a trap or exception should be predicted through a 1-block ahead for the first branch, and a 2-block ahead for the second branch.

Predictor updates are performed after a commit. This means that update logic is not present in the critical path. If a prediction ends up being correct, then the number of prediction counters that must be updated are bounded by 2, composed of the useful counter or the prediction counter. As opposed to this, an incorrect prediction leads to a new allocation. The tagged component must allocate a new entry in this case. Thus, an incorrect prediction is bounded by at most three accesses. These include reading all the predictor tables at commit time, reading all the tables at prediction time, and writing two predictor tables during update time. We can help mitigate the issue of reading at commit time by utilizing a couple bits available at prediction time. These are the numbers of the alternate and provider components along with the two mentioned components' counter and useful counter values.

Algorithm TAGE Predictor Component

```
# get bimodal index
bimodalIndex = PC % bimodalSize

# initialize tags
for i in TAGE_TABLES:
    tageTag[i] = getTag(PC, i)

# initialize index
offset[4] = {0, 0, 2, 5}
for i in TAGE_TABLES:
    tageIndex[i] = getIndex(PC, i)

# initialize prediction
pred.pred = -1
pred.altPred = -1
pred.table = TAGE_TABLES
pred.altTable = TAGE_TABLES

for i in TAGE_TABLES
    # check for tag hits
    if (tagHit(i))
        pred.alTable = i
        pred.altIndex = tageIndex[i]
        break

p = tagTables[pred.table][pred.index]

# did not miss table
if (pred.table < TAGE_TABLES):
    # alternate missed a table
    if (pred.altTable == TAGE_TABLES):
        pred.altPred = useBimodal()
    # alternate hit table
    else if (p.pred >= TAGE_PRED_MAX/2):
        pred.altPred = TAKEN
    else:
        pred.altPred = NOT_TAKEN
    u = p.useful
    # return best prediction
    if (p != WEAKLY_NOT_TAKEN    ||
        p != WEAKLY_TAKEN        ||
        u != 0                   ||
        altBetterCount < ALTPRED_B_INIT):
            # pred >= max pred / 2
            pred.pred = gtPredMax(p)
            return pred.pred

    # return alternate prediction
    else:
        return pred.altPred

# if both did miss
else:
    # use bimodal table prediction
    pred.altPred = biPred > BI_MAX / 2
    return pred.altPred
```

B. Piecewise Linear Branch Predictor

The piecewise linear branch predictor follows along a similar path of popular neural predictors like perceptron. Neural predictors hold a table of vectors that correspond to addresses. The vectors are then combined to a single value that corresponds to the decision to take or leave a branch. Their approach uses a linear for the correlation of a branch path. By "path" we refer to the history of a sequence of branches leading up to the branch you are predicting. A hyperplane from the piecewise linear function is then used to decide whether to take the branch or not. There are many different paths leading up the branch decision, so the linear function for each path is combined into a piecewise linear surface. The piecewise linear surface separates paths by two categories. The first is the paths that lead to a predicted branch that was also taken and the second is branches that were predicted but not taken.

The two main functions of the piecewise linear branch predictor are making predictions and updating the predictor. The history is tracked using a 3 dimensional array with the dimensions being the branch address, an address of a branch in the path history, and the index of that address in the path history. A single register is also used to signal the outcomes of branches to the predictor as they are executed along with another register containing the address of the branch outcome. These two registers are used to give the path history of the branch we are trying to predict. The prediction function takes in an address of the branch. The initial output of the prediction is set to the weight = `W[address,0,0]` where `W` is the 3 dimensional vector. Then we go down the path of history branches and add/subtract to output based on each historical branch prediction. If the branch was taken we add to output otherwise we subtract from output. We index `W` for the branch address, the history branch at `i` where `i` is the n^{th} branch in history. After going back a set amount of branches in history if the output is > 0 we will take the branch. Otherwise we will not.

Training involves using the branch in history and decrementing or incrementing based on whether the branch at that address was taken. Training is limited by a theta value to stop from over-training a branch if it is already sufficiently known. Because of the increasing size of a 3 dimensional vector we have to limit some of the sizes of the vector. For implementation we will have to choose a constant value h for how long the path of history will be. Additionally we will use set sizes for the branch address and history address to modulo so that those dimensions are a constant size. We also will look into using the idea of ahead pipelining from the paper. This is not completely necessary as we are not as concerned with

latency as we are with storage space. We implemented our Piecewise linear branch predictor based on the pseudocode from the paper. For reference we have added the pseudocode in figure 1. We chose values for m, n, h of 8,288, and 50 respectively. This uses about 120KB of memory for the 3 dimensional array `W` used in the algorithm.

III. Hybridization of TAGE and Linear Branch Prediction

Because linear branch prediction is a slower algorithm, we think it is best to use it when TAGE does not have a long history for the branch available. TAGE's biggest advantage is its use of a long history which in turn increases the prediction accuracy. We believe that linear branch prediction will yield a higher accuracy with a smaller history available. So our hybrid approach will pick TAGE when its history is X times longer than the available history for linear branch prediction (where we will decide X after doing simulations to find the best factor).

Additionally, we will simulate the accuracy of both TAGE and piecewise linear with close to equivalent history lengths. From this we can develop static weights for both TAGE and piecewise linear under similar history lengths. Because piecewise linear yields an output score based on its weights we can combine its score along with the static weight to decide between TAGE and piecewise linear. An example of this in implementation would be something like this: If we learned that piecewise was 40 percent accurate but TAGE was 60 percent accurate on similar history lengths we would give TAGE a static weight of 0.6 and linear 0.4. Then when TAGE predicts no branch but piecewise predicts branch we can look at the output score for piecewise. If the score is 7 we may still pick TAGE but if the score is 250 then we would pick piecewise linear.

We could normalize the output score from piecewise linear to make the decision a simple equation. This gives our hybrid technique more information to make a better decision when the two predictors disagree. If that does not work out we can instead use a choice table where we increment the score of TAGE when we choose it and it is correct, decrement it when it is wrong. And then use the difference between TAGE and piecewise choice scores to make a decision between the two predictors. We can also use two different scores for the two outcomes (1. TAGE predicts, Linear does not. 2. TAGE does not predict, Linear does predict). If we observed that the outcomes yielded different results between branch predictions.

IV. Results

First we will measure our implementation of the Piecewise Linear Predictor on the given traces. The results for each trace and the overall average MPKI can be seen in table 1. We can see that the Piecewise Linear Predictor struggles more on some traces than others. The average MPKI is 7.04 which is higher than we would want, but that is likely the best we can get with the hardware budge for this predictor. Piecewise Linear predictor with 64KB yields average MPKI of 7.429. So doubling the memory for piecewise only gives minimal improvement to 7.04 MPKI. The results for TAGE are shown in table 2. TAGE yields better results than Piecewise with an average MPKI of 5.27. This is expected as TAGE is one of the best predictors available. TAGE did better on most traces but particularly made major improvements on few traces.

Our first attempt at optimization was to utilize the output from the piecewise linear predictor. Our logic was to use the output when TAGE and piecewise differed in predicted directions. If they differed and piecewise had a strong output we would choose piecewise over TAGE. We tried different values of output to decide over TAGE. The output for piecewise linear output ranges from -128 to 128. We tried choosing piecewise for $|\texttt{output}| > 20, 40, 60, 80, 100, 120$ but this only yielded an average MPKI of 5.8 in the best experiment. From this we concluded that the output should only be used as a reason not to use piecewise.

Next we decided it might be a good idea to create a choice table to pick between the two. The choice table would be a table with a number of entries and addresses would map to a 2 bit counter. If the bit is less than two we would choose TAGE. If it was higher than two we would pick piece wise linear. Since TAGE is a stronger predictor in terms of MPKI, it would be the default choice and only if it made two wrong predictions we would choose piecewise linear. The first attempt we decided to increase the 2 bit counter only if TAGE was wrong and decrement it if piecewise linear was wrong. This way TAGE gets two shots to correctly predict it but piecewise linear only gets one chance to get it right. This yielded an average MPKI of 7.104 which is not any better than piecewise on its own. Instead we decided we would structure the counter as a normal two bit counter so piecewise gets two chances instead of one. This yielded an average MPKI of 6.550. From there we enlarged the counter and saw better results. We tried varying the size of the choice table and the amount of bits available to the saturating counter for each choice table entry. We found that performance was about leveled off with a choice table of size 25,000 entries. Then we tried making each entry limited to one byte. We could improve the average MPKI down to 5.261 with the larger choice table. This table has 25000 entries each containing 1 byte. This takes up 25KB of data of storage available.

We tried using a majority wins mechanism to find improvement by adding a simple gshare predictor into our hybrid predictor. We would only use gshare when TAGE and Piecewise Linear disagree. If TAGE disagreed with piecewise linear but gshare agreed, we thought that the majority would likely be correct. So if Piecewise and gshare agreed against TAGE we would take the piecewise prediction instead. This showed little to no improvement. We even tried using the output from piecewise as a part of that decision. We changed the logic

```
function predict (address: integer): boolean
begin
    output := W[address, 0, 0]                    (* Output is initialized to bias weight *)
    for i in 1..h do                              (* Find the sum of weights (or their negations) chosen *)
        if GHR[i] = true then                     (* using the addresses of the last h branches *)
            output := output + W[address, GA[i], i]   (* If the i^th branch in the history was taken, *)
        else                                      (* add the chosen weight *)
            output := output − W[address, GA[i], i]   (* otherwise subtract it *)
        end if
    end for
    predict := output ≥ 0                         (* Predict the branch taken if the output is at least 0 *)
end
```

Figure 2. Prediction algorithm

```
procedure train (address: integer; taken: boolean)
begin
    if |output| < θ or predict ≠ taken then       (* If magnitude of output is less than θ or prediction was *)
        if taken = true then                      (* incorrect then update the weights *)
            W[address, 0, 0] := W[address, 0, 0] + 1   (* If branch was taken, then increment the bias weight, *)
        else
            W[address, 0, 0] := W[address, 0, 0] − 1   (* otherwise decrement it (with saturating arithmetic) *)
        end if
        for i in 1..h                             (* For each address and branch outcome in recent history... *)
            if GHR[i] = taken then                (* If the i^th most recent outcome is equal to current outcome *)
                W[address, GA[i], i] := W[address, GA[i], i] + 1   (* then increment the weight that contributed to this prediction *)
            else
                W[address, GA[i], i] := W[address, GA[i], i] − 1   (* otherwise decrement it (with saturating arithmetic) *)
            end if
        end for
    end if
    GA[2..h] := GA[1..h − 1]                      (* Shift the current address into the global address array *)
    GA[1] := address
    GHR[2..h] := GHR[1..h − 1]                    (* Shift the current outcome into the global history register *)
    GHR[1] := taken
end
```

Figure 3. Training algorithm

Fig. 1. Piecewise Linear Prediction Predict and Update logic

to take piecewise only if piecewise and gshare agreed and if `piecewise output` $> 5, 10, 20, 50, 100$. None of these gave much improvement on what we already had. However, we did see improvement once we used TAGE output instead of piecewise output. This gave us minor improvement down to 5.247 MPKI. We used piecewise when it agreed with gshare and TAGE was a weak prediction. This added about 8KB in memory usage which was not a huge addition for the improved accuracy.

A. Hardware Budget

In the table IV you can see the hardware budget allocation for our hybrid predictor. We had to use aggressive values for our 3 dimensional array to get a good accuracy for the Piecewise linear predictor. That predictor alone used up about 120KB. The large amount of memory used by the piecewise linear predictor constrained the amount of memory we could use for TAGE. TAGE is best at sizes of 256KB and 128KB, however we saw a good accuracy for TAGE with just using 32KB. From this we decided that 32KB is sufficient for our TAGE implementation. For TAGE we only ended up using about 32KB because we used smaller TAGE tables and a 2 bit counter for each one. After allocating memory for both predictors we were left with about 100KB for our hybrid work.

V. Concluding Remarks

In this paper, we have introduced a new hybrid branch predictor that predicts wider range of branches efficiently while being sensitive to branch change information. We were able to show some amount of gain on TAGE by incorporating our hybrid predictor with only 180KB. There is lots of room for improvement for our predictor and we believe we could get MPKI below 5 with more techniques.

Trace Name	MPKI for that trace
gzip	13.747
vpr	11.786
cc	17.570
mcf	17.643
crafty	6.099
parser	13.498
compress	8.185
jess	2.282
raytrace	2.596
db	5.027
javac	2.569
mpegaudio	2.408
mtrt	2.437
jack	2.865
eon	2.177
perlbmk	5.210
gap	5.006
vortex	1.334
bzip2	0.124
twolf	17.620
Total Avg	7.040

TABLE I
PIECEWISE LINEAR PREDICTOR RESULTS

Trace Name	MPKI for that trace
gzip	11.076
vpr	11.850
cc	7.909
mcf	14.658
crafty	5.612
parser	10.026
compress	6.946
jess	3.664
raytrace	1.275
db	2.974
javac	1.879
mpegaudio	1.849
mtrt	1.330
jack	1.215
eon	1.028
perlbmk	0.815
gap	2.589
vortex	1.334
bzip2	0.464
twolf	17.610
Total Avg	5.283

TABLE II
TAGE RESULTS

VI. ACKNOWLEDGMENT

We gratefully acknowledge the support of our Professor Dr. D.A. Jimenez, without which it would not have been possible to undertake this work.

Trace Name	MPKI for that trace
gzip	11.915
vpr	12.00
cc	9.915
mcf	15.022
crafty	5.920
parser	10.023
compress	7.153
jess	0.758
raytrace	1.163
db	2.930
javac	1.801
mpegaudio	1.794
mtrt	1.278
jack	1.207
eon	0.960
perlbmk	0.686
gap	2.968
vortex	0.406
bzip2	0.056
twolf	17.719
Total Avg	5.247

TABLE III
OUR HYBRID BRANCH PREDICTION RESULTS

Section of Predictor	Bytes
4 TAGE Tables	16KB
TAGE BiModal Counters	16KB
Piecewise Global History Register	204B
Piecewise W Array	120KB
Piecewise Global Address Array	204B
Choice Table with 1 byte counters	25KB
Simple GShare Predictor	8KB
Total Bytes	185.4KB

TABLE IV
TOTAL BYTES USED BY THE PREDICTORS AND HYBRID LOGIC

REFERENCES

[1] Jimenez, D.A. “Piecewise Linear Branch Prediction.” 32nd International Symposium on Computer Architecture (ISCA’05), 4 June 2005, https://doi.org/10.1109/isca.2005.40.

[2] Seznec, André. ”A 256 kbits l-tage branch predictor.” Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2) 9 (2007): 1-6.

[3] Seznec, André. Analysis of the o-gehl branch predictor. In Pro- ceedings of the 32nd Annual International Symposium on Computer Architecture, June 200

[4] Daniel A. Jim´enez and Calvin Lin. Neural methods for dynamic branch prediction. ACM Transactions on Computer Systems, 20(4):369–397, November 2002

[5] Daniel A. Jim´enez and Calvin Lin. Neural methods for dynamic branch prediction. ACM Transactions on Computer Systems, 20(4):369–397, November 2002

[6] Boubinjg Branch Prediction https://github.com/boubinjg/BranchPrediction.

[7] Choice Table for Perceptron https://github.com/taraeicher/HybridBranchPredictor.

[8] Po-Yung Chang, Y.N. Patt and M. Evers. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches, 10.1145/232973.232975, 27 December 2005.

[9] P. Chang and U. Banerjee, “Profile-guided Multiheuristic Branch Prediction”, Proceedings of the International Conference on Parallel Processing, July, 1995.

[10] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y.N. Patt, “Branch Classification: a New Mechanism for Improving Branch Predictor Performance”, Z?lh ACM/IEEE International Symposium on Microarchitecture, Nov. 1994.

[11] P,-Y. Chang, E. Hao, and Y.N. Patt, “Alternative Implementations of Hybrid Branch Predictors”, 28th ACM/IEEE International Symposium on Microarchitecture, Nov. 1995.

[12] T.-Y. Yeh and Y.N. Patt, “A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History”, 20th Annual International Symposium on Computer Architecture, May 1993.
[13] T.-Y. Yeh and Y.N. Patt, “Alternative Implementations of Two-level Adaptive Branch Prediction,” 19th Annual International! Symposium on Computer Architecture, May 1992.
[14] T.-Y. Yeh and Y.N. Patt, “Two-level Adaptive Branch Prediction,” 2Jth ACM/IEEE International Symposium on Microarchitecture, Nov. 1
[15] J.E. Smith, “A Study of Branch Prediction Strategies,” 8th International Symposium on Computer Architecture, June 1981.
[16] S. Sechrest, C.-C. Lee, and Trevor Mudge, “The Role of Adaptivity in Two-Level Adaptive Branch Prediction,” 28th ACM/IEEE International Symposium on Microarchitecture, Nov. 1995.
[17] C. Perleberg and A.J. Smith, “Branch Target Buffer Design and Optimization,” IEEE Transactions on Computers, 42(4):396-412, Apr. 1993. 28th ACM/IEEE International Symposium on Microarchitecture, Nov. 1
[18] The PowerPC Architecture: A Specification for a New Family of RISC Processors, Ed. C. May et al, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1994.
[19] R. Nair, “Dynamic Path-Based Branch Correlation”, 28th ACM/IEEE International Symposium on Microarchitecture, Nov.