Coherence Communication Prediction in Shared-Memory Multiprocessors

Stefanos Kaxiras and Cliff Young
Bell Laboratories, Lucent Technologies
{kaxiras,cyoung}@research.bell-labs.com

Abstract—*Sharing patterns in shared-memory multiprocessors are the key to performance: uniprocessor latency-tolerating techniques such as out-of-order execution and non-blocking caches have proved unable to completely hide the latency of remote memory access. Recently proposed prediction mechanisms accelerate coherence protocols by guessing where data will be used next and forwarding them to potential users before they are requested. Prior work in such shared-memory prediction schemes resulted in address-based and instruction-based predictors. Our work innovates in three areas. First, we present a taxonomy of prediction schemes that includes all previously-proposed prediction schemes in a uniform space. Second, we show how statistical techniques from epidemiological screening and polygraph testing can be applied to better measure the effectiveness of sharing prediction schemes; earlier work had reported only the ratio of incorrect predictions to correct predictions but neglected the ratio of correct predictions to actual sharing. Third, we provide simulation results of the accuracy of a practical subset of the space of schemes in our taxonomy, then analyze which components of each scheme contribute the most to prediction accuracy. Through this process, we discovered prediction schemes more accurate than those previously proposed.*

1 Introduction

In the never-ending quest for increased performance, parallelism at various levels has supplied many of the gains of recent years. Even though uniprocessor performance continues to grow exponentially, parallel machines remain attractive because they can combine the power of multiple processors on parallelizable problems. While in the past, the introduction of a parallel system trailed significantly behind the introduction of the uniprocessor system based on the same commodity processor, the first versions of today's chips are delivered SMP-ready [3]. Small-scale SMPs are often used as nodes in larger, distributed, multiprocessor implementations (HP Exemplar [7], Sequent STiNG [21], and SGI Origin 2000 [17]). The popularity of SMPs and the need to maintain programming continuity across various distributed systems established the shared-memory model as the prevalent form of parallel programming.

The user's view of a shared-memory system is elegantly simple: it appears that all processors read and modify state in a single shared store. The difficulty in implementing any such system—especially distributed systems—is propagating values from one processor to another: the actual values are created close to one processor but might be used by any or many others. If the implementation could perfectly guess the sharing patterns of the program, the nodes of a distributed multiprocessor could spend more of their time computing and less of their time waiting for values that reside at remote locations. Even with non-blocking caches and out-of-order processors, studies show that the relatively long access latency in a distributed shared-memory remains a serious impediment to performance [25]. Our work uses prediction to reduce access latency in distributed shared-memory systems by attempting to move data from their creation place to their use points as early as possible.

In a distributed shared-memory system, a coherence protocol—typically directory-based [5] (e.g., DASH [18], SCI [12], DIR$_i$NB [1], etc.)—keeps processor caches coherent and transfers data among the nodes. In essence, the coherence protocol carries out all communication in the system. Coherence protocols can either invalidate or update shared copies of a data block whenever the data block is written. *Update* protocols forward data from producer nodes to consumer nodes but lack a feedback mechanism to determine the usefulness of data forwarding.[1] *Invalidation* protocols provide a natural feedback mechanism (invalidated readers must have used the data) but no means to forward data to their destination. Our approach uses prediction on top of an invalidation protocol (e.g., Dir$_i$NB) in a typical distributed shared-memory system. We use prediction to forward data to their destination; we use the set of invalidated readers as feedback to the prediction mechanisms.

Although prediction can be used to optimize various sharing patterns (migratory sharing, wide sharing, etc. [14]), we concentrate on predicting future readers of newly created data. We expect this to work particularly well for static producer-consumer sharing [28]. However, our work takes an all-encompassing approach. Any shared-data writer is considered a producer, and any shared-data reader is considered a consumer. Thus, we include all sharing patterns in our predictions. We do not even exclude migratory sharing [28], where the succession of producers and consumers—governed by the

[1] Competitive update protocols have been proposed where the responsibility for participating in an update operation lies on the readers, not the sender [8].

locking in the program—is effectively random. We take this approach because we do not assume any other filter in the system which could distinguish sharing patterns. This also makes our results conservative, since in many cases we can increase accuracy by excluding some of the sharing from our predictors.

The first two approaches proposed for prediction were *address-based prediction* [24,16], which tracks the access history of data blocks, and *instruction-based prediction* [13,14], which tracks the history of instructions in relation to coherence events (such as cache misses or write faults). In this paper, we examine address-based and instruction-based prediction not as two different approaches but rather as two ends of a spectrum which includes *hybrid* prediction.

The design space for an implementation of coherence communication prediction is huge. An actual implementation needs at least two components: an accurate prediction scheme and a data-forwarding protocol. Assessing performance seems premature without understanding both components. This paper focuses on the first component, evaluating plausible prediction schemes in isolation. This approach gives us valuable insight on the predictability of coherence communication. With knowledge of the characteristics of the prediction schemes, we can then select interesting combinations of predictors and data-forwarding protocols to assess performance.

1.1 Contributions of this paper

1. We propose a taxonomy of prediction schemes, encompassing: (i) access of a predictor entry (address-based, instruction-based, and hybrid) including the implications of predictor location, (ii) prediction functions (union/intersection prediction, last prediction, and two-level prediction), and (iii) history update of the predictors (direct update, forwarded update, and ordered update).

2. Using trace-driven simulation, we evaluate prediction accuracy, and bit cost per scheme for representative programs drawn mainly from the SPLASH suite [26].

3. To interpret the results we use statistical concepts and terms widely used in epidemiological screening and polygraph testing. We apply the ideas of *prevalence, sensitivity*, and the *predictive value of a positive test* to the above space of predictor schemes.

1.2 Structure of this paper

Section 2 reviews previous work on prediction in uniprocessors and in shared-memory. We then describe the design space for predictors in Section 3. The same section introduces a taxonomy for predictors. Before showing results (Section 5), we discuss the metrics we use to evaluate predictors in Section 4. Simulation results for the predictor space appear in Section 5. Finally, we summarize in Section 6.

2 Background and related work

Prediction has a long history in computer architecture, going back to the branch prediction studies of Smith [27] and Lee and Smith [19]. More recently, Yeh and Patt [29] introduced two-level adaptive predictors, which exploited patterns of branch directions to achieve higher prediction accuracy; we will examine the usefulness of pattern predictors to sharing prediction as part of this study.

Prediction has also proved useful in other parts of the microprocessor. Moshovos and Sohi [23] and independently Chrysos and Emer [6] predicted when loads and stores were unlikely to access the same location and therefore need not be stalled to enforce true data dependencies. Recently, research has focused on value prediction [20], a generalization of binary branch predictions that tries to guess whole register contents. As we shall see, the sharing bitmaps that we exploit appear similar to values but are actually vectors of single-bit predictions.

Prediction in the context of shared memory was first studied by Mukherjee and Hill who showed that it is possible to use address-based 2-level predictors at the directories and caches to track and predict coherence messages [24]. Subsequently, Lai and Falsafi modified these predictors to use less space (by coalescing messages from different nodes into bitmaps) and showed how they can be used to accelerate reading of data [16]. Independently, Kaxiras and Goodman proposed instruction-based prediction for migratory sharing, wide sharing and producer-consumer sharing [14]. Since static instructions are far fewer than data blocks, instruction-based predictors require less space to capture sharing patterns in the system.

3 Predictor design space

Our task is to predict the nodes that will read newly created data. Our prediction scheme will therefore observe sharing bitmaps of previous readers, using them to guess a new bitmap of predicted readers. There are three major divisions in how the work is performed: access, prediction, and update. *Access* tells us which predictor entry to use for each prediction. Prior work categorized schemes as address- or instruction-based; one of our contributions is to explain the more general space from which these two kinds of schemes are drawn. *Prediction* tells us how to interpret the predictor entry state to generate a new prediction. *Update* not only updates the state of the predictor; it also covers the timing of when old bitmaps arrive and are incorporated into the state. A second major contribution of this work is to explain some of the subtleties involved in updates.

Sections 3.1, 3.2, and 3.4 respectively, cover access, prediction, and update. Before discussing update, Section 3.3 summarizes the issues involved in the data-forwarding component; an actual data forwarding protocol remains outside the scope of our work. Lastly, Section 3.5 describes a naming scheme for our predictor space.

3.1 Access

When new data are written the information we have available comprises the *processor-id (pid)* along with the *program counter (pc)* of the store instruction that writes the data and the *directory-id (dir)* and *address (addr)* of the data block being written. We use this information to access a specific predictor entry. For example, in address-based predictors we use the data-block's address (*addr*) as an index to the predictor table [24,16] and similarly in instruction-based predictors we use the instruction's *pc* as an index [14].

Initial proposals placed instruction-based predictors next to the processors and address-based predictors next to the directories because these locations are where such information is most readily available. However, this information (*pid*, *pc*, *dir*, *addr*, and history information) could be piggybacked on request/response messages and transferred to any desired location—assuming we can extract the *pc* from the processor. The location of the predictors is an *implementation* choice that has performance implications but does not affect the accuracy of the *system-wide* prediction scheme. We therefore consider system-wide predictor designs without regard to the location of the predictors. Location should be dictated by the ease of implementation and the overhead to transfer the relevant information to the correct place.

To abstract away from implementation details, we analyze a single global predictor that represents all the predictors in the system (depicted in Figure 1a). Any combination of *pid*, *pc*, *dir*, and *addr* can be used to index this global predictor.

In an actual distributed implementation we would divide the global predictor in N equal parts and distribute them either over N processors (Figure 1b) or over N directories (Figure 1c). An implementation where the predictor was distributed over the processors would be equivalent to a global abstraction including pid indexing: the physical distribution into N processors gives equivalent predictions to using $\log_2 N$ bits of indexing in the global abstraction. The same holds for dir indexing and distribution across directories. In general, a global predictor cannot be distributed simultaneously at both the processors and the directories since its individual parts could be different depending on their location. Thus, we consider the distribution shown in Figure 1d as two different predictor schemes coexisting in the same system rather than a single distribution of a global predictor.

To guarantee that a hypothetical global predictor and its distributed implementation behave the same, we never use subsets of the *pid* or *dir* information: we use all or none of each. We do, however, freely truncate the *pc* and *addr* fields to meet a given implementation cost. Thus, we need only be concerned of the behavior and cost of the global predictor regardless of its distribution.

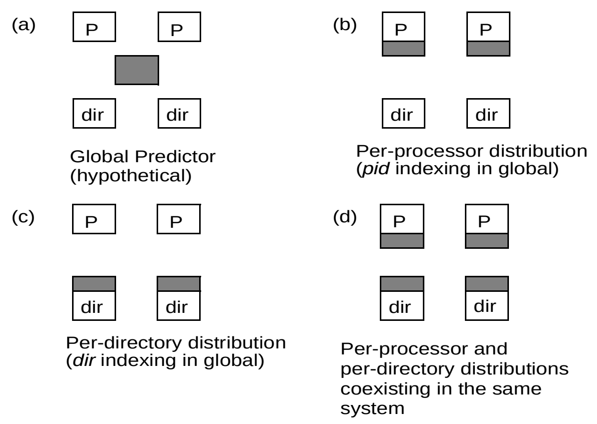
(a)
P
P
dir
dir
Global Predictor (hypothetical)
(b)
P
P
dir
dir
Per-processor distribution (*pid* indexing in global)
(c)
P
P
dir
dir
Per-directory distribution (*dir* indexing in global)
(d)
P
P
dir
dir
Per-processor and per-directory distributions coexisting in the same system

Figure 1. Location and indexing

The 16 possible indexing schemes for the global predictor are shown in Table 1. With no indexing (case 0) we have a single entry for the whole system. When neither *pid* nor *dir* are used we cannot distribute the global predictor to the processors or to the directories (cases 0, 1, 4, and 5 in Table 1). This necessitates centralized implementations and although we do not expect such schemes to be common in distributed systems we examine them for completeness.

When either the *pid* or the *dir* but not both appear in indexing they permit distribution at their respective location. Cases 2, 3, 6, and 7 can be distributed at the directories while cases 8, 9, 12, and 13 can be distributed at the processors. Note that cases 2 and 8 represent single-entry predictors at the processors or at the directories, respectively. When both *pid* and *dir* appear in indexing (cases 10, 11, 14 and 15) only one of them can denote location.

					Possible distributions		
No.	**pid**	**pc**	**dir**	**addr**	**at proc.**	**at dir.**	**Comments**
0					—	—	1-entry, Centralized
1				Y	—	—	Centralized
2			Y		—	Y	1 entry per directory
3			Y	Y	—	Y	
4		Y			—	—	Centralized
5		Y		Y	—	—	Centralized
6		Y	Y		—	Y	
7		Y	Y	Y	—	Y	
8	Y				Y	—	1 entry per processor
9	Y			Y	Y	—	
10	Y		Y		Y	Y	
11	Y		Y	Y	Y	Y	
12	Y	Y			Y	—	
13	Y	Y		Y	Y	—	
14	Y	Y	Y		Y	Y	
15	Y	Y	Y	Y	Y	Y	

Table 1: Indexing schemes for global predictor

3.2 Prediction

After we have accessed an appropriate predictor entry using some indexing scheme, we then interpret the information in this entry to make a prediction. By "prediction function" we include the entire design of the individual predictor entry: what state it maintains, how a prediction is produced, and what action occurs when an update arrives. We predict an entire sharing bitmap, since any of the other processors in the system might read the newly created data. All of the schemes proposed so far (Cosmos/VMSP [24,16], Last-prediction and Intersection-prediction [13,14], Sticky-Spatial [4]) predict sharing bitmaps as values.

Prediction is best at doing binary decisions, rather than values. While the sharing bitmaps of our prediction schemes look like values, they can be predicted individually, bit per bit. This means that we can treat each reader independently of other readers in a prediction bitmap. Unlike value prediction [20], the sharing bitmap prediction does not have to be exact. As long as the predicted bitmaps are close to the real bitmap the prediction can still be useful.

Most commonly, the state for a predictor is just the sequence of previous bitmaps, or even just the most recent bitmap.[2] Since implementations are necessarily finite, this sequence has a bounded size which we call *history depth*. Updates involve shifting out the oldest bitmap and shifting in the newest, while predictions are some combinatorial function over the sequence of bitmaps.

The possibilities for a predictor function are many. We examine previously proposed predictor functions and in addition pattern-based (2-level) prediction functions. For all prediction functions we examine various history depths. The prediction functions we simulate are:

- Last prediction (last). Predict the last sharing bitmap stored in the corresponding predictor entry. To update, replace the last bitmap of an entry with the feedback bitmap. Lai and Falsafi proposed a Last predictor indexed by both *addr* and *pid* [16].
- Union/Intersection prediction (union, inter). Predict the union or intersection of all the bitmaps stored in the corresponding predictor entry. The number of bitmaps stored in an entry is determined by the history depth. To update, replace the oldest bitmap in an entry with the feedback bitmap. Kaxiras and Goodman proposed an intersection predictor with history depth of two [14]. Last prediction described above is identical to union/intersection prediction of history depth one.
- Two-level PAs prediction [29]. Each entry contains a separate history register and pattern table for each potential reader. For *N* nodes, there are *N* history registers (with as many bits as the history depth) recording patterns for *N* potential readers, and there are $N \times 2^{depth}$ 2-bit counters that supply predictions. Using the history registers we index the pattern tables to get a binary prediction per node. The aggregate of all the binary predictions is the prediction bitmap. The corresponding bits of the feedback bitmap are used to update the history registers and pattern tables for each possible reader.

[2] The one scheme for which this is not true is the Sticky-Spacial scheme [4], where the bitmaps of neighboring cache lines also play a part. Currently we do not examine Sticky-Spacial functions but our work can be expanded to include such schemes.

3.3 Data-forwarding

Once we have a prediction we can forward data to predicted readers using a *data-forwarding* protocol. Such protocols represent *optimizations* and are not part of the prediction space we study. This section is helpful, however, in understanding the next section's discussion of updating.

The behavior of the prediction mechanisms is largely independent of the data-forwarding protocol as long as history information is based on *true sharing* in the system. We assume a protocol similar to Koufaty's and Torrellas' protocol for *forwarding instructions* [15]. Other protocols, such as speculative pre-sends proposed by Kaxiras and Goodman [14], could be used equally well.

Soon after a data block is written, data are forwarded to predicted readers. Forwarding can be initiated by the directory or by the writer. In the former case, the triggering of the forwarding is based on some heuristic such as those proposed by Lai and Falsafi [16]. In the latter case, the writer sends a request to the directory to trigger forwarding.

Ideally, data arrive at readers in time to eliminate the miss latency they would experience otherwise. In practice, only some of the forwarding would be successful: late forwarding is ineffective since the readers would go ahead and request the data on their own; early forwarding is useless when we mistakenly forward intermediate values before the final values that need to be communicated are produced.[3] In addition, forwarded data might conflict with other useful data in the caches. Such issues are partly addressed in other work [14,16]. In this paper, we are not concerned with these problems since our main focus is prediction accuracy. We consider data forwarding to be correct as long as the destination node is a true reader.

3.4 Update

Predictors should be updated with reasonably accurate history information to be useful. By studying address-based, instruction-based and hybrid predictors we realized that there are differences in the ways these predictors can be updated with history information. To our knowledge this is the first detailed examination of update mechanisms for such predictors.

The source of all our history information is the invalidation of nodes. At invalidation, the sharing bitmap of a data block corresponds to its previous readers.[4] Cache replace-

[3] Correctness is guaranteed in this case because the writer always gives up its write permission upon forwarding.

ments prior to invalidation can obscure our view of the true sharing in the system but we try to minimize such effects in our evaluation by using relatively large caches (Section 5.1).

Furthermore, prediction could also obscure true sharing. When the forwarding protocol records predicted readers in the directory (as it does in our case), the sharing bitmap at invalidation might be polluted by bad predictions. For accurate information we must be able to distinguish the set of *true* readers form the set of predicted readers *that did not read the data*. We require all nodes being invalidated to report back to the directory whether they accessed the data or not. *Access* bits in the cache lines, piggybacked on the invalidation acknowledgment messages, can be used for this purpose. Hence, to simplify our discussion we will assume that at invalidation-time the sharing bitmap of a data block corresponds to its *true* previous readers.

In address-based predictors (where we only use *dir*/*addr* indexing) the sharing bitmap at invalidation is exactly the history information we want for the new prediction. Each time a data block is written, we use the set of invalidated (true) readers as history to generate the new prediction. We call this ***direct update*** of the predictor.

However, in instruction-based predictors we seek history that corresponds to a writer and not necessarily to the arbitrary data block that is being written at that time. The following example illustrates the difference between the history of a data block and the history of a writer.

Consider the time-line of a data block which is written by a single writer (some arbitrary *pid*/*pc* combination) in Figure 2. By writing to the data block, writer *A* always invalidates its own previous readers. In this case, direct update accurately uses the writer's own history for the current prediction.

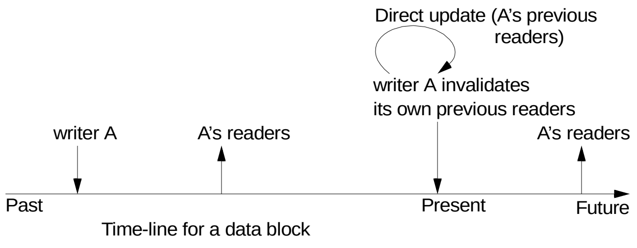
Direct update (A's previous readers)
writer A invalidates its own previous readers
writer A
A's readers
A's readers
Past
Present
Future
Time-line for a data block

Figure 2. Direct update

However, when multiple writers alternate writing the data block (Figure 3), each writer might invalidate someone else's readers and thus learn of someone else's history. In this case, direct update (using potentially another writer's history for the current prediction) is a heuristic. To use the correct writer's history we define ***forwarded update*** for instruction-based and hybrid predictors. In this scheme, when a writer invalidates someone else's readers, it forwards this history to the appropriate predictor entry so it can be used by the correct writer. Forwarded update requires last-writer information (pid/pc) for each data block so invalidated readers can be associated to a specific writer.

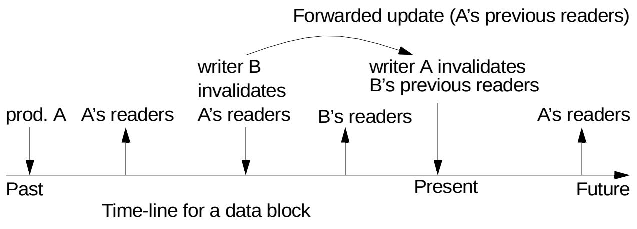
Forwarded update (A's previous readers)
writer B invalidates A's readers
writer A invalidates B's previous readers
prod. A
A's readers
B's readers
A's readers
Past
Present
Future
Time-line for a data block

Figure 3. Forwarded update

Forwarded update distinguishes among writers but there is yet another problem with the timing of updates. Forwarded update does not guarantee correct update order for all predictions. Consider for example the situation in Figure 4. Writer *A* writes to data block *X*. Well before writer *B* has a chance to forward history information, *A* decides to write data block *Y*. *B* forwards history to *A* too late to be used in *A*'s second prediction.

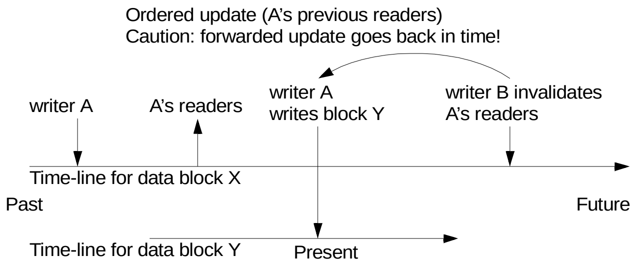
Ordered update (A's previous readers)
Caution: forwarded update goes back in time!
writer A
A's readers
writer A writes block Y
writer B invalidates A's readers
Time-line for data block X
Past
Future
Time-line for data block Y
Present

Figure 4. Ordered forwarded update

We define ***ordered update*** to be forwarded update but correctly ordered with respect to all predictions. Ordered update is not possible for some implementations since it would require updates to go back in time as in the example in Figure 4. However, forwarded update is naturally ordered in hybrid predictors that use full *dir* and *addr* indexing. In such predictors, no entry can be used for two consecutive predictions without an intervening update. Since ordered update represents the most accurate information for the predictors, we simulate it in our evaluation for all schemes despite the fact that it cannot be implemented for many of those. Note that for pure address-based schemes (*dir*/*addr* indexing only) the direct, forwarded and ordered update schemes are equivalent.

3.5 Notation

We describe our prediction schemes using names of the form *prediction-function(index)*^{depth}, where *prediction-func-*

[4] Alternatively, this information could be gathered one reader at a time as readers come along and read the data block. However, this method could prove costly if this information needs to be transferred elsewhere and defeats the purpose of data-forwarding which would prevent the reader from going to the directory in the first place.

tion indicates the form of prediction scheme state and the function used to update the state, *index* indicates the pieces of address or instruction information used to look up prediction state, and *depth* is a history depth parameter by which all of our update functions are parameterized. In some cases we will append the suffix *[direct]*, *[forwarded]*, or *[ordered]*, to indicate the update mechanism used with a scheme, but most of our results are segregated by update mechanism so we seldom need this additional information.

For example, *union*$(pid+dir+add_4)^2$*[direct]* would represent a scheme using direct update, indexing its prediction state using the processor number, directory node, and four bits of data-block address, and unioning the last two sharing bitmaps to predict the next one for each index. This scheme can be distributed either to the processors or to the directories.

A last-bitmap scheme indexed by directory node and eight bits of address would be called *union*$(dir+add_8)^1$ or *inter*$(dir+add_8)^1$. This scheme can only be distributed at the directories. We remind the reader that a union- or intersection-based scheme with a history depth of one is the same as a last-bitmap scheme.

Lai and Falsafi's predictor was located at the directory, indexed using address and processor node, and used a last-bitmap predictor. This is *union*$(dir+pid+add_n)^1$*[forward]*, which is the same as *inter*$(dir+pid+add_n)^1$*[forward]*, where n is the number of bits used to represent the address dimension of the table.[5]

Kaxiras and Goodman discussed an intersection-based predictor which is just *inter*$(pid+pc_n)^2$*[direct]*. They also described a "last" predictor which at first glance appears to be *inter*$(pid+pc_n)^1$*[direct]*. On closer examination, what Kaxiras and Goodman call a last-prediction scheme uses a slightly different update function that predicts the last sharing bitmap only if the current and last bitmap overlap. We would invent a new update function name for such a predictor, such as "*overlap-last*$(pid+pc_n)^1$*[direct]*". For space reasons, we do not simulate the *overlap-last* predictor in this paper.

4 Metrics for sharing prediction

The penalties for sharing prediction are asymmetric and have varying performance penalties depending on the way in which the predicted and actual sharing patterns agree or disagree.[6] There are four cases shown in the Venn diagram of Figure 5: *true positive* (correctly predicted shared), *true negative* (correctly predicted not shared), *false positive* (incorrectly predicted shared), and *false negative* (incorrectly predicted not shared). True positive cases are potential performance wins for our approach: true positive predictions will allow us to correctly forward data and to hide some of the data access latency in the program. False positive cases can be punitive, especially if the communications network of the multiprocessor is heavily loaded. False negative and true negative cases appear to do no harm: they generate no forwarding traffic. But false negatives are missed opportunities for data forwarding.

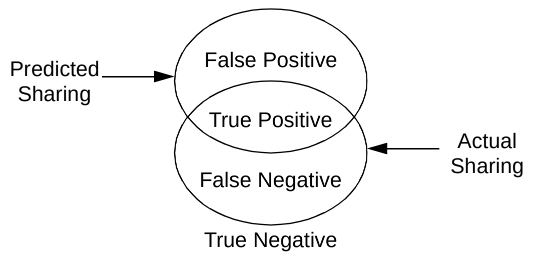

Figure 5. Venn diagram of the cases for multiprocessor sharing prediction

Sharing prediction is not unlike a medical screening test. An ideal prediction scheme would exactly predict the underlying sharing pattern of an application. Similarly, an ideal medical screening test would always diagnose correctly. And like medical screening, there are different costs to false positives and false negatives. Following Grunwald et al's work on confidence estimation and analysis [11], we liberally reuse statistical terms from the epidemiological screening and polygraph testing communities [2,10]. These terms examine a number of other ratios among the sharing prediction cases. Table 2 defines the terms we use in this paper.[7]

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 2: Definitions of statistics

The *prevalence* or *base rate* is the rate at which true sharing takes place. Prevalence bounds the total possible benefit due to our prediction scheme. The *sensitivity* of a test is the ratio of true positive events to all shared (true positive plus false negative) events. Sensitivity indicates how well the test predicts sharing when sharing will indeed take place. A sensitive predictor is good at finding and exploiting opportunities for sharing; an insensitive predictor misses many opportunities.

The amount of additional sharing traffic generated by a prediction scheme is the sum of the true positive and false positive traffic.[8] A useful measure of the yield of a prediction

5 We do not classify Mukherjee and Hill's predictors because they were predicting coherence messages, not sharing bitmaps.

6 This differs from, e.g., branch prediction, where the misprediction penalty is roughly the same regardless of the direction of a branch.

7 *Specificity* and *predictive value of a negative test (PVN)* are two related statistical terms [2,10] but we do not use them in this paper.

scheme is the ratio of the true positive traffic to the sum of the true positive and false positive traffic: this represents the percent of useful data forwarding. Prior studies [9,24,14] have held up this metric as “prediction accuracy;” this ratio is known as the predictive value of a positive test, or PVP. PVP is intuitively useful; it corresponds to the percentage of useful data-forwarding traffic out of all data-forwarding traffic. But using PVP alone misses the benefit lost to false negatives. By understanding and evaluating schemes on both sensitivity and PVP, we can understand not just the fruitfulness of our forwarding traffic, but also how well we are doing compared to the best possible in both positive and negative terms.

5 Results

Before we present results in this section we briefly describe our simulation methodology. We continue with basic statistics, and discuss prevalence of our predictions. We present results for previously proposed predictors and for the top performing predictors we have uncovered. We continue with results for the whole design space and show how prediction accuracy is affected by various prediction functions, indexing schemes, update schemes, history depths, and indexing field sizes.

5.1 Methodology

To study the large design space for predictors we used trace-driven simulation. Because the metrics we study are not affected by the timing of events in the execution of a program, trace-driven simulation is adequate for our purposes. Although data-forwarding can change the timing of a program, it does not change fundamental sharing patterns which we try to capture. Cache size and line size have a much greater effect on sharing patterns. We tried to minimize cache size effects by using relatively large L2 caches. Line size affects false sharing in the system which in turn affects our prediction schemes. We used a 64-byte line size. Limited experimentation with smaller line sizes showed that our predictor schemes exhibit qualitatively the same behavior.

In addition, trace-driven simulation allows us to study ordered update which cannot be implemented for some prediction schemes. To simulate ordered update we used a first pass through the trace and the final state of the memory to discover all relevant sharing information.

We generate traces for several SPLASH and other programs using the RSIM execution-driven simulator [25]. These programs have been described extensively and we will refer to other work for their descriptions [26,24]. Table 3 lists the inputs we use to generate traces.

Benchmark	Input
barnes	8K particles
em3d	9600 nodes, degree 5, 15% remote
gauss	512x512 array
mp3d	50K molecules
ocean	258x258 grid
unstruct	2K mesh
water	512 molecules

Table 3: Benchmark input size

We simulate 16-node systems with a fast 2-D torus interconnect, out-of-order processors, and non-blocking caches. System parameters are listed in Table 4. We trace all write misses, write faults, directory writes, and invalidations. Data placement in our programs is either done explicitly by the programmer or by RSIM which uses a first-touch policy on a cache-line granularity. Thus, initial data-placement is quite effective in terms of reducing traffic in the system.

CPU	300 MHz, 4-issue per cycle
	64-entry reorder buffer
	64-entry load/store queue
L1	16Kbyte direct-mapped, 64-byte lines
L2	512Kbyte 4-way set-associative, 64-byte lines
	52 cycles local memory latency
	133 cycles remote memory latency

Table 4: RSIM system parameters

5.2 Basic statistics

Table 5 lists statistics about the static store instructions executed, the store instructions involved in predictions, the total number of cache blocks touched, and the total number of store cache misses incurred during runs of our benchmarks. We list the maximum number of store instructions executed and predicted at a node. We list only the store instructions that touched shared data. Stores that were never executed or accessed only local data are not shown.

Benchmark	Executable Size (bytes)	Maximum Static Stores per node	Maximum Predicted Stores per node	Total Cache Blocks Touched	Total Coherence Store Misses
barnes	213852	164	61	22241	161911
em3d	181936	35	23	51889	262451
gauss	145624	21	13	32946	129528
mp3d	196652	160	71	30182	212828
ocean	284004	380	230	239861	2871656
unstruct	222740	69	67	2832	633607
water	216120	69	27	2896	172925

Table 5: Store instruction and cache block statistics

The number of live static store instructions is very small, as observed before by Kaxiras and Goodman. This suggests that predictable behavior that occurs on a per-*pc* basis can be much more easily exploited by an instruction-based or hybrid predictor than by an address-based predictor. The low counts also suggest that the SPLASH benchmarks (except *ocean*)

[8] We can save some bandwidth because nodes that used forwarded data need not request that data from the directory. Depending on the degree of sharing and the accuracy of the prediction scheme, the net effect on bandwidth will vary.

may not be large enough to adequately stress a predictive sharing scheme; similar criticisms have been leveled against the SPEC benchmarks for their relatively small number of static branches [22]. On the other hand, the small number of static stores can also be seen as encouraging: store misses mark the true sharing patterns of the program, so examining sharing patterns per-*pc* may give a lot of leverage.

5.3 Prevalence of sharing

Table 6 indicates the prevalence of sharing for each of our traces. Prevalence records the percent of sharing that actually takes place; another way to view it is the percentage of set bits in the sharing bitmap. Prevalence is equivalent to the *degree of sharing* in shared memory programs [28].

Prevalence is in general very low (as also attested in [28]), attaining its maximum in the *barnes* and *unstruct* traces, with about 15% prevalence. Low prevalence is unsurprising, since high prevalence would only occur if most nodes read every value written by every store. However, low prevalence has interesting implications for the design of prediction schemes. Conditional branches have a taken prevalence of roughly 65% [27]; it is likely that most branch prediction structures are built to exploit this nearly-even bias. The much lower prevalence of sharing suggests that a different sort of predictor might do better than prediction schemes like those used for branch prediction.

Benchmark	Dynamic Sharing Events	Dynamic Sharing Decisions	Prevalence (%)
barnes	391085	2590576	15.10
em3d	133926	4199216	3.19
gauss	205666	2072448	9.92
mp3d	306990	3405248	9.02
ocean	983085	45946496	2.14
unstruct	1300764	10137712	12.83
water	335482	2766800	12.13

Table 6: Prevalence of sharing: the average percentage of nodes that read a cache line

The arithmetic average of prevalence over all of our benchmarks is 9.19% (or equivalently a degree of sharing of 1.5).

As discussed by Gastwirth [10], low prevalence also compounds the errors in measuring the accuracy of a prediction scheme. As the prevalence of the underlying phenomenon decreases, the measurement error increases, exacerbating the problem of a high fraction of false positives by a high degree of uncertainty.

5.4 Prediction Accuracy

We explored the space of predictor schemes up to an implementation cost of 2^{24} bits, or 2Mbytes across the entire machine, or 128Kbytes per node, which is comparable to on-chip caches on some modern machines. In our accounting, we counted the bit costs for both the history shift registers and the pattern history tables of 2-bit counters in per-processor pattern schemes.

Table 7 shows the size, sensitivity, and PVP of sharing schemes reported by previous studies, run under our benchmarks. It also shows the same statistics for baseline last-bit-map prediction, a scheme which requires no storage and simply predicts that the next sharing bitmap will be the same as the last direct sharing bitmap in the system (the bitmap of readers invalidated by a directory the last time a line changed to exclusive access). The baseline case is a useful filter function: it costs no storage, so other schemes must do better to warrant using them.

	description	scheme	size as $\log_2$(bits)	sensitivity	PVP
direct update	baseline-last	last()[1]	0	0.57	0.66
	Kaxiras-instr.-last	last(pid+pc_8)[1]	16	0.57	0.66
	Kaxiras-instr.-inter.	inter(pid+pc_8)[2]	17	0.45	0.80
	Lai-address+pid-last	last(pid+mem_8)	16	0.57	0.66
forwarded update	Kaxiras-instr.-last	last(pid+pc_8)[1]	16	0.51	0.61
	Kaxiras-instr.-inter.	inter(pid+pc_8)[2]	17	0.43	0.80
	Lai-address+pid-last	last(pid+mem_8)	16	0.55	0.66

Table 7: Schemes reported by earlier work

5.4.1 Top performers. Tables 8 and 9 show the schemes in our search space with the highest PVPs under direct update and forwarded update, respectively. All of the schemes are deep-history intersection schemes; this makes sense, as intersection schemes will maximize PVP by speculating only on very stable sharing relationships. Two of the top-ten schemes are common to the two tables (shown shaded). Direct update and forwarded update have very little influence on PVP. However, the forwarded schemes on average are more sensitive. None of the high-PVP schemes is sensitive compared to a last- or union-predictor scheme. This means that they will generate very productive traffic, but they will miss many opportunities for sharing.

Table 10 shows the ten most sensitive schemes in our space using direct update. All are union schemes with the maximum history depth that we allowed, 4. All schemes are roughly comparable in sensitivity, but with different values of PVP. What is most interesting is that by far the least expensive scheme (union(dir+add_2)^4) is fifth-best overall; it uses only directory node number and two address bits with a history of depth 4 to predict.

Table 11 shows the ten most sensitive schemes in our space using forwarded update. There is very little difference between the direct- and forwarded-update schemes: six of the top ten schemes are common to the two lists (shaded), and the statistics differ little from column to column.

None of our top-ten tables shows any two-level adaptive history schemes. This is surprising, as the best branch predictors exploit local or global patterns to improve prediction accuracy. We examined our traces by hand, and we found that wherever we saw repeating patterns of sharing bitmaps for a particular index, the same disambiguation of cases could be gotten by using additional (possibly different) index bits and no pattern history.

5.4.2 Access, prediction and update. The interaction of access mechanism, prediction function, and update mechanism results in prediction schemes with a variety of values for sensitivity, PVP and cost. In this section, we show how sensitivity and PVP change with various predictor designs. We limit the maximum index size to 16 bits, and we use a history depth of two (two 16-bit bitmaps per entry). The total active bits in the index (the sum of bits used for *pid*, *pc*, *dir* and *addr*) and the size of an individual entry define a cost for the predictor. Forwarded and ordered update assume additional information at the directories, so they are more expensive than direct update. Figures 6, 7, and 8 show sensitivity and PVP results (arithmetic average over all benchmarks) for intersection prediction, union prediction and PAs prediction respectively. Each figure also shows the effects of the three update schemes. The index labels use "Y" to denote *pid* or *dir* indexing (four bits each) and the number of bits used for *pc* and *addr* indexing (2, 4, 6, 8, 12, and 16 bits).

For intersection prediction schemes (Figure 6), *pid* indexing (and to a lesser extent *pc* indexing) tends to increase both sensitivity and PVP. Direct update smooths out differences between indexing schemes, while forwarded and ordered update result in larger variations for the indexing schemes which contain *pid*, *pc* and *addr*.

Union prediction (Figure 7) behaves similarly with the only difference that the sensitivity curve is higher than the PVP curve. Union prediction makes more, but less good, predictions.

Because PAs predictors (Figure 8) are inherently expensive (each entry comprises history registers and pattern tables) the indexing schemes cannot include as many bits as the intersection/union predictions for comparable cost. PAs predictors also benefit from *pid* indexing. Forwarded and prefect update increase both sensitivity and PVP.

An all-around bad performer is *pc*-only indexing. When *pc* is used without *pid* both sensitivity and PVP suffer. It seems that it is not a good idea to mix the history of store instructions belonging to different nodes.

5.4.3 History depth and field size. All top performers utilize deep history predictors. However, history depth has different effects on different types of predictors.

- For intersection prediction, increasing history depth worsens sensitivity and improves PVP for all indexing schemes (Figure 9). Deep history causes the intersection function to predict less sharing but more accurately.
- For union prediction the opposite occurs. Deep history causes the union function to predict more sharing but with less accuracy. Thus, sensitivity improves with history depth while PVP worsens.
- For PAs predictors, history defines the resolution of the predictor. Deeper history allows a PAs predictor to detect more complex patterns (given enough learning time). Unfortunately, our benchmarks do not generate many complex patterns and when they do, there are not enough events to let the deep-history PAs predictors learn and preform to their potential. Thus, in Figure 9 we see practically no difference for sensitivity and PVP as we increase the history depth from 2 to 4 for the PAs predictors.

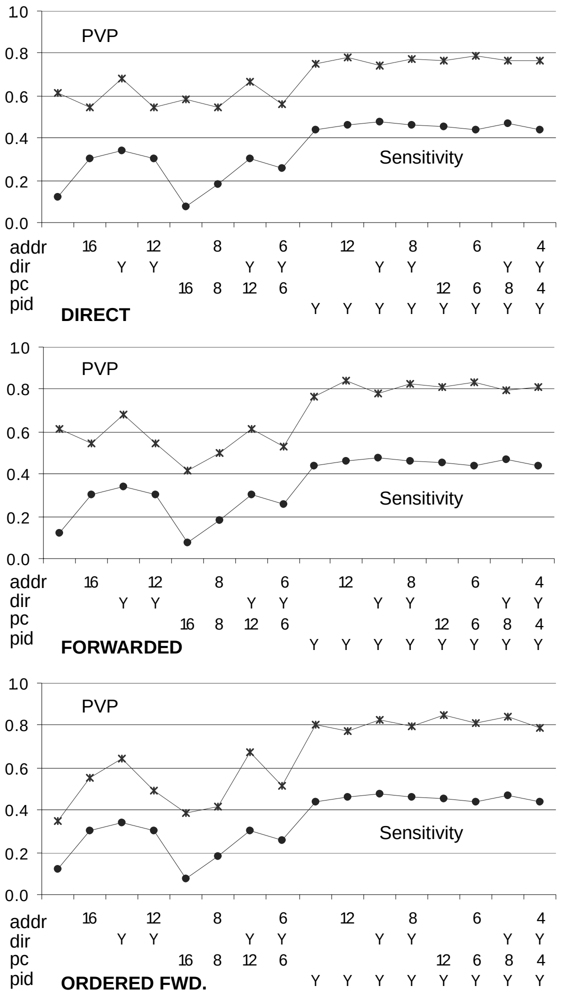

Figure 6. Intersection prediction (history depth 2, 16-bit max index) with direct, forwarded, and ordered update. Labels denote bits used in indexing fields

Although we always use *pid* and *dir* in their entirety (4 bits for a 16-node system) we truncate the *addr* field and the *pc* field to fit our maximum index size or to obtain smaller predictors. This results in a large number of cases depending on

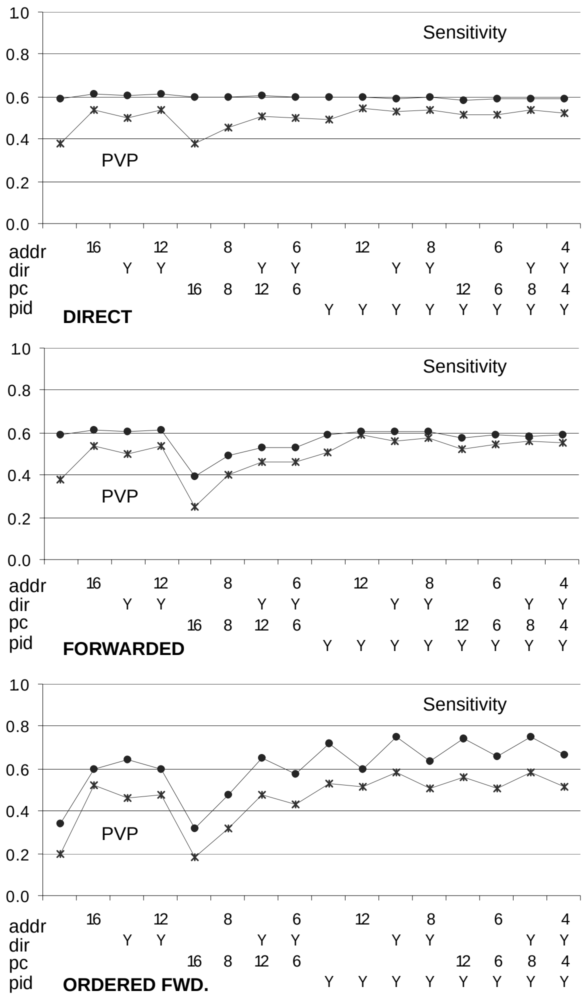

Figure 7. Union prediction (history depth 2, 16-bit max index) with direct, forwarded, and ordered update

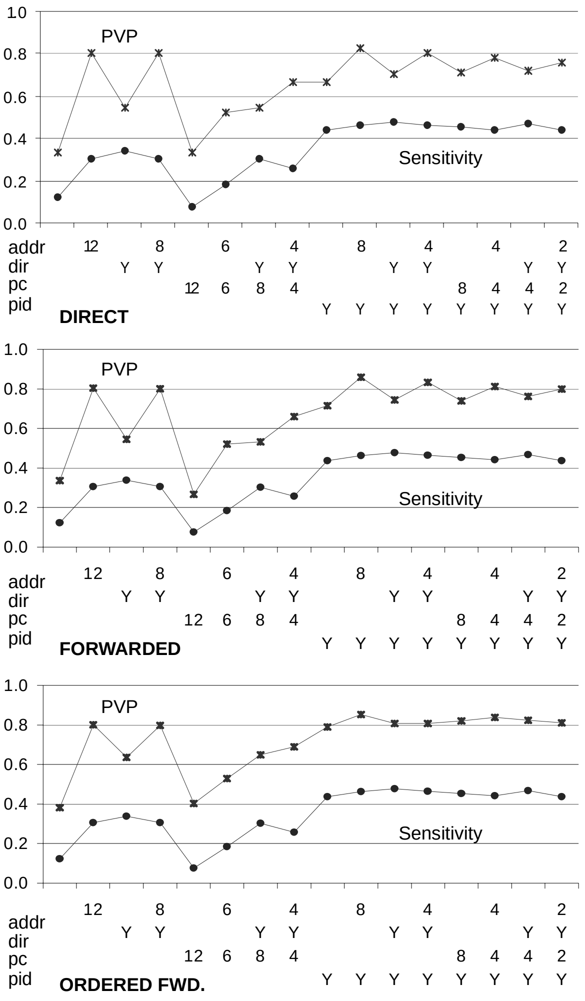

Figure 8. PAs prediction (history depth 1, 12-bit max index) with direct, forwarded, and ordered update

the number of bits we chose to include from the *addr* and *pc* fields. Because we have few store instructions it follows that prediction accuracy does not change significantly with the size of the *pc* field. However, because we have many more addresses to deal with, prediction accuracy is a little more sensitive to the size of the *addr* field. Again, for intersection and PAs prediction, sensitivity increases and PVP decreases with larger *addr* fields. The opposite holds for union prediction.

6 Summary

We made three major contributions in this paper: we presented a taxonomy that encompasses prior shared-memory producer-consumer prediction schemes (and can be expanded to include new ones), we described a statistical model for analyzing the results of shared memory prediction schemes, and we presented empirical results that explore the space of affordable implementations.

In completing the taxonomy, we discovered and described a space of schemes that includes past address-based and instruction-based predictions as points. While our taxonomy is not exhaustive, it suggests a useful way to think about sharing prediction schemes: as different ways to index predictors. It is also key that placing prediction tables at the directory or at the processor implicitly indexes into a global prediction abstraction. By exploring different update strategies (direct, forwarded, ordered) in our taxonomy, we were able to look at how useful some heuristics are and to use ordered update to show one practical upper bound on the predictive abilities of currently-known techniques. New update mechanisms, new prediction functions, and new indexing inputs can be incorporated into our model as they prove to be useful. The space of

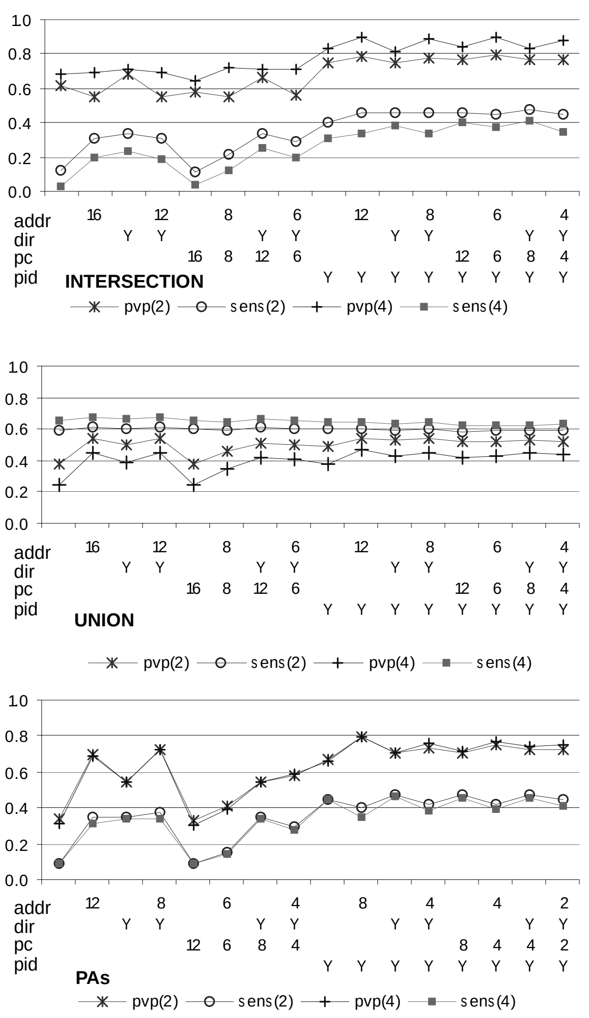

Figure 9. Direct update, intersection, union and PAs predictors with history depths of 2 and 4

schemes provided the basis for our third contribution, the empirical study.

By transplanting screening-test statistics into our field, we introduced a number of metrics that shed light on the sharing prediction problem. All prior studies focused on PVP, an admittedly useful metric that misses the big picture of how much sharing opportunity was not captured. Using both sensitivity and PVP to measure the efficacy of sharing prediction schemes allowed us to explore schemes that trade them off. Using these results we can focus on optimizing either or both of sensitivity and PVP, as the bandwidth and cost constraints of their designs require. And understanding that prevalence, the amount of truly dependent sharing in a multiprocessor application, is the true upper bound on the benefits due to sharing prediction, tells us how much we have captured and how far we still have to go.

Lastly, our empirical study explored the space of possible predictor designs. We showed the relative value of different indexing components: *Pid* and history *depth* are paramount, while *addr* has some value and *dir* and *pc* have the least value as index components for our update mechanisms and prediction functions. We were surprised to find a lack of correlated sharing patterns in the SPLASH benchmarks: two-level adaptive schemes such as PAs were unable to exploit such patterns to improve prediction accuracy.

Achieving the best overall performance depends on a number of system-related factors. Schemes with high PVP measurements will make only the most sure bets: when they generate data-forwarding traffic it is very likely to pay off. On a machine with a very busy communications network, only sure bets should be made to improve performance. However, high-PVP schemes suffer from low sensitivity: they miss many opportunities for sharing.

In contrast, the most sensitive schemes in our study are high-depth union schemes. Such schemes achieve high sensitivity at the cost of lower PVP: they win more often in absolute terms, but they also generate more extra traffic that does not pay off. This suggests a bandwidth-latency trade-off: with more communications network bandwidth, we could use a higher-sensitivity (and therefore higher data-forwarding traffic) predictor. Since latency is the crux of multiprocessor performance [25], finding ways to eliminate latency while using more bandwidth seems like a profitable way to increase overall performance.

7 Acknowledgments

We would like to thank Michel Dubois for his help in producing the final version of this paper as well as James R. Goodman, Mark D. Hill, Michael D. Smith, and the anonymous reviewers for their helpful comments.

8 References

[1] A. Agarwal, M. Horowitz and J. Hennessy, "An evaluation of Directory schemes for Cache Coherence." *15th ISCA*, June 1988.

[2] A.O. Allen, "Probability, Statistics, and Queueing Theory with Computer Science Applications." Academic Press, Harcourt Brace Jovanovich, Boston, 1990.

[3] P. Bannon, "Alpha 21364: A Scalable Single-chip SMP." Microprocessor Forum, Oct 1998. `www.digital.com/alphaoem/present/index.htm`

[4] E.E. Bilir, R.M. Dickson, Y. Hu, M. Plakal, D.J. Sorin, M.D. Hill, and D.A. Wood, "Multicast Snooping: A New Coherence Method Using a Multicast Address Network." *26th ISCA*, May 1999.

[5] L.M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems." *IEEE Trans. Computers*, 27(12):1112-1118, Dec. 1978.

[6] G. Chrysos, J. Emer, "Memory Dependence Prediction using Store Sets." *25th ISCA*, June-July 1998.

[7] Convex Computer Corp., "The Exemplar System" 1994.

[8] F. Dahlgren, P. Stenström, "Reducing the Write Traffic for a Hybrid Cache Protocol." *ICPP*, Aug. 1994

[9] B. Falsafi et al., "Application-Specific Protocols for User-Level Shared Memory." *Supercomputing '94*, Nov. 1994.

[10] J. Gastwirth, "The Statistical Precision of Medical Screening Procedures: Application to Polygraph and AIDS Antibodies Test Data." *Statistical Science*, 2(3), Aug. 1987.

[11] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun, "Confidence Estimation for Speculation Control." *25th ISCA*, June-July 1998.

[12] IEEE Scalable Coherent Interface Standard 1256, 1992.

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

[14] S. Kaxiras and J.R. Goodman, "Improving CC-NUMA Performance Using Instruction-based Prediction." *5th HPCA*, Jan. 1999.

[15] D. A. Koufaty et al, "Data Forwarding in Scalable Shared-Memory Multiprocessors." *ICS*, July 1995

[16] A. Lai, B. Falsafi "Memory Sharing Predictor: The Key to a Speculative Coherent DSM." *26th ISCA*, May 1999.

[17] J. Laudon, D. Lenoski. "The SGI Origin: A cc-NUMA Highly Scalable Server." *24th ISCA*, June 1997.

[18] D. Lenoski et al., "The Stanford DASH Multiprocessor." *IEEE Computer* 25(3), March 1992.

[19] J. Lee and A. Smith. "Branch Prediction Strategies and Branch Target Buffer Design." *Computer*, 17(1):6-22, Jan. 1984.

[20] M. H. Lipasti, C. B. Wilkerson and J. P. Shen "Value Locality and Load Value Prediction," *ASPLOS VII*, Oct. 1996.

[21] T. Lovett and R. Clapp, "STiNG: A CC-NUMA Computer System for the Commercial Marketplace." *23rd ISCA*, May 1996.

[22] N. Mirghafori, M. Jacoby, and D. Patterson, "Truth in SPEC Benchmarks." *Computer Architecture News* 23(5):34-42, Dec. 1995.

[23] A. Moshovos at al., "Dynamic Speculation and Synchronization of Data Dependences." *24th ISCA*, June 1997.

[24] S.S. Mukherjee and M.D. Hill "Using Prediction to Accelerate Coherence Protocols." *25th ISCA*, June-July 1998.

[25] V. S. Pai, P.Ranganathan, and S. V. Adve "The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology." *3rd HPCA*, Feb. 1997.

[26] J.P. Singh, W-D. Weber, A. Gupta. "SPLASH: Stanford Parallel Applications for Shared Memory." *Computer Architecture News*, 20(1):5–44, March 1992.

[27] J.E. Smith, "A Study of Branch Prediction Strategies." *8th ISCA*, 1981.

[28] W.D. Weber and A. Gupta, "Analysis of Cache Invalidation Patterns in Multiprocessors." *ASPLOS III*, April 1989.

[29] T. Yeh and Y. Patt. "Two-Level Adaptive Branch Prediction," *Proc. 24th Annual ACM/IEEE Intl. Symp. and Workshop on Microarchitecture*. Los Alamitos, Calif.: IEEE Computer Society, Nov. 1991.

scheme	size	prev	pvp	sens
inter(pid+add$_6$)^4	16	0.10	0.93	0.32
inter(pid+pc$_2$+add$_6$)^4	18	0.10	0.92	0.34
inter(pid+add$_8$)^4	18	0.10	0.92	0.32
inter(pid+pc$_4$+add$_6$)^4	20	0.10	0.91	0.36
inter(pid+add$_{10}$)^4	20	0.10	0.91	0.33
inter(pid+pc$_2$+add$_8$)^4	20	0.10	0.91	0.33
inter(pid+add$_4$)^4	14	0.10	0.90	0.32
inter(pid+pc$_6$+add$_6$)^4	22	0.10	0.90	0.37
inter(pid+add$_8$)^3	18	0.10	0.90	0.36
inter(pid+pc$_4$+add$_4$)^4	18	0.10	0.90	0.36

Table 8: Top 10 PVP, direct update

scheme	size	prev	pvp	sens
inter(pid+pc$_8$+add$_6$)^4	24	0.10	0.94	0.36
inter(pid+pc$_6$+add$_6$)^4	22	0.10	0.94	0.36
inter(pid+pc$_6$+dir+add$_4$)^4	24	0.10	0.94	0.34
inter(pid+pc$_{10}$+add$_4$)^4	24	0.10	0.93	0.37
inter(pid+pc$_4$+dir+add$_4$)^4	22	0.10	0.93	0.34
inter(pid+pc$_4$+add$_6$)^4	20	0.10	0.93	0.35
inter(pid+pc$_6$+add$_8$)^4	24	0.10	0.93	0.35
inter(pid+pc$_8$+add$_4$)^4	22	0.10	0.93	0.36
inter(pid+pc$_4$+dir+add$_6$)^4	24	0.10	0.93	0.33
inter(pid+pc$_6$+add$_4$)^4	20	0.10	0.93	0.36

Table 9: Top 10 PVP, forwarded update

scheme	size	prev	pvp	sens
union(dir+add$_{14}$)^4	24	0.10	0.47	0.68
union(add$_{16}$)^4	22	0.10	0.45	0.67
union(dir+add$_{12}$)^4	22	0.10	0.45	0.67
union(dir+add$_{10}$)^4	20	0.10	0.42	0.67
union(dir+add$_2$)^4	12	0.10	0.39	0.67
union(dir+add$_8$)^4	18	0.10	0.41	0.67
union(pc$_2$+dir+add$_6$)^4	18	0.10	0.39	0.67
union(add$_{14}$)^4	20	0.10	0.42	0.67
union(pc$_4$+dir)^4	14	0.10	0.40	0.66
union(pc$_2$+dir+add$_2$)^4	14	0.10	0.40	0.66

Table 10: Top 10 sensitivity, direct update

scheme	size	prev	pvp	sens
union(dir+add$_{14}$)^4	24	0.10	0.47	0.68
union(pid+dir+add$_4$)^4	18	0.10	0.46	0.68
union(pid+dir+add$_2$)^4	16	0.10	0.46	0.68
union(add$_{16}$)^4	22	0.10	0.45	0.67
union(dir+add$_{12}$)^4	22	0.10	0.45	0.67
union(dir+add$_{10}$)^4	20	0.10	0.42	0.67
union(dir+add$_2$)^4	12	0.10	0.39	0.67
union(pid+dir+add$_6$)^4	20	0.10	0.47	0.67
union(dir+add$_8$)^4	18	0.10	0.41	0.67
union(pid+add$_6$)^4	16	0.10	0.43	0.67

Table 11: Top 10 sensitivity, forwarded update