SVMs for Improved Branch Prediction

ECS201A Computer Architecture, Prof. Matthew Farrens

Benjamin J. Culpepper
culpepper@ucdavis.edu

Mark Gondree
gondree@cs.ucdavis.edu

Abstract

This technical report is a preliminary investigation into the use of Support Vector Machines *(SVMs) as a method of branch prediction. We present a new dynamic branch predictor based on SVMs. The SVM predictor, at the cost of a much larger hardware budget, can return a greater accuracy than current state-of-the-art predictors by exploiting its ability to learn linearly inseparable boolean functions, a limitation of many well-known dynamic branch predictors. Our untuned SVM predictor yields a 24% improvement over the best available dynamic neural-method branch predictor and a 16% improvement over* gshare *on the* `SPEC95 go` *benchmark at a cost of 10 MB. Tuning the SVM parameters would likely result in further performance gains. Branch prediction with SVMs is largely unexplored in the literature. These favorable results suggest it is worthy of further investigation.*

1 Introduction

Modern computer architecture design increasingly puts pressure on the performance of accurate branch predictors to exploit instruction-level parallelism. Absurdly accurate branch prediction is required to support speculative execution in deeply pipelined machines and in machines with a high rate of instruction issue [1]. The merit in discovering better performing predictors is well known.

There is a growing trend among researchers to apply machine learning techniques to the problem of branch prediction, starting with Vintan *et al* [2]. Our work follows this direction, exploring a new avenue of research: the use of *Support Vector Machines* (SVMs) [3, 4] for dynamic branch prediction.

1.1 Paper Outline

This paper is organized as follows: Section 2 reviews previous work involving machine learning techniques applied to branch prediction. Section 3 gives a basic introduction to SVMs and the notation used throughout this paper. Section 4 proposes an abstract dynamic branch predictor using SVMs. Section 5 explains the experimental methodology and gives accuracy results from our initial experiments. Section 6 delineates important recommendations for future experimental work.

2 Machine Learning in Branch Prediction

The scenario for branch prediction is perfectly suited for machine learning: a mechanism makes a prediction (in this case the boolean response "taken" or "not taken") and soon afterwards gets feedback on whether its guess was correct or not, so it can improve future predictions.

Due to the computational complexity of many machine learning algorithms, online learning, which incrementally adjusts the classifier to account for new examples as they are presented, is often not realizable. Calder *et al* [5] train neural networks on static data (*evidence-based static prediction*) with the expectation that the pre-trained net performs well when deployed as a predictor. Such techniques often suffer from over-training and do not generalize well to unseen data.

Emer and Gloy [6] use genetic algorithms to "evolve" predictors within a high-dimensional design space to search for accurate predictors. The prediction in this model is not a boolean value "taken" or "not taken", but is instead an individual from the family of predictors described by a certain set of design constraints. This is a machine learning technique that is used off-line to design predictors.

Vintan *et al* [2] investigate the use of neural networks trained online using the Learning Vector Quantization algorithm as dynamic prediction mechanisms. Unlike previous efforts that emphasized the static application of machine learning, this application moves the intelligence into the predictor, demonstrating online learning.

Fern *et al* [7] use *decision trees* to do *dynamic feature selection* for classical branch predictors, which use a table of saturating counters. In this model, online machine learning is used to combat the problem that classical predictors cannot exploit large feature spaces (i.e. long branch histories) for predictions. The most useful features are dynamically culled from a larger feature space using decision trees, and then fed to classical predictors.

2.1 Perceptron Branch Prediction

The use of the perceptron as a predictor was first suggested by Vintan *et al* [2]. The perceptron is one of the simplest models of a neuron and was developed by Rosenblatt [8] 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 $\mathbf{w} = (w_1, \ldots, w_n)$ and the inputs $\mathbf{x} = (x_1, \ldots, x_n)$, with the bias input b, which can be thought of as a weight w_0 with constant input $x_0 = 1$

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

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

Jiménez *et al* introduce three dynamic predictors based on the perceptron:

the *global perceptron predictor* [9, 10] uses global branch pattern history data for input $\mathbf{x}$.

the *global/local perceptron predictor* [11] uses both global and local pattern histories for $\mathbf{x}$.

the *fast path-based perceptron predictor* [12] is a global perceptron predictor where each $x_i w_i$ calculation is performed incrementally along the path leading to the branch. This both reduces the observed calculation time and correlates each output y with the path leading to the branch (whereas before y is calculated at, and therefore correlated with, only the branch's address).

The results are promising. The global perceptron improved branch misprediction rates on the `SPEC2000` integer benchmarks by 10.1% over the best dynamic branch predictor available in the literature [9]. Follow-up work [13, 14, 15, 16] has explored methods to improve predictor accuracy, to make predictions available earlier, and to make operation less expensive on-chip. The sudden activity in perceptron prediction research is testament to the power of this technique. The recent inclusion of a neural branch prediction toolsuite in Intel's internal IA-64 simulation environment is further evidence [17].

The perceptron, however, has well-established limitations: Minsky and Papert [18] show that perceptrons cannot learn *linearly-inseparable* functions, like XOR, with 100% accuracy. Accordingly, Jiménez introduces the notion of *linearly-inseparable branches*, a series of branch pattern history vectors $\mathbf{x}$ that *cannot* be discriminated by a linear function into the classes "next branch taken" and "next branch not taken." The poor performance of the perceptron predictor for linearly-inseparable branches is acceptable only in light of the fact that classical predictors perform nearly as poorly for linearly-inseparable branches [9].

Our work is a natural extension of this previous work in neural methods for dynamic prediction. There are two basic options to learn linearly-inseparable functions: use a combination of linear classifiers to approximate a non-linear classifier, or map data into a higher-dimensional space and then use a linear classifier. The first option yields multi-layer neural networks. These are limited in many respects: there are many parameters to tune (and few techniques for tuning them, other than exhaustive search), the learning algorithms rely on heuristics and are prone to getting "stuck" in local minima of the solution space [19]. The second option yields Support Vector Machines.

if $ty \le 0$ or $|y| \le \theta$ then
    for all w_i
        $w_i \leftarrow w_i + tx_i$
	if $ty \le 1$ or $|y| \le \theta$ then
	    for all $\mathbf{x}_i \in SV$
	        update α_i to
	        minimize $\langle \mathbf{w}, \mathbf{w} \rangle$ with $ty \ge 1$

(a)	(b)

Figure 1: The update algorithms for the Perceptron (a) and SVM (b). θ is a learning threshold parameter. $t \in \{\pm 1\}$ is the true classification of the vector $\mathbf{x}$. The sign of y is the classification hypothesis for $\mathbf{x}$, given by Equations 1 and 4 for the perceptron and SVM respectively.

3 Introduction to Support Vector Machines

Support Vector Machines (SVMs) [3, 4] 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 $\phi(\cdot) : \mathcal{X} \rightarrow \mathcal{H}$. The SVM learning algorithm is a linear classifier that discriminates between the samples of $\phi(\mathbf{x})$ in the new feature space $\mathcal{H}$. Similar to Equation 1, SVMs classify a new pattern $\mathbf{x}$ using the output y given by

$$y = b + \langle \phi(\mathbf{x}), \phi(\mathbf{w}) \rangle = w_0 + \text{Ker}(\mathbf{x}, \mathbf{w}) \tag{2}$$

$\text{Ker}(\cdot)$ 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 $\mathbf{x}$ does not need to be mapped to $\phi(\mathbf{x})$. 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 $\mathbf{w}$ are related to the previously seen m training vectors in the set $Training = \{x_j\}$ by the following

$$\mathbf{w} = \sum_{j=1}^{m} \alpha_j \phi(\mathbf{x_j}) \tag{3}$$

By adjusting the values for α_j, 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 [20]. These are called the *support vectors*. As these are discovered, they populate a set of support vectors $SV = \{x_i\} \subseteq Training$. Combining this notion with Equations 2 and 3 yields

$$y = b + \langle \phi(\mathbf{x}), \phi(\mathbf{w}) \rangle = w_0 + \sum_{\mathbf{x_i} \in SV} \alpha_i \langle \phi(\mathbf{x}), \phi(\mathbf{x_i}) \rangle = \alpha_0 + \sum_{\mathbf{x_i} \in SV} \alpha_i \text{Ker}(\mathbf{x}, \mathbf{x_i}) \tag{4}$$

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, etc... See [4, 3] for a more detailed introduction to these topics. We offer suggestions for optimized, online, sequential learning and selection algorithms from the literature which may be applied to the problem described here in Section 6.3.

4 Dynamic Branch Prediction with SVMs

SVMs can be used to learn correlations between the behavior $t \in \mathcal{T} = \{\pm 1\} = \{$“not taken”, “taken”$\}$ of the current branch and the global history of previous branch behavior $\mathbf{x} \in \mathcal{X} = \mathcal{T}^h$. In this section, we give some

intuition of how the predictor can be implemented in hardware and some rough storage costs for the predictor. For this preliminary investigation, we ignore issues such as the time of calculation and the time to access values from cache.

We use a global pattern history register of h bimodal values to store $\mathbf{x}$, 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.

Our predictor uses n SVMs, similar to how Jiménez *et al* employ a table of perceptrons [9]. 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 ith SVM, which we informally call $SVM[i]$, is distributed between two tables in hardware: the weight table (SVM_α) and the table of support vectors (SVM_{SV}). 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, $SVM_\alpha[i]$, are fetched into an $(m+1)$-dimensional vector register of floating point weights, $\alpha = (\alpha_0 \ldots \alpha_m)$. In parallel, $SVM[i]$'s m support vectors are fetched from $SVM_{SV}[i]$ into m h-dimensional bimodal vectors $SV_1 \ldots SV_m$.

3. The dot products $k_i = \text{Ker}(PHT, SV_i)$ for $i \in \{1 \ldots m\}$ are calculated in parallel. There are many simple kernels available for which this is a fast computation. (Section 6.2 offers some suggestions on kernel selection.)

4. The floating point multiplications $a_i * k_i$ for $i \in \{1 \ldots m\}$ are calculated in parallel. The parallel floating point multiplications can be approximated in one cycle using an array of CID/DRAM cells and analog-to-digital converters, as in Genov and Cauwenberghs [21].

5. The results of the multiplications and the bias, $y = a_0 + \sum_{i=1}^{m} a_i k_i$ are summed. This sum of $m+1$ values may be done quickly with a Wallace-tree of carry-save adders as in Jiménez [11], or using techniques from Genov and Cauwenberghs [21].

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 SVM_α table of size $bn(m+1)$ and a SVM_{SV} table of size nmh, where b is the weight's floating point precision in bits. Combined, the basic hardware cost of the predictor is $bn + bnm + nmh + h$. For any particular fixed hardware budget we can tune each of the parameters $\{n, m, h\}$. We estimate that the cost of the hardware logic for computing each of the prediction and update steps is small in comparison to the cost of the PHT and SVM tables.

5 Experimentation

We evaluated the performance of the SVM predictor using trace driven simulation. Traces were gathered from the `SPEC` Integer benchmarks using the SimpleScalar Tool Set[1] simulating the Alpha instruction set.

5.1 Design Space

The main design parameters for the SVM branch predictor are summarized below.

[1] Available online at `http://www.simplescalar.com/`

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
$\text{Ker}(\mathbf{u}, \mathbf{v})$	The kernel function
$\text{Algo}_{SVM}(\theta, \ldots)$	The algorithm to train each SVM, a function of the learning error θ

We did not (yet) tune these parameters by exhaustively searching the design space to minimize misprediction results, unlike Jiménez *et al* [9]. For its accessibility and reputation as a well-known kernel function, we chose to experiment with the radial basis kernel function, $\text{Ker}_{radial}(\cdot)$. As a naive first hash function, we chose $\text{Hash}(PC) = PC/4 \bmod n$. We divide by 4 because each branch address is word-aligned.

5.2 Simulation

Ad-hoc Incremental SVM learning algorithm. The Algo_{SVM} used here is similar to the classical SVM training algorithm most often provided in the literature, based on solving a quadratic optimization problem, described by Vapnik [22]. The specific algorithm[2] is described by Joachims [23].

To simulate incremental training in an online scenario, each SVM is trained so its j-th prediction is based only on the previous w branches it has witnessed. It can be thought of as training under a sliding window. That is, $SVM[i]$'s j-th prediction is based on training with set $Training_j = \{j-w, \ldots, j-1\}$. Likewise, its $j+1$ prediction is based on training with set $Training_{j+1} = \{j-w+1, \ldots, j\}$.
This ensures that the number of support vectors has a maximum, with $m \leq w$. It also roughly simulates the behavior of the fully-online incremental SVM algorithm [24] we were not able to implement (further discussion in Section 6.3) using the *leave-one-out* feature on the $j-w$-th support vector.

After a little trial-and-error investigation, we chose $\theta = 0.001$.

Sampled Ad-hoc Incremental SVM learning simulation. Let the function $\text{N}(i)$ count the number of branches in the trace whose address will hash to i. It is the number of times $SVM[i]$ will be queried for a prediction.

To speed-up the simulation, we only simulated a fraction k of the activity of each SVM. That is, for each i, the training and prediction of $SVM[i]$ is simulated for $k\text{N}(i)$ randomly selected branches. For each randomly selected branch, the PHT is populated with the results of the h preceding branches in the trace, $SVM[i]$ is trained on the results of the w previous branches from the trace whose addresses hash to i, the prediction is made, and it is compared with the actual behavior from the trace. This technique is known as stratified sampling; *stratification* is the process of grouping members of the population into relatively homogeneous subgroups before sampling. Statisticians believe this technique yields an accuracy representative of the predictor's accuracy throughout the entire trace.

5.3 Simulation Results

Performance was simulated over traces of 10,000,000 branches for the `SPEC95` benchmark `099.go`. We chose `go` because it is the "most difficult" benchmark in terms of accuracy according to Jiménez *et al* [12], and therefore leaves the most room for improvement. We also believe that performance gains on `go` will generalize to other benchmarks.

Figure 2 summarizes some simulation parameter values that were simulated: for *gshare*, n and h are related by $n = 2^h$; for the Fast Path-Based Neural ($FPBN$) predictor, h was tuned using Newton's method; and for SVM, we began optimizing n and h using Newton's but report on intermediate values (before convergence) for now. Figure 3 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. We also report the surprising result that

[2]The software that implements this algorithm, SVM^{light}, is available online at `http://svmlight.joachims.org/`

Hardware Budget	*gshare*		*fpb perceptron*		*SVM predictor*		
	h	n	h	n	h	n	m
1 MB	21	$2^{21} = 2097152$	41	4000	20	50	1200
10 MB	24	$2^{24} = 16777216$	43	200000	20	500	739
20 MB	25	$2^{25} = 33554432$	43	390167	20	1000	668

Figure 2: 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 is the number of saturating counters, perceptrons, or SVMs respectively. For the SVM predictor, m is the number of support vectors.

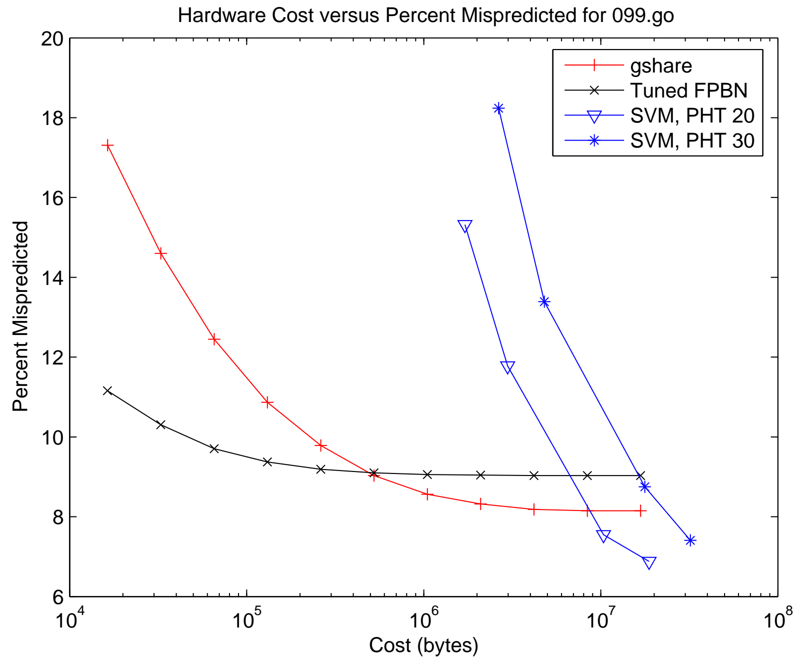

Figure 3: Hardware budget versus average misprediction rate over the `SPEC95` benchmark `099.go`.

gshare appears to outperform *FPBN* at hardware budgets greater than 768 KB. Our untuned 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. At hardware budgets smaller than 1 MB, the performance of SVM is poorer than that of both other classifiers, though we believe tuning the SVM parameters will diminish the difference considerably (as well as improve performance at higher costs).

The SVM predictor was evaluated for $n = 1000, 500, 100,$ and 50 machines, and PHTs $h = 20$ and 30. The number support vectors, m, was adjusted dynamically by the learning algorithm. The poorer performance of PHT $h = 30$ as compared to PHT $h = 20$, with fixed $n = 1000$ and higher cost, indicates that the optimal h value lies on the lower side of 20 ($h < 20$).

6 Future Work

As there is no available evidence of research in applying SVMs to branch prediction, there is much future work to be done. This report indicates SVMs are promising branch predictors, capable of providing increased accuracy as the available hardware budgets grow and cause other techniques' accuracy per cost functions to "saturate" as

do *gshare* and *FPBN* in Figure 3.

Beyond branch prediction, there may be useful application of SVMs to the problems of value prediction (to enable speculation on register values), next trace prediction (for fetching traces from a trace cache), and as the basis for a cache replacement policy (to reduce miss rate by adapting dynamically to the program's cache accesses).

6.1 Improvements Specific to this Investigation

We have sampled only sparsely from the design space parameters, and believe that optimization of the parameters will lead to performance gains. For each hardware budget, Jiménez *et al* exhaustively search the perceptron predictor's design space for the h value that maximizes performance. h is thusly *tuned* for performance. Jiménez *et al* used experimentation to derive a relationship between the learning variable θ and the design parameters h and n, and we expect to find a similar relation in SVM's learning parameter θ. We also suspect there may be a near-linear relationship between n and m (the number of machines and the number of support vectors each machine will need during training). Data to support these suspicions are not yet available, due to lack of CPU cycles for simulation. To address this deficiency we have developed an extension to our simulator that spreads the computational load of sampling from the design space parameters over many workstations. The problem is receives enormous speedup from parallelization as no inter-node communication is necessary, but storing the simulation results for SVM will require us to procure additional disk space (we have, however, run simulations of *gshare* and *FPBN* this way).

It should be noted that it is unfair to compare the untuned SVM predictor's best performance with that of a tuned perceptron predictor.

To better refine the experimental work here, we would like to further explore the design parameters, and implement an online, incremental SVM learning algorithm that is called as the simulator runs. This will enable more comprehensive comparisons between our technique and other, well-known methods.

6.2 Kernel Selection

Experimentation with different kernels would be worthwhile. There are many kernels available and custom easily-computable kernels could be developed for the problem of branch prediction. We experimented with the radial basis kernel, $\mathrm{Ker}_{\mathrm{radial}}$, because it was the most accessible for simulation.

Specifically, we suggest experimentation with a kernel developed independently by both Sadohora [25] and Khardon *et al* [26] specifically for learning boolean functions:

$$\langle \phi(\mathbf{x}), \phi(\mathbf{v}) \rangle = \mathrm{Ker}_{\mathrm{DNF}}(\mathbf{x}, \mathbf{v}) = -1 + \prod_{i=1}^{h} (2x_i v_i - x_i - v_i + 2)$$

where $\phi(\cdot)$ maps the h-variable boolean data to a much larger feature space of dimension $3^h - 1$. For each design choice of h, the product can be expanded statically and optimized to simplify calculation even further.

6.3 Learning Algorithm Selection

A more accurate simulation would utilize an online algorithm, and there are several available in the literature that would serve our purpose. Cauwenberghs and Poggio [24] have developed an incremental learning algorithm for SVMs that has successfully been deployed in silicon [21]. Other incremental online SVM learning algorithms may also be fruitful to investigate.

6.4 Other Suggestions

Many lessons learned from 3 years of research on neural branch prediction can also be applied to SVMs. To decrease the time to produce a prediction, one might incrementally compute the values in Equation 4 as we approach the branch [12], or tweak the prediction algorithm to begin computation of Equation 4 early by using old history data and injecting newer data as it becomes available [14]. To increase predictor accuracy, one might

use a redundant history table [13, 16], try a combination of global and local pattern histories [11], or find a better behaving hash function [15].

A potential weakness of the SVM predictor is its increased hardware budget. We expect it will be possible to trim costs by tuning the parameters, careful avoidance of duplication of support vectors from one SVM to another, and using lower precision floating point registers. The focus of this paper is on foundational work and on initial experiments to investigate accuracy, not trimming costs. We hope to explore this issue further and discover parameterizations that improve the SVM predictor's accuracy per cost at limited cost budgets. Many of the suggestions in Section 6 may reveal insight towards this goal.

Prediction delay is an important factor to be considered when evaluating a branch prediction technique. Ideally, a predictor can be evaluated in a single cycle, but using Wallace-tree adders may preclude operations in fewer cycles than the depth of the tree. Fortunately, these trees can be shallow, and there is no reason the SVM branch predictor cannot run at a multiple of the host CPU clock, or even completely asynchronously so long as the prediction is available in time.

We present the SVM's prediction as the sum of m values, executed in a single stage. One method of reducing the delay incurred by this stage is to calculate the prediction incrementally, as Jiménez has done for the perceptron predictor [12]. Another approach is to reduce the impact of a delay, as suggested by Jiménez *et al* [27]. For example, the SVM predictor can be used as an overriding predictor. In this scenario the CPU consults a small, quick method for an initial prediction, and overrides this initial prediction at a small cost if the accurate-but-complicated SVM predictor later disagrees.

7 Conclusion

In this report we have proposed an entirely new application of SVMs, one which the computer architecture community has yet to explore in the literature: branch prediction. The key advantage of SVMs is their ability to robustly predict linearly-inseparable branches, a problem for both current state-of-the-art neural-method and saturating-counter predictors.

This initial investigation has revealed that an untuned SVM predictor with a large hardware budget can have 24% fewer mispredictions than Jiménez's tuned path-based perceptron at a comparable budget. We believe that these results are just a suggestion of the SVM predictor's full capability. As noted in Section 6, there are many avenues to explore in making a SVM-based predictor faster and more accurate in the future.

References

[1] John L. Hennessy and David A. Patterson, *Computer Architecture A Quantitative Approach Third Edition*, Morgan Kauffman Publishers, 2003.

[2] Lucian N. Vintan and Mihaela Iridon, "Towards a high performance neural branch predictor," in *Proceedings of the 1999 International Joint Conference on Neural Networks*. July 1999, vol. 2, pp. 868–873, IEEE Computer Society.

[3] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines*, Cambridge University Press, 2000.

[4] Bernhard Schölkopf, Chris Burges, and Alex Smola, Eds., *Advances in Kernel Methods - Support Vector Learning*, MIT Press, 1999.

[5] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn, "Evidence-based static branch prediction using machine learning," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 1, pp. 188–222, 1997.

[6] Joel Emer and Nikolas Gloy, "A language for describing predictors and its application to automatic synthesis," in *Proceedings of the 24th annual international symposium on Computer architecture*. 1997, pp. 304–314, ACM Press.

[7] Alan Fern, Robert Givan, Babak Falsafi, and T. N. Vijaykumar, "Dynamic feature selection for hardware prediction," Tech. Rep. TR-ECE 00-12, Purdue University, School of Electrical and Computer Engineering, 2000.

[8] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, Spartan, 1962.

[9] Daniel A. Jiménez and Calvin Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the Seventh International Symposium on High Performance Computer Architecture*, 2001, pp. 197–206.

[10] Daniel A. Jiménez and Calvin Lin, "Perceptron learning for predicting the behavior of conditional branches," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN-01)*, July 2001.

[11] Daniel A. Jiménez and Calvin Lin, "Neural methods for dynamic branch prediction," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 369–397, November 2002.

[12] Daniel A. Jiménez, "Fast path-based neural branch prediction," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. 2003, p. 243, IEEE Computer Society.

[13] A. Seznec, "Redundant history skewed perceptron predictors: pushing limits on global history branch predictors," Tech. Rep. 1554, IRISA, 2003.

[14] David Tarjan, Kevin Skadron, and Mircea Stan, "An ahead pipelined alloyed perceptron with single cycle access time," in *Proceedings of the 2004 Workshop on Comlexity Effective Design*, June 2004.

[15] David Tarjan and Kevin Skadron, "Revisiting the perceptron predictor again," Tech. Rep. CS-2004-28, University of Virginia, Dept of Computer Science, September 2004.

[16] A. Seznec, "Revisiting the perceptron predictor," 1620, IRISA, 2004.

[17] Edward Brekelbaum, Jeff Rupley II, Chris Wilkerson, and Bryan Black, "Hierarchical scheduling windows," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. 2002, IEEE Computer Society.

[18] M.L. Minsky and S.A. Papert, *Perceptrons*, MIT Press, 1969.

[19] Nello Cristianini, "A tutorial on support vector and kernel machines," June 2001, `http://www.support-vector.net/icml-tutorial.pdf`.

[20] Bernhard Schölkopf, "Svm and kernel methods," December 2001, `http://www.kernel-machines.org/papers/tutorial-nips.ps.gz`.

[21] Roman Genov and Gert Cauwenberghs, "Kerneltron: Support vector "machine" in silicon," *IEEE Transactions on Neural Networks*, vol. 14, no. 5, September 2003.

[22] Vladimir N. Vapnik, *The Nature of Statistical Learning Theory*, Springer, 1995.

[23] T. Joachims, *Advances in Kernel Methods - Support Vector Learning*, chapter 11, Making large-Scale SVM Learning Practical, pp. 169–184, MIT Press, 1999.

[24] Gert Cauwenberghs and Tomaso Poggio, "Incremental and decremental support vector machine learning," in *Advances in Neural Information Processing 13*, 2000, pp. 409–415.

[25] Ken Sadohara, "Learning of boolean functions using support vector machines," in *Proceedings of the 12th International Conference on Algorithmic Learning Theory*. 2001, pp. 106–118, Springer-Verlag.

[26] R. Khardon, D. Roth, and R. Servedio, "Efficiency versus convergence of boolean kernels for on-line learning algorithms," Tech. Rep. TR-2004-12, Tufts University CS Dept, 2004.

[27] Daniel A. Jiménez, Stephen W. Keckler, and Calvin Lin, "The impact of delay on the design of branch predictors," in *International Symposium on Microarchitecture*, 2000, pp. 67–76.