The SCI Cache Coherence Protocol

Stein Gjessing
University of Oslo
P.O.B. 1080 Blindern
N-0316 Oslo 3
NORWAY

David B. Gustavson
SLAC Computation Research Group
Stanford University,
P.O. Box 4349, M/S 88
Stanford, CA 94309

James R. Goodman
Computer Sciences Dept.
4211 Computer Sci. & Stat. Bldg.
1210 West Dayton Street
Madison, Wisconsin 53706

David V. James
Advanced Technology Group
Apple Computer, MS 76-2H
20525 Mariani Avenue
Cupertino, CA 95014

Ernst H. Kristiansen
Dolphin Server Technology A.S.
P.O. Box 52, Bogerud
N-0621 Oslo 6
NORWAY

Abstract

This article discusses the current status of the Scalable Coherent Interface (SCI), IEEE standards project P1596. The SCI cache coherence protocol is scalable (up to 64K processors can be supported), efficient (memory is not involved in the common pairwise-sharing updates), and robust (data can be reliably recovered by software after transmission errors). Scalability is achieved by having the memory directory identify only the first processor sharing a cache line; other processors sharing the same line are identified by entries in a distributed doubly linked list.

M. Dubois et al. (eds.), *Scalable Shared Memory Multiprocessors*

1 Introduction

To simplify programming, multiprocessors have often assumed a shared-memory data-access model. When caches are used to improve the performance of these processors, cache-coherence protocols are needed to maintain the simple shared-memory model assumed by software. Such coherence is achieved by exchanges of read or write transactions between processors or between processors and memory. In a traditional bus-based multiprocessor, coherence is enforced by broadcasting all or a subset of these transactions to other processors (Goodman 83). Such a protocol is usually called snooping or eavesdropping.

This paper focuses on the directory-based cache coherence protocol being defined for the Scalable Coherent Interface (SCI). SCI is an IEEE standards project (P1596), which began as a spin-off from the Futurebus standardization work (Sweazey 86). Brief presentations of preliminary results of the SCI cache coherence protocol design effort are given in (Sweazey 89) and (James 90).

It became clear early in the SCI work that no bus-based connection scheme could handle the demands of the next generation of processor chips in multiprocessor configurations. The speed of buses is inadequate because of multidrop transmission line physics problems and because buses are inherently a bottleneck, i.e. they can be used by only one processor at a time.

SCI solved these problems by using a large collection of point-to-point links instead of a bus. Because links have better physical properties than buses, their signalling rates can be much higher (1000 MegaBytes/sec in the first version). Also, the bus bottleneck is avoided: distinct packets can be sent concurrently over different links within the system.

However, new protocols were needed to provide the desired bus-like services without re-introducing the bus-related bottlenecks. For example, snooping cache coherence mechanisms depend on having all transactions serialized so that all participants can snoop, thus the coherence protocols developed for Futurebus cannot be used by SCI.

A major effort was required to develop protocols to achieve these goals. Our design goals included scalability (up to 64K nodes), high performance (faster than eavesdrop or memory-directory based protocols), and robustness (recovery from an arbitrary number of transmission errors).

For a long time SCI's success was in doubt because the feasibility of meeting such ambitious goals was not obvious. However, we have been pleasantly surprized; although SCI's distributed-list structure was mandated by the scalability goal, this distributed-list structure has been found to be efficient and robust as well. We know of no other cache-coherence protocol that addresses all these goals as comprehensively as SCI does.

The rest of this paper is organized as follows: we first give a brief overview of directory-based cache coherence protocols and discuss the basic design of the SCI cache coherence protocol. We then describe its special properties, which include scalability, high performance, and robustness.

2 Previous Directory-Based Protocols

Directory-based protocols use tags to identify the processors that are actively sharing the same cache line. For each coherently cached memory line, a directory identifies the set of caches that contain this line (called the sharing set), the state of the memory line, and the state of the cache lines. In central-directory schemes the directory is contained in one system memory; in distributed-directory schemes there may be multiple memory controllers, each of which has a directory for the addresses that it supports. We are primarily concerned with distributed-directory schemes, since the bottleneck of central-directory schemes limits their usefulness to a small number of processors.

We use the terms central list and distributed list to differentiate between the coherence protocols that maintain the lists of sharing caches in memory and the coherence protocols that distribute the list among the sharing caches.

One of the first directory-based cache coherence protocols was suggested by Tang (Tang 76). Tang uses a central list to keep track of caches that cache lines. He also uses a "write-back" model, i.e. the data of a cache line is written back into memory only when needed. Store back is needed only when the cache line is dirty, the cache space is needed by another line, or the line is about to be shared by another cache. The amount of directory space used in this solution can become large (it is proportional to the number of used cache lines) and the central directory can become a performance bottleneck.

In (Agarwal 88b) it is observed that there are two difficulties in making scalable central lists: broadcasts are sometimes required, and the memory can become a performance bottleneck. For example, Archibald and Baer (Archibald 84), proposed a central list which keeps the state of the memory lines but not the location of the sharing caches, thus requiring broadcasts in order to access the caches. The Aquarius project (Carlton 90) uses directories to selectively route transactions from one bus to a grid of buses. With their interconnect topology (a bus mesh) and selective routing, the Aquarius-project computer provides scalability by increasing the bandwidth when the number of processors increases. Other computers also use hierarchical cache organizations (Wilson 87).

The cache protocol described by Censier and Feautrier (Censier 78) also uses the "write-back" model, but they call it "nonstore-through". They maintain a MODIFIED bit for each memory line, and a PRIVATE flag for each cache line. In addition there are as many PRESENT bits in each memory line as there are caches in the system. Such coherence protocols where the complete sharing set is stored with the memory line is called a "full-mapped directory" in (Chaiken 90).

(Chaiken 90) also introduces the notion of a "limited directory." A limited directory can hold only a portion of the largest potential sharing set. The sharing-set size can be reduced by invalidating old cache copies when the set size would be exceeded (Agarwal 88b). In the DASH computer (Lenoski 89) the limited directory is used to hold multicast lists when the set size has been exceeded.

In the Alewife computer (Agarwal 90) a limited directory is implemented in hardware, but a software based overflow mechanism is provided. This lets the directory overflow into RAM when the sharing set becomes large.

The last class of directories defined in (Chaiken 90) is called "chained directories." John Willis reports that variations of the chained directory protocol have been used in several research computers including Lawrence Livermore's S-1 Mark III, Symbolic's Aurora and Philips Laboratory's Strand (Willis 88). In a linked list protocol the directory information for each memory line is distributed over all cache lines that cache this memory line. Hence such a directory is also called a "distributed directory". The Stanford Distributed-Directory protocol (Thapar 90) and the SCI protocol are of this class.

The Stanford Distributed Directory uses a singly linked list to define the sharing set. The main memory line contains a (head) pointer to the last cache to access this line. The cache line in that cache contains a pointer to the previous cache that accessed the line, etc. Insertions are done at the head of the list. When one cache line wants to be deleted from the list it invalidates itself and the rest of the list.

The SCI directory uses a doubly linked list to identify the sharing set, and the list entries are distributed among the sharing caches. The memory or directory bottleneck that is reported in (Agarwal 88b) is minimized by distributing many of the directory-update operations out to the caches themselves, and by simplifying the few memory directory operations that are left. For example, memory operations can always be performed immediately, independent of the processor-cache state.

Sharing lists scale better when distributed, since the directory at the memory needs only be large enough to identify the cache at the head of the sharing list.

3 The SCI Standard

The SCI standard defines the physical and logical interface and interaction protocol for up to 64K nodes. A node is a processor (or a multiprocessor), a memory module, an I/O adapter or a combination of these, as shown in Figure 1.

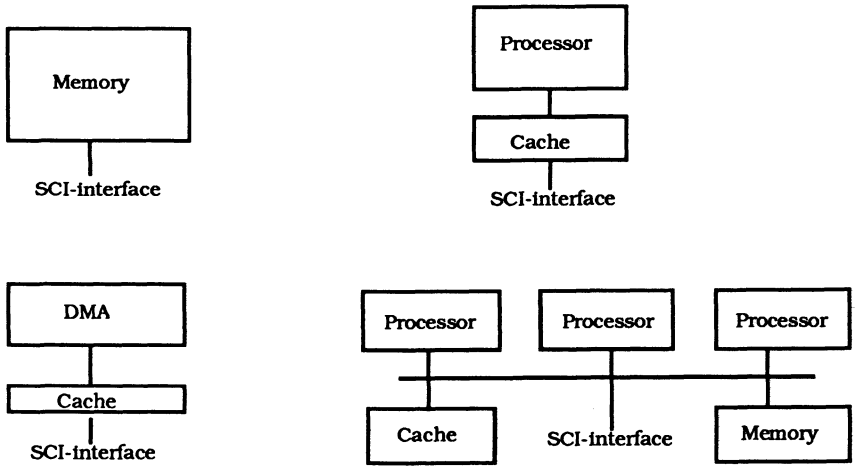

Figure 1: Four possible SCI configurations.

Note that a DMA adapter needs a cache with at least one line in order to participate in the coherence protocols.

Physical SCI memory addresses are 64 bits long; the most significant 16 bits are the node identification, while the remaining 48 bits are the physical address within the memory of the node. The coherent transactions are split into a request message and a response message. The request and response messages are called subactions.

A subaction also contains command, data, and status fields. A 16-bit CRC is included to detect transmission errors reliably. Any damaged subactions are discarded when the damage is detected; timeouts are used to detect the loss of a request or response subaction.

To simplify the coherence protocols, a standard coherence line-size of 64 bytes has been assumed. For this transfer size, the size of the subaction headers (16 bytes) is significantly less than the size of the data transfers (64 bytes). We expect to keep the same coherence line size as technological improvements increase the bandwidth of our links, since the ratio of the header size to the data size is independent of the link bandwidth for SCI.

The SCI interface consists of an input link and an output link, operating at 1 GByte/sec in initial implementations. For small systems neighboring input and output links may be directly connected, forming a ring. For larger systems these links could be dynamically connected by a switch, or a large number of small rings could be interconnected by bridging nodes. The SCI protocols were designed to support all these models transparently.

4 SCI Sharing Lists

4.1 Sharing List Structure

There are three basic operations on the SCI list: The insertion of a new cache element, the deletion of an element, and the reduction to a single-entry list. The insertion operation is used when a new cache wants to get a readable copy of the line. The deletion operation is used when a cache needs a cache line for other uses (also called roll-out). The reduction operation is used when data is written, to remove all but one of the duplicate copies in the sharing set. The remaining copy is then private and can be modified.

To facilitate these operations the SCI sharing set is identified as a doubly linked list. There is a pointer in memory to the first node in the sharing list and the other nodes in the list contain pointers to their predecessor and successor in the list (Figure 2). In memory each line that may be coherently cached contains (in addition to its data) some status information and a 16-bit node identifier which points to the first cache line (node) in the sharing list. In the caches each cache line contains (in addition to its status and data) two 16-bit node identifiers that are the backward and forward pointers to other nodes in the sharing list.

The cache coherence protocol is built on the transaction mechanism described in the previous section. The directory is modified by requests sent from one cache line to another or from a cache line to memory. These requests contain the destination address (the node identification of another cache) as well as the 64-bit physical memory address. Given a memory line address, a sharing set is a set of nodes actively sharing that memory line address, as illustrated in Figure 2.

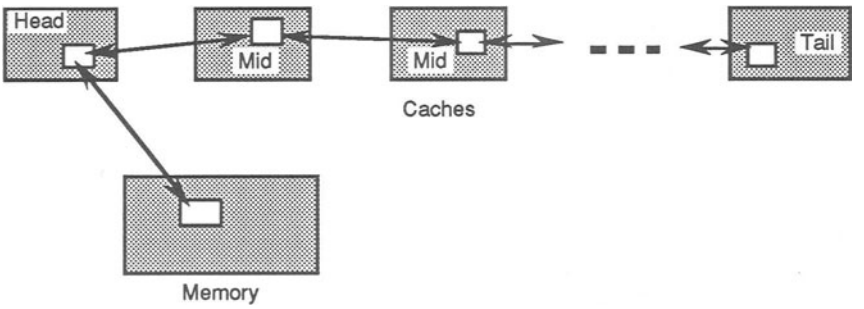

Figure 2: An SCI sharing list.

The status field of a cache line defines the meaning of the forward and backward pointers; if the line is not in a sharing list, the pointers are undefined; at the head of a list, only the forward pointer is defined; at the middle of a list, both pointers are defined; at the tail of a list only the backward pointer is defined. The status field of a memory line defines whether a list exists, and whether the memory data is valid.

The linked list data structure also makes SCI scalable in the sense that the size of the SCI directory is proportional to the total cache and memory sizes; all (maximum 64K) nodes can theoretically be members of one sharing set. In Section 9 we will discuss optimizations that improve the efficiency of updating such large sharing sets.

4.2 Sharing-List States

As stated above, the status fields in cache lines and memory lines define not only the meaning of the pointers, but also the privileges and status of the data in cache or memory. In this section we describe these status values in more detail. The coherence protocols support the stable states shown in Table 1 (to simplify the descriptions, some optional states are not described).

mem	first	(other)	last	Description
home	----	----	----	Uncached data
fresh	only_fresh	----	----	One fresh copy
"	head_fresh	----	tail_valid	Two fresh copies
"	head_fresh	mid_valid	tail_valid	More fresh copies
gone	only_dirty	----	----	One dirty copy
"	head_dirty	----	tail_valid	Two dirty copies
"	head_dirty	mid_valid	tail_valid	More dirty copies
gone	head_excl	----	tail_stale	Two copies, stale tail
"	head_stale	----	tail_excl	Two copies, stale head

Table 1: Stable Sharing-List States.

When the sharing set is empty the memory line is in the state *home*. Otherwise memory is in one of the two states *fresh* or *gone*. In the state *fresh*, memory is known to contain valid data, while in the state *gone*, memory data may be stale. When the memory state is *gone*, valid data is found in one or more cache lines of

the sharing set. Since memory updates are always completed indivisibly, memory has no transient intermediate states.

A cache line that is not in use is in the state *invalid* (shown as ---- in the table). The data in a cache line is always valid for at least one entry in the sharing list. The head node knows when memory contains *fresh* data, and reflects this by being in the state *only_fresh or head_fresh*. The contents of head entries in other states must be written back to memory when the sharing list collapses.

Additional cache line state information gives the position of the cache line in the sharing list: *head*, *mid* and *tail*. A cache line that is both the head and the tail is in the state *only*. Special two-entry list states (*head_excl&tail_stale* or *head_stale&tail_excl*) are provided to efficiently support pairwise sharing. A cache line in the *only_dirty*, *head_excl*, or *tail_excl* states can be modified.

The basic sharing-list updates are described in the following sections. The optional pairwise-sharing updates are described in a later chapter.

5 Sharing-List Updates

5.1 Insertion

On a load miss, a node joins an existing sharing list by sending a *prepend request* (R1 in Figure 3) to the addressed memory line. The target of this subaction is defined by the 16-bit node identification of the memory controller, plus the 48-bit physical line address within the node. The memory controller swaps its old head pointer with a pointer to the new node, and returns the old head pointer in an immediate *response* (S1 in Figure 3) back to the requester.

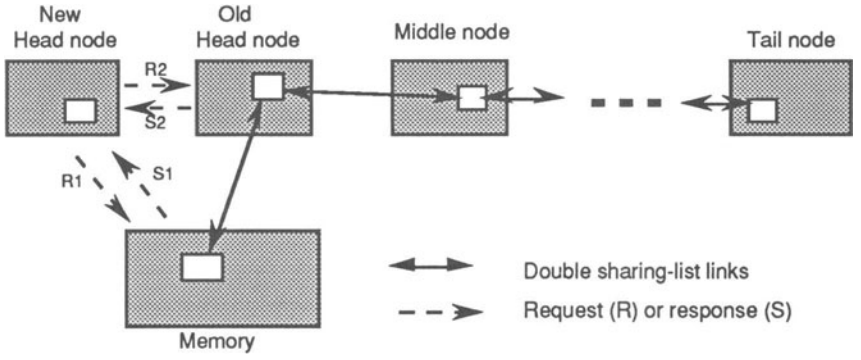

Figure 3: Sharing list insertion initiated.

After receiving this response, the new head (if the sharing list was not empty) informs the old head of its predecessor in the list by sending a *new-head* request, R2, to the old head. The old head updates its pointers and (if asked) returns a copy of its data in the response, S2. After these two transactions have completed, the sharing list has one more entry and a new head, as illustrated in Figure 4.

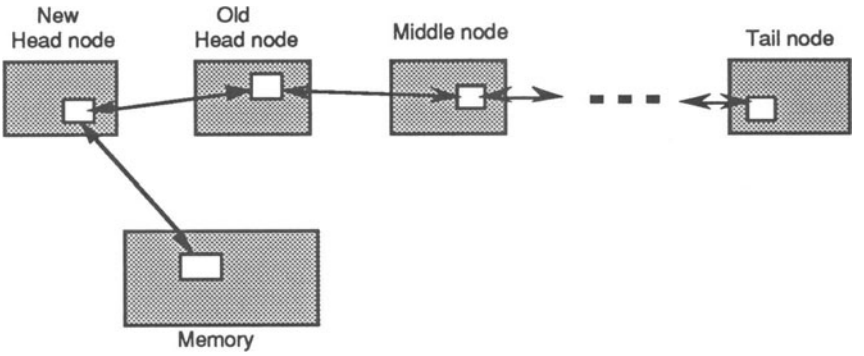

Figure 4: Sharing list insertion completed.

Several new nodes might (nearly) simultaneously prepend themselves to an existing list. Memory immediately responds to the prepend requests in the order that they are received. The new head, however, delays other would-be new heads until it has prepended itself to the sharing list. This scheme is fast and ensures forward progress. Insertion bandwidth is however limited by the constant time it takes to execute the prepend request in memory. A truly scalable architecture would insert elements into the sharing set even faster; we will briefly elaborate on this in Section 9.

5.2 Deletions

When a node no longer wants to share a memory line, it deletes its entry from the sharing list. The entry in a single-entry list is deleted by writing the data back to memory. Otherwise, the delete operation involves transfers between adjacent entries in the sharing list.

A middle-of-list cache line (called ML) has a forward pointer with value Vf and a backward pointer with value Vb, as illustrated in Figure 5. Note that backward pointers point to the left while forward pointers point to the right. If the deleting node is in the middle of the list, it first sends an *update backward* request (R1) to its successor. This request contains the node identification Vb, which normally updates the backward pointer in the successor.

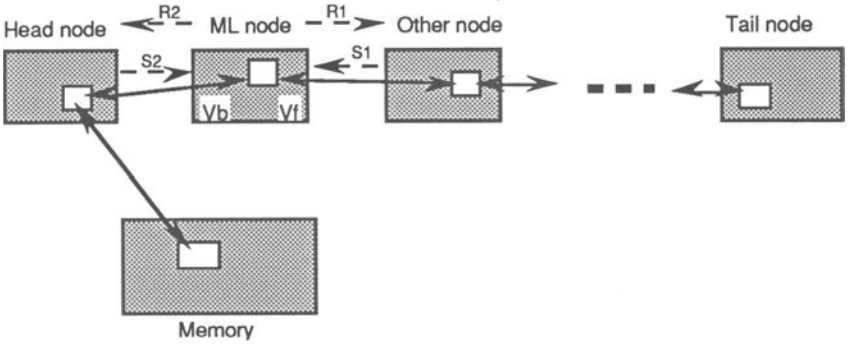

Figure 5: Sharing set deletion.

The deleting node then sends an *update forward* request (R2) to the predecessor, containing the identification, Vf, of the successor. This leaves the sharing list with one less entry, as illustrated in Figure 6.

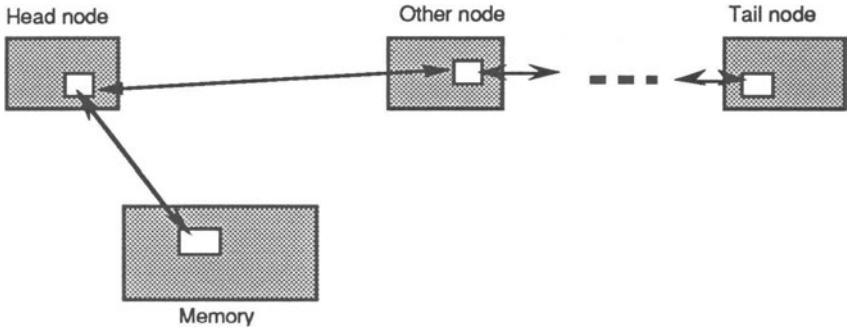

Figure 6: Sharing set deletion.

Note that these are the normal protocols for deleting an entry from a doubly-linked list. However, special care is required to resolve conflicts when deletions are concurrently performed by adjacent nodes.

The above described deletion algorithm is scalable to a large set of nodes, since many deletions can occur in parallel. In the extreme case, however, deletions in one sharing list will be serialized when all nodes delete themselves at the same time. In this case, the deletions occur only at the (dynamically changing) tail entry.

5.3 Reduction to Single-Entry Sharing List

The SCI cache coherence protocol is invalidation based: in order to write, a node must invalidate copies in the other sharing list entries. A head node has the privilege to delete all the other nodes from the list, and hence may reduce the sharing set to a single-entry list with itself as the only member. The head deletes or purges the rest of the list by first sending a purge request to the second node in the list. This node responds by returning the identification of its successor (the third node in the list). The head then sends a purge request to this next node (as seen in Figure 7). This node gives a purge reply, containing the identification of the next node in the list, etc.

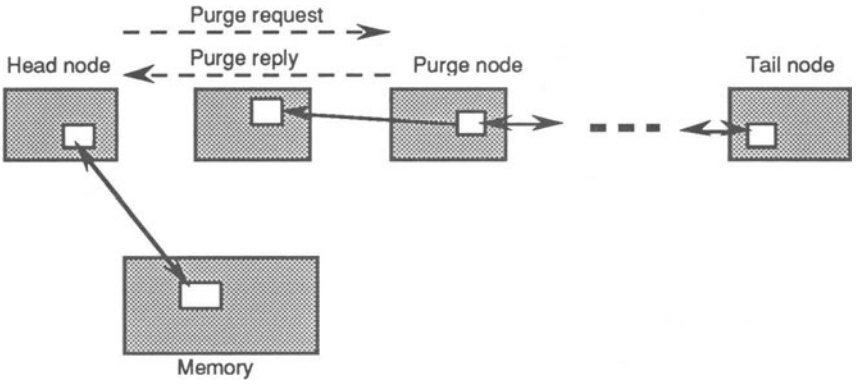

Figure 7: Reduction to single-entry list.

The time used to reduce the sharing list to a single-entry list when the data structure is a linear list is proportional to the size of the list. Previous studies have shown that sharing sets are usually not large (Eggers 88, Agarwal 88b). Hence the linear time that this operation uses will normally not be of any concern.

In the future, however, new highly parallel algorithms will perhaps not have such characteristics. An optimization of this purge operation is defined in optional extensions to the SCI base protocol: a node can, under certain circumstances, forward the purge request to both the adjacent and further distant nodes. In this way, the purge process can be concurrently active at multiple sites within the sharing list. In Section 9 we will briefly describe the data structures which support these more-efficient sharing list reductions.

SCI supports both weak and strong ordering. Strong ordering is enforced by waiting for the purge to complete before allowing the processor to proceed; for weak ordering, the processor may proceed before the purge is complete, only checking completion as needed for program correctness.

6. Scalability

A computer architecture is scalable if it is possible to add new components (e.g. processors or memory modules) and get an increased performance appropriate to the added cost.

SCI supports up to 65,000+ nodes, hence it has the potential to be scalable up to this limit. The SCI coherence protocol is scalable in the sense that that there is no central control and no globally-shared resource. Any number of memory or sharing-list operations can take place at the same time provided they do not use conflicting resources (the same part of the interconnect, the same memory module, or the same cache).

The SCI directory is scalable in the sense that any number of nodes can share the same memory line, and the size of the directory is proportional to the number of memory lines and cache lines. Deletion of elements from the sharing set is fully scalable. This operation is executed in a decentralized way and does not (usually) involve the central directory (in memory) at all.

Ideally a shared-memory system should be able to handle a large (proportional to the total number of nodes in the system) number of accesses to the same memory location "at the same time." We define a system to be logarithmically scalable if the maximum delay for any operation is proportional to log(N), where N is the number of nodes in the system. We are considering extensions to the base protocol that make SCI scalable even according to this definition. The implementation of the sharing set as a list makes two operations linear: 1) set insertion (including distribution of data to all new members), and 2) the reduction of the set. In order to make these two operations scalable, the sharing set can optionally be structured as a tree. See Section 9 for details.

7. Performance

Previous sections have described the basic SCI sharing-list operations required for coherently-shared data. SCI also provides a rich set of interoperable performance-enhancement options, which (at some cost in complexity) can be implemented as required to enhance system performance. Several of these options (DMA, pairwise-sharing, and QOLB) are discussed in the following sections.

7.1 DMA

In some of the existing RISC-based systems, DMA I/O transfers are performed non-coherently into what are otherwise coherently managed pages of memory address space. For data transfers, this dramatically complicates the I/O driver software, which has to explicitly manage the flushing of relevant cache lines. For instruction-page replacements, the problems are significant but less severe (since the data is read-only). However, special treatment of instruction-cache pages is less practical for languages which support or encourage the use of self-modifying code.

To simplify software, SCI provides coherence check options for DMA transfers; these options improve the performance of DMA-related coherence checking, while reducing the complexity of their implementation. These simple protocol optimizations are possible because of the nature of DMA participation – a DMA controller doesn't need to be added to the sharing list. However, the DMA controller is required to have a minimal (one line) cache in order to participate in the coherence protocols.

For example, consider the coherent DMA read protocols. If the memory state is home or fresh, the memory read returns the data and leaves the sharing-list state unchanged. The DMA reader is prepended to the existing sharing list (from which the read data is copied) only when the memory state is gone. We rejected the idea of reading from the sharing-list without prepending to it, since the structure of the sharing list (and hence its validity) may change between the time that memory is read and the old sharing-list head is accessed.

7.2 Pairwise sharing

The pairwise sharing option is invoked when the sharing set consists of two nodes and is constant over a period of time. At most one of the nodes may modify the data at a time, but this capability can be negotiated between the nodes without involving memory. If processor A needs to modify the line (that was last read by processor B), it sends a request (R1) to processor B. When the request is processed at processor B,

its state is changed to *stale* and a (modifiable) *excl* copy is returned (S1) to processor A, as illustrated in Figure 8 (showing processors A and B as the head and tail respectively).

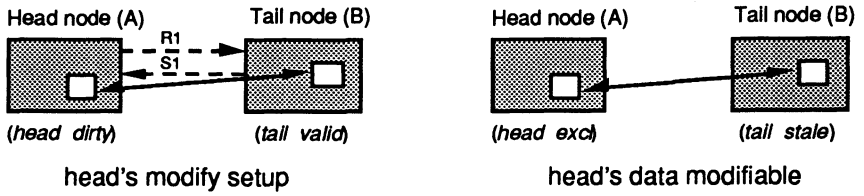

Figure 8: Pairwise sharing (head's data modified).

Similarly, when node B needs to read the line (that was last written by processor A), it sends a request (R2) to processor A. When the request is processed at processor A, its state is changed to *dirty* and a (readable) *valid* copy is returned (S2) to processor B. These steps are illustrated in Figure 9.

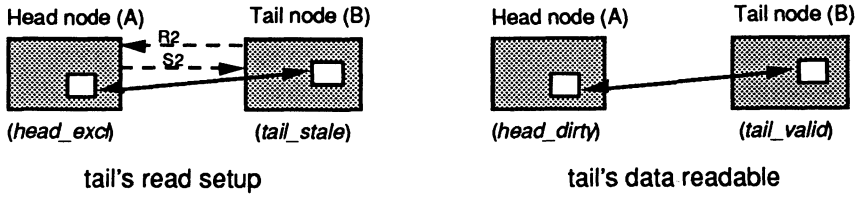

Figure 9: Pairwise sharing (tail's data read).

In this way data can be transferred directly between the caches of the two processors sharing the data. This is a fundamental advantage of distributed-list protocols over their central-list counterparts, since the memory bottleneck is avoided and the transfers are much more efficient. The performance advantages (when compared to central-list protocols) can be calculated when the pairwise-write components and node delays (as illustrated in Figure 10) are considered.

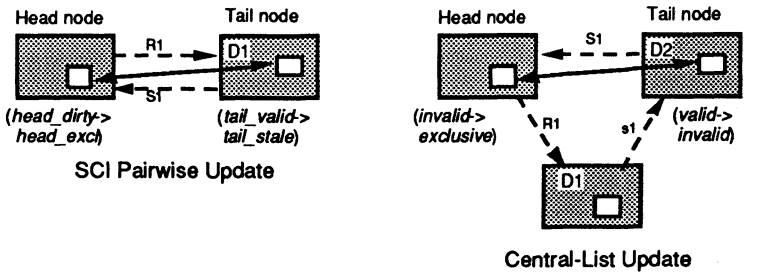

Figure 10: Pairwise-Sharing Write Sequences (SCI and Central Lists).

To simplify the comparison, assume that the node-access delays are twice the subaction transmission delays (a reasonable assumption for SCI). The pairwise performance of SCI and central-list protocols (which require an additional transaction and node-access delay) is summarized in Table 2.

Parameter	SCI	Central List
Total transaction latency	4	7
Number of subactions	2	3
Nodes accesses performed	1	2

Table 2: Pairwise-write Performance Comparison (smaller is better).

This performance distinction indirectly affects system performance, since pairwise-sharing is only one form of data sharing. However, we expect to see large amounts of pairwise-sharing, so this distinction should significantly affect overall system performance. Note that this performance comparison does not consider the performance loss associated with implementing specialized 3-phase transaction sequences (R1,s1,S2), which are assumed by the central-list coherence protocols (DASH and SDD).

7.3 QOLB

In a multiprocessor computer, implementation of primitives for data sharing and synchronization is of vital importance. Sharing and synchronization operations can be done by software, with simple coherence hardware support. Special software algorithms, based on the use of the indivisible swap and compare&swap primitives, are sufficient to implement a variety of efficient non-blocking enqueue and dequeue operations. These algorithms can be designed to minimize the number of unnecessary coherence transactions sent through the interconnect (Mellor-Crummey 90b).

However, better performance is possible when the synchronization mechanisms are integrated with the cache coherence protocol. These generally involve direct cache-to-cache transactions that are delayed until the data is in a useful (unlocked) state. It should then be possible (and easy) to write efficient programs that utilize the shared memory and the caches in an optimal way.

In SCI, a special QOLB (Queue On Lock Bit, Goodman 89) mechanism creates a queue of processors, where only the first processor has access to the line in question. When this first processor has finished using the data, the ownership is automatically transferred to the next processor in the queue. Where possible, enrollment in the queue also effects a prefetch of the desired data.

The SCI version of QOLB uses a sharing list where all but one processing node (at the tail) is in the *head_idle* or *mid_idle* states, and the owner of the cache line is in the *tail_need* state. The processing nodes prepend themselves into the sharing list and passively wait in the idle state until the tail's copy is passed to them. When a QOLB lock is released, the tail passes its copy to the previous entry in the sharing list, which becomes the new tail.

The QOLB protocols use the sharing-list tags provided by the basic SCI protocols, but additional cache-line states (like *head_idle* and *mid_idle*) are required to support this optional extension. The QOLB mechanism also works efficiently for two-element sharing lists, using the pairwise-sharing protocols described previously.

Such an SCI QOLB mechanism only assures exclusive access to a line as long as it resides in cache. Therefore, the QOLB locks are only used for scheduling synchronization instruction sequences; indivisible instruction sequences (such as swap) are relied on to maintain the integrity of shared data structures.

8 Fault tolerance and error recovery

The SCI transaction protocol ensures that requests and responses are correctly transferred between the nodes. The SCI cache coherence protocol is based on split transactions (request and response subactions) that time out when a response is not returned within a software-specified timeout interval. When the timeout is exceeded, software is expected to invoke a fault recovery routine.

The SCI cache coherence protocol can recover from any number of transmission errors. To support this recovery, the deletion of a previously dirty cache line is always delayed until after the dirty data has been reliably copied to another location. Otherwise, dirty data could be lost if the transaction containing the dirty data were dropped.

When data is being modified, for example, a new head prepends to an *only_dirty* entry and leaves this entry in the *tail_stale* state. If the prepend transaction (which contains the dirty data) is damaged, the dirty data can be recovered from the *tail_stale* copy. Note that the robust prepend algorithm also leaves the sharing-list entries in efficient pairwise-sharing states (*head_excl* and *tail_stale*).

Special care is also required when an *only_dirty* copy is returned to memory; a two transaction sequence is required. The dirty data is first written back to memory (transaction 1) and the cache-line ownership of the clean line is then returned to memory (transaction 2). The increased use of interconnect bandwidth is small, since the data (which is the largest part of the packet) is only transmitted once.

Recovery from errors is handled by fault-recovery software that is activated by the transaction-timeout error. The recovery strategy is to 1) disable coherent memory accesses to the affected cache-line address(es), 2) return dirty cache-line copies to memory, 3) discard the residual sharing-list structures, and 4) re-enable coherent access to the affected cache-line addresses.

To simplify the recovery, the fault-recovery software is expected to use a non-coherent memory-access mode. Special transactions are provided to selectively disable coherent memory accesses while the recovery is being performed. The recovery software extracts the affected cache lines from remote processors and leaves their cumulative state in a pre-reserved range of memory address space. Software searches through these residual sharing-list states and identifies the cache line with the most-recently modified data. After the modified data is copied to its proper memory location, the memory is re-enabled for accesses to the affected cache line.

No attempt is made to reconstruct the sharing-list structure, since similar structures can be dynamically re-created after the fault-recovery process has completed. Identifying the cache-line with the most-recently modified data can be a complex task, but with our low transmission error rates the performance of the recovery process should not be a concern.

9 Ongoing Research

The linear list structures appear to be adequate for moderate-sized systems, but in very large systems (thousands of processors) more efficient structures may become important. We are working on extensions to the SCI protocols that have logarithmic rather than linear performance, supporting combining of requests in the interconnect when congestion occurs, rapid parallel distribution of responses to combined requests, and fast purging of lists.

The strategy we are pursuing uses a third pointer for each line in each cache controller, that points to a distant (or at least non-adjacent) part of the list. Each sharing-list would have its forward and backward pointers (to its adjacent neighbors) as well as an approximate pointer to a more distant (but closer to the tail) sharing-list entry, as illustrated in Figure 12. Note that this is an illustration of an approximate pointer structure; the analysis and selection of approximate pointer structures is being addressed as an extension to the base SCI standard.

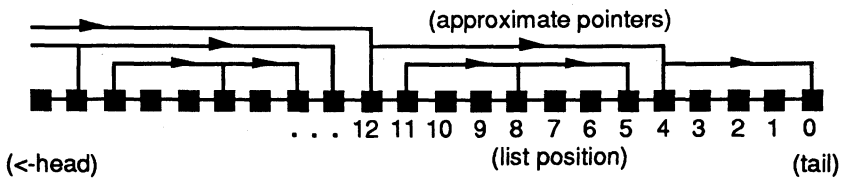

Figure 11: Approximate pointer structures.

These pointers allow tree structures to be created in order to achieve logarithmic performance. However, the tree is not maintained (which would be prohibitively expensive) so the pointer is checked for validity as it is used (in case the target node has subsequently rolled out the line, for example). Thus we call them approximate pointers: they approximately formed an optimal tree when created, and they may still point to a useful place when they are used.

Memory requests can be combined in the interconnect (Gottlieb 83, Edler 85) or at the memory controller. As requests combine, the originating nodes are informed of their position in the list (measured as distance from the tail). Each node uses its position value to determine how to create its approximate pointer, following the linear list pointers initially, in a series of hops we call recursive doubling. Choosing optimal pointer strategies is a subject of our current SCI extensions work. One needs good performance in both directions (the initial data fetch as well as the final data purge), with gradual degradation of performance when a few of the nodes leave the list.

The linear lists are always correct, and are relied on for correct performance wherever the approximate pointers fail. The approximate pointer scheme is thus merely an optimization that gives logarithmic performance in many situations, and can be added to the SCI standard at some later date when it is well understood.

10 Conclusion

The cache coherence protocol of SCI is invalidation based and uses distributed sharing lists and directories. The basic cache coherence operations are efficient and simple. The high performance is particularly due to the fact that a new cache line can join the set of sharers by one simple and extremely fast memory operation. Other operations such as deletion and breaking down of the sharing set are executed decentralized and possibly in parallel while new caches concurrently join the sharing list at full speed.

The SCI cache directory is scalable. The directory overhead is a maximum of 3.5% of the memory and 7.4% of the cache size. In this data structure up to a theoretical maximum of 65,000+ caches can share the same line.

The SCI cache coherence protocol is based on point-to-point subactions arranged as transactions, where the request subaction is sent and a response subaction is returned. By avoiding specialized 3-party transactions (as are used in the SDD and DASH protocols), the simpler SCI transactions should be faster and more reliable (errors can be trapped at the earliest possible moment). Unlike the DASH protocols, the SCI coherence protocols allow transactions to be completed in any order, which places fewer constraints on the design of the interconnect.

The distributed-list protocols have other advantages over their memory-list counterparts. The size of the memory-resident directory remains largely constant as the number of processors is increased; special directory overflow protocols (as implemented in the DASH and Alewife protocols) are not required. The memory-update protocols can always be completed immediately, which eliminates the livelock/deadlock conditions that arise if memory rejects new requests while processing others.

We are not aware of any other coherence protocol that has addressed the issue of data recovery after transmission errors. SCI not only addresses this issue, but supports recovery from an arbitrary number of transmission errors. Although other protocols might be extendable to provide such capabilities, we suspect that the specialized transaction-set requirements of some distributed protocols (3-party transactions and constrained ordering, for example) would make them difficult to extend.

Because it is based on three simple directory operations, the SCI cache coherence protocol is relatively easy to understand. However, SCI supports a high degree of parallelism, and several of these directory operations could be performed simultaneously on the same sharing list. Thus, it is not immediately obvious that the operations will give correct results in all cases of interaction. Simulations are being performed in order to find errors in the draft specifications of the cache coherence algorithms. Since the SCI coherence protocols are specified as C-code routines, exact simulations of the specification can be performed.

Given the large numbers of combinations of transient states, these simulations might not find all errors in the draft specifications. A mathematical proof is therefore being performed on the cache coherence protocol (Gjessing 90a-c).

The SCI cache coherence protocol has been designed based on currently known technology and research results. Since no real execution traces exist for directory-based shared-memory multiprocessors, some design choices have probably been made without sufficiently sound arguments. When the first SCI based computers are operational we will be able to evaluate the design and eventually correct and improve it. It will be very interesting to see what kinds of new algorithms evolve when new large-scale cache-coherent multiprocessors are in regular use.

11 Acknowledgements

The SCI cache protocol was designed by a number of people. The first initiative was taken by Paul Sweazey, former leader of the cache coherence task group for the IEEE P896 Futurebus, when he started the SuperBus study group in the IEEE Computer Society. Others initially or currently involved with cache-coherence issues include Knut Alnes, Bjørn Bakka, Håkkon Bugge, Craig Hansen, Marit Jensen, Sverre Johansen, Ross Johnson, Michael Koster, Stein Krogdahl, John Moussouris, Ellen Munthe-Kaas, Randy Rettberg, Alan Smith, Guri Sohi, and Phil Woest. Thanks are due to these and all others that have contributed to SCI in one way or another.

12 References

Agarwal, A., Hennessy, J., Horowitz, M.: Cache Performance of Operating System and Multiprogramming Workloads. ACM Trans. on Computer Systems, Vol. 6, No. 4, Nov. 1988a.

Agarwal, A., Simoni, R., Hennessy, J., Horowitz, M.: An Evaluation of Directory Schemes for Cache Coherence. 15th Annual International Symposium on Computer Architecture, 1988b.

Agarwal, A., Chaiken, D., Fields, C., Johnson, K., Kranz, D., Kubiatowicz, J., Kurihara, K., Lim, B.H., Maa, G., Nussbaum, D.: The MIT Alewife Machine: Promoting a Fuzzy Hardware-Software Boundary. Workshop on Scalable Shared-Memory Architectures, Seattle, May 1990.

Archibald, J., Baer, J-L.: An Economical Solution to the Cache Coherence Problem. 11th International Symposium on Computer Architecture, 1984.

Borg, A., Kessler, R.E., Wall, D.W.: Generation and Analysis of Very Long Address Traces. 17th Annual International Symposium on Computer Architecture, Seattle, Washington, 1990.

Bugge, H.O., Kristiansen, E.H., Bakka, B.O.: Trace-driven Simulations for a Two-level Cache Design in Open Bus Systems. 17th Annual International Symposium on Computer Architecture, Seattle, Washington, 1990.

Carlton, M., Despain, A.: Aquarius Project. IEEE Computer, June 1990.

Censier, L.M., Feautrier, P.: A New Solution to Coherence Problems in Multicache Systems. IEEE Trans. on Computers, Vol. 27, No. 12, Dec. 1978.

Chaiken, D., Fields, C., Kurihara, K., Agarwal, A.: Directory-Based Cache Coherence in Large-Scale Multiprocessors. IEEE Computer, June 1990.

Edler, J., Gottlieb, A., Kruskal, C.P., McAuliffe, K.P., Rudolph, L., Snir, M., Teller, P.J., Wilson, J.: Issues Related to MIMD Shared-Memory Computers: the NYU Ultracomputer Approach. 12th International Symposium on Computer Architecture, 1985.

Eggers, S., Katz, R.: A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. 15th Annual International Symposium on Computer Architecture, 1988.

Eggers, S., Katz, R.: The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. 3rd Symposium on Architectural Support for Programming Languages and Operating Systems, 1989.

Gjessing, S., Krogdahl, S., Munthe-Kaas, E.: Formal Specification and Verification of SCI Cache Coherence, Univ. of Oslo Dept. of Informatics Research Report No. 142, ISBN 82-7368-048-7, August 1990a.

Gjessing, S., Krogdahl, S., Munthe-Kaas, E.: Approaching Verification of the SCI Cache Coherence Protocol, Univ. of Oslo Dept. of Informatics Research Report No. 145, ISBN 82-7368-051-7, August 1990b.

Gjessing, S., Krogdahl, S., Munthe-Kaas, E.: A Top Down Approach to the Formal Specification of SCI Cache Coherence, Univ. of Oslo Dept. of Informatics Research Report No. 146, ISBN 82-7368-052-5, August 1990c.

Goodman, J.R.: Using Cache Memory to Reduce Processor-Memory Traffic. 10th International Symposium on Computer Architecture, 1983.

Goodman, J., Vernon, M., Woest, P.: Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. 3rd Symposium on Architectural Support for Programming Languages and Operating Systems, 1989.

Gottlieb, A. et al.: The NYU Ultracomputer — Designing an MIMD Shared Memory Parallel Computer. IEEE Trans. on Computers, C-32, 2 pp 175-189 Feb. 1983.

James, D.V., Laundrie, A.T., Gjessing, S., Sohi, G.S.: Scalable Coherent Interface. IEEE Computer, June 1990.

Lamport, L.: How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Trans. on Computers, Vol. 28, No. 9, pp 690–691, Sept. 1979.

Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A., Hennessy, J.: The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. Technical Report No. CSL-TR-89-404, Computer Systems Laboratory, Stanford University, December 1989.

Mellor-Crummey, J.M.: Concurrent Queues: Practical Fetch-and-Φ Algorithms. Technical Report 229, University of Rochester, Computer Science Department, November 1987.

Mellor-Crummey, J.M.: private communication, May 1990a.

Mellor-Crummey, J.M., Scott, M.L.: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. Rice Technical Report COMP TR90-114, Rice University, Department of Computer Science, May 1990b.

Przybylski, S., Horowitz, M., Hennessy, J.: Characteristics of Performance-Optimal Multi-Level Cache Hierarchies. 16th International Symposium on Computer Architecture, 1989.

Sweazey, P., Smith, A.J.: A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. 13th International Symposium on Computer Architecture, 1986.

Sweazey, P.: Cache Coherence on SCI. IEEE/ACM Computer Architecture Workshop, Eilat, Israel, May 1989.

Tang, C.K.: Cache System Design in Tightly Coupled Multiprocessor Systems. AFIPS Conference Proceedings, Vol 45, National Computer Conference, 1976.

Thapar, M., Delagi, B.: Stanford Distributed-Directory Protocol. IEEE Computer, June 1990.

Willis, J.: Cache Coherence in Systems with Parallel Communication Channels and Many Processors. Document no. TR-88-013, Philips Laboratories – Briarcliff, March 1988.

Wilson Jr., A.W.: Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors. 14th International Symposium on Computer Architecture, 1987.