Applications of Machine Learning Techniques to Systems

Lecture #11: Thursday, 22 May 2003
Lecturer: Chi Ho Yue, Ilya Katsnelson
Scribe: Jean Suh, Arjun Singh

1 Dynamic Branch Prediction with Perceptrons Introduction

Modern computer architectures increasingly rely on speculation to boost ILP. Accurate prediction increases the performance benefit of speculation. Recent effort to improve branch prediction focuses on aliasing to eliminate destructive interference. Perceptrons, a new method for branch prediction, target at improving the branch prediction accuracy itself.

Perceptrons is a simple neural network that works as an alternative to the commonly used two-bit counters branch history table (BHT) branch predictor. Perceptrons achieves increased accuracy than traditional BHT branch predictors because they can make use of longer branch histories, given the same hardware budget. Although having very similar organization to BHT branch predictors, Perceptrons' hardware requirement scales linearly with the branch history length used, while BHT predictors' increases exponentially.

In a system with branch prediction using Perceptrons. A perceptron is assigned to every single static branch. Each perceptron consists of one artificial neuron connecting several input units by weighted edges to one output unit, as a function Y(x0, …,xn) of n inputs. The xi represents the bits of the global branch history shift register. The predictor uses the following weight function Y to make every branching decision.

$$Y = w0 + w1*x1 + w2*x2 + \ldots + wn*xn$$

The w's are weight in signed integers, assigned to each bits of history. When a previous global branch was taken, its corresponding xi is set to be 1. When that previous branch was not taken, xi is set to be -1. The predicting branch is taken when Y is positive, and vice versa. When the actual branch output is known, the perceptron system will train the predicting perceptron by updating the w's with its corresponding previous branch using correlation concept.

Since Perceptrons uses a dot production function to compute predictions, there lays an issue with the function. Correlations that make the dot product Y zero will not be noticed and make uses by

the perceptron predictor. Such correlation, as introduces by in the paper, is said to be linearly inseparable correlation. Traditional branch predictors using BHT such as gshare and bimode do not have this issue. Therefore, it would be beneficial to use a hybrid of perceptron and a BHT based predictor such as gshare.

Obvious advantages on using Perceptrons-based branch predictors includes that Perceptrons are not difficult to both understand and implement on hardware. Given the same hardware budget, Perceptrons can make use of longer branch history for higher accuracy. On the other hand, Preceptrons would perform very well when branches are linearly inseparable. Perceptrons also concentrate mostly on global branch history but less on local history, which is they only use one weight for local branch history. Lastly, even with the circuit techniques mentioned in the paper, it is not certain that if prediction can be computed in a single cycle when history length becomes longer because quite a series of computation (hash function, loading from SRAM, and arithmetic operations) must be executed before a prediction is obtained.

2 Automated Predictors Synthesis

As processor architectures have increased their reliance on speculative execution to improve performance, the importance of accurate prediction of what to execute speculatively has increased. In addition to that, the types of values that systems attempt to speculate on have expanded from the ubiquitous branch and call/return targets to the prediction of indirect jump targets, cache ways and data values. In general, the prediction process identifies the current state of the system, and makes a prediction for some yet un-computed value based on that state. Prediction accuracy is improved by learning, what is a good prediction for that state using a feedback process at the time the predicted value is actually computed. This paper makes an attempt to formally characterize the process of prediction.

Currently, the majority of predictor designs are based on several well understood structures. This presents some advantages in speed due to the fact that these structures are highly optimized. However, it also prevents some new some specialized designs from appearing. In order to solve this problem the paper presents an algebraic-style notation for describing wide variety of predictor structures in uniform way. A simple primitive predictor is presented as a basic building block of any higher level predictor. The notation for the primitive predictor is as following:

P[w,d](I;U),
Where: w,d – static part of the predictor (memory);
I – input selection signal;
U – feedback update signal.

Special parser language – BP language – is also presented. It allows automatic creation of simulators for predictors. In this case the traces from real benchmark programs can be used to run on the simulators and equally compare the performance of different branch prediction techniques.

The uniformity of predictor nation also permits the use of different automated design search techniques. The paper presents the genetic programming as one of these methods. In order to apply genetic optimization algorithm to predictors each particular predictor is represented by a tree structure, which directly corresponds to BP expression of the predictor. The nodes in the tree represent different logical parts of the predictor – Primitive predictors, Functions, Terminals, and constants. With genetic algorithms new predictors are constantly generated. Their performance is evaluated using the BP parser specified above and the traces from SPEC benchmarks. Replication, Crossover, Mutation, Encapsulation, and Expansion operations were used to generate new predictor structures. Several restrictions to the growth of the predictors had to be put in place in order to ensure their implementation feasibility. In particular the authors restricted the internal memory size of each predictor to 512Kb.

As a result of the application of genetic programming to the problem of finding the optimal branch or jump predictors, the algorithms were able to 'invent' several well known structures staring from just very simple ones. This shows that it is be possible to create better predictor schemes in the future using the same approach.

However, the direct application of genetic programming still presents some challenges. In particular, the automatically generated predictors tend to be very complex and might not be directly implementable. In addition to that, significant amount of manual tuning was still used to 'direct' the genetic optimization process.

3 Discussion

1. *Why is x_0 always set to 1?*
   Xo is set to 1 as an initialization for the system. It also gives an initial bias towards a taken branch to the system at initialization. If more branches are not taken, w_0 starts becoming negative and this bias is reduced.

2. *What is the importance of the threshold in the training process?*
   The threshold value directly affects the confidence of the Perceptrons' prediction. A high threshold value implies a high confidence in the predicted outcome so one can speculate more aggressively. On the other hand, lower confidence on the prediction implies less aggressive speculation. Also if we have a hybrid Perceptrons (with several predictors) we can speculate according to the one with the highest confidence.

The threshold value also determines how long we train the Perceptrons. After reaching the threshold, the Perceptrons gives a high prediction rate and it should not be trained any further, and overtraining would be an overhead to the system. Overtraining could also lead to adversarial behavior if the branching behavior starts to change.

3. *Why does perceptions prediction requires less training time than gshare and bimode which use BHT method?*
In the Perceptron's case, the training is independent of the history length while for BHT every bit per history pattern has to be trained.

4. *How do these schemes do for global history?*
Global branch history affects more than local history in Perceptrons. In other word, there is only one weight for the local branch history.

5. *Where can we apply ML techniques to other application related to CMP?*
ML techniques are a heuristic to automate the exploration of the design space for a given multi-dimensional problem. So, these techniques are especially useful when the design space is very large.

- ML techniques can be used when deciding the configuration of a CMP. Designing the configuration of a CMP has several parameters such as the number of processor cores, caches and memory hierarchy and sizes, bus connections, etc, which the design space is huge.
- ML techniques can be used in the compiler or the operating system to schedule operations or threads.
- ML techniques can be applied to reconfigurable architecture CMP to decide when the architecture needs to be reconfigured and if so, how to configure the processor for better performance.
- ML techniques can also be used for power management i.e. they can be trained to predict the idle time of a processor and power off the processors if they won't be used for a while.

In general ML techniques work only when the application has fixed patterns of behavior. These patterns may change but they change in phases. If the application behaves completely randomly, ML techniques will not help much.

6. *What is the benefit of using the genetic algorithms over the manual search or the simulated annealing?*
In practice, the genetic algorithm gives reasonable results. But, theoretically there are no guarantees that it will give neither optimum results nor reasonable results. The algorithm dose not even guarantees to give reasonable results in a fixed amount of

time. On the other hand, the simulated annealing gives reasonable results in a fixed amount of time and it can also give provably optimal results if there is no timing constraint.

There is more randomness in search procedure and it can be automate, in the way that genetic algorithm searches the design space. But, in opposition, there are limitations of randomness and hard to automate the search in the manual searching (whereby a designer may identify a subset of the design space by its knowledge) or the simulated annealing (whereby the searching is more local with occasional random jumps).

7. *How do we choose the benchmarks for performance evaluation?*

A good benchmark suit consists of applications from the real world (and not just synthetic applications) and a few applications that stress the design. The benchmarks should also contain applications that are important to the end clients as these are the applications that are going to be run by the consumers. However, the number of benchmarks may be large so a subset of them must be chosen for performance evaluation. In order to do that, we run all the benchmarks the first phase and quantify them by weights, and from the result we can select one from each by grouping them according to the weights.

References

[1] J. Emer, N. Gloy. A Language for Describing Predictors and its Applications to Automatic Synthesis. *Proceedings of the 24th International Symposium on Computer Architecture*, Denver, CO, June 1997.

[2] D. Jimenez, C. Lin. Dynamic Branch Prediction with Perceptrons. *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, Monterrey, Mexico, January 2001.

[3] M. Stephenson, S. Amarasinghe, M. Martin, U. O'Reilly. Meta Optimization: Improving Compiler Heuristics with Machines Learning. *Proceedings of the Conference on Programming Languages Design and Implementation*, San Diego, CA, June 2003.

[4] M. Sakr, D. Chiarulli, B. Horne, C. Giles. Predicting Multiprocessor Memory Access Patterns with Learning Models. *Proceedings of the 4th International Conference on Machine Learning*, Nashville, TN, July 1997.

[5] S. McFarling, "Combining Branch Predictors", *WRL Technical Note TN-36*, Digital Equipment Corporation, June 1993.