Perceptron Based Consumer Prediction in Shared-Memory Multiprocessors

Sean Leventhal and Manoj Franklin
School of Electrical and Computer Engineering
University of Maryland at College Park
{sleventh, manoj}@glue.umd.edu

***Abstract*— Recent research has shown that forwarding speculative data to other processors before it is requested can improve the performance of multiprocessor systems. The most recent work in speculative data forwarding places all of the processors on a single bus, allowing the data to be forwarded to all of the processors at the same cost as any subset of the processors. Modern multiprocessors however often employ more complex switching networks in which broadcast is expensive. Accurately predicting the consumers of data can be challenging, especially in the case of programs with many shared data structures.**

Past consumer predictors rely on simple prediction mechanisms, a single table lookup followed by a static mapping of the table values onto a prediction. We make two main contributions in this paper. First, we show how to reduce the design space of consumer predictors to a set of interesting predictors, and how previous consumer predictors can be tuned to expand the range of available performance. Second, we propose a perceptron consumer predictor that dynamically adapts its reaction to the system behavior, and uses more history information than previous consumer predictors. This predictor outperforms the previous predictors by 21% while using only 1KByte more storage than previous predictors.

I. Introduction

The increase in transistor count and decrease in hardware cost over the last several years have caused multiprocessor systems to become more common. Entire multiprocessor systems are now available to consumers on a single chip. Traditionally, shared-memory multiprocessors specify the way in which they communicate — either over a bus, or a more complicated network — through a coherence protocol. This coherence protocol is responsible for assuring that the memory system behaves in a way that guarantees correct execution by managing all communication between processors. A variety of techniques use speculation, modifying these coherence protocols in order to improve performance. For instance, the authors of [1] propose several predictors that are able to potentially skip a level of indirection in requests for access to a line by predicting the current sharers and sending requests to them in parallel to a request to the directory.

Other recent work highlights the possiblity of using speculation to simplify the design of multiprocessors [2] and to implement previously difficult-to-verify features in the coherence protocol [3]. Coherence decoupling [4] allows an out-of-order core to execute speculatively based upon potentially incoherent data. The authors of [4] provide two seperate coherence decoupling schemes, one which seeks to eliminate false sharing, and another which seeks to distribute data to its consumers preemptively. The preemptive data distribution assumes a single bus based architecture. This simplifies distribution of data, in that all transmissions are broadcasts on the bus, and thus reach all processors. In order to extend this system to an arbitrary network some form of consumer prediction [5] would be needed[1].

Methods similar to this are used in software [6] to identify likely consumers using profiling and other compiler techniques, and insert special instructions to forward data to them at appropriate times. Consumer set prediction [5], [7] attempts to identify the processors which will consume data. This allows forwarding of data to its destination before a request for the data is sent.

We propose that consumer set prediction should be combined with coherence decoupling in order to implement an update mechanism on an arbitrary topology efficiently. In this paper we show how to use a perceptron to design a consumer predictor that represents a unique tradeoff between bandwidth usage and coverage. We show how to tune the behavior of a perceptron predictor to acheive a wider range of tradeoffs between extra transmissions, and correct transmissions. Finally, we show that a perceptron predictor is able to outperform previous predictors by 21% when the goal is to achieve an approximately even tradeoff between these two factors.

II. Background

A. Consumer Predictors

Kaxiras and Young [5] provide a taxonomy of consumer predictors based on three parameters: (i) the indexing scheme of the history table, (ii) the depth of that table, and (iii) the function used to generate a prediction. Using these a predictor is represented in the form $function(index)^{depth}$. The history table contains entries, referenced by a combination of bits from the address of the block, and the PC of the instruction that wrote to that block. This table can be located at each processor, at the directories, or in a global location. In the given taxonomy this is represented by including bits corresponding to the directory or processor in the indexing scheme.

Entries in the history table are comprised of bitmaps, each of which corresponds to a group of sharers between two sets of invalidates. A bitmap contains a single bit for each processor in

[1] Broadcasting to everyone in such a system would itself be a naive form of consumer prediction.

1-4244-9707-X/06/$20.00 ©2006 IEEE

Term	Definition
Prevalence	$\frac{TP+FN}{TP+TN+FP+FN}$
Sensitivity	$\frac{TP}{TP+FN}$
Predictive Value of a Positive Test (PVP)	$\frac{TP}{TP+FP}$

TABLE I
THE THREE TERMS WE USE IN THIS PAPER TO QUANTIFY THE BEHAVIOR OF CONSUMER PREDICTORS.

the system, set to one to indicate that a processor is a sharer and zero to indicate that it is not a sharer. The number of bitmaps stored at each index is called the depth of the table. When an invalidate occurs, a new bitmap of sharers is created, and one of the old bitmaps is deleted. Thus a Consumer Set predictor $function(pid{+}pc_4)^2$ indicates a predictor located at each processor, indexed using 4 bits of the program counter, with a depth of 2. Readers interested in the details of each specific permutation of indices are referred to [5].

Kaxiras and Young [5] also describe three functions to use in consumer set predictors.

(i) **Union**: Predict that the next sharer bitmap will be the union of those present in the history table.
(ii) **Intersection**: Predict that the next sharer bitmap will be the intersection of those present in the history table.
(iii) **Two-Level PAs Prediction**: Keep a set of two bit up/down saturating counters for each potential consumer. These are indexed and updated using the history of that specific processor. Thus for N processors $N * 2^{depth}$ counters are needed.

B. Quantifying the Behavior of Consumer Set Predictors

We use the following terminology (same as that proposed in [5]) to describe the behavior of an individual predictor. Predictions can be sorted into (i) false positives (FP), (ii) false negatives (FN), (iii) true positives (TP), and (iv) true negatives (TN) depending on the prediction made (P/N), and its correctness (T/F). It is important to note that in the case of consumer prediction the two false cases have different results. A false positive incurs a penalty over normal execution. It uses up bandwidth in transmitting extra data to no effect. A false negative results in normal execution. When that processor needs information, it will send a request to the directory, as it would have without any form of consumer set prediction. Both false positives and false negatives are mispredictions, but only one of the two will cause a performance penalty.

Similarly, true negatives do not result in any benefit, while true positives can yield an improvement in performance. Both of these are correct predictions, but only one of the two is of any value. Ideally, a predictor would maximize the number of true positives, and minimize the number of false positives. To quantify this behavior three terms are defined in Table I.

The prevalence, or frequency of positive cases, is a property of the values being predicted and not the predictor itself. Thus, we can reduce comparisons of consumer set predictors to two terms: *sensitivity* (the number of potential positives that were correctly predicted), and *PVP* (the reliability of a positive prediction). Notice that when the number of true positives is maximized the *sensitivity* will be one, and when the number of false positives is zero the PVP will be one.

	Epoch 1	Epoch 2	Epoch 3	Epoch 4	Epoch 5	Epoch 6	Epoch 7	Epoch 8
Proc. A	1	0	0 / 1 / 0	0 / 0 / 0	1 / 0 / 0	0 / 1 / 0	0 / 1 / 0	0 / 0 / 0
Proc. B	1	0	0 / 1 / 0	0 / 0 / 0	1 / 0 / 0	0 / 1 / 0	0 / 1 / 0	0 / 0 / 0
Proc. C	0	1	0 / 1 / 0	0 / 1 / 0	0 / 0 / 0	1 / 0 / 0	0 / 1 / 0	0 / 1 / 0
Proc. D	0	1	0 / 1 / 0	0 / 1 / 0	0 / 0 / 0	1 / 0 / 0	0 / 1 / 0	0 / 1 / 0
Proc. E	0	0	1 / 0 / 0	0 / 1 / 0	0 / 1 / 0	0 / 0 / 0	1 / 0 / 0	0 / 1 / 0
Proc. F	0	0	1 / 0 / 0	0 / 1 / 0	0 / 1 / 0	0 / 0 / 0	1 / 0 / 0	0 / 1 / 0
Proc. G	0	0	0 / 0 / 0	1 / 0 / 0	0 / 1 / 0	0 / 1 / 0	0 / 0 / 0	1 / 0 / 0
Proc. H	0	0	0 / 0 / 0	1 / 0 / 0	0 / 1 / 0	0 / 1 / 0	0 / 0 / 0	1 / 0 / 0

Actual / Union / Intersection

Fig. 1. Predictions made by previous functions with a depth of two on a simple pattern. The predictions made by a two-level predictor would depend on its depth. Depending on initialization conditions a two-level predictor with a depth of two would make different predictions for the above example, but all such cases will contain mispredictions.

III. PERCEPTRON CONSUMER PREDICTORS

A. Evaluating Consumer Predictors

Previous work has chosen to focus on predictors that perform best in terms of either PVP or sensitivity. However, depending on the details of a multiprocessor system the potential penalties for transmitting unneeded data and the potential benefits of correctly forwarding data will differ widely. Each system will represent a potentially unique trade off between sensitivity and PVP. In fact, if the goal is to maximize sensitivity the predictor design is trivial: simply predict that every processor will be a consumer and the sensitivity will be one. Maximizing PVP is a more challenging problem. The opposite of the perfectly sensitive predictor would never predict positive. However, when TP and FP are both zero PVP is undefined. It is possible to bring the PVP of any predictor closer to one using some form of confidence estimation.

Rather than assuming that these penalties and benefits are in an extreme case, we leave decisions about this trade off to those designing specific systems, and instead investigate the set of co-optimal predictors. These are the predictors which are optimal for some trade off of sensitivity and PVP.

B. Why Perceptrons Work For Consumer Prediction

All of the previous techniques of consumer set prediction have one common limitation. In determining whether some processor will be a sharer, they look only at the history of that processor [2]. Making a prediction is simple, but potentially useful information is thrown away.

Figure 1 shows an example of a simple sharing pattern for a particular memory block. The vertical axis represents the different processors, and the horizontal axis represents different epochs over time. The sharing pattern is shown in the portion of each cell labeled Actual. A "1" in a cell indicates that the corresponding processor is a sharer of that memory block during that epoch. In this pattern two processors have access to a piece of data at any time. Each column corresponds to the set of sharers for some interval of time, with invalidations occuring between them. Each row corresponds to a single

[2] Some coherence predictors, such as the one proposed in [15] [18] do look at this information. But to our knowledge no such predictor has been directed specifically at consumer prediction.

processor. Thus processors A and B have read permission on the data, one writes and it is passed to processors C and D. E and F receive the data next, followed by G and H. The pattern then repeats.

The cells labeled Union and Intersection show the predictions made by each of the two functions with a history of depth two. The first two columns are not marked, as those predictions will depend on the initial conditions. When "1"s appear in both the actual and predictor entries, a true positive has occured. When a "1" appears for a predictor and a "0" appears for the actual result a false positive has occured, and so on through all four cases. For instance, in the case of the third set of sharers, the union predictor sees that in the last two sets of sharers processors A through D had posession of the data at some time. Union predicts that processors A through D will want the data this time. Intersection on the other hand sees that no processor had read permission to the data two times in a row. Thus intersection predicts that none of the processors will read the data.

Notice that the pattern is extremely simple and repetitive, but the previously proposed predictors cannot identify it. In fact, neither union nor intersection has a single true positive. It is clear that taking additional information into account could yield a better prediction. A reasonable question is whether such behaviors occur in practice. Does the presence of a processor in the set of sharers ever correspond to the presence of a *different* processor in a previous set of sharers? We now address this question by analyzing the amount of correlation present across processor boundaries.

Figure 2 shows the amount of correlation that a perceptron could exploit. In each group of sharers each processor has a state, either present, or not present. The top histogram shows the percentage of lines for which the state of a processor in one group of sharers is correlated to the state of the same processor in the next group of sharers. The bottom histogram shows the percentage of lines for which the state of a processor in one group of sharer is correlated to the state of a different processor in the next group of sharers. A correlation of one indicates that either the presence or absence of a processor can be linked directly to the presence or absence of another processor in the next set of sharers. On the top histogram this means that if processor P_1 is a sharer now, it will be a sharer after the next invalidate, and if it is not a sharer now it will not be a sharer after the next invalidate. On the bottom histogram this means that if processor P_1 is a sharer now, some specific processor P_2 will always be a sharer after the next invalidate. A correlation of negative one indicates that the presence or absence of a processor can be linked to the opposite behavior in the next set of sharers. Other correlations indicate a relation between one event and the next that is not absolute. A correlation of 0.9 would indicate that the vast majority of the time (95%) the value at the next time is the same as the value at this time. A correlation of -0.5 would indicate that the next value was different 75% of the time. For instance, in the top histogram this means that if processor P_1 is a sharer now, it will not be a sharer after it is invalidated; and if processor P_1 is not a sharer now, it will be a sharer after the next invalidate.

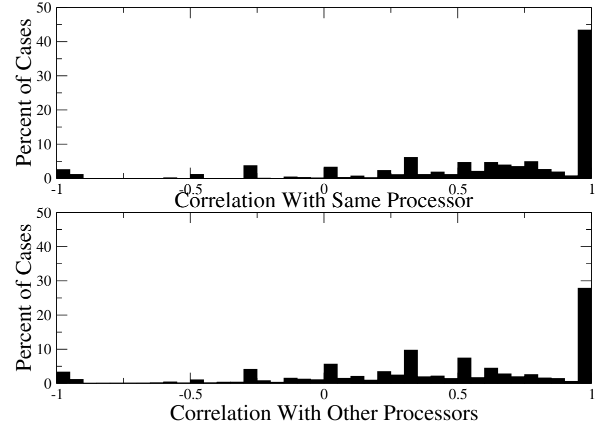

Fig. 2. Histogram of the correlations of a sharer's presence in one iteration based upon both its own, and other sharers' presence in the previous iteration. This data was collected from the SPLASH2 benchmark FMM on a 16 processor simulation, but is representative of the behavior seen in other benchmarks.

As we can see, a processor's own history is the single biggest indicator in whether it will be a sharer in the next group. However, in many cases there is a strong relationship with other processors as well. In fact, in almost 30% of cases the presence of a processor in a group of sharers can be linked to another processor in the previous group of sharers. Thus there is reason to believe that the history of other processors could improve the performance of a consumer predictor. Also, there is a measurable amount of negative correlation. This negative correlation cannot be addressed by previous predictors, except to a small extent the Two-Level predictor for which no results were published [5].

We propose taking advantage of these correlations using a perceptron. The computer science community has done a great deal of work developing neural networks constructed of *perceptrons*, each of which is trained to identify correlations between its inputs, and the desired output. By tracking correlations between the desired prediction and the inputs, perceptrons dynamically isolate the relevant portions of the input from irrelevant portions of the input. In addition, perceptrons use information about the correlations between input and output to generate predictions, as shown in Figure 2. Thus a perceptron can take advantage of both negative and positive correlations.

C. Perceptron Consumer Predictor

The perceptrons we use are identical in structure to those proposed in [8] for branch prediction, and are located with the history table. Each history table has a separate perceptron for each potential consumer, with an overall topology shown in Figure 3. Predictions are made as follows:

(i) The history table is indexed using some combination of bits from the program counter, address, and directory.
(ii) The entry at that location is used as input to as many perceptrons as the number of processors in the system.

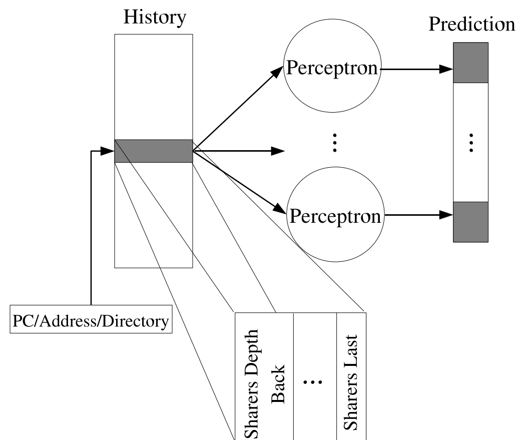

Fig. 3. Predictor Architecture. Each history table has as many perceptrons as the number of processors in the system. These perceptrons are used for every entry in the history table. Each entry in the history table has a number of bitmaps equal to the history depth, each of which contains a bit for each processor.

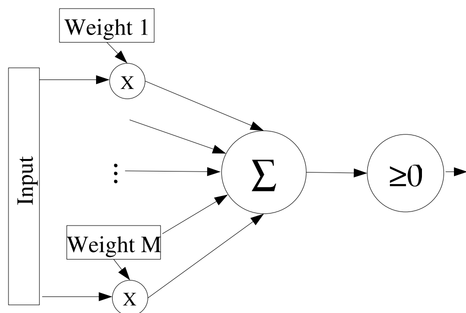

Fig. 4. Structure of a Perceptron Predictor

(iii) The outputs at each perceptron correspond to the predictions made for each processor.

We study a number of different history table configurations; a single global history table, a history table at each directory, and a history table at each processor. In all cases we find that it is best to place a history table at each processor.

To make a prediction the history for a given index is used as input to the perceptron, which has a corresponding weight for each bit. The weights are either added to, or subtracted from a sum, depending on the corresponding bit in the input. If the sum is greater than zero the perceptron predicts positive, otherwise the perceptron predicts negative. Figure 4 shows how this works. The perceptron treats the presence of a processor in a group of sharers as a 1, and its absence as a -1.

D. Update Mechanism

There are two structures that need to be maintained for the perceptron to work, the history table, and the perceptron weight table. We address each of these tasks here.

The history table keeps $depth$ bitmaps for each index. Each bitmap contains the last set of sharers corresponding to this index, which may include both PC and address information. If the history tables are located at the directory it is relatively easy to track all sharers, as the directory is responsible for tracking that information in order to maintain coherence. If the history tables are located at each processor it is slightly more complicated. In this case information about sharers can be piggybacked onto an existing response message from the directory whenever a processor requests exclusive access.

Maintaining the perceptron weights will require an extra message to be sent in a few rare cases. The perceptron weights are updated only when another processor requests exclusive access. At this time we know the set of all consumers of the last write, and would like to pass that information to the producer. If the consumer still has read permission, which is likely given that this is the first write to occur since the producer had exclusive access, we can attach this information to the invalidate request sent to the producer. If the producer has released access permission, the directory sends a message to them with the consumer bitmap. We choose to update the perceptron based on the prediction it would make *when it has recieved all the information needed to update*. Doing this prevents hysteresis effects, and reduces storage requirements. It would also be possible to store the information needed to update the perceptron when the original prediction was made, but this would fail to account for more recent changes to the perceptron, and would require additional information.

Once data arrives each perceptron is evaluated to see if it needs to be updated based upon the prediction it would have made. A perceptron is updated when its output disagrees with the actual behavior of the system or if the magnitude of the sum was less than some threshold. Each weight in the perceptron is incremented if the corresponding input agreed with the output, and decremented if the input disagreed with the output. Thus the threshold decides when the processor stops training. A low threshold means that the resulting weights are able to adapt more quickly if the behavior of the program changes. A high threshold means that the perceptron itself will be slower to change, and thus be less influenced by brief changes in program behavior. In the taxonomy proposed in [5] we denote this predictor as $Perceptron_{threshold}(index)^{depth}$.

E. Implementation Issues

There are three main concerns with implementing a perceptron based consumer predictor; the resources needed to do so, the prediction latency, and the modifications needed to the coherence protocol. One important thing to note with regard to size is that the number of perceptrons used is relatively low, $N \times H$, where N is the number of processors and H is the number of history tables. At any one history table there will never be more than N perceptrons, each of which has a total number of weights, $N \times depth$. For a 16 processor system with a depth of 4 (the largest we test) this corresponds to 1024 different weights at each processor or directory. The number of bits needed for each weight is $1 + log_2(thresh)$ [8]. We explored a range of weights consumes between five and ten bits, for a total cost of 0.63 to 1.25 KB. A few adders are needed to compute the predictions.

In addition to a potentially large size, perceptrons are also slower than many predictors. In our case it takes a the latency of six additions to fully calculate a prediction, where

Union and Intersection predictors only need to evaluate a single bitwise logical operation. However, memory and request latencies in multiprocessor systems are typically quite large. Recent work in CMPs shows that the latencies of memory requests on realistic CMPs is at least 120 cycles [9]. Given that SMP systems will have longer communication times we expect that the additional latency of 6 additions will have little effect.

Implementing consumer set prediction as part of a traditional coherence protocol is potentially challenging for two reasons. First, it is necessary to identify times at which to distribute data. Our study focuses on identifying the consumers of data so that the data can be forwarded to them before it is requested. We do not describe when to do so. If data is distributed too early it may be requested by the original processor before another processor reads the data. This will result in a delay, as the original processor aquires write permission again, and extra communication on the bus. These penalties are paid even if the consumer prediction was correct. Second, data races could be introduced into the coherence protocol. Eliminating all of these race conditions is a challenging problem, which has led to most coherence protocols in practice being simple, or unverified.

Coherence decoupling proposed by Huh et. al. [4], greatly simplifies these design issues. Coherence decoupling allows speculative execution to occur in an out-of-order core based on incoherent data. One proposed form of coherence decoupling includes a speculative update. This speculative update writes data to the bus before the coherence protocol would. Other processors that possess an invalid copy of the line in their cache update this invalid line with the new results. This allows them to speculatively execute using data that they could not have possessed yet if they had obeyed the coherence protocol. Because this was implemented on a snoopy bus it is effectively the same as predicting that all of the other processors are consumers. On a bus this makes perfect sense, as transmitting to additional consumers uses no extra resources. However, broadcast can be expensive in other, more complex topologies. Some form of consumer set prediction would be a natural extension to such an update mechanism in arbitrary interconnect topologies.

IV. Experimental Results

A. Methodology

Our study evaluates a large number of different predictors, searching the design space across depth, index, and function. To facilitate this we use trace-based simulation. The sharing patterns we study would be unchanged by implementing coherence decoupling, and so feedback of the predictor on the logical program execution can be ignored. We assume that the L2 cache of each processor is infinite and use 128-byte lines.

We gathered traces from the SPLASH-2 [10] benchmark suite using GEMS [11]. We used only the Ruby module of GEMS, simulating a 16 processor system with in-order execution, a 64KB L1 cache, and a 16MB L2 cache. The default input set was used for each benchmark.

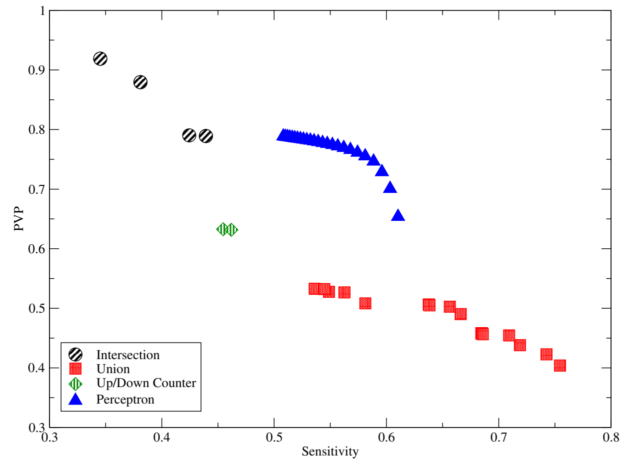

Fig. 5. The set of co-optimal predictors found for each prediction function. The perceptron has many more points because threshold was varied. With threshold held constant only a few predictors are co-optimal. Note the offsets on both axes.

We explore the space of predictors which use as many as 1M entries per history table, depths as high as 4, and the prediction functions Union, Intersection, Two-Level PAs, and Perceptron. We varied the threshold of the perceptron from 10 to 500. All the dynamic predictors were evaluated based upon predictions made as the results became known.

While the history tables proposed are quite large, recent research shows that it is possible to reduce this size substantially with little effect on performance [19]. We also show that the overall trends in these large tables are also present in much smaller tables. Rather than display the top performers according to either sensitivity or PVP we look at predictors that are co-optimal in terms of sensitivity and PVP.

B. Prediction Accuracy

Figure 5 shows the set of co-optimal predictors generated by each function in a 16 processor system. As you can see, the perceptron completely dominates the Two-Level predictor, as well as the more sensitive intersection predictors and higher PVP union predictors. In order to choose a predictor for a full system design it is necessary to know about the relative worth of PVP and sensitivity. This will vary depending on a wide range of design decisions, including the interconnect topology, processor speed, processing core design, and coherence protocol. In general PVP is more important relative to sensitivity when less bandwidth is available.

If PVP and sensitivity are of equal value a predictor's performance can be measured as the distance from itself to the perfect predictor ($PVP = 1$, $Sensitivity = 1$). In the case of this particular metric, perceptron prediction is on average 21% better than the next best predictor. Perceptron is at best a distance of 0.483 from a perfect predictor. Intersection is the next best at a distance of 0.608, followed by Union at a distance of 0.609. Regardless of the metric chosen, the perceptron predictor provides performance in a region of the consumer-set design space that was previously unavailable.

Table II displays each of the predictors in the co-optimal set. We found that the perceptron predictor did best with a single indexing scheme, and can be tuned across a wide

Predictor	Sensitivity	PVP
$Intersection(pid + pc_{16} + addr_2)^2$	0.434	0.789
$Intersection(pid + pc_{16})^2$	0.439	0.789
$Intersection(pid + pc_{16})^3$	0.381	0.880
$Intersection(pid + pc_{16})^4$	0.345	0.919
$Two-Level(pid + pc_{10} + addr_2)^4$	0.461	0.632
$Two-Level(pid + pc_{14})^4$	0.455	0.633
$Perceptron_{10}(pid + pc_6 + addr_{12})^4$	0.610	0.654
$\vdots$	$\vdots$	$\vdots$
$Perceptron_{490}(pid + pc_6 + addr_{12})^4$	0.508	0.789
$Union(pid + pc_{18})^4$	0.755	0.404
$Union(pid + pc_{16} + addr_2)^4$	0.743	0.423
$Union(pid + pc_{18})^3$	0.719	0.438
$Union(pid + pc_{16} + addr_2)^3$	0.709	0.454
$Union(pid + pc_{10} + dir + addr_4)^3$	0.685	0.456
$Union(pid + pc_{12} + dir + addr_2)^3$	0.685	0.458
$Union(pid + pc_{14} + dir)^3$	0.685	0.458
$Union(pid + pc_{16} + addr_2)^2$	0.656	0.503
$Union(pid + pc_{10} + dir + addr_4)^2$	0.638	0.505
$Union(pid + pc_{12} + dir + addr_2)^2$	0.638	0.506
$Union(pid + pc_{14} + addr_4)^2$	0.638	0.507
$Union(pid + pc_{10} + addr_8)^2$	0.581	0.508
$Union(pid + pc_8 + addr_{10})^2$	0.563	0.527
$Union(pid + pc_8 + dir + addr_{10})^2$	0.549	0.528
$Union(pid + pc_6 + addr_{12})^2$	0.545	0.532
$Union(pid + pc_6 + dir + addr_{12})^2$	0.536	0.533

TABLE II

CO-OPTIMAL CONSUMER SET PREDICTORS FOR EACH OF THE PREDICTOR FUNCTIONS. WHERE PERFORMANCE MATCHED WITHIN THREE SIGNIFICANT FIGURES ONLY THE SMALLEST PREDICTOR IS SHOWN.

range by varying only the threshold. The perceptron achieves performance between Intersection and Union in both PVP and sensitivity at the same time. It is possible to adjust the sensitivity of the intersection predictors by decreasing their depth, but they never achieve the sensitivity of any of the other predictors. None of the other predictors can offer as high a PVP as the intersection predictor. Similarly, the Union predictors dominated in terms of sensitivity, but had relatively low PVPs.

While intersection based predictors can be tuned across a range of PVP and sensitivity, they offer a very discrete range of performance based upon the depth of the history table. The perceptron predictor's behavior is relatively predictable. It performs best when located at the processors, and when the history table is indexed with twice as many address bits as program counter bits.

The behavior of the perceptron predictor can be adjusted using its threshold. Intuitively, the higher the threshold the slower the perceptron will be to adapt, and thus it will miss opportunities where a more flexible predictor could have reacted. At the same time a higher threshold should result in a perceptron that is less affected by a few occurences that do not exactly match more common patterns. As you can see from Figure 6 the results match the intuition. Here, the sensitivity and PVP of the four best perceptron predictors are shown as the threshold is varied. Raising the threshold means a higher PVP, but a lower sensitivity.

Notice that in all cases the best predictors are located at the processors, and not at the directory. There are several good intuitive reasons why this is so. Different threads may interact with some parts of memory differently than other threads. This information can be captured when the history table is located

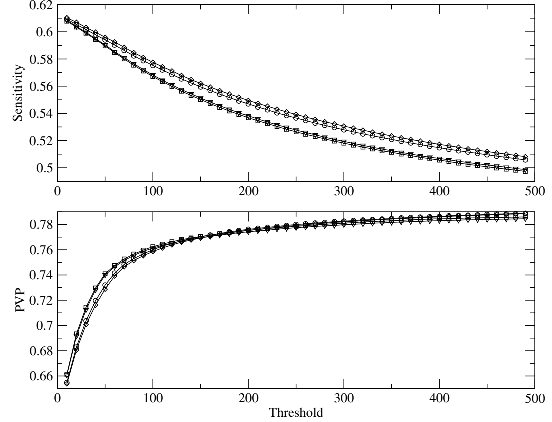

Fig. 6. Behavior of several perceptrons across variations in threshold.

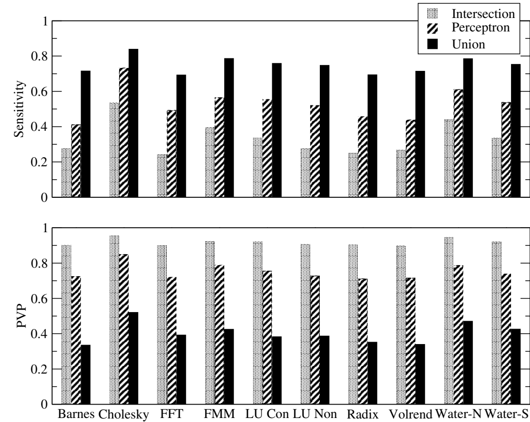

Fig. 7. Behavior of three consumer-set predictors on the SPLASH benchmarks used, $Intersection(pid + pc_{18})^4$, $Perceptron_{120}(pid + pc_{10} + addr_2)^4$, and $Union(pid + pc_18)^4$. The Intersection and Union predictors represent the maximum PVP and Sensitivity achievable.

at each processor, but not when it is located at each directory. The result of using directory based predictions rather than processor based predictions is that several different behaviors are being predicted by a single entity. This is similar to the difference between trace prediction and branch prediction [12]. Trying to make a predictor do more makes the predictor's job harder and thus hurts its performance.

Figure 7 shows the behavior of one co-optimal perceptron, instersection and union predictor on each of the benchmarks tested. The overall trends of the various functions are constant across all of the benchmarks tested. In general PVP is more consistent across benchmarks than sensitivity.

Figure 8 shows the co-optimal set of predictors if the total size of a predictor is reduced below the larger predictors we used earlier. Though the predictor performance does decrease, the overall trends remain the same.

V. CONCLUSIONS

We show how consumer predictors can be compared to each other without selecting specific interconnect details by

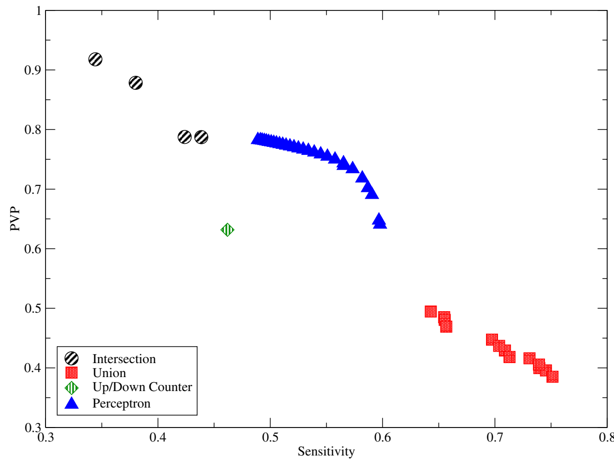

Fig. 8. The set of co-optimal predictors that can be produced using only 4K entries at each history table. Note the offsets on both axes.

comparing the chance of a predictor sending incorrect messages with its chance of missing opportunities. This method produces a range of different consumer predictor options that a system designer could pick from depending on a specific overall implementation.

We develop a perceptron consumer-set predictor that requires little more space than previous predictors. Previous predictors have populated the extremes of PVP and sensitivity. Our perceptron predictor is able to achieve a tradeoff between the two, sensitivity between 0.5 and 0.65 and PVP between 0.6 and 0.8. This range represents a previously unexplored tradeoff in consumer predictors. Unlike previous predictors, it is possible to tune the perceptron across a wide range of PVP and Sensitivity without any structural changes by adjusting the percepntrons threshold. We show that a perceptron based consumer-set predictor substantially outperforms previous predictors for systems looking for an even balance between PVP and sensitivity.

VI. Related Work In Coherence Prediction

Previous work in coherence prediction has covered a fairly broad spectrum. Some of these techniques [13], [14] add states to the coherence protocol. Others focus on table based predictors [1], [15]–[18] that predict messages before their arrival and react appropriately.

In [13] Stenstrom et. al. propose to identify certain behaviors, such as migratory sharing with a simple state machine, passing write permission on some read requests. [14] augments the state machine of the coherence protocol allowing the owner to invalidate blocks independently. In this manner many invalidations can be avoided, freeing bandwidth and potentially increasing the response time of requests for exclusive access to data.

Later forms of coherence prediction rely on table based approaches. The first of these, proposed by Mukherjee and Hill [15], keeps a history of all coherence events that occured at a processor, and uses that history as a lookup into a table which shows the last event to follow that sequence. This simple scheme is able to identify 62% - 93% of the coherence events in scientific workloads. Other works [15], [18] show how such a predictor can be used to improve the performance of a system.

Other work focuses on predicting specific coherence events and acting appropriately. Last Touch prediction [16] predicts invalidation requests before they happen. Destination-Set prediction [1], [17] predicts the invalidation requests that will be issued by the directory before sending a request to the directory.

Acknowledgment

This work is supported in part by an IBM Faculty Award and NSF award CCF-0325393.

References

[1] M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-Memory Multiprocessors. Proc. 30th Int'l Symp. on Computer Architecture, 2003.

[2] D. J. Sorin, M. K. Martin, M. D. Hill, D. A. Wood. Using Speculation to Simplify Multiprocessor Design. Int'l Parallel and Distributed Processing Symp., 2004.

[3] M. K. Martin, M. D. Hill, D. A. Wood. Token Coherence: Decoupling Performance and Correctness. Proc. 30th Int'l Symp. on Computer Architecture, 2003.

[4] J. Huh, J. Chang, D. Burger, G. S. Sohi. Coherence Decoupling: Making Use of Incoherence. Proc. 11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, 2004.

[5] S. Kaxiras and C. Young. Coherence Communication Prediction in Shared-Memory Multiprocessors. Proc. 6th Int'l Symp. on High-Performance Computer Architecture, 2000.

[6] D. K. Poulsen and P. Yew. Data Prefetching and Data Forwarding in Shared Memory Multiprocessors. Proc. 1994 Int'l Conf. on Parallel Processing.

[7] S. Kaxiras. Identification and Optimization of Sharing Patterns for High-Performance Scalable Shared Memory. Ph.D. Thesis, University of Wisconsin-Madison, Aug 1998.

[8] D. A. Jimenez and C. Lin. Dynamic Branch Prediction with Perceptrons. Proc. 7th Int'l Symp. on High-Performance Computer Architecture, 2001.

[9] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling. Proc. 32nd Int'l Symp. on Computer Architecture, 2005.

[10] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. Proc. 22nd Int'l Symp. on Computer Architecture, 1995.

[11] M.K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. Computer Architecture News (CAN) 2005.

[12] J. Gummaraju and M. Franklin, Branch Prediction in Multi-Threaded Processors, Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques, pp. 179-188, 2000.

[13] P. Stenstrom, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. Proc. 20th Int'l Symp. on Computer Architecture, 1993.

[14] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. Proc. 22nd Int'l Symp. on Computer Architecture, 1995.

[15] S. S. Mkherjee and M. D. Hill. Using Prediction to Accelerate Coherence Protocols. Proc. 25th Int'l Symp. on Computer Architecture 1998.

[16] A. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. Proc. 6th Int'l Symp. on High Performance Computer Architecture, 2000.

[17] M. E. Acacio, J. Gonzlez, J. M. Garca, and J. Duato. Owner Prediction for Accelerating Cache-to-Cache Transfers in a cc-NUMA Architecture. Proc. SC2002.

[18] A. Lai and B. Falsafi. Memory Sharing Predictor: The Key to a Speculative Coherent DSM. Proc. 26th Int'l Symp. on Computer Architecture 1999.

[19] J. Nilsson, A. Landin, and P. Stenstrom. The Coherence Predictor Cache: A Resource-Efficient and Accurate Coherence Prediction Infrastructure. Proc. 6th IEEE Int'l Parallel and Distributed Processing Symp., 2003.