Exploring Perceptron-Based Register Value Prediction

John Seng
Department of Computer Science
Cal Poly State University
San Luis Obispo, CA 93407
jseng@calpoly.edu

Greg Hamerly
Department of Computer Science
Baylor University
Waco, TX 76798
greg_hamerly@baylor.edu

Abstract

Register value prediction has been proposed as a technique to exploit register value reuse, a form of locality where the result produced by an instruction is the same as the value that is already in a destination register or other registers in the register file. Register value prediction allows increased performance by breaking true dependencies between an instruction that exhibits this locality and its dependents.

This paper presents a study into using perceptron-based predictors to guide one form of register value prediction. For a given storage budget, we find that on the average a perceptron predictor performs better than previously proposed register value predictors. Secondly, we demonstrate the impact of perceptron history length on register value prediction. Lastly, we analyze predictor structures which improve upon previous predictors targeting register value reuse. With a 4KB hybrid perceptron predictor we show an average speedup of 7.5% for the benchmarks studied.

1. Introduction

Value prediction has been proposed as a technique to break true dependencies between instructions. By predicting the input and/or output values of instructions, instructions that were once required to execute serially can be executed in parallel. When a high number of dependencies can be correctly predicted, this parallel execution can lead to higher overall performance.

Implementing value prediction in a processor often requires the use of a storage structure to hold the values which will be used in the prediction. These value files may need to be large in order to obtain good performance. In order to reduce the need for large value files, Tullsen and Seng propose a technique called *register value prediction* [11]. Register value prediction exploits a type of locality called register value reuse. Register value reuse occurs when an instruction produces a value that is the same as the value that is already in the destination register of the instruction or in another register in the register file. Tullsen and Seng demonstrated prediction techniques that allow register value reuse to be exploited via predictor structures, giving higher performance.

One recent innovation in predictor design has been the proposed use of perceptrons in order to implement branch prediction [5, 6]. A perceptron is a simple model of an artificial neuron which can predict boolean events after having been trained on past events. Recent research has demonstrated how perceptrons work well when being trained on the global history of branch outcomes during program execution.

In this paper, we present techniques using perceptron-based predictors to perform a limited form of register value prediction, only the case where the value written to a register will be the same as the value that is currently in the register. We will refer to instructions that produce the same value as that already in the destination as *redundant* instructions. If a dynamic instance of an instruction is redundant and is correctly predicted early in a processor pipeline, then instructions dependent upon that instruction need not wait for the execution of the predicted instruction.

This paper is organized as follows: Section 2 provides background information on perceptrons. Section 3 discusses related studies. Section 4 describes the simulation methodology and tools. Section 5 discusses our results on perceptron-based prediction. Section 7 concludes.

2. Perceptron-Based Prediction

A perceptron [9] is a simple, easy-to-implement model of an artificial neuron from artificial intelligence. See Figure 1 for an example of a typical perceptron. The perceptron typically takes a fixed number of inputs and produces a single numeric output. The perceptron is specified by the number of inputs N, and the weights connecting the inputs to the output node. The weights are the parameters which must be either set by hand or learned by a learning algorithm. Learning the weights online allows the perceptron to adapt to time-varying behavior and does not require expert intervention. A perceptron can be considered a simple single neuron in larger neural networks, which are much more complex and which we do not consider here.

The input to our perceptron is a vector of values (1 or -1) corresponding to the global history of most recent committed instructions. A value of 1 indicates that the instruction was redundant and a -1 indicates otherwise. The history length is the number of inputs to the perceptron. In our predictor, we have chosen not to use the typical bias input, instead allowing one more instruction in the global history.

Perceptrons typically produce a numeric output, which we compare with a preselected threshold value θ to produce a binary value. Our perceptron outputs one (predict redundant) if $\sum i_j w_j \geq \theta$, and zero (no prediction) otherwise.

We use a table of perceptrons for making predictions. To select the perceptron used for a given instruction, the table is indexed by the lower bits of the instruction address. Each perceptron trains and makes predictions independently of all other perceptrons in the table.

We train the perceptron in a manner similar to that described by Jiménez and Lin [5]. Each time an instruction is executed, a perceptron may make a prediction of redundant or not redundant. That prediction is compared with the actual outcome (when the instruction commits), and the perceptron can receive positive or negative feedback, depending on whether or not its prediction was correct. This feedback is used to further train the perceptron. Each weight is updated individually, and the update rule is best depicted in a table:

	was ghr$_i$ redundant?	
is this instr. redundant?	yes	no
yes	increment w_i	decrement w_i
no	decrement w_i	increment w_i

This update rule is used for training on each committed instruction when the absolute value of the output is less than the threshold θ or if the prediction was incorrect. In the case that the prediction is correct and the absolute value of the perceptron output is greater than θ, we do not train the perceptron. The perceptron uses saturating weights.

The two keys to using a perceptron for redundant instruction prediction are finding an appropriate history length and setting the weights well. The history length is limited by the memory available, as is the range of values the weights can use. We try various history lengths (as we show later) and use signed integer weights with the range of 6 or 7 bits.

3. Related Work

Register value prediction [11] has been proposed as a method for value prediction without the need for a large value table. The output values for instructions are chosen from the values in the architectural register file. That work demonstrates that good performance improvements can be obtained from register value prediction. In addition to same register prediction (which is studied in this paper), the authors studied predicting instructions that produce values that are the same as those in dead registers (registers whose value will not be read before being overwritten). Also in that work, the authors looked at predicting instructions statically (by marking instructions in the code) and also dynamically. Our work focuses only on dynamic prediction of register value reuse.

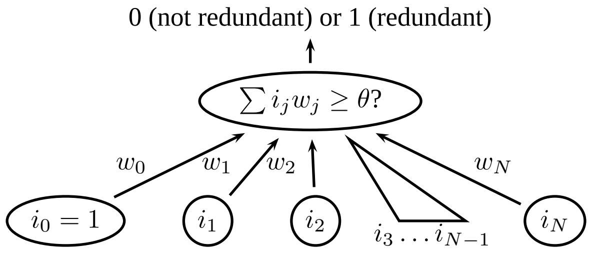

Figure 1. The classical perceptron has a fixed number of inputs $i_1 \dots i_N$ and weights $w_1 \dots w_N$, and a bias term i_0 (constant input of 1) with weight w_0. To make a prediction, the inputs are multiplied by the weights and then summed. If the sum is less than a fixed threshold θ, then the perceptron predicts 0, otherwise 1. The weights of a perceptron are learned with a learning algorithm. In this work we did not use a bias term.

The predictor used by Tullsen and Seng [11] is designed to target local repetitive behavior of individual instructions, but does not look at using global history in their prediction scheme. The perceptron predictor inherently makes predictions based on global history and not on a per instruction basis.

Burtscher and Zorn [3] present a technique which uses outcomes of prediction in order to generate a confidence estimate for future predictions. The patterns they used to trigger a prediction are gathered through profiling and subsequently encoded into the predictor. The perceptron predictor we use does not require any profile information and can be used to generate a confidence for a prediction dynamically.

Balakrishnan and Sohi [1] note that a number of values produced by instructions are often already located in the register file. The authors state that a value produced by an instruction is often the same value that was produced by a recent instruction. This behavior is somewhat similar to the behavior we target in this work. The authors exploit this form of locality by not allocating a register if the instructions are executed closely in time.

Burtscher and Zorn study using a register value predictor as part of hybrid load value predictors [2]. The authors demonstrate that hybrid predictors can outperform single component predictors. They conclude that a register value predictor is a good addition to many hybrid load value prediction schemes. The predictor they use for the register value prediction is similar to the one described by Tullsen and Seng [11].

Jiménez and Lin [5] proposed using perceptrons when developing microarchitectural predictor structures. The authors looked at using perceptron-based predictors to im-

Benchmark	input	Fast forward (millions)
applu	applu.in	1000
art	c756hel.in	2000
bzip2	input.program	2000
crafty	crafty.in	1000
eon	kajiya	100
equake	inp.in	3000
galgel	galgel.in	2600
gap	ref.in	1000
gcc	200.i	10
gzip	input.program	50
mcf	inp.in	1500
mesa	mesa.in	1000
mgrid	mgrid.in	2000
parser	ref.in	300
perlbmk	perfect.pl	2000
twolf	ref	2500
vortex	lendian1.raw	2000
vpr	route	1000

Table 1. The benchmarks used in this study, including inputs and fast-forward distances used to bypass initialization.

prove branch prediction. For this work, we use similar approaches for training and computing the perceptron values, but use the prediction for predicting redundant instruction outputs instead of targeting branch predictions.

4. Methodology

We use the SMTSIM simulator [10] in single-thread mode to perform simulations for this research. The simulator provides an accurate cycle-by-cycle model of an out-of-order processor executing the Compaq Alpha instruction set architecture. Benchmarks were taken from the SPEC 2000 benchmark suite (not all benchmarks could be simulated). All simulations execute 300 million committed instructions. The benchmarks are fast forwarded (emulated but not simulated) a sufficient distance to bypass initialization and startup code before measured simulation begins. The benchmarks used, their inputs, and the number of instructions fast forwarded are shown in Table 1. In all cases, the inputs were taken from among the reference inputs for those benchmarks.

Details of the simulated processor model are given in Table 2. The processor model simulated is that of an 8-fetch 8-stage out-of-order superscalar microprocessor with 6 integer functional units. The instruction and floating-point queues contain 32 entries each. The simulations model a processor with level 1 instruction and data caches, along with a 2MB on-chip secondary cache.

The baseline register value predictor is similar to the configuration used by Tullsen and Seng [11]. It consists of a table of 3-bit saturating counters. The counter values are incremented whenever an instruction produces the same value

Parameter	Value
Fetch bandwidth	8 instructions per cycle
Functional Units Instruction Queues	3 FP, 6 Int (4 load/store) 32-entry FP, 32-entry Int
Inst Cache	64KB, 2-way, 64-byte lines
Data Cache	64KB, 2-way, 64-byte lines
L2 Cache (on-chip)	2 MB, 4-way, 64-byte lines
Latency (to CPU)	L2 18 cycles, Memory 150 cycles
Pipeline depth	8 stages
Min branch penalty	6 cycles
Branch predictor	4K gshare
Instruction Latency	Based on Alpha 21164

Table 2. The processor configuration.

as the value already in the destination and the counter is reset whenever the value is different. The prediction threshold is 6, and an instruction is predicted redundant whenever its counter value exceeds the threshold.

The perceptron predictors we use are similar in configuration to those used by Jiménez and Lin [5]. For a given hardware budget, we use the same history lengths and thresholds used by Jiménez and Lin. Table 3 shows the configurations used for the perceptron predictors. These are the configuration values used in the experiments unless otherwise specified. We have not yet had the opportunity to perform exhaustive threshold and history length studies. A study into the impact of predictor history length is shown in Figure 3. We believe that with further study into threshold selection and history length even more predictor accuracy is achievable.

In this work we assume that loads, integer arithmetic instructions, and floating point arithmetic instructions are all candidates for register value prediction. Stores and control flow instructions are not considered for prediction. We found that on the average an additional 2.28 register read ports per cycle is required for verification for this type of value prediction.

For the recovery mechanism, we simulate a refetch type recovery. Misspeculations are detected in the execution stage of the pipeline. After a misspeculation, all instructions that are fetched after the misspeculated instruction are flushed from the pipeline and fetch begins with the instruction following the misspeculated instruction. This is similar to the recovery mechanism that is used to recover from branch mispredictions in the simulated pipeline. With a more advanced recovery mechanism (such as a reexecution of dependent instructions), we believe even higher speedups can be obtained.

We assume that the latency of the predictor can be pipelined. A possible configuration of low latency perceptron predictors was presented by Jiménez [4]. The maximum perceptron weights used in our studies are 7-bit signed integers, and in a number of cases 6-bit signed integers. We simulated the perceptron predictor using 5 through 8 bits

Parameter	Value		
Total size	4KB	8KB	16KB
History length	28	34	36
Maximum weight	32	64	64
Prediction threshold	68	80	83

Table 3. Table of perceptron predictor configurations.

and select the weight size with the best performance for a given predictor configuration.

5. Results

In this section, we test the effectiveness of perceptrons as register value predictors and proceed to find ways to improve their performance. First, we study the performance of a perceptron predictor on various individual benchmarks. Secondly, we look at the effect that history length has on perceptron predictor performance. Lastly, we look at hybrid predictor structures to improve predictor effectiveness.

5.1. Performance of Perceptron-Based Register Value Prediction

In this section, we compare the performance of an dynamic register value predictor (from now on referred to as *rvp*) as proposed by Tullsen and Seng [11] and a perceptron predictor.

Figure 2 shows the performance improvement achieved for each of the benchmarks when using an 8KB *rvp* predictor, an 8KB perceptron predictor, and the speedup achieved with perfect prediction. There is significant variation in the speedups between applications. The perceptron predictor significantly outperforms the *rvp* predictor on `gap`, `mcf`, and `vpr`. In these three cases we find the perceptron predictor to predict more instructions with register reuse than does *rvp* (for the average of the three benchmarks, 84.7% of the actual number of redundant instructions versus 63.4%) and does so with a higher accuracy (for the average of the three benchmarks, 98.6% prediction accuracy for the perceptron versus 96.5% for *rvp*).

The *rvp* predictor performs better on `crafty`, `mesa`, and `perlbmk`. We find that for each of these benchmarks, the perceptron predictor encounters significantly more mispredictions than with *rvp*.

With perfect prediction, the average speedup across the benchmarks is 13.2%. The perceptron predictor achieves 61% of the maximum speedup possible. There is still more performance to be obtained from register value prediction.

Because of the nature of the *rvp* predictor, it tends to detect local predictability during the execution of individual instructions. In contrast, the perceptron uses a global history register containing the behavior of the last N instructions, where N is the length of the history stored. Therefore it is not surprising to see that there are some applications where the *rvp* predictor excels and some where the perceptron excels. We do find though that over the average of the benchmarks, the perceptron does better. This demonstrates that register value reuse can be effectively predicted with global history information.

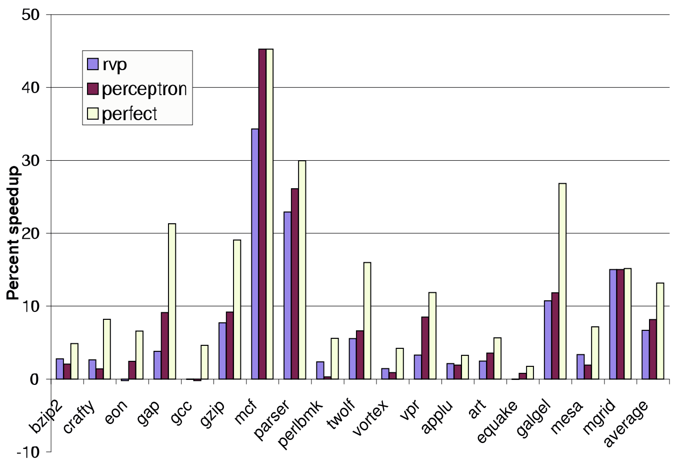

Figure 2. Performance speedup for an 8KB *rvp* predictor and an 8KB perceptron predictor. The potential speedup achieved with perfect prediction is also shown.

5.2. Effect of History Length

Because the perceptron predictor is a global prediction scheme, its accuracy is dependent upon the number of bits stored in the global history register. In this section, we examine the impact that history length has on overall performance.

Figure 3 shows the average speedup obtained when using a perceptron predictor with a varying amount of global history. The predictor used is a perceptron predictor consisting of 8192 perceptrons with 6-bit weights. We use a predictor with a large number of perceptrons to minimize the effect of aliasing. We simulate global histories from 4 to 60 inputs in increments of 4. We choose thresholds based on the thresholds given by Jiménez and Lin [5].

As expected, performance improves with increasing history length. 90% of the speedup achieved with a history length of 60 can be achieved with a history length of 40. 80% of speedup with a history length of 60 can be achieved with a history length of greater than 16.

When increasing history length, both predictor coverage (the fraction of correctly predicted redundant instructions to actual number of redundant instructions) and accuracy are affected positively. We find that coverage increases from 85.6% to 87.8% with history length of 4 and 32 respectively,

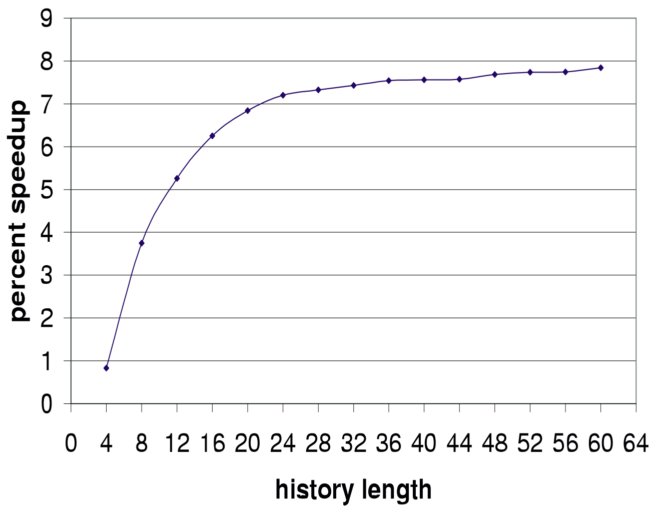

Figure 3. Performance speedup for the average of the benchmarks with varying lengths of global history. The perceptron predictor used in each data point consists of 8192 perceptrons with 6-bit weights.

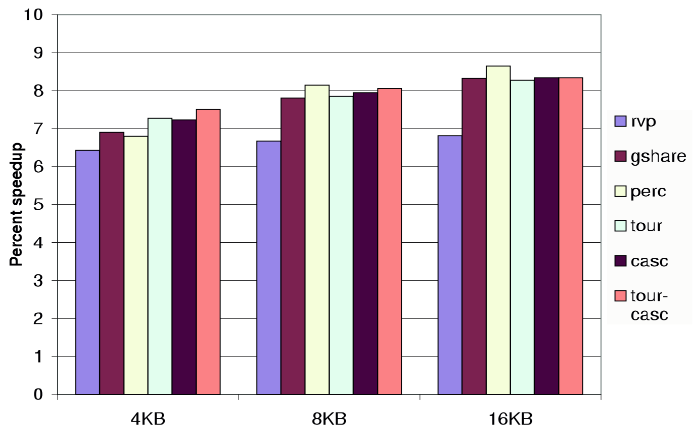

Figure 4. Speedup for the various predictors for the average of the benchmarks. Results are shown for the rvp, gshare, perceptron, tournament, cascade, and tournament-cascade predictors.

but does not increase any further beyond a history length of 32. When increasing history from 4 bits to 16, accuracy increases from 94.2% to 97.7%. When there are 60 bits of global history, the prediction accuracy increases to 98.6%.

Not shown in this work is the effect that history length has on individual benchmarks. Some benchmarks greatly benefit from an increased history length while others do not. We find that `mcf` reaches near perfect prediction with only 12 global history inputs and does not improve in performance with increasing history. The `mgrid` benchmark continues to improve in performance even beyond a global history of 60. `Galgel` performs poorly at low history lengths (30.4% and 15.1% slowdown at 4 and 8 bits respectively), but achieves speedup beyond 16 bits (5.1%) and continues improving up through 60 bits (15.2%).

5.3. Hybrid Predictors

The design of the perceptron predictor makes it well suited for global pattern detection, whereas the *rvp* predictor targets exploiting local history. In this section, we analyze the impact of creating hybrid predictors using an *rvp* predictor combined with a *perceptron* predictor, as previously described.

We look at three different techniques of combining an *rvp* with a *perceptron* predictor to form a hybrid predictor. The first is a tournament style selection mechanism similar to the mechanism described by McFarling [7]. We use an additional 1K entry selection table of 2-bit saturating counters indexed by the lower 10 bits of the program counter. A counter in the selection table is incremented if the perceptron predicts a redundant register value and is correct. The counter in the selection table is decremented if the *rvp* predictor predicts a redundant value and it is correct. If both predictors are correct, the counter value does not change. If the counter value for an instruction is 2 or 3, then the prediction from the perceptron is used. If the counter value is 0 or 1, then the prediction from the *rvp* predictor is used. In this hybrid configuration, half of the storage space is used by each of the predictors, although the size of the rvp predictor is reduced by the size of the tournament selector table. We refer to this predictor configuration as *tour*.

The other hybridization technique studied involves cascading the *rvp* predictor with a *perceptron* predictor. Michaud and Seznec [8] studied this cascading technique for perceptrons used for branch prediction. The prediction bit from the *rvp* predictor is used as additional input to the perceptron. One less bit from the global history register is used to compute the perceptron weight and the *rvp* prediction bit is used instead. We refer to this predictor configuration as *casc*.

We also look at a hybrid predictor that combines the tournament selector mechanism and the cascaded prediction scheme. The same prediction from the *rvp* is used as an input to the perceptron and also as a prediction itself. A 1K entry table of saturating counters is used to select between the cascaded predictor and the *rvp* predictor. We will refer to this predictor configuration as *tour-casc*.

For comparison, we also study a *gshare* register value predictor, which in configuration is similar to the gshare branch predictor [7]. For our experiments, we use a table of 3-bit saturating counters. The index into the table is the result of the XOR of the lower bits of the instruction address and the global history register. If the instruction is verified

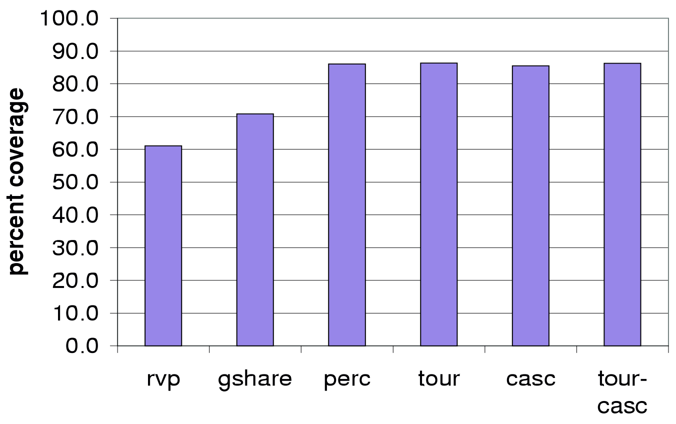

Figure 5. Predictor coverage for the various predictor configurations. Coverage is the ratio of correctly identified redundant instructions to the number of actual redundant instructions. Data is shown for 8KB predictors.

to be redundant, then its counter is incremented; otherwise, the counter value is reset. If the counter value exceeds a threshold, then a prediction is made. The same threshold is used for the *gshare* predictor as for *rvp*.

Figure 4 shows the performance results of the various predictors for the average of the benchmarks. The speedup results are shown for *rvp*, *gshare*, *perceptron*, *tour*, *casc*, and *tour-casc* predictors in 4KB, 8KB, and 16KB total predictor sizes.

At all storage sizes, *gshare*, *perceptron*, and the hybrid predictors outperform *rvp*. With a storage budget of 8KB, *perceptron* performs the best, achieving a higher speedup than any of the hybrids. The same is true at a 16KB storage budget. At 4KB, the performance of *perceptron* is slightly lower than that of *gshare*. We find that the accuracy of *gshare* is slightly higher (97.6% for *gshare* versus 97.5%). The best performing predictor at a 4KB budget, is *tour-casc* with a speedup of 7.5%.

Upon analyzing the behaviors of the various predictors, we find that although the accuracy of the hybrid predictors is slightly better that a single component *rvp* or *gshare*, it is in fact the coverage of the redundant instructions which separates the predictors. Figure 5 shows the coverage for the various predictors with a size of 8KB. Upon comparing the coverage of *rvp* or *gshare*, we find that any predictor with a perceptron component correctly identifies many more redundant instructions. This is due to the ability of the perceptron to identify redundancy that spans longer history lengths.

6. Future Work

In future work, we intend to look at limiting the scope of the predictions made by the perceptron predictor. That is, we would like to limit the number or types of instructions predicted. The current predictor targets a number of different instruction which produce register values. We would like to look at targeting load instructions only or floating-point instructions only. Some phase-based behavior may also provide some guidance into which types of instructions to target at different times in program execution.

In addition to, or in place of, the history of redundancy we consider in this paper, we would also like to consider a value predictor perceptron that uses the branch history. Since processors already keep track of branch history, using this information may require less hardware than a perceptron based on value redundancy. Branch history may yield other insights that allow further success at value prediction as well.

7. Conclusions

Perceptrons are a simple model of neuron behavior. Recently, perceptrons have been proposed for use in branch predictors and have demonstrated good performance. In this work, we study the impact that a perceptron-based predictor has on register value prediction, a type of value locality where an instruction produces the same value that already exists in the instruction destination register. When an instruction can be correctly predicted to exhibit this locality, the dependents of that instruction can be issued before the predicted instruction is done executing.

We demonstrate that for a given size predictor, a perceptron predictor performs better than a saturating counter based register value predictor. On the average of the benchmarks studied, an 8KB perceptron predictor achieves a speedup of 8.1%, with speedups of as much as 45.2% on one benchmark.

We study using global history lengths of up to 60 bits and find that using a history of 40 bits allows a perceptron predictor to achieve 90% of the performance of a predictor with a history of 60 bits. We find that some benchmarks can benefit from even longer histories.

Finally, we study hybrid predictors which combine a perceptron predictor with a saturating counter based predictor. With a 4KB hardware budget, a speedup of 7.5% is achieved using the tournament-cascade hybrid.

8. Acknowledgments

We would like to thank the UC San Diego Processor Architecture and Compilation lab and Intel for providing the simulation environment for this work. We would also like to thank our anonymous reviewers for their valuable feedback.

References

[1] S. Balakrishnan and G. S. Sohi. Exploiting value locality in physical register files. In *36th Annual International Symposium on Microarchitecture, 2003*, Dec 2003.

[2] M. Burtscher and B. Zorn. Hybrid load-value predictors. *IEEE Transactions on Computer*, 51(7), July 2002.

[3] M. Burtscher and B. G. Zorn. Prediction outcome history-based confidence estimation for load value prediction. *Journal of Instruction-Level Parallelism*, 1, May 1999.

[4] D. A. Jiménez. Reconsidering complex branch predictors. In *the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, Feb 2003.

[5] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *Seventh International Symposium on High Performance Computer Architecture (HPCA-7)*, Jan 2001.

[6] D. A. Jiménez and C. Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems*, 20(4), Nov 2002.

[7] S. McFarling. Combining branch predictors. Technical Report TN-36, DEC-WRL, June 1993.

[8] P. Michaud and A. Seznec. A comprehensive study of dynamic global history branch prediction. Technical Report 1406, IRISA, June 2001.

[9] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[10] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[11] D. Tullsen and J. Seng. Storageless value prediction using prior register values. In *26th Annual International Symposium on Computer Architecture*, pages 270–279, May 1999.