Improving CC-NUMA Performance Using Instruction-Based Prediction

Stefanos Kaxiras

Bell Laboratories, Lucent Technologies
600 Mountain Ave., Murray Hill, NJ 07479
kaxiras@cs.bell-labs.com

James R. Goodman

University of Wisconsin-Madison
1210 W. Dayton St., Madison, WI 53706
goodman@cs.wisc.edu

Abstract

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. Typically, in this environment, prediction is based on data-block access history (address-based prediction) in the form of adaptive cache coherence protocols. The advantage of instruction-based prediction is that it requires few hardware resources in the form of small prediction structures per node to match (or exceed) the performance of address-based prediction. To show the potential of instruction-based prediction we propose and evaluate three different optimizations: i) a migratory sharing optimization, ii) a wide sharing optimization, and iii) a producer-consumer optimization based on speculative execution. With execution-driven simulation and a set of nine benchmarks we show that i) for the first two optimizations, instruction-based prediction, using few predictor entries per node, outpaces address-based schemes, and (ii) for the producer-consumer optimization which uses speculative execution, low mis-speculation rates show promise for 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 [8] and Sequent STiNG [24] that use Scalable Coherent Interface (SCI) networks and cache coherence [15], and the SGI Origin 2000 [22] that uses a directory-based cache coherence protocol originating in Stanford's DASH [23].

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 SMP-based (with a snoopy-bus CC-protocol) or cluster-based (with a directory-based CC-protocol). Recently, Hill argued that hardware-based shared-memory should be kept as simple as possible, presenting a sequentially consistent transparent shared-memory model to the programmer [14]. Hill argues that speculation could be used to offer transparently 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 [9,33], for pair-wise sharing and producer-consumer sharing [15,16], and for widely shared data [19]. Recently Mukherjee and Hill [26] 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 [36]. 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.

In this work we propose *Instruction-based Prediction* 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 structure. Whenever dynamic instances of load and store instructions generate coherence events (such as cache misses, or write-faults on read-only cache blocks) we consult the predictors for optimization hints.

Instruction-based prediction 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 address-based prediction optimizations affect the behavior of the CC-protocol toward the processor (e.g, the CC-protocol may decide to return a writable block to a processor that asks for a read-only block). The benefits of instruction-based prediction can be significant because it offers a 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.

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 [32] and Yeh and Patt [36]. Abraham et al. showed that very few loads are responsible for most cache misses [1] and subsequently Tyson et al. proposed instruction-based prediction to bypass selectively the cache for such loads [34]. Gonzalez, Aliagas, and Valero used instruction-based prediction to steer data on caches optimized differently for spatial and temporal locality [12]. Moshovos, Breach, Vijaykumar and Sohi introduced memory dependence prediction [25]. They proposed dependence predictors accessed using the address of memory instructions. Chen and Baer were the first to propose prefetching based on instruction-based prediction for parallel systems [7]. Although we believe that instruction-based optimizations can be generally applicable (from bus-based cache coherence to software based coherence) we restrict this presentation to hardware-based, directory-based coherence [15,5,2] (e.g., CC-NUMA).

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 optimize effectively many different aspects of shared-memory using few hardware resources. To support this claim we apply instruction-based prediction to optimize *transparently* three sharing patterns. The optimizations affect performance but not correctness. These three optimization schemes are intended to provide proof-of-concept and we expect that with future research in this area more 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 [9], and of Stenström et al. [33]) is to convert the coherent read to a coherent write. We compare the instruction-based scheme to previously proposed adaptive migratory protocols and show that it matches or exceeds their performance using less than 128 predictor entries.
- **Predict whether a load will access widely shared data.** We 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 [17,18]. The instruction-based schemes consistently outperform an address-based adaptive scheme proposed for wide sharing [19], using less than 128 predictor entries per node.
- **Predict which nodes are 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. Upon encountering a known store we can *speculatively pre-send* the newly created value 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. Using few prediction resources, many pre-send messages are verified as correct.

Structure of this paper—Section 2 discusses implementation issues for instruction-based prediction in shared-memory. Section 3 discusses our evaluation methodology. We propose and evaluate instruction-based prediction for optimizing migratory sharing patterns in Section 4. In Section 5 we evaluate instruction-based prediction for wide sharing. Section 6 describes instruction-based prediction for producer-consumer sharing. Finally, Section 7 wraps up this work.

2 Implementation issues

In contrast to previous work where various schemes try to learn the coherence history of a data block at the directories and/or caches, our approach is based on observing the history of load and store instructions in relation to coherence events.

In this paper we emphasize prediction on coherence events since it is a technique whose implementations stand between the processor core and the cache coherence mechanisms—and as such it is a natural point to study first. 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 program counter (PC) of an instruction that generates a coherence event (e.g., cache miss, write fault, etc.) and information from the cache coherence mechanisms be available to the predictors. If 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 [13] nodes complete with caches, directories and local memory (such as the new Compaq ALPHA 21364 [3] or such as those studied by Saulsbury, Pong, and Nowatzyk [30] in the context of IRAM [27]). These devices would be an ideal platform in which to implement instruction-based prediction. In the case where there is a boundary between the processor core and the coherence mechanisms, implementation of our techniques becomes difficult: a channel through which information can flow among the processor core, the coherent mechanisms, and the predictors must be established.

With respect 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 coherent events. Three events are relevant in this paper: i) cache miss, ii) write fault, and iii) external cache read/invalidation. Upon 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 neither a potential bottleneck nor add cycles to the critical path.

3 Evaluation setup

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

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 (WWT) [29] is a well-established tool for evaluating large-scale parallel systems through the use of massive, detailed simulation. For speed WWT uses *direct execution* but this also 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.

***SCI*—**We have chosen to use SCI as the underlying cache coherence protocol. We chose SCI because it has a rich set of options that can be used to implement optimizations and in addition we have extended it to handle widely shared data [17,18]. In our initial attempt in this area [20], we found the complexity of the SCI protocol impedes simplicity in some of the mechanisms we previously considered. Here, the various instruction-based prediction schemes we propose are not dependent on the specifics of SCI and they can be applied equally well to other directory-based cache coherence protocols.

***Hardware parameters*—**We simulated SCI systems made of readily available components such as SCI network and workstation nodes. For the evaluation in Section 5, which requires detailed network simulation, we have simulated K-ary 2-cube systems (2 dimensions). We simulate contention throughout the network but messages are never dropped since we assume infinite queues. For the evaluation of Section 4 and Section 6 we simulated a constant latency network (which takes 100 processor cycles to transfer any message) with contention at the endpoints. The nodes comprise a processor, an SCI cache, memory, memory directory, and a network interface. The processors run at 500MHz and execute one instruction per cycle in the case of a hit in their caches. 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 1.2GB/sec, 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 (1GB/sec).

***Benchmarks*—**For this study we use nine 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 [31,6,11]. 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 [9], and by Stenström, Brorsson, and Sandberg [33]. We use the same input for comparisons. For optimization of wide sharing we use the following benchmarks: GAUSS, SPARSE, All Pairs Shortest Path (APSP) and Transitive Closure (TC) [18], and BARNES (taken from the SPLASH benchmark suite [31]). These benchmarks (except BARNES) were used to evaluate scalable extensions to SCI in both static [18] and adaptive flavors [19]. 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 OCEAN (taken from SPLASH [31]), 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.

Bench.	Input Size	Cache /Block	Sharing			Ref.
			Migr	**Wide**	**P-C**	
CHOLESKY	bsstk14	64K/32	Yes	—		[31,9,33]
MP3D	10K/10 iter	64K/32	Yes			[31,9,33,16]
PTHOR	risc	64K/32	Little			[31,9,33,16]
GAUSS	512x512	64K/64	—	Yes (dyn.)	Yes	[6,18]
SPARSE	512x512	64K/64		Yes (static)	Yes	[18]
APSP	256x256	64K/64	—	Yes (dyn.)		[18]
TC	256x256	64K/64		Yes (dyn.)		[18]
BARNES	4K part.	64K/64	—	Yes (static)	Yes	[31,18,16]
OCEAN	130x130	64K/32	—	—	Yes	[31,16]

Table 1: Benchmarks used in this paper.

4 Migratory sharing prediction

In this section we describe an instruction-based prediction that can handle migratory sharing patterns. Migratory data, defined by Weber and Gupta [35], are accessed by one processor at a time. Typically, these data are protected by locks and are accessed inside critical sections.

The idea of our scheme 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. This mechanism can be classified as **cache block anti-dependence** prediction since it detects writes after reads on the same cache blocks. Let us examine why this optimization is related to migratory sharing patterns. Migratory data are continuously read-modified-written but each time by a different processor [35]. Each processor brings them into its cache as *Read-Only* (RO) cache block, tries to modify them, generates a write fault, converts them to a *Read-Write* (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 is to convert the coherent read to a coherent write ending up with a RW cache block and thus avoiding the write fault. The optimization comes from the work of Carter et al. (Munin) [4] and from the adaptive CC-protocols proposed independently by Cox and Fowler [9], and by Stenström, Brorsson, and Sandberg [33]. In general, in invalidation-based cache coherence protocols the optimization is to “migrate” the data whenever a new processor access them by giving it both read and write permissions, even if it first accesses the data with a read. To give the new processor exclusive access to the data the previous copy (in the processor that last accessed them) is invalidated. The optimization works well because it folds two coherent transactions into one: both the latency and the transaction traffic of the store-write-fault are completely eliminated.

The folding of the two transactions can be initiated either by the home node directory when it decides to return a RW cache block in response to a read request [9,33], or by the processor when, upon a read, it asks for a RW block knowing that it accesses migratory data [4]. SCI originally required three non-overlapping transactions for the folded transaction (directory access, attach to previous node, invalidation of previous node) but we have upgraded it to support an *attach&invalidate* transaction thus making it equivalent to Dir_iNB protocols [2]. Also, in SCI a head node is not required to communicate with the directory to write a cache block, but to implement the address-based optimization we modified it (with negligible effects on performance) to behave like Dir$_i$NB or the DASH protocol [23].

4.1 Cache block anti-dependence prediction

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 in a RW cache block to prevent the write-fault. The predictor is a small fully associative table accessed by the load program counter (PC). Each predictor entry contains the PC of the load, the address of the last cache block on which the load missed, a small n-bit saturating counter used to make predictions, and a prediction bit (P-bit) that indicates unconfirmed predictions and provides the means for adapting back. Thus, the size of each entry is about 9 bytes (assuming 32-bit addressing). To further reduce the size of the predictor entries, the PC field can be truncated to a few bits or omitted altogether. In the latter case, the predictor is a direct mapped structure indexed by the PC. The address field can also be truncated. Although truncating or eliminating fields may introduce mistakes in accessing or updating the predictor, the smaller size of each predictor entry translates to many more entries for a fixed transistor budget. This may be preferable for large programs. For clarity, in this paper we only discuss full, non-truncated predictor entries.

We only use load PC in the predictor entries and not the store PC. This means that unique load/store pairs are not tracked but all the pairs that have a common load PC are lumped together. A predictor entry, therefore, refers to a single load but it can be affected by multiple distinct stores. We examined alternative implementations where the predictor entry contains both the load PC and the store PC but we did not find enough evidence for their usefulness.

A working example—Figure 1 shows how the predictor is updated by load-misses, stores and store-write-faults, and by external events. On a load-miss the predictor is probed and at first it is empty. A new entry is allocated and includes the load PC and the address of the cache block (Figure 1a). The counter is initialized to zero and the prediction bit (P-bit) is reset since no positive prediction has been made yet (Figure 1a). When a store generates a write-fault the predictor is searched (associatively) using the address of the cache block to find the corresponding load (Figure 1b). Note that the store PC is not needed. A store-write-fault increments the counter of the corresponding entry (Figure 1b). This load-miss/store-write-fault scenario repeats until the counter exceeds a prediction threshold (Figures 1c and 1d). The next time the load-miss occurs (Figure 1e) a positive prediction is made and the coherent read is converted to a coherent write.

Implementation details—In the above description each predictor entry has an address field which holds the address of the last cache block loaded by the corresponding load. This is the "meeting" point, the point that establishes the correspondence of a load-miss and subsequent store-write-faults. Since, more than one load might miss on the same cache block (at different times) the address of the cache block could exist in multiple predictor entries. To avoid this situation, only one instance of a cache block address is allowed in the predictor and specifically only for the load that missed last on this cache block—if another entry contains the same address its address field is purged. More elaborate implementations that can handle concurrent instances of the same load instruction are described elsewhere [19].

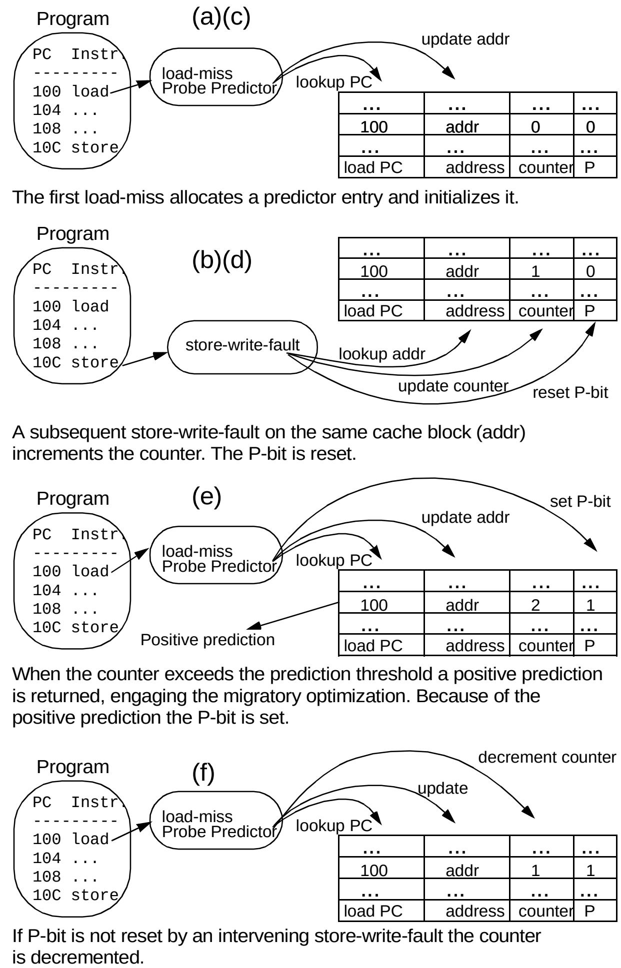

Figure 1. Cache block anti-dependence prediction mechanism.

Adapting back—The prediction bit (P-bit) provides the method for adapting back. Its purpose is to confirm the prediction that the cache-block will be written by an ensuing store. When a positive prediction has been made the P-bit is set. To confirm the prediction, the P-bit must be reset when a store writes the block. The next time the load probes the predictor it will find the P-bit either set or reset. If it is set the cache block was not written. In this case (shown in Figure 1f), the prediction counter is decremented and the P-bit is reset (if, however, the counter did not fall below the threshold a new positive prediction will set the P-bit again). On the other hand, if a load finds the P-bit reset a previous positive prediction has been confirmed.

Special care is needed for a store to reset the P-bit. Because of the positive prediction no write-fault will occur to trigger a predictor update. To circumvent this problem, we can update the predictor with any dynamic store instance (regardless of whether it generates a write-fault or not). The disadvantage of this method is the increased pressure on the predictor. Alternatively, we can trap stores that write on cache-blocks brought in with positive predictions. This can be accomplished by bringing in the cache block to an intermediate state which enjoys write privileges but denotes unmodified data. Such state exists in many protocols and it commonly referred to as "Exclusive." Thus, the transition from "Exclusive" to "Modified" (which can be considered as a soft write-fault) triggers a predictor update and resets the P-bit. This is the scheme implemented for the evaluation.

External events— A simple-minded implementation of this instruction-based prediction scheme can be fooled by other-than migratory sharing patterns (e.g., when more than one node read the data block before it is written). To avoid such complications, we only consider read-modify-write operations on cache blocks that are the only copies in the system (i.e., exclusive) and are not affected in any other way throughout the operation. If the cache block is read by another node between the time of the load-miss and the time of the store-write-fault we disable the update of the predictor. This is accomplished by deleting the cache block address field of the corresponding predictor entry. Thus, the correspondence of a load-miss and a store-write-fault cannot be established through the address of a cache block that was externally read. Cache block replacements and invalidations also have a similar effect because they lead to subsequent misses—not store-write-faults.

Prediction threshold— The prediction threshold provides hysteresis in adapting to migratory sharing and back. A low threshold allows the optimization to be applied soon after the load-miss store-write-fault behavior is detected, but it delays adapting back in the face of unconfirmed predictions when the saturating counter has reached its highest value. The opposite behavior is obtained by using a high threshold. In this work we used a 2-bit saturating counter with a prediction threshold of 1.

4.2 Results

We have studied the instruction-based prediction optimizations on CHOLESKY, and MP3D that exhibit migratory sharing and on five other benchmarks (PTHOR, GAUSS, APSP, BARNES and OCEAN). Table 2 shows the speedups of instruction-based prediction (**Instr.**) and of the address-based adaptive protocol (**Address**) over SCI. In addition, we show results applying the migratory optimization for all accesses (denoted by **All** in Table 2). Using the migratory optimization indiscriminately (instead of selectively as the other schemes do) ranges from positive (CHOLESKY), to harmless (MP3D), to disastrous (all other programs).

Bench.	SC model				Relaxed model r1 [16]	
	SCI	**Instr.**	**Address**	**All**	**SCI**	**Instr.**
CHOLESKY	1.00	1.13	1.12	1.08	1.01	1.14
MP3D		1.16/1.30	1.21	1.02	1.22	1.29
PTHOR		1.02	1.00	0.69	1.07	1.10
GAUSS		1.00	1.00	0.42	1.01	1.01
APSP		1.03	1.00	0.45	1.03	1.04
BARNES		0.99	1.00	0.73	1.06	1.05
OCEAN		1.02	1.00	0.83	1.05	1.06

Table 2: Simulation results for migratory sharing optimizations (32 nodes, speedup over SCI).

Results show that instruction-based prediction works better than the address-based scheme for CHOLESKY (speedup of 1.13 vs. 1.12). For MP3D, the instruction-based method lags behind the address-based method (speedup of 1.16 vs. 1.21) but we discovered that disabling the adapt-back mechanism increases the speedup to 1.30. In MP3D some migratory accesses (to the particle array) are *not* protected by locks. Thus, although they are migratory "in spirit," in reality external events (e.g., reads, invalidations) interfere with the adapt-back mechanism, which rules out quite a few. The rest of the programs (PTHOR, GAUSS, APSP, BARNES, and OCEAN) do not exhibit migratory sharing and instruction-based prediction provides small performance improvements.

Under a relaxed memory model that hides write latency, the instruction-based prediction still offers performance improvements albeit smaller than the sequentially consistent memory model case. We have used the relaxed memory model for SCI, called **r1**, described by Kägi et al. [16]. These results are consistent to the results reported by Stenström, Brorsson and Sandberg. With a relaxed memory model the argument for the migratory optimization (regardless of how it is applied) becomes primarily an argument of traffic: the migratory optimization reduces coherent traffic in proportion to the amount of migratory sharing in the program [33].

Our results are comparable to those reported previously for address-based prediction [9,33] given the differences in the simulated systems and in particular the cache coherence protocols (e.g., SCI vs. DASH), the number of nodes and the 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). WWT does not allow a block size smaller than 32 bytes but we briefly examined larger blocks (64 bytes) and observed similar effects for the instruction-based prediction scheme: large block sizes (64 bytes) reduce the performance benefit. The effects of the block size on prediction mechanisms merit further investigation. The instruction-based prediction mechanism could be protected from the adverse effects of larger block sizes by relying on the exact word addresses of the load and store instructions rather than the coarse-grain cache-block addresses.

Statistics		**CHOLESKY**	**MP3D**	**PTHOR**
Static loads considered	all 32 nodes	1844	2264	6309
	average per node	58	71	197
Active predictor entries (loads followed by stores)	all 32 nodes	597	751	1687
	average per node	19	24	53
	maximum	46	34	71

Table 3: Statistics for instruction-based prediction

The most striking results, however, are presented in Table 3 (for the migratory sharing benchmarks). The number of predictor entries allocated is very low. On average, 19 predictor entries are needed for CHOLESKY, 24 for MP3D and 53 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.

5 Large-scale sharing prediction

Widely shared data (that are accessed by many processors and are frequently updated) can be a serious performance bottleneck in larger shared-memory systems [17,18]. We have proposed extensions to SCI (called GLOW extensions) that provide scalable reads and writes for widely shared data [17,18]. 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. Besides request combining in the network, which we do not study here, an adaptive method where the directory identifies widely shared data has been proposed [19]. 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.

Here, we examine 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 is 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 accurate for all our benchmarks.

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

- **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” [28]. 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 in [19]. 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 2 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 (latency or directory feedback). 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 [18].

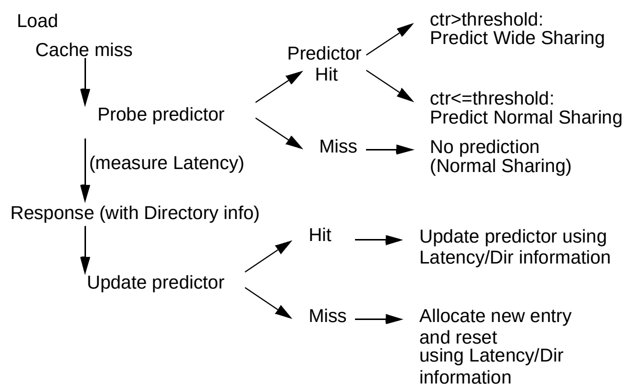

Figure 2. Instruction-based prediction for wide sharing.

5.1 Adapting back

The simple instruction-based prediction described above adapts easily to wide sharing but it is not trivial to adapt the opposite way. 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 [18]. 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 use an expiration 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 counter hits zero the predictor entry is deleted. Such expiration counters 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 [18]. 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 the writer node invalidates all other sharing nodes rather than the directory) [19].

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.

5.2 Results

We present results for the two instruction-based prediction schemes (latency-based and directory-feedback). To compare against an address-based scheme we use the adaptive "directory detection" scheme [19]. Since the negative performance impact of wide sharing is more pronounced in larger machines we present results for both 32 and 64 nodes. Table 5 shows the speedups for the five benchmarks with wide sharing and for the two control benchmarks. The two instruction-based prediction schemes (**IL** using the latency criterion and **ID** using directory feedback) 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 (**A-DD**) 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 (shaded cells in Table 4). The other control benchmark (OCEAN) is affected negatively by the address-based scheme.

	32 nodes			**64 nodes**		
	IL	**ID**	**A-DD**	**IL**	**ID**	**A-DD**
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
OCEAN	1.00	1.00	0.91	1.00	1.00	0.95
CHOLESKY	1.00	0.91/0.99	0.96	1.00	0.92/0.99	1.02

Table 4: Results for wide sharing optimizations (speedup over SCI).

N.	Statistics	GAUSS	SPARSE	APSP	TC	BARNES	CHOLESKY
32	Total entries	249	643	94	91	1555	349
	Average per node	8	20	3	3	49	11
	Maximum	9	25	3	3	54	35
64	Total entries	557	1238	180	185	3251	711
	Average per node	9	20	3	3	51	11
	Maximum	11	24	3	3	56	39

Table 5: Statistics for wide sharing prediction (Directory-feedback scheme).

Table 5 contains predictor statistics for the directory-feedback scheme (results for the latency-based scheme are similar). Again, the striking result is the very small number of predictor entries allocated for each benchmark. The number of predictor entries required depends on the number of points in the program which access widely shared data (such as counters, global variables, pivot rows, etc.) and not on the size of the code. Unless many points in the source code access widely shared data, even in much larger programs this number is likely to remain low.

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 [15,16] or switch to an update protocol. Using the second form we can *pre-send* data *speculatively* to consumers.

Pairwise sharing allows two nodes to communicate without going to the home-node directory. The pairwise sharing optimization is very well implemented in SCI and even if it is heavily misused it does not affect performance [16]. Thus, using instruction-based prediction to apply selectively pairwise sharing does not offer significant advantage over indiscriminate use of this optimization for all accesses [20].

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.

Subsequently, we propose a new optimization for producer-consumer sharing and apply it using instruction-based prediction.

6.1 Prediction with speculative execution

The most advanced prediction scheme we propose predicts the identity of the consumers and sends the data speculatively to the consumers. The prediction is influenced by the work of Moshovos et al. [25] on optimizing producer-consumer communication in uniprocessors. The optimization (to send the data speculatively to potential consumers) is influenced by the work of Koufaty, Chen, Poulsen, and Torrellas on Data Forwarding [21] and prompted by Hill's views on speculative execution in shared-memory [14].

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 keeps track of the consumers (using a full bitmap) and can supply all the relevant information to a producer. In SCI, a node that wishes to write a cache block is responsible for invalidating the sharing list. Thus, any nodes that are invalidated by a node (the "producer") are considered consumer nodes. As an option, any node that at a later point attaches in front of the producer (i.e., reads the producer's cache block) can be considered a consumer.

Prediction—We use a predictor structure similar to the predictors described previously but in each prediction entry we use a bit-map (32 bits) to track multiple consumers (a total of about 13 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 in a temporary bit-map. The logical AND of the temporary bit-map and the predictor entry 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 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*. The hope is that the data will arrive at the consumer(s) before they even ask for them. Speculative pre-send is not an update: (i) it is *outside the coherence domain*, (ii) the set of the predicted consumers is not cumulative (as in the update) but it changes dynamically, (iii) it allows feed-back through the coherence protocol. 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 small buffer which is emptied on synchronization operations (i.e., barriers and unlocks) and we send the whole cache line (if it is available at the time of the actual send). This scheme is based on the concept of *write caches*, that improve the performance of update protocols by coalescing writes on the same cache blocks [10]. The same reasoning applies in our case.

On the consumer side pre-sends are accumulated in the cache by taking advantage of invalid cache blocks. A speculative pre-send is only accepted if an invalid cache block with the same address exists in the consumer's cache. We impose this restriction for two reasons: (i) a correct pre-send is likely to encounter a corresponding invalid cache block since the producer previously invalidated all the consumers—assuming a stable producer-consumer relationship—and (ii) it provides a cost-effective implementation since no additional resources (to store pre-sends) are required at the consumers.

How can a processor read speculative data?—To make a convincing argument for the feasibility of the speculative schemes we sketch a method for a processor to read speculative data from outside. 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 dependent 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 (which never really executes) is to control the fate of the load that reads external speculative 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 (an additional *shadow load* can be used to read this address). 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) that does not affect the execution of the program. If, however, the data were found to be wrong their address is supplied to the shadow store thereby squashing all incorrect execution.

6.2 Results

In this section we present preliminary results for the producer-consumer prediction using the pre-send optimization. We implemented all the mechanisms described in the preceding sections in the WWT 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.

Table 6 and 7 show the results gathered using this setup for four benchmarks (OCEAN, BARNES, GAUSS, and SPARSE) using the Last-prediction scheme and the Intersection-prediction scheme. These tables 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 all stores encountered are tracked. Similar to the other two instruction-based predictions described in previous sections, the number of predictor entries required is very low for all programs. The total number of predictor probes gives an indication of the usage of the predictors (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

Statistics	LAST PREDICTION OCEAN	BARNES	GAUSS	SPARSE
Static Stores considered (all)	2499	1636	471	310
Average per node	79	51	15	10
Predictor entries allocated (all)	2499	1636	471	310
Average per node	79	51	15	10
Total number of predictor probes	1378698	106535	175564	813815
% returned non-null prediction	56%	63%	54%	3%
Total pre-send messages sent	402404	100696	87930	81756
% of non-null predictions	52%	150%	93%	335%
Pre-sends as % of data messages.	10%	3%	12%	1%
pre-sends rejected at consumers	168463	60106	2466	24085
% of total pre-sends	42%	60%	3%	29%
pre-sends accessed in consumers	158103	24984	85183	56421
% of total sent	39%	25%	97%	70%
Correct pre-sends accessed	135866	19439	85123	55032
% of total accessed	86%	78%	100%	98%
% of total sent	34%	19%	97%	67%
Incorrect pre-sends accessed	22237	5545	60	1390
% of accessed	14%	22%	0%	2%
% of total sent	5%	6%	0%	3%

Table 6: Statistics for producer-consumer Last-Prediction with speculative pre-send (32 nodes).

Statistics	INTERSECTION PREDICTION OCEAN	BARNES	GAUSS	SPARSE
Static Stores considered (all)	2500	1644	471	310
Average per node	78	51	15	10
Predictor entries allocated (all)	2500	1644	471	310
Average per node	78	51	15	10
Total number of predictor probes	1378658	106263	175564	813960
% returned non-null prediction	100%	97%	100%	100%
Total pre-send messages sent	247428	36530	87809	39480
% of non-null predictions	18%	34%	50%	5%
Pre-sends as % of data messages.	7%	1%	12%	1%
Pre-sends rejected at consumers	57268	15636	2032	10480
% of total pre-sends	23%	43%	2%	26%
pre-sends accessed in consumers	129465	14142	85612	28025
% of total sent	52%	39%	97%	71%
Correct pre-sends accessed	109501	11227	85567	26745
% of total accessed	85%	79%	100%	95%
% of total sent	44%	31%	97%	68%
Incorrect pre-sends accessed	19964	2915	45	1280
% of accessed	15%	21%	0%	5%
% of total sent	8%	8%	0%	3%

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

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 63% for BARNES. 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 the first two times a store is encountered.

For both schemes, 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 can be 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 columns. The total number of pre-sends is also expressed as a percentage of the non-null predictions.

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 percentage of pre-sends accessed in the consumer nodes and the percentage of them verified as correct determine the mis-speculation rate and ultimately affect the performance. For OCEAN and GAUSS these numbers are comparable for the two prediction schemes. For BARNES and SPARSE the second prediction scheme (Intersection-prediction) performs better. The percentage of the accessed pre-sends ranges from 25% (BARNES) to 97% (GAUSS) for the first scheme and from 39% (BARNES) to 97% (GAUSS) for the second scheme. The percentage of the accessed pre-sends that are verified through the CC-protocol as correct is high: for Last-prediction it ranges from 78% for BARNES (22% mis-speculations) to 100% for GAUSS (0% mis-speculations); for Intersection-prediction it ranges from 79% for BARNES to 100% for GAUSS. For all programs the percentage of correct pre-sends is comparable for the two schemes ranging from 19% to 97% for Last-prediction and 31% to 97% for Intersection-prediction. More sophisticated prediction schemes (e.g., two-level adaptive predictors [36,26]) have the potential to increase these percentages.

The optimization based on speculative execution is a tradeoff between bandwidth and latency: in hope of reducing the apparent latency of reads we send more data that, nevertheless, will be re-sent for verification. Our results show that: i) a significant number of pre-sends will allow processors to go ahead and execute useful work and ii) the pre-send traffic is low ranging from 1% to 12% of the total data traffic (see Table 6, “Pre-sends as % of data messages”). If these pre-sends are in the critical path of the program execution, then we could considerably reduce latency consuming a small amount of bandwidth.

7 Conclusions

In this paper we explore instruction-based prediction to optimize transparently 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 [7]). 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 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 instruction generates a new coherence event we can take action to optimize it.

To make the case that instruction-based prediction is a serious competitor both in terms of resource usage (in most cases using less than 128 predictor entries) and 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).
- **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).
- **Producer-consumer sharing**. We found that prediction can be accurate especially for some programs and a significant number of speculative pre-sends can be successful. However, the optimization trades bandwidth for latency and further research is needed to quantify its performance.

Future directions—We believe that this work will be a starting point for novel instruction-based prediction optimizations. Similarly to work that examined the coherence behavior of data [35] we need to examine the behavior of the instructions in relation to the coherency events and in relation to hardware parameters such as cache and block size.

Regarding the instruction-based predictions we study here, we considered them a starting point. In future work, we intent to examine alternative implementations (using other directory-based protocols), sophisticated predictors, and the combined effects of the different prediction schemes. Finally, we are investigating novel instruction-based predictions, such as the prediction of producers at consumer nodes as an alternative method to reduce access latency.

8 Acknowledgments

We would like to thank Alain Kägi, Doug Burger, Ravi Rajwar, David Wood, Mark Hill, and Guri Sohi for their helpful comments on drafts of this paper.

9 References

[1] S. G. Abraham, et al. “Predictability of Load/Store Instruction Latencies” *26th Micro*, November 1993.

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

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

[4] John Carter et al. “Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence.” *Proceedings of the Conference on the Principles and Practices of Parallel Programming*, 1990.

[5] 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.

[6] S. Chandra et al., “Where is Time Spent in Message-Passing and Shared-Memory Programs?” *ASPLOS VI*, Oct. 1994.

[7] T.-F. Chen, J-L Baer, “A Performance Study of Software and Hardware Data Prefetching Schemes.” *21st ISCA*, April 1994

[8] Convex Computer Corp., “The Exemplar System” 1994.

[9] Alan L. Cox, Robert J. Fowler, “Adaptive Cache Coherency for Detecting Migratory Shared Data.” *20th ISCA*, 1993.

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

[11] B. Falsafi et al., “Application-Specific Protocols for User-Level Shared Memory.” *Supercomputing ‘94*, Nov. 1994.

[12] A. Gonzalez et al. “A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality.” *ICS*, 1997.

[13] E. Hagersten et al., “DDM — A Cache-Only Memory Architecture.” *IEEE Computer*, Vol 25. No 9, September 1992.

[14] Mark D. Hill, “Multiprocessors Should Support Simple Memory Consistency Models”, *IEEE Computer*, Vol. 31 No. 8, Aug. 1998.

[15] IEEE Standard for Scalable Coherent Interface (SCI) 1596, IEEE 1993.

[16] Alain Kägi et al., “Techniques for Reducing Overheads of Shared-Memory Multiprocessing.” *ICS*, July 1995.

[17] S. Kaxiras, “Kiloprocessor Extensions to SCI.” *10th IPPS*, Apr. 1996.

[18] S. Kaxiras, J. R. Goodman “The GLOW Cache Coherence Protocol Extensions for Widely Shared Data.” *ICS*, May 1996.

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

[20] S. Kaxiras, “The Use of Instruction-Based Prediction in Hardware Shared-Memory.” *Univ. of Wisconsin CS TR-1368*, April 1998.

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

[22] James Laudon, Daniel Lenoski. “The SGI Origin: A cc-NUMA Highly Scalable Server,” *24th ISCA*, June 1997.

[23] Daniel Lenoski et al., “The Stanford DASH Multiprocessor.” *IEEE Computer*, Vol. 25 No. 3, March 1992.

[24] Tom Lovett and Russell Clapp, “STiNG: A CC-NUMA Computer System for the Commercial Marketplace.” *23rd ISCA*, May 1996.

[25] A. Moshovos at al., “Dynamic Speculation and Synchronization of Data Depenences.” *24th ISCA*, 1997.

[26] Shubhendu S. Mukherjee and Mark D. Hill “Using Prediction to Accelerate Coherence Protocols”, ISCA, 1998.

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

[28] G. F. Pfister and V. A. Norton, “Hot Spot Contention and Combining in Multistage Interconnection Networks.” *ICPP*, Aug. 1985.

[29] Steven K. Reinhardt et al., “The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers.” *Proceedings of the 1993 ACM SIGMETRICS*, May 1993.

[30] A. Saulsbury et al., “Missing the Memory Wall: The Case for Processor/Memory Integration.” *23rd ISCA*, May 1996.

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

[32] James E. Smith, “A Study of Branch Prediction Strategies.” *8th ISCA*, 1981.

[33] P. Stenström, M. Brorsson, L. Sandberg, “An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing,” *20th ISCA*, 1993.

[34] G. Tyson et al., “A New Approach to Cache Management” *28th Micro*, Nov 28 - Dec 1, 1995.

[35] W-D. Weber and Anoop Gupta, “Analysis of Cache Invalidation Patterns in Multiprocessors.” *ASPLOS III*, April 1989.

[36] T.-Y. Yeh and Yale Patt, “Alternative Implementations of Two-Level Adaptive Branch Prediction.” *19th ISCA*, 1992.