Perceptron-based Coherence Predictors

Devyani Ghosh
University of Utah
devyani@cs.utah.edu

John B. Carter
University of Utah
retrac@cs.utah.edu

Hal Daumé III
University of Utah
me@hal3.name

Abstract

Coherence misses in shared-memory multiprocessors account for a substantial fraction of execution time in many important workloads. Just as branch predictors reduce the performance impact of branches, coherence predictors can reduce the performance impact of coherence misses. Two-level pattern-based coherence predictors have offered a general prediction method to trigger appropriate coherence actions. This paper presents the design and evaluation of a perceptron-based coherence predictor that extends a conventional directory-based write-invalidate protocol to predict when to push updates to remote nodes. When predicted correctly, the update eliminates a coherence miss on the remote node. We also present a simple mechanism for predicting to which nodes we should push updates.

We evaluate our perceptron-based update predictor on a variety of SPLASH-2 and PARSEC benchmarks. Simulation indicates that the update predictor eliminates an average of 30% of coherence misses. Our simple consumer prediction mechanism sent very few useless updates – on average 87% of updates were consumed (eliminated misses).

1 Introduction

Coherence activities remain an increasingly challenging problem for traditional shared memory processors and are likely to become a major performance bottleneck in future chip multiprocessors (CMPs). With increasing cache sizes and number of cores in future CMPs, coherence misses [7] will account for a larger fraction of all cache misses. As communication latencies increase, coherence misses may take hundreds of cycles, which can severely degrade performance.

A common approach to reduce memory latencies in multiprocessors is to use private or hierarchical caches in conjunction with a directory-based cache coherence protocol. Write-invalidate protocols are efficient for situations where data is used exclusively by a single processor or mostly read-shared, but are inefficient for other sharing patterns, like producer-consumer, wide-sharing, and migratory sharing [6, 19, 37, 39]. In contrast, pure write-update protocols are inappropriate because they tend to generate excessive network traffic by sending useless updates, updates that are unlikely to be consumed before the cache line is modified again [5, 12].

Previous studies have argued for hybrid coherence protocols that employ a mix of invalidate and update messages [38]. Such techniques typically rely on complex adaptive coherence protocols that directly capture sharing patterns in protocol states [20, 24] and are limited to learning one sharing pattern per-memory-block at a time. An alternative to building complex hybrid coherence protocols is to employ a pattern-based coherence predictor. Mukherjee and Hill [33] adapted the classical two-level PAp branch prediction scheme [41] to learn and predict coherence activities for a memory block. [22] improved upon the first generation of pattern-based predictors by providing better accuracy and implementation over general message predictors. Recent efforts have focused primarily on hardware optimizations that reduce the high storage overheads of such predictors [3, 34, 9]. We take a different approach and focus on using machine learning techniques to improve predictor accuracy.

Our main contribution is the design and evaluation of an online machine learning technique using *perceptrons* [36] to reduce coherence protocol overhead in a CMP. We attach a small perceptron to the directory cache entries of shared blocks and train them using the stream of memory accesses to that block. Over time, our predictor learns to identify *when* a write update should be sent. After a producer is done writing to a cache block, the predictor can recommend that we speculatively forward new data to its likely consumers and downgrade the coherence state of the owner from *Modified* to *Shared*.

We use a simple, yet very effective, heuristic to predict the likely consumers of each update. Our coherence predictor is not tied to a particular coherence optimization, e.g., data forwarding in our case, and could be used to enable other optimizations instead.

Akin to the way the outcome of a branch predictor is a notion of whether or not a particular branch will be taken or not taken, the outcome of our coherence predictor is

whether or not a particular write to a cache line will be *consumed* remotely before the line is modified again. Put another way, we predict whether or not a particular write is the last by a given writer before some remote node reads the cache line. We refer to a write operation as a *last write* if it is the last time a processor (producer) writes to a cache line before the line is read by another processor (consumer). Much like Jiménez and Lin [17] advocate the use of perceptrons for making fast accurate branch predictions, we use perceptrons to predict whether a write is a *last* write and is going to be *consumed*. Predicting who the consumer(s) of this data are going to be is an orthogonal question that has been addressed previously [6, 16, 21, 35], and in a more sophisticated manner in [25]. Our focus is on the prediction of *when* to perform a push, and could use any of the above consumer set prediction mechanisms.

Our novel perceptron-based coherence predictor is simple, yet highly accurate. We evaluated a simple single-layer unbiased perceptron mechanism on a mix of applications from the SPLASH-2 [40] and PARSEC [2] benchmark suites. Our predictor achieved a prediction *accuracy* of over 99%. Most predictions were true negatives (i.e., "do not push"), but we correctly identified sufficient opportunities to push data to eliminate an average of 30% (and up to 69%) of coherence misses, despite being very conservative. The prediction *precision* was given by the percentage of updates that are consumed by remote nodes prior to an intervening write, i.e., the percentage of updates that eliminate a coherence miss compared to a traditional write-invalidate protocol. Our predictor had an average precision of 87% (and at best 99%). The prediction *sensitivity* [13] demonstrated that on average 73% (and at best 97%) of all available update opportunities were successfully identified.

Like pattern-based predictors, our perceptron-based predictor is able to learn from and adapt to an application's runtime sharing patterns, able to capture multiple distinct sharing patterns for a memory block, and does not require modifications of the base coherence protocol. Although our design evaluates only perceptron-based predictors, we believe other online machine learning techniques [11, 14, 26] are also worth exploring.

The rest of the paper is organized as follows. Section 2 provides the background needed to understand our predictor. Section 3 explains how a perceptron can be used in coherence prediction and gives the details of our implementation. Section 4 presents our evaluation methodology and results. Section 5 discusses related work. We conclude and suggest future work in Section 6.

2 Perceptron Learning Model

In this section we review how perceptron learning algorithms work. Perceptrons [36] have been widely studied by the machine learning community and are capable of recognizing many classes of patterns. We selected perceptron for our study because they are simple to implement, have no tunable parameters (learning rate, norm, etc.) and tend to work well empirically. We now outline how perceptron predictors are trained using an error correction mechanism.

A perceptron is represented by a vector whose elements are the current *weights* associated with specific *features* that are used to predict a particular outcome. For our purposes, weights are signed integers and inputs are binary values. The output y of a perceptron is binary and is given by the sign of the dot product of a weights vector $w_{1..n}$ and an input features vector $x_{1..n}$:

$$y = sign\left(\sum_i x_i w_i\right) \quad (1)$$

If the value of y is positive, the perceptron is said to predict a positive outcome. A feedback-based learning algorithm is used to train a perceptron to identify positive correlations between its inputs and the desired output. The learning algorithm continually adjusts the weights vector based on whether (and the degree to which) the perceptron's previous prediction was correct. It is well known that the perceptron algorithm will converge after a finite number of examples to an optimal linear classifier (of a form given by Eq.(1)), should one exist. By tracking correlations between the desired prediction and the inputs, perceptrons can isolate the relevant portions of its input from the irrelevant portions.

The following pseudocode illustrates a simple unbiased prediction algorithm using additive updates to adjust weights:

Initialize w to zero
while $(true)$ **do**
  Obtain input x
  Predict $y = sign(\sum_{i=1}^{n} w_i \cdot x_i)$
  if y is incorrect **then**
    Do Update: $w \leftarrow w - y \cdot x$
  end if
end while

3 Design and Implementation

Recall that the goal of our coherence predictor is to decide at the time of each write whether that write is the last one by a producer and if it is likely to be consumed by another node. If so, we perform a write-update by downgrading the writer to *shared* mode and pushing the dirty data to a set of predicted consumers. We predict consumers using a simple heuristic described in Section 3.4.

Our perceptron model is designed to identify correlations between particular write outcomes tracked in a *block access*

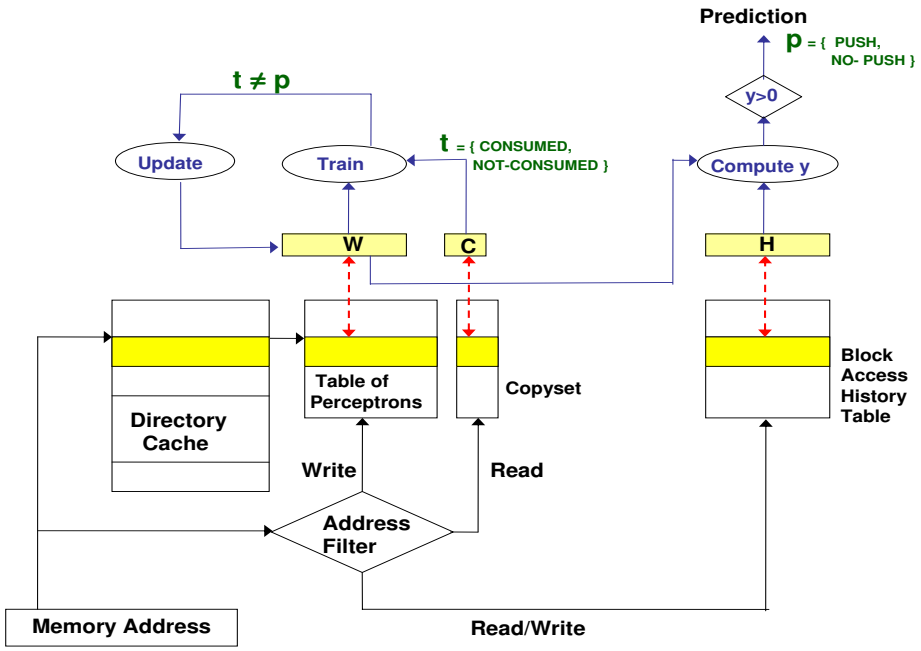

Figure 1. Perceptron Predictor Block Diagram

history table and the current write. These correlations are represented by weights. The larger a weight, the stronger the correlation of that feature with a positive (update) outcome, and the more likely that a particular memory access appearing in the history table contributes to the prediction of current write.

The number and size of perceptrons that we can employ is dictated by a hardware budget. Existing coherence predictors require fairly large history tables, which are likely to limit their use in commercial systems. However, a majority of coherence misses have been found to be confined to a fairly small part of an application's memory footprint [34]. Since we need to make predictions for cache lines that are subject to coherence misses, we tracked perceptron weights only for cache lines that have ever been invalidated (marked by an extra bit), referred to as *coherence blocks* [34]. In particular, we introduce a table of perceptrons that tracks the weights associated with coherence blocks, and track a small amount of their access history. This enables us to train the perceptrons and predict to which nodes to push updates.

Figure 1 presents a high-level block diagram for the resulting perceptron predictor. We assume a centralized prediction scheme wherein a single global predictor is accessed by all processors in the system. It maintains separate set of weights and memory access history for each coherence block. All tables are initialized to zero. We also assume the presence of an *address filter* to identify memory references to coherence blocks. Our implementation simply filters memory blocks that were ever invalidated in past because of a remote write operation.

In the remainder of this section, we delve into some of the implementation details of our design.

3.1 Feature Set (Input Set)

A *feature* is an observable value that can be used to make a prediction. For example, features can include data such as the last writer of a cache line or whether the last access was a read or write. The number of features that we track per-line is a design parameter that we can adjust, similar to how the history length tracked can be adjusted for perceptron-based branch predictors [17].

To minimize space requirements, we characterize a data memory access by two simple attributes: (i) the requesting processor ID and (ii) whether the access was a READ or a WRITE. Perceptrons work best when features can be encoded as a vector of boolean values, so we encode features as follows. For a system with n processors, we use a bitmap of length $n+2$ to represent a memory access. Each processor is uniquely assigned a single bit from the first n bits - which is set to 1 if it is the requesting processor. The $(n+1)^{th}$ bit is set to 1 to indicate that the operation was a *READ*, while the $(n+2)^{th}$ bit is set to 1 to indicate that it was a *WRITE*. Since we considered a naive unbiased perceptron, a representation with separate R/W bits was convenient. Note that, only one of first n bits can be a 1, and either of $(n+1)^{th}$ or $(n+2)^{th}$ bit can be 1; all remaining bits will be 0. For example, in a 4-core system ($P1, P2, P3, P4$), a

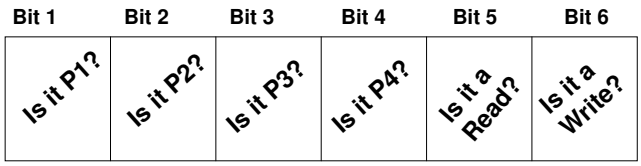

Figure 2. Format of input *feature* set corresponding to a 4-processor system. The bit vector is representative of a memory access

read by processor $P2$ will be represented as 010010 while a write by processor $P1$ will have a representation of 100001. Figure 2 illustrates our feature set (input format) for a 4-core system. Although our calculation needs such a representation, information storage does not. We describe at the end of this section 3.2 how to store block access information economically by using access *signatures*.

3.2 Storage Estimation

Having defined the encoding for a memory access, we next must decide how much history to track. In general, a perceptron-based predictor considers the last h operations when making a prediction. h is referred to as the *history length*, and like the feature set itself is a tunable design parameter – a larger *history length* typically improves accuracy but incurs greater storage and computation overhead. To explore how well we could do while minimizing space overhead, we model a very small history length of 2 throughout our experiments.

To determine whether or not our last prediction was correct, we need to track what that prediction was (PUSH or NO-PUSH, 1-bit) and whether or not there has been a remote read to that cache line since the PUSH was performed (a bitmap of n-bits consumer set). Thus, the amount of state required to make predictions and update the weight tables is $h(n+2)+n+1$ binary values. For our specific design, where n varies from 4 to 16 and h is fixed at 2, we track 17-53 bits of history information per cache block in the *block access history table*. Further, since every feature has its corresponding weight, there are $h(n+2)$ weights per block. Currently we employ integer weights, but we believe that similar results could be obtained with 4-bit weights, thereby decreasing storage overhead. In general, if a weight is b bits long, then $bh(n+2)$ bits are needed for representing an entry in the weight table. Hence, the total amount of storage required by our prediction scheme is $h(n+2)(b+1)+n+1$ bits per memory block.

We can reduce the amount of storage required by the history table by using block access *signatures*, whereby an access is simply characterized by its type (R/W, 1-bit) and a processor-id ($\log_2 n$ bits). Thus, it suffices to store $h(\log_2 n+1)+n+1$ bits in the block access history table. For our specific design parameters, this works out to 11-27 bits.

3.3 Update-Predict Algorithm

We realized that at the time of a memory write, we are able to both determine whether the last prediction was correct and make a prediction about whether this write should perform a write-update. Consequently, no training is needed at the time of a memory read. This observation highly simplified our perceptron design and allowed us to integrate our coherence predictor with a traditional coherence protocol.

To make this work, we had to make a small modification to a generic perceptron algorithm that did not affect its correctness. Specifically, in our design, the update step, where the weights associated with a perceptron are updated based on the accuracy of the last prediction, is performed before the prediction step.

As described in Section 2, each time the perceptron makes a prediction, we need to update its weights based on whether or not the prediction was correct.

To determine whether our last prediction was correct, we track what that prediction was (PUSH or NO-PUSH) and the set of nodes to which we pushed (if any). Let S_0 (consumer copyset) and S_1 (current copyset) denote the complete set of nodes that have a shared copy at the time of the previous and current write operation, respectively. If we predicted PUSH, then our prediction is accurate iff at least one node in S_1 is also in S_0, not including the node that performed previous write. If we predicted NO-PUSH, then our prediction is accurate iff no node in S_1 is in S_0. Otherwise, our prediction was incorrect.

We now describe the operation of our perceptron-based coherence predictor. On a write access to a coherence block, the following steps are taken:

1. *Fetch*: The corresponding perceptron weights W, copyset C and block access history H are fetched.
2. *Determine Truth*: We determine whether any of the nodes that had a shared copy before the *last* write (S_0) was also a sharer before the *current* write operation (S_1) (not including the last writer) If so, the truth (correct outcome) was that we should have pushed after the last write ($t=1$), otherwise the truth was not to push ($t=-1$).
3. *Update Weights* (based on previous prediction): We compare the correct outcome with the prediction made at the time of the last write (p). The training algorithm updates the weights in W as described in Section 2. If we predicted correctly, we do nothing. If we predicted incorrectly, we increase weights if truth was positive ($W \leftarrow W+H$) or decrease weights if truth was negative ($W \leftarrow W-H$).

4. *Predict*: y (prediction for this write) is computed using the dot product of W and block access history H, ($y = sign(\sum_{i=1}^{n} W_i \cdot H_i)$). If y is positive, we predict PUSH, otherwise we predict NO-PUSH.

5. *Manage History* : H is updated with the current write access information by shifting the oldest feature set out of H and adding the current feature set, i.e., the bit vector that encodes whether this was a read or write and which processor is performing the operation. Further, copyset S_0 is set to S_1, and S_1 is reset to null.

On a read access to a coherence block, the following steps are taken:

1. The reading node is added to the current copyset (S_1)

2. H is updated with this read access information, i.e., we shift the oldest feature set out of H and shift in a bit vector that encodes this read operation.

3.4 Tracking Consumers

We employ an extremely simple mechanism to predict likely consumers of a particular write – we just use the set of most recent readers as tracked in the *copyset*. Copyset is a bitmap representing the readers since last write to that block, and are reset after each write operation. Subsequent readers continue to add themselves to this list until the next write operation, at which point this new copyset is used to predict the consumers, if a push is predicted. For example, let S_0 and S_1 be the set of nodes that have a shared copy at the time of the previous and current write, respectively. On a PUSH prediction, we send updates to the set of nodes given by $(S_1 \cap S_0) \cup (S_1 - S_0)$. Despite its simplicity, we found that this mechanism does a good job of predicting likely consumers and does not cause a high number of useless updates.

4 Evaluation

In this section we discuss our evaluation methodology and then present our results.

4.1 Simulation Environment

We evaluate our system using the Virtutech Simics full system execution-driven simulator [30]. We model a CMP architecture consisting of 4, 8 or 16 in-order, single-issue processors. Each core has a private split L1 cache, with a 64KB I-cache and a 32KB D-cache, plus a single large shared 8MB L2. We chose this configuration to minimize conflict or capacity misses and let us focus on coherence activities. The block size is 64 bytes, the L1 caches are 4-way set-associative, and the unified L2 cache is 8-way set-associative. We assume a simple MSI directory-based cache coherence protocol among the L1 caches. Since our primary focus is reducing the coherence miss rate, this simple processor model is sufficient to evaluate our predictor's fundamental performance.

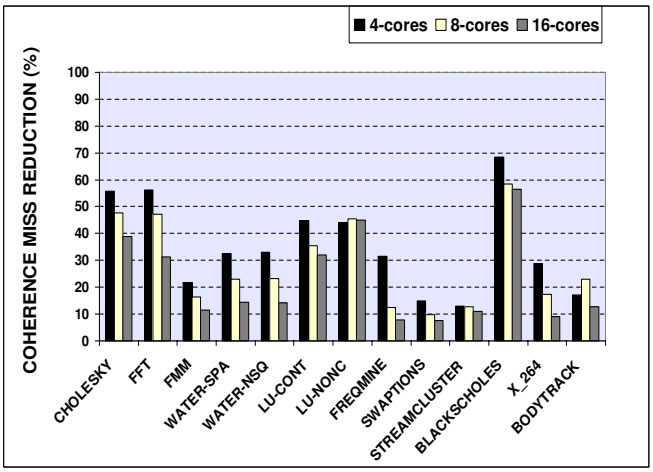

Figure 3. Coherence Miss Reduction Percentage in First Level Caches

All experiments were conducted with a history depth of 2. This means that while making a prediction, only the previous two accesses to a block (reads or writes) are considered. We anticipate that we could achieve even better results if we increased the history depth, so the results presented here are a conservative estimate of the potential of perceptron-based coherence predictors.

We considered 13 applications from SPLASH-2 [40] and PARSEC [2] benchmark suites. We used the default input sets for SPLASH-2 and sim-medium configuration for PARSEC applications. SPLASH-2 applications were run to completion starting from the beginning of their parallel sections, and the PARSEC applications were run for a billion instructions starting from their region of interest.

4.2 Results

Our perceptron predictor achieved a prediction accuracy of roughly 99%. This high prediction accuracy is primarily due to the large number of true-negative (NO-PUSH) predictions that arise because the prediction is performed for each access to the private L1 caches. Non-shared accesses make up the vast majority of accesses to the L1 cache, so the predictor had a little scope for making a positive prediction. For the remaining (shared) accesses, the predictor's accuracy ranged from 50% to well over 90%, and in particular the true positive (useful push) prediction rate far exceeded the false positive (useless push) rate.

We compare the results of a system extended to include our percepton-based coherence predictor against a baseline system that employs a conventional directory-based coher-

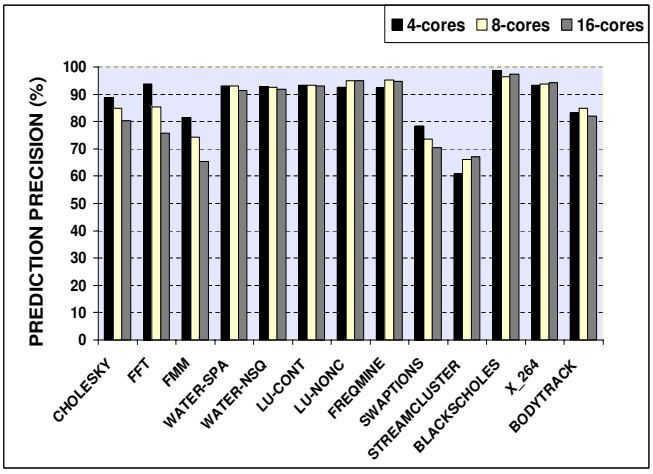

Figure 4. Prediction Precision: Percentage of speculative updates that were consumed

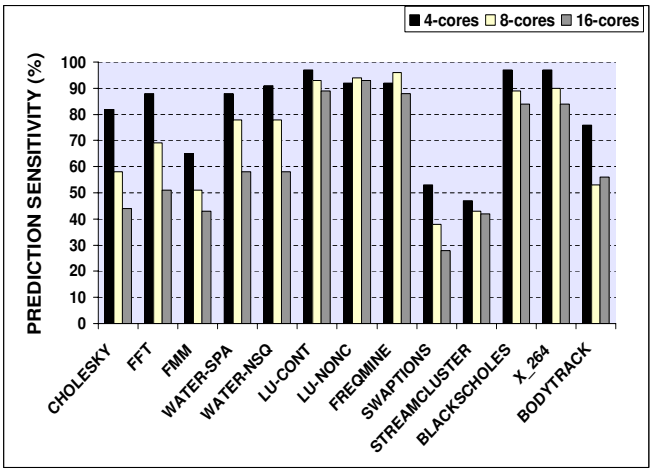

Figure 5. Prediction Sensitivity: Percentage of update opportunities that were identified

ence mechanism. Figure 3 presents the net L1 cache coherence miss reduction achieved by our predictor, for 4-, 8-, and 16-core configurations. It should be noted that these results include cold start effects, so they are a bit pessimistic. Nevertheless, our coherence predictor eliminated on average 34% , 30%, and 23% (and up to 69%, 59%, and 57%) of coherence misses on 4-, 8-, and 16-processor systems, respectively. Blackscholes benefitted the most among all benchmarks, while Swaptions benefitted the least. In no case did the coherence miss rate increase, but the value of coherence prediction varied considerably across the set of applications.

Because a positive prediction generates updates, it is important to verify that our predictor does not cause a flood of useless updates, which doomed the original work on write-update protocols. To assess our predictor's ability to only push updates that will be usefully consumed, we use the *precision* metric. Our predictor's *precision* is given by the ratio of number of pushes consumed over the total number of pushes that were speculatively sent.

Consider a situation where the coherence predictor recommends pushing a cache line to six nodes. If three of these nodes read the data before the cache line is next modified, the prediction precision is 50%; if five of the nodes read the data prior to when it is next modified, the precision is 87%. Higher the precision, the better job the predictor has done at avoiding useless update messages. Every consumed push saves a cache miss, so our goal is to balance the predictor design to identify as many opportunities as possible to push data, without flooding the network with useless updates.

Figure 4 plots the precision achieved by our predictor combined with our simple consumer predictor heuristic. The average precision is 87%, and ranges between 61-99%. Thus, on average five of our six speculative updates eliminate remote cache misses, and only one in six are useless.

To further evaluate the performance of our predictor, we use the *Sensitivity* metric. Also commonly known as *Recall*, it demonstrates the ability of a prediction model to select positive instances from data. Sensitivity measures the percentage of positive predictions versus all positive opportunities and is defined by the ratio of true-positive to the sum of true-positive and false-negative predictions. Figure 5 plots the sensitivity and shows that our predictor correctly identifies on average 73%, and at best 97%, of write-update opportunities available.

Combining the results from Figures 3, 4 and 5, we conclude that our perceptron-based coherence predictor is *sensitive* enough not to miss too many opportunities for sending useful updates (Fig. 5). Its high prediction *precision* (Fig. 4) ascertains that most speculative updates are fruitful, which results in a significant reduction in the number of coherence misses eliminated (Fig. 3).

5 Related Work

Speculative data forwarding in a multiprocessor was first proposed by Lebeck and Wood [24]. Their technique, *dynamic self-invalidation*, triggers the invalidation of shared data blocks, identified via coherence protocol hints, at annotated critical section boundaries. *Last touch prediction* (LTP) proposed by Lai et al [23] associated invalidation events with the sequence of instructions accessing a cache block prior to its invalidation. Hu et al [15] investigated timekeeping approaches for predicting memory sys-

tem events that are straightforward to adapt for coherence prediction.

Write-update and hybrid update-invalidate protocols have been extensively studied [1, 31]. Producer-initiated communication and prefetching are two commonly used techniques to hide long miss latencies [4]. Predicting the subsequent sharers of a newly produced value, typically referred to as *consumer set prediction*, was first proposed by Mukherjee and Hill [32]. They took the first step towards using general prediction to accelerate coherence protocols by developing the *Cosmos* coherence message predictor. Cosmos's design was inspired by Yeh and Patt's two-level PAp branch predictor [41]. Branch prediction has made tremendous progress since then. Current designs based on neural learning are known to be the most accurate predictors [18, 28]. Jiménez and Lin [17] were the first to adapt perceptrons to branch predictors. Our work is related in the sense that we employ perceptrons to make accurate predictions, but we tackle a very different problem.

The design of general coherence predictors was extensively studied after Mukherjee and Hill's initial work [32]. Lai and Falsafi propose the *memory sharing predictor* (MSP) and the more efficient *vector memory sharing predictor* (VMSP) [22]. MSP is a generalization of the *Cosmos* predictor [32], which eliminated prediction of acknowledgement messages and offered better implementation and accuracy. However, all of these predictors are fundamentally two-level pattern-based predictors with saturating counters, which in our opinion can be augmented or replaced by neural networks to achieve better accuracy with lower space overheads.

Levanthal and Franklin [25] recently proposed using perceptrons to make predictions within the framework of a coherence protocol. However, they use perceptrons to perform consumer set prediction, whereas our work addresses the problem of predicting *when* to perform speculative updates, a problem they specifically do not address. The two mechanisms are orthogonal, and likely very complimentary.

6 Conclusions and Future Work

In this paper, we have introduced a new class of coherence predictors that uses perceptrons to identify opportunities for sending useful write-updates. We have shown how a perceptron-based coherence predictor can be added to an existing directory-based cache coherent system. Our results demonstrate that perceptrons can eliminate a significant number (on average 30%) of coherence misses on a wide range of benchmarks. When coupled with a simple consumer set prediction heuristic, over 87% of speculative updates generated by our predictor are usefully consumed.

There are many ways to extend and improve our work. One weakness of perceptrons in general is their inability to learn linearly inseparable functions. Despite this weakness, perceptrons have been an attractive choice for branch predictors, because most programs possess linearly separable branches [17]. We would like to investigate whether such relationships can be observed in the case of write-update predictions, or explain why and when a perceptron-based predictor is likely to fail.

The performance and accuracy of our simple predictor, despite employing a very simple consumer prediction heuristic (push to the last set of readers), a very small history window (the last two accesses), and a small set of features (the processor ID and whether the access was a read or write), demonstrate the effectiveness of perceptron-based coherence predictors.

A major focus of our ongoing work is to perform a detailed sensitivity analysis to changes in: (i) the prediction threshold levels (i.e., the cutoff in perceptron values between a positive and negative prediction), (ii) history lengths, (ii) input feature set size, and (iv) the number of bits we use to represent perceptron weights. We believe that we can tune the perceptron-based coherence predictor to further improve prediction accuracy while maintaining a small perceptron storage overhead. Recent studies on perceptron branch predictors have proposed many fast, power- and resource-efficient implementation strategies [10, 27, 29]. We would also like to study the potential of *cost sensitive classification* [8, 42] in discouraging false positive predictions.

Finally, the machine learning field is replete with prediction mechanisms that have long been used to develop classifiers that learn patterns from data and group them into classes. We have explored the use of only one online learning scheme, perceptrons. There are other learning algorithms [11, 14, 26] that may also be useful, and are waiting to be evaluated.

References

[1] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, 1997.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Princeton University Technical Report TR-811-08*, January 2008.

[3] I. Burcea, S. Somogyi, A. Moshovos, and B. Falsafi. Predictor virtualization. In *Proceedings of the 13th International conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[4] G. Byrd and M. Flynn. Producer-consumer communication in distributed shared memory multiprocessors. *Proceedings of the IEEE*, 87(3), March 1999.

[5] J. Carter, J. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, Aug. 1995.

[6] L. Cheng, J. Carter, and D. Dai. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *International Symposium on High-Performance Computer Architecture*, 2007.

[7] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrm. The detection and elimination of useless misses in multiprocessors. In *Proceedings of the 20th annual International Symposium on Computer Architecture*, 1993.

[8] C. Elkan. The foundations of cost-sensitive learning. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

[9] M. Ferdman and B. Falsafi. Last-touch correlated data streaming. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2007.

[10] H. Gao and H. Zhou. Adaptive information processing: An effective way to improve perceptron branch predictors. In *Journal of Instruction-Level Parallelism*, 2005.

[11] C. Gentile. Easy learning theory for highly scalable algorithms. In *Tutorial at the International Conference on Machine Learning (ICML)*, 2005.

[12] H. Grahn, P. Stenström, and M. Dubois. Implementation and evaluation of update-based cache protocols under relaxed memory consistency models. *Future Generation Computer Systems*, 11(3):247–271, June 1995.

[13] D. Grunwald, A. Klauser, S. Manne, and A. R. Pleszkun. Confidence estimation for speculation control. In *Proceedings of the 25th annual International Symposium on Computer Architecture*, 1998.

[14] E. Harrington, R. Herbrich, J. Kivinen, J. Platt, and R. Williamson. Online Bayes point machines. In *Proceedings of the Seventh Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2003.

[15] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *Proceedings of the 29th annual International Symposium on Computer Architecture*, 2002.

[16] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence decoupling: Making use of incoherence. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[17] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[18] D. A. Jiménez and C. Lin. Neural methods for dynamic branch prediction. In *ACM Transactions on Computer Systems (TOCS)*, 2002.

[19] S. Kaxiras. Identification and optimization of sharing patterns for high-performance scalable shared-memory. In *Ph.D. Thesis, University of Wisconsin-Madison*, 1998.

[20] S. Kaxiras and J. R. Goodman. Improving CC-NUMA performance using instruction-based prediction. In *International Symposium on High-Performance Computer Architecture*, 1999.

[21] S. Kaxiras and C. Young. Coherence communication prediction in shared-memory multiprocessors. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture,*, 2000.

[22] A.-C. Lai and B. Falsafi. Memory sharing predictor: The key to a speculative coherent DSM. In *Annual International Symposium on Computer Architecture*, 1999.

[23] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[24] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. In *Annual International Symposium on Computer Architecture*, 1995.

[25] S. Leventhal and M. Franklin. Perceptron based consumer prediction in shared-memory multiprocessors. In *Proceedings of the 24th International Conference on Computer Design (ICCD)*, October 2006.

[26] N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2:285–318, 1988.

[27] G. H. Loh. A simple divide-and-conquer approach for neural-class branch prediction. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques PACT*, 2005.

[28] G. H. Loh and D. S. Henry. Predicting conditional branches with fusion-based hybrid predictors. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, 2002.

[29] G. H. Loh and D. A. Jiménez. Reducing the power and complexity of path-based neural branch prediction. In *Proceedings of the 5th Workshop on Complexity Effective Design (WCED5)*, 2005.

[30] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer Society*, 35(2):50–58, February 2002.

[31] M. R. Marty and M. D. Hill. Coherence ordering for ring-based chip multiprocessors. 2006.

[32] S. S. Mukherjee and M. D. Hill. Using prediction to accelerate coherence protocols. In *Annual International Symposium on Computer Architecture*, 1998.

[33] S. S. Mukherjee, S. D. Sharma, M. D. Hill, J. R. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed-memory machines. In *Proceedings of the 5th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1995.

[34] J. Nilsson, A. Landin, and P. Stenstrom. The coherence predictor cache: A resource-efficient and accurate coherence prediction infrastructure. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003.

[35] D. K. Poulsen and P. C. Yew. Data prefetching and data forwarding in shared memory multiprocessors. In *Proceedings of the 1994 International Conference on Parallel Processing, volume II*, 1994.

[36] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958. Reprinted in *Neurocomputing* (MIT Press, 1998).

[37] P. Stenström, M. Brorsson, and L. Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[38] J. Veenstra and R. Fowler. A performance evaluation of optimal hybrid cache coherency protocols. In *Proceedings of the 5th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 149–160, Sept. 1992.

[39] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, C. Gniady, A. Ailamaki, and B. Falsafi. Store-ordered streaming of shared memory. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[40] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 1992.

[41] T.-Y. Yeh and Y. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[42] B. Zadrozny, J. Langford, and N. Abe. Cost sensitive learning by cost-proportionate example weighting. In *Proceedings of the Third IEEE International Conference on Data Mining*, 2003.