The Coherence Predictor Cache: A Resource-Efficient and Accurate Coherence Prediction Infrastructure

Jim Nilsson Anders Landin[†] Per Stenström

Department of Computer Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
{j, pers}@ce.chalmers.se

[†]Sun Microsystems
901 San Antonio Rd,
Palo Alto, CA 94303, USA
anders.landin@sun.com

Abstract

Two-level coherence predictors have shown great promise to reduce coherence overhead in shared memory multiprocessors. However, to be accurate they require a memory overhead that on e.g. a 64-processor machine can be as high as 50%.

Based on an application case study consisting of seven applications from SPLASH-2, a first observation made in this paper is that memory blocks subject to coherence activities usually constitute only a small fraction (around 10%) of the entire application footprint. Based on this, we contribute with a new class of resource-efficient coherence predictors that is organized as a cache attached to each memory controller. We show that such a Coherence Predictor Cache (CPC) can provide nearly as effective predictions as if a predictor is associated with every memory block, but needs only 2–7% as many predictors.

1 Introduction

Coherence activities in shared-memory multiprocessors remain an increasingly important bottleneck preventing high performance. In write-invalidate protocols, coherence misses as well as invalidations may take hundreds of cycles. Obviously, there has been a lot of research that targets this performance deficiency.

Early work on reducing coherence overhead attacked common static sharing patterns, such as migratory, producer-consumer, and wide sharing with hardware optimizations [3, 5, 9, 13, 21], or with software/compiler approaches [7, 20]. Three properties limit these approaches: (i) they target only simple and supposedly a minor part of the possible coherence message signatures present in a cache-coherent multiprocessor; (ii) they are static in the sense that they are specialized for a certain coherence message signature; and, (iii) they make the cache coherence protocol more complex and specialized for the targeted signatures.

As a remedy, Mukherjee and Hill [17] adapted the classical two-level branch prediction scheme [24] to record global coherence events, e.g., read miss, write miss, and upgrade requests. Based on the past sequence of such events to a memory block, they showed that one can predict the next event with a high accuracy. Such predictions were demonstrated to avoid coherence overhead for common access patterns such as migratory and producer-consumer sharing in addition to wide sharing [8]. Even if Lai and Falsafi in their VMSP proposal [11] improved on the first generation of coherence predictors by coding the history information more densely, the memory overhead for a predictor of depth four can range between 15–50% for machine sizes between 16 and 64 nodes, independent of the total memory capacity.

In this paper, we seek a more resource-efficient coherence predictor infrastructure than previous *memory-wide* coherence predictors. A key observation we exploit in the paper is that coherence activities are confined to a small fraction (typically less than 10%) of the entire data set footprint. Additionally, this 'coherence footprint' typically exhibits a substantial reuse making a cache that only hosts about 5% of the entire number of blocks contained in the data set exhibit a capacity miss ratio that is less than 7%. We made this observation by analyzing the coherence activity in seven applications from SPLASH-2.

The main contribution is the design and evaluation of a new class of coherence prediction schemes called the *Coherence Predictor Cache* (CPC). This cache sits at each memory controller in a distributed shared memory multiprocessor and dynamically associates a predictor with only such blocks that are subject to coherence activities.

A perfect CPC should maintain predictors for all blocks that are subject to coherence activity. Then, given that the CPC has a total of C blocks out of M memory blocks, the number of predictors in a memory-wide predictor is reduced by a *hardware reduction factor* of M/C. Although the total amount of memory blocks is the union between shared and private data, we make the pessimistic assumption that M equals the maximum of the total application footprint and the total size of the shared data set. Unfortunately, owing to its limited size, the CPC will suffer from predictor-entry replacements which reduce the number of successful predictions made. Thus, there is a tradeoff between the hardware reduction factor and the *coverage*, i.e., the ratio between successful predictions in the CPC and the memory-wide (one predictor per memory block) predictor. We explore the design space of CPCs and conduct a detailed analysis of how the organizational parameters affect its performance.

Based on seven applications from the SPLASH-2 suite, the most important finding is that the CPC makes close to

90% as many successful predictions as the memory-wide predictor but requires between 14–47 times fewer predictors as the memory-wide predictor. Thus, using a CPC may make coherence prediction cost-effective.

The next section introduces the architectural framework. We then study the nature of the footprint and working sets for blocks subject to coherence activities in Section 3. This justifies our proposed CPC infrastructure whose design principles are presented in Section 4. Section 5 evaluates the performance of the CPC. Finally, we conclude in Section 6.

2 Background

2.1 Architectural Framework

Our baseline is a cache-coherent NUMA multiprocessor. Each node contains a processor, two levels of cache, and a memory module with a directory controller implementing a straight-forward write-invalidate protocol such as the DASH protocol [15] which works as follows.

There is a home node for each memory block in which the corresponding physical page is mapped. The home directory keeps track of the sharing set of each of its designated memory blocks by a presence flag vector with one bit for each node and one of two memory states, clean and dirty, designating whether the memory copy is up-to-date or not. The directory protocol conforms to an MSI-protocol, i.e., a cache copy can be in the M(odified), S(hared), or I(nvalid) state. On a read or write miss, a *Read* or *Write* message is sent to the home, that depending on the state of the memory copy returns an up-to-date copy from either the memory or a remote cache. Additionally, a *Write* message also invalidates all other copies. On a local write to a block in the Shared state, an *Upgrade* message is sent to home which issues invalidations to remote copies, if any.

The baseline message predictor is the *vector memory sharing predictor* (VMSP); a two-level predictor, as proposed by Lai and Falsafi [11]. VMSP is a generalization of the Cosmos predictor [17] which in turn contributed with applying the two-level *PAp* branch predictor according to Yeh and Patt [24] to coherence message prediction. In contrast to the prediction schemes studied in [11], we focus only on the memory-side predictors which associates a predictor with each memory block that make predictions based on global coherence events, i.e., *Reads*, *Writes*, and *Upgrades*. Let us now review how VMSP works in detail.

Figure 1 displays the architecture of a VMSP predictor. It consists of a *message history table* (MHT) that associates a history recording of coherence messages with each memory block. The history patterns are implemented as statically allocated shift-register queues, one for every memory block. On every access to a home memory block, the queue is shifted and the oldest queue element is discarded, while the new coherence message is shifted in. History information is used to index a *pattern history table* (PHT), containing predictions of the coherence messages following a particular history of coherence events. As in [11], we assume that the MHT is a structure with a fixed size with one entry per memory block, whereas the PHT entries are allocated dynamically when needed. The issue of deallocation has not been addressed by previous work. The total memory overhead of the predictor is thus composed of a *static part*, the MHT, and a *dynamic part*, the PHT.

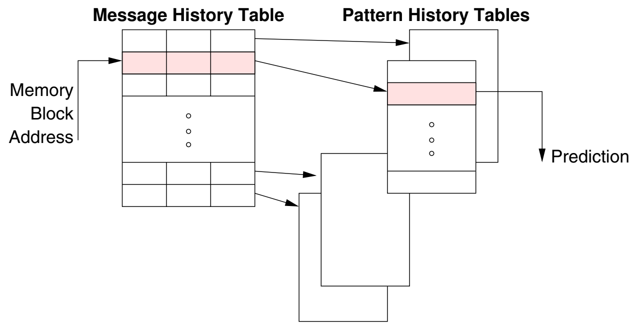


Figure 1. Baseline VMSP predictor architecture.

The MHT records global coherence actions to specific memory blocks and keeps information on the last *h* (history depth) messages. A message is stored as a tuple (*type*, **P**), where *type* is *Read*, *Write*, or *Upgrade*, and **P** is a vector of reader processors or a single writer (upgrade) processor. When the history register has been filled up, the next incoming message to a memory block allocates a pattern entry which becomes the prediction the next time the same history of messages reaches the block. If the predicted message differs from the incoming, the incoming message replaces the prediction.

2.2 Evaluation Methodology

We model a CC-NUMA architecture consisting of 16 nodes where each node contains a processor, a two-level cache hierarchy whose sizes are 64 kB and 1 MB, respectively. The block size is 64 bytes, the L1 caches are four way set-associative and the L2 caches are direct-mapped. We assume that memory pages, 8 KBytes each, are allocated round-robin among the nodes.

We use a processor model which is single-issue and is modeled using the Simics infrastructure [16]. Since we primarily focus on reduction of coherence miss rate, such a simple processor model is adequate. This is because under strict memory consistency models, such as TSO [22], there are limited opportunities to reorder memory operations as more aggressive processor models would enable.

We use seven applications from SPLASH-2 [23] to drive our experiments, with measurements confined to the parallel section of the programs. The particular input data set used and some relevant statistics for these applications are displayed in Table 1.

All applications were compiled using gcc 3.0 with -O2 optimization and parallelized using a pthread implementation of the ANL macros. Solaris 8 was the operating system used in all simulations and no distinction was made between memory references coming from the application or the kernel. The software interface conforms to an UltraSparc II based machine from Sun Microsystems, except for the memory system which is modeled according to the description above.

3 Coherence Block Footprints and Locality

3.1 Memory Overhead of Coherence Predictors

The memory overhead of currently proposed correlation-based coherence message predictors consists of a static part associated with the message history table and a dynamic part associated with the pattern history table. The size of

Application	Input data	Workload statistics ($\times 10^6$) #instructions	#mem refs	Shared data set size kBytes	64 B Lines	Footprint 64 B lines	8 k pages
Barnes	4 k particles	8,243	2,286	24,273	388,365	43,095	1,236
Cholesky	*tk16.O*	1,221	357	7,140	358,101	207,936	2,807
FMM	4 k particles	1,468	442	22,381	114,232	25,122	552
LU	512 x 512 matrix	1,950	504	2,185	34,957	41,448	940
Radix	64 k keys	152	49	10,946	175,140	51,741	758
Volrend	*head4*	1,576	369	501	8,021	26,855	1,097
Water	1024 molecules	16,344	4,408	888	14,212	58,672	1,203

Table 1. Benchmarks, baseline input data set sizes, and basic statistics.

the static part is proportional to the number of memory blocks, number of processors/nodes, and the history depth. The number of pattern history tables is proportional to the number of memory blocks. The size of each table depends however on application behavior. In the following we will focus on the static part.

The system organization has a strong influence on the memory overhead and the key component of interest is the number of coherent nodes involved. A coherent node is defined as the smallest entity between which some, perhaps unified, cache is maintained coherent; in our case, the second-level cache in each node.

Figure 2 shows the static memory overhead of VMSP as a function of history depth and the number of nodes. Each curve is calculated from the formula: $Overhead = \frac{(N+2)\cdot h}{8\cdot B} \cdot 100\%$, N being the number of coherent nodes, h the history depth, and B the cache block size (64 bytes throughout this paper). The term two in the numerator comes from the two bits needed to store the access type (read, write, or upgrade). The denominator is introduced to adjust for the block size (in bytes). In their study, Lai and Falsafi [11] found that the prediction depth has a significant and positive impact on the accuracy of the prediction, i.e., the fraction of predictions that are correct. While the impact of prediction depth is outside the scope of this paper, we see that the memory overhead is between 15 and 50% for machines with 16–64 nodes assuming a history depth of four. Note that this overhead does not account for the dynamic overhead.

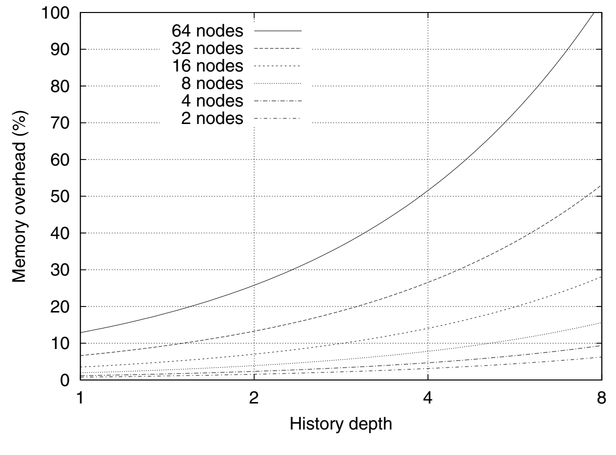


Figure 2. VMSP history information memory overhead as a function of history depth for varying number of nodes.

Even though this cost might be tolerable, it becomes a significant part of total system cost. With a history depth of eight, the memory overhead becomes more than 50% for machines with 32 nodes or more. Cosmos and MSP have a slightly smaller memory overhead, but their resulting prediction accuracies become notably lower [12] due to less efficient encoding of coherence events. Evidently, memory overhead could be reduced if history information is only stored for a minor part of the total memory. Or more specifically, which is our hypothesis, for the memory blocks that experience coherence misses.

3.2 Coherence Footprint and Locality

In the following experiment, we filtered out the memory accesses that arose due to coherence misses in the second-level cache according to the miss classification in [4]. In Figure 3, we show a histogram of how these accesses were distributed over the total data footprint of each application, averaged over all nodes. On the x-axis, unique memory addresses are sorted in decreasing order of the number of accesses to that address. The number of coherence miss requests are plotted on the y-axis.

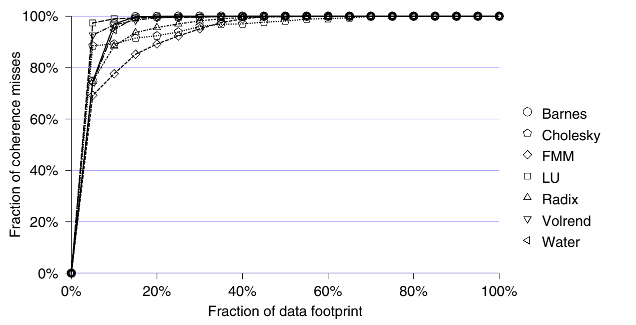


Figure 3. Distribution of fraction of coherence miss requests across the data footprint.

We can see that the majority of coherence misses are confined to a fairly small part of the entire footprint; for all applications but FMM, 95% or more of all coherence misses go to less than 10% of the total footprint. In fact, for Cholesky, LU, and Volrend, only 5% of the footprint is subject to 90% or more of all coherence misses. For FMM, about 78% of all coherence misses target 10% of the footprint. Unfortunately, these numbers say nothing about the timing of coherence misses; the temporal locality of coherence accesses can be even better than suggested by Figure 3.

To understand whether coherence footprints also are subject to reuse, we implemented a cache attached to each memory module that only caches memory blocks subject

Application	Cache entries	Ratio
Barnes	512	47.4
Cholesky	512	43.7
FMM	512	13.9
LU	64	40.5
Radix	512	21.4
Volrend	64	26.2
Water	256	14.3

Table 2. Coherence block cache sizes (# of entries per node) and their respective *hardware reduction factor*.

to coherence misses, referred to as *coherence blocks*. Such a cache would work similar to a directory cache [6,18]. The major difference is that only blocks that experience at least one coherence miss over their lifetime of the application are cached. We will later in Section 4 explain how such a selection can be implemented.

We measured the miss ratios in a four-way set-associative coherence block cache, with (AF) and without (NAF) restricting allocations to coherence blocks. NAF consequently makes no distinction between memory blocks, while AF only lets blocks that are subject to coherence misses enter the cache. Coherence block cache sizes were chosen to approximately keep the miss ratio below 20%, and details on the coherence block cache sizes and the ratios between cache size and the total memory requirement of the applications are listed in Table 2. For example, a hardware reduction ratio of 47.4 for Barnes means that the coherence block cache covers $\frac{1}{47.4} = 2.1\%$ of the total data set. For LU, Volrend, and Water, the default cache size of 512 entries proved to be unnecessary large, showing almost perfect hit-ratios. The number of cache entries for these applications were consequently reduced in order to provide additional insights for the coherence block cache. For the 16-node systems studied, at a history depth of four and excluding the tag, each entry in the history cache consumes 73 bits = a valid bit plus four times 18 bits (16 nodes plus two type-bits) per history entry. The simulation results are shown in Figure 4. In the figure, NAF numbers correspond to the left bar whereas AF numbers correspond to the right bar, respectively, for each application. The top part of each bar is the amount of capacity misses, while the bottom part is the amount of cold misses, in the coherence block cache.

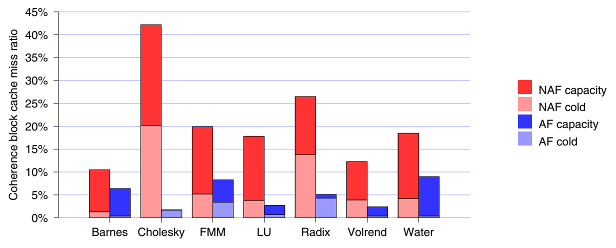


Figure 4. Coherence block cache miss ratios. Left bar: NAF. Right bar: AF

The first observation from this figure is that for some applications, even NAF manages to capture a large part of all blocks subject to coherence misses, with miss ratios below 13% for three applications. However, four applications

Application	Input data	Shared data set size		Ratio
		kBytes	64 B Lines	
Barnes	4 k particles	8,433	134,925	16.5
	16 k particles	33,585	537,357	16.4
	64 k particles	134,193	2,147,085	16.4
FMM	4 k particles	7,140	114,232	13.9
	16 k particles	28,460	455,366	13.9
	64 k particles	116,552	1,821,124	13.9

Table 3. Benchmarks, input data set sizes, and basic statistics, for the scaling experiment. *Ratio* is defined in the text.

(Cholesky, FMM, LU, and Radix) suffer from very high miss ratios, up to 42% for Cholesky. Misses are evenly distributed between cold and capacity misses. Restricting allocation in the cache to coherence blocks (AF), the capacity miss ratio drops dramatically for all applications, yielding miss ratios well below 10%. For two out of seven applications, the capacity miss ratio drops with an order of magnitude compared to NAF.

We also compared a fully associative coherence block cache to the baseline four-way set-associative cache. For two of the applications with the highest amount of remaining capacity misses; Barnes and FMM, the miss ratio was further reduced by 27% and 11%, respectively. For Water, 90% of the capacity misses were removed, reducing the miss ratio with 87%. Higher associativity is clearly desirable for the coherence block cache, which as with processor caches can be traded off for a smaller cache size.

While our analysis suggests that a coherence block cache whose size is typically a small fraction of the data set suffices, it is important to understand whether this also holds for larger data sets. To answer that question, we scaled up the application data sets and the coherence block caches by the same amount for Barnes and FMM. We have not afforded to do these scaling experiments for the other owing to the very time consuming simulations.

The three data set sizes and corresponding coherence cache sizes, denoted *small*, *medium*, and *large*, correspond to a sixteen-fold increase of the default sizes. The respective cache sizes are 512, 2k, and 8k entries per node. See Table 3 for details on data set sizes. The rightmost column in Table 3, *Ratio,* denoting the hardware reduction factor, lists the size ratio between the shared data set and the coherence block cache. Note that the difference for Barnes between these numbers compared to the ratios in Table 1, arises because of the need to limit simulation time for the larger system sizes. As the allocated memory of Barnes increases with the number of time steps, the ratio in Table 3 is smaller than in Table 1.

We measured the capacity and cold miss rates by assuming a perfect address filter by preloading 'infinitely' large L2 caches. Figure 5 shows the distribution of coherence block cache hits and misses as we scale the data set and cache size. The leftmost bar correspond to the default data set and cache size. The data clearly shows that the relative amount of capacity misses in the coherence block cache stay constant as the system is scaled up. The fluctuations of the capacity miss fraction is less than 0.1%. Even for the difficult applications Barnes and FMM, with many capacity misses in the coherence block cache compared to other applications, this gives evidence that as the data set is scaled up, the coherence block cache will perform equally well as

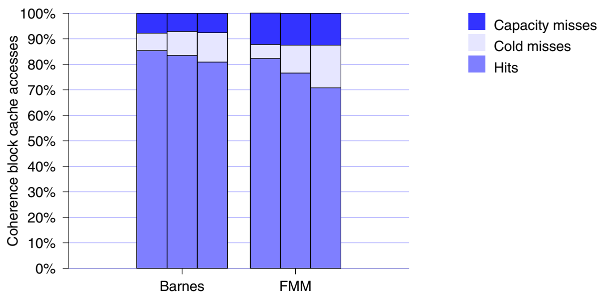


Figure 5. This diagram shows the distribution of cache hits, capacity, and cold misses. From left to right, the three bars correspond to the *small*, *medium*, and *large* configurations, respectively.

for smaller systems.

In this experiment, we have assumed that a user would like to run as big data sets as the memory permits. In practice, this is only feasible if the computational complexity grows as O(N) with the data set size (N). As shown in [19], the computational complexity in many scientific/engineering often grows faster, which suggests that it is not feasible to assume memory-constrained scaling. As a result, we would expect that the hardware-reduction factor is higher in practice than what our data suggests.

4 Coherence Predictor Cache

We now introduce a novel class of coherence message predictors that are based on the observations in the previous section. In essence, the *coherence predictor cache* (or CPC for short) consists of a number of coherence predictors that are associated with coherence blocks in a dynamic fashion according to the cache mapping function. Under the intuition that the capacity miss rate is low, this structure will generate nearly as many successful predictions as associating a coherence predictor with each memory block. A CPC is associated with each memory controller and is accessed in parallel with retrieving directory information. Thus, it should not affect the access time.

The CPC, whose organization is shown in Figure 6, consists of two main components: the history cache with controllers and the pattern memory.

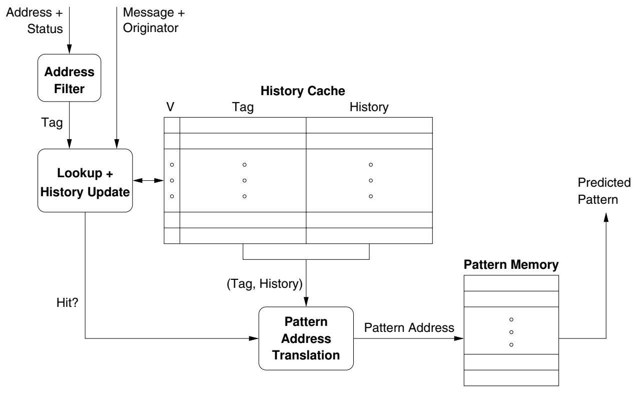


Figure 6. Coherence Predictor Cache architecture.

4.1 CPC Algorithm Overview

The basic operation of the CPC is the following. It intercepts all incoming memory access messages and filters out messages of importance; in our case coherence messages, i.e., Reads, Writes, and Upgrades. It then does a tag lookup in the history cache. If a match is found (the valid-bit is set and the tags match), two cases are possible:

1. All slots in the matched history entry are filled up. The history is retrieved and will be presented to the pattern address translation mechanism to be discussed below.

2. The history entry contains empty slots. The current message is inserted in the history after which there is no further action.

On the other hand, if there is a miss in the history cache, another entry is replaced and the valid bit in the history cache of the new entry is set. The history entry is again implemented as a shift register. Using the outcome of the *pattern address translation*, a prediction of the next coherence message, if available, is retrieved.

We now discuss how the various mechanisms involved can support various prediction scenarios and describe in detail the design choices of the implementation that we later evaluate.

4.2 Implementation

The CPC contains three hardware blocks: the address filter, the lookup and history update, and the pattern address translation mechanism. Below, we describe these mechanisms in more detail.

Address Filter. The *address filter* limits the interference between the memory access messages entering the CPC. One way to implement an address filter that filters out coherence block is to associate with each memory block a *last-writer* identifier. A block is deemed a coherence block if on a cache miss, the reader does not match the last writer. Another alternative is to use hardware counters. Such counters can easily be extended with a capability to count, e.g., coherence misses for memory blocks. This information can be used by the operating system to guide the allocations in the cache, by for instance tagging a page.

We experiment with an address filter that works in a similar fashion to the *last-writer* scheme proposed above, with the exception that it uses the coherence miss classification in [4] to tag a memory block as a *coherence block*. It is only coherence blocks that can be associated with predictors in the CPC. In the measurements in later sections, we make a comparison between using this scheme to a CPC without an address filter.

Lookup and History Update. The tag produced by the address filter is fed into the *lookup mechanism* which, depending on the contents of the history cache and the value of the valid-bit (V), reports a “hit” or “miss” to the *pattern address translation*. In any case, the history information for a block, if present, is updated with the incoming (*message type*, *originator*) tuple. *Originator* is normally a processor identifier, but can in the general case be, e.g., the node identifier, assuming multiple processors per node.

The associativity of the history cache can, of course, range from direct-mapped to fully associative and an interesting issue for associative coherence predictor cache is to consider replacement policies that include run-time information such as recorded prediction accuracy, length of history recording, number of prediction patterns allocated, etc. In this paper, however, we consider an LRU replacement algorithm and a four-way set-associative history cache.

Pattern Address Translation. As patterns are not stored for all tag/history combinations, a *pattern address translation* is needed to correctly find the corresponding prediction pattern in the *pattern memory*.

In the evaluation, we assume a pattern address translation mechanism which is built on a hash structure, where the history information is stored as compact as possible, i.e., with the least amount of bits. Each entry in the hash table is dynamically allocated and contains a copy of the history information index and a prediction. Pattern entries are never deallocated, not even at CPC replacements. The handling of dynamic memory overhead is a topic for future research.

5 Effectiveness of CPC

This section will present three important results for the CPC: (i) how the prediction accuracy is preserved; (ii) how the amount of predictions can be maintained; and finally (iii) some performance aspects.

5.1 Prediction Accuracy

For the CPC to be successful, it should conserve the prediction accuracies obtained with a predictor spanning the whole memory. We therefore investigated the prediction accuracies obtained with a CPC and compared those with the accuracies of a global VMSP predictor which associates a predictor with each memory block. We focus on the accuracy in predicting that a write or upgrade request is followed by read request, called *read accuracy*. This is an interesting metric in that it tells us the precision by which it can be predicted that the next access is a coherence miss.

The results of this investigation are shown in Figure 7, for a history depth of one (left) and four (right), respectively. For each application, four bars corresponding to different predictor organizations are displayed: global predictor (Global) and CPC, with (AF) and without (NAF) an address filter. A comparison between Global-AF and Global-NAF shows almost no difference in accuracy numbers, while the total number of predictions for Global-AF is slightly lower than for Global-NAF because the tagging of a memory block inhibits one prediction per memory block. Refer to Table 2 for information on CPC sizes and corresponding hardware reduction factors.

Overall, Figure 7 indicates that the CPC does not have a significant impact on the accuracy of the predictions. Another observation is that the address filter does not have any significant influence on the prediction accuracy; neither for the global predictor nor for the CPC. Comparing the prediction accuracies for Barnes to those reported in [11], the significantly lower numbers presented in this paper, as explained in Section 2.1, arise because this paper only considers predictors at the memory and not at the processor caches. Another effect is that CPC-NAF actually exhibits slightly higher accuracies than Global-NAF, attributed to the fact that the conflicts in the CPC have a tendency to favor stable blocks with good locality, resulting in fewer, but more accurate, predictions. All in all, both the CPC and the address filter are robust techniques that essentially retain a high read accuracy.

Some anomalies can be noted for Cholesky though, and to a lesser degree for some other applications. Looking at the read accuracy and a history depth of one, the global predictor without an address filter actually has the lowest accuracy of all configurations. A plausible explanation to this, is that because the total amount of L2 cache misses for Cholesky to a large extent consists of cold misses, both in relative and absolute numbers. These L2 cold misses tend to cause a global predictor to produce poor predictions, while instead the CPC experiences cold misses, which do not produce a prediction at all. Support for this explanation can be found in Figure 4, where Cholesky without an address filter has not only a very high capacity miss ratio, but also a high cold miss ratio. In Section 5.2 below, it will become clear that without an address filter (NAF), the global predictor indeed produces significantly more predictions than the CPC. However, this is a transient condition, completely removed with the address filter, with a small impact on prediction accuracy.

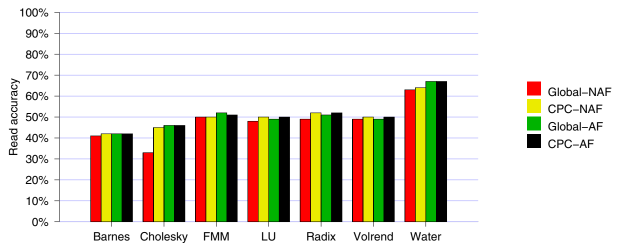
5.2 Prediction Yield and Coverage

The most important metric for the CPC is perhaps not the resulting accuracy for the performed predictions, but rather the accuracy in combination with how many predictions that are actually lost due to CPC misses. We define the *yield* for a CPC as the ratio of the number of successfully predicted reads for the CPC to the number of successfully predicted reads for the memory-wide predictor. We also define *coverage* as the ratio between successful predictions in the CPC and the memory-wide predictor, including write and upgrade predictions. The left diagram in Figure 8 lists the yield for the applications, with and without the address filter and for history depths one and four, while the right diagram shows the corresponding coverage.

Looking at the yield at a history depth of one, all applications except Radix have a yield above 80%. At a history depth of four however, the trend is that yield is reduced, most notably for Barnes. We conjecture that this is attributable to that we use LRU in the CPC, in combination with the relatively large number of remaining capacity misses for these applications. As LRU is unaware of both the length of recorded history for an entry, and the status of any stored prediction information, this replacement mechanism does not necessarily make the optimal choice of which block to evict from the CPC. Higher associativity helps to some extent, but the replacement strategy could probably be improved if taking these factors into account.

Interestingly, AF always outperforms NAF for both depth one and depth four, diminishing the effect of poor replacement decisions by reducing the pollution in the CPC.

The obvious exception in terms of yield is Radix, significantly worse than other applications. To understand the poor yield numbers of Radix, we need to investigate the stream of accesses that enters the CPC. In Figure 9, the distribution of post-L2 accesses are listed, divided into the categories *Reads*, *Writes*, and *Upgrades*. The diagram gives that Radix has much fewer upgrades and a significantly higher fraction of write misses than the other applications, in agreement with previous results [23] showing that Radix communicates values to other nodes through writes, rather than with reads. The many global write actions pollute the CPC, and although the prediction accuracy is maintained, read predictions are traded for write predictions, not affect-



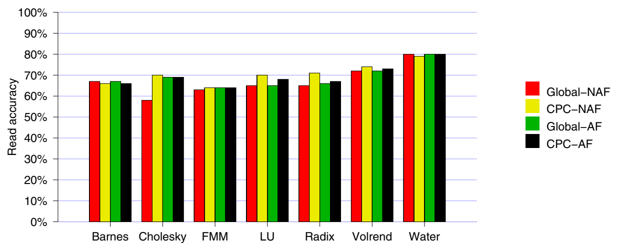
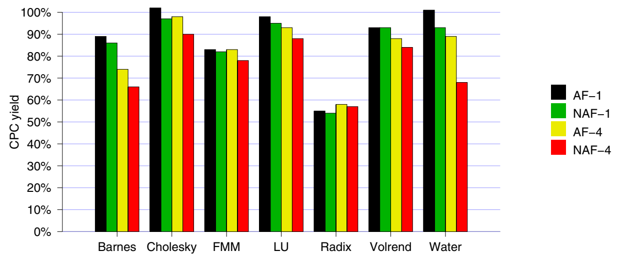


Figure 7. Read accuracy for the CPC with a history depth of one (left) and four (right).



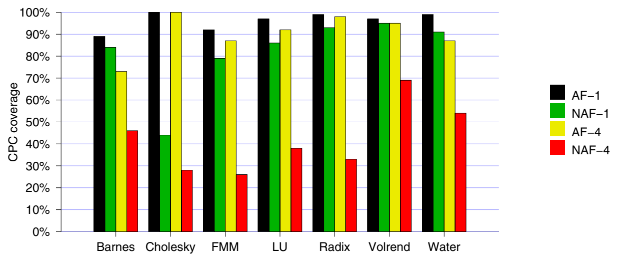


Figure 8. CPC yield (left) and coverage (right).

ing the coherence miss reduction. A possible remedy for this would be to narrow the definition of coherence blocks to blocks that experience a write followed by a read miss from another node. Maybe even restrict history information to only reads and upgrades. The conclusion is that the yield of the CPC seems to be sensitive to workloads with many write misses.

Nonetheless, relating the yield numbers to Table 2 implies that a potentially very high coherence miss reduction is achieved with CPC sizes that only cover a fraction of the total memory of the machine.

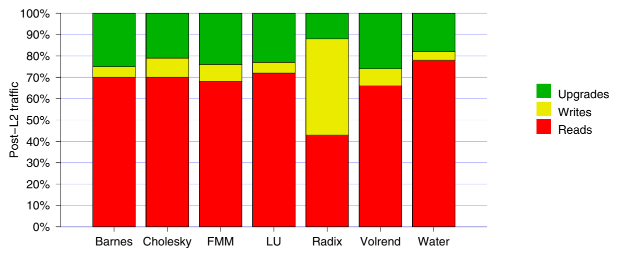


Figure 9. Post-L2 traffic, divided into reads, writes, and upgrades

Turning to coverage (right diagram of Figure 8), the most important result is that a coverage of more than 87% is achieved when using the address filter, for all applications except Barnes at a depth of four. As a longer history is expected to require a block to stay for a longer period of time in the CPC to start producing predictions, a corresponding reduction in both yield and coverage is noted for a history depth of four compared to a history depth of one. For a history depth of one, coverage is above 89% for all applications.

For Cholesky and a history depth of one without an address filter (NAF-1), we again see the effect of many CPC cold misses. Even though the CPC upholds the prediction accuracy of a memory-wide predictor, the dramatic drop in the number of upgrade predictions lower the coverage. The same effect is not seen for the yield since that metric does not include upgrade predictions. Overall, NAF-4 has problems keeping up with the memory-wide predictor in terms of number of predictions, resulting in very low coverage. With the address filter however, coverage is above 70% for all applications, and above 87% for all applications except one. This coverage is achieved with hardware reduction factor of more than 14 for all applications, as shown in Table 2.

As noted in Section 3.2, the associativity of the CPC can affect its miss ratio considerably. The resulting increase in yield and coverage for Water, for which 90% of the capacity misses were removed by going from a four-way set-associative to a fully associative CPC at a history depth of four, was 15% and 14%, respectively. No change in prediction accuracy was noted. The associativity of the CPC is clearly important, but plays a minor role in determining the important metrics of yield and coverage.

Next, we will discuss the bottom line performance potential of the CPC compared to a memory-wide predictor.

5.3 Performance Potential

We now study the potential reduction of coherence misses for a memory-wide prediction scheme and the CPC. To obtain an upper-bound for both schemes, we assume that if a read is correctly predicted, it will remove one coherence miss. While difficult to achieve in practice, it assumes that the last write self-invalidate [12, 14] and ship the block to the reader through data forwarding. [1, 2, 10, 12] that performs a timely delivery of data to the consumers. Under these assumptions, the coherence miss reduction using the CPC for a history depth of four is shown in Figure 10. The diagram displays four bars for each application: the average coherence miss reduction in the second-level processor caches for a global predictor with and without an address filter (Global-AF/NAF), for comparison, and the coherence miss reduction for the CPC with and without an address filter (CPC-AF/NAF).

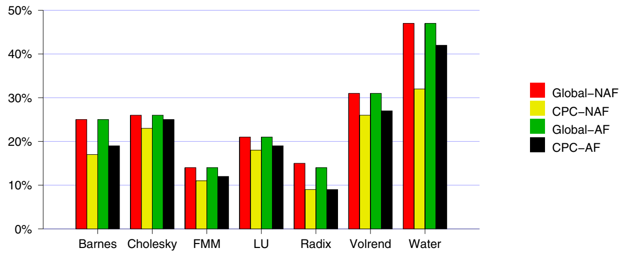


Figure 10. Average coherence miss reduction in second-level caches for a history depth of four.

It should be noted that these numbers are obtained over the entire run of the applications, including cold start effects, which make them a bit pessimistic. Nevertheless, coherence miss reductions in the range 9–42% are achieved for CPC-AF, which is within 8% of the memory-wide predictor for all applications, while only using 2–7% of the amount prediction hardware.

6 Concluding Remarks

We introduced a new class of coherence predictors – coherence predictor caches – that are capable of implementing a wide range of prediction mechanisms. We have shown that by considering cache sizes that are typically only 2–7% of the entire shared data sets, we are able to demonstrate prediction accuracies that are virtually equivalent to memory-wide predictors at a coverage of 87% for six out of seven applications. Finally, we demonstrated that with such a prediction infrastructure, one can potentially remove as much as 20% of the coherence misses, with at most an 8% difference compared to a memory-wide predictor. For Water, about 42% of the coherence misses can be removed with a cache of only 5% the size of the total data set. This study makes an important step towards predictors that can be implemented with an affordable cost.

References

[1] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. In *Proc. of HPCA-3*, pages 204–215, Feb. 1997.

[2] G. T. Byrd and M. J. Flynn. Producer-Consumer Communication in Distributed Shared Memory Multiprocessors. *Proc. of the IEEE*, 87(3):456–466, Mar. 1999.

[3] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proc. of ISCA-20*, pages 98–108, 1993.

[4] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proc. of ISCA-20*, pages 88–97, 1993.

[5] H. Grahn and P. Stenström. Evaluation of a Competitive-Update Cache Coherence Protocol with Migratory Data Detection. *Journal of Parallel and Distributed Computing*, 39(2):168–180, Dec. 1996.

[6] A. Gupta, W.-D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *Proc. of ICPP'90*, pages 312–321, 1990.

[7] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *IEEE Trans. on Computer Systems*, 11(4):300–318, Nov. 1993.

[8] S. Kaxiras. *Identification and Optimization of Sharing Patterns for Scalable Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin-Madison, 1998.

[9] S. Kaxiras and J. R. Goodman. Improving CC-NUMA Performance Using Instruction-Based Prediction. In *Proc. of HPCA-5*, pages 161–170, Jan. 1999.

[10] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-Memory Multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 7(12):1250–1264, Dec. 1996.

[11] A.-C. Lai and B. Falsafi. Memory Sharing Predictior: The Key to a Speculative Coherent DSM. In *Proc. of ISCA-26*, May 1999.

[12] A.-C. Lai and B. Falsafi. Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction. In *Proc. of ISCA-27*, pages 139–148, June 2000.

[13] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proc. of ISCA-24*, pages 241–251, 1997.

[14] A. R. Lebeck and D. A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proc. of ISCA-22*, pages 48–59, June 1995.

[15] D. E. Lenoski, J. P. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of ISCA-17*, pages 148–159, May 1990.

[16] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. SimICS/sun4m: A Virtual Workstation. In *Proc. of Usenix Annual Technical Conf.*, June 1998.

[17] S. S. Mukherjee and M. D. Hill. Using Prediction to Accelerate Coherence Protocols. In *Proc. of ISCA-25*, May 1998.

[18] B. W. O'Krafka and A. R. Newton. An Empirical Evaluation of Two Memory-Efficient Directory Methods. In *Proc. of ISCA-17*, pages 138–147, 1990.

[19] E. Rothberg, J. P. Singh, and A. Gupta. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. In *Proc. of ISCA-20*, pages 14–25, 1993.

[20] J. Skeppstedt and P. Stenström. Simple Compiler Algorithms to Reduce Ownership Overhead in Cache Coherence Protocols. In *Proc. of ASPLOS-6*, pages 325–337, 1994.

[21] P. Stenström, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proc. of ISCA-20*, pages 109–118, 1993.

[22] L. D. Waever and T. Germond, editors. *The SPARC Architecture manual*. PTR Prentice Hall, 1994.

[23] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of ISCA-22*, pages 24–36, June 1995.

[24] T.-Y. Yeh and Y. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Proc. of ISCA-19*, pages 124–134, 1992.