The Use of Instruction-Based Prediction in Hardware Shared-Memory

Stefanos Kaxiras
University of Wisconsin-Madison
kaxiras@cs.wisc.edu

Abstract— *In this paper we propose* Instruction-based Prediction *as a means to optimize directory-based cache coherent NUMA shared-memory. Instruction-based prediction is based on observing the behavior of load and store instructions in relation to coherent events and predicting their future behavior. Although this technique is well established in the uniprocessor world it has not been widely applied for optimizing transparent shared-memory where prediction —in the form of adaptive cache coherence protocols— is typically address-based. The advantage of this technique is that it requires* very few *hardware resources in the form of very small prediction tables per node. In contrast, address-based prediction typically requires storage proportional to the memory and/or cache size. To show the potential of instruction-based prediction we propose and evaluate four different optimizations: i) a migratory sharing optimization, ii) a wide sharing optimization iii) a pairwise sharing optimization, and iv) a producer-consumer optimization based on speculative execution. With execution-driven simulation and a set of ten benchmarks we show that: i) for the first two optimizations, instruction-based prediction performs comparably to and in some cases outperforms address-based schemes while never using more than 72 (5-byte) entries in any node's prediction table; ii) for pairwise sharing there is no significant benefit over the default pairwise optimization of our base protocol. Finally we provide evidence that the producer-consumer optimization based on speculative execution can yield performance improvements.*

1 Introduction

Hardware-based shared-memory architectures are becoming prominent with the popularity of bus-based symmetric multiprocessors (SMPs). Larger shared-memory machines are also advancing in the marketplace. For economic reasons, larger shared-memory machines are built by connecting SMP nodes with high speed interconnects. Typically, in such architectures a directory-based coherence protocol is employed to maintain *cache coherence* (*CC*) among the SMP nodes. Examples of such architectures include the HP/Convex Exemplar [12] and Sequent STiNG [11] that use Scalable Coherent Interface (SCI) networks and SCI cache coherence [2], and the SGI Origin 2000 [13] that uses a directory-based cache coherence protocol originating in Stanford's DASH multiprocessor [9].

The widespread use of SMPs is an opportunity to promote shared-memory parallel programming to a much larger audience of programmers than ever before. However, for widespread use of shared-memory we need standardization: a single view of shared-memory should be presented regardless of whether the underlying architecture is bus-based or directory-based, SMP-based or cluster-based. This has long been advocated by —among others— Reinhardt, Larus, and Wood [29]. Recently, Hill argued that hardware-based shared-memory should be kept as simple as possible, presenting a sequentially consistent transparent shared-memory model especially to the low-level programmer [23]. Hill argues that speculation could be used to *transparently* offer high performance while preserving programmers' sanity.

Thus, there is compelling reason to examine transparent hardware optimizations. Indeed, many adaptive cache coherence protocols that optimize various sharing patterns at run-time have been proposed: for migratory data [16][17], for pairwise sharing and producer-consumer sharing [2][5], and for widely shared data [8]. Recently Mukherjee and Hill [22] showed that address-based prediction in coherence protocols can be generalized using two-level adaptive predictors —which were proposed in the context of branch prediction by Yeh and Patt [26]. However, it is not clear at this point whether the gains of this generalized address-based prediction outweigh its costs which involve a predictor entry per memory and cache block.

A new proposal: In this work we propose to use *Instruction-based Prediction* to as a general technique to optimize various aspects of hardware shared-memory. The main idea is to examine —at run-time— the behavior of load and store instructions in relation to coherence events. In every node the past behavior of its load and store instructions is stored in a *small* predictor table. Whenever dynamic instances of load and store instructions generate coherence events (such as misses, or write-faults on read-only cache blocks) we consult the predictors for optimization hints. This means that the optimizations affect the behavior of the processor toward the CC-protocol (e.g., on a load-miss the processor may ask for permission to write) in contrast to address-based prediction optimizations that affect the behavior of the CC-protocol toward the processor (e.g, the CC-protocol —on its own— may decide to return a writable block to a processor that asks for a read-only block).

Instruction-based prediction is not new in the uniprocessor world: it is established research and it already appears in commercial processors. Branch prediction is the pioneering instruction-based prediction studied extensively by many researchers including Smith [25] and Yeh and Patt [26]. Abraham et al showed that very few loads are responsible for most cache misses [27] and subsequently Tyson et al proposed instruction-based prediction to selectively bypass the cache for such loads [28]. Gonzalez, Aliagas, and Valero used instruction-based prediction to steer data on caches optimized differently for spatial and temporal locality [18]. Moshovos, Breach, Vijaykumar and Sohi introduced memory dependence prediction [19]. They proposed dependence predictors accessed using the address of memory instructions. Subsequently, Moshovos and Sohi proposed memory optimizations based on dependence predictions [20]. Independently, Tyson and Austin proposed similar memory optimizations [21]. Chen and Baer were the first to bring these techniques in the world of parallel shared-memory architectures by proposing prefetching based on instruction-based prediction [38]. Although we believe that such techniques can be generally applicable (from bus-based cache coherence to software based coherence) we restrict this presentation to level of hardware-based, directory-based coherence [2][32][33] (e.g., CC-NUMA architectures).

The benefits of instruction-based prediction/optimization can be significant:

1. Concise representation of history: Code is much smaller than datasets — static load and stores can be only so many while the dataset can be arbitrarily large— and keeping track of the history of load and store instructions rather than memory blocks and/or cache blocks consumes far fewer resources.
2. A single technique for many optimizations: The technique we propose can be used to optimize several sharing patterns *using a common small predictor structure per node*. In contrast, each address-based prediction scheme is tailored for a specific sharing pattern. Each may require its own states in the coherence protocol and its own storage (usually on a per block basis) for history information. Although Mukherjee and Hill showed how to generalize address-based prediction the issue of excessive storage for history information remains.

However, there are important issues involved with instruction-based prediction in shared-memory:

1. Implementation issues: Instruction-based prediction calls for a tight integration of the processor core and the coherence mechanisms because information from both places is needed in the predictor.
2. Performance issues: Address-based prediction inherently keeps large amounts of history information and in some situations this might be preferable to the “concise” information we can gather regarding load and store instructions. The instruction-based prediction optimizations we examine in this paper perform reasonably well (in many cases outperforming address-based prediction but in others lagging behind).

Contributions of the paper: We propose instruction-based prediction as a general technique to optimize hardware shared-memory architectures. We believe that this technique has the potential to effectively optimize many different aspects of shared-memory using very few hardware resources. To support this claim we propose and evaluate three schemes to *transparently* optimize different sharing patterns. The optimizations affect performance but not correctness. These schemes are intended to provide proof-of-concept and

as such they may not be optimal. We do not claim that they are the only ones, or the best ones. In fact, we expect that with future research on this area more and better instruction-based prediction optimizations will emerge. The three schemes implement the following predictions:

- **Predict whether a load-miss will be followed by a store-write-fault.** This prediction can lead to optimization of migratory sharing patterns. The reasoning is that migratory sharing patterns often generate load-misses closely followed by store-write-faults. The optimization we propose (inspired by the work of Cox and Fowler [16], and of Stenström, Brorsson, and Sandberg [17]) is to convert the coherent read-miss to a coherent write-miss. We examine three variations of this scheme and show that it works well for programs with migratory sharing while requiring no more than 72 entries in any node's prediction table.
- **Predict whether a load will access widely shared data.** We propose and evaluate two schemes to predict whether a load instruction will access widely shared data. The optimization is to convert the coherent read to a special form that is recognized and handled by scalable extensions to our base CC-protocol (SCI) designed to offer scalable reads and writes [6][7][8]. This scheme consistently outperforms an address-based adaptive scheme previously proposed for wide sharing [8] while requiring no more than 56 entries in any node's prediction table.
- **Predict which node is going to consume a value generated by a store (Producer-Consumer prediction).** We examine store instructions that generate write-faults and keep track of the potential readers of the newly written cache-blocks using very few resources. Using simple predictors, we can predict upon seeing a store-write-fault, whether there is a stable producer-consumer relation. Furthermore, using a more advanced predictor structure we can predict the identity of the consumer(s). There are three degrees of optimization (from conservative to aggressive): i) Using a simple predictor we can initiate pairwise sharing with direct cache-to-cache transfers without involving the home directory, ii) alternatively we can switch to an update protocol —but this is not transparent in the case of a sequential consistent memory system—, and iii) using enhanced predictors we can *speculatively pre-send* the newly created values to the predicted consumers who can use these values *speculatively* at miss time but they have to *verify* them through the normal cache coherence protocol. In this paper we explore the first and third —most aggressive— optimization (see Section 6).

Structure of this paper: Section 2 discusses in general instruction prediction in shared-memory. Section 3 discusses our evaluation methodology and in particular the WWT simulator and the parameters we used, the SCI cache coherence protocol we use as the platform to express and evaluate our ideas, and the benchmark set we use. This section appears early so we can integrate the description of the ideas and their evaluation in the following three sections. In Section 4 we propose and evaluate instruction-based prediction for optimizing migratory sharing patterns. In Section 5 we study optimizations for wide sharing. Section 6 describes instruction-based prediction for pairwise sharing and producer-consumer sharing. Section 7 wraps up this work.

Nomenclature: In this paper we use the following naming conventions: *load*, *store* are the actual instructions; *read*, *write* are the cache coherence actions resulting from loads and stores. A cache block that is not *Invalid* can be either *ReadOnly* (*RO*) or *ReadWrite* (*RW*). A load or store can experience a cache *miss* which results in a coherent read or write; furthermore a store can experience a *write fault* on a RO cache block which results in a coherent write.

2 Using instruction-based prediction in shared-memory

In contrast to previous work where various schemes try to learn the coherence history of a data block and then make predictions whenever it is accessed, our approach is based on observing the history of load and store instructions *in relation to coherence events* and make predictions every time a known load or store generates a coherence event. So, contrary to the uniprocessor/serial-program context where predictors are updated and probed continuously with every dynamic instruction instance we *only* update the prediction

history and *only* probe the predictor to retrieve information in the case of a coherent event. Three events are relevant in this paper:

- **Cache miss**: a load or store misses in the cache. At this point we probe the predictor using the PC of the load or store and request information on what to do. The return of the coherent response to the cache miss is also an opportunity to update the predictor.
- **Write fault**: a store accesses a RO cache block. The predictor is probed to retrieve information about the store instruction. Coherence information available at the time of the write fault can also be used to update the predictor.
- **External Cache Read**: another processor or the directory reads (or in some way affects) a cached block. This event is an opportunity to update a predictor with coherence information. This event is not connected directly to an instruction. However, such a connection can be established through the predictor structure (if we store the block address information) or the cache blocks themselves if we tag them with the instruction PC related to most recent coherence event they sustained.

Using these coherent events we can trigger optimizations according to the information we get from the predictors. Since we do not probe or update the predictors continuously, the prediction mechanisms are infrequently accessed. Their latency can be hidden from the critical path since we only need their predictions on events which are of significant latency anyway. Thus, we believe that the predictors are not a potential bottleneck nor add cycles to the critical path.

Restricting predictor probes and updates to coherence events is not without disadvantages. Many times an optimization triggered by a prediction will prevent a future coherence event from happening (e.g., predicting that a write-fault is going to succeed a load-miss and optimizing this situation by bringing in a RW cache block prevents the write-fault from happening). This reduces our ability to confirm the success of the optimization. In this paper we concentrate on prediction on coherence events for two reasons: i) it is a technique whose implementations stand halfway between the processor core and the cache coherence mechanisms —and as such it is a natural point to study first— ii) our tool set is based on direct execution (see Section 3) and does not allow us to observe every dynamic instruction instance but just those that generate coherence events.

That we probe and update the predictors on coherence events calls for a tighter integration of the processor core and the cache coherence mechanisms implemented at the coherent cache. In particular our technique requires that both the PC of an instruction that generates a coherent event (e.g., miss, write fault, etc.) and information from the cache coherence mechanisms be available to the predictors. If there is a boundary that separates the processor core from the CC-mechanisms we can accomplish the convergence of all necessary information to a single point by either:

- incorporating the predictor into the processor core and establishing a channel through which the CC-mechanisms can supply feedback information
- or, incorporating the predictor with the CC-mechanisms and establishing a channel through which the processor can supply the PC of the “faulting” instructions to the predictors.

If on the other hand the coherent cache and the processor core are on the same chip then implementing instruction-based prediction will not be difficult: both the instruction PC and all the coherency information are readily available in the same place. In the future, with hundreds of millions of transistors on a single chip, we may see devices that are stand-alone CC-NUMA or COMA [31] nodes complete with caches, directories and local memory (first proposed by Saulsbury, Pong, and Nowatzyk [35] in the context of IRAM [34]). These devices would be ideal for implementing instruction-based prediction.

3 Evaluation setup

In this section we describe the simulator, the base coherence protocol, and the benchmarks we use for all evaluation that appears in following sections.

3.1 Wisconsin Wind Tunnel

A detailed study of the methods we propose requires execution driven simulation because of the complex interactions between the program's instructions and the coherence mechanisms. The Wisconsin Wind Tunnel [10] is a well-established tool for evaluating large-scale parallel systems through the use of massive, detailed simulation. It executes target parallel programs at hardware speeds (*direct execution*) without intervention for the common case when there is a hit in the simulated coherent cache. In the case of a miss, the simulator takes control and takes the appropriate actions defined by the simulated protocol. The WWT keeps track of virtual time in processor cycles. The direct execution nature of the WWT poses certain limitations: only instructions that generate coherence events are observable; the coherent caches are blocking; the cache block size must be a power-of-two multiple of the hardware cache block size (in our case 32 bytes); speculative execution is not supported. Despite these limitations our work provides considerable evidence for the potential of the techniques we propose.

3.2 SCI

We have chosen to use SCI as the underlying cache coherence protocol. The various instruction-based prediction schemes we propose are not depended on the specifics of SCI and they can be tailored to other directory-based cache coherence protocols. We chose SCI because it has a rich set of options that can be used to implement optimizations and in addition Kaxiras and Goodman extend it to handle widely shared data [6][7]. On the other hand, we found the complexity of the protocol to impede simplicity in some of the mechanisms we propose. We will point out mechanisms dependent on SCI's idiosyncrasies, but again we believe that our work can be applied to any directory-based CC-protocol. At the very least any CC-protocol can be enhanced to supply to the prediction mechanisms information comparable to that of SCI.

3.3 Hardware parameters

We simulated SCI systems made of readily available components such as SCI rings and workstation nodes. For the evaluation in Section 5 which requires detailed network simulation we have simulated K-ary 2-cube systems (2 dimensions). For the evaluation of Section 4 and Section 6 we simulated a constant latency network. The nodes comprise a processor, an SCI cache, memory, memory directory, and a number of ring interfaces. The processors run at 500MHz and execute one instruction per cycle in the case of a hit in their cache. Each processor is serviced by a 64KB 4-way set-associative cache with a cache line size of either 32 or 64 bytes. The cache size of 64KB is intentionally small to reflect the size of our benchmarks. Processor, memory and network interface communicate through a 166 MHz 64-bit bus. The SCI K-ary N-cube network of rings uses a 500 MHz clock; 16 bits of data can be transferred every clock cycle through every link. We simulate contention throughout the network but messages are never dropped since we assume infinite queues. The constant latency network takes 100 processor cycles to transfer any message.

3.4 Benchmarks

For this study we use ten benchmarks taken from various sources (see Table 1). We will avoid repeating a detailed description of the benchmarks since they have been described in detail in other work [1][14][15][24]. Instead, we discuss why we chose them for this study: We chose the CHOLESKY, MP3D, and PTHOR benchmarks to study our first prediction scheme. These benchmarks have migratory sharing and they were also used by Cox and Fowler [16], and by Stenström, Brorsson, and Sandberg [17]. We use the same input for comparisons. For optimization of wide sharing we use the following benchmarks: GAUSS [7], SPARSE, All Pairs Shortest Path (APSP) and Transitive Closure (TC)[1] and BARNES (taken from the SPLASH benchmark suite [14]). These benchmarks (except BARNES) were used to evaluate scalable

[1] APSP and TC solve classical graph problems using a dynamic-programming formulation based on the Floyd-Warshall [1] algorithm.

extensions to SCI in both static [7] and adaptive flavors [8]. For these benchmarks we use a block size of 64 bytes since this gives better performance for the base case (SCI). Finally, to study the producer-consumer optimizations we use APPBT (from the NAS parallel benchmarks [30]), OCEAN (taken from SPLASH [14]), BARNES, GAUSS and SPARSE. For the first two cases we use "control" benchmarks that do not exhibit the desired sharing patterns to study potential negative effects of the optimizations.

Benchmark	**Input Size**	**Cache size/ Block size**	**Large Scale Sharing**	**Migratory Sharing**	**Prod.-Cons. Sharing**	**Sections in this paper**	**References**
CHOLESKY	bsstk14	64K/32		Yes		4,5	[14][16][17]
MP3D	10K/10 iter	64K/32		Yes		4	[14][16][17][5]
PTHOR	risc	64K/32		Little		4	[14][16][17][5]
GAUSS	512x512	64K/64	Yes (dyn.)		Yes	4,5,6	[4][15][7]
SPARSE	512x512	64K/64	Yes (static)		Yes	5,6	[7]
APSP	256x256	64K/64	Yes (dyn.)			4,5	[4][1][7]
TC	256x256	64K/64	Yes (dyn.)			5	[1][7]
BARNES	4K part.	64K/64	Yes (static)		Yes	4,5,6	[14][7][5]
OCEAN	130x130	64K/32			Yes	4,5,6	[14][5]
APPBT	12x12x12 10it	64K/32			Yes	6	[30][24]

Table 1: Benchmarks used in this paper. For each benchmark we describe the input size, the cache & block size, and prominent sharing patterns. The references point to papers where the benchmarks are described or used in the same way as in this work.

4 Migratory sharing prediction

In this section we describe an instruction-based prediction that can handle migratory sharing patterns. The idea is to detect when a load-miss is followed by a store-write-fault on the same cache block. If such a load/store pair is recurring often we can predict, upon seeing the load-miss, that a write-fault is soon to follow.

Lets examine why this optimization is related to migratory sharing patterns. Migratory data are continuously read-modified-written but each time by a different processor [16][17]. Each processor brings them in its cache as RO cache block, tries to modify them, generates a write fault, converts them to a RW cache block, writes them, and subsequently loses them to another processor that will go through the same cycle. The connection to the instruction-based prediction is straightforward: migratory data are likely to generate load-misses closely followed by store-write-faults.

The optimization we propose is to convert the coherent read to a coherent write ending up with a RW cache block and thus avoiding the write fault. The inspiration is from the adaptive CC-protocols proposed independently by Cox and Fowler [16], and by Stenström, Brorsson, and Sandberg [17]. Both these groups proposed schemes where the directory discovers migratory data and returns RW cache blocks (instead of RO) whenever a new processor reads the data. The performance benefit comes from collapsing two coherent transactions (read and then write) into one. The cost associated with these adaptive protocols is additional storage *per directory entry* to maintain the identity of the last writer [16][17].

A simple-minded implementation of this instruction-based prediction scheme can be fooled by other-than migratory sharing patterns. An example is pairwise sharing where only two processors alternate reading and writing a cache block. Such sharing patterns benefit from different optimizations (such as choosing updates over invalidates). In the following subsections we will discuss in more detail i) a simple prediction scheme that makes no effort to distinguish migratory sharing, ii) a scheme that tries to distinguish migra-

tory sharing using additional coherence information, and iii) a feedback mechanism that tries to validate whether migratory data were actually involved *after* a prediction was made. Finally, we will present the results of our evaluation for seven benchmarks. We found that if a program *does have* migratory sharing patterns the simple prediction scheme works well; in contrast it gets confused by other sharing patterns in programs that *do not have* significant migratory sharing and needs to be augmented with a feedback mechanism to avoid degrading performance.

4.1 Simple predictor

The idea of this scheme is simple: if we observe a load-miss/store-write-fault pattern a few times then every time we encounter the load-miss we will bring a RW cache block to prevent the write-fault. The predictor is a small fully associative table accessed by the load PC. Each predictor entry contains the PC of the load and a small 2-bit saturating counter used to make predictions. Thus, the size of each entry is less than 5 bytes. We only use load PC in the predictor entries. This means that we do not keep track of unique load/store pairs but we lump together all the pairs that have a common load PC. A predictor entry, therefore, can be updated by multiple distinct stores. We examined alternative implementations but we did not find enough evidence for their usefulness.

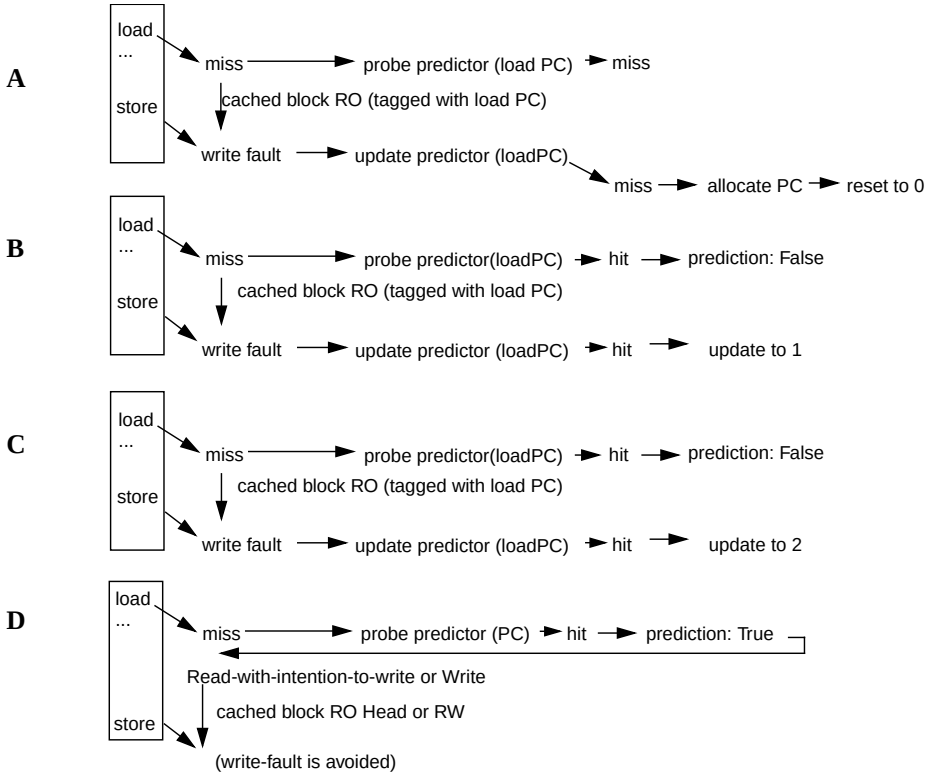

FIGURE 1. Working example of instruction-based prediction for migratory sharing

A working example: Figure 1 shows a detailed example of how the instruction-based prediction works. Each time we encounter a load-miss we probe the predictor (Figure 1A). At first the predictor is empty so we have a predictor miss. The cache block that is brought into the cache as a result of the load-miss is tagged with the PC of the load instruction.[2] This tag serves as the link between load and store instructions. If a store-write-fault occurs for the same cache block we update the predictor using the PC of the load. Since the predictor is empty we allocate a new entry and set the predictor counter to its initial value of zero. The next time the same load generates a new miss we probe the predictor again (Figure 1B). This time we find an entry in the predictor table but the prediction counter has not exceeded the threshold value of one.

Since no optimization is invoked the store that follows generates again a write fault on the RO block. The predictor is updated as before and the relevant counter is incremented. On the third time, the counter is updated once more (Figure 1C). The fourth time we encounter the same load-miss and probe the predictor (Figure 1D) we invoke the optimization (since the prediction counter has exceeded the threshold). The optimization is to use SCI's "read-with-intent-to-write" or alternatively a coherent write to obtain a RW block[3]. The counter threshold is a *tuning* parameter that introduces *hysteresis* in the prediction. The counter is decremented using the enhanced scheme and/or the feedback mechanism described below.

4.2 Enhanced prediction that distinguishes migratory sharing

The simple predictor described above makes no effort to distinguish migratory sharing patterns. Since this may lead to misuse of the optimization, we propose an enhanced version that tries to apply the optimization when there is a high probability of migratory sharing. To understand the enhanced version we must examine a node's view of the information available at the time of the write-fault[4]. This information is used to perform informed updates on the predictor: the goal is to increment the predictor counter when there is high probability of migratory sharing and decrement it otherwise.

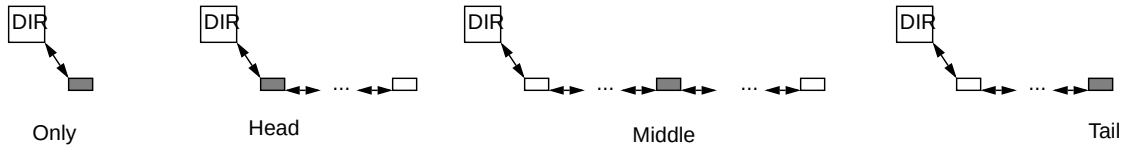

FIGURE 2. Possible position in the SCI sharing list at the time of a write-fault

Description	SCI state	Frequency on write-faults			What the node infers:	Migratory? Update?
		CHOLESKY	MP3D	PTHOR		
only copy, unmodified,RO	ONLY_FRESH	97%	13.3%	74.5%	I am the only reader	Unlikely; ctr - -
head, unmodified, RO	HEAD_FRESH			5.2%	I am the last of a string of readers	Unlikely; ctr - -
head, modified, RO	HEAD_DIRTY	3%	37.3%	15%	Someone wrote the block and I have read its modified copy	Possibly; ctr ++
only copy, modified, RO->RW	ONLY_DIRTY				Like HEAD_DIRTY but all others have left the list. The write fault occured because of the delayed RO to RW transition.	Possibly ; ctr ++
middle, unmodified, RO	MID_COPY				There are multiple readers	Unlikely; ctr - -
middle, modified, RO	MID_VALID		27.0%	1.5%	Someone wrote the block and there are multiple readers	Unlikely; ctr - -
tail, unmodified, RO	TAIL_COPY				There are only readers	Unlikely; ctr - -
tail, modified, RO	TAIL_VALID		22.3%	3.7%	Someone wrote the block (maybe I) and there are readers	Unlikely, ctr - -

Table 2: A node experiences a write fault on a RO cache block. The state of the block determines how the predictor will be updated

At the time of a write fault the cache block can be the "ONLY" cache block on the SCI sharing list, or it

---

2 In reality a separate structure can be used to keep track of the correspondence of addresses to load PCs. This structure needs not be large since the information needs to be kept around for short periods of time (from the time of the load-miss to the time of the store-write-fault). If this information is prematurely lost, the store-write-fault will be unable to update the predictor but this is not fatal.

3 For the purposes of this work the effects of the "read-with-intent-to-modify" and the coherent write are equivalent.

4 We describe the enhanced scheme in terms of SCI idiosyncrasies but it can be adapted for other protocols.

can be “HEAD,” “MIDDLE,” or “TAIL” in a larger sharing list (Figure 2). The cache block can also be modified (i.e., memory has a stale copy) or unmodified (i.e., the cached blocks are the same as the memory block). The SCI states that describe both the position in the list and whether the block is modified or not are listed in Table 2. The states that are of interest are the HEAD_DIRTY and ONLY_DIRTY states. These states are an indication of migratory sharing because they suggest that the block was previously written by another node (migratory data must be written in succession by different processors). In the enhanced prediction scheme the counter is incremented only when the cache block is in any of the two aforementioned states at the time of the write fault—otherwise it is decremented.

4.3 Feedback mechanism

We propose an additional feedback mechanism intended to restrict the optimization further to migratory sharing patterns. This mechanism is invoked just after a positive prediction is made for migratory sharing and updates the predictor using coherency information from the directory. Note that the predictor update is performed at the end of the load-miss —because of the positive prediction the write-fault will not occur.

This mechanism examines the directory’s state which is returned with all its responses. The reasoning is the same as in the enhanced scheme of Section 4.2: we increment the prediction counter if the directory suggests that the cache block was previously modified (written) by another node and decrement it otherwise.

4.4 Results

We studied the instruction-based prediction optimizations on CHOLESKY, MP3D and PTHOR that exhibit migratory sharing and on four control benchmarks (GAUSS, APSP, BARNES and OCEAN). Table 3 shows the speedups we obtained from the simple and the enhanced prediction schemes, with and without the feedback mechanism. In addition we show results for leaving the read-with-intention-to-write (RWITW) option on for all data. The optimization of the prediction schemes is to use this option selectively for migratory sharing. Table 3 shows that RWITW ranges from harmless to disastrous. The differences of the simple scheme, the enhanced scheme, with and without feedback are small for the migratory sharing benchmarks. In contrast, the performance of three of the four control benchmarks suffers with the simple and the enhanced scheme *without* the feedback mechanism. Fortunately, the feedback mechanism provides the necessary corrective action to eliminate negative performance effects.

	Benchmark	**No feed-back**		**Feed-back**		
		Enhanced	**Simple**	**Enhanced**	**Simple**	**RWITW**
Migratory Sharing Benchmarks	CHOLESKY	1.11	1.10	1.10	1.05	1.02
	MP3D	1.17	1.17	1.18	1.17	1.00
	PTHOR	1.03	1.01	1.02	1.01	0.69
Control Benchmarks	GAUSS	1.00	1.00	1.00	1.00	0.42
	APSP	0.99	1.03	1.00	1.00	0.45
	BARNES	0.89	0.94	1.01	1.00	—
	OCEAN	0.92	0.88	1.00	0.99	—

Table 3: Simulation results for migratory sharing optimizations (32 nodes, speedup over SCI).

Our results are satisfactory compared to those reported previously for address-based prediction [16][17] given the differences in the simulated systems, and in particular the larger block size. Cox and Fowler reported that the block size has significant effects on the performance of their adaptive protocol for migratory data: increasing block size leads to smaller performance improvements. They reported speedups of 1.23 for CHOLESKY and 1.11 for MP3D in 16 nodes and with a block size of 16 bytes. Similarly, Stenström, Brorsson and Sandberg report good speedups (1.54 for MP3D and 1.25 for CHOLESKY) again in 16 nodes and for a small block size (16 bytes). Although our evaluation setup does not allow us to go below a block

size of 32 bytes we examined larger blocks (64 bytes) and arrived at the same conclusions. In our instruction-based prediction schemes large block sizes (64 bytes) may reduce the performance benefits as much as 50%.

Statistics	CHOLESKY	MP3D	PTHOR
Static loads considered (all 32 nodes / ave. per node)	1809 57	2211 69	6148 192
Loads allocated in the predictor (all 32 nodes / ave. per node / maximum)	597 19 46	919 29 34	1649 52 72
Total predictor probes	283473	550647	1550132
Hits in the predictor (% of total probes)	159018 (56%)	535133 (97%)	1185286 (76%)
Optimizations invoked (% of predictor hits, % of predictor probes)	37807 (24%, 13%)	477308 (89%, 87%)	649743 (55%, 42%)

Table 4: Statistics for the enhanced scheme with feedback (shaded in Table 3).

The most striking results, however, are presented in Table 4 (for the migratory sharing benchmarks, using the enhanced prediction scheme with feedback). The number of predictor entries allocated is very low. On average, 19 predictor entries are needed for CHOLESKY, 29 for MP3D and 52 for PTHOR. In comparison, the adaptive protocols for migratory data (i.e., address-based prediction) require storage in proportion to the size of the directories! The maximum number of predictor entries was allocated in node 0 (which also executes initialization code) for all three benchmarks. Table 4 also contains statistics about the behavior of the predictors. The number of optimizations in CHOLESKY and PTHOR is low. This is because the frequency of the HEAD_DIRTY state (that suggests migratory sharing) on write-faults is also low (Table 2).

5 Large-scale sharing prediction

Kaxiras and Goodman —among others— argue that widely shared data (that are accessed by many processors and are frequently updated) can be a serious performance bottleneck in larger shared-memory systems [4][6][7][8]. They proposed extensions to SCI (called GLOW extensions) that provide scalable reads and writes for widely shared data. However, these extensions should not be invoked for other than widely shared data because the overhead may outweigh the benefit. Thus, there are two options for making effective use of these extensions: either wide sharing should be defined statically (undesirable because it is not transparent) or dynamically. An adaptive method where the directory identifies widely shared data has been proposed [8][5]. In this method the directory detects widely shared data (by keeping track of the number of readers) and subsequently informs the nodes in the system to use the GLOW extensions for such data.

In contrast we propose to use instruction-based prediction to predict which load instructions are likely to access widely shared data. The prediction is based on previous history: if a load accessed widely shared data in the past then it is likely to access widely shared data in the future. This behavior can be traced to the way parallel programs are structured. For example in Gaussian elimination the pivot row —which changes in every iteration— is widely shared and it always accessed in a specific part of the program. Therefore, once the load instruction that accesses the pivot row has been identified it can be counted on to continue to access widely shared data. We have found that this prediction is very strong for all our benchmarks.

We have identified two citeria for making a determination of whether a load accessed widely shared data: latency and directory feedback:

[5] Another method to handle widely shared data is request-combining [3]. However, in this paper we concentrate on comparing only address-based and. instruction-based prediction schemes.

- **Latency**: Whether a load accessed widely shared data can be judged by its miss latency: very large miss latency is interpreted as an access to widely shared data. Using latency as the basis for the prediction is not as farfetched as it sounds: access latency of widely shared data is significantly larger than the average access latency of non-widely shared data. This is because of network contention and most importantly because of contention in the home node directory which becomes a "hot spot" [36]. The latency threshold for widely shared data is a *tuning* parameter that can be set independently for different applications. For this work we set the threshold latency to double the average miss latency of the benchmark.
- **Directory feedback**: This scheme is inspired by the address-based scheme proposed by Kaxiras and Goodman [8]. Information about the nature of the data is supplied by the directory. The directory counts the number of reads between writes and if this number exceeds a certain threshold the directory's responses indicate that the data block is widely shared. The threshold is again a *tuning* parameter and for this work we set it to a low number of 4 (i.e., more than 4 out of the 32 nodes reading is considered wide sharing). We believe that this scheme is more focused on wide sharing than the latency-based scheme which could be fooled by random long latency operations.

As in Section 4 the predictor is a small fully associative table. Each predictor entry contains the PC of a load and a 2-bit saturating counter to make predictions. Each predictor entry is about 5 bytes. Positive predictions are made when the counter exceeds a threshold value of 1.

A working example: Figure 3 depicts instruction-based prediction for wide sharing. Upon a load-miss the predictor is probed for information. At first the predictor is empty (predictor miss). The load-miss generates a coherent read. The response to this read will update the predictor according to the criterion used. A new entry is allocated in the predictor and its counter is reset to 0, if the read latency exceeds the latency threshold or if the response from the directory indicates that the block is widely shared. This will happen twice more before the counter exceeds the threshold. At this point a predictor probe returns a positive prediction for wide sharing and instead of an ordinary read a special GLOW read is issued. The GLOW read will trigger the creation of sharing trees in the network.

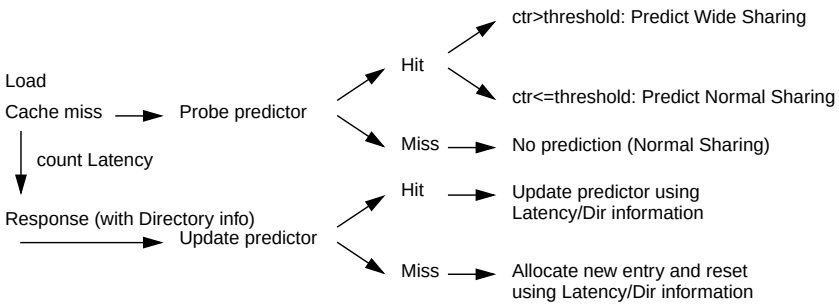

FIGURE 3. Instruction-based prediction for wide sharing.

5.1 Adapting back

The simple instruction-based prediction described above adapts easily to wide sharing but it is not trivial to adapt the other way around. Using the GLOW extensions for non-widely shared data (e.g, when only very few nodes share the data simultaneously) results in lower performance since very few nodes incur all the overhead of building scalable sharing trees in the network without any other nodes benefiting [7]. Thus we need to detect when wide sharing has ceased and refrain from using GLOW. Here, we briefly describe two such schemes to adapt back.

It is virtually impossible to obtain reliable feedback for the latency-based prediction because a low miss

latency can be attributed either to *lack* of wide sharing *or success* of the GLOW extensions in handling wide sharing. To adapt back in this situation we can use an expiration counter (we call it *poison* counter) for each predictor entry. To use this feature we set the counter to a non-zero value and each time the predictor entry is used we decrement it. When the poison counter hits zero the predictor entry is deleted. This scheme can also be used in the directory-feedback prediction scheme.

The problem with the directory-feedback prediction is that the actual number of sharers cannot be reliably tracked by the directories when GLOW is in use because of GLOW's read-combining. To solve this problem once a directory discovers a widely shared block it continuously indicates this in its responses until it is directed to do otherwise. The writers are responsible to verify (and correct if necessary) the directory's claim that a data block is widely shared by counting the number of nodes they invalidate (in SCI it is the writer node that invalidates all other sharing nodes rather than the directory).

For the benchmarks that *do have* widely shared data we found no benefit in using the schemes for adapting back. In fact the performance benefit diminishes slightly. These schemes are mainly intended for situations where the wide sharing prediction can have harmful effects on performance (see next section).

5.2 Results

		32 nodes			**64 nodes**		
		Instruction-based prediction		**Address-based prediction**	**Instruction-based prediction**		**Address-based prediction**
		Latency	**Directory-feedback / adapt-back**	**Adaptive Directory-detection**	**Latency**	**Directory-feedback / adapt-back**	**Adaptive Directory-detection**
Wide Sharing Benchmarks	GAUSS	1.20	1.19	1.13	1.66	1.64	1.43
	SPARSE	1.06	1.04	1.13	1.32	1.28	1.25
	APSP	1.11	1.12	1.00	1.53	1.52	1.00
	TC	1.14	1.14	1.01	1.53	1.54	1.02
	BARNES	1.30	1.29	1.27	1.13	1.14	1.13
Control Benchmarks	OCEAN	1.00	1.00	0.91	1.00	1.00	0.95
	CHOLESKY	1.00	0.91 / 0.99	—	1.00	0.92 / 0.99	—

Table 5: Results for wide sharing optimizations (speedup over SCI).

We present results for the two instruction-based prediction schemes (latency-based and directory-feedback). To compare against an address-based scheme we implemented the adaptive "directory detection" scheme as described by Kaxiras and Goodman in [8]. Since the negative performance impact of wide sharing is more pronounced in larger machines we present results for both 32 and 64 nodes. Table 6 shows the speedups for the five benchmarks with wide sharing and for the two control benchmarks. The two instruction-based prediction schemes perform almost identically yielding speedups of up to 1.30 for BARNES in 32 nodes and up to 1.54 for TC in 64 nodes. They both outperform the address-based scheme in all benchmarks (and in the case of APSP and TC by a significant margin). Only the performance of one of the control benchmarks (CHOLESKY) suffers from instruction-prediction optimizations and in particular from the directory-feedback scheme. However, when the mechanism to adapt back is enabled the negative performance impact is minimized. The other control benchmark (OCEAN) is affected negatively by the address-based

scheme.

Nodes	Statistics	GAUSS	SPARSE	APSP	TC	BARNES	CHOLESKY
32	Loads allocated in the predictor (all 32 nodes / ave. per node / maximum)	249 8 9	643 20 25	94 3 3	91 3 3	1555 49 54	349 11 35
64	Loads allocated in the predictor (all 64 nodes / ave. per node / maximum)	557 9 11	1238 20 24	180 3 3	185 3 3	3251 51 56	711 11 39

Table 6: Statistics for wide sharing prediction (Directory-feedback scheme).

Table 6 contains predictor statistics for the directory-feedback scheme (results for the latency-based scheme are similar). For these schemes —because they do not adapt back— the number of predictor hits is approximately equal to the number of predictor probes and the number of optimizations is approximately equal to the number of predictor hits. A striking result is the amazingly small number of predictor entries allocated for each benchmark.

6 Producer-Consumer sharing prediction

Finally we present instruction-based prediction for producer-consumer sharing. The producer is a store instruction that generates misses or write-faults. Its potential consumer(s) are tracked using information from the CC-protocol. The prediction can take the following two forms: i) a binary prediction for the existence of stable producer-consumer sharing and, ii) prediction of the identity of potential consumers. Using the first form we can invoke pairwise sharing optimization or switch to an update protocol. Using the second form we can switch to an update protocol or *pre-send* data *speculatively* to consumers.

An update protocol would not constitute a *transparent* optimization in the case of a sequentially consistent memory system because such protocols can violate sequential consistency and therefore need support from the programmer/compiler to guarantee correctness. Because of this reason and because SCI does not yet support an update protocol we did not study this optimization. However, we do believe that it is an interesting future direction for instruction-based prediction.

6.1 Pairwise sharing prediction

This scheme predicts whether there is a stable producer-consumer relationship with a unique consumer. A simple predictor tracks for each store its last known consumer (see also “In search of the consumers” in the next section). The predictor is similar to those proposed in the previous two sections but with the addition of an extra field per predictor entry to store the identity of the last consumer (a total of about 7 bytes per entry). The 2-bit saturating counter is used to indicate whether the last consumer remains the same or not. When the predictor entry is updated with a different consumer the counter is decremented; otherwise it is incremented. The identity of the consumer can be changed while the counter remains below the threshold.

If a store always has the same consumer we can optimize their communication. SCI provides such an optimization called *pairwise sharing* [2][5]. Pairwise sharing allows the head and the tail node of a two-node sharing list to communicate without going to the home node directory. We have tried this prediction/optimization scheme on two benchmarks: OCEAN and APPBT. We achieved a speedup of 1.07 (1.10 with 256KB caches) for OCEAN using on average 77 predictor entries per node. The speedups were negligible for the other benchmark. The pairwise sharing optimization is very well implemented in SCI and even if it is heavily misused it does not affect performance much. Thus, turning this optimization on for all data leads to comparable results (1 or 2 percentage points lower than our instruction-based prediction optimization).

6.2 Producer-consumer prediction with speculative-execution optimization

The most advanced prediction scheme we propose predicts the identity of the consumer(s). This scheme is

influenced by Moshovos and Sohi's [20] and Tyson and Austin's [21] work on optimizing producer-consumer communication in uniprocessors and prompted by Hill's views on speculative execution in shared-memory [23]. Evaluation of this scheme presents considerable difficulties because our current tools do not support speculative execution. Thus, we are unable to provide execution time measurements. Instead, — analogous to studying branch prediction— we study this scheme by presenting prediction accuracies and hit rates for the speculative pre-sends. Finally we discuss possible implementations of this scheme, including how to read speculative data external to the processor.

In search of the consumers: Before we describe the prediction scheme we need to explain how to identify possible consumers. The following discussion is dependent on the idiosyncrasies of SCI —in other directory-based CC-protocols the directory itself is an excellent source of information about consumers. In SCI, a node that wishes to write a cache block is responsible to invalidate the sharing list. Thus any nodes that are invalidated by the "producer" are considered consumer nodes. Additionally, any node that at a later point attaches in front of the producer (i.e., reads the producer's cache block) is considered a consumer. Since the attach is unrelated to any particular store instruction we use the same tagging technique as in Section 4: a cache block is tagged with the PC of the store that generated the last coherent event on it.

Prediction: We use a predictor structure similar to the pairwise predictor but in each prediction entry, instead of the field that holds the identity of a single consumer, we use a bit-map to track multiple consumers (a total of about 9 bytes per entry). We examined two simple predictor schemes: Last-prediction that predicts the last set of consumers to be the new set and Intersection-prediction that predicts the intersection of the last two sets of consumers to be the new set. The two schemes work as follows:

1. **Last-prediction**: The predictor is both updated and probed on a store-miss or a store write fault. The predictor is updated when the producer node invalidates a sharing list. The update collects the identities of the invalidated nodes on a temporary bit-map and compares it to the bit-map stored in the predictor entry. If there is significant overlap between the bit-maps the entry's 2-bit saturating prediction counter is incremented; otherwise it is decremented. The temporary bit-map (new) is then installed in the predictor entry. The predictor is then probed and if the counter exceeds a threshold the bit-map containing the possible consumers is returned. This predictor has two *tuning* parameters: the counter threshold and a parameter that defines what is "significant overlap" between bit-maps. In this work the threshold is 1 and the overlap parameter requires at least 2 common consumers (in bit-maps that *do* have 2 or more consumers).
2. **Intersection-prediction**: The predictor is again updated when the producer invalidates a sharing list. Again, the identities of the consumers are collected on a temporary bit-map. The logical `AND` of the temporary bit-map and the predictor enrty bit-map (that contains the consumers of the previous store-miss or store-write-fault) constitutes the prediction bit-map. After the prediction bit-map is calculated, the the temporary bit-map is installed over the predictor entry's bit-map.

Speculative *pre-send*: After obtaining a prediction about the identity of the consumers we can send them the data on condition that they use them speculatively until they verify the data's correctness through the coherence protocol. We call this *speculative pre-send*[6]. The hope is that the data will arrive at the consumer(s) before they even ask for them. A speculative pre-send is *outside the coherence domain* and this is how it differs from an update. Since everything has to be verified through the CC-protocol, speculative pre-sends affect only performance but not correctness. There are two questions concerning pre-sends: what to send and when to send. Regarding the first question we must decide whether to send just the new value written by the store or the whole cache block, while for the second question we must decide whether to send it immediately (at the end of the write fault) or wait until a later time. In this work we accumulate pre-sends in a queue which is emptied on synchronization operations (i.e., barriers and unlocks) and we send

[6] Inspired by "*pre-fetch*."

the whole cache line (if it is available at the time of the actual send)[7]. On the consumer side pre-sends are accumulated in the cache by taking advantage of invalid cache blocks. A speculative presend is only accepted if an invalid cache block with the same address exists in the consumer's cache. The reasoning for this restriction is that a correct presend is likely to encounter a corresponding invalid cache block since the producer previously invalidated all the consumers. Note that, no additional storage in the consumer nodes is needed for the presends.

How can a processor read external speculative data? To make a convincing argument for the feasibility of the speculative schemes we propose we sketch a method for a processor to read speculative data from the *pre-send cache*. Our proposal is compatible (at a high level) with existing memory speculation mechanisms in advanced processor designs.

In modern microprocessors that support speculative execution, loads can speculatively bypass stores that issued earlier and whose target address is unknown. If at a later time the address of the store is resolved and there is no dependence to the speculative load then the latter is committed; otherwise, if there is a dependence the speculative load is "squashed" along with all speculative instructions that followed (or in the case of *selective invalidation* along with all speculative instructions depended on the speculative load).

To read speculative data from the outside world, the processor creates a hypothetical *shadow store* whose address is unknown. The purpose of this shadow store (that never really executes) is to control the fate of the load that reads the external data. This load is executed speculatively, pending confirmation of absence of dependence to the shadow store. After the load reads the external speculative data, the address of the shadow store remains to be resolved. The outside mechanisms control the speculative execution by supplying the appropriate address for the shadow store. Eventually, the validity of the speculative data will be verified by the CC-protocol. If the data were correct the outside mechanisms supply to the shadow store an irrelevant address (e.g. 0x0000). If, however, the data were found to be wrong their address is supplied to the shadow store thereby squashing all incorrect execution. Note that there is some sort of random value speculation involved in our schemes: even if there never was a producer-consumer relationship but the data just happened to be correct the speculative execution is committed.

6.3 Results

In this section we present preliminary results (that we expect to improve by tuning the current predictors and by applying more sophisticated two level adaptive predictors) for the producer-consumer prediction using the pre-send optimization. We implemented all the mechanisms described in the preceding sections in our simulation environment except speculative execution. Thus, the producers use the predictors to send cache blocks to the consumers; the pre-send messages are accepted in the consumer nodes only if there is available space in their cache in the form of invalid cache blocks; the consumers upon a miss access their caches to read speculative data. However they cannot execute speculatively so they wait until they obtain a coherent cache block through the CC-protocol. When the coherent cache block is brought into the cache the consumers compare the speculative data to the coherent data to determine mis-speculations. In Table 7 and in Table 8 we report statistics we gathered using this setup for five benchmarks (OCEAN, APPBT, BARNES, GAUSS, and SPARSE).

Table 7 shows the results using the Last-prediction scheme and Table 8 using the Intersection-prediction scheme. For both tables we list the number of static stores that generated misses or write-faults and the number of entries in the prediction tables. These two numbers are the same since we track all stores encountered. Similarly to the other two instruction-based predictions described in previous sections, the number of predictor entries required is very low for all programs (APPBT requires the most: an average of 156 entries per node). The total number of predictor probes gives an indication of the usage of the predic-

[7] Since this is the initial exploratory work in this area, the full design space needs to be studied in future work.

Statistics	OCEAN	APPBT	BARNES	GAUSS	SPARSE
Static Stores considered (all 32 nodes / ave. per node)	2499 / 79	4991 / 156	1636 / 51	471 / 15	310 / 10
Predictor entries allocated (all 32 nodes / ave. per node)	2499 / 79	4991 / 156	1636 / 51	471 / 15	310 / 10
Total number of predictor probes (all nodes)	1378698	1120249	106535	175564	813815
% of probes that return a prediction	56%	70%	63%	54%	3%
Total pre-send messages sent	402404	975843	100696	87930	81756
(% of non-null predictions)	52%	125%	150%	93%	335%
pre-sends rejected (no corresponding invalid cache block)	168463	184833	60106	2466	24085
(% of total pre-sends)	42%	19%	60%	3%	29%
pre-send messages accessed in consumer nodes	158103	750177	24984	85183	56421
(% of total sent)	39%	77%	25%	97%	70%
accessed pre-send messages verified as correct	135866	222898	19439	85123	55032
(% of accessed)	86%	30%	78%	100%	98%
(% of total sent)	34%	23%	19%	97%	67%
accessed pre-send messages failed to verify as correct (mis-speculations)	22237	527279	5545	60	1390
(% of accessed)	14%	70%	22%	0%	2%
(% ot total sent)	5%	54%	6%	0%	3%

Table 7: Statistics for producer-consumer Last-Prediction with speculative pre-send (32 nodes).

Statistics	OCEAN	APPBT	BARNES	GAUSS	SPASRE
Static Stores considered (all 32 nodes / ave. per node)	2500 / 78	4985 / 156	1644 / 51	471 / 15	310 / 10
Predictor entries allocated (all 32 nodes / ave. per node)	2500 / 78	4985 / 156	1644 / 51	471 / 15	310 / 10
Total number of predictor probes (all nodes)	1378658	1118530	106263	175564	813960
% of probes that return a prediction	100%	100%	97%	100%	100%
Total pre-send messages sent	247428	828452	36530	87809	39480
(% of non-null predictions)	18%	74%	34%	50%	5%
pre-sends rejected (no corresponding invalid cache block)	57268	83378	15636	2032	10480
(% of total pre-sends)	23%	10%	43%	2%	26%
pre-send messages accessed in consumer nodes	129465	718429	14142	85612	28025
(% of total sent)	52%	87%	39%	97%	71%
accessed pre-send messages verified as correct	109501	193450	11227	85567	26745
(% of accessed)	85%	27%	79%	100%	95%
(% of total sent)	44%	24%	31%	97%	68%
accessed pre-send messages failed to verify as correct (mis-speculations)	19964	524979	2915	45	1280
(% of accessed)	15%	73%	21%	0%	5%
(% ot total sent)	8%	63%	8%	0%	3%

Table 8: Statistics for producer-consumer Intersection-Prediction with speculative pre-send (32 nodes).

tors (equivalent to the number of coherence events generated by stores). The percentage of the probes that return a prediction is a metric that depends on the prediction scheme employed. For the first scheme, Last-prediction, a saturating counter is used to indicate whether the successive store instances have common consumers. When the predictor is probed and the counter is below the threshold a null prediction is returned. The percentage of a non-null prediction ranges from 3% for SPARSE to 70% for APPBT. This percentage can be changed by tuning the threshold of the saturating counter and the parameter that defines the overlap in the consumer bit-maps. For the second scheme, Intersection-prediction, we do not employ a saturating counter. A null prediction is returned first two times a store is encountered. The non-null predictions can generate from zero to 32 pre-send messages (depending on the number of consumers predicted). However a prediction may be nullified *if the cache block is not available at the time of the pre-send*. Because the pre-send is delayed until a synchronization point, many times cache blocks are lost before they can be sent. Under these conditions the total number of pre-sends is shown in the corresponding rows of the two tables. The total number of pre-sends is also expressed as a percentage of the non-null predictions in the same row. A number of these pre-sends is rejected in the consumer nodes because there is no free space in their cache in the form of invalid cache-blocks. This number is quite high (e.g., 60% for BARNES). A possibility here is to implement a *speculative pre-send cache* that holds the pre-sends that do not fit in the main cache. To read speculative data, the processor would access this cache in parallel with its main cache. Such a cache is additional hardware but it would increase the number of pre-sends accessed and verified as correct. Finally, the numbers of interest are the number of pre-sends accessed in the consumer nodes and the percentage of them verified as correct. For OCEAN, APPBT, and GAUSS these two numbers are comparable for the two prediction schemes. For BARNES and SPARSE the second prediction scheme (Intersection-prediction) preforms better. The percentage of pre-sends accessed ranges from 25% to 97% for the first scheme and from 39% to 97% for the second scheme. The percentage of the accessed pre-sends that are verified through the CC-protocol as correct ranges from 30% to 100% for the first scheme and from 27% to 100% for the second. For all programs the percentage of correct pre-sends is comparable for the two schemes. It is high (78% or better) for most programs except from APPBT (for which only 30% and 27% of the accessed pre-sends are verified correct for the first and second prediction scheme respectively).

7 Conclusions

In this paper we explore instruction-based prediction to transparently optimize hardware shared memory. Instruction-based prediction is well established in the uniprocessor world but fairly novel in the world of parallel shared-memory architectures (where it has been used only for prefetching [38]). The compelling advantage of instruction-based prediction compared to address-based prediction is that it requires *very few* prediction resources.

We propose and study instruction-based prediction that *logically* stands halfway between the processor and the CC-protocol mechanisms. It requires two streams of information to converge to the prediction structures: from the processor we require the PC of the load and store instructions that generate coherence events; from the CC-mechanisms we require coherence information. Thus, we can track the history of loads and stores in relation to coherence events such as cache misses or write-faults. Subsequently, each time a known load or store generates a new coherence event we can take action to optimize it. In contrast, uniprocessor instruction-based prediction is deeply embedded in the processor and it is invoked with every dynamic instruction instance.

To make the case that instruction-based prediction is a serious competitor —not only in terms of resource usage but also in terms of performance— to previously proposed address-based prediction mechanisms we propose optimizations for three different sharing patterns:

- Migratory sharing. This prediction/optimization works well for three benchmarks (CHOLESKY, MP3D, and PTHOR) that exhibit migratory sharing and it is competitive to previously proposed address-based adaptive protocols. Equipped with safeguards to avoid applying the optimization to non-migratory

sharing it shows no negative performance impact on four other control benchmarks (GAUSS, APSP, BARNES, OCEAN). No more than 72 (5-byte) predictor entries were ever needed in any node's prediction table.

- Wide sharing. This prediction/optimization works very well and consistency outperforms an address-based scheme on five benchmarks which exhibit wide sharing (GAUSS, SPARSE, APSP, TC, and BARNES). With appropriate mechanisms for adapting back to non-wide sharing there is no negative performance impact on two other control benchmarks (CHOLESKY and OCEAN). No more that 56 (5-byte) predictor entries were ever needed in any node's prediction table.
- Pairwise sharing and producer-consumer sharing. For the pairwise sharing prediction/optimization we found that it does not offer significant advantages over the default optimization of our base CC-protocol. For the producer-consumer optimization that is based on speculation we found a significant number of speculative pre-sends can be successful. For this prediction, no more than 156 9-byte predictor entries were ever needed in any node.

Future directions: We believe that this work will be a starting point for novel and better instruction-based prediction optimizations. Similarly to work that examined the coherence behavior of data [37] we need to examine the behavior of the instructions in relation to the CC-protocol features and in relation to hardware parameters such as cache and block size.

Regarding the optimizations we study here, we considered them a starting point. We have implemented them on top of a complex CC-protocol and future work is needed to apply them to other —simpler and more streamlined— directory-based protocols. Also research is needed to examine how instruction-based prediction can be applied to bus-based shared-memory systems and even software-based shared-memory. Yet another direction is to examine hybrid prediction schemes that use both instruction-based and address-based prediction. This is especially interesting with the advent of generalized address-based prediction [22].

Finally, we subscribe to Hill's opinion that speculation will play an increasingly important role in transparently optimizing shared-memory. In this work we propose such an optimization and we have performed a preliminary evaluation using our current tool set. We are actively working in this area and hope that with the future wide availability of tools that integrate speculation and parallelism (such as the RSIM simulator form Rice University), speculation in shared-memory (including our proposal) will be researched in depth.

8 Acknowledgements

My thanks to Alain Kägi for his help with the WWT, SCI, and many of the benchmarks. I also thank Jim Goodman and Mark Hill who offered comments on early drafts of this paper.

9 References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990

[2] IEEE Standard for Scalable Coherent Interface (SCI) 1596-1992, IEEE 1993.

[3] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, "The NYU Ultracomputer Designing a MIMD Shared-Memory Parallel Computer." *IEEE Transactions on Computers*, Vol. C-32, no 2, pp. 175-189, February 1983.

[4] R. Bianchini and T. J. LeBlanc, "Eager Combining: A Coherency Protocol for Increasing Effective Network and Memory Bandwidth in Shared-Memory Multiprocessors." *Proceedings of the 6th Symposium on Parallel and Distributed Processing*, October 1994.

[5] Alain Kägi, Nagi Aboulenein, Douglas C. Burger, James R. Goodman, "Techniques for Reducing Overheads of Shared-Memory Multiprocessing." *International Conference on SuperComputing*, July 1995.

[6] Stefanos Kaxiras, "Kiloprocessor Extensions to SCI." *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[7] Stefanos Kaxiras and James R. Goodman “The GLOW Cache Coherence Protocol Extensions for Widely Shared Data.” *International Conference on Supercomputing*, May 1996.

[8] Stefanos Kaxiras and J. R. Goodman, “Two Dynamic Methods for Efficient Large Scale Sharing” University of Wisconsin TR-1351. Available at ftp.cs.wisc.edu

[9] Daniel Lenoski *et al*., “The Stanford DASH Multiprocessor.” *IEEE Computer*, Vol. 25 No. 3, pp. 63-79, March 1992.

[10] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood, “The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers.” *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pp. 48–60, May 1993.

[11] Tom Lovett, Russell Clapp, “STiNG: A CC-NUMA Computer System for the Commercial Marketplace.” In *Proc. of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[12] Convex Computer Corporation, “The Exemplar System” 1994.

[13] James Laudon, Daniel Lenoski. “The SGI Origin: A cc-NUMA Highly Scalable Server,” in *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[14] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[15] Satish Chandra, James R. Larus, Anne Rogers. “Where is Time Spent in Message-Passing and Shared-Memory Programs?” *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 61-73, October 1994.

[16] Alan L. Cox Robert J. Fowler, “Adaptive Cache Coherency for Detecting Migratory Shared Data.” In Proc. of the 20th ISCA, 1993.

[17] Per Stenstrom, Mats Brorsson, Lars Sandberg, “An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing,” In Proc. of the 20th ISCA, 1993.

[18] A. Gonzalez, C. Aliagas, M Valero. “A Data Cache with Multiple Caching Strategies Tunned to Different Types of Locality.” In *Proc. of the International Conference on Suprecomputing*, 1997.

[19] A. Moshovos, S. E. Breach, T. N. Vijaykumar, G. S. Sohi, “Dynamic Speculation and Synchronization of Data Depenences.” In *Proc. of the 24th Annual International Symposium on Computer Architecture*, 1997.

[20] A. Moshovos and G. S. Sohi, “Streamlining Inter-operation Memory Communication via Data Dependence Prediction” In *Proceeding of the 30th Annual Symposium on Microarchitecure*, Dec. 1997.

[21] Gary S. Tyson and Todd M. Austin “Improving the Accuracy and Performance of Memory Communication Through Renaming” In *Proceeding of the 30th Annual Symposium on Microarchitecure*, Dec. 1997.

[22] Shubhendu S. Mukherjee and Mark D. Hill “Using Prediction to Accelerate Coherence Protocols”, Final version to appear in *International Symposium on Computer Architecture* (ISCA), 1998 (Preliminary web version: www.cs.wisc.edu/~markhill).

[23] Mark D. Hill, “Multiprocessors Should Support Simple Memory Consistency Models”, Univ. of Wisconsin Computer Sciences Technical Report #1353, October 1997.

[24] Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, Ioannis Schoinas, Mark D. Hill James R. Larus, Anne Rogers, David A. Wood, “Application-Specific Protocols for User-Level Shared Memory” *Supercomputing* ‘94, Nov. 1994.

[25] James E. Smith, “A Study of Branch Prediction Strategies” In *Proc. of the 8th Annual International Computer Archtecture Symposium*, 1981.

[26] T-Y Yeh and Yale Patt, “Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Proc . of the 19th Annual International Computer Archtecture Symposium*, 1992.

[27] Santosh G. Abraham, Rabin A. Sugumar, Daniel Windheiser, B. R. Rau and Rajiv Gupta. “Predictability of Load/Store Instruction Latencies” *Proceedings of the 26th Annual International Symposium on Microarchitecture*, November 1993.

[28] G. Tyson et al. A New Approach to Cache Management" *Proceeding of the 28th Annual Symposium on Microarchitecure, Nov 28* - Dec 1, 1995.

[29] Steven K. Reinhardt, James R. Larus, David A. Wood, “Tempest and Typhoon: User-Level Shared Memory.” *Proc. of the 21st Annual International Symposium on Computer Architecture*, pp. 325-336, April 1994.

[30] David H. Bailey et al., “The NAS parallel benchmark: Summary and Preliminary Results.” IEEE Supercomputing ‘91, pp 158-165. Nov., 1991.

[31] Eric Hagersten, Anders Landin, and Seif Haridi, "DDM — A Cache-Only Memory Architecture." *IEEE Computer*, Vol 25. No 9, September 1992.

[32] Lucien M. Censier and Paul Feautrier, "A New Solution to Coherence Problems in Multicache Systems." IEEE Trans. Computers, Vol. 27, No. 12, pp. 1112-1118, Dec. 1978.

[33] A. Agarwal, M. Horowitz and J. Hennessy, "An Evaluation of Directory Schemes for Cache Coherence." *Proceedings of the 15th International Symposium on Computer Architecture*, pp. 280-289, June 1988.

[34] David Patterson et al., "The case for Intelligent RAM," *IEEE Micro*, Vol 17, No. 2, March/April 1997

[35] Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk, "Missing the Memory Wall: The Case for Processor/Memory Integration." In *Proceedings of the 23rd ISCA*, May 1996.

[36] Gregory F. Pfister and V. Alan Norton, "'Hot Spot' Contention and Combining in Multistage Interconnection Networks." *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 790-797, August 20-23, 1985.

[37] Wolf-Dietrich Weber and Anoop Gupta, "Analysis of Cache Invalidation Patterns in Multiprocessors." *Proc. of the 3rd International Conference on Architectural support for Programming Languages and Operating Systems*, pp. 243-256, April 1989.

[38] T-F. Chen and J-L Baer, "A Performance Study of Software and Hardware Data Prefetching Schemes." *Proc. of the 21st Annual International Symposium on Computer Architecture*, April 1994