Constructing Neural Branch Prediction with Memristive Device

Guangyu Shi

Supervisor: Mikko H. Lipasti

University of Wisconsin-Madison
Department of Electrical and Computer Engineering
1415 Engineering Drive
Madison, WI 53706

Submitted in partial fulfillment of the M.S. Degree in Electrical and Computer Engineering Project Option

May 2011

Constructing Neural Branch Prediction with Memristive Device

Abstract

In superscalar computers, average branch instruction latency is crucial to the overall performance of the computer. Among all the branch predictors that have been proposed, neural branch predictors tend to have a better performance than traditional branch predictors, especially on benchmarks with long branch history correlation. However, huge latency of training process and power overhead makes it impossible to integrate neural branch predictors with modern computer chips. On April 30, 2008, HP Labs announced the development of switching memristor. The electrical property of memristor makes it a promising candidate of building fast neural branch predictors. We propose architectural algorithm enhancement as well as circuit level design with memristors to overcome the shortcomings of existing neural branch predictions.

Introduction

In modern superscalar computers, average branch instruction latency is crucial to the overall performance of the computer. Branch mis-prediction could result in a significant performance lost. Therefore, accurate branch predictor is desired for further improvement on instruction-level parallelism. Among all the branch predictors that have been proposed, perceptron-based branch predictors [1] [2] [3] [4] [5] tend to have a better performance than traditional branch predictors, especially on application with long history information. Despite their reported high accuracies, however, there are two disadvantages of perceptron branch predictors. First, perceptron branch predictors cannot predict linearly inseparable branches. Although some perceptron-based branch predictor, such as piecewise linear branch predictor [piecewise] can efficiently predict the behavior of certain linearly inseparable branches, it has to rely on execution path information to distinguish the predictions. Second, to make the prediction, perceptron branch predictor needs to compute the summation of the weights, which results in large computational latency that makes the predictor difficult for hardware implementation.

In this project report, we propose architectural algorithm enhancement as well as circuit level design to overcome the aforementioned limitations of neural branch prediction. On architectural level, we proposed several algorithmic changes, one of which is perceptron branch prediction based on separated Taken/Not-taken weight tables, which resolves some linearly inseparable branches aliasing. Simulation result shows separated weight tables help improve the accuracy of branch prediction. On circuit level, we proposed

perceptron branch prediction design based on memristors, in which the weights are stored as states in memristors.

The following content is structured as two major sections: Architectural Enhancement and Circuit Level Design. In both sections, we will describe the design in details and provide simulation results.

Architectural Enhancement on Neural Branch Prediction

1. Methodology

The simulation framework we use is from JWAC-2: Championship Branch Prediction [6]. The framework models a simple out-of-order core with the following basic parameters:
a. 256-entry reorder buffer, and three schedulers: an integer scheduler with 64 entries and an FP and load/store schedulers with 32-entries each.
b. The processor has a 14-stage, 4-wide pipeline except in the execution stage where it has a 12-wide execution scheduler (6 int, 4 FP, and. 2 load.store).
c. The memory model will consist of a 2-level cache hierarchy, consisting of an L1 split instruction and data caches, and an L2 last level cache. All caches support 64-byte lines. The L1 instruction cache is 32KB 8-way set associative cache. The L1 data cache is 32KB 8-way set-associative. The L2 data cache is a 4 MB, 8-way set-associative cache.

The trace set includes 40 traces, classified into 5 categories: CLIENT, INT (Integer), MM (Multimedia), SERVER and WS (Workstation). The traces include both value and timing information of each micro-op from a detailed out-of-order timing simulator. The timing simulator is configured with perfect branch prediction so that there are no wrong path micro-ops in the traces. As a result, the branch history used in the branch predictor is a perfect branch history. That is, branch history is updated at fetch stage, not at retire stage.

The metric we use to evaluate the predictor is Mis-prediction Penalty Per Kilo Instructions (MPPKI). It is the average number of cycles during which the fetch unit is on the wrong execution path because of a mis-prediction. We use this metric to keep consistent with CBP-3, and to evaluate the impact of branch prediction at cycle level.

2. SWP: Separated Weights Predictor

2.1 Intuition

Table 1 shows four possible correlations between the branch prediction outcome and the history leading to it.

	History	Outcome
1	Taken	Taken
2	Taken	Not-taken
3	Not-taken	Taken
4	Not-taken	Not-taken

Table 1. Four types of correlations

Note that weights of perceptron branch predictors are trained for either positive correlations or negative correlations. That is, one can choose to strengthen correlations 1 and 4, or to strengthen correlations 2 and 3. However, for some applications, it is desired to have strong correlation 1 but weak correlation 4, or strong correlation 2 but weak correlation 3, and vice versa. A pseudo code example is given in figure 1.

```
// x is an unknown value
If (x>=1000) // Branch A
{ /* do some task */ }
If (x>= 500) // Branch B
{ /* do some other task */}
```

Figure 1. A code example

In figure 1, Branch A serves as a history of younger branch B. It is not difficult to observe that, if branch A is taken, then branch B will also be taken. However, if A is not taken, then we do not know if B will be taken or not. This is a simple example of branch patterns that has a strong correlation 1, but weak correlation 4 in table 1. In perceptron branch predictors, if the branch A is taken at certain frequency, it is likely that the weight associated with this history will be trained toward a positive value. When branch A is not taken, branch B will be indicated as Not-Taken, although in fact there is no strong correlation between them.

Although piecewise linear branch predictor can predict certain types of linearly inseparable branches, the prediction relies on the different execution path information. If the different outcome of a branch has the

same execution path (similar as the example in figure 1), piecewise linear predictor cannot distinguish the predictions. Therefore, it is necessary to have different weight tables for taken and not-taken histories.

2.2 Prediction and Update Algorithms

Figure 2 presents the pseudo-code of the prediction and update algorithm of SWP. In this code, W0 is the bias weight table. WT and WNT are the Taken and Not-Taken weight tables, respectively. GHR is the global history register. HA is the path history address register that stores the addresses of the past executed branches. Integer *ghl* is and the length of the global history. Integer *address* is the address of the branch to be predicted.

```
function predict: boolean
begin
  sum := W0[address];
  for i in 1 to ghl do
    index := hash (address, HA[i]);
    if GHR[i] = true then
      sum := sum + WT[index, i];
    else
      sum := sum + WNT[index, i];
  end for
  predict := (sum>=0);
end
```

```
function update
begin
  if |sum|<threshold or predict != br_taken
    for i in 1..ghl do
      index := hash (address, HA[i]);
      if GHR[i] = true && br_taken = true
        WT[index,i] := WT[index,i] +1;
      else if GHR[i]=true && br_taken=false
        WT[index,i] := WT[index,i] -1;
      else if GHR[i] = false && br_taken=true
        WT[index,i] := WNT[index,i] +1;
      else if GHR[i]=false && br_taken=false
        WT[index,i] := WNT[index,i] -1;
      end if
    end for
  end if
end
```

Figure 2. Pseudo-code of prediction and update algorithms

Figure 3 further illustrates the weight selection of SWP. Now weights in traditional perceptron predictors become weight pairs: Taken and Not-Taken weights. To calculate the summation, only one of the weights in each pair will be chosen. In figure 3, solid squares are weights that are selected to compute the sum. Only addition operation will be performed during prediction, although both increment and decrement operations are performed during updates. These algorithms allow the predictor to treat the correlations separately.

Figure 4 shows the simulation result on 11 traces out of the 40 traces from CBP-3. We compare SWP with piecewise linear branch predictor [3]. We can observe that SWP outperform piecewise linear branch prediction, especially when weights are small.

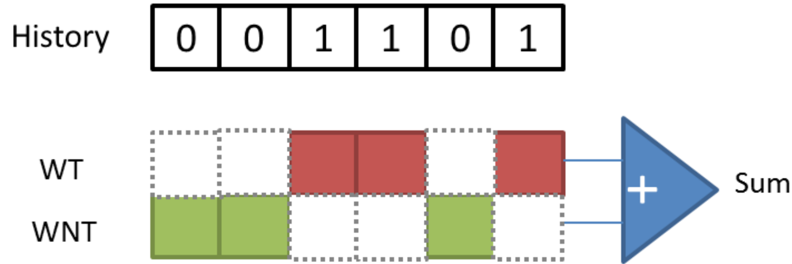

Figure 3. An example of weight selection in SWP

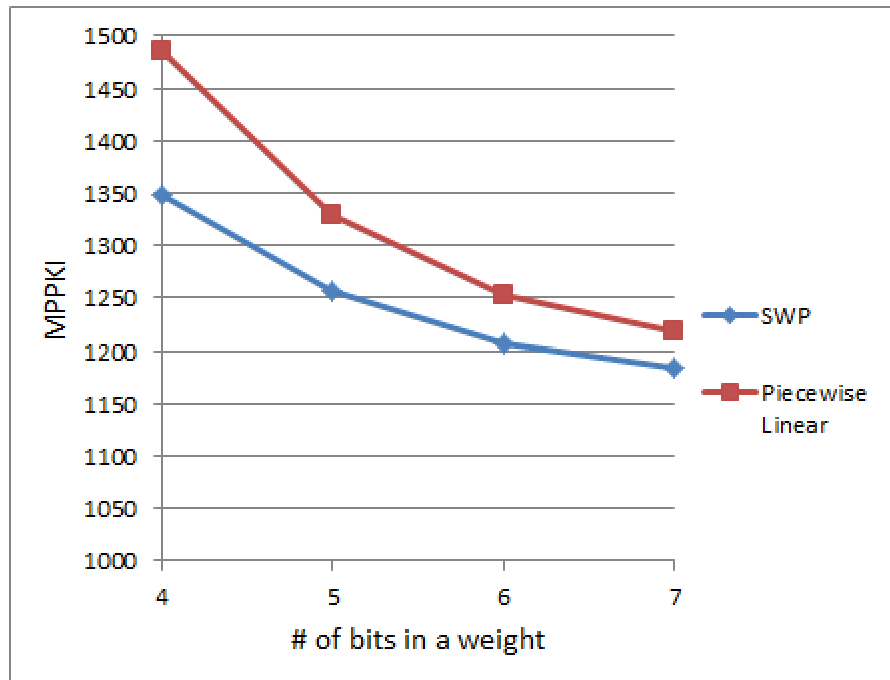

Figure 4. SWP vs. Piecewise Linear Branch Prediction

2.3 SWP can predict some linearly inseparable branches

We claim that SWP can predict some linearly inseparable branches [1] [15]. Perceptron branch predictor cannot predict linearly inseparable branches effectively, because the decision surface is always a hyperplane (e.g. a line if n = 2) of the n-dimensional space (n is the history length). Several research works has been proposed to overcome the incapability of perceptron branch predictor on linearly inseparable branches [3] [5] [7] [16]. Here we use an example to explain that the decision surface of SWP is not a hyperplane.

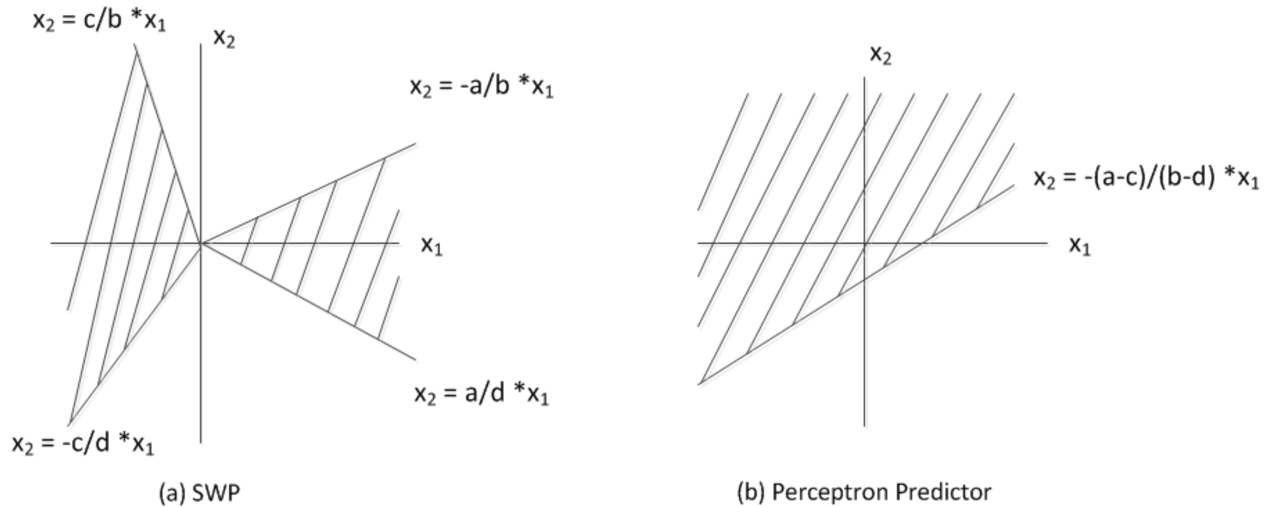

Figure 5. Decision surface of SWP and perceptron branch predictor

Let's assume the history length is 2. The weights in "taken" table are a and b, and the weights in "not-taken" table are c and d. Therefore, the decision surface of SWP is shown in figure 5 (a). In this figure, either shaded area or non-shaded area is the "taken" zone, depending on the sign of a, b, c and d. If we use the same branches to train a perceptron branch predictor, we will have the weight values of (a-c) and (b-d). The decision surface of this perceptron branch predictor is shown in figure 5 (b), which is a hyperplane of the space. Therefore, SWP can form up piecewise linear decision surface without relying on the path information.

However, not all linearly inseparable behaviors can be learned by SWP. For a 2 history space, for example, XOR function cannot be learned by SWP. Learning such behavior will require multiple levels of neural network.

2.4 Implementation of SWP

Other than the separated weight tables, the implementation of SWP has no different with previous proposed perceptron predictors. A number of multiplexors are used in the weight table to select the desired weight from Taken and Not-Taken weight tables. This weight selection is performed in parallel and thus will not increase latency of the prediction.

Another advantage of SWP is that no 2's complement of weight needs to be calculated. In perceptron branch predictor, if the history is not-taken, the negation of corresponding weight needs to be calculated. In practical implementation, this is done by flipping all the bits in that weight. In SWP, since there is not

subtract operation needed, we do not need to calculate the negation of a weight, which further improves both latency and accuracy.

2.5 Optimize for Space

It is not difficult to realize that for the same length of history and the same number of entries in the weight table, SWP requires twice as much storage space as a regular perceptron predictor. [4] observes that recent branches have stronger correlations with the branch to be predicted than old branches. Therefore, we use partially separated weight table in our predictor.

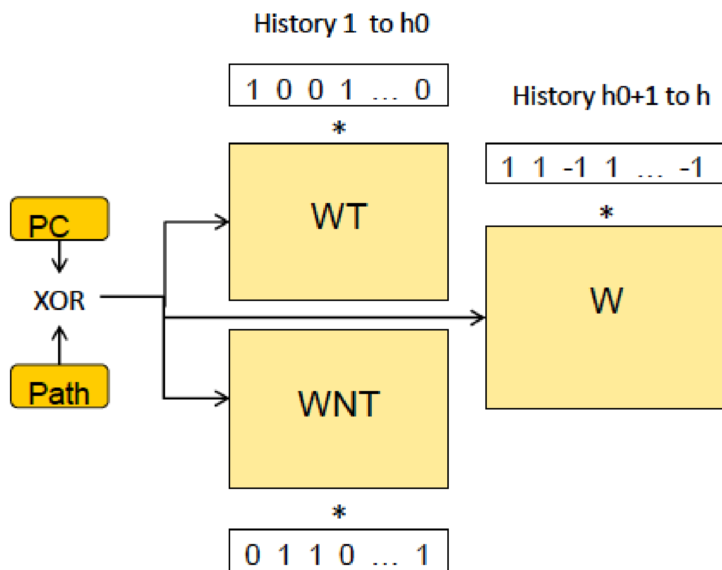

Figure 6. Partially separated weight table: structure and MPPKI

Figure 6 is the structure of SWP with partially separated weight tables. Figure 7 is a set of simulation results with partially separated weight table. The horizontal axis shows the history length associated with separated weight tables (h0 in figure 6). The total length of the history is 64. We observe that when the most recent 20 branches are associated with separated weight tables, the MPPKI is reduced by 2.0%. When we further increase h0 to 64 (fully separated weight tables), the MPPKI is further reduced by only 0.9%. Therefore, it is most efficient to use separated weight tables on only a few most recent branches, and use a single weight table to explore long history information.

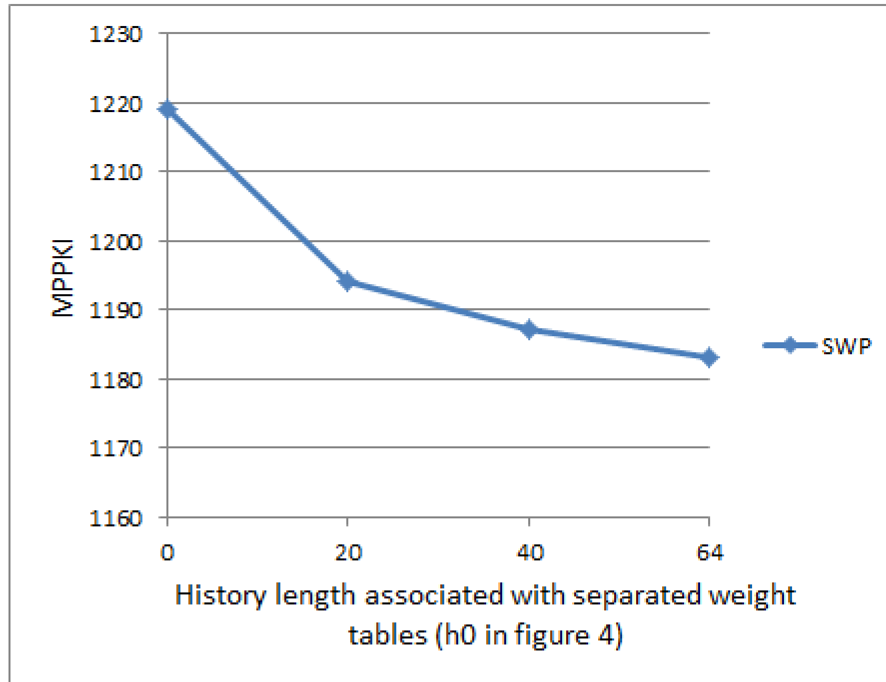

Figure 7. History length with separated weight tables vs. MPPKI

2.6 Combined with other optimization schemes

Most of the optimization schemes for perceptron predictors can be applied on top of SWP. In particular, we use the proposed method in piecewise linear branch prediction design [3] that uses path information to reduce aliasing in weight table. We also used dynamic threshold [7] to set threshold adaptively. Result shows that these optimization schemes further improve the branch prediction accuracy.

2.7 Conclusion

We proposed perceptron branch prediction with separated weight tables. In our predictor, Taken and Not-Taken correlation of the same history is treated separately using two different weight tables. Experiment result shows that our proposed branch predictor realizes high prediction accuracy. Its advantage on predicting some linearly inseparable branches allows it to use less number of bits in the weights and a shorter history length, which can reduce the latency of prediction. Furthermore, using separated weight table on most recent branches and using single perceptron on the rest branches is proved to be a good strategy to improve prediction accuracy with minimum storage overhead.

3. AFPBP: Adaptive Four-state Perceptron Branch Prediction

In this section, we will analyze another type of perceptron branch prediction that encodes 4 different correlations between past branches and current branch into 4 states, and calculate the prediction depending on the type of correlation for each weight.

3.1 Intuition

In section 2 we have already explained the advantage of using separate weight tables. To overcome the 2x storage cost problem, we use partially separated weight tables. However, when we analyze the weight values by running profiling simulation, we notice that most of the entries still have weights with different sign in two weight tables. In this case, perceptron mode is enough, even beneficial to contribute to calculate the sum. Therefore, we do not necessarily need separated weights for all the entries, not to mention its storage overhead.

If we analyze the correlations in table 1, we should realize that there are four fundamental correlations between past branches and current branch that we really need to distinguish. They are:

a. *No correlation or very weak correlation*. That is, WT and WNT has the same polarity and similar magnitude.
b. *Positive (1&4) or negative (2&3) correlation*. That is, WT and WNT has different polarities and similar magnitude.
c. *Strongly correlated with taken history, but weakly correlated with not-taken history*. That is, WT has a large magnitude, while WNT has a very small magnitude.
d. *Strongly correlated with not-taken history, but weakly correlated with taken history*. That is, WT has a large magnitude, while WNT has a very small magnitude.

SWP is able to detect all four correlations. Perceptron predictor can detect a and b. Ideally, the predictor will magically know what type of correlation each past branch has with the current branch. After that, we only need to store one weight, and treat the other as either negation of the weight (b) or zero (c and d). Based on the above observation, we designed an adaptive encoding algorithm that uses two extra bits to represent the state. Compare to SWP which doubles the storage budget, this encoding scheme significantly reduces storage overhead of SWP.

3.2 Encoding Algorithm of AFPBP

The main idea of AFPBP is as follows: we use 2m+2 bits to represent each weight. 2 bits are used to encode the state. Initially, weights are in SWP state. We use m bits as WT and the rest m bits as WNT. After a few training cycles, the predictor will detect which of the four correlations this particular branch has with the branch to be predicted. Then, the predictor change the state and uses the entire 2m bits to represent the weight, since any state other than SWP, only one weight value is meaningful, while the other is either its negation or very small in magnitude.

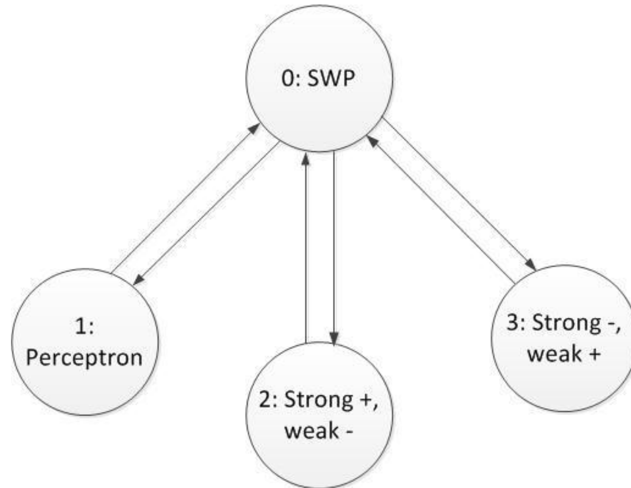

Figure 8. State diagram of AFPBP

Figure 8 is the state diagram of AFPBP algorithm. In this case, m=4, therefore the range of WT and WNT is [-8, 7]. Weights can switch from state 0 to state 1, 2 and 3, and backwards, but not between stage 1, 2 and 3. In state 0, state switching happens when one of the weights is about to exceed the saturation value (-8 or 7). Similarly, in state 1, 2 and 3, state switching happens when weight falls back into the range of [-8, 7]. Table 2 summarizes four different state codes and their transition condition and other operations for each state.

As a result of using AFPBP, the storage budget increases by 25%, instead of doubled. Figure 9 is a comparison between perceptron branch predictor, SWP and AFPBP under the same history length. It appears that AFPBP improves the performance significantly at short history lengths. When the history length increases, the improvement diminishes on AFPBP.

State code	State explanation	Weight range	Operation performed for prediction	Switch condition from state 0 when one weight exceeds saturation value	Update operation when switching back to state 0
0	Separated Weight Tables	[-8, 7]	+WT if h=1; +WNT if h=-1	WT, WNT > thw or WT, WNT < -thw-1	WT = 0; WNT = 0;
1	Perceptron	[-128, 127]	h*W	WT >7 && WNT<-*thw*-1 or WT<-8 && WNT>*thw* or WNT>7 && WT<-*thw*-1 or WNT<-8 && WT>*thw*	WT = W; WNT = -W
2	Correlation with taken only	[-128, 127]	+W if h=1	WT>7 && WNT ϵ [-*thw*-1, *thw*] or WT<-8 && WNT ϵ [-*thw*-1, *thw*]	WT = W
3	Correlation with not-taken only	[-128, 127]	+W if h=-1	WNT>7 && WT ϵ [-*thw*-1, *thw*] or WNT<-8 && WT ϵ [-*thw*-1, *thw*]	WNT = W

Table 2. State codes and descriptions of AFPBP

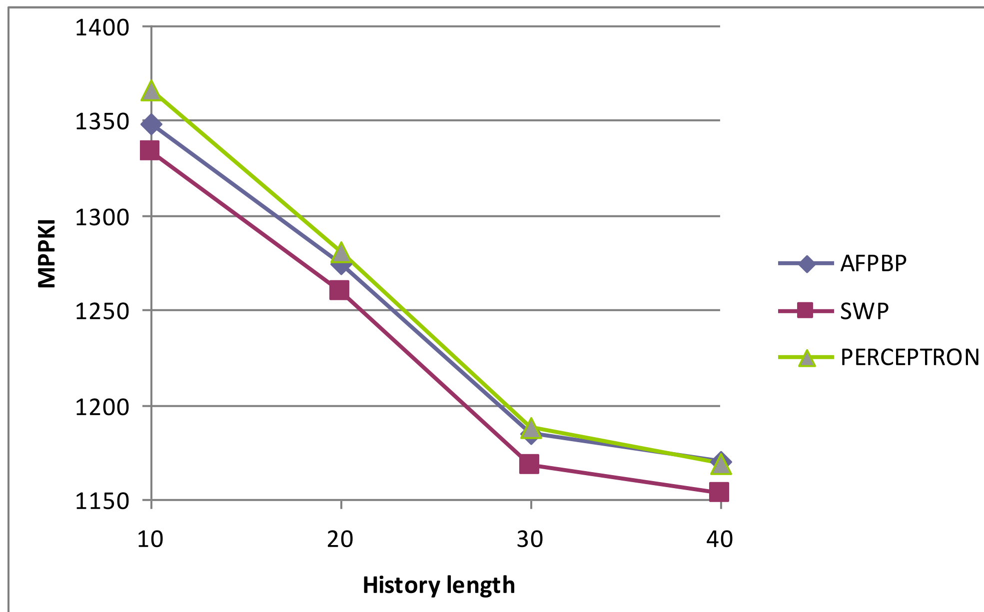

Figure 9. AFPBP vs SWP vs Perceptron on same history length

Circuit Level Design of Neural Branch Prediction

Section 2 and 3 Introduces the architectural enhancement on neural branch prediction. In practical, huge prediction latency and power overhead prevent perceptron branch predictor from real hardware implementation.

Several ways has been proposed to overcome latency and power challenges. [2] uses ahead pipelining, which calculates the sum of weights ahead of time. This method can mitigate the latency problem, but the accuracy of prediction is degraded since the weights may not be up-to-date. [4] uses analog current summing circuit to calculate the sum of all the weights, which does not require ahead pipelining. However, the digital-to-analog converter requires transistors sizing from 1x to 128x (for 7-bit weight), which is a big challenge for chip manufacturer. How to overcome latency and power overhead of perceptron branch prediction remains a major question for researchers.

Memristor, first defined in [8], is considered the fourth fundamental electrical element (the other three are resistor, capacitor and inductor). Memristors are two-terminal passive devices just like resistors, except that its resistance are related to the total amount of charge flowing through it. More generally, a memristor is a two-terminal passive circuit element in which there is a relationship between charge and magnetic flux linkage [8].

In 2008, a team at HP Labs announced the development of a switching memristor based on a thin film of titanium dioxide [9]. It has a regime of operation with an approximately linear charge-resistance relationship as long as the time-integral of the current stays within certain bounds. These devices are being developed for application in nanoelectronic memories, computer logic, and neuromorphic computer architectures. Memristors based on different materials has been proposed in various research works, including STT-RAM, PCM, some of which are considered as replacement of current DRAM technology [10].

Besides high-density memories, memristors are also used to build neuromorphic circuits and architectures. We propose memristor-based perceptron branch prediction, which uses a pair of memristors to store a weight. Properties of the memristor are set based on a predictive model [11]. Simulation in LTSPICE shows a promising result in terms of latency.

1. Methodology and SPICE model of memristor

We use LTSPICE as the design and simulation tool. For any CMOS logic, we use 22nm high performance model from Predictive Technology Model (PTM) [12].

The SPICE model of memristor we use is derived from [13]. Basically, it is modeling the behavior of the memristor + rectifier device in [11]. The I-V relation is depicted by the following equation:

$$I = w\^{}n*c1*sinh(d1*V)+c2*(exp(d2*V)-1) \quad (1)$$

which is a fitting of the observed I-V curve in [11]. Figure 10 presents the real experimental I-V curve and the I-V curve obtained from the SPICE simulation.

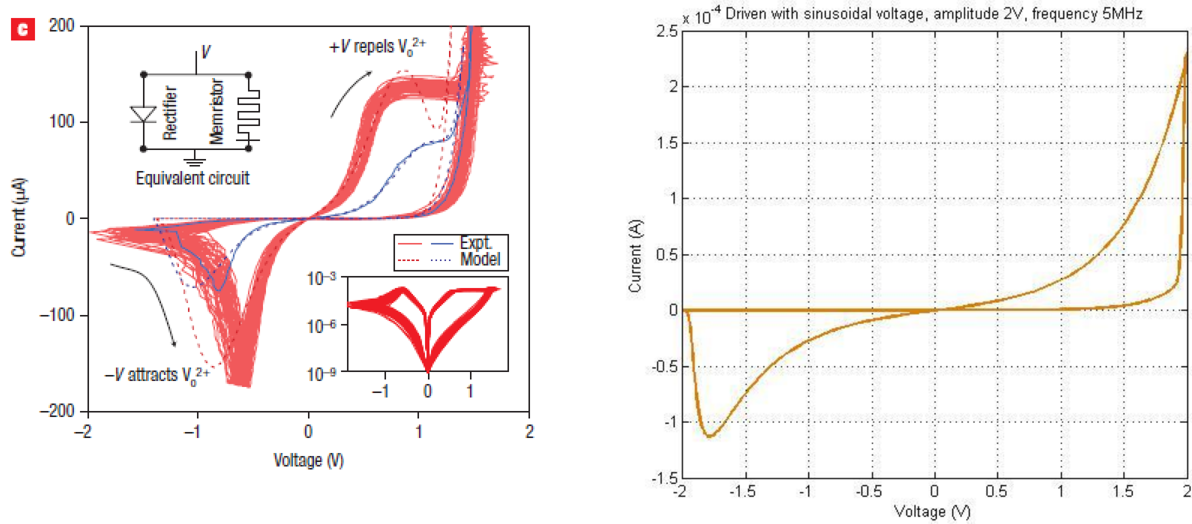

Figure 10. Observed I-V characteristic in real device (left) vs. I-V characteristic of SPICE model (right)

2. Memristor based perceptron branch prediction

We adopt the current summation logic and the voltage comparator and amplifier in the final prediction stage in [4]. We replace the SRAM and D/A converter circuit – since it is too difficult to fabricate – with complementary memristors to store the weights.

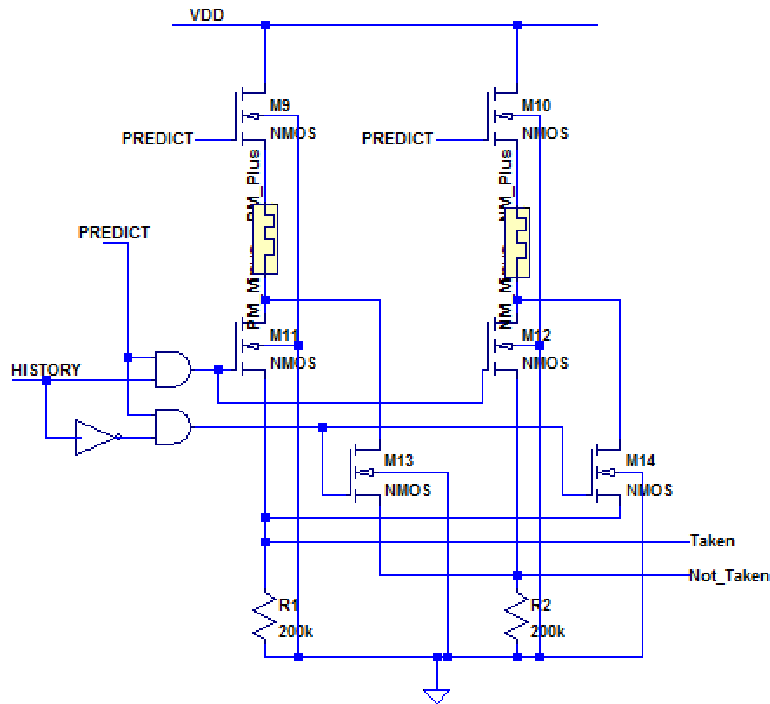

Figure 11. Memristor-based perceptron branch prediction: prediction circuitry

Figure 11 is the circuit-level diagram of a 1-bit history perceptron branch prediction circuitry. In this design, we use memristor as multi-level cell (MLC), that is, a storage element with more than 1 bit information. Theoretically, resistive memories can be configured with arbitrary levels, although more levels result in less noise margin for the cell. We will discuss the design of single-level cell memristor in later sections. A primary memristor (left) is used to store the weight value, and a second memristor (right) is used in order to compute the complement the weight value.

During prediction phase, VDD is applied to the positive terminal of both memristors and GND is applied to the negative terminal of both memristors. The current is steeled to positive line or negative line depending on the history bit. Finally, current from both lines will be converted to voltage signals by passing through resistors of the same resistance. The amplitude of voltage from "Taken" line and "Not-taken" line are compared by differential amplifier. Therefore, prediction is made by the output of the differential amplifier.

For example, if the weight is positively strong, the primary memristor is turned on and the complementary memristor is turned off. Therefore, the current that passes the primary memristor is bigger

(we denote it as *I*) than the current that passes the complementary memristor (we denoted it as *i*). Consequently the Taken voltage *IR* is greater than Not-taken voltage *iR*, and thus the prediction is given as "taken".

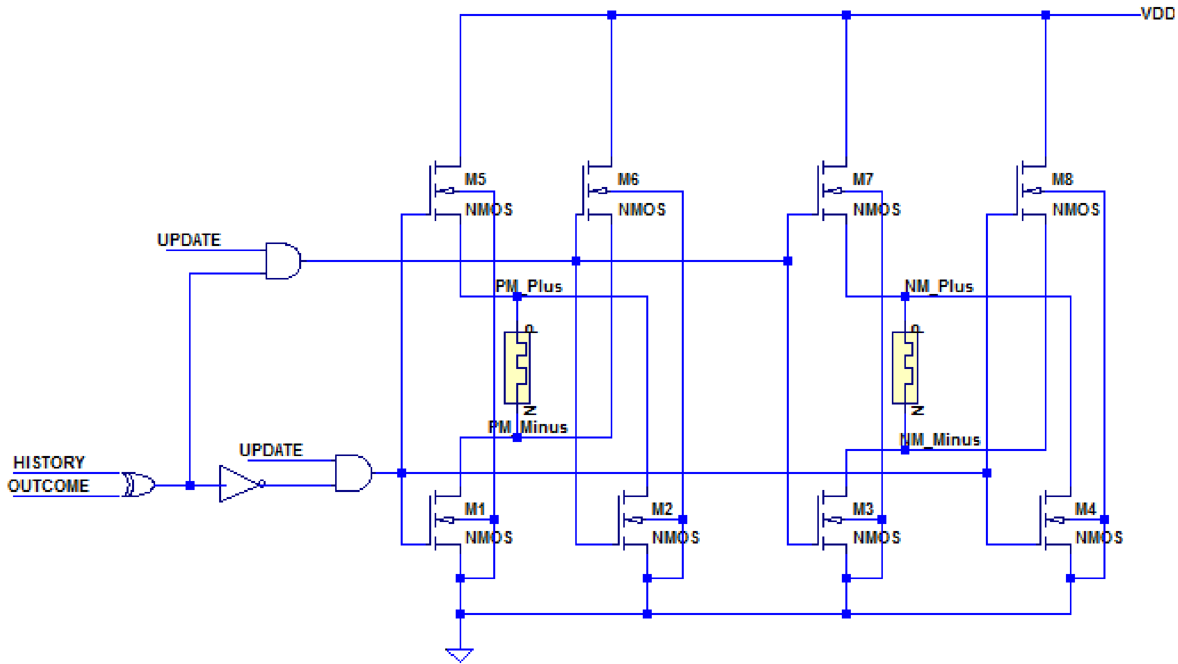

Figure 12. Memristor-based perceptron branch prediction: update circuitry

Figure 12 is the circuit-level diagram of the update circuitry. Perceptron branch predictor strengthens a weight if positive correlation is observed, and weakens a weight if negative correlation is observed. Therefore, directions of the current that passes memristors are controlled by the XOR of history bit and outcome. If history is the same with the outcome (positive correlation), primary memristor is strengthened (resistance decreased) and complementary memristor is weakened (resistance increased). If history is different from the outcome, memristors are trained oppositely.

Although HP lab has announced that they are aiming at 3nm memristors that can switch at GHz level, currently there are no such fast-switching memristors fabricated. At this stage, we assume the physics characteristic (equation 1) of memristor is scalable. We choose the parameters in equation as follows:

a=29; b=30; $c1$=9; $c2$=0.01; $d1$=2; $d2$=4; s=10u; n=4; p=1; $lmin$=0.05; $lmax$=0.95.

3. Simulation Results

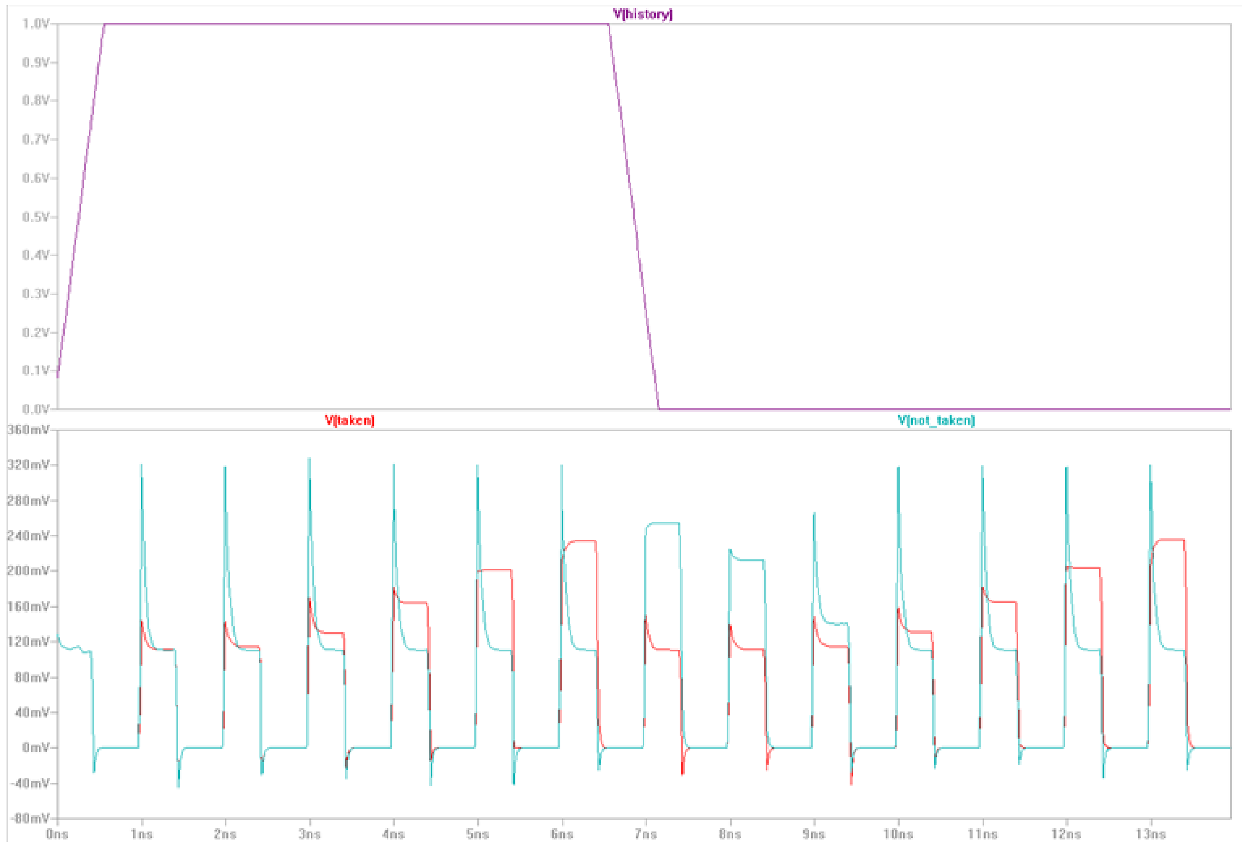

Figure 13. Simulation result of 1-bit predictor: change of history

Figure 13 is the SPICE simulation result of 1-bit memristor-based perceptron branch predictor. In this experiment, outcome remains to be "taken" and history changes from 1 to 0 at around 7 ns. Note that in this experiment, memristor is used as a MLC. During the first 7 ns, prediction changes from neutral to strongly positive value, because repetitive "taken" outcome keeps strengthening the weight, and thus increases the confidence of the prediction. After history bit changes from 1 to 0, there is a sudden mis-prediction since we did not change the outcome, and the correlation between history and outcome changes from positive to negative. After the misprediction being detected, the weight is decremented, and eventually the prediction becomes strong "taken" again at 14 ns. Note that ideally, we would expect the weight to be neutral again after equal number of training cycles for positive and negative correlations. However, since memristor is not a symmetric device, positive terminal is always on the side of doped area, the latency of turning on the memristor is bigger than turning off it. As a result, training from neutral state toward strong positive or negative correlations is much faster than training the other way around.

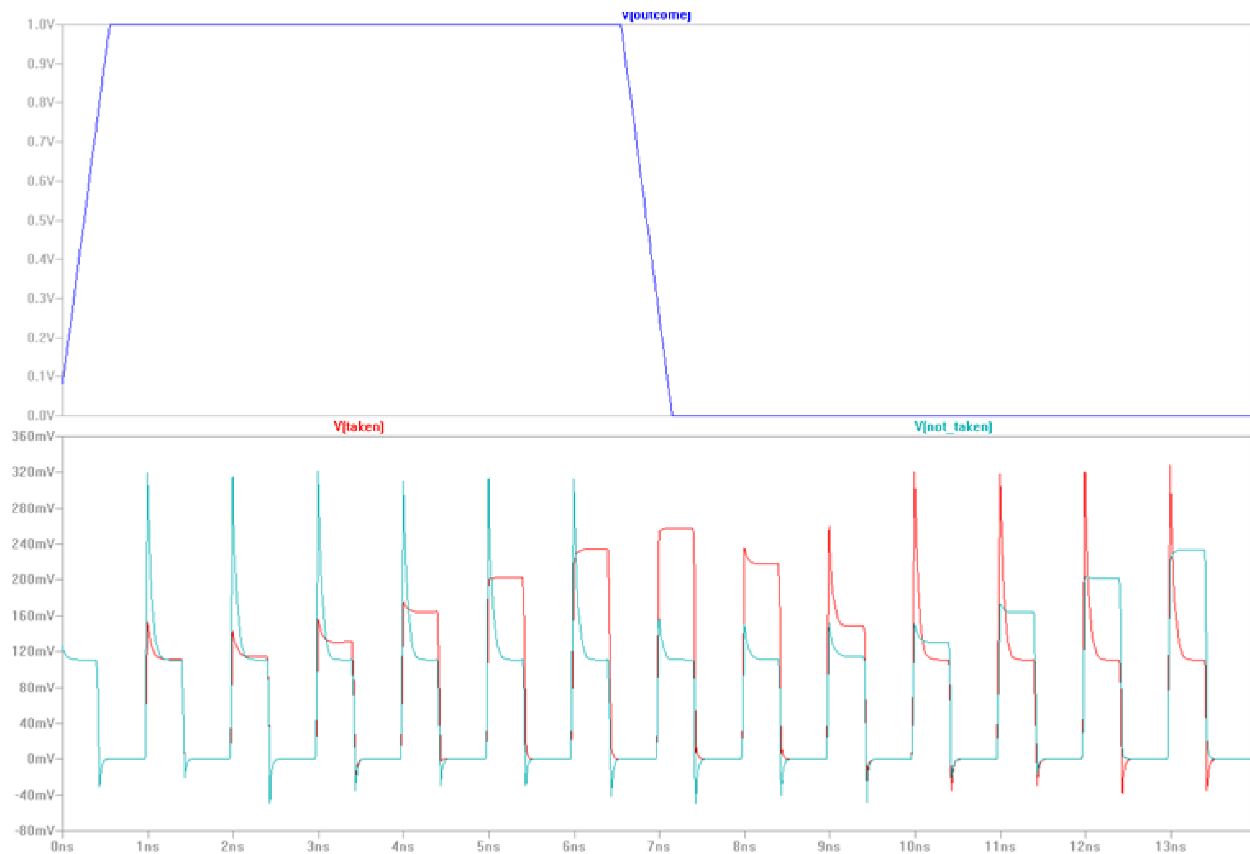

Figure 14. Simulation result of 1-bit predictor: change of outcome

Figure 14 is the SPICE simulation result of 1-bit memristor-based perceptron branch predictor. This time, the history bit remains as "taken", but the outcome changes from "taken" (1) to "not-taken" (0). From the simulation result, we can observe that the weight is trained from neutral toward strongly positive value. At 7 ns, prediction remains unchanged, but it becomes a mis-prediction. During the next few cycles, weight is decremented because of the detected negative correlation, and eventually changes the prediction from strong "taken" to strong "not-taken".

From both simulations, we can observe the learning capability of memristors. Moreover, the voltage difference can be used to evaluate the confidence of the prediction, and can be used in hybrid predictors.

Remember that the proposed design and simulation are based on memristor-based MLC. In practice, memristor-based MLC has to deal with many non-ideal effects. First, the more levels it has, the less noise it can tolerate. Second, resistance of memristors will drift from original value, Compared to the large

noise margin of CMOS gates, equal-robust design of memristor-based circuitry allows the designer to use only a limited number of levels in a memristor (<10). However, previous perceptron predictor uses at least 6 bits in a weight, that is, 64 different states in a weight. Small weight will sacrifice the prediction accuracy significantly. Therefore, implementation of large weights with multiple memristors for each weight is required for future study of the perceptron branch predictor.

3. Conclusion and Future Work

In this part we proposed memristor-based perceptron branch prediction design. Memristor is a promising potential candidate for implementing weights in perceptron branch prediction, although its physics characteristic is still not determined or understood by researchers. We use SPICE simulation to analyze the timing information of memristor-based perceptron branch prediction. Simulation result shows a promising result in terms of accuracy and latency on a 1-bit predictor.

We believe more opportunity can be exploited by our memristor-based perceptron branch predictor. One opportunity is to study thermometer code implementation [14] and use it to implement large weights. Thermometer code is designed for neuromorphic circuit implementation with memristors, and it fits the design of calculating prediction by current summation very well. Another opportunity is to

Reference

[1] Jimenez, D.A.; Lin, C., "Dynamic branch prediction with perceptrons," *The Seventh International Symposium on High-Performance Computer Architecture, 2001. HPCA.*, pp.197-206, 2001

[2] Jimenez, D.A., "Fast path-based neural branch prediction," *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003*. MICRO-36, vol., no., pp. 243- 252, 3-5 Dec. 2003

[3] Jimenez, D.A., "Piecewise linear branch prediction,". *Proceedings. 32nd International Symposium on Computer Architecture, 2005*, *ISCA '05* pp. 382- 393, 4-8 June 2005

[4] St. Amant, R.; Jimenez, D.A.; Burger, D.; , "Low-power, high-performance analog neural branch prediction," *41st IEEE/ACM International Symposium on Microarchitecture, 2008. MICRO-41*, vol., no., pp.447-458, 8-12 Nov. 2008

[5] Y. Ninomiya and K. Abe, "Path traced perceptron branch predictor using local history for weight selection", Second JILP Championship Branch Prediction Competition (CBP-2), pp. 7 - 12, 2007

[6] 2nd JILP Workshop on Computer Architecture Competitions (JWAC-2): Championship Branch Prediction. http://www.jilp.org/jwac-2/

[7] Seznec, A., "Analysis of the O-GEometric history length branch predictor," *Proceedings. 32nd International Symposium on Computer Architecture, 2005. ISCA '05*, vol., no., pp. 394- 405, 4-8 June 2005

[8] Chua, L.; , "Memristor-The missing circuit element," *IEEE Transactions on Circuit Theory*, vol.18, no.5, pp. 507- 519, Sep 1971

[9] Strukov, D. B.; Snider, G. S.; Stewart, D. R.; Williams, S. R., "The missing memristor found", *Nature 453 (7191)*, pp. 80–83, May 2008

[10] Lee, B.C.; Ping Zhou; Jun Yang; Youtao Zhang; Bo Zhao; Ipek, E.; Mutlu, O.; Burger, D.; , "Phase-Change Technology and the Future of Main Memory," *Micro, IEEE* , vol.30, no.1, pp.143, Jan.-Feb. 2010

[11] Yang, J.J.; Pickett1, M.D.; Li1, X.; Ohlberg1, D. A.; Duncan R.; Stewart1, D. R.; Williams R.S., "Memristive Switching mechanism for metal/oxide/metal nanodevices" Nature Nanotechnology 3, pp. 429-433, June 2008

[12] Predictive Technology Model (PTM). Available: http://ptm.asu.edu/

[13] "Modeling the HP memristor with SPICE". Available: http://www.memristor.org/news/222/hp-touts-3nm-memristor-fabrication-milestones

[14] Snider, G, "Instar and outstar learning with memristive nanodevices", *Nanotechnology 22 015201*, December 2010

[15] Faucett, L., *Fundamentals of Neural Networks: Architectures, Algorithms and Applications.* Prentice-Hall, Englewood Cliffs, NJ, 1994

[16] Seznec, A., "Redundant history skewed perceptron predictors: Pushing limits on global history branch predictors," *IRISA, Technical Report*. 1554, September 2003.