More Accurate Value Prediction using Neural methods
Pradipta Kumar Mishra and Snigdha M. Mohapatra
Centurion Institute of Technology, Bhubaneswar, Orissa, India.

ABSTRACT
Data dependencies between instructions have traditionally limited the ability of processors to execute instructions in parallel. Data value predictors are used to overcome these dependencies by guessing the outcomes of instructions in a program. Because mispredictions can result in a significant performance decrease, most data value predictors include a confidence estimator that indicates whether a prediction should be used or not. Much research has been done recently in the area of data value prediction as a means of overcoming these dependencies [7,8,9,10,11,17,18,20,21]. The goal of data value prediction is to guess the outcome of an instruction before the instruction is actually executed, allowing future instructions that depend on its outcome to be executed sooner. Data value predictors are usually designed to look for patterns among the data produced in repeated iterations of static instructions. Accurate prediction can be attained when the repeated outcomes of a particular instruction follow easily discernable patterns.

Many researchers have used Support Vector Machines (SVM) for data value prediction. SVMs are used to identify which past instructions affect the accuracy of a prediction and to decide based on their results whether the prediction is likely to be correct or not. A typical confidence estimator approach tries to determine the accuracy of a prediction (an instruction’s predictability) by looking at whether the last several predictions for that instruction were correct. If they were all correct, intuitively the next prediction should also be correct. But if the instruction was recently predicted incorrectly, then the new prediction is not trusted. This localized approach does not consider the effect that other surrounding instructions may have on the instruction being predicted. Correlations often exist between the predictability of different instructions, especially if one instruction is a source of data for another [17]. Hence, one instruction’s prediction outcome may be correct only if a certain prior instruction’s prediction outcome was correct.

However, in order to make use of other instructions’ prediction accuracies, one must determine which surrounding instructions affect the current instruction. We base our estimator on the SVM, a simple form of neural network. Earlier, perceptrons were used for prediction. But here, a SVM is assigned to each instruction whose outcome needs to be predicted. Each SVM identifies which past instructions tend to affect the instruction’s prediction confidence. It then uses the prediction accuracies of those past instructions to determine a confidence value for the current prediction.

Problem Statement:
This paper presents a global approach to confidence estimation in which the prediction accuracy of previous instructions is used to estimate the confidence of the current prediction. Support Vector Machines are used to identify which past instructions affect the accuracy of a prediction and to decide based on their results whether the prediction is likely to be correct or not.

Early work on Value prediction:
Lipasti, Wilkerson, and Shen introduced the earliest confidence estimator used in data value prediction in [4]. It is
comprised of a 2-bit saturating up-down counter that chooses between three prediction states: 0 or 1 = “don’t predict”, 2 = “predict” and 3 = “constant” (highly predictable). If a given instruction makes a correct prediction, the counter is incremented; otherwise, it is decremented. Regardless of whether the instruction predicts correctly or mispredicts, the counter is not allowed to exceed 3 or go under 0. This approach is used in many other proposed data value predictors [8, 12, and 13].

Perceptrons:

The use of the perceptron as a predictor was first suggested by Vintan et al [14]. The perceptron is one of the simplest models of a neuron and was developed by Rosenblatt [1] to help study brain function. The simplest perceptron is a neuron that connects several weighted inputs to a single output. Classically, the output $y$ of the perceptron is the dot product of the weights $w = (w_1, \ldots, w_n)$ and the inputs $x = (x_1, \ldots, x_n)$, with the bias input $b$, which can be thought of as a weight $w_0$ with constant input $x_0 = 1$

$$y = b + \langle x, w \rangle = w_0 + \sum_{i=1}^{n} x_i w_i$$

The output $y$ is used to classify a new pattern $x$. The perceptron’s performance in classifying is improved by incrementally adjusting its weights during training using the perceptron learning algorithm (Figure 1a).

$$\text{if } y \leq 0 \text{ or } y \leq \vartheta \text{ then}
\begin{align*}
\text{for all } i, & \\
\text{update } a_i \text{ to minimize } \|w\| \text{ with } & \|y \|
\end{align*}$$

(a)

$$\text{if } y > 0 \text{ or } y > \vartheta \text{ then}
\begin{align*}
\text{for all } i, & \\
\text{update } a_i \text{ to minimize } & \|w\| \text{ with } \|y \|
\end{align*}$$

(b)

Figure 1: The update algorithms for the Perceptron (a) and SVM (b). $\vartheta$ is a learning threshold parameter.

$t \subseteq \{0,1\}$ is the true classification of the vector $x$. The sign of $y$ is the classification hypothesis for $x$, given by Equations 1 and 4 for the perceptron and SVM respectively.

How perceptrons are used in value prediction:

M. Black[22], in his M.S. thesis, “Perceptron-based Global Confidence Estimation for Value Prediction” showed important work in value prediction using perceptrons.

There are several reasons why to apply perceptrons to value prediction.

First, perceptrons can capture global value correlations that a table-based predictor cannot capture, allowing potentially greater prediction accuracy.

Second, value prediction has several characteristics in common with branch prediction that make a similar perceptron approach promising: prediction times must be low latency, predictions are made by instruction requiring a per-address framework, some past values correlate while others do not, and so on. Third, value prediction requires the learning of correlations between whole data values, instead of correlations between individual binary decisions. The perceptron model used in branch prediction consequently cannot be directly applied to value prediction. Thus a novel approach is required, giving further insights into the perceptron.

He presented four basic perceptron approaches to value prediction. The first approach is a local approach that makes a prediction using information solely from previous instances of the instruction under prediction. This approach directly replaces the previous table-based approaches. The second approach uses global information to predict a local value; it can only predict a value previously seen locally, but it uses information from other instructions to choose that value. The third approach uses both global information and global past values to make a prediction. The fourth approach is a bitwise prediction approach that does not explicitly predict a past data value. Instead it tries to detect correlations between individual bits of past data values to potentially predict new data values.

The local table-based context-based predictor [8] is generally considered one of the best practical value predictors. He proposed two perceptron-based local value predictors that are based on the table-based predictor. The first, which replaces the counters in the pattern table with perceptrons, has a 1.4% to 2.8% lower accuracy than the table-based predictor. The second, which eliminates the second-level pattern table, and uses the local value history to train, is capable of considering significantly longer local histories than the table-based predictor. It performs with 2.4 to 5.6% better accuracy, and 0.5 to 1.2% higher instructions-per-cycle.

He proposed a perceptron-based predictor that uses the past global value history to choose a past local value. He used three different perceptron topologies to learn multiple-bit value correlations: a disjoint topology that considers correlations only between corresponding bits of the different inputs, a fully-coupled topology that considers correlations between all bits of the different inputs, and a weight-per-value topology that considers correlations between past values for each input. The global-local predictor using disjoint perceptron achieves an average accuracy increase of 3.12% and an average relative performance increase of 1.59%, with a storage requirement of 1.18MB. With a weight-per-value perceptron it achieves an accuracy increase of 10.67% and a performance increase of 4.36%, but with a prohibitive storage of 21.5MB.

These are compared to the table-based predictor with a history size of 4 and a history size of 8; the first consumes 69.9KB of storage and the second 33.7 MB of storage, however, they both perform within 0.26% of each other. He proposed a perceptron-based predictor that uses the past global value history to choose a value from a global value cache. When implemented using a disjoint perceptron topology, it achieves an average accuracy increase of 7.56% and a performance increase of 6.69%, with a storage of 1.31 MB.

He finally proposed a bitwise perceptron-predictor that does not save past values, but instead learns correlations between individual bits of each past value and the target values. This perceptron achieves an accuracy increase of 12.67% and a performance increase of 5.28%, while requiring a storage of 4.19MB.

Training-by-error is used as a training strategy for each predictor. Both training strategies are evaluated on the global-local predictor; training-by-correlation performs with 4.82% lower accuracy than training-by-error. This is due to the low percentage of correlated inputs in global value prediction. Exponential weight growth is also considered on the global-local predictor, but it results in an accuracy decrease of 4.51%. He used perceptrons to improve the accuracy of data value prediction. Value prediction was proposed nearly ten years ago as a way of speculatively removing data dependencies in superscalar processors.
A value predictor allows instructions that are dependent on the result of a long latency instruction to execute by guessing the outcome of that instruction and feeding that guess to dependent instructions. These dependent instructions can then execute simultaneously with their parent. The guess is, of course, verified when the parent instruction finishes execution. If the guess is correct, the dependent instructions are permitted to commit; otherwise, they must be executed again. Accurate value prediction may be counter-intuitive, considering the quantity of different possible values that could be produced. However, prediction is possible because data values used by programs often follow easily discernable patterns. Prior research has demonstrated the existence of value locality, or the reuse of data values in a program. In general, a given section of a typical program has a small quantity of data values that it reuses over and over again [5]. Value predictors focus on observing patterns in this value reuse to guess the data value that will be produced by a given instruction.

The original work in value prediction was focused solely on predicting the results of load instructions, particularly those that are undergoing a cache miss [4]. With memory latencies ever increasing, load value predictors remain attractive. Subsequent work extended the research to predicting the results of any long latency instruction, such as floating point arithmetic, multiplication, and division, which can be valuable for computation-intensive programs [5]. While data value prediction has attracted a fair amount of research, it has yet to be widely implemented in actual processors. There are two reasons that are most likely to be responsible for this. The first is that value predictors that have been proposed so far that are feasible to implement typically have fairly poor accuracy rates, ranging broadly from 30 to 80%, depending on the benchmark and processor characteristics. The second reason is that it is difficult to re-execute dependent instructions without high performance penalties. A highly accurate value predictor might be able to withstand high misprediction cycle penalties, while a good misprediction recovery method might be able to allow a low accuracy value predictor to produce performance gains. However, the combination of these two problems presently hinders the actual construction of value predictors.

Previously proposed value prediction strategies have typically captured only a part of the existing value locality. Traditional table-based predictors have difficulty observing value patterns stretching globally between instructions, without becoming too massive to be implementable. Alternative value prediction strategies have already shown themselves to have higher prediction accuracy rates than the table-based approaches. However, in many cases these strategies are either themselves impractical, or capture only part of the global value locality.

His first area of future work is applying the studies to the already existing fields of perceptron-based branch prediction, perceptron-based branch confidence estimation, and perceptron-based confidence estimation for value prediction.

As, According to Daniel A. Jiménez, Composite confidence estimators are able to achieve high degrees of accuracy even when misprediction rates are low[20].

So, Michael Black and Manoj Franklin, in their paper named “Perceptron-based Confidence Estimation for Value Prediction”[24,26], presented a perceptron-based confidence estimator for data value prediction that makes use of correlations between the predictability of different instructions. Perceptrons are used to identify for each instruction which other instructions affect its prediction confidence. The confidence estimator uses this information to raise the accuracy of value prediction. Simulation results show that the perceptron confidence estimator generally offers significant improvement over the conventional up-down counter confidence estimator.

Stride and Last-Value predictors using a Neural-256 confidence estimator can predict 7.8% and 9.1% more instructions, respectively, with a 2.8% and 5.5% accuracy increase on the average than with a 2-bit up-down counter estimator. The Context predictor using the Neural-256 confidence estimator predicts 5.7% more instructions than with the Counter-4 up-down counter but suffers a 2.1% decrease in accuracy.

To uncover predictability dependencies, he used a perceptron-based confidence estimator. A perceptron is a simple neural network consisting of an adder, a threshold function, and a set of weights implemented by saturating signed integer counters. The perceptron uses these components to guess an output based on a series of inputs.

Given a set of input bits, it computes the dot product of the inputs and the weights, and compares the result to a threshold value, typically 0 (an extra weight is hardwired to an input of 1 to provide a bias). If the result is greater than 0, the perceptron returns “True”; otherwise it returns “False.”

The perceptron determines the values of its weights by learning. When a correct value is found, the perceptron is “trained.” That is, an error value is computed by the difference between the training value and the perceptron output. This error value is multiplied by each input bit and is added to the corresponding weight. In this way, each weight is adjusted so that the desired output is realized from the particular input combination. When applied to confidence estimation, each weight value determines the relationship between a particular past instruction and the current instruction. If a weight value is positive and large, the past instruction’s predictability tends to have a direct effect on the current instruction’s predictability[17]. That is to say, the current instruction’s data predictor tends to predict correctly only when the past instruction’s data predictor predicted correctly. If the weight value is negative and large, the past instruction’s predictability effect is inverse: the current instruction’s data predictor tends to predict correctly only when the past instruction mispredicted. If the weight value’s magnitude is small, the past instruction has been found to have little effect on the current instruction.

**Confidence Estimator Organization:**

The prediction system works as follows: The instruction address is used to select a table entry. This table entry consists of a value predictor, which predicts a value, and a perceptron. The perceptron takes the GPH as its input and uses its weights to determine whether its output is “predict” or “don’t predict”. If the output is “predict”, the value predictor outcome is used as a prediction; otherwise the prediction is not used. Regardless of the perceptron outcome, when the actual result of the instruction is known, it is compared against the prediction.
If they match, a 1 (predicted correctly) is shifted into the GPH at the instruction’s completion stage. Otherwise, a 0 (predicted incorrectly) is shifted into the GPH. The difference between the actual result and the prediction is then used to adjust the perceptron weights and train the perceptron.

**Why Support Vector Machines?**

**Linear separability:**

A limitation of perceptrons is that they are only capable of learning *linearly separable* functions. Minsky and Papert [2] show that perceptrons cannot learn *linearly-inseparable* functions, like XOR, with 100% accuracy. Minsky’s work originally claimed that this was the case for all neural networks, but it was later discovered that linearly inseparable functions can be learned in larger neural networks using hidden layers and more advanced training mechanisms. However, this is still a handicap for the simple single layered perceptron.

Linear separability is classically pictured geometrically in an \( n \)-dimensional space, where \( n \) is the number of inputs. All the possible outputs are placed in the space. If the space can be divided by a plane so that all positive outputs are on one side of the plane and all negative outputs are on the other side, the function is linearly separable [3]. If no plane can be drawn, the function cannot be learned by a perceptron.

Imagine the set of all possible inputs to a perceptron as an \( n \)-dimensional space. The solution to the Equation

\[
\mathbf{w}_0 + \sum_{i=1}^{n} x_i w_i = 0
\]

is a hyperplane (e.g. a line, if \( n=2 \)) dividing the space into the set of inputs for which the perceptron will respond *false* and the set for which the perceptron will respond *true* [8]. A Boolean function over variables \( \mathbf{x}_{1...n} \) is linearly separable if and only if there exist values for \( \mathbf{w}_{0...n} \) such that all of the *true* instances can be separated from all of the *false* instances by that hyperplane. Since the output of a perceptron is decided by the above equation, only linearly separable functions can be learned perfectly by perceptrons. For instance, a perceptron can learn the logical AND of two inputs, but not the exclusive-OR, since there is no line separating *true* instances of the exclusive-OR function from *false* ones on the Boolean plane.

![Diagram](https://via.placeholder.com/150)

**Linear inseparability:**

Linear inseparability arises if a correct prediction on a past instruction causes the current instruction to predict correctly sometimes and incorrectly at other times. Because a correct prediction on a past instruction rarely causes the current instruction to predict incorrectly [24].

**But sometimes this can happen.**

In a perceptron, the effect of an input on the output is determined by its weight. As stated before, a positive weight means that the output varies directly with the input, while a negative weight causes the output to vary inversely with the input. Based on its weight, a 1 at a particular input can make the total output more positive or more negative. However, a 1 at a particular input cannot make the total output more positive sometimes and more negative at other times. Functions tend not to be linearly separable if one input’s effect on the output relies on another input’s effect which can happen in value prediction. And as support vector machines belong to a family of generalized linear classifiers and can be interpreted as an extension of the perceptron, They are both linear and non-linear classifiers.

Thus here I used SVM’s for generating confidence value for prediction.

**Introduction to Support Vector Machines:**

Support Vector Machines (SVMs) [15, 16] are a type of kernel machine, one of a family of learning algorithms. The general concept behind SVMs is that the original input \( \mathbf{x} \in \mathcal{X} \) is mapped onto a higher dimensional feature space \( \mathcal{H} \) by a (potentially non-linear) function \( \phi : \mathcal{X} \rightarrow \mathcal{H} \). The SVM learning algorithm is a linear classifier that discriminates between the samples of \( \phi(x) \) in the new feature space \( \mathcal{H} \). Similar to Equation 1, SVMs classify a new pattern \( x \) using the output \( y \) given by

\[
y = \mathbf{b} + \sum_{i=1}^{m} \alpha_i < \phi(x_i), \phi(x) > , \quad w_0 = \mathbf{b} \quad \text{Ker}(x, w) \quad \text{(2)}
\]

\( \text{Ker}(\cdot) \) is a kernel function, a function that returns the dot product of the image of the two inputs in the higher dimensional feature space. The existence of a kernel function means each input vector \( x \) does not need to be mapped to \( \phi(x) \). In fact, given a kernel function, we do not even need to know \( \phi(\cdot) \). This means we can exploit the features of the higher dimensional space \( \mathcal{H} \) without calculating there; the dot product in \( \mathcal{H} \) can be computed in \( \mathcal{X} \). There are many choices for kernels and, so far, no simple way of choosing the optimal kernel for any particular problem instance. The weights \( w \) are related to the previously seen \( m \) training vectors in the set

\[
\text{Training} = \{ x_i \} \quad \text{by the following}
\]

\[
w = \sum_{i=1}^{m} \alpha_i \phi(x_i) \quad \text{Training} \quad \text{(3)}
\]

By adjusting the values for \( \alpha_i \), the SVM training algorithm discovers a hyperplane in \( \mathcal{H} \) that discriminates between the two categories of training data. However, there may be many discriminating hyperplanes in \( \mathcal{H} \). The SVM learning algorithm chooses the one with the maximum margin around it (the maximum-margin hyperplane).

Intuitively, this prevents overfitting and creates a robust discriminator that will generalize well to new patterns. As it turns out, not every training vector contributes to Equation 3: only those training vectors which are on the margin of the discriminating hyperplane are significant [19]. These are called the support vectors. As these are discovered, they populate a set of support vectors \( \text{SV} = \{ \mathbf{x}_i \} \subset \text{Training} \). Combining this notion with Equations 2 and 3 yields

\[
y = \mathbf{b} + \sum_{i=1}^{m} < \phi(x_i), \phi(x) > = w_0 + \sum_{i \in \text{SV}} \alpha_i < \phi(x_i), \phi(x) > , \quad \text{SV} \quad \text{(4)}
\]

The SVM update algorithm that finds the maximum-margin hyperplane (Figure 1b) and the SV selection algorithm can be implemented in several ways: via gradient ascent, by solving a quadratic optimization problem, by doing sequential minimal optimization, through chunking and decomposition, by doing heuristic selection.
How SVM’s are used in Branch prediction:
Culpepper and Gondree suggested Dynamic branch prediction using SVM’s as follows [27]:

SVMs can be used to learn correlations between the behavior \( \mathcal{E} T = \{ \pm 1 \} \) (“not taken”, “taken”) of the current branch and the global history of previous branch behavior \( \mathcal{E} \mathcal{X}_T = T^h \). In this section, we give some intuition of how the predictor can be implemented in hardware and some rough storage costs for the predictor ignoring issues such as the time of calculation and the time to access values from cache. They used a global pattern history register of \( h \) bimodal values to store \( x \), the behavior of the last \( h \) branches. In other words, \( \mathcal{X} = \{ \pm 1 \}^h \). This vector, called PHT, can be represented as a register of \( h \) boolean values.

Their predictor uses \( n \) SVMs, similar to how Jimž’ilov et al employ a table of perceptrons [20]. Under this model, each SVM only needs to learn a small category of branches, effectively splitting the work of classifying all branches in the program between several SVMs. The \( i \)th SVM, which we informally call SVM[\( i \)], is distributed between two tables in hardware: the weight table \( (\text{SVM})_w \) and the table of support vectors \( (\text{SVM})_v \). For each SVM, we set a strict maximum of \( m \) on the number of support vectors that can be collected during incremental training.

When a branch is encountered:
1. The branch address is hashed to index \( i \), to access SVM[\( i \)].
2. SVM[\( i \)]’s weights, SVM[\( i \)]\(_{w} \), are fetched into an \((m+1)\)-dimensional vector register of floating point weights, \( \alpha = (\alpha_0, \ldots, \alpha_m) \). In parallel, SVM[\( i \)]’s support vectors are fetched from \( (\text{SVM})_v \) into \( m \) \( h \)-dimensional bimodal vectors \( \mathcal{S}_V \).
3. The dot products \( k_i = \text{Ker}(\text{PHT}, \mathcal{S}_V) \) for \( i \in \{1 \ldots m\} \) are calculated in parallel. There are many simple kernels available for which this is a fast computation.
4. The floating point multiplications \( \alpha_k \cdot k_i \) for \( i \in \{1 \ldots m\} \) are calculated in parallel.
5. The results of the multiplications and the bias, \( y = a_0 + \sum_{k=1}^{m} a_k k_i \), are summed. This sum of \( m+1 \) values may be done quickly with a Wallace-tree of carry-save adders.
6. The final prediction is the sign of \( y \). The predictor stores the value of \( i \) for later training of SVM[\( i \)].
7. When the true behavior \( t \) of the branch is known, shift the values in PHT, add \( t \), and train SVM[\( i \)].

There are many training and selection techniques, each of which would incur a different computational cost but would not, in general, increase the hardware complexity of the predictor. The global pattern history table PHT is of size \( h \). To store the \( n \) SVMs described above requires a \( \text{SVM}_w \) table of size \( bm(m+1) \) and a \( \text{SVM}_v \) table of size \( nnh \), where \( b \) is the weight’s floating point precision bits. Combined, the basic hardware cost of the predictor is \( bn + bnm + nnh + h \).

**Experimental results:**
By taking, the convention below, the experimental results obtained using SVM’s in branch prediction are:
- \( h \)- The size of the global pattern history table PHT
- \( n \)- The number of SVMs used
- \( m \)- The maximum number of support vectors each SVM may accumulate

\( \text{Hash} (\cdot ) \)- The hash function used to map each branch address to one of the \( n \) SVMs

\( \text{Ker} (u, v) \)- The kernel function

**Algorithm** \( \text{SVM}_{\text{train}}(\theta, \ldots) \)- The algorithm to train each SVM, a function of the learning error \( \theta \)

They chose to experiment with the radial basis kernel function, \( \text{Ker}_{\text{radial}}(\cdot ) \). As a naive first hash function, we chose Hash (PC) = PC/4 mod n. (divide by 4 because each branch address is word-aligned.)

Figure 3: A breakdown of the hardware budgets for each simulation of \( gshare \), tuned fast path-based perceptron, and untuned SVM predictors. Above, \( h \) is the length of the PHT and \( n \) the number of saturating counters, perceptrons, or SVMs respectively. For the SVM predictor, \( m \) is the number of support vectors.

Figure 4 shows the misprediction rates of each simulated predictor for each hardware budget. We find that at much larger hardware budgets than have been previously considered in the literature, the SVM predictor can yield improved accuracy over the current state-of-the-art predictors. SVM predictor was 7% more accurate than \( gshare \) and 16% more accurate than FPBN at 1 MB, and 16% more accurate than \( gshare \) and 24% more accurate than FPBN at 10 MB.

**Comparison between Perceptron and SVM:**
Today, support vector machines and along with other learning based-kernel algorithms show better results than artificial neural networks and other intelligent or statistical models, on the most popular benchmark problems[23]. A. Zanaty[29], introduced a new kernel function for improving the accuracy of the Support Vector Machines (SVMs)
classification for both linear and non-linear data sets. The proposed kernel function is stated in general form and is called Gaussian Radial Basis Polynomials Function (GRPF) that combines both Gaussian Radial Basis Function (RBF) and Polynomial (POLY) kernels.

A comparative analysis of SVMs versus the Multilayer Perception (MLP) for data classifications is also presented to verify the effectiveness of the proposed kernel function.

**Table 3** The mean accuracy SVMs and MLP accuracy.

<table>
<thead>
<tr>
<th>Support vector machine</th>
<th>Multilayer neural networks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poly</td>
<td>84.38</td>
</tr>
<tr>
<td>RBF</td>
<td>85.77</td>
</tr>
<tr>
<td>PRBF</td>
<td>92.38</td>
</tr>
<tr>
<td>GRPF</td>
<td>95.79</td>
</tr>
</tbody>
</table>

Table 3 shows the comparison between the SVMs and MLPs classifiers, it is clear that the SVMs with the kernel (GRPF) achieve higher accuracy than MLP classifier and other kernels.

**Figure 5. The relation between mean accuracy of SVMs kernel and MLP**

His proposed GRPF kernel has achieved the best accuracy. So, from the results it is clear that whatever kernel function we use, except POLY, SVM’s work better than any other artificial neural network.

**Conclusion:**

Based on the previous work, it will be safe to assume that SVM’s work better than perceptrons. In case of branch prediction and confidence estimation, both produce same two classes, i.e. “taken”, “not taken” or “predict”, “don’t predict”. So, I propose a SVM based confidence estimator that estimates the confidence value for prediction, where prediction will be done by data value predictors. Support Vector Machines are used to identify which past instructions affect the accuracy of a prediction and to decide based on their results whether the prediction is likely to be correct or not. The confidence estimator raises the accuracy of value prediction.

In my future work I would like to use perceptrons for value prediction and use a confidence estimator using SVM’s along with it to gain more accuracy.

**References:**


[27] Culpepper and Gondree, “SVMs for Improved Branch Prediction” *ECS201A Computer Architecture* [2008].


[29] E.A. Zanaty “Support Vector Machines (SVMs) versus Multilayer Perception (MLP) in data classification” *Egyptian Informatics Journal* (September 2012) 13, 177–183