A Study of Three Dynamic Approaches to Handle Widely Shared Data in Shared-Memory Multiprocessors

Stefanos Kaxiras[1], Stein Gjessing[2], James R. Goodman[1]
[1] University of Wisconsin-Madison, {kaxiras,goodman}@cs.wisc.edu
[2] University of Oslo, Norway, steing@ifi.uio.no

Abstract— *In this paper we argue that widely shared data are a more serious problem than previously recognized, and that furthermore, it is possible to provide transparent support that actually gives an advantage to accesses to widely shared data by exploiting their redundancy to improve accessibility. The GLOW extensions to cache coherence protocols —previously proposed— provide such support for widely shared data by defining functionality in the network domain. However, in their static form the GLOW extensions relied on the user to identify and expose widely shared data to the hardware. This approach suffers because: i) it requires modification of the programs, ii) it is not always possible to statically identify the widely shared data, and iii) it is incompatible with commodity hardware. To address these issues, we study three dynamic schemes to discover widely shared data at runtime. The first scheme is inspired by read-combining and is based on observing requests in the network switches — the GLOW agents. The agents intercept requests whose addresses have been observed recently. This scheme tracks closely the performance of the static GLOW while it always outperforms ordinary congestion-based read-combining. In the second scheme, the memory directory discovers widely shared data by counting the number of reads between writes. Information about the widely shared nature of data is distributed to the nodes which subsequently use special wide sharing requests to access them. Simulations confirm that this scheme works well when the widely shared nature of the data is persistent over time. The third and most significant scheme is based on predicting which load instructions are going to access widely shared data. Although the implementation of this scheme is not as straightforward in a commodity-parts environment, it outperforms all others.*

1 Introduction

Shared-memory multiprocessing is only attractive if it can support a programming paradigm and programming languages efficiently. Numerous studies have characterized the sharing patterns of programs that have been written for such multiprocessors [25], and it is generally believed that widely shared data occur infrequently and do not significantly affect performance.

In this paper we argue that in fact widely shared data inherent in some parallel algorithms are a more serious problem than previously recognized, and that furthermore, it is possible to provide support that actually gives an advantage to widely shared data. The idea of read-combining [11] evolved because of the concern for network contention for widely shared data. Read-combining is highly dynamic, and reduces traffic in the network by recognizing that simultaneous requests can be merged. The probability of occurrence of simultaneous requests only becomes a factor when serious network contention extends the latency of individual requests, and in general, the best that combining can hope to achieve is a reduction in latency of access to widely shared data to the latency that would be experienced in an unloaded network.

However, it is possible to access widely shared data even faster than non-widely shared data. The presence of redundant copies of a datum in multiple caches throughout the system offers this possibility. This situation has some resemblance to cache-only machines [12], where data is quickly accessed if it resides in a cache close to the requester. If some data that is needed by a node are widely shared, it is likely that a cached copy of the data is closer than the original data in the home node. If an architecture can exploit this fact to improve accessibility of widely shared data, programmers would find that the best algorithms make extensive use of widely shared data rather than eschewing. Thus the potential for systems that provide high-quality support for widely shared data may be much larger than would be indicated by a sample of current shared-memory programs, which generally avoid such data wherever possible.

Several classes of sharing patterns in shared-memory applications have been identified (migratory, read-only, frequently-written sharing, etc.)[25]. Hardware protocols (e.g, pairwise sharing and QOLB [9] in SCI [1]) or software protocols, or application specific protocols have been devised to deal with such patterns effectively. Widely shared data that are read simultaneously by many —usually all— processors is a distinct sharing pattern that imposes increasingly significant overhead as systems increase in size [15]: when all processors read widely shared data there is much contention in the home node for servicing the requests as well as in the network around the home node which becomes a hot spot [22]; similarly, when the widely shared data are written there is a large number of invalidations (or updates) to be sent all over the system (i.e., non-locally). For many systems with no provision for efficient broadcast or multicasts these invalidations consume much network bandwidth, perhaps in a wasteful manner.

Previously, scalable coherence protocols have been proposed [13,20,21] but they were applied indiscriminately on all data. This diminishes the potential benefit since the overhead of the more complex protocols is incurred for all accesses. A tree-directory cache coherence protocol scales better than those based on centralized or linear list directories, but building a sharing tree does not come for free and doing so for data that are not widely shared may result in performance degradation for the most common access patterns. Only when the number of nodes that participate in the sharing tree is large, the overhead is sufficiently leveraged.

Bianchini and LeBlanc distinguished widely shared data (“hot” data) from other data in their work [5]. Similarly, the GLOW extensions for cache coherence protocols [15,16] are intended to be used exclusively for widely shared data. The distinguishing characteristic of the GLOW extensions is that they create sharing trees very well mapped on top of the network topology of the system, thus exploiting “geographical locality” [16]. Bennett et al also distinguished widely shared data in their work with proxies [4]. However, in all the aforementioned work widely shared data were statically identified by the user (the programmer or, potentially, the compiler). Such static methods of identifying widely shared data have three major drawbacks: i) user involvement complicates the clean shared-memory paradigm, ii) it may not always

International Conference on Supercomputing 1998

be possible to identify the widely shared data statically, and most importantly iii) mechanisms are required to transfer information from the user to the hardware; these mechanisms are hard to implement when the parallel system is built with commodity parts. This last consideration is crucial since vendors must leverage existing commodity parts (e.g., processors, main-boards, and networks) in order to drive development costs down and shorten the time-to-market.

Because of these reasons, in this paper we discuss how well we can dynamically identify and handle widely shared data. We propose and study three dynamic schemes to detect widely shared data. For reference we include in our comparisons congestion-based read-combining. The three novel schemes differ in where and how the detection takes place:

AGENT DETECTION: In this scheme (also discussed in [17]) the request stream is observed in the network, at the exact places where the GLOW extensions are implemented (namely at GLOW agents that are switch nodes in the network topology). Changes are required only in the GLOW-specific hardware without affecting other parts of the system making it the most transparent of the three dynamic schemes. Requests for widely shared data can be identified in the request stream if their addresses are seen often enough. The GLOW extensions are then invoked for such requests as in the static GLOW. This technique is similar in spirit to combining [11], but can better exploit requests scattered in time because the critical information hangs around in the combining node after a request is gone.

DIRECTORY DETECTION: In this scheme the directory is responsible for identifying widely shared data. This scheme resembles the limited pointer directories such as Dir$_i$B [2]. These directories switch from point-to-point messaging to broadcasting if the number or readers exceeds a threshold. Similarly in our scheme, the directory detects widely shared data (by keeping track of the number of readers) but —instead of broadcasting— it informs the nodes in the system about the nature of the data. After the nodes learn that an address is widely shared they use the GLOW extensions to access it. This DIRECTORY DETECTION scheme depends on widely shared data remaining as such through multiple read-write cycles.

INSTRUCTION-BASED (PC) PREDICTION: The last scheme is a novel method based on predicting which load instructions are likely to access widely shared data according to their past history. Instruction-based (PC) prediction is well established in uniprocessors but it has only been used for prefetching in multiprocessors [6]. This scheme does have implementation difficulties for commodity processors but on the other hand it is the most successful scheme we have studied.

The rest of this paper is organized as follows: In Section 2 we further motivate the importance of widely shared data and introduce *read-run* analysis. For the benefit of the reader unfamiliar with how GLOW handle widely shared data, we give a very brief description in Section 3. In Section 4 we expand on the problems associated with the static methods of identifying widely shared data. We introduce the dynamic methods in Section 5. In Section 6 and Section 7 we present our evaluations and results. Finally, we conclude in Section 8.

2 Implications of widely shared data

When widely shared data exist they are usually a very small percentage of the dataset of a program. Studies have also shown that the average degree of sharing (the number of nodes that simultaneously share the same data) in application programs is low [25]. These observations however, do not indicate the serious performance degradation resulting from *accessing* such data. Even if widely shared data are a negligible percentage of the dataset they can be detrimental to performance because the number of *reads* (or *invalidates*) corresponding to such data can be excessively large: widely shared data imply that a great many processors read them simultaneously.

To make this point clear we use the concept of *read-runs* as a tool to investigate sharing behavior (in Section 7 we are making extensive use of read-run analysis to explain the performance of the various schemes). Analogous to a *write-run* [8], we define a read-run for a data block as a sequence of reads (from any processor) between two writes (from any processor). The size of a read-run is thus directly related to the number of simultaneous cache copies in the system (if we ignore for a moment multiple reads because of replacements). In Figure 1 we show the sharing behavior of the GAUSS program running on 128 nodes (discussed further in Section 6). The left graph in Figure 1 is the read-run histogram for GAUSS. The horizontal axis is the read-run size and the vertical axis is the number of times a read-run appears in the execution of the program. Despite the fact that the number of read-runs of size 128 (corresponding to widely shared data) is negligible, and despite the modest degree of sharing of 2.75, about one half of all the reads (or alternatively invalidates) in the program correspond to widely shared data. The abundance of reads (invalidates) corresponding to widely shared data is evident in the right graph of Figure 1 which shows the number of accesses (reads or invalidates) that correspond to read-runs of various sizes (horizontal axis). The explanation for this is that each read-run of size *W* implies *W* reads or *W* invalidates and even the very few but large read-runs encompass as many reads and invalidates as a great number of small read-runs.

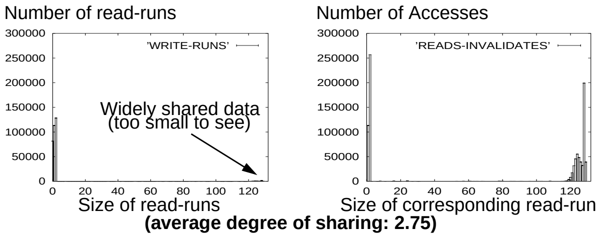

FIGURE 1. read-run histogram and corresponding accesses (Reads/Invalidations) for GAUSS running in 128 nodes.

Not only the accesses to widely shared data are numerous but they are also the most expensive in terms of latency because they create contention in the network and even worse contention in the home node directories of the data. When many nodes simultaneously access a single data block, each experiences much greater latency than if all nodes accessed different data blocks in different home node directories. Thus providing support for such accesses is essential for scalability.

3 GLOW extensions

The GLOW extensions provide support for widely shared data. They are independent of how the widely shared data are exposed to the hardware. For purposes of discussing the extensions we assume that special requests are used to access widely shared data. In subsequent sections we describe how to generate such special requests either statically or dynamically. GLOW extensions improve on previous efforts (EC [5], STP [21], STEM [13]) by embodying the following four characteristics:

Protocol-transparency: The GLOW extensions are not a protocol themselves but rather a method of converting other protocols to handle widely shared data. The functionality of the GLOW extensions is implemented in selected network switch nodes called GLOW agents that intercept special requests for widely shared data. These nodes behave both as memory and cache nodes using the underlying cache coherence protocol recursively: toward a local cluster of nodes they service, GLOW agents impersonate remote memory nodes; toward the home node directory, agents behave as if they were ordinary caches.

Geographical Locality: A sharing tree out of the GLOW agents and other caches in the system is constructed to match the tree that fans-in from all the sharing nodes to the actual home node of the widely shared data. GLOW captures geographical locality so that neighboring nodes in the sharing tree are in physical proximity.

Scalable reads: Since the GLOW agents intercept multiple requests for a cache block and generate only a new request toward the home node, a combining effect is achieved, eliminating hot spots [22].

Scalable writes: Upon a write, GLOW invokes in parallel the underlying protocol's invalidation or update mechanisms: on receipt of an invalidation (update) message, an agent starts recursively the invalidation (update) process on the other agents or nodes it services. The parallel invalidation (update), coupled with the geographical locality of the tree permits fast, scalable writes that require low bandwidth.

3.1 GLOW extensions to SCI

The first implementation of GLOW [16] is done on top of Scalable Coherent Interface (SCI) [1] —unlike most other directory-based protocols (such as DASH [18]) that keep all the directory information in memory, SCI distributes the directory information to the sharing nodes in a doubly-linked sharing list. The sharing list is stored with the cache lines throughout the system. A version of this implementation (described in [15]) defines the functionality of network switch nodes and it is fully compatible with current SCI systems.

SCI has two characteristics that make it an ideal match for GLOW. The first is that its invalidation algorithm is serial and a tree protocol is especially attractive for speeding up writes to widely shared data. The second concerns SCI topologies. SCI defines a ring interconnect as a basic building block for larger topologies. GLOW extensions can be implemented on top of a wide range of topologies constructed of SCI rings, including hypercubes, meshes, trees, butterfly topologies, and many others. GLOW can also be used in irregular topologies (e.g., an irregular network of workstations). In this paper, we study GLOW on highly scalable k-ary n-cube topologies [10] constructed of rings. As we mentioned in the general description, all GLOW protocol processing takes place in strategically selected switch nodes (the GLOW agents) that connect two or more SCI rings in the network topology.

GLOW agents cache directory information; caching the actual data is optional. Multilevel inclusion [3] is not enforced to avoid protocol deadlocks in arbitrary topologies. This allows great flexibility since the involvement of the GLOW agents is not necessary for correctness: it is at the discretion of the agent whether it will intercept a request or not. Details on how the sharing trees are created and invalidated are described in [15] and in [16].

4 Static approaches for wide sharing

In the previous section we described the GLOW extensions to handle requests for widely shared data. The GLOW mechanisms are independent of how the widely shared data are distinguished from other data. Here, we describe the static methods to define the widely shared data (also discussed in [16]). Identifying the widely shared data in the source program is only the first step. The appropriate information must then be passed to the hardware so special requests for widely shared data can be generated and invoke the GLOW agents. We divide the static methods depending on whether the programmer identifies the *data* that are widely shared or the *instructions* that access such data. The following two subsections describe the two alternatives.

Identifying addresses: This is the simplest method to implement and we have used it for the evaluations in later sections. A possible implementation of this method uses address tables, structures that store arbitrary addresses (or segments) of widely shared data. The address tables can be implemented in the network interface or as part of the cache coherence hardware. In both cases the user must have access to these tables in order to define and "un-define" widely shared data. Implementing such structures, however, is not trivial because of problems relating to security, allocation to multiple competing process, and address translation. The address tables could be virtualized by the operating system, but this solution is also unsatisfactory since (i) it requires operating system support and (ii) it will slow down access to these tables.

Identifying instructions: If specific code is used to access widely shared data, the programmer can annotate the source code and the compiler can generate memory operations for this code that are interpreted as widely shared data requests. We have proposed the following implementations:

- COLORED OR FLAVORED LOADS: The processor is capable of tagging load and store operations explicitly. Currently this method enjoys little support from commercial processors.
- EXTERNAL REGISTERS: A two-instruction sequence is employed. First a special store to an uncached, memory mapped, external register is issued, followed by the actual load or store. This special store sets up external hardware that will tag the following memory operation as a widely shared data operation. The main drawback of this scheme is that it requires external hardware close to the processor.
- PREFETCH INSTRUCTIONS: If the microprocessor has prefetch instructions they can be used to indicate to the external hardware which addresses are widely shared. Again, external hardware is required close to the processor.

The static approaches are plagued with a number of problems—also mentioned in the introduction—including the implementation problems of the hardware interfaces described herein. Thus in the next section we present dynamic approaches to alleviate these problems.

5 Dynamic approaches for wide sharing

Without the a priori knowledge of the addresses or the instructions that access widely shared data, this information needs to be discovered at run-time. In this section we describe three schemes to accomplish this. The first scheme relies exclusively on the GLOW agents for the detection, the second scheme relies on the memory directories and the third relies on detecting instructions that access widely shared data.

5.1 Agent detection

Conceptually a GLOW agent could intercept every request that passes through and do a lookup in its directory cache. This would result in slowing down the switch node, polluting the directory caches with non-widely shared data, and incurring the overhead of building a sharing tree for non-widely shared data. Instead, we want to filter the request stream and intercept only the requests that are likely to refer to widely shared data. The dynamic scheme described here is intended to perform such filtering.

Agents observe the request traffic and detect addresses that are repeatedly requested. Requests for such addresses are then intercepted in the same way as the special requests in the static methods. In an implementation of this scheme each agent, besides its ordinary message queues, keeps a small queue (possibly implemented as circular queue) of the last N read requests it has observed. The queue contains the target addresses of the requests, hence its name: *recent-addresses queue*. Using this queue each agent maintains a sliding window of the request stream it channels through its ports.

When a new request arrives at the agent, its address is compared to those previously stored in the recent-addresses queue. If the address is found in the queue the request is immediately intercepted by the agent as a request for widely shared data. Otherwise, the request is forwarded to its destination. In both cases its address is inserted in the queue. This method results in some lost opportunities: for example we do not intercept the first request to an address that is later repeated in other requests. Also, if a stream of requests for the same address is diluted sufficiently by other intervening requests we fail to recognize it as a stream of widely shared data requests. This scheme might also be confused by a single node repeatedly making the same request frequently enough to appear more than once within the agent's observation window (this could happen in producer-consumer or pairwise sharing). A safeguard to protect against this is to avoid matching requests from the same node against each other.

In the absence of congestion (i.e., when the agent's message queues are empty) we need to search the recent-addresses queue in slightly less time than it takes for a message to pass through the agent. Since the recent-addresses queue is a small structure located at the heart of the switch it can be searched fairly fast. Of course, the minimum latency through the switch will dictate the maximum size of the queue. For the switches we model in our simulations we expect that a size of up to 128 entries to be feasible. We have shown that this scheme is remarkably insensitive to the size of the recent-addresses queue and even queues as small as four to eight entries are quite able to distinguish widely shared data [17].

When the agents observe the reference stream only when there is congestion (in other words when multiple requests are queued in the agent's message queues) our method for detection of widely shared data defaults to read-combining as was proposed for the NYU Ultracomputer [11]. In this case, the observable requests are only the ones delayed in the message queues. The problem with such combining (that our method effectively solves) is that it is based too much on luck: requests combine only if they happen to be in the same queue at the same time which might happen only in the presence of congestion. Combining is highly dependent on the network timing and queuing characteristics as well as the congestion characteristics of the application [17]. In the Section 7 we show that we can effectively discover widely shared data using a sliding window whereas combining fails in most cases.

5.2 Directory detection

In this scheme the memory directory is responsible to discover widely shared data. In contrast to the previous scheme that is transparent to the rest of the system this scheme requires some modifications to the coherence protocols. This is feasible in many commercial or research systems where the cache coherence protocols are implemented as a combination of software and hardware and they can be upgraded (e.g., STiNG [19]).

The directory is a single point in the system that can observe the request stream for its data blocks. It is therefore in a position to distinguish widely shared data. In directories such as Dir_iX [2] the number of readers is readily available. However, in SCI where the directory keeps a single pointer to the head of the sharing list, the directory must count the number of reads between writes. A counter, associated with each data block, counts up for each read and it is reset with a write. Data blocks for which the corresponding counter reaches some threshold are deemed widely shared. In SCI this is a heuristic since the directory might incorrectly deem a data block as widely shared just by seeing multiple reads from the same node. However, even in this case the involvement of a GLOW agent is advantageous since it can rectify a pathological replacement situation by providing caching in the network. Determining whether read requests actually come from different nodes is possible if we keep a bitmap of the readers (similarly to Dir_iX). However, this would be an expensive addition to the SCI directory and we do not examine it further. In our evaluations we extended the SCI directory tag with a small 2-bit saturating counter.

The first time a data block is widely accessed all the read requests reach the directory without any intervention from the GLOW agents. If the directory finds that the data are widely shared it notifies the nodes in the system so the next time they access the block they will use special requests that can be intercepted by the GLOW agents. This information is transferred to the nodes when the data block is written. Upon a write the directory (or the writer in SCI) sends invalidation messages that notify the previously reading nodes that this data from now on is considered widely shared. Only the nodes that participated in the first read will learn this. The information is stored in each node in the invalidated caches with a tag value which we call "hot tag." If a node tries to access a "hot tag" it will send a request for widely shared data which will be handled by the GLOW agents. Alternatively, the information about which data blocks have been found to be widely shared can be kept in address tables similar to those described for the static GLOW. However, address tables would make the whole scheme more difficult to implement and are not examined further.

This scheme is based on the premise that data blocks are widely shared for many read-write cycles. Since the opportunity to optimize the first read-write cycle is lost, this scheme does not provide any performance improvement when data blocks are widely shared only once. Furthermore, it may degrade performance by incorrectly treating such data blocks as widely shared when they are not. A further consideration about this scheme is that it is easier to adapt from non-widely shared to widely shared than the other way around. If a data block is widely accessed only once, the directory will observe very few read requests between writes after the first read-write cycle. However, it cannot determine whether it sees very few requests because the data block is not widely shared anymore or because the GLOW extensions absorb most of the requests in the network. Even if the directory recognized a transition to a non-widely shared state, it would have to notify again the nodes in the system about this change. Fortunately, the "hot tag" concept provides a natural way to adapt from widely shared to non-widely shared. If the data block is not widely accessed the "hot tags" around the system will be replaced —they are invalid tags after all— and the nodes will lose the information that the block was widely shared.The only pathological case that can result from the inability of this scheme to adapt quickly from widely shared to non-widely shared is when the data block becomes migratory after it was widely shared. In this case many subsequent reads will incur the overhead of widely shared data because it will take many read-write cycles to erase the information about the nature of the data block from all the nodes. As of yet, we have not encountered this situation in any benchmark.

5.3 Instruction-based prediction

Instruction-based prediction (PC PREDICTION) in various forms has been proposed as a mechanism to accelerate serial programs. In this context it comes at a fairly small cost because it can be entirely encapsulated in the processor die and all the pertinent information is local. In parallel programs instruction-based prediction has been proposed for prefetching [6]. In this paper we propose instruction-based prediction as a mechanism to accelerate parallel programs with wide sharing. Specifically, we propose a mechanism 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 is widely shared and it 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.

Whether a load accessed widely shared data is judged by its miss latency: very large miss latency is interpreted as an access to widely shared data. Using latency as the feed-back information is not as farfetched as it sounds: we have observed that the access latency of widely shared data (without GLOW support) 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." For example, the latency of 128 requests going to the same node is much higher that the latency of 128 requests going to 128 different nodes. Microbenchmark results previously reported [15] confirm this observation. Although the latency criterion does not guarantee that we will apply GLOW *only* to widely shared data or to *all* widely shared data, it is a valuable criterion because it applies GLOW to accesses that are detrimental to performance. The latency threshold for widely shared data is a tuning parameter that can be set independently for different applications.

We have chosen to study a simple predictor. The first time a load misses and its latency is longer than a threshold (that represents the average latency of the shared-memory system) its PC is inserted in a small 16-entry fully associative cache with LRU replacement. In subsequent misses we probe this small cache using the PC of the load. In case of a hit in the cache we issue a special request for widely shared data.

Contrary to the uniprocessor/serial-program context where predictors are updated and probed continuously we *only* update the prediction history and probe the predictor in the case of a miss. This makes the prediction mechanism much less frequently accessed. Furthermore, its latency is not in the critical path since we only need its prediction on misses which are of significant latency anyway. Thus, it is not a potential bottleneck nor does it add any cycles to the critical path.

There are two choices for the location of such a mechanism: either inside the processor or outside. When the mechanism is inside the processor it is updated/probed when a load misses in the internal (L1) cache. When we have a hit in the prediction cache a special request for widely shared data is issued outside the processor. Since the type of the request only matters when we also have a miss in the external (L2) cache, it may lag a cycle behind the external (L2) cache access without degrading performance. The mechanism operates similarly when it is implemented outside the processor with the assumption that the PC of the corresponding load instruction is available outside the processor on an external (L2) cache miss. The resulting request could be delayed one cycle until a prediction is obtained. However, this cycle can be hidden by cache coherence protocol or network access latencies. In our study we do not distinguish between the two implementations since we model a processor with a single cache and a 16-entry prediction cache that does not delay the corresponding requests.

Wherever this mechanism is implemented it necessitates a custom approach: if it is inside the processor it requires a custom designed core and if it is outside the processor it requires that the PC of a load that misses be known outside the processor (we are not aware of any commercial processor with this feature). Despite this drawback we have two arguments why this method is important to consider: (i) it is highly successful in the context of this work, and (ii) we believe that such prediction mechanisms will be increasingly important in optimizing not only widely shared data but various access patterns such as migratory sharing or producer-consumer sharing — thus the cost of the prediction hardware will be amortized by many optimizations.

6 Experimental evaluation

A detailed study of the methods we propose requires execution driven simulation because of the complex interactions between the protocols and the network. The Wisconsin Wind Tunnel (WWT) [23] 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 (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. SCI has previously been simulated extensively under WWT [14] and the GLOW extensions have been applied to this simulation environment.

Hardware parameters: We simulated systems that resemble SCI systems made of readily available components such as SCI rings and workstation nodes. We have simulated k-ary n-cube systems from 16 to 128 nodes in two and three dimensions. The nodes comprise a processor, an SCI cache, memory, memory directory, a GLOW agent, and a number of ring interfaces. Although we assume uniprocessor nodes, GLOW applies equally well to symmetrical multiprocessor (SMP) nodes. In this case the GLOW agent resides in the network interface of the SMP node and is responsible to service the processors inside the node. 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 64 bytes. The cache size of 64KB is intentionally small to reflect the size of our benchmarks. Processor, memory and network interface (including GLOW agents) 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. Ring and bus interfaces as well as switches incur a 10 cycle latency for every message. We simulate contention throughout the network but messages are never dropped since we assume infinite queues. Each GLOW agent is equipped with a 1024-entry directory cache and 64KB of data storage. We stress here that the agent's data storage is optional and can be omitted without significantly affecting GLOW's performance [16]. To minimize conflicts the agent's directory it is organized as a 4-way set-associative cache.

Benchmarks: To evaluate the performance of GLOW we used five benchmark programs: GAUSS, SPARSE, All Pairs Shortest Path, Transitive Closure, and BARNES. Although these programs are not in any way representative of a real workload, they serve to show that GLOW can offer improved performance. Additionally, these programs represent the core of many scientific applications used for research in many engineering and scientific disciplines. We did not consider programs without widely shared data because such programs would hardly activate the GLOW extensions. The GAUSS program solves a linear system of equations (512 by 512 in our case) using the well known method of Gaussian elimination. Details of the shared-memory program can be found in [16]. In every iteration of the algorithm a pivot row is chosen and read by all processors while elements of previous pivot rows are updated. For the static method, we define a pivot row as widely shared data for the duration of the corresponding iteration. The SPARSE program solves $AX=B$ where A and B are matrices (A being a sparse matrix) and X is a vector. The main data structures in the SPARSE program are the N by N sparse matrix A and X, the vector that is widely shared (N is 512 for our simulations). In the static method we define vector X as widely shared data. The All Pair Shortest Path (APSP) and the Transitive Closure (TC) programs solve classical graph problems. For both programs we used dynamic-programming formulations, that are special cases of the Floyd-Warshall algorithm [7]. In the APSP, an N vertex graph is represented by an N by N adjacency matrix. The input graph used for the simulations is a 256 vertex dense graph (most of the vertices are connected). In the TC program an N by N matrix represents the connectivity of the graph with ones and zeroes. The input is a 256 vertex graph with a 50% chance of two vertices being connected. For both programs and for the static method the whole main matrix is defined as widely shared data. Finally the BARNES benchmark from the SPLASH suite [24] is an example of a program with very little widely shared data that can be identified statically. The main data structure in BARNES is an *octree* [24] whose top is widely shared. However, in the static version of GLOW we can only define the root of the octree as widely shared.

7 Results

In this section we present simulation results for the five programs and for the various system configurations (2-dimensional and 3-dimensional networks, 16 to 128 nodes). We use the 3 dimensional topologies to show how network scalability affects the GLOW extensions. In general our results show that GLOW offers greater performance advantage with higher dimensionality networks because it can create shorter trees with larger fan-out.

We compare SCI, static GLOW, congestion-based read-combining (COMBINING) and three versions of the dynamic GLOW. The first version employs a 128-entry recent-addresses queue to discover repetition in the addresses (we refer to this as AGENT DETECTION). In the second version the directory discovers the widely shared data and we refer to this as DIRECTORY DETECTION. In the third version we predict instructions that access widely shared data (we refer to this scheme as PC PREDICTION). We use a small 16-entry cache and a miss latency threshold of 1000 cycles to determine which misses correspond to widely shared data. As a reference the average latency of a "normal" access ranges from about 400 to 600 cycles for our benchmarks.

We measure execution time, and for each program we present speedup normalized to a base case. We selected the base case to be SCI on the appropriate number of nodes and with the appropriate 2- or 3-dimensional network. The actual speedups over a single node for the base cases are shown in Table 1. Note here that the programs we are using do not scale beyond 32 or 64 nodes for SCI. The GLOW extensions allow these programs to scale to 128 nodes but the performance difference between 64 and 128 nodes is small. A limitation of our simulation methodology is that we keep the input size of the programs constant and —because of practical constraints— relatively small. With larger datasets these programs scale to more nodes for the base SCI case and the GLOW extensions yield performance improvements for even larger numbers of nodes.

Figure 2 shows the normalized speedups for the GAUSS program. The two graphs present results for the 2- and 3-dimensional networks. GAUSS on SCI does not scale beyond 32 nodes, showing serious performance degradation with higher numbers of nodes. The GLOW extensions, however, scale to 64 nodes in 2 dimensions and to 128 nodes in 3 dimensions (although the additional speedup is negligible).

	2 DIMENSIONS					3 DIMENSIONS				
N.	GAUSS	SPARSE	APSP	TC	BARNES	GAUSS	SPARSE	APSP	TC	BARNES
16	16.6	5.9	11.7	14.4	7.2	16.8	6.09	11.8	14.5	7.2
32	25.3	8.6	19.4	20.1	8.5	26.3	10.01	19.9	20.6	8.5
64	22.9	12.7	21.0	19.3	12.6	25.3	15.73	21.9	20.1	12.6
128	12.9	12.5	14.7	13.2	—	16.2	18.54	15.9	14.3	—

Table 1: Actual speedups (over a single node) of the base cases, i.e. SCI with its linear sharing lists.

Static GLOW is up to 2.22 times faster than SCI in 2 dimensions and up to 2.44 times faster in 3 dimensions. COMBINING reaches about halfway the performance improvement of static GLOW while AGENT DETECTION remains within 5% of the performance of static GLOW. The DIRECTORY DETECTION scheme also works well staying within 10% of the static GLOW. Using PC PREDICTION results in the largest speedups over SCI (up to 2.27 times in 2 dimensions and up to 2.82 times in 3 dimensions).

To explain the behavior of the various GLOW schemes we examine how they appear to change the read-runs of the program from the directories' point of view. Specifically, for each scheme we plot the reads that correspond to read-runs of different sizes. The GLOW schemes “compress” these accesses toward the small read-run sizes. Each scheme's “compression” relates to its performance improvement.

Figure 3 plots the number of reads that correspond to read-runs ranging in size from 1 to 128. The horizontal axis is the size of the read-run, the vertical axis is the number of accesses (reads/invalidates). The data are for GAUSS on 128 nodes on a 2-dimensional network. These data represent what the SCI directories observe. The SCI directories count the same node multiple times in the same read-run if —because of replacements— it sent multiple requests, so the correspondence of the read-run size and the degree of sharing is not exact. Because these graphs contain very large and very small numbers we selectively use a logarithmic scale in the vertical axis. This tends to emphasize smaller numbers that would otherwise be invisible (as in Figures 1 and 9).

The SCI graph shows a large number of accesses corresponding to large read-runs. COMBINING, AGENT DETECTION, static GLOW, DIRECTORY DETECTION and PC PREDICTION all absorb a large number of requests in the network and as a result the directories see fewer requests between writes. Static GLOW compresses many accesses to read-runs of size 9 (pointed out in the graph). This number corresponds to 8 GLOW agents plus an extra node: for any widely shared data block in the 2-dimensional 128-node system (8 by 16 nodes) there are 8 agents covering all nodes except the data block's home node. In contrast, COMBINING which does not perform as well manages to compress the read-runs from a size of 128 down to a size of about 38. AGENT DETECTION manages to compress most of the large read-runs to a size of 24. This means that before all 8 GLOW agents are invoked for a widely shared block, 16 requests slip by and reach the directory. DIRECTORY DETECTION eliminates the largest read-runs and converts them to read-runs of size 9 (similarly to static GLOW). However, they still leave a significant number of accesses corresponding to large read-runs unaffected. PC PREDICTION gives the cleanliest spectrum of read-runs pushing most of the large ones down to a size of 9.

SPARSE scales to 128 nodes for both 2- and 3-dimensional networks although the increase in performance from 64 to 128 nodes in 2 dimensions is negligible (see Table 1 and Figure 4). For this program AGENT DETECTION outperforms static GLOW (in the 64- and 128-node systems in 2 dimensions and in the 128-node system in 3 dimensions). This is because SPARSE actually contains more widely shared data than just the vector *X* and AGENT DETECTION can handle them at run-time. AGENT DETECTION performs up to 1.29 times faster than SCI in 2 dimensions and up to 1.33 times faster in 3 dimensions. COMBINING fails to provide any significant performance improvement and DIRECTORY DETECTION performs on a par with static GLOW. PC PREDICTION is again the most successful (speedups of up to 1.33 and 1.50 for 2 and 3 dimensions respectively).

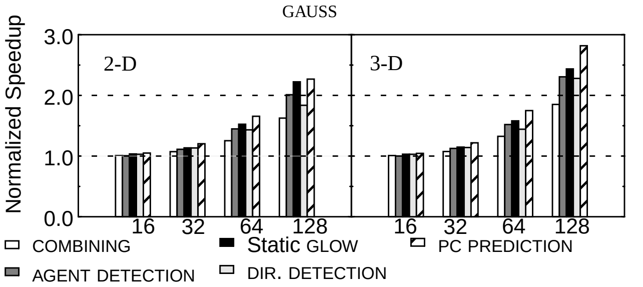

FIGURE 2. Normalized speedup (over SCI) for GAUSS in 2 and 3 dimensions (16 to 128 nodes).

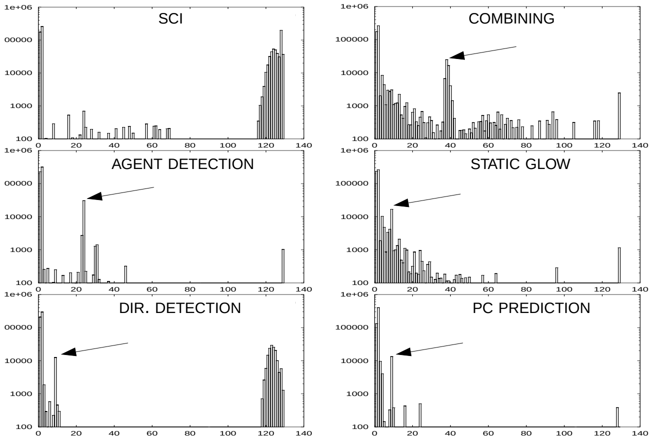

FIGURE 3. Read-run compression for GAUSS (128 nodes 2-dimensions). Y-axis (number of accesses) in logarithmic scale. Accesses corresponding to large read-runs are shifted toward smaller read-runs using GLOW extensions.

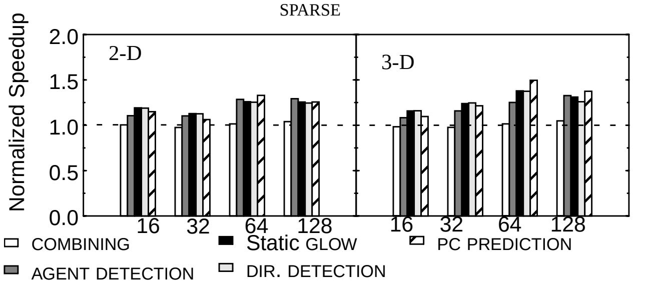

FIGURE 4. Normalized speedup (over SCI) for SPARSE in 2 and 3 dimensions (16 to 128 nodes).

Figure 5 shows the compression of read-runs for SPARSE (again on 128 nodes with a 2-dimensional network). COMBINING does not perform very well for SPARSE and this is also evident in its failure to affect the large read-runs. AGENT DETECTION spreads the largest read-runs all over the read-run spectrum with a center around 60. This means that the number of requests that slip through the agents before they detect widely shared data has significant variance. Static GLOW and PC PREDICTION perform very well, most of the time allowing the directories to see only 9 requests (8 GLOW agents and the local node).

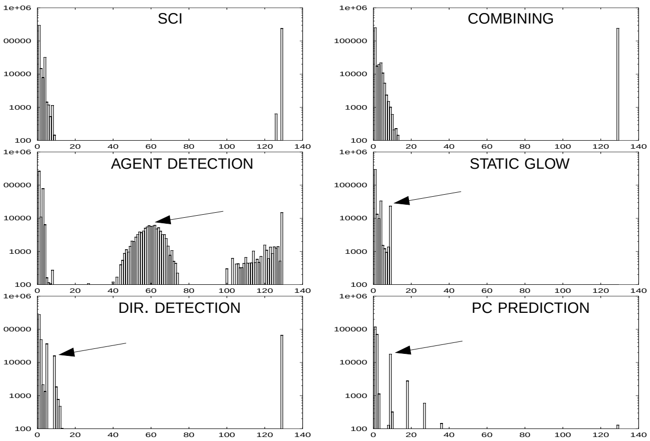

FIGURE 5. Compression of read-runs for SPARSE (128 nodes, 2 dimensions). Y-axis (number of accesses) in logarithmic scale.

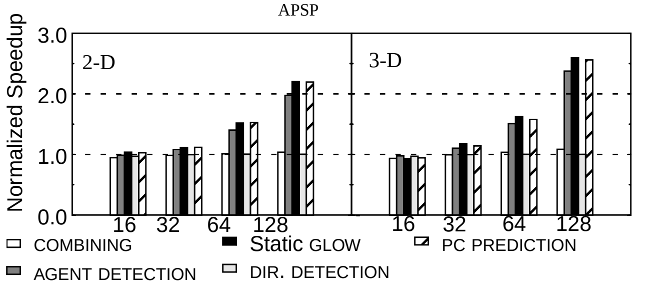

FIGURE 6. Normalized speedup (over SCI) for APSP for 2 and 3 dimensions (16 to 128 nodes).

APSP and TC exhibit similar behavior (we present graphically only the APSP speedups in Figure 6). With SCI, APSP does not scale beyond 64 nodes and TC does not scale beyond 32 nodes. For APSP, static GLOW is up to 2.20 times faster than SCI in 2 dimensions and up to 2.59 times faster in 3 dimensions. Similarly, for TC static GLOW is up to 2.22 and up to 2.64 times faster than SCI for 2 and 3 dimensions respectively. For both programs COMBINING and DIRECTORY DETECTION fail to show any performance improvement while AGENT DETECTION performs closely to the static GLOW. PC PREDICTION performs almost as well as static GLOW.

Since APSP and TC exhibit similar behavior we only demonstrate read-run compression for APSP (Figure 7). A significant percentage of the reads of the program correspond to large read-runs. As expected COMBINING is not successful in hiding accesses from the directories. Although it shifts accesses to smaller read-runs, it is not enough to make a difference in performance. AGENT DETECTION is quite successful compressing the read-runs to a size of around 30 (this translates to about 22 requests slipping trough 8 GLOW agents while the rest are intercepted). Static GLOW works very well leaving only read-runs of size 9 (similarly to the previous two programs). A common characteristic of the APSP and TC programs is that their data blocks are widely shared only once. Not surprisingly DIRECTORY DETECTION fails to change the read-runs of the program. The read-run histogram for PC PREDICTION is almost a carbon copy of the static GLOW and this explains their almost identical performance.

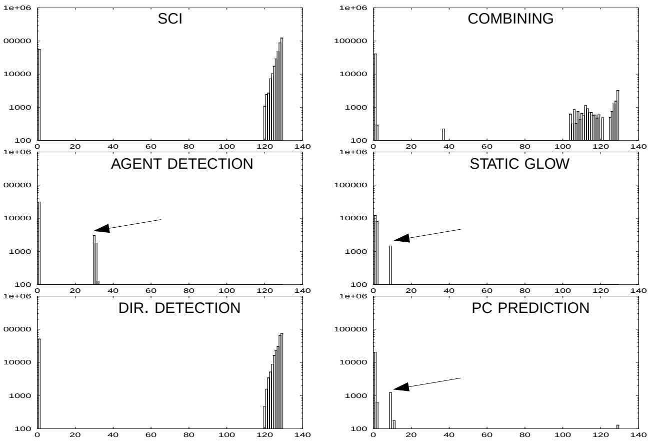

FIGURE 7. Read-run compression for APSP (128 nodes, 2 dimensions). Y-axis (number of accesses) in logarithmic scale.

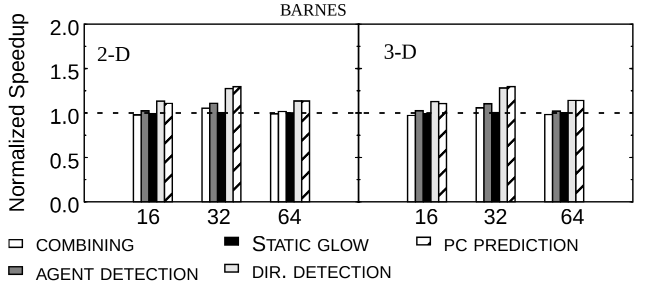

FIGURE 8. Normalized speedup (over SCI) for BARNES for 2 and 3 dimensions (16 to 128 nodes).

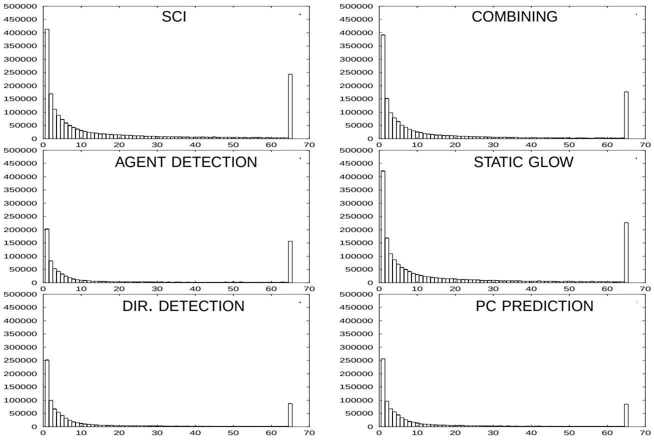

FIGURE 9. Read-run compression for BARNES (64 nodes, 2 dimensions). Y-axis (no. of accesses) *not* in logarithmic scale.

BARNES is not affected much by the dimensionality of the network and does not speedup considerably with higher numbers of processors (Table 1). This is due to the very small dataset we were able to simulate with our tools (4K particles). With larger datasets BARNES should exhibit better scaling. Nevertheless, the schemes we propose show speedups over SCI (Figure 8) —as much as 1.3 for 32 nodes. COMBINING and AGENT DETECTION as well as static GLOW do not show significant speedups over SCI. However, DIRECTORY DETECTION and PC-PREDICTION work very well, the former recognizing the top of the tree as widely shared and the latter identifying the instructions that access the top of the tree. The best speedups are 1.27 for DIRECTORY DETECTION and 1.3 for PC PREDICTION for 32 nodes. Read-run analysis shows that all schemes exhibit different behavior for BARNES than for the other benchmarks (Figure 9). All schemes redistribute both large and small read-runs among the smaller read-runs but without any particular peaks and in various degrees of success. DIRECTORY DETECTION and PC PREDICTION do the best job in reducing the large read-runs.

To summarize the results: AGENT DETECTION consistently tracks the performance of the static GLOW while COMBINING only works for one program (GAUSS). The results show that COMBINING is indeed sensitive to the congestion characteristics of the application. The behavior of COMBINING also changes depending on the network characteristics (e.g., link and switch latency, bandwidth) while the behavior of the AGENT DETECTION with regard to the number of intercepted requests remains largely unaffected. DIRECTORY DETECTION gives mixed results working only for three of the five programs. PC PREDICTION proved to be not only the most successful dynamic scheme but also better than the static in many cases.

8 Conclusions

In this paper we have shown that the number of accesses to widely shared data can be large even if the amount of widely shared data is small. The time spent in accessing such shared data can be considerable, hence there is considerable benefit in providing transparent hardware support for widely shared data. This benefit increases with system size since large systems suffer the most from widely shared data.

For economic reasons, hardware support for specific sharing patterns must be transparent and non-intrusive to the commodity parts of the system. The GLOW extensions to cache coherence protocols are designed with transparency in mind: they are implemented in the network domain, outside commodity workstation boxes, and they are transparent to the underlying coherence protocol. The GLOW extensions work on top of another cache coherence protocol by building sharing trees mapped well on top of the network topology thus providing scalable reads and writes. However, in their static form they require the user to define the widely shared data and issue special requests that can be intercepted by GLOW agents. This is undesirable for various reasons including implementation difficulties that inhibit transparency. In this paper we propose and study three schemes that can detect widely shared data at run-time and we compare them against SCI, static GLOW and combining. Each scheme exhibits different performance and cost characteristics, hence it is valuable to try to explain why they perform as they do. To this end we examined each scheme's effects on the read-runs of five programs.

The first scheme, AGENT DETECTION, discovers widely shared data more reliably than read-combining by expanding the window of the observable requests. Switch nodes remember recent requests even if these have long left the switch. Requests whose addresses have been seen in the window are intercepted (as requests for widely shared data) and passed to the GLOW extensions for further processing. The interesting characteristic of this scheme is that in large systems even a small window performs very well. This scheme achieves a significant percentage of the performance improvement of the static GLOW and has the potential to outperform the static version in programs where it is difficult for the user to define the widely shared data. Since it requires modification only in the switch nodes we believe it is the least intrusive of all the schemes. As for congestion-based combining (COMBINING) which is slightly simpler we have found that it is highly dependent on the congestion characteristics of the applications.

In the second scheme, DIRECTORY DETECTION, the directories are modified to discover the widely shared data by counting reads between writes. When a directory finds a data block to be widely shared it notifies the nodes in the system to subsequently request this data block as widely shared data. The applicability of this scheme is limited: it works well when data blocks are widely accessed more than once.

The third scheme, PC PREDICTION, is the most successful and it is based on predicting which load instructions are going to access widely shared data. Although its implementation is intrusive to the processor itself it offers the best performance. The potential for further optimizations based on PC PREDICTION could increase its value. Finally, in this paper we used read-run analysis to gain insight on how these schemes affect accesses to widely shared data. This tool enabled us to visualize these effects and reason about the behavior of the various schemes.

9 References

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

[2] A. Agarwal, M. Horowitz and J. Hennessy, "An Evaluation of Directory Schemes for Cache Coherence." *Proc. of the 15th ISCA*, pp. 280-289, June 1988.

[3] J. L. Baer and W. H. Wang, "Architectural Choices for Multi-Level Cache Hierarchies." *Proc. of the 16th ICPP*, 1987.

[4] A. J. Bennett, P. H. J. Kelly, J. G. Refstrup, S. A. M. Talbot. "Using Proxies to Reduce Controller Contention in Large Shared-Memory Multiprocessors" *EURO-PAR 96*, August 1996.

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

[6] T-F. Chen, J-L. Baer, "A Performance Study of Software and Hardware Data Prefetching Schemes." *Proc. of the 21st ISCA*, April 1994.

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

[8] S. J. Eggers, R. H. Katz, "A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation," In *Proc. of the 15th ISCA*, Jun. 1988.

[9] J.R. Goodman, M. K. Vernon, Ph. J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache Coherent Multiprocessors." *Proc. of ASPLOS-III*, April 1989.

[10] J. R. Goodman, P. J. Woest, "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor." In *Proc. of the 15th ISCA*, May 1988.

[11] A. Gottlieb et al "The NYU Ultracomputer: Designing a MIMD Shared-Memory Parallel Computer." IEEE Transactions on Computers, Vol. C-32, no 2, February 1983.

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

[13] R. E. Johnson, "Extending the Scalable Coherent Interface for Large-Scale Shared-Memory Multiprocessors." Ph.D. Thesis, University of Wisconsin-Madison, 1993.

[14] A. Kägi, N. Aboulenein, D. C. Burger, J. R. Goodman, "Techniques for Reducing Overheads of Shared-Memory Multiprocessing." *Proc. of the ICS*, July 1995.

[15] S. Kaxiras, "Kiloprocessor Extensions to SCI." *Proc. of the 10th IPPS*, April 1996.

[16] S.Kaxiras and J. R. Goodman "The GLOW Cache Coherence Protocol Extensions for Widely Shared Data." *Proc. of the ICS*, May 1996.

[17] S.Kaxiras and J. R. Goodman "Improving Request-Combining for Widely Shared Data in Shared Memory Multiprocessors", *Proc. of the 3rd International Conference on Massively Parallel Computing Systems*, 1998.

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

[19] Tom Lovett, Russell Clapp, "STiNG: A CC-NUMA Computer System for the Commercial Marketplace." In *Proc. of the 23rd ISCA*, May 1996.

[20] Yeong-Chang Maa, Dhiraj K. Pradhan, Dominique Thiebaut, "Two Economical Directory Schemes for Large-Scale Cache-Coherent Multiprocessors." *Computer Architecture News*, Vol 19, No. 5, pp. 10-18, September 1991.

[21] Håkan Nilsson, Per Stenström, "The Scalable Tree Protocol—a Cache Coherence Approach for Large-Scale Multiprocessors." *4th IEEE Symposium on Parallel and Distributed Processing*, pp. 498-506, 1992.

[22] G. F. Pfister and V. A. Norton, "'Hot Spot' Contention and Combining in Multistage Interconnection Networks." *Proc. of the ICPP*, Aug. 1985.

[23] S. K. Reinhardt, et al, "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers." *Proc. of the 1993 ACM SIGMETRICS*, May 1993.

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

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