Kiloprocessor Extensions to SCI†

Stefanos Kaxiras‡

Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison, Wisconsin 53706 USA
kaxiras@cs.wisc.edu

Abstract

To expand the Scalable Coherent Interface's (SCI) capabilities so it can be used to efficiently handle sharing in systems of hundreds or even thousands of processors, the SCI working group is developing the Kiloprocessor Extensions to SCI. In this paper we describe the proposed GLOW and STEM kiloprocessor extensions to SCI. These two sets of extensions provide SCI with scalable reads and scalable writes to widely-shared data. This kind of datum represents one of the main obstacles to scalability for many cache coherence protocols. The GLOW extensions are intended for systems with complex networks of interconnected SCI rings, (e.g., large networks of workstations). GLOW extensions are based on building k*-ary sharing trees that map well to the underlying topology. In contrast, STEM is intended for systems where GLOW is not applicable (e.g., topologies based on centralized switches). STEM defines algorithms to build and maintain binary sharing trees. We show that latencies of GLOW reads and writes grow only logarithmically with the number of nodes sharing, in contrast to SCI where latencies grow linearly, therefore validating GLOW as a good solution to efficient wide sharing of data. Previous work showed the same for STEM.*

1 Introduction

For most cache coherence protocols, widely-shared data represent one of the major obstacles to scalability to large numbers of processors. This type of data does not appear in abundance in application programs [12]. However, accessing these data is expensive and, worse yet, it becomes progressively more expensive as the size of the parallel computer increases. While a program running on a small multiprocessor spends only a fraction of its time accessing such data, the same program when run on a system with hundreds of processors may devote a significant part of its time to widely-shared data accesses.

The ANSI/IEEE Standard 1596 Scalable Coherent Interface (SCI) [1] defines a cache coherence protocol based on distributed sharing lists. To improve the performance for widely-shared data in SCI, the GLOW [4] and STEM [2] kiloprocessor extensions are developed. Both are designed to handle accesses to widely-shared data and provide good scalability to large numbers of processors.

To enable scalability of programs, both scalable reads and scalable writes for widely-shared data are essential. Request combining, originally proposed for the NYU Ultracomputer [5], is the main vehicle for achieving scalable reads. The effect of request combining is achieved efficiently in GLOW because of the nature of the protocol itself, which caches certain information in the network. The request combining that STEM uses does not require information to be stored in the network. For scalable writes, the traditional approach is to devise sharing tree protocols. Examples include the Scalable Tree Protocol (STP) [9], the Tree Directory (TD), and the Hierarchical Full-Map Directory (HFMD) [8]. GLOW is based on *k*-ary sharing trees, while STEM defines binary sharing trees.

GLOW and STEM are optimized for different types of target system. Each has its own advantages that may be better realized in a specific class of SCI environments. The GLOW extensions are intended to be used in SCI multiprocessor systems that are comprised of many SCI rings (the basic topology component defined in SCI) connected through bridges (e.g., building-wide networks of workstations). In such systems the GLOW extensions are intended to be used only for accesses to widely-shared data, while the rest of the sharing in the system is left to the standard SCI cache coherence protocol. This specialization is required because GLOW extensions, despite their high performance in accessing widely-shared data, incur higher overhead for very low degrees of sharing compared to the basic SCI protocol. GLOW extensions are plug-in compatible to SCI systems. The extensions are implemented in the bridges that connect the SCI rings. In order to upgrade an SCI network of workstations to GLOW, only a set of bridges need change. The SCI cache coherence protocol, however implemented in the workstations, need not be modified in any way.

On the other hand, STEM is intended to be used in tightly integrated systems (e.g., massively-parallel systems). In such systems, large high-performance crossbar switches are likely to be used as the means to interconnect multiple SCI rings. In such an environment, GLOW extensions are not well suited, since they are most practical for environments with many small bridges. STEM is a complete upgrade of the SCI protocol, though STEM and SCI nodes interoperate at the lower level of performance. In contrast to GLOW, STEM does not distinguish between widely-shared data and the rest of the data, since it can be used for all accesses. In the case of non-widely-shared data STEM has the same performance as the SCI protocol with sim-

† This work was supported in part by NSF Grants CCR-9207971 and CCR-9509589, funding from the Apple Computer Advanced Technology Group, and donations from Thinking Machines Corporation. Our T.M.C. CM-5 was purchased through NSF Grant CDA-9024618, with matching funding from the University of Wisconsin Graduate School.

‡ Editor, IEEE P1596.2 proposed Standard. The views expressed in this paper are the author's and do not imply endorsement from IEEE.

ilar transactions.

In the rest of this paper we will give a short description of the SCI cache coherence protocol in Section 2, describe the GLOW extensions in Section 3, and the STEM extensions in Section 4. We will then present a brief performance evaluation of GLOW in Section 5 and conclude in Section 6.

2 SCI

The ANSI/IEEE Standard 1596 Scalable Coherent Interface represents a thorough and 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 section of SCI defines a 1Gbyte/s ring interconnect, and the transactions that can be generated over it. A performance analysis by Scott, Vernon, and Goodman [11] showed that an SCI ring can accommodate small numbers of high-performance nodes, in the range of four to eight. To build larger systems, topologies constructed out of smaller rings must be used (*e.g.*, *k*-ary *n*-cubes, multistage topologies) [13]. Topologies can be built either by using multiple bridges, each connecting a small number of rings, or centralized switches, each connecting many rings.

SCI also defines a distributed directory-based cache-coherence protocol. In contrast to most other directory-based protocols (*e.g.*, DASH [10]), which keep all the directory information in memory, SCI distributes the directory information among 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 requested the data. In a stable list, this node is called *head.* As the head of the sharing list, a node has both read and write permission to the cache line; all other nodes in the sharing list only have read permission (except in pairwise sharing mode). However, when many nodes are trying to join the list concurrently, a singly linked *prepend* queue is formed in front of the head. The head may be asked to link itself to its prepending successor at any time, but it may briefly postpone its response until it is ready to give up its write permission. In figure 1, the SCI sharing list and a prepend queue (in front of a sharing list) are depicted. Caches, represented by small rectangles, are connected in linked lists with their *forward* and *backward* pointers.

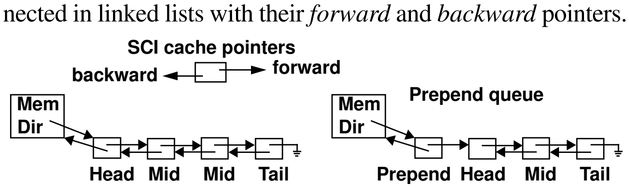

Figure 1. SCI sharing list

In the following sections we will describe the three basic operations of SCI: construction of sharing lists, node removal from a sharing list (rollout), and invalidation of sharing lists. In general the latency of the sharing list construction may be affected by the number of nodes sharing (due to contention, data propagation delays, etc.). However, addition of a single node to a stable SCI list as well as node removal from a sharing list (rollout) does not depend on the number of nodes in the list. The latency of the invalidation of a sharing list, however, is *O(N)* where *N* is the number of nodes in the list.

2.1 SCI sharing list construction

The first node that requests data from memory triggers the creation of the sharing list. This node becomes the head and only node of the list. The memory directory is set to point to this node. The node points back to the memory directory by means of the memory address. Subsequently, nodes can join the sharing list by asking the memory directory for data, thus joining the prepend queue and eventually become the head of the sharing list (Transaction A, subactions 1 and 2 in figure 2). They each eventually attach to their previous head (Transaction B, subactions 3 and 4 in figure 2). The node might get the data from the memory itself or from the previous head of the list, depending on whether memory has a valid copy of the cache line (denoted by the memory state FRESH) or the memory copy is potentially stale (denoted by the memory state GONE). In the second case, data distribution in the sharing list proceeds from the TAIL node toward the HEAD node. Nodes prepend to the sharing list and wait for previous nodes to pass them the data (Transaction C, subactions 5 and 6 in figure 2).

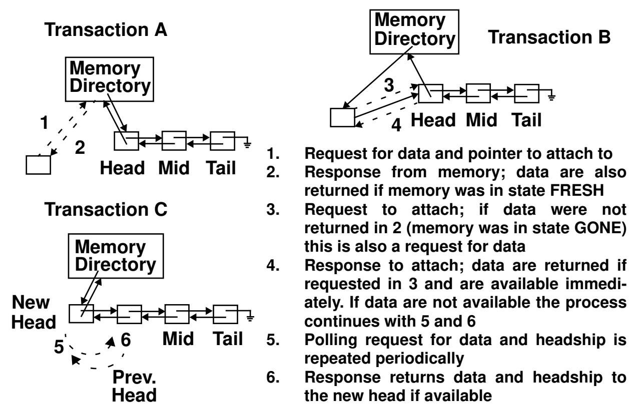

Figure 2. SCI sharing list construction

2.2 Rollout

A node can also leave the sharing list, by communicating with its neighbors. This operation is called *Rollout* in the SCI terminology, and it takes place in two situations: First, when there is a conflict in a cache and the cache line has to be replaced, the node rolls out of the sharing list. Second, a node other than the head of the sharing list has to roll out and become the new head in order to write the cache line.

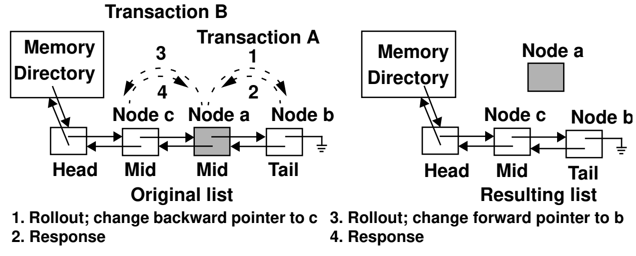

Figure 3. SCI cache rollout

In figure 3, node **a** leaves the sharing list. With subaction 1 of Transaction A it notifies its downstream neighbor node **b** to point to node **c**; subaction 2 is the positive acknowledgment. Similarly, it notifies node **c** to point to node **b**. The resulting sharing list is shown in the right of figure 3. Notice that the order of Transactions A and B is critical, to allow concurrent rollouts in the sharing list. In case of conflicts the downstream node (the one closer to the tail) has priority.

2.3 SCI sharing list invalidation

As the head of a sharing list, a node has to 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. In figure 4 the head of the list invalidates the other nodes. For Transaction A, subaction 1 invalidates node **a** and subaction 2 returns the pointer to **b**; similarly for the rest of the transactions.

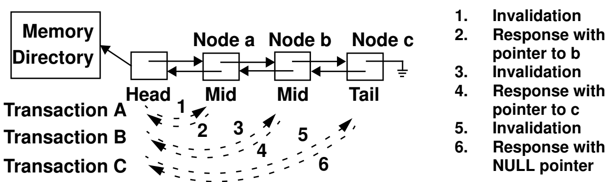

Figure 4. SCI sharing list invalidation

3 GLOW kiloprocessor extensions to SCI

As is evident from the above description, widely-shared data represent a serious obstacle to the scalability of SCI to hundreds or thousands of nodes. When many nodes read the same cache line, contention at the memory for additions to the list is very high. Data distribution through the SCI prepend list also becomes progressively more expensive as the number of readers increases. Furthermore, writes to widely-shared data result in invalidation of long sharing lists. As we have seen, this operation is serial in SCI.

To overcome these obstacles, the GLOW extensions are based on building *k*-ary sharing trees that map well onto the network topology of a system. Most other tree protocols like TD, STP, and STEM capture *temporal* locality in the sharing tree: requesting nodes, in close proximity in time, end up neighbors in the sharing tree. However, none of these protocols capture *geographical* locality well, since the structure of the sharing trees depends on the timing of the requests. By geographical locality we mean that nodes in physical proximity become neighbors in the sharing tree, regardless of the timing of their requests. Such locality in the tree leads to protocols where messages have low latency. GLOW captures the geographical locality of all topologies so far explored by mapping the sharing trees on top of the trees formed from the natural traffic patterns.

All GLOW protocol processing takes place in strategically selected *bridges* (that connect two or more SCI rings) in the network topology. In general, all read requests for a specific data block from the nodes of an SCI ring will be routed to a remote memory (located on another SCI ring) through the same bridge. If this bridge contains directory information or a copy of the data block, these requests can be *intercepted* and satisfied locally on the ring. Such a bridge is called a *GLOW agent*. The GLOW agents do not intercept all SCI read requests, just those specially tagged as requests for widely-shared data.

The GLOW sharing tree is comprised of the SCI caches that

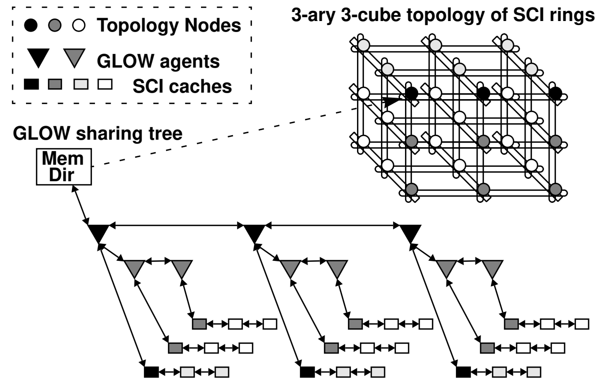

Figure 5. GLOW sharing tree on a *3*-ary *3*-cube

have a copy of the data block and the GLOW agents that hold the relevant directory information. The sharing trees are built out of small linear SCI sharing lists. In figure 5, a full, perfectly ordered, sharing tree on a 3-ary 3-cube topology made of SCI rings is shown. The agents are represented by triangles and the SCI caches by small rectangles. The shading of the agents and the SCI caches shows their position in the topology. An agent, and the similarly shaded cache directly connected to it, coincide in the same node.

Generally, in a GLOW tree each list is confined to one physical ring. The job of the agent is to impersonate the remote memory, however far away it is, on the local ring. The small SCI lists are created under the agent when read requests from nodes on a ring are intercepted and satisfied either directly by the agent (as if it were the remote memory) or by another close-by node (usually on the same ring). Without GLOW agents, read requests would go all the way to the remote memory and would join a global SCI list. The agent itself, along with other nodes in the higher level ring (the next ring toward the remote memory), 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 lists continues up to the ring containing the actual remote memory.

A GLOW agent has a dual personality: toward its children it behaves as if it were the SCI memory directory; toward its parent agent (or toward the memory directory itself) it behaves as if it were an ordinary SCI cache. For example, in figure 5, as far as the memory directory is concerned, it points to a list of SCI caches, whereas in reality it points to the first level of agents holding the rest of the sharing tree. Similarly, as far as the SCI caches (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 memory.

Although the GLOW agents can behave like memory directories there are two differences in the way GLOW agents and memory directories hold SCI lists. The first difference is in the number of SCI lists that the agents can hold. In contrast to the SCI memory directory which can only hold one SCI list (per cache line), the agents have a number of pointers, called child pointers, so they can hold an SCI list (called a child list) in any of the rings they service (up to one less than the total number of rings they are connected to). The actual number of child lists is not specified, but is considered an implementation parameter, reflecting both topology and cost considerations.

The second difference is in the way the GLOW agents hold the child lists. In contrast to the memory directories, which only point to the head of a list, the GLOW agents hold lists from both ends. This is mainly to facilitate the rollout algorithm (Section 3.2) and it is achieved, in an SCI compatible way, by the agent presenting itself as a *virtual tail* to the first node that joins each child list. In figure 6 the agent is represented by a triangle and the SCI caches by rectangles. For every child list the GLOW agent has a *head* and a *tail* pointer (see figure 6). The agents also have a *forward* and a *backward* pointer, permitting them to join SCI lists and act like ordinary SCI caches.

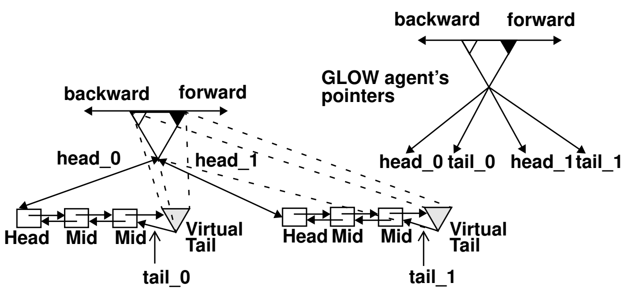

Figure 6. GLOW agent holding 2 child lists

As is evident from this general description, there is an overhead in building the sharing tree, namely the overhead of invoking all the levels of GLOW agents from the leaves to the memory directory. However, the first node in an SCI ring will invoke the GLOW agent, which will subsequently service, at a small additional cost, the rest of the nodes on the ring. The degree of sharing (how many nodes are actually sharing a data block) is important in determining whether the overhead of building the tree is sufficiently leveraged by the nodes that have their requests satisfied locally on the ring by the agent.

Request combining occurs because the GLOW agents generally do not let intercepted requests pass them by. Instead, when the agent is invoked, it generates its own request and sends it toward the remote memory directory. The memory directory sees only the requests of the first level of agents. Such behavior has two effects: First, it eliminates memory hot spots [3]. Second, because the requests are satisfied locally, messages travel only short distances, thus decreasing the load on the network.

Finally, note that it is neither necessary to have a copy of the data in the GLOW agents nor is it necessary to maintain multilevel inclusion in the caching of the directory information in the GLOW agents. Caching the data in the GLOW agents is completely optional and it is usually avoided to conserve storage. The implication of not enforcing multilevel inclusion is twofold: First, it is easier to avoid protocol deadlocks in arbitrary topologies. Second, it is possible to avoid invalidating all descendents whenever an entry high up in the hierarchy is replaced. Since we do not enforce multilevel inclusion, the involvement of the GLOW agents is not necessary for correctness.

3.1 GLOW sharing tree construction

The GLOW sharing trees are constructed with the involvement of the GLOW agents that impersonate the memory directory in various places in the topology. A node uses a special request to access a line of widely-shared data. The request is intercepted at the first agent (where the request would change rings), *at the agent's discretion*. The intercept causes a lookup in the agent's directory storage to find information (a tree tag) about the requested line which will result in a *hit* if there is an allocated tag or a *miss* otherwise:

- If the lookup results in a miss, the agent sends its own special request for the data block toward the memory node. The agent instructs the requesting node to attach to the virtual tail (the agent itself) of a child list and wait for the data. As soon as the agent gets a copy of the line it will pass it to the child lists through the virtual tails.
- If the lookup results in a hit, the requesting node is instructed to attach to the appropriate child list. It will get the data from either the agent (if it caches data) or the previous head of the child list. If the requesting node attaches directly to the virtual tail (i.e., it is the only real node in this child list) the agent has to *fetch* the data from one of its other child lists. The agent cannot repeat its request to get the data, since this would result in re-joining the sharing tree.

As mentioned before, the agent may choose to ignore a request completely since multilevel inclusion is not enforced. Requests are ignored in order to avoid deadlocks due to storage conflicts between tree tags in transient states. When an agent ignores a request, it is passed to the next hierarchical level where it may be serviced by the higher level agent. The request may be ignored by multiple agents all the way to the remote memory directory where eventually it will be serviced. The requesting node will end up in a child list higher up in the tree.

3.2 GLOW rollout

An ordinary SCI cache leaves the tree (rolls out) because of a replacement or as a prerequisite for writing the data. The SCI caches follow the standard SCI protocol to roll out from the sharing tree. Agents roll out because of conflicts in their directory (or data) storage or because they are left childless. Childless and dataless agents are not allowed in the tree since they cannot repeat their request to get data to service new children as this would put them in the tree more than once. When an agent finds itself childless it is only connected to the tree with its *forward* and *backward* pointers, just like an SCI cache. In this case the rollout is the standard SCI rollout.

The agent rollout permits the structure of the tree to degrade gracefully. The rollout is based on chaining the child lists and subsequently substituting the chained child lists in place of the agent in the tree. This is feasible because of the policy of not enforcing multilevel inclusion in GLOW. In the opposite case, the subtree beneath the agent would have to be destroyed.

In figure 7 A we depict a segment of a sharing tree where the agent in the middle of the high level list is about to rollout. The agent becomes *virtual head* in all its child lists with *attach* requests (figure 7 A and B). Notice that all the virtual nodes are in reality only one entity (the agent itself) and there is no change in the pointers of any node. Since the virtual tail of the leftmost child list and the virtual head of the rightmost child list are the same node (the agent) they can roll out as one and leave the two child lists connected into one (figure 7 B). In this way any number of child lists can be concatenated into one list in one step. Concurrently, the agent rolls out as virtual head and virtual tail of the concatenated child lists (figure 7 C). The sharing tree after the agent rollout is shown in figure 7 D.

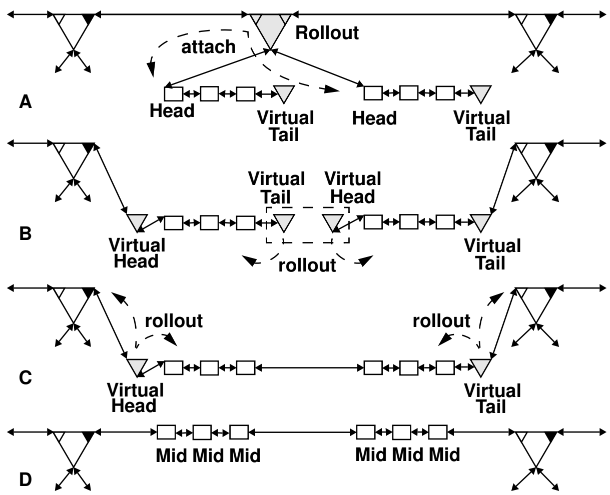

Figure 7. GLOW agent rollout

3.3 GLOW sharing tree invalidation

In order to write a cache line, a node 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 and it is the only node that has write permission for the cache line. A node has to roll out from the sharing tree before becoming root.

After the cache line is written, the node starts invalidating the highest level of the sharing tree using the SCI invalidation protocol. However, the GLOW agents react differently than the SCI caches to invalidation messages. On receipt of an invalidation message, the GLOW agent concurrently does the following:

- replies to the node that sent the invalidation message with a negative acknowledgement pretending that it is about to roll out
- forwards the invalidation to its downstream neighbor; if the downstream node happens to be an SCI cache and returns a new pointer the agent proceeds with invalidating the new node
- attaches to its child lists, and as a virtual head starts invalidating them using the SCI invalidation algorithm.

When the agent is done invalidating its child lists, it waits until it becomes tail in its list. This will happen because it will either invalidate all its downstream nodes, or they will rollout by themselves (if they are GLOW agents). When the agent finds itself childless and tail in its list, it will invalidate and roll out from its upstream neighbor, freeing that to invalidate itself.

4 STEM kiloprocessor extensions to SCI

The STEM extensions to SCI, originally developed by Johnson [2], 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. STEM employs combining in the interconnect to provide scalable reads.

As employed in STEM, combining operates on two requests that happen to be in a queue at the same time, generating a single new request along with a response to one of the original requests. Combining depends heavily on traffic patterns, and in fact will rarely occur except in the presence of substantial congestion [3]. Combining only returns list-pointer information (and not data, which are returned later). This is much simpler than other approaches [5], which leave residual state within the interconnect for modifying the responses when they return.

STEM defines one additional pointer for the SCI caches. The representation (triangle) of a STEM cache (devised by James [14]) is shown in figure 8. The *forward* and *backward* pointers are the same as the ones in SCI caches. The *down* pointer is used to build the binary sharing trees. Notice that in figure 8 the different pointers correspond to the different vertices of the triangle.

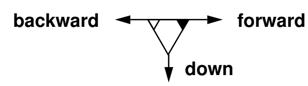

Figure 8. STEM cache pointers

4.1 STEM sharing tree creation

The STEM sharing tree creation starts with an ordinary SCI prepend list (figure 9 A). Nodes prepend to this list and eventually they will get the data. A merge process converts this linear list into a list of subtrees. In this structure, the subtrees are strictly ordered according to their height. The first subtree (connected directly to the memory directory) is of height one. Larger subtrees are placed further away from the memory directory. Nodes attaching in front of a stable list of subtrees will trigger further merges: adjacent equal-height subtrees merge to form a single higher subtree, until a stable subtree structure is formed. Up to three transactions are needed to merge subtree pairs.

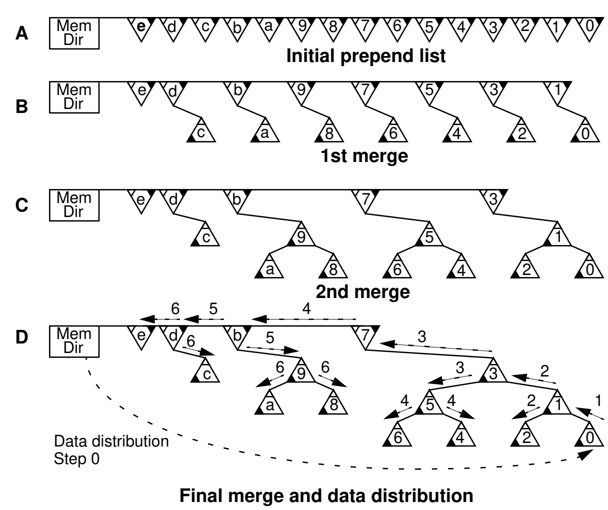

Figure 9. STEM sharing tree construction

The merge process is distributed and performed concurrently by the sharing caches. Data distribution is partially performed during the merging process. Multiple steps are involved (some overlapped in time), one for each level of the binary tree that is created. Under ideal conditions, the first step of the merge process generates a list of subtrees of height one (figure 9 B), the second generates a list of subtrees of height two (figure 9 C), etc. The process continues until the sharing list has reached a stable state, as illustrated in figure 9 D.

During the subtree-merge process, a distributed algorithm selects which pairs of subtrees are merged. To avoid conflicts, nodes two and one cannot be merged if one and zero are being merged. To generate balanced trees, nodes three and two should be merged if nodes one and zero are being merged. Techniques for selecting the nodes to be merged are discussed by James [14] and by Johnson [2]. Following the merging, data are distributed to the nodes. Data are propagated by cache-to-cache writes, as illustrated in figure 9 D. Some of these transactions can take place concurrently with the sharing-tree creation.

4.2 STEM rollout

Rollout in STEM occurs for the same reasons as for SCI caches. Rollouts are more complex for trees than for lists, when the rolling out node has two children. In this case, the rolling out node *walks* down the tree to a leaf node, removes the leaf from the tree, and finally replaces itself with the leaf node.

If a node has none or just one child, then the basic SCI rollout protocol is used. If a node has two children, it must find a replacement for itself in order to keep the tree structure intact before it rolls out. The leaf node that is used to replace the rollout node is found by first using the *down* pointer and then the *forward* pointer. This search procedure creates a one-to-one correspondence among nodes with two children and leaf nodes and there is no overlap between paths that connect pairs of nodes. After a leaf node has been located with the walkdown, the leaf is swapped with the rollout node. The walkdown and node replacement protocols can introduce up to eight transactions per deleted node. However, half of the nodes in any binary tree are leaf nodes, so the walkdown is only performed half of the time. Furthermore, the average number of walkdown steps required for nodes in a balanced binary tree is one and this average becomes even smaller as the tree becomes unbalanced.

4.3 STEM sharing tree invalidation

Sharing tree invalidation is initiated by the writer that is the *root* of the tree (connected directly to the memory directory). In the first phase, invalidations are distributed to other nodes in the sharing tree, as illustrated in figure 10 A. The forwarding of invalidations stops with the leaf nodes. In the second phase, leaf nodes invalidate themselves and notify their parent. This trims the leaf nodes from the sharing tree all the way up to the root node, as illustrated in figure 10 B and C.

The sharing tree can also be updated with new values rather than invalidated. Similarly to the invalidation, in the first phase, updates rather than invalidates are distributed all the way to the leaves (figure 10 A). In the second phase, acknowledgements are returned to the *root* node (writer) without affecting the sharing tree (figure 10 D).

5 Preliminary performance evaluation

In this section, we present simulation results for GLOW and SCI. We have micro-benchmarked reads and writes in various simulated systems. The STEM extensions have not yet been implemented in full detail. However, Johnson simulated the behavior of STEM and in his thesis [2] he shows that indeed its performance for both reads and writes is *O(logN)* where *N* is the number of nodes sharing. The simulations were performed with the assumption of unit-latency messages.

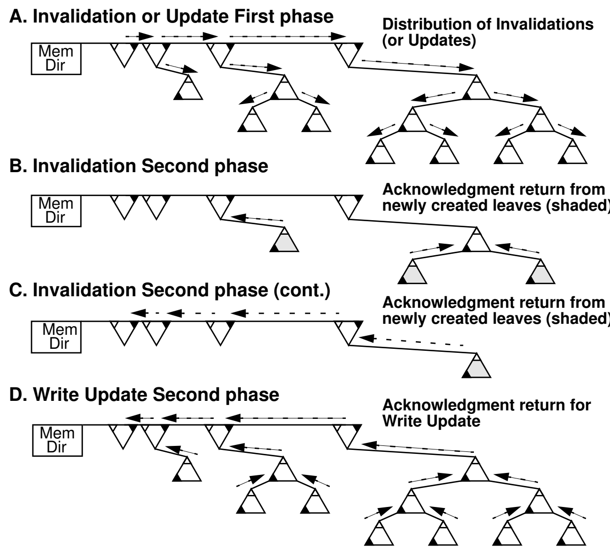

Figure 10. STEM sharing tree invalidation

5.1 GLOW in the Wisconsin Wind Tunnel (WWT)

We implemented GLOW on the WWT [7] and, in particular, as extensions to the WWT SCI simulator [6]. WWT is a parallel discrete event simulator that runs on a Thinking Machines CM-5. It executes application programs on the simulated machine with a very small slow-down compared to other simulation methods. The simulated machine was based on a *k*-ary *n*-cube topology constructed with SCI rings. We simulated message latency but we assumed contention only in the end-points (the rest of the network was contention-free). The simulation parameters were the same as those used in a previous study [6].

5.2 Micro-benchmarks

We measured the average read and write latency of nodes repeatedly accessing shared variables for both SCI and GLOW. We simulated systems ranging from 32 to 256 nodes in two, three and four dimensional *k*-ary *n*-cubes. We present the results for the small (32-node) and large (256-node) topologies in figure 11. All graphs are logarithmic in both axes; the vertical axis represents latency in cycles as reported by the WWT and the horizontal axis represents the degree of sharing.

For the small systems the overhead of building GLOW sharing trees is pronounced: GLOW reads are slower than SCI when less than about three nodes are sharing and GLOW writes are slower when less than about six nodes are sharing. Notice that SCI read latencies are not affected by the dimensionality of the network, since they are dominated by congestion in the directories, and data propagation delays (message travel time is negligible). On the other hand, GLOW latencies are affected much more since the dimensionality determines the depth of the GLOW sharing trees.

For the large systems (256 nodes) GLOW reads and writes are still slower for very small degrees of sharing but they show

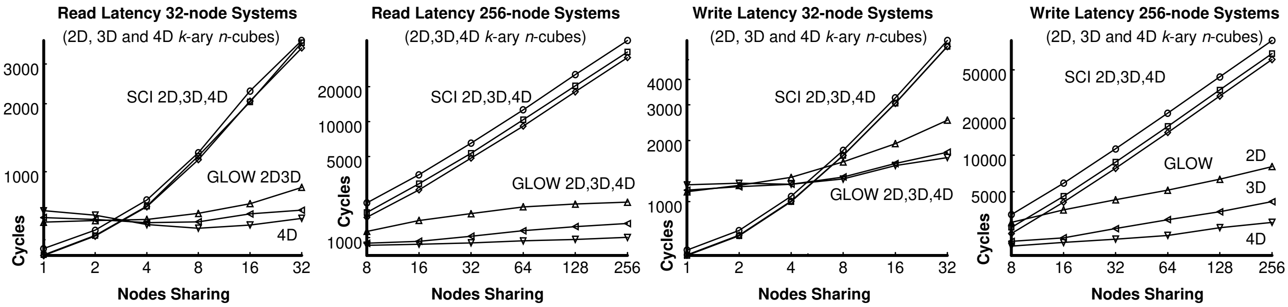

Figure 11. Micro-benchmark results

excellent scaling compared to SCI. The speed-ups in high degrees of sharing are more than ten-fold. Notice also that in large topologies the dimensionality of the network starts to play a role in SCI latencies as well.

6 Summary

We have described the GLOW and STEM extensions to SCI that increase 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. The two sets of extensions are optimized for different SCI environments. GLOW is intended for systems where many small bridges interconnect many SCI rings, while STEM is intended to be used in tightly coupled systems where network locality is of secondary importance. Through the use of micro-benchmarks, we demonstrated that GLOW latencies grow only logarithmically as the number of readers grows. In previous work, simulation of the STEM behavior showed that it also exhibits logarithmic latencies. SCI latencies, on the other hand, grow linearly with the number of nodes sharing.

7 Acknowledgements

We wish to thank James R. Goodman, David V. James, Stein Gjessing, Ross E. Johnson, David B. Gustavson, as well as the whole IEEE P1596.2 Working Group for their suggestions, contributions and constructive comments on this work. David V. James wrote the STEM part of the IEEE P1596.2 proposed Standard (draft 0.35) on which the description of STEM in this paper is based. We also wish to thank Doug Burger, Babak Falsafi, Alain Kägi, and Andreas Moshovos for reviewing drafts of this paper. Finally, we thank the anonymous referees who provided useful suggestions and comments.

8 References

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

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

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

[4] Stefanos Kaxiras and James R. Goodman, "Implementation and Performance of the GLOW Kiloprocessor Extensions to SCI on the Wisconsin Wind Tunnel." *Proc. of the 2nd International Workshop on SCI-Based High-Performance Low-Cost Computing,* March 1995.

[5] 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 Trans. on Computers*, Vol. C-32, no 2, pp. 175–189, Feb. 1983.

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

[7] 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." *Proc. of the 1993 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pp. 48–60, May 1993.

[8] 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.

[9] 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.

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

[11] 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.

[12] David Chaiken, John Kubiatowicz, Anant Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme." *Proc. of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 224–234, April 1991.

[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] David V. James, "IEEE Standard for Cache Optimization for Large Numbers of Processors using the Scalable Coherent Interface (SCI) Draft 0.35," September 1995.