The GLOW Cache Coherence Protocol Extensions for Widely Shared Data

Stefanos Kaxiras and James R. Goodman
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton St., Madison, WI 53706
{kaxiras,goodman}@cs.wisc.edu

Abstract—*Programs that make extensive use of widely shared variables are expected to achieve modest speedups for non-bus-based cache coherence protocols, particularly as the number of processors sharing the data grows large. Protocols such as the IEEE Scalable Coherent Interface (SCI) are optimized for data that is not widely shared; the GLOW protocol extensions are specifically designed to address this limitation. The GLOW extensions take advantage of physical locality by mapping K-ary logical sharing trees to the network topology. This results in protocol messages that travel shorter distances, experiencing lower latency and consuming less bandwidth. To build the sharing trees, GLOW caches directory information at strategic points in the network, allowing concurrency, and therefore, scalability, of read requests. Scalability in writes is achieved by exploiting the sharing tree to invalidate or update nodes in the sharing tree concurrently. We have defined the GLOW extensions with respect to SCI and we have implemented them in the Wisconsin Wind Tunnel (WWT) parallel discrete event simulator. We studied them on an example topology, the K-ary N-cube, and explored their scalability with four programs for large systems (up to 256 processors).*

1 INTRODUCTION

Current trends in research for distributed shared-memory parallel machines, favour the use of specialized protocols for different classes of shared data [4]. By classifying data according to its use, a compiler or programmer can exploit data sharing patterns to improve performance. This technique, which is crucial to the success of software-based schemes [19,23], can also be exploited to enhance the performance of hardware-based cache coherence methods. The Scalable Coherent Interface [11], for example, optionally implements the QOLB primitive [9] in this way.

Several classes of shared data have been identified, including migratory, read-only, etc., [4,26]. Widely shared data is a class of data that imposes increasingly significant overhead as systems increase in size. Studies have shown that the average degree of sharing (the number of nodes that simultaneously share the same data) in application programs is low [5]. This is somewhat misleading for two reasons. First, under most non-bus-based protocols, accessing widely shared data is very expensive and even a small amount of widely shared data can be a serious bottleneck in large systems. Second, because of the well-known performance problems, programmers tend to avoid algorithms that make extensive use of widely shared data. Many algorithms exhibit large scale sharing, albeit for a very small percentage of their dataset. As we will show, even in cases where less than one percent of the dataset is widely shared, the performance loss can be substantial.

For many otherwise attractive network topologies, the bisection bandwidth does not grow linearly with bandwidth needed for random communication. Such networks must exploit locality to scale. Methods that exploit locality are thus highly desirable. In this respect, widely shared data provides greater opportunity for exploiting locality than less widely shared data, which must be carefully mapped to the topology to exploit locality.

The GLOW extensions offer scalable reads and scalable writes to widely shared data. Scalable reads are achieved by caching directory information in the network, a technique inspired by the request combining originally proposed for the NYU Ultracomputer [10]. Because the directory information is more long-lived, however, this technique can be effective even when multiple requests are not generated simultaneously. Scalable writes are achieved by exploiting the topology to invalidate or update sharing nodes in logarithmic time. Previous examples of sharing tree protocols, such as the STEM Kiloprocessor Extensions to SCI [12] and others [3,20,21] have largely ignored locality in the network. In addition, previous work has attempted to treat all shared data in a uniform manner, even though the overhead for such support limits the benefit, or even reduces performance, for data that is not widely shared.

The GLOW cache coherence protocol extensions are specifically designed to handle accesses to widely shared data. The three main goals that guided the design of GLOW were:

TO CREATE SHARING TREES THAT MAP WELL ONTO TOPOLOGIES, thus achieving low latency protocol messages.

TO PROVIDE SCALABLE READS by exploiting request combining independent of the timing of requests

TO PROVIDE SCALABLE WRITES by using the tree structure to invalidate or update sharing nodes in parallel.

In this paper the GLOW protocol extensions are presented as extensions to the SCI cache coherence protocol. The presentation starts with an abstract overview of GLOW in Section 2. In Section 3 the SCI-specific implementation of GLOW is described along with a brief discussion of other possible implementations. Section 4 elaborates on the GLOW extended SCI protocols. Section 5 defines the simulation environment and defines parameters used in our study. Section 6 describes the benchmarks and presents the results of the simulations. Section 7 reviews some important relevant work. Finally Section 8 summarizes our work.

This work was supported in part by NSF Grants CCR-9207971 and CCR-9509589, funding from the Apple Computer Advanced Technology Group, an unrestricted grant from the Intel Research Council, and equipment donations from Sun Microsystems. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618, with matching funding from the University of Wisconsin Graduate School. This work was also partially supported by the National Center for Supercomputing Applications and utilized the Thinking Machines CM-5 at NCSA, University of Illinois at Urbana-Champaign.

2 GLOW PROTOCOL EXTENSIONS

Large-scale systems are likely to be based on topologies where requests going toward memory—as well as data distributed back to nodes—follow traffic patterns that naturally form trees. The GLOW protocol extensions are intended to capture and exploit these trees. The general structure of GLOW trees is independent of the timing of the requests (something that is not true for many other tree protocols). This feature leads to increased locality in the network, resulting in low latency protocol messages.

Here we assume systems comprising nodes connected in a network. A node can be a processor with a cache, a memory module with a memory directory, a switch that routes requests from one network link to another, or any combination of the above. Any node that includes a switch is called a *switch node*.

GLOW itself does not provide transactions for reading and writing shared blocks from scratch. Instead, it works as an enhancement to another cache coherence protocol, borrowing its mechanisms. The functionality of the GLOW extensions is implemented in selected network switch nodes called GLOW *agents*. These agents intercept requests for widely shared data. Other requests (for non-widely shared data) remain unaffected and proceed through the switch nodes at full speed. The GLOW agents keep directory information (and optionally, data). Nodes become children of a GLOW agent when their requests for widely shared data are serviced by that agent. The GLOW agents themselves join the sharing tree when they send their own requests toward the remote memory in order to service other nodes or other agents. The sharing tree is thus created recursively, bottom-up. The sharing nodes will be the leaves of the sharing tree and the GLOW agents will comprise all of the internal nodes. All of the sharing tree, other than the leaves, represents a form of hierarchical caching of directory information. In the various implementations of GLOW it is neither necessary to have a copy of the data in the GLOW agents nor is it necessary to maintain multilevel inclusion [2] in the hierarchical caching of the directory information. These properties are important because they provide attractive implementation choices for avoiding deadlock and indiscriminate invalidation of descendents in case of replacements.

Since the GLOW agents intercept multiple requests for a cache line and generate only a new request toward the 'home node,' the same effect as request combining is achieved. The memory directory receives only the requests of a few agents, in effect eliminating hot spots [22].

An example of a general K-ary sharing tree that can be formed using GLOW agents is shown in Figure 1. In this figure the memory and memory directory in the 'home node' are represented by the square in the top left corner, nodes that have a copy of the data block by white circles and GLOW agents by black circles. The GLOW agents can impersonate the remote memory, however far away it is, on a local cluster of nodes and thus satisfy their requests locally. In levels of the tree between itself and the root, an agent itself behaves as if it were an ordinary node sharing the data block. This behaviour of the agents allows the use of the underlying protocol mechanisms for accessing the shared data. For example, in Figure 1, as far as the memory directory is concerned, it points to a number of sharing nodes, whereas in reality it points to the first level of agents that hold the rest of the sharing tree. Similarly as far as the nodes that truly share (the leaves of the sharing tree) are concerned, they have been serviced directly by memory where in fact they were serviced by the agents impersonating the memory. GLOW can fall back to the underlying protocol at any time, thus providing flexibility at handling extraordinary cases.

GLOW invokes in parallel the underlying protocol's invalidation or update mechanisms for the writes to widely shared data. On receipt of an invalidation (or update) message, an agent starts the invalidation (or update) process on the other agents or nodes it services. This parallel invalidation or update of the sharing tree permits fast, scalable writes.

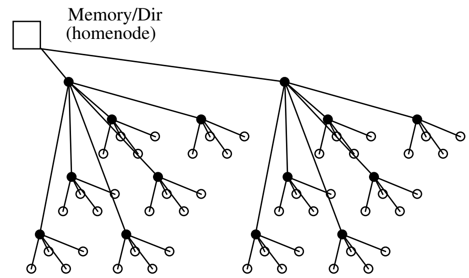

FIGURE 1. Logical K-ary sharing tree. Black nodes are GLOW agents, white nodes are sharing nodes.

In order for the GLOW protocol extensions to be usable the widely shared data in a program must be identified and a method provided to generate the specially tagged requests for such data. We believe that in many programs the programmer is able to identify in the source code the widely shared structures. We have not yet investigated whether this can be done automatically by a compiler. In cases where discrimination is difficult, profiling tools can possibly identify such data.

In addition, a method is required to permit the program to convey this information to the hardware. While specific implementations depend on the opportunities presented by the processor used, we describe several general techniques.

COLOURED OR FLAVOURED LOADS: This is the first choice if the processor is capable of tagging load and store operations explicitly. The compiler can then directly indicate which memory operations should be treated as widely shared.

MEMORY MAPPED: Widely shared data can be allocated in certain regions (or pages) of memory. Any address that falls into these regions is treated specially. The programmer or compiler identifies the ranges of addresses that are widely shared.

EXTERNAL REGISTERS: A two-instruction sequence is employed. First a store to an uncached, memory mapped, external register is issued, followed by the read or write. This special store sets up external hardware that will tag the following load. Again the compiler inserts stores to the external registers just before the desired load instructions. While it is desirable that the two operations be performed atomically, this is not a requirement for correctness, so no extraordinary measures need be employed to handle discontinuities caused by exceptions.

The above methods for generating specially tagged requests are all static. We are currently examining ways to discover widely shared data at run-time. The designation as widely shared is made at the home directories by counting the number of reads between writes for each data block. However, the methods of building the sharing trees considered so far for the dynamic GLOW are topology dependent.

3 GLOW EXTENSIONS TO SCI

In this section we briefly describe the SCI cache coherence protocol and various GLOW implementations on it. We also discuss a GLOW implementation for a full map protocol ($\mathrm{DIR_iX}$ [1]). While the version described herein is not fully compatible with SCI, such a compatible version is a proposed component of the IEEE P1596.2 Kiloprocessor Extensions to the SCI standard currently under development. The details of this implementation are reported elsewhere [15].

3.1 BACKGROUND: SCI

The ANSI/IEEE standard 1596 Scalable Coherent Interface represents a robust hardware solution to the challenge of building cache-coherent, shared memory multiprocessor systems. It defines both a network interface and a cache coherence protocol. The network interface provides a 1GByte/sec ring interconnect and a set of defined transactions. Scott *et al*. showed [25] that a ring can accommodate small numbers of high-performance nodes, in the range of four to eight. For larger systems, more complex topologies can be constructed from rings (*e.g.*, K-ary N-cubes, multistage topologies) [13]. In these topologies some or all nodes provide low-latency switches to connect more than one ring.

SCI defines a distributed, directory-based cache coherence protocol. 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. The memory directory has a pointer to the last node that joined the sharing list, which is called *head* of the list. A node can join the sharing list by notifying the memory directory to point to it, and attaching in front of the previous head. The node that joins the sharing list gets the data either from the memory or, if memory has no valid copy, from the previous head of the sharing list. As the head of the sharing list, a node may have both read and write permission to the cache line; all other nodes in the sharing list have only read permission. The SCI sharing list is depicted in Figure 2 where caches, represented by small rectangles, are connected in a doubly-linked list with their *backward* and *forward* pointers.

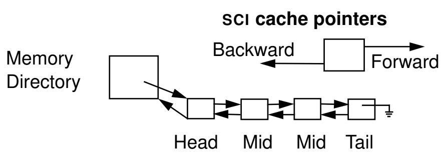

FIGURE 2. SCI Sharing List. The last node connected to the memory directory is the head of the list. The first node in the list is the tail and the rest are middle entries.

A node can also leave the sharing list by communicating with its neighbours. This operation, called *Rollout*, takes place in two situations: (1) When there is a conflict in a cache and the cache line must be replaced, the node rolls out of the sharing list. (2) In order to write the cache line, a node other than the head of the sharing list must first rollout, then become the new head.

As the head of a sharing list, a node can invalidate, or purge, the rest of the sharing list upon writing the cache line. The head sends invalidation messages serially to the other nodes in the sharing list. Each of these nodes acknowledges the invalidation by returning its forward pointer (*i.e.*, the pointer to the next node to be invalidated) to the head and invalidating its data.

3.2 GLOW ON SCI

The GLOW agents in an SCI network are placed where requests are switched from one ring to another. They can be stand-alone on the network or part of another SCI node. We have studied the second case as it is likely that agents will be implemented as part of the ring interface card on a workstation. Our view of this case is depicted in Figure 3 where we show a GLOW-capable workstation SCI node. The switch is not slowed down by the GLOW agent as far as normal SCI requests and responses are concerned. These messages are unaffected by the existence of the agent. Only specially tagged requests are pulled out of the network and are delivered for further processing to the GLOW agent.

Our experience in developing the specification for the GLOW protocol in C-code indicates that an implementation of a GLOW agent is very similar to that of an SCI cache controller but with less complexity. The GLOW agent implements a small part of the SCI cache coherence protocol (just the basic functionality with no options) and part of the directory controller. Some added complexity stems from the fact that GLOW has to deal with multiple pointers per cache line instead of the two and one pointer of the SCI cache and SCI directory respectively. A GLOW controller also requires some amount of memory (to be used as a directory cache). Optionally, data storage memory can be included.

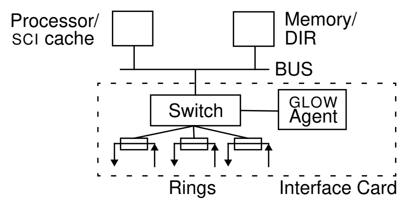

FIGURE 3. GLOW agent/SCI node: The switch, that connects the three rings and the node, is augmented with a GLOW agent (directory cache and control). Normal SCI messages (requests and responses) are unaffected by the existence of the GLOW agent. Only specially tagged requests are delivered to the GLOW agent.

Using caching in the agents the sharing trees are built out of small, linear SCI sharing lists. In a well formed GLOW sharing tree each SCI list is confined to one physical ring. Although the sharing trees are not optimal in the theoretical sense (because some of their parts are linear) the latency of the protocol operations is low because messages are generally confined to a single ring where they travel very fast. In addition, the shorter distances travelled by these messages can dramatically reduce network traffic under the appropriate circumstances.

The SCI lists are created under the agent when requests from nodes on a ring are intercepted and satisfied either directly by the agent or by another close-by node (usually on the same ring). These lists are called *child lists* and the agent is their *parent*. Without GLOW, requests would go all the way to the remote memory and would join a global list. As we have explained the agent has a dual personality: toward its children it behaves as if it were an SCI memory directory; toward its parent it behaves as if it were an ordinary SCI cache. The agent impersonating the remote memory on the local ring links the requesting nodes in the small child lists. In order to keep the child lists confined to their rings agents can accommodate multiple child lists per tree tag. Toward the ring the leads to the memory node the tree tag appears to have a backward and forward pointer just like any other SCI cache.

The agent itself, along with other nodes in the higher level ring (the next ring toward the home node), will in turn be serviced by yet a higher level agent impersonating the remote memory on that ring. This recursive building of the sharing tree out of small child lists continues up to the ring containing the true remote memory. Since there is an overhead in building the tree (*i.e.*, to invoke all the levels of agents) we use GLOW only for the widely shared data.

3.3 GLOW ON FULL-MAP CACHE COHERENCE PROTOCOLS

Similarly to the way GLOW is implemented on top of SCI, it can be implemented on top of other traditional full map cache coherence protocols such as DIR_iX or DASH. In this case the directory information in the agents can be kept as full or partial bitmaps. When partial bitmaps are used the agents can only point to the immediate nodes they service or alternatively to all the nodes that can potentially be their descendents in a sharing tree. In any case, multilevel inclusion can be ignored as long as the memory directory has a full map directory (or a LimitLESS directory [5]) that can accommodate nodes or agents that do not appear in their proper position in the sharing tree. In this way the absence of an agent in the sharing tree will only affect performance since the nodes, or other agents it would service, will be serviced by the memory directory in the usual manner specified by the underlying protocol.

3.4 GLOW ON AN EXAMPLE TOPOLOGY

GLOW extensions can be implemented on top of a wide range of topologies, including hypercubes; meshes; trees; butterfly topologies [13] and many others. GLOW can also be used in irregular topologies (*e.g.*, an irregular network of workstations). We chose to implement and study the GLOW extensions on a popular topology we believe is a likely candidate for implementation. It is the K-ary N-cube topology made of rings. We stress here that the GLOW extensions do not depend on this topology and can be equally well applied to other topologies.

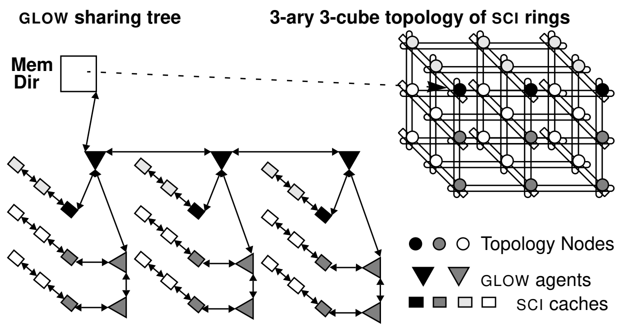

FIGURE 4. GLOW Tree on *3*-ary *3*-cube. The agents (triangles) and SCI caches (rectangles) of the same shade reside in the same physical node (circle).

In our example K-ary N-cube topology the routing of messages is fixed in dimension-order to have deadlock free routing. At the place where GLOW requests switch to the ring of the next dimension they are intercepted by the agent and processed. A new GLOW request is generated and continues in the next dimension. The response to an intercepted GLOW request travels back one dimension to the ring from which the request originated.

In Figure 4 the mapping of a *k*-ary sharing tree on the 3-ary 3-cube is shown. The GLOW agents are represented by triangles and the SCI caches by circles. Observe that lists are in hierarchical levels and the agents can have child lists in different levels. The nodes in the lists are drawn in perfect order which is not the usual case. The nodes' positions within a list depend on the timing of their requests as seen by the GLOW agent. (The general structure of the tree, however, does not depend on the timing of requests.)

4 GLOW EXTENDED SCI PROTOCOLS

In this section we elaborate on how the GLOW trees are constructed on SCI, how they are invalidated and how replacements are performed in SCI caches and in the directory caches of the GLOW agents.

4.1 CONSTRUCTION

When a node needs to read a remote cache line designated widely shared, it initiates a specially tagged request addressed to the home node of the cache line. At the point where it would change rings, the message is (optionally) intercepted by an agent. The request can be completely ignored and passed toward the next hierarchical level if there is danger of deadlock. This might occur, for example, when the request requires directory storage (a tree tag) that is occupied by another tree tag in a transient state. The agent checks in its directory storage for information relating to this cache line.

IF THE AGENT DISCOVERS AN ENTRY, it responds to the request directly, prepending the requesting node to the appropriate child list. The node will get the data from the previous head of the child list, or from the head of one of the agent's other child lists, or from the agent itself, if it keeps copies of the cache lines. Agents do not repeat their requests toward the home node to fetch the data as this would put them in the sharing tree twice (see also Section 4.2).

IF THE AGENT FINDS NO ENTRY, it sends its own request for the cache line toward the home node. The requesting node becomes the first child of the agent. As soon as the agent receives a copy of the line it passes it to the waiting node(s).

In the time it takes for an agent to get the data, new requests may arrive. Each of the new requesting nodes is instructed to prepend to the appropriate child list and wait for the data. Three options have been identified for distributing the data to a child list where all nodes are waiting for the agent to get the cache line:

TAIL-TO-HEAD: As is currently defined in SCI. The agent instructs the node to prepend to the child list and request the data from the old head (exactly as the SCI memory would do). The very first node to request the data becomes the waiting tail of the child list. It will eventually get the data from the agent. Note this scheme requires the agent to maintain two pointers: one to the tail for providing the data, and one to the head for adding additional nodes. This scheme is the most compatible with SCI and its details are discussed elsewhere [15].

HEAD-TO-TAIL: If a requesting node prepends to a child list, where the old head is waiting for the data, then it assumes responsibility to forward the data. When the agent gets the data it will pass it to the waiting heads of its child lists. The heads will then forward the data toward the waiting tails. A characteristic of this scheme is that it increases the variance of the latency experienced by the requesting nodes.

BROADCAST ON THE RING: This is a broadcast confined within a single ring. All nodes that are waiting will receive the broadcast and consequently the data.

When all nodes simultaneously read a cache line (for example, after a global barrier), construction of the tree is quite fast: All the child lists can be constructed in parallel. Copies of the cache line are distributed down the tree concurrently with list construction.

4.2 ROLLOUT

An ordinary node rolls out either because of a cache line replacement or in order to become head for writing the data. The standard SCI protocol is applied for rollout. Agents may roll out because of conflicts in their directory (or data) storage or because they are left childless. Childless agents are not permitted in the tree unless they also cache the data—the SCI protocol does not permit caches already in the sharing list to issue a second request, since this would put them in the list twice. As soon as the last child rolls out the agent rolls out too. When an agent finds itself childless it is only connected to the tree with its forward and backward pointers. In this case the rollout is the standard SCI rollout.

When an agent with children must roll out because of a conflict in its directory storage, it must first deal with its child lists, and then roll out as before. Two approaches for dealing with the child lists are described below. When the first is used the rollout is called DESTRUCTIVE because it destroys parts of the tree. When the second is used the rollout is called LINEARIZING because it connects the child lists into extended linear lists.

4.2.1 Destructive Rollout

The simple solution is borrowed from hierarchical caches that enforce multilevel inclusion. When a node rolls out, it invalidates all of its descendents using the invalidation algorithm described in Section 4.2.3. When the invalidation completes, the agent is childless and it can rollout as usual. This simple scheme has great appeal because of its simplicity. However, it can lead to thrashing behaviour, constantly invalidating nodes which then immediately repeat their requests for the cache line when the amount of widely shared data in the system significantly exceeds the directory capacity of the agents. Such behaviour is difficult to avoid because the node rolling out does not have information about the access frequency of data being referenced by nodes in its child lists. Nevertheless, this scheme may be generally viable because in many programs the amount of widely shared data (actively accessed at any point in time) is rather low. In this paper this scheme is used because of its simplicity.

4.2.2 Linearizing Rollout

This scheme attempts to preserve the sharing tree as much as possible, degrading the structure of the tree gracefully. Note that it is possible only because GLOW does not require multilevel inclusion. The LINEARIZING rollout is based on concatenating the child lists and subsequently substituting the concatenated child lists in place of the agent in tree. The child lists are concatenated, *i.e.*, they are chained tail-to-head into a single, linear list. The head of the first child list connects to the tree in the position of the agent's *backward* pointer. The tail of the last list connects to the tree in the position of the agent's *forward* pointer. In order to chain the child lists efficiently the GLOW agent keeps an extra pointer to the tail of each child list (as in the case of Tail-to-Head data distribution scheme). The chaining process involves only the heads and tails of the child lists. The process is shown graphically in Figure 5.

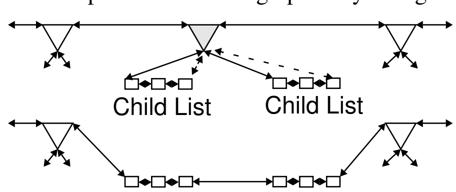

FIGURE 5. Linearizing rollout. The agent that rolls out (gray triangle) chains its two child lists into one (tail to head) and substitutes the resulting list in its place. Notice the two extra pointers the GLOW agent keeps to the tails of its child lists.

In the LINEARIZING rollout protocol, when an agent rolls out and later rejoins the tree the relation of the agent and its children is lost at the point of the rollout. If the agent rejoins the tree it will do so in another position in the linear list of the appropriate hierarchical level. New requesting nodes that would otherwise become heads in the old child lists, will join newly created child lists under the agent. Although the sharing tree degrades over time as the agents leave and rejoin, leaving child lists scattered over multiple rings, this scheme is potentially more effective than the DESTRUCTIVE rollout. Two arguments support this claim: First, there is minimal interference to other nodes: While in the DESTRUCTIVE rollout many shared copies that are potentially in use are invalidated, this scheme requires only the participation of the heads and tails in the concatenation of the child lists. Second, the latency of invalidating a subtree can be a lot higher than the latency of chaining the child lists together and substituting the agent. Therefore replacements in the agent's directory storage can be much faster. The LINEARIZING rollout is used in the SCI compatible version of GLOW and additional details can be found elsewhere [15].

4.3 INVALIDATION

Before a node can modify a cache line, it must first become the head of the top-level list connected directly to the memory directory in the home node. In this position the node is the root of the sharing tree, the only node granted write permission. A node has to rollout from the sharing tree before becoming root. After the cache line is written, invalidation messages are forwarded down the tree in parallel. When the invalidation messages reach the tails of the lists they invalidate themselves and send an acknowledgment back. GLOW agents wait for the acknowledgment of all the invalidation messages they forwarded, invalidate themselves, and return their own acknowledgment. This two-phase method (invalidation distribution down the tree and then acknowledgement return to the root) can be used to implement an update protocol where updates are serialized at the root of the tree, the new values are distributed down the tree and they are confirmed with acknowledgments all the way back to the root. In a well-structured tree, invalidation is fast because all the messages are exchanged locally—between nodes within the same ring. Note here that, for a scheme that can exploit broadcast effectively, confining lists to a single ring potentially offers very significant reductions both in message traffic and invalidation latency.

The IEEE P1596.2 GLOW Kiloprocessor Extensions use SCI's invalidation mechanism, as described in Section 3.1, rather than full request forwarding described in this paper. There, all the agents in parallel assume a role similar to the head nodes in SCI and invalidate their child lists in the same manner as the SCI protocol [15].

5 SIMULATION METHODOLOGY

To evaluate the performance of GLOW we used four Scientific benchmarks. We do not claim that these programs are in any way representative of a real workload. We did not consider programs without widely shared data because such programs would never activate the GLOW extensions. Most parallel programs access little shared data, at least in part because most programmers eschew such algorithms, believing—not incorrectly for many parallel systems—that such algorithms will not perform well. We have simulated K-ary N-cube systems from 16 to 256 nodes in two and three dimensions. In the following sections we present the speedups that SCI and GLOW achieve with respect to the parallel program running in one node. We have shown previously† that when the benchmark is scaled as processors are added, SCI performance degrades much more rapidly than with the GLOW extensions. In this evaluation we have kept the input size constant, decreasing the work that each processor must perform as more processors are added. In some cases we have explored beyond the useful limit of the benchmark, where the SCI implementation actually slows down because the heavy network traffic comes to dominate the execution time of the program. This range shows, however, that when the network latencies dominate the performance, GLOW becomes increasingly attractive because of the reduced average path length.

5.1 IMPLEMENTATION OF GLOW IN THE WWT

The Wisconsin Wind Tunnel [24] 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 interested reader can find the results of the simulations without network contention and for up to 512 nodes in [16]. Micro-benchmark results for the IEEE P1596.2 GLOW Kiloprocessor extensions, a variation of the protocol described herein can also be found in [15].

the simulator takes control and takes the appropriate actions defined by the simulated protocol. The WWT keeps track of virtual time in processor cycles. The Scalable Coherent Interface has previously been simulated extensively under WWT[14] and the GLOW extensions have been applied to this simulation environment.

We consistently made conservative choices to establish a lower bound on the performance of GLOW. We have studied the DESTRUCTIVE rollout algorithm, which has lower performance than the LINEARIZING rollout. To date the first two data distribution schemes, *Tail-to-Head* and *Head-to-Tail* have been implemented. The third scheme, *Broadcast-on-the-ring*, is more complex and requires broadcast not easily simulated in the WWT SCI. The data distribution scheme for all reported experiments is *Head-to-Tail.*

5.2 WWT NODE

In this section we describe the models we used for the system components and the network. We simulated a node comprising a processor, an SCI cache, memory, memory directory, a GLOW agent, and a number of ring interfaces. We assumed 200MHz processors which execute one instruction per cycle in the case of a hit in their cache. Each processor is serviced by a 256K 4-way set-associative cache with a cache line size of 64 bytes. Processor, memory and network interface (including GLOW agents) communicate through a 66MHz 64-bit bus.

An SCI K-ary N-cube network of rings with a 200MHz clock is assumed; 16 bits of data can be transferred every clock cycle through every link. We simulate contention throughout the network. However our model deviates from the SCI specification in that we did not simulate SCI's bandwidth allocation algorithms, nor the appropriate scheduling between ring traffic and outgoing traffic from the nodes. Finally we assumed infinite queues in the network. For the interested reader a discussion of the network model can be found elsewhere [16].

Each GLOW agent is equipped with a 4096-entry directory storage and an optional 256K data storage. The directory storage is rather small: A directory entry (tree tag) for the K-ary *3*-cube needs approximately 12 bytes (five 16-bit pointers and state) making the total directory storage (*4096*×*12* bytes) 48 Kilobytes. In order to minimize conflicts the directory storage is organized as an 8-way set-associative cache.

In our simulation environment we have used the 'memory mapped' method (see Section 2) to generate special requests. Again this results in a conservative assessment because a dynamic scheme could potentially perform much better. Choices of what data are widely shared were made by one of the authors, after studying the code. This decision is again conservative: the programmer who wrote the code could easily do as good a job at identifying widely shared variables, and possibly might do much better.

6 BENCHMARKS

6.1 GAUSSIAN ELIMINATION

The GAUSS program solves a linear system of equations using the well known method of Gaussian elimination. Details of the shared memory program can be found in [6]. A coefficient matrix $N \times N$ is filled with random numbers and then the linear system is solved using a known vector. The work is divided among the processors by distributing the rows block-wise.

GAUSS consists of two phases: In the first phase, for each column, a pivot row is chosen between all processors. Subsequently this pivot row is read by all processors, multiplied by a factor and subtracted from the rest of the rows. In the second phase (the backward substitution phase) processors compute the vector of the unknown variables, starting from the last row. As each variable is computed every processor reads it and the appropriate factor is subtracted from every row.

The program has three structures that are widely shared: First in every iteration of the first phase the pivot row is read by all processors; in subsequent iterations elements of previous pivot rows are updated. Potentially every row of the coefficient matrix can be widely shared. Second, the unknown variable vector is widely shared in the second phase as every variable that is computed is read by most processors. Third, some variables used in a software tree for reduction operations are widely shared. To convert this program to use the GLOW extensions, the coefficient matrix, the unknown variable vector, and the reduction tree variables are memory mapped as widely shared data. In this way we have incorrectly defined the amount of widely shared data to nearly the total of the dataset of the program, because at any time only one row is the widely accessed pivot row (less than 1% of the dataset). GAUSS represents the case where memory mapping the widely shared data is not an adequate solution since we apply GLOW to non-widely data resulting in unnecessary overhead.

In Figure 6 we plot the speedups of SCI, GLOW, and GLOW WITH DATA STORAGE for a range of systems from 16 to 128 nodes, in 2 and 3 dimensions. SCI is represented by black bars, GLOW and GLOW WITH DATA STORAGE with grey bars. The extra white segments on top of the bars represent the additional speedup when switching from 2 dimensions to 3 dimensions. SCI does not exhibit any speedup beyond 32 nodes while GLOW continues up to 64 nodes (both in 2 and 3 dimensions) and up to 128 nodes (in 3 dimensions). Higher dimensionality benefits GLOW more than SCI since the GLOW trees depend on the topology of the network. GLOW WITH DATA STORAGE is not significantly faster than GLOW (at most 3% improvement).

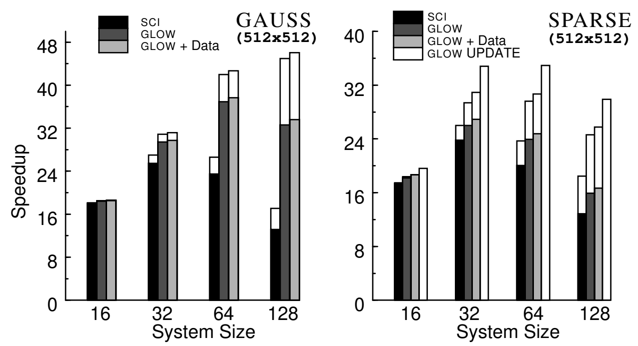

FIGURE 6. GAUSS and SPARSE in 2-D and 3-D

	2D				3D			
	16	32	64	128	16	32	64	128
SCI Cycles (M)	107	76	82	147	106	71	72	113
GLOW Cycles (M)	105	65	52	59	104	62	46	42
GLOW Speedup	1.02	1.17	1.58	2.49	1.02	1.14	1.56	2.69
Reads (K)	185	359	712	1376	185	359	725	1392
GLOW R. (%)	100	100	98	97	100	100	98	97
Lat. R (Cycles)	459	905	1744	4049	450	768	1423	3057
Lat. Reduction (%)	38	59	74	82	42	60	76	85
Writes (K)	63	84	154	267	63	84	154	266
GLOW W (%)	73	80	78	76	73	80	77	75
Lat. W (Cycles)	308	483	585	886	298	425	485	648
Lat. Reduction W (%)	-7	-13	-31	-30	-6	-10	-26	-19

Table 1: GAUSS 2-D and 3-D

In Table 1 we present statistics for GAUSS. The first two rows show the number of cycles for the execution of the program in various systems. The third row gives the speedup of GLOW relative to SCI. The next four rows are statistics for read misses: the number of read misses, the percentage that is converted to special requests

(GLOW reads), the average latency of a read miss in SCI and the reduction in the average latency when using GLOW. The last four rows are the corresponding statistics for write misses. For GAUSS, the average latency of reads in SCI increases dramatically with system size. GLOW average latency for reads grows more slowly but it actually increases faster for the average write latency. This is the result of applying GLOW to the whole coefficient matrix: Every write to this matrix is a GLOW write, though few actually affect pivot row elements, which are the widely shared elements.

6.2 SPARSE MATRIX SOLVER

This 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 A, the $N \times N$ sparse matrix and X, the vector that is widely shared. This vector has N elements (the number of columns in the sparse matrix). We memory-mapped this vector along with three variables used in software reduction trees as widely shared data. SPARSE is a good example of a program where memory mapping the widely shared data is effective.

SPARSE with a 512×512 matrix, as it is depicted in Figure 6, does not exhibit speedup beyond 32 nodes for either SCI or GLOW, though the latter is always faster than SCI for any system size. As reported elsewhere, the benefit of GLOW increases for larger data sets [16]. GLOW also benefits more than SCI when upgrading to a 3-dimensional network. Data storage improves GLOW speedups by at most 5%.

In Table 2, which follows the same format as Table 1, we present the statistics for SPARSE. Notice that the X vector along with the three variables that are accessed as widely shared data are responsible for 43% to 50% of all the reads (Table 2) of the program but their average latency (in SCI) does not increase dramatically as the system size increases. Consequently, the latency reduction in reads is smaller than the corresponding reduction in GAUSS and it does not translate in significant performance gains. In contrast to GAUSS, the percentage of GLOW writes remains at, or below, 8% across the range of systems and the average latency of writes decreases, which confirms that GLOW is indeed used only in widely shared data.

	2D				3D			
	16	32	64	128	16	32	64	128
SCI Cycles (M)	54	55	63	83	54	51	56	65
GLOW Cycles (M)	53	54	60	78	53	50	53	61
GLOW Speedup	1.02	1.02	1.05	1.06	1.02	1.02	1.06	1.07
Reads (K)	207	305	667	1380	206	305	667	1380
GLOW R. (%)	43	58	54	52	43	58	54	52
Lat. R (Cycles)	273	463	630	1098	265	384	500	729
Lat. Reduction (%)	8	9	19	20	11	18	26	29
Writes (K)	67	73	81	108	67	73	81	107
GLOW W (%)	8	7	6	5	8	7	6	5
Lat. W (Cycles)	378	737	1348	2754	368	621	1056	1898
Lat. Reduction W (%)	19	27	44	32	20	39	55	48

Table 2: SPARSE 2D and 3D

6.3 ALL-PAIRS SHORTEST PATH AND TRANSITIVE CLOSURE

These two parallel algorithms solve classical graph problems. The first problem is finding the shortest paths between all pairs of vertices in a graph. The second is to find the transitive closure of a graph, (*i.e.*, a new graph where two vertices are connected if there is a path in the original graph that connects these vertices). For both programs we used dynamic-programming formulations, that are special cases of the Floyd-Warshall [7] algorithm.

In the All-Pairs Shortest Path program (APSP), an N vertex graph is represented by an $N \times N$ adjacency matrix. The (i,j) element of this matrix represents the weight or distance between the i and j vertices. The Floyd-Warshall algorithm computes a series of N new matrices, each based on the previous matrix. We have parallelized the algorithm, using row decomposition and optimizing it by grouping reads, computations, and writes, to reduce the number of necessary barriers between iterations. The resulting program is reasonably optimized for SCI. The input graph used for the simulations is a 256 vertex dense graph (most of the vertices are connected). The adjacency matrix is memory-mapped as widely shared data.

In Figure 7 we show the speedups (with respect to one node) for GLOW and SCI. While SCI stops scaling beyond 32 nodes GLOW continues up to 128 nodes and in 3 dimensions up to 256 nodes. As in the other benchmarks GLOW benefits more in 3 dimensions than SCI. Again, GLOW WITH DATA STORAGE does not provide any significant advantage over GLOW (less than 2%). In Table 3 statistics for APSP are presented. Note the very high latencies SCI experiences for widely shared data. As in the other programs the number of reads increases with the number of nodes. The reduction in the average read latency also increases. The number of writes to the main data structure of the program however remains the same (about twelve thousand). This is evident from the declining percentage of GLOW writes. The rest of the writes are initialization writes that increase with the number of processors. The latency reductions for writes are also substantial mainly because SCI does not perform well with very large sharing lists.

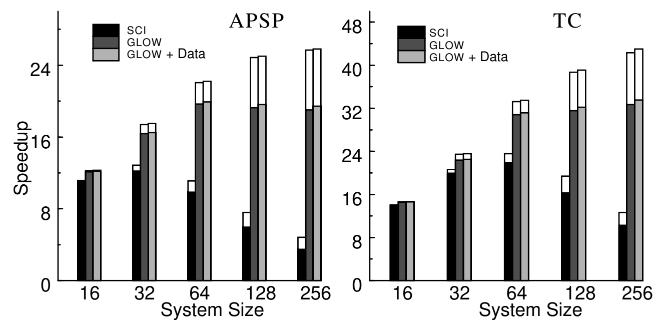

FIGURE 7. APSP and TC speedups in 2-D and 3-D

	2D					3D				
	16	32	64	128	256	16	32	64	128	256
SCI Cycles (M)	67	61	76	125	214	67	58	67	98	155
GLOW Cycles (M)	62	46	38	39	39	61	43	34	30	29
GLOW Speedup	1.08	1.33	2.00	3.20	5.45	1.10	1.35	1.97	3.27	5.34
Reads (K)	65	131	262	524	1049	65	131	262	524	1049
GLOW R. (%)	100	100	100	100	100	100	100	100	100	100
Lat. R (Cycles)	403	686	1150	2400	4803	389	572	931	1735	3414
Lat. Reduction (%)	28	47	61	70	81	32	47	64	78	84
Writes (K)	13	14	16	20	29	13	14	16	20	29
GLOW W (%)	92	85	75	60	42	92	85	75	60	42
Lat. W (Cycles)	801	1639	3533	7554	13215	783	1445	2913	5579	9188
Lat. Reduct. W(%)	39	55	74	82	88	42	62	78	86	89

Table 3: APSP 2D and 3D

Transitive Closure (TC), is another application of the Floyd-Warshall algorithm. In this program an $N \times N$ matrix represents the connectivity of the graph with ones and zeroes. Again the algorithm iterates N times and in each step it updates the matrix based on what was computed in the previous step. The input is a 256 vertex graph with a 50% chance of two vertices being connected. Similarly to APSP the program has also been reasonably optimized for SCI. The whole matrix is memory-mapped as widely shared data.

TC achieves better speedups than APSP and for SCI the speedups increase up to 64 nodes (both in 2 and 3 dimensions). When we use GLOW, the speedup continues to increase up to 256 nodes (Figure 7). A 3 dimensional network helps GLOW significantly. Data storage, again, has very little to offer (less than 2%). Table 4 presents the statistics for TC. As in APSP, the number of reads increases with system size, while the number of writes to the main data structure remains constant (around 8 thousand excluding the increasing number of initialization writes). As with APSP the latency of both reads and writes doubles each time the system size doubles and GLOW does a good job reducing the latencies.

	2D					3D				
	16	32	64	128	256	16	32	64	128	256
SCI Cycles (M)	42	30	27	36	57	42	29	25	30	46
GLOW Cycles (M)	40	26	19	19	18	40	25	18	15	14
GLOW Speedup	1.05	1.15	1.42	1.89	3.16	1.05	1.16	1.39	2.00	3.28
Reads (K)	72	139	272	535	1054	72	139	272	535	1054
GLOW R. (%)	100	100	100	100	100	100	100	100	100	100
Lat. R (Cycles)	352	590	1017	2145	3868	342	509	923	1867	3802
Lat. Reduction (%)	23	40	60	69	79	27	50	66	80	88
Writes (K)	9	10	12	16	25	9	10	12	16	25
GLOW W (%)	89	80	67	50	33	89	80	67	50	33
Lat. W (Cycles)	378	605	969	1937	4346	359	509	756	1163	2686
Lat. Reduction W (%)	8	26	37	54	75	8	21	24	30	61

Table 4: TC 2D and 3D

6.4 RELAXED CONSISTENCY

The benchmarks we have presented all assume a sequentially consistent view of memory. While our results indicate that GLOW achieves the bulk of its speedup from scalable reads, relaxing the memory model provides the opportunity to overlap write operations, thereby achieving greater concurrency. We relaxed the consistency model for SCI by allowing the processor to continue on writes after it becomes head of the sharing list, thus overlapping the invalidation of the sharing list—by far the most expensive part of the write—with computation or other writes. We studied APSP because it could easily tolerate a relaxed memory model, permitting concurrent write operations to shared data between barriers. In Figure 8 we see that the performance of SCI is significantly improved, but GLOW, even without relaxing consistency, still outperforms SCI. Of course, relaxing consistency should also benefit GLOW, though perhaps to a lesser extent, since the cost of sharing list invalidations on writes is smaller to begin with.

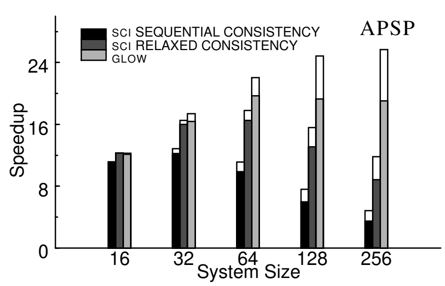

FIGURE 8. Relaxed consistency in APSP

6.5 UPDATE PROTOCOL

We have also extended the GLOW protocol to do updates rather than invalidates. This extension does not assume sequentially consistent memory, but again we have been able to ascertain that there are no data races in SPARSE. In Figure 6 the solid white bars represent the GLOW update in three dimensions which is 6-21% faster than GLOW invalidate. This is an indication that, at least for some applications, an update protocol can be used successfully with widely shared data. In preliminary experimentation using update with the other benchmarks we found that only SPARSE showed improvement over invalidation.

7 RELATED WORK

7.1 STEM

The STEM extensions to SCI [12] provide a logarithmic-time algorithm to build, maintain and invalidate a binary sharing tree (in contrast to GLOW's K-ary trees) without regard to the topology of the interconnection network. The complexity of the algorithm, however, is high, requiring complex transactions that generate increased traffic. The STEM algorithm was proposed under the assumption that the latency of any message is unit latency (*i.e.* all messages have the same latency, regardless of the distance they travel). This makes STEM actually an $O(\lg^2 N)$ algorithm since the longest distance a message must travel is $O(\lg N)$. Request combining is used in the network to reduce the bandwidth requirements of the STEM algorithm and possibly capture some network locality. As employed in STEM, request combining does not require information to be stored in the network.

7.2 OTHER WORK

The Scalable Tree Protocol [21] defines a sharing tree protocol that speeds up the writing of shared data by invalidating the tree in logarithmic time. The notable feature of the protocol is that additions to and deletions (rollouts/replacements) from the tree leave the tree balanced. Like STEM, the tree does not map to the underlying topology (thus the protocol does not benefit from physical locality). This is because its structure depends on the timing of the requests, and deletions from the tree (replacements) change its structure without respect to the underlying topology. Furthermore no support has been proposed to speed up reads (*e.g.* combining). Simulation [21] shows a 15% speedup over SCI for the GAUSS algorithm in a 16 node system but also a 35% *increase* in network traffic. This may be a serious drawback for larger systems.

The Tree Directory (TD) protocol [20] is based on a K-ary tree structure that is maintained in the sharing caches. It behaves as a limited directory in a tree structure. This scheme does not take into account physical locality and it does not provide for scalable reads. On the other hand, the Hierarchical Full Map Directory also proposed [20] is similar to our approach and exhibits network locality. It is based on full-map directories embedded in the network topology. Both TD and HFMD are strictly based on the inclusion property. In our work we do not impose the inclusion property, thus having increased flexibility to avoid deadlocks and to use schemes like the LINEARIZING rollout for replacements. This is important for performance since the inclusion property imposes destructive invalidation of the children in the case of a parent replacement and may result in erratic behaviour in pathological cases. For the HFMD the authors report a 6% performance improvement over a full-map scheme and for the TD a decrease of 25% in performance compared to a chained directory scheme such as SCI.

Eager Combining (EC) [3] uses specified nodes as servers for widely shared pages or *hot* pages. These pages are updated in the server nodes (using eager sharing). Clients request the data from the servers rather than the actual home node. It is similar to our work in that the authors only use it for widely shared data. However EC does note take into account network topology, uses updates for the servers, and caches the actual data, which in our case is optional. The authors report speedup over DASH in the range of 2 to 3 for up to 128 processors.

8 SUMMARY

We have defined GLOW, a new cache coherence scheme that increases the range of scalability by eliminating hot spots for widely shared reads. Widely shared variables can be efficiently accessed, even for very large systems, because the number of accesses to the home memory and the directory does not grow as the number of processors grows. We demonstrated this capability with the specification of an extension to the ANSI/IEEE 1596 Scalable Coherent Interface, and implemented the extension on SCI as simulated under the Wisconsin Wind Tunnel.

Though the method scales, it is less efficient than the base SCI protocol for accesses to memory that is not widely shared. We therefore depend on the programmer and/or compiler to identify accesses to widely shared variables and invoke the GLOW protocol only for these variables. We used a static allocation strategy, determining which variables should be accessed via GLOW by the region of memory to which they are assigned.

We studied the extension using four application programs, GAUSS, SPARSE, APSP, and TC. For all the programs we showed that for this implementation of GLOW the optional data storage is not cost-effective and higher dimensionality networks benefit GLOW more because of the increased depth and parallelism of the sharing trees. We demonstrated that for GAUSS consistent improvements are obtained, and that the improvements become more dramatic as the number of processors grows. Even greater speedups would be possible if each time the choice between base SCI and GLOW were made according to the current degree of sharing for the data in question. Accessing only the pivot row (and nothing else) through GLOW would then be possible, achieving still greater speedup. For SPARSE, the static selection of the access method (memory mapping of widely shared data) is effective, and GLOW provides improved performance. For APSP and TC, the SCI read and write latencies increase dramatically with system size and GLOW effectively keeps the latencies low leading to impressive speedups. Despite the impressive improvements in SCI from relaxing the memory consistency model, GLOW still outperforms without having to relax the memory model.

9 ACKNOWLEDGEMENTS

We wish to thank David V. James, Ross E. Johnson, Stein Gjessing and David B. Gustavson for their suggestions, contributions and constructive comments on this work. We are indebted to Dionisios Pnevmatikatos, Doug Burger, Alain Kägi, and Babak Falsafi for providing and many useful and constructive comments on drafts of this paper.

10 REFERENCES

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

[2] J. L. Baer and W. H. Wang, "Architectural Choices for Multi-Level Cache Hierarchies." *Proceedings 16th International Conference on Parallel Processing*, pp. 258-261, 1987.

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

[4] John Carter, John Bennett and Willy Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence." *Proceedings of the Conference on the Principles and Practices of Parallel Programming*, 1990.

[5] David Chaiken, John Kubiatowicz, Anant Agarwal. "LimitLESS Directories: A Scalable Cache Coherence Scheme," *Proc. of the 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 224-234, April 1991.

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

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

[8] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill, "Programming for Different Memory Consistency Models." *Journal of Parallel and Distributed Computing*, 15(4), 1992.

[9] J.R. Goodman, Mary K. Vernon, Philip J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache Coherent Multiprocessors." *Proc. of the 3rd Int. Conf. on Architectural support for Programming Languages and Operating Systems (ASPLOS-III)*, April 1989.

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

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

[12] Ross E. Johnson, "Extending the Scalable Coherent Interface for Large-Scale Shard-Memory Multiprocessors." PhD Thesis, University of Wisconsin-Madison, 1993.

[13] Ross E. Johnson, James R.Goodman, "Interconnect Topologies with Point-to-Point Rings," *Proc. of the International Conference on Parallel Processing*, August 1992.

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

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

[16] S. Kaxiras and J. R. Goodman, "The GLOW Cache Coherence Extensions for Widely Shared Data." University of Wisconsin-Madison, C.S. Dept., Technical Report 1305, March 1996. (ftp.cs.wisc.edu)

[17] Leslie Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs." *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

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

[19] Kai Li, Paul Hudak, "Memory Coherence in Shared Virtual Memory Systems." *ACM Transactions on Computer Systems*, Vol. 7, No. 4, pp. 321-359, November 1989.

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

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

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

[25] Steven L. Scott, James R. Goodman, Mary K. Vernon, "Performance of the SCI Ring." *Proc. of the 19th Annual International Symposium on Computer Architecture*, pp. 403-414, May 1992.

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