

3-12-1999

Enhancements to the Scalable Coherent Interface Cache Protocol

Robert J. Safranek
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds



Part of the [Electrical and Computer Engineering Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

Safranek, Robert J., "Enhancements to the Scalable Coherent Interface Cache Protocol" (1999).
Dissertations and Theses. Paper 3977.
<https://doi.org/10.15760/etd.5858>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

THESIS APPROVAL

The abstract and thesis of Robert J. Safranek for the Master of Science in Electrical and Computer Engineering were presented March 12, 1999 and accepted by the thesis committee and the department.

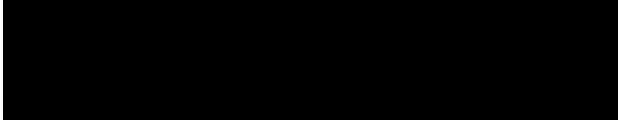
COMMITTEE APPROVALS:



Michael A. Driscoll, Chair



Douglas V. Hall



Andrew Tolmach
Representative of the Office of Graduate
Studies

DEPARTMENT APPROVAL:



Rolf Schaumann, Chair
Department of Electrical and Computer
Engineering

ABSTRACT

An abstract of the thesis of Robert J. Safranek for the Master of Science in Electrical and Computer Engineering presented on March 12, 1999.

Title: Enhancements to the Scalable Coherent Interface Cache Protocol.

As the number of NUMA system's cache coherency protocols based on the IEEE Std. 1596-1992, Standard for Scalable Coherent Interface (SCI) Specification increases, it is important to review this complex protocol to determine if the protocol can be enhanced in any way. This research provides two realizable extensions to the standard SCI cache protocol. Both of these extensions lie in the basic confines of the SCI architectures.

The first extension is a simplification to the SCI protocol in the area of prepending to a sharing list. Depending if the cache line is marked "Fresh" or "Gone", the flow of events is distinctly different. The guaranteed forward progress extension is a simplification to the SCI protocol in this area; making the act of prepending to an existing sharing list independent of whether the line is in the "Fresh" or "Gone" state. In addition, this extension eliminates the need for SCI command, as well as distributes the resource requirements of supplying data of a shared line equally among all nodes of the sharing list.

The second extension addresses the time to purge (or invalidate) an SCI sharing list. This extension provides a realizable solution that allows the node being invalidated to acknowledge the request prior to the completion of the invalidation while maintaining the memory consistency model of the processors of the system.

The resulting cache protocol was developed and implemented for Sequent Computer System Inc. NUMA-Q system. The cache protocol was run on systems ranging from eight to sixty four processors and provided between 7% and 20% reduction in time to invalidate an SCI sharing list.

ENHANCEMENTS TO THE SCALABLE COHERENT INTERFACE CACHE PROTOCOL

by

ROBERT J. SAFRANEK

**A thesis submitted in partial fulfillment of the
requirements for the degree of**

**MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING**

**Portland State University
1999**

Acknowledgements

It is with a great deal of gratitude that I thank Sequent Computer Systems Inc. for the opportunity to conduct this research. Sequent Computer Systems allowed me access to state of the art workstations, system platforms, and test equipment that cost millions of dollars.

I would like to specifically thank several individuals who were key in making this research a success.

- Jim Wilson, the Director of Hardware Engineering, who allowed access to the hardware for this project.
- Don Desota, Ruth Forester, Carl Love, and James Donnelly, the Performance Group, who were the first to volunteer to run my new cache coherency protocol on their 100Gbyte database.
- Paul McKenney, who developed the Share List Test software which was key in measuring the performance differences between the two cache protocols.
- Thomas Lovett, who pointed out my initial omission of not including write-back requests with interrupts and read responses in the Reduced List Invalidation extension as events that must be queued behind pending invalidates.

- Frieda Safranek, who took the time to edit this entire paper.
- Frieda, Stefan and Lee Safranek, my family, who have tolerated living me through this endeavor.

It should be noted that this research was done with the consent of Sequent Computer Systems Inc. and is to be considered the intellectual property of Sequent Computer Systems Inc.

Table of Contents

List of Tables	v
List of Figures	vi
1. Introduction	1
1.1 Guaranteed Forward Progress	3
1.2 Reduce List Invalidation Time	4
1.3 Organization of the Document	7
2. Symmetric Multiprocessing Overview	9
2.1 Concepts of Cache Coherence Protocols for a Bus Based System	11
2.2 Review of NUMA Terminology	13
2.3 High Level Description of the NUMA-Q	16
3. Detailed System Architectural Description	19
3.1 Node Module Description	19
3.2 Interconnect Description	22
4. Guaranteed Forward Progress Extension	26
4.1 Reading of a Home Line	27
4.2 Reading of a Fresh Line	29
4.3 Reading of a Gone Line	30
4.4 Description of the Guaranteed Forward Progress Extension	31
4.5 Reasoning for the Guaranteed Forward Progress Extension	32
4.6 Comparison between the Standard and the New Protocol	36
4.6.1 Communication with the Local Node	36
4.6.2 Communication with the “Old” Head of a SCI Sharing List	37
5. Reduce List Invalidation Time Extension	39
5.1 Comparison of the Two Invalidation Methods	43
5.2 Comparison of the Invalidation Methods of the Standard and New Protocols	53
5.3 Merging of Reduced List Invalidation with List Invalidation Method	54
6. Testing and Measured Results	55
6.1 Development Strategy	55
6.1.1 Simulation Environment	57
6.1.2 Development Environment	58
6.2 Debug Environment	59
6.3 Measured Results	60
6.3.1 Logic Analyzer Traces	60
6.3.2 Lock and Invalidate Tests	65

6.3.3 Read Measurements	74
6.3.4 Database Measurements	77
7. Observation Section	86
7.1 Performance Gains and Drawbacks	86
7.1.1 Ideal versus Real Performance Gain for Invalidation Extension	86
7.1.2 Elimination of Additional Ten Clock Penalty	87
7.2 Cache Coherency Validation Techniques	88
7.2.1 Formal Verification	88
7.2.2 Simulation Environment	89
7.2.3 System Level Cache Coherency Tests	89
7.3 Additional Areas of Research Uncovered with the SCI Protocol	90
7.3.1 Merging the two Invalidate Extensions	91
7.3.2 Reducing Processor Read Latency when Prepending to a Fresh List	94
7.3.3 Parallelizing the Rollout/Installation Process	95
8. Summary and Conclusion	97
References	103
Appendix - Glossary of SCI Terminology	108

List of Tables

Table 5.1 - Possible Cacheable Ordering Scenarios	40
---	----

List of Figures

Figure 2.1 – Simple SMP Block Diagram.....	9
Figure 3.1 - Simple Node Block Diagram.....	20
Figure 3.2 - High Level Block Diagram of the IQ-Link.....	21
Figure 3.3 - SCI Memory State Diagram.....	24
Figure 3.4 - SCI Cache State Diagram	25
Figure 4.1 – Reading of a Home Line	29
Figure 4.2 - SCI Cacheable Read Request Flow.....	33
Figure 4.3 – Guaranteed Forward Progress Cacheable Read Request Flow	35
Figure 5.1 Standard SCI Invalidation Flow of Events.....	44
Figure 5.1A - Standard SCI List Invalidation	46
Figure 5.2- Reduced List Invalidation Flow of Events.....	47
Figure 5.3 - Reduced List Invalidation Sequence	50
Figure 5.4 - Pending Invalidation Flow of Events.....	51
Figure 6.1 – Example of Protocol Engines Program Language	56
Figure 6.1 – System Debug Environment	59
Figure 6.3 Logic Analyzer Trace of a Read Response	62
Figure 6.4 - Trace of Std. Protocol Invalidation Sequence.....	63
Figure 6.5 - Trace of New Protocol Invalidation Sequence.....	64
Figure 6.6 – Share List Invalidation Time.....	69
Figure 6.7 – Share List - Atomic Invalidation Time	70
Figure 6.8 – Share List - List Invalidation Time.....	71
Figure 6.9 – Share List – List Atomic Invalidation Time	72
Figure 6.10 – Invalidation Time Differences between Standard and New Cache Protocols	73
Figure 6.11 – Percentage change Between the Two Protocols	74
Figure 6.12 – Read Response versus List Invalidation Time	76
Figure 6.13 – Four Node Database System.....	80
Figure 6.14 – Number of Database Transactions	81
Figure 6.15 – Local Cacheable Accesses.....	82
Figure 6.16 – Remote Cacheable Accesses.....	83
Figure 6.17 – Rd. Requests versus SCI Inval. Requests.....	84
Figure 7.1- Ideal SCI List Invalidation Method	93
Figure 7.2 - Total Remote Cache Read Requests versus Rollout Requests .	95

1. Introduction

“Given the limitations of bus-based multiprocessors, CC-NUMA is the scalable architecture of choice for shared-memory machines. The most important characteristic of the CC-NUMA architecture is that the latency to access data on a remote node is considerably larger than the latency to access local memory” [51]. In addition to the academic based NUMA systems, several commercially available systems are based on a NUMA architecture. Examples of commercial systems based on a CC-NUMA architecture are now available from companies such as HP (Exemplar)[41], Data General (NUMALiINE), SGI (Origin 2000), and Sequent (NUMAQ)[53]. A typical design has a number of nodes, each node consisting of one or more processors, a portion of the system's I/O and a portion of the system's memory [51].

For these systems to be viable the issue of remote latency must be addressed. One primary method to address the discrepancy in latency of remote versus local accesses is minimizing remote accesses. Systems with buffer allocation algorithms that take into account the locality of the memory being allocated add greatly to the performance of the system. However, eventually the issue of remote latency must be addressed. For CC-NUMA architectures to be viable solutions, effective methods to reduce latency of remote accesses must be identified.

The systems developed by Convex, Data General, and Sequent Computer Systems Inc. are based on the Scalable Coherent Interface (SCI) cache coherency protocol. This protocol is an IEEE standard (Std 1596-1992). "The SCI coherence protocols are based on a distributed directories scheme [21]. The SCI protocol is based on a doubly linked list structure. Each node sharing a cache line keeps track of its forward and backward neighbor as well as the state of the cache line relative to its position on the list. An issue with these systems is the additional latency of reading a remote cache line compared to a local access. Another issue with this type of protocol is the time to purge a sharing list of a cache line when the line is being written. Since the sharing list is distributed, the list must be purged one node at a time. Therefore, the longer the sharing list, the longer the latency.

Our research provides two extensions to the SCI protocol to decrease the latency of an SCI based system. The first is a simplification to the protocol in the area of when and how to supply data for a cacheable read. The second is in the area of reducing the list invalidation time of a sharing list. Both extensions are realizable and provide for a processor memory consistency model. The first of these extensions is referred to as "Guaranteed Forward Progress" extension. The other is referred to as the "Reduced List Invalidation Time" extension.

The goals of this research are to develop, implement, test and report the results for the guaranteed forward progress and reduce list invalidation time extensions on a Sequent Computer System Inc. NUMA-Q system. The Sequent system is a symmetric multiprocessing (SMP) system that is a combination of bus based four processor nodes that are interconnected with a hierarchical interconnect. This system is a CC-NUMA architecture and is based on the Scalable Coherent Interface (SCI) cache protocol for the hierarchical interconnect [53]. As stated earlier, the extensions are in the areas of:

- Guaranteed forward progress for cache read accesses of a remote line.
- Reduced latency during the list invalidation sequence.

What makes this an achievable research project is the ability of the third level cache controller of the NUMA-Q system to be “programmable”. This cache controller is based on a protocol engine architecture, where the protocol is represented as a “program” which is downloaded to the cache controllers at the time of initialization.

The characteristics of the cache protocol of the system are changed with the changing of the “program”. By exploiting the programmability characteristic of the third level cache controller, these extensions were developed, implemented and tested on the commercially available NUMA-Q system.

1.1 Guaranteed Forward Progress

To simplify the NUMA-Q implementation, a cache line was considered “Home”

if it was held in any device local to that particular node (i.e. a processor, memory, IO interconnect). Also, the bus of the “home” node is considered the serialization point for accesses. These considerations give local processors an unfair advantage to memory, which is resident on this node. The resulting condition is that remote processors can be denied access to “hot spots” in memory, like cache base locks. The resulting condition is a system with poor scaling characteristics. Another issue to consider is the SCI protocol’s current level of complexity. Currently SCI provides two distinct sequences for a remote node to read a line which is “Fresh” versus “Dirty”.

This research provides a simplification to the current SCI protocol. This simplification is realizable in the SCI two bit memory state diagram and causes only minor changes to the SCI cache state diagram. This “guaranteed forward progress” approach is limited to a field of two bits (limiting the maximum number of local states to four). This is due to the directory structure of the SCI protocol. The directory structure allocates for each local cache line a byte of information. Two bits of the byte represent the state of the line at the home node and the additional six bits for the node ID of the first element in the sharing list.

1.2 Reduce List Invalidation Time

Another area, which limits multi-node performance, is the time to invalidate a sharing list for a cache line. Most CC-NUMA machines (FLASH, Alewife, NUMAchine, Sequent’s NUMA-Q specifically) maintain their respective memory consistency models by not acknowledging any invalidation request

prior to the actual completion of the invalidate request of the sharing nodes [47, 4, 50, 53]. In general a processor's cacheable write is allowed to complete only after all other copies of that particular cache line have been invalidated. Our extension deviates from this procedure by providing a mechanism to acknowledge the invalidation early and still maintain the same memory consistency model.

This extension is applicable to any CC-NUMA cache coherency protocol. Since the actual implementation was done on an SCI base platform the description of this extension is done based on the SCI invalidation sequence. In SCI, only the head of the sharing list can write a cache line. This act of writing must also include the invalidation of the sharing list. SCI communication is based on a simple request/response packet format. The head of the list issues an invalidate request to the node immediately below it on the sharing list. That node would then invalidate its copy of the line and respond with its state and pointer information. The head continues to reissue invalidate requests until it receives the response from the tail of the list (i.e. the last element on the sharing list).

The list invalidation procedure continues until every node on the sharing list has invalidated its copy of the cache line. The SCI protocol has overlooked two issues in the invalidation time. These are:

- The time of node to invalidate a cache line.
- The time to process a SCI request or response packet.

Our extension provides a realizable improvement in performance for reducing

the list invalidation time while maintaining a processor consistency model and staying in the general constructs of the SCI protocol. This extension provides a higher performance invalidation sequence. This is done based on earlier acknowledging of invalidates at a remote node and stalling read responses at the remote node until all posted invalidates are completed.

What makes this a beneficial enhancement to the protocol is that the size of a typical system keeps growing. This was not an issue for systems comprised of two to four nodes (an eight to sixteen processor system). The invalidation time of a sharing list becomes an issue as the size of the system approaches sixteen nodes (or 64 processors).

Unlike the previous extension, the reduced list invalidation extension is a more complex solution, due to the fact that the cache controller is acknowledging the completion of a task early. Another way of stating the requirements of the cache controller in this role is when any request is acknowledged early, it becomes the responsibility of that agent to maintain the ordering requirements of the individual processors to guarantee “correct” operation. The fundamentals of this solution are as follows:

- Remote node receives an Invalidate Request. It immediately sends the SCI response acknowledging the request. This response contains the node’s current state and pointer information.
- Invalidating Cache Controller sets a bit in an array of bits. These bits signal that an Invalidate was acknowledged early.
- Remote node issues the invalidate request to its local bus. Upon

completion of the request the bit in the array is cleared.

- Remote node upon receiving either a cacheable read response, interrupt, non-cacheable write or read, defers posting them to the bus until all the currently posted pending invalidate bits have been cleared.

These deferred actions are required to prevent the following cacheable errors from occurring:

- Cacheable read passing a write.
- Write passing a write.

Note in a processor consistency model, all processors do not have to observe all the writes in the system as the writes occurred. Also, a processor has no ordering requirement of observing writes from different processors. However, the processors can only have access to the data in the order the data was written by the given processor.

It should be mentioned that an obvious extension to purging a sharing list is developing an extension to forward requests and eliminate the intermediate responses. A detailed description of an Invalidate request forwarding extension is described in our paper “Fast Invalidate Extension for the Scalable Coherent Interface”[52]. It should be noted that this extension does not preclude the merging of the two invalidate extensions. These extensions are completely compatible.

1.3 Organization of the Document

The rest of the thesis is organized as follows:

- Section 2 - Symmetric Multiprocessing Overview
- Section 3 – Detail Architectural Description of the system used to develop these extensions.
- Section 4 – Guaranteed Forward Progress Extension
- Section 5 – Reduce List Invalidation Time Extension
- Section 6 – Testing and Measured Results
- Section 7 – Research Observations
- Section 8 – Summary and Conclusion
- Glossary of Terms
- References
- Appendix

2. Symmetric Multiprocessing Overview

Systems that utilize a single global address space and multiple processors are referred to as Shared-Memory Multiprocessor systems (or SMP). The basic structure of an SMP system is provided in figure 2.1.

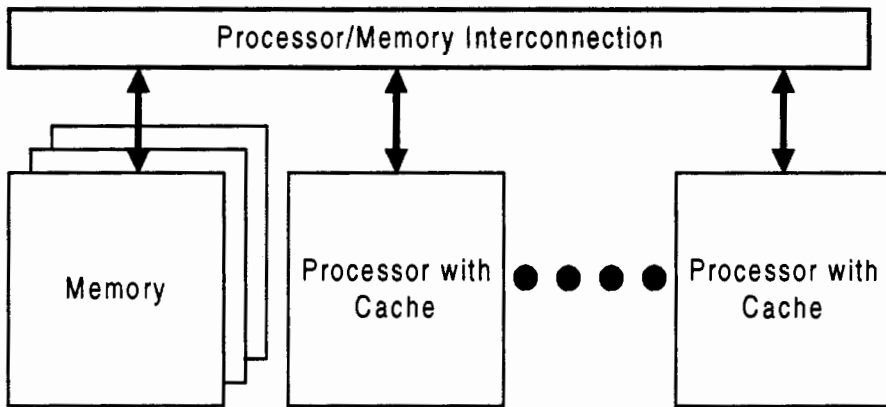


Figure 2.1 – Simple SMP Block Diagram

These systems, to deal with the issue of the “memory bottleneck” problem, typically utilize some type of caching structure for the processors. The cache contains those lines that a particular processor (or group of processors) is accessing. At any given time and for a particular cache line, zero to “n” of the caches can have the line in some cacheable state (or not). As with the caches, the memory subsystem for a particular line could have a valid copy (or not). The common interconnect for an SMP system was the system bus. The bus was the cache coherence mechanism of the system [26]. As the

processor increased in frequency operation so did the bus speed. As the speed increased the physical length of the system bus decreased. Today “commodity” bus based systems typically can only support four processors, memory and I/O bridge connections [19,20]. A very common bus based cache protocol found in bus based systems is a four state protocol referred to as “MESI” [19]. The general concepts of a bus based MESI system are provided in the following subsection. A single bus based topology has the added benefit in that all processors are equidistant to the memory. These systems are referred to as having a UMA (Uniformed Memory Access) architecture. This simply implies that no processor has an unfair advantage in accessing any location in memory. The system characteristics of this type of architecture are as follows:

- All processors observe all accesses in the order issued (and completed).
- All processors are the same distance (number of clocks) from the memory of the system.

To build larger SMP systems, different interconnect networks have been used, such as rings, mesh, etc. SMP Systems which are not based on a single bus topology are referred to as NUMA (Non-Uniformed Memory Access). Systems based on this architecture have the following characteristics:

- All processors do not observe all accesses in the order issued (and completed).
- Processors do not have the same access time to a given memory location.

NUMA based systems are becoming an accepted architecture in today's SMP systems. Several systems based on a NUMA architecture are:

- DASH/FLASH from Stanford [46,47].
- Alewife from MIT [37].
- NUMAchine from the University of Toronto [50].
- Origin 2000 from SGI.
- S3MP from Sun.
- Exemplar from Convex (now HP) [41].
- NUMA-Q from Sequent Computer System Inc. [28].

2.1 Concepts of Cache Coherence Protocols for a Bus Based System

There are many types of cache coherence protocols. A very common type of protocol is based on a simple four-state protocol. The states are commonly referred to as **Modified**, **Exclusive**, **Shared**, and **Invalid**, hence the name "MESI" protocol. It should be noted that there are many variations of this protocol. An example of a MESI protocol is provided in figure 2.2.

Common definitions of the MESI states are as follows:

- **Modified** – The cache holds a modified version of the line (i.e. an agent has written a portion or all of the line). In this state, the cache holding the modified line has the **ONLY VALID** copy of the data.
- **Exclusive** – The cache holds a valid copy of the line and it is the only copy of the line. Memory in this case also maintains a valid copy of the line.
- **Shared** – The cache holds one of the valid copies of the line. Due to cache evictions, 0, 1 or more caches can hold a valid copy of the line.

Memory in this case also maintains a valid copy of the line.

- **Invalid** – This particular cache entry does not contain valid copy of data.

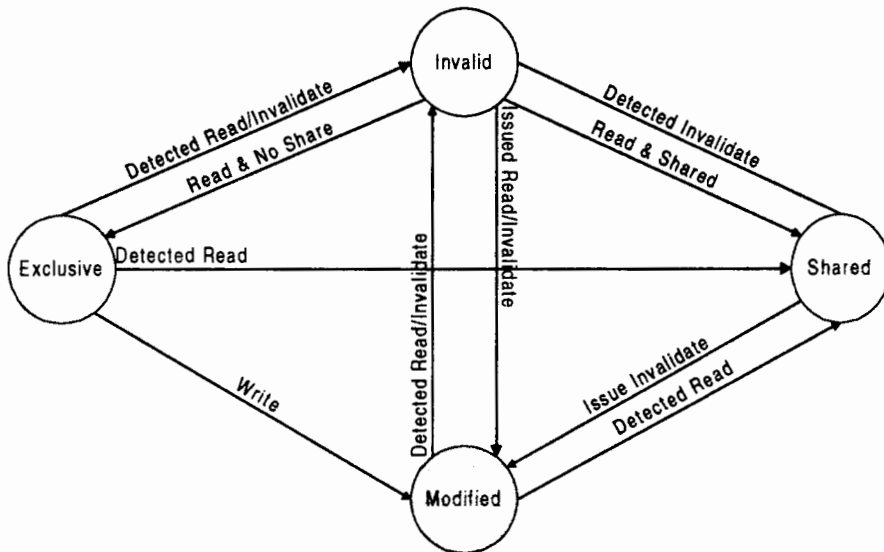


Figure 2.2 - An Example of an MESI Cache Protocol

The primary reason for caching is to remove the memory “bottleneck” issues. This is accomplished by constructing the cache memory out of “faster” memory devices. By definition the size of the cache is significantly smaller than the total size of the memory subsystem(s). When the cache is full, before new line requests can be installed into the cache, some lines currently held in the cache must be evicted (or rolled back into the memory subsystem). To this end, the cache protocol must also handle the eviction scenarios without the loss of data. The following are the required actions for each possible MESI states:

- **Modified** – Valid data must be written back prior to installing a different

line in this location in the cache. Also during the eviction process, any copies of the cache line in caches (or processors) below this level must be purged.

- **Exclusive** – Mark the line invalid and insure any caches or processors below this level invalidate their copy of the line.
- **Shared** – Mark the line invalid and insure any caches or processors below this level invalidate their copy of the line.
- **Invalid** – This location is currently available to install a new entry into the cache.

2.2 Review of NUMA Terminology

The following is a list of commonly used terms in describing systems that are based on a Non-Uniformed Memory Access architecture.

NUMA -(Non Uniformed Memory Access) refers to systems where the memory accesses happen non-uniformly due to the fact that memory is dispersed throughout the system. The access time is determined by its relative location compared to the accessing processor. A NUMA based system can be cache coherent, message based or both.

CC-NUMA - (Cache-Coherent NUMA) Cache coherence computer architecture for a large scale distributed shared-memory system based on a directory-based protocol. The directory scheme can either be a centrally located (as in the Stanford Flash architecture [11]) or distributed (as in a SCI based architecture [2]).

List - The mechanism to track which nodes of a system have a particular cache line. A List exists for every cache line that at any given time is being shared by a processor.

- **Dirty List** - A List of nodes that share a line in which the memory (or home) node does not hold a valid copy.
- **Fresh List** - A List of nodes that share a line in which the memory (or home) node contains a valid copy.
- **Head of the List** – The first remote node on a list is considered the “Head” of the list. In SCI all nodes prepend to the list at the “Head” position. Also the only node which can purge a sharing list is the head of the list.

For SCI, the list is a very dynamic structure. New nodes can be prepending to the list as old nodes are getting off the list.

Node or Quad – For the NUMA-Q system a node (or quad) is a four processor bus based SMP module and the basic building block of the NUMA-Q system. It is based on the Intel Pentium Pro or XEON processor. The term quad is a Sequent Computer System Inc. term. Most other papers and systems refer to the processor/memory building block as a node. [22]

SCI - (Scalable Coherent Interface) is a IEEE Specification (1596-1992) that defines a directory based cache protocol that also defines the physical interface, packet formats, as well as coherence protocol. SCI is based on a

distributed directory based scheme. This scheme is based on a doubly linked list where (for each cache line) each node contains the state of the cache line, a backward and forward pointer. [2] SCI is not only a cache coherent protocol but also defines a physical and data layer. SCI defines a 1Gbyte/sec interface based on a point to point interconnect. The physical interface is based on a Low Voltage Differential Signaling (LVDS). The interface is eighteen bits wide (sixteen data, with two bits signaling). SCI packets contain header information (destination ID, source ID, transaction number, and command), data, and CRC. The basic SCI protocol is based on a simple request/response transaction concept.

SCI uses a split transaction protocol. Therefore, when a node sends a request, it will wait for the packet to traverse the network to the target node. The target node will handle the request and send a response to the requesting node. When the requesting node receives the response it will take the proper actions and the transaction is complete [34].

“To avoid deadlock, the SCI protocol is based on certain fundamental premises.

- SCI Requests have absolutely no circular dependencies. In particular, no SCI request is dependent on the completion of a dependent request for its completion.
- SCI requests cannot be issued without first guaranteeing space for the response in the requesting node.
- Every device on the SCI ring must contain a bypass FIFO large

enough to store the largest packet the device is capable of transmitting.

- Packets in the output queue are sent when the bypass FIFO is empty and the node's flow-control mechanism permits it. Another packet (or packets) may arrive on the input link while an output packet is being sent. If the packet is not addressed to this node, the bypass FIFO holds these incoming packets for delayed transmission until the output queue packet has been sent, the output queue is empty or the bypass filter is in danger of overflowing.
- When a send packet is emitted, the packet is saved in the output queue until a confirming echo packet is received. There are two types of echo packets (accepted and rejected). If the echo packet was rejected, the original request packet is reissued. If the responder has space to save the request packet, it issues the "echo accepted" packet. At this time the requester passes the responsibility of the packet to the responder.
- To avoid deadlock there are distinct input request and response queues (the same is true for the output side). Forward progress is ensured because at least one entry is always available for holding input request, input response, output request, and output response packets." [2], [52]

2.3 High Level Description of the NUMA-Q

As stated earlier, the "NUMA-Q" is a Cache Coherent Non-Uniform Memory Access (CC-NUMA) multiprocessor architecture of Sequent Computer Systems Inc.. It's basic building blocks are:

- Off-the-shelf 4 processor SMP module.
- Directory Based Cache Protocol.
- Programmable hierarchical cache controller.

Architecturally, NUMA-Q uses Intel standard four-processor chipset to build a bus based SMP system and uses this as it's basic building block to build larger systems. In addition to the processors, memory, and I/O busses in the SMP module, all nodes have a system interconnect with a third level remote cache. The system interconnect is a hierarchical connection which allows the SMP module to be a sub-component of a larger SMP system. At the SMP module level the processors maintain cache coherency based on MESI protocol. Between SMP modules, SCI (a distributed directory based cache protocol) is used to maintain cache coherency. The SCI standard defines a cache protocol based on a distributed doubly linked list. The "list" for any particular cache line is the list of all nodes that contain a valid copy of the line. The node that actually possesses the physical memory for a particular address is referred to as the "Home" node. The home node maintains a pointer to the "Head" of the sharing list and a state of the cache line. The Head of the list maintains two pointers and the state of the line. The backward pointer points up the list "back" towards the home and the forward pointer points down the list toward the tail of the list. The state of the line reflects whether the line is "Fresh" or "Dirty" and the position in the sharing list (i.e. a node could be a "Only", "Head", "Mid", or "Tail"). It should be noted that a sharing list has only one "Head" and one "Tail", but can have an arbitrary number of "Mid's" [2].

The node's remote cache controller is based around the concept of a programmable multi-threaded protocol engine. This protocol engine contains global and thread specific registers and maintains cache coherence by executing a set of RAM based instructions. To experiment with a new system level cache protocol, all that has to be done is to change the RAM based instructions. A specialized table driven assembler has been developed to aid in developing new cache protocols. The instruction set for the protocol engine resembles in structure and complexity the instruction set of an 8051. The realization of the new cache extension relies primarily on the protocol engine's ability to execute a series of instructions that will emulate the checks and procedures of the extensions.

3. Detailed System Architectural Description

The project will use the "NUMA-Q" system to develop a new hybrid of the SCI cache coherency protocol. The basic architecture of the NUMA-Q uses as its fundamental building block a "4x Pentium Pro Processor Module", which is referred to as a node. These modules are then interconnected via a hierarchical interconnect based on the SCI physical layer.

3.1 Node Module Description

The node's design is based on the Intel P6 system architecture description. The node design integrates CPU's (Up to 4 Pentium Pro Processors), APIC Bus (a distributed interrupt interface), memory subsystem, and two I/O bridges. The node's basic internal interconnect is a multiprocessor bus. This bus is a transaction oriented bus design that has the processors (with their L1/L2 caches), memory, and I/O Bridges directly connected to it. Refer to the figure 3.1 for a simple block diagram of the node. Also connected to this bus is the subsystem that supports the hierarchical interconnect. This subsystem is referred to as the "IQ-Link". Note the basic building block (or subsystem) is a complete "four processor" SMP system.

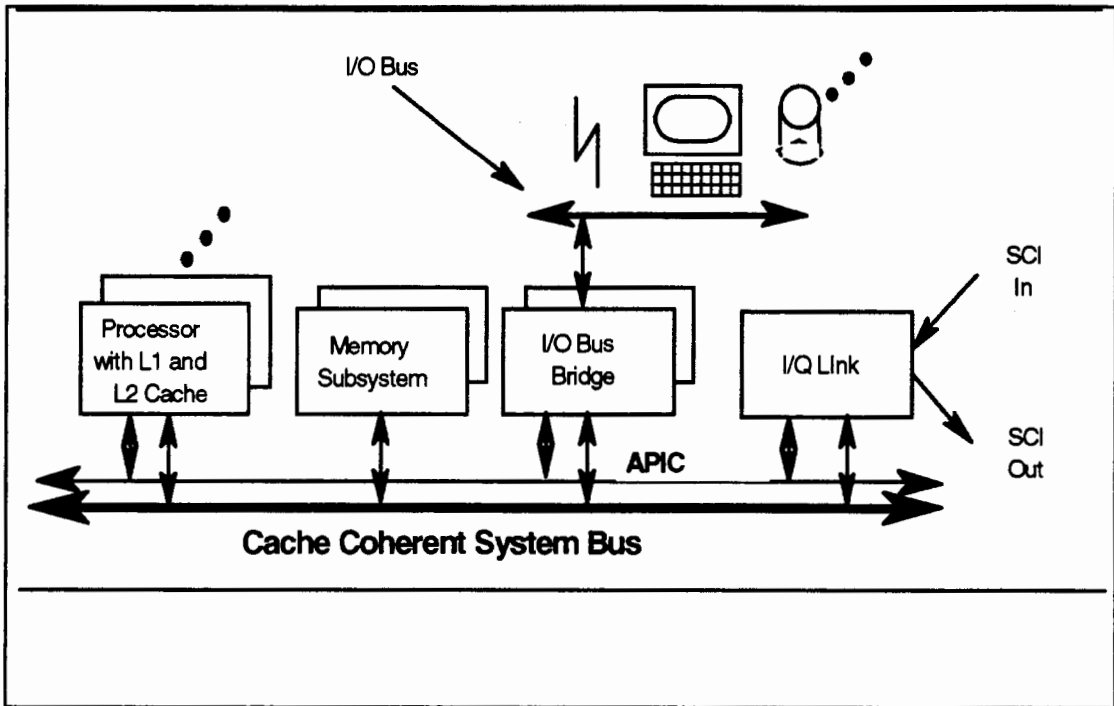


Figure 3.1 - Simple Node Block Diagram

The system bus, as stated earlier, is based on a transaction oriented protocol. The bus provides enough information to implement a MESI cache coherency protocol. The IQ-Link monitors accesses on the cache coherent bus of the node. If an access is to a cache line that is naturally resident to this node, a look-up is issued to the node's memory tags. If the type of access can be supplied locally (i.e. the line is currently resident on this node) then the access is allowed to proceed. However, if this is not the case, then the interface would intervene via the standard MESI mechanisms. It would then issue the appropriate requests to retrieve the line and/or invalidate other copies of the line. The access for that particular address would then be allowed to continue (this assumes that all system cache coherency requirements have been fulfilled). In the standard MESI protocol the subsystem acts as a cache agent

for the requester.

Cacheable accesses that do not fall in this node's memory region, instead of being looked-up in the memory tags, are looked up in the remote cache tags (or L3 tags). If the request missed in the cache or the state of the line does not support the type of access, then the IQ-Link would build the appropriate requests to get the line at its node in the correct state to fulfill the bus request. A simple block diagram of the IQ-Link is contained in the following figure.

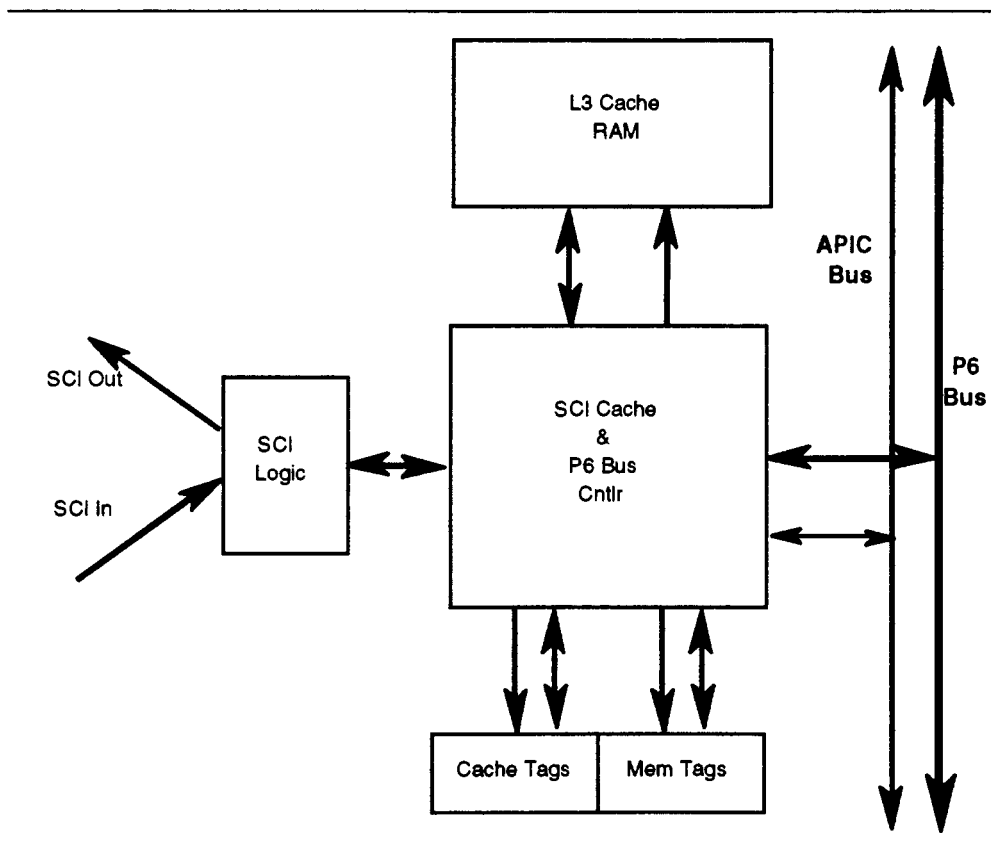


Figure 3.2 - High Level Block Diagram of the IQ-Link

As stated earlier, the IQ-Link ASIC that provides the translation between the

MESI and SCI protocols is based on a programmable protocol engine. The protocol engine converts between the two protocols via the instructions downloaded into its instruction RAM. This feature of the design provides for the ability to change the cache protocol of the hierarchical interconnect. The basic instruction set of the protocol engine consists of register to register moves (with mask and rotate), move immediate values, compare, logical operators (AND, OR, Negate, XOR), addition, and subtraction.

3.2 Interconnect Description

The interconnect between nodes is based on the SCI physical interface definition. The SCI physical interface is based on a high bandwidth (1GByte/sec) point to point connections. With this definition simple rings or more complex networks can be constructed. The physical interconnect is based on a two byte wide connection based on an LVDS voltage swings. In addition to the physical definition, SCI provides a distributed directory-based cache coherence protocol.

The protocol uses transaction oriented request and response mechanisms founded on a well defined packet format. The specification has the following characteristics:

1. Defines a cache line size of 64 bytes.
2. Defines the physical point to point connection (the connection is 2 bytes wide, uses Low Voltage differential Signaling).
3. Defines Non-Coherent Memory Transfers.

4. Defines Coherent Memory Transfers.

The protocol is based on a simple head to tail pointer algorithm where an agent (Node in this case) attaches to the head of the list when accessing a cache line. Only the "Head of the List" has the permission to modify the line. If the "Head" intends to modify a line, it first must invalidate the line at each node on the list.

The SCI base protocol is represented in the following two figures. Figure 3.3 represents the SCI memory protocol. Memory state diagram consists of four states:

Home - The only copy of the line exists on the Node that the physical memory is resident (i.e. there is no sharing list).

Fresh - The home Node does have a valid copy of the line, but there is a sharing list (i.e. there are copies of the line on other Nodes).

Gone - The home Node does not have a valid copy of the line. The home Node provides a pointer to the Head of the List, which does (or will) have a valid copy of the line.

Wash - The line is in the process of being updated to the Fresh State.

Figure 3.4 shows the SCI base cache protocol. It should be noted that this figure is only the base protocol. It does not contain the locking options or pair-wise sharing options of the SCI protocol.

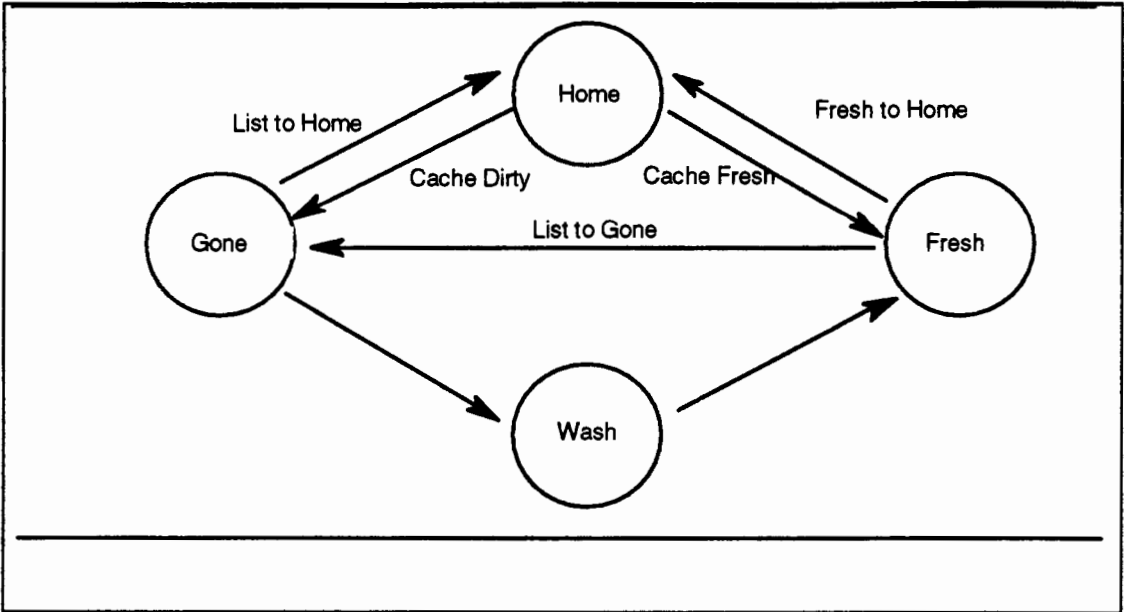


Figure 3.3 - SCI Memory State Diagram

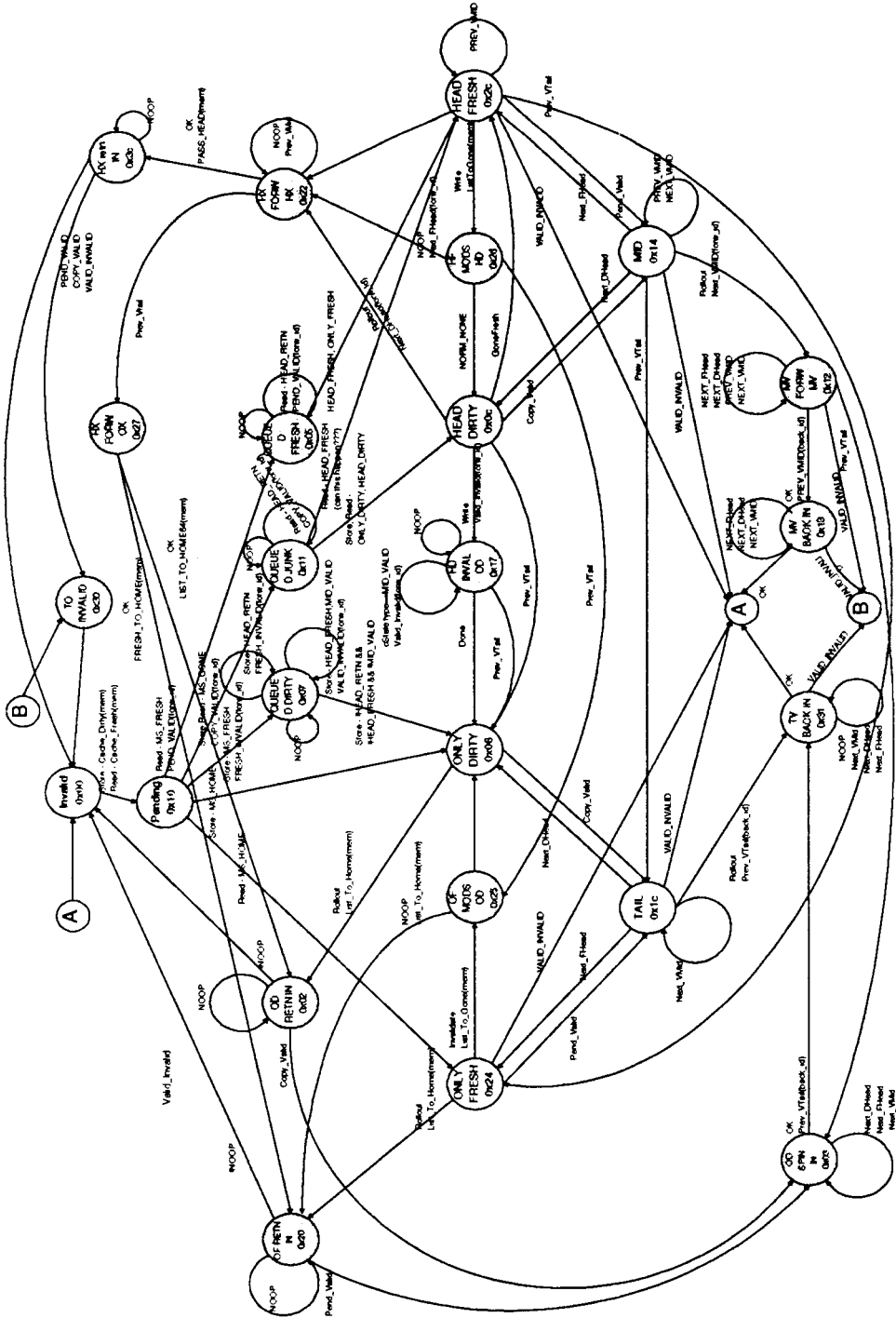


Figure 3.4 - SCI Cache State Diagram

4. Guaranteed Forward Progress Extension

Currently in the NUMA-Q system, accesses to memory are not fair. This is due to several issues:

- The bus interface chip does not allow invalidates for local lines to be converted to read invalidates. For this reason local invalidate requests can continually prevent remote nodes from prepending to the sharing list. The result is the local processors have an advantage in accessing local lines. This is not the case for invalidates to remote lines. It should be pointed out that this particular attribute is specific to the NUMA-Q system.
- SCI requires the “home” node to respond with data if it contains a valid copy of the line [2]. Combined with the first issue this creates the possibility that a remote node access to a particular line can be delayed.
- SCI limits the memory states of a line to a 2-bit field [2]. This 2-bit field limits all implementation to a maximum state machine of four states.
- An SCI network is a completely unordered network topology.

Due to these restrictions, the current protocol for the NUMA-Q system does not allow for the same cache controller to issue multiple requests down to a bus for a local line. This is due predominately to the first and fourth issues previously stated. There are several things that should be noted. This restriction causes several issues:

Requests for “hot” lines can be “NOOP’ed” at the local node. NACK’ing requests to prepend to a sharing list produces an environment that could potentially prevent a processor from ever gaining access to a cache line. Processing NOOP responses and reissuing the initial requests to the local node increases processor latency. SCI and the cache controller bandwidth are consumed generating and processing these NOOP responses.

For accesses to remote lines the hierarchical cache controller is the final arbiter on how the requests for a given cache line are serviced. In a remote node, the hierarchical cache controller can turn an invalidate request into a read-invalidate request. Once a node prepends to an SCI sharing list, the requests are serviced in the order that the nodes had prepended to the list. The basic premise of the guaranteed forward progress extension is to exploit the natural serialization process of the SCI sharing list.

4.1 Reading of a Home Line

When a remote node issues a read request for a line in the “home” state, there is currently no sharing list for that particular line. The flow of events for a remote line to read a line with a memory state of “Home” is as follows:

- Cacheable read requests that are made by a processor (or I/O device) of the remote node cannot be serviced by the third level cache.
- The bus Interface chip issues a cache read request to its hierarchical cache controller.
- Hierarchical cache controller issues an SCI “Cache_Read” Request to the home node’s cache controller.
- After the local state of the line is checked, the “Home” node’s cache controller issues Local read Request down the bus.
- Read request is serviced by either memory or a local processor of the “Home” node.
- Read Response, at the local cache controller, causes an SCI response packet to be issued to the requester. Also at this time, the local memory state and pointer are updated to reflect the change in state and the new “head”.
- Remote node receives the response, issues the data response to the bus and updates its state and pointer information in its remote cache.

This flow of events is represented in figure 4.1. Figure 4.1 shows the request as it is issued on the remote node, the SCI packet built and sent, the steps taken on the local node, and finally the response being sent back through the remote cache controller to its bus.

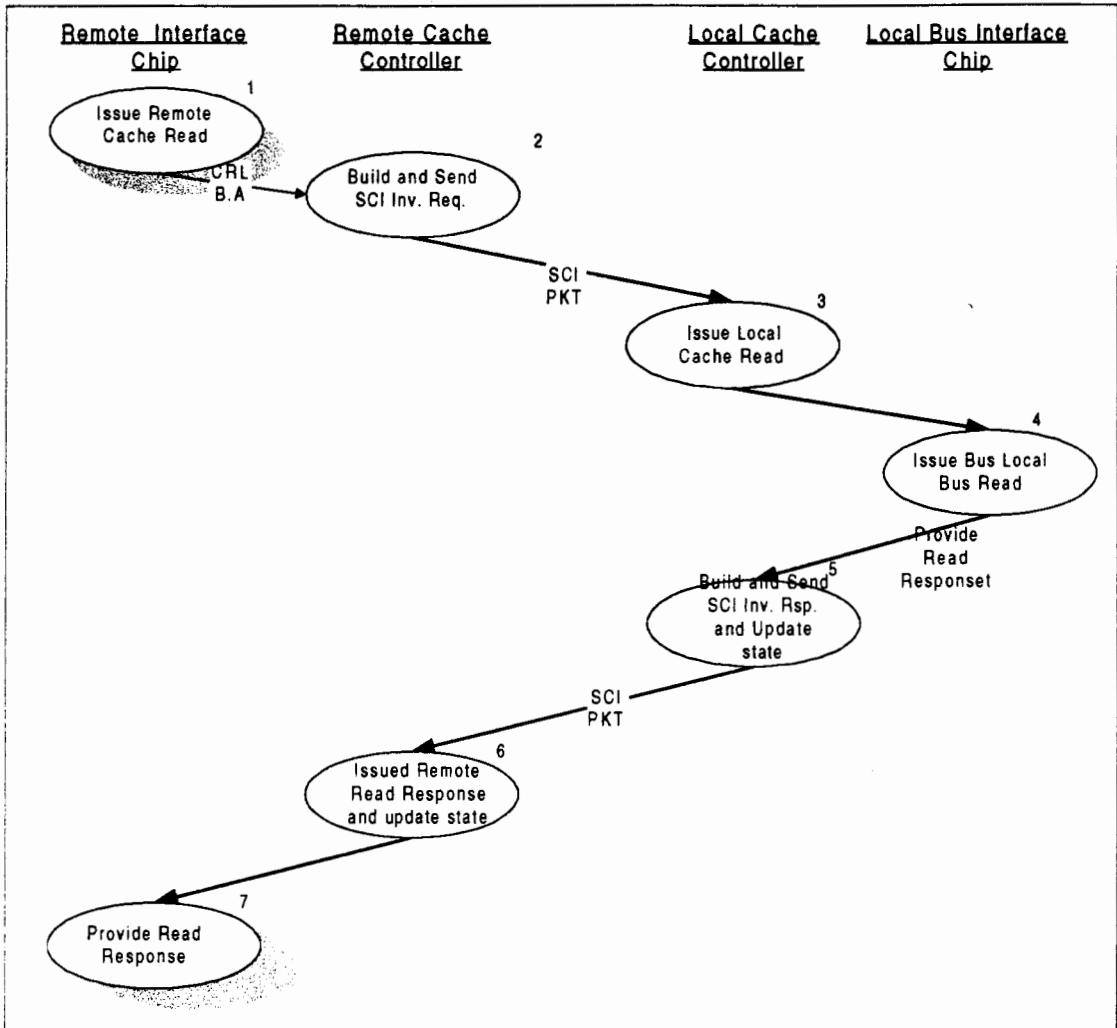


Figure 4.1 – Reading of a Home Line

4.2 Reading of a Fresh Line

In reading a line with a memory state of “Fresh”, everything is the same as a line that is home until the last step of the process. The remote node checks the state and sees that the line is “Fresh”. Prior to issuing the response to the

bus, the remote node must notify the “old” head to change its state and update its list pointer information.

This means that if a list already exists for a cache line, at minimum two SCI requests must be generated and surfaced prior to the response being issued on the bus of the remote node. The first is to the “home” node. It will return the data, state, and pointer to the “Old” head. (Also at this time, the local state and pointers are updated to reflect the completion of this request.) The second request (which is issued after the response from the home node) is issued to the “Old” head. The request issued is a Pend_Valid command. This command notifies the “old” head to update its backward pointer and transition its state to reflect its new position in the sharing list.

4.3 Reading of a Gone Line

A memory state of “Gone” implies that a sharing list for this line does exist. Also, the home node does not currently maintain a valid copy of the cache line. In reading a line which is marked “Gone” at the home node, everything is the same as a line that is home until the last step of the process, except that no request is issued to the local bus. Pointer information is updated and a “data less” response packet is issued back to the requesting node.

This means that if a gone list already exists for a cache line, at minimum two SCI requests must be generated and serviced prior to the response being issued on the bus of the requesting node. The first SCI request is to the “home” node. It will return the data, state, and pointer to the “Old” head. (Also at this time, the local state and pointers are updated to reflect the completion of this request.) The second request (which is issued after the response from the home node) is issued to the “Old” head. The request issued is a Copy_Valid command. This command notifies the “old” head to update its backward pointer and transition its state to reflect its new position in the sharing list and also provide a valid copy of cache line.

4.4 Description of the Guaranteed Forward Progress Extension

In reading the previous subsections (4.2 and 4.3), note the similarities of servicing a read request of a fresh and gone cache line. In both cases:

- Two SCI requests are issued. One to the home node and the other to the “old” head.
- A read request is issued to a processor bus (either at the home node or the “old” head).

Our extension is just a simplification of the SCI protocol. This extension first eliminates the need of the local node to issue read requests down to the bus for a line that is in the state of “Fresh”. This step has three advantages:

- It distributes the bus consumption of shared lines across all the sharing nodes, this is in comparison with the standard protocol which requires the local node to provide all the resources to supply a copy of the cache line to the requesting node.
- It eliminates the possibility of a read request being superceded by an invalidate request coming up from the bus (i.e. reduces the number of SCI NOOP response packets). Note for remote accesses, the cache controller of the node is the serialization point.
- It simplifies the SCI protocol by making the flow for reading a “fresh” and “gone” line the same.

In addition to these advantages, this extension eliminates the need of the SCI Pend_Valid Command.

4.5 Reasoning for the Guaranteed Forward Progress Extension

Figure 4.2 represents the current flow of events with the “standard” SCI cache protocol. Note there are issues with the flow of events through the remote read process. The first is that the local node can “NACK” a request. Since a remote request can be “NACK’ed” on actively contested cache lines, some nodes can be denied access to this line. The resulting situation from this is either live-lock (in the worst case scenario) or some node’s processors exhibit a lower processor utilization compared with other nodes in the system.

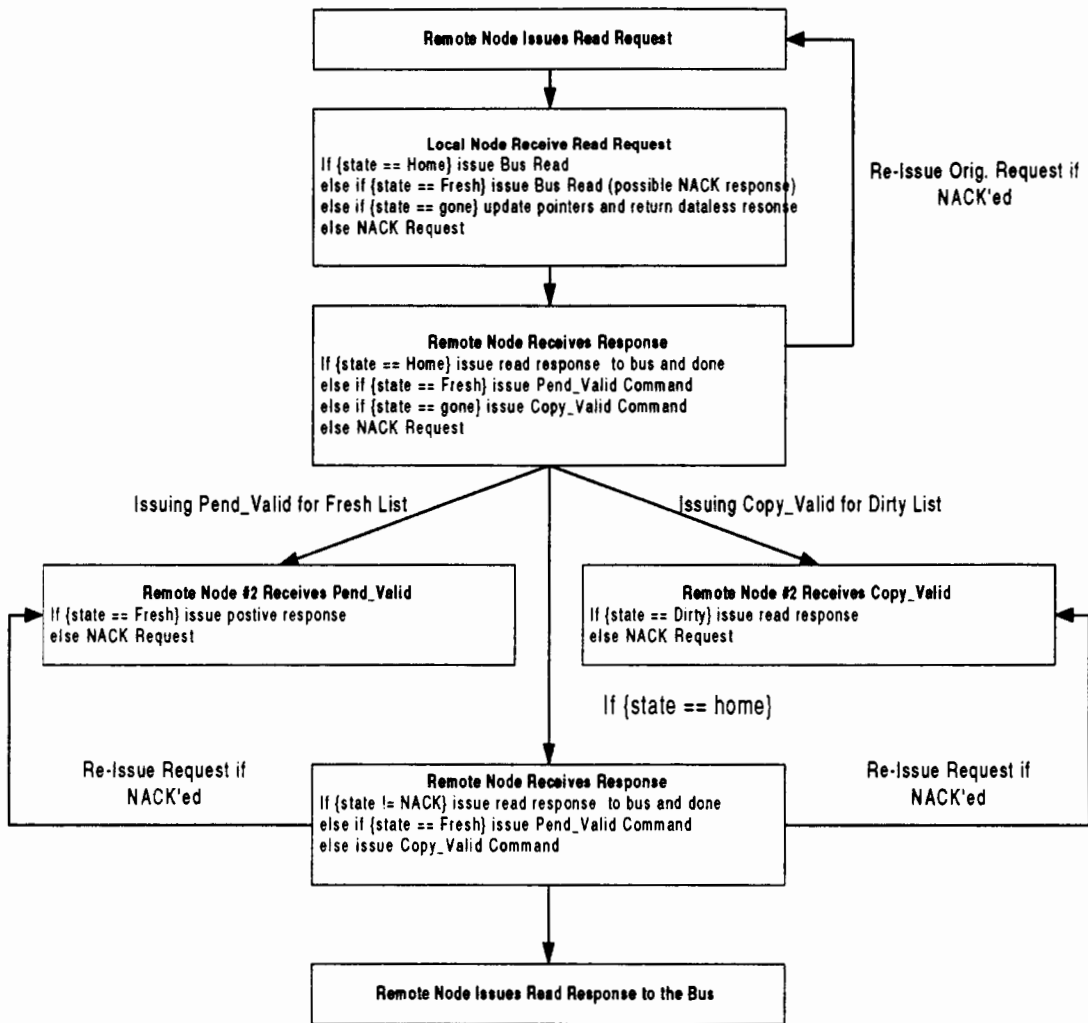


Figure 4.2 - SCI Cacheable Read Request Flow

In contrast, compare figures 4.3 and 4.2. Figure 4.3 represents the guaranteed forward progress extension. In the guaranteed forward progress approach the possibility of the “local” node to NACK the response is removed. This is done by eliminating the NACK’ing scenario. The scenario that is avoided is as follows:

- Local Cache Line is held in the state of “Fresh”.

- A Request to prepend to this list (the Cache_Read Command) is received and processed just to a request from the local bus to invalidate the list.

Since the bus will never issue a request to invalidate a line that is held exclusively, and this is the only instance where a read request is issued to the bus, the result is that the NACK condition is removed. The resulting scenario is that read requests are allowed to fairly serialize as the sharing list grows. Also, each node as it becomes the “old head” consumes a small piece of its local bandwidth to supply the cache line to the “new” head allowing for the load of supplying data to be shared equally among all the nodes on the sharing list. This is in comparison with the standard SCI approach where the “Home” node must commit the resources to supply the cache line to all the requestors of a shared “Fresh” list.

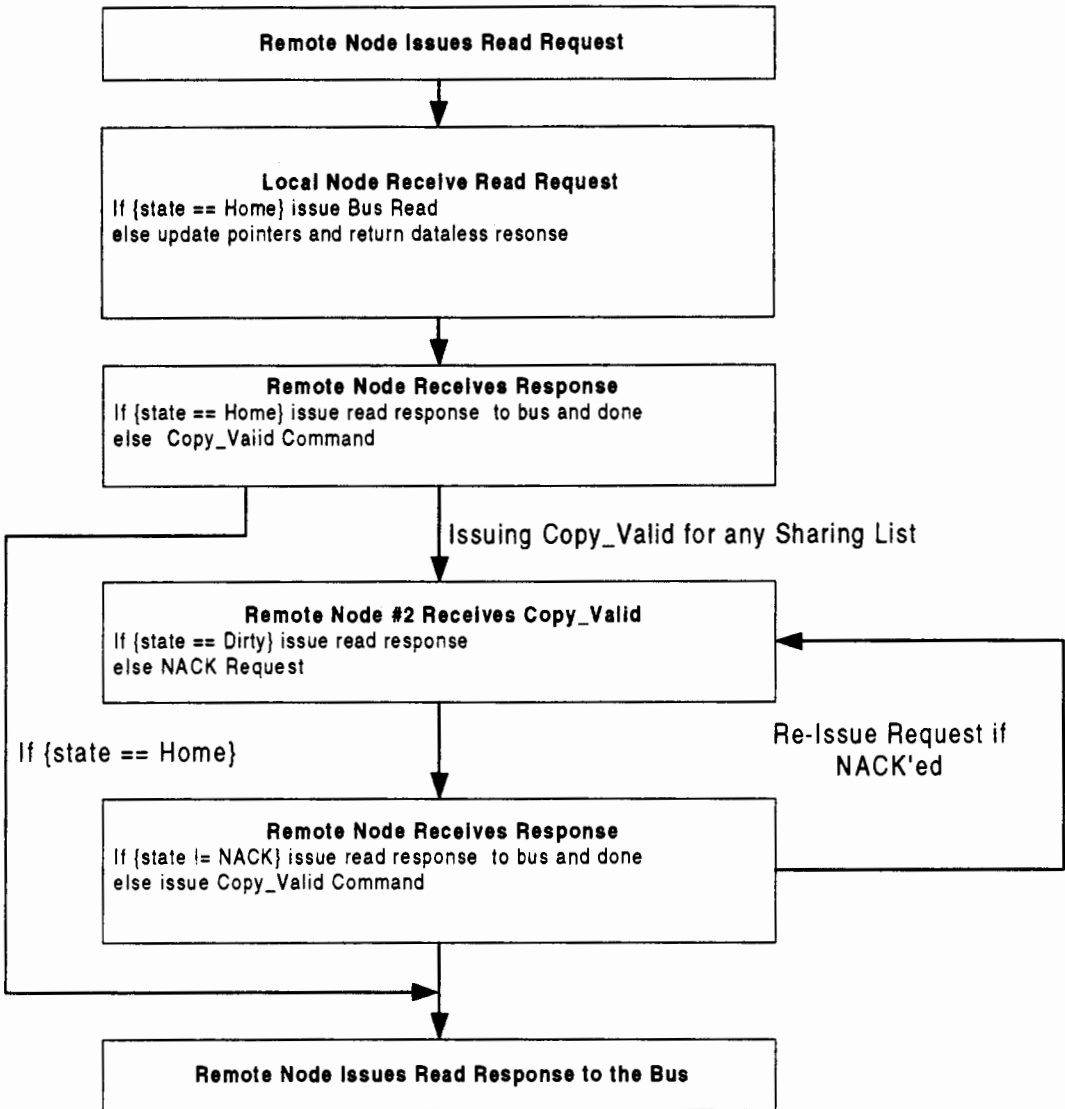


Figure 4.3 – Guaranteed Forward Progress Cacheable Read Request

Flow

Another simplification in figure 4.2 is the elimination of the SCI Pend_Valid command, simplifying the number of SCI states and commands to check. The obvious drawback concerning this extension is quantifying the performance gain. There was no measured performance difference between the two

protocols for a system under a “reasonable” load. The read latency between the two protocols was the same. Again the primary benefits of the guarantee forward progress extension are the following:

- The elimination of the possibility of receiving a “NACK” from the Local node in the prepending to the sharing list. Note, once on a sharing list, the list’s order of nodes predetermines how the cache line is manipulated. Also the act of successfully “prepending” to the list guarantees that the requesting processor has access to the cache line.
- The simplification of the SCI protocol, by making the “prepending” to a pre-existing sharing list the same whether the list is a Dirty or Fresh.

4.6 Comparison between the Standard and the New Protocol

The standard and new protocols differ in two areas when supplying read data of a cache line. These are at the processing of the initial request at the local node and in how the requesting node communicates with the “old” head of a SCI sharing list.

4.6.1 Communication with the Local Node

The flow of events for the standard and new cache protocol are identical until

the SCI read request is received at the local node. Per the standard SCI protocol if the local node has a valid copy of a cache line it must provide a copy of line. Due to this requirement in the case of a “Fresh” list, if a processor on the local node issues an invalidate request for this line, the read request will be NACK’ed prior to the requestor being prepended to the sharing list. In NACK’ing the request the potential now exists that the condition to fulfill the read request might never exist creating the potential dead-lock scenario. With the new protocol requests to the local bus are issued only if the line is in the “home” state. By definition the local node’s bus would never issue a request to invalidate other nodes on an SCI sharing list because the line is currently not shared, thus avoiding the possibility of a read request of a line in the “Fresh” state colliding with a local bus invalidate request.

4.6.2 Communication with the “Old” Head of a SCI Sharing List

In both the standard and new protocols the new head of the list must notify the “Old” head of a sharing list that it must change its backward pointer and cache state to reflect its new position in the sharing list. For the standard protocol two different SCI commands are used to communicate with the “old” head.

These commands are

- **Pend_Valid Command** for prepending to a “Fresh” list. The response for this command is a data-less response.
- **Copy_Valid Command** for prepending to a “Dirty list. The response

for this command commands a valid copy of the cache line.

In the case of the new protocol, the need for the Pend_Valid Command has been completely eliminated. With the new protocol only the Copy_Valid command is used to prepend to a pre-existing sharing list.

5. Reduce List Invalidation Time Extension

The “Reduce List Invalidation Time” extension’s primary goal is to reduce the time to invalidate a sharing list while residing in the basic constructs of the SCI invalidation scheme. The enhancement in the invalidation sequence is to attempt to parallelize the bus invalidation sequence with the “SCI acknowledgment”. The “SCI acknowledgement” can be the issuing of the response back to the initiator or it could be the forwarding of the invalidation request to the next node on the sharing list.

Any time a cache controller acknowledges an “event” early, the cache controller must assume the responsibility of maintaining the ordering of events on this particular node. The cache controller, in order to maintain a processor consistency model, must prevent the following situations from occurring:

- A processor’s remote read request to complete prior to the completion of the currently posted invalidates on this node. This is commonly referred to as the “read passing a write” scenario.
- A processor’s writes to be observed by any other processor in the system out of the order issued. This issue is referred to as “a write passing a write”. With processors using MESI based bus interface, writes are observed by read completion to the same address. Writes typically happen in the L1 or L2 cache of the processor.

It should be noted that the order of writes from any single processor must be observed by all other processors of the system in the order issued to maintain a processor consistency model. Also the ordering of writes to the same cache line by multiple processors is done in the order that the requests to prepend to

the SCI sharing list were processed at the home node. This implies that the second, third, etc. writes to the same cache line by different processors require those writes to become a read-modify-write sequence. The serialization point of the home node ensures that the system obeys the processor consistency model. Note that in the processor consistency model, the order of writes from multiple processors to different cache lines does not have to be maintained throughout the system. The following table provides all combinations of a distant processor reading two unique cache lines that are in the process of being written. In this example, the order of writes is “A” completes followed by “B”.

Line “A”	Line “B”	Comment
Old Data	Old Data	Distant Processor reads old values of both lines.
New Data	Old Data	Distant Processor observed write of A but not B’s.
New Data	New Data	Distant Processor observed write of A then B’s.
Old Data	New Data	Distant Processor observed the write of B before A’s.

Table 5.1- Possible Cacheable Ordering Scenarios

Of the four possible scenarios reflected by table 5.1, the first three are acceptable scenarios to happen and have the system maintain a processor consistency model. For this extension to maintain a processor consistency model the cache controller, when it acknowledges the invalidate request, “early” it must prevent the last entry of table 5.1 from occurring.

The scenario that must be prevented in table 5.1 is the following:

Processor 1 is writing some “datum” held in Line “A” and then

writes the “completion signal” that is in line “B”. Other processors of the system are spinning reading line “B” waiting for the “completion signal. The act of writing B is the signal to all other processors that the “data” is valid. If the write of B passes the write of A for any reason then the system no longer maintains a processor consistency model.

It should be noted that the other processors in the system observe the write by reading the cache line. If a processor reads line “B” and the line is not in the L1/L2 cache, a bus access is issued to install the line. It is the act of reading the updated line, which conveys the occurrence of the write. It was stated earlier that a “distant” processor observes the order of writes by the completion of the reads issued. It is the read responses from the node that convey the occurrence of the write. Therefore, the read responses provide the mechanism for other processors in the system to observe the ordering of writes from any particular processor. In addition to read responses, writes can be conveyed by two other mechanisms.

- The first mechanism is the “interrupt”. The “interrupt” mechanism is a very commonly used “completion” signal, which can notify all other processors in a system that the “datum” is valid.
- The second mechanism is the “write-back”. Suppose the write of “B” required no invalidate because the processor held the exclusive copy of line B. Also suppose, just after the completion of the write of “B”, a capacity miss occurred and line “B” was selected to be evicted from the processor’s cache. The processor would then just write-back “B” to memory. Note the memory might not be located

local to the writing processor.

The fast invalidation methodology is based on a posting methodology for all invalidates at a remote node. When a remote node receives an invalidate request from the SCI network, it would immediately issue the SCI response. Every time an invalidate request is acknowledged early (i.e. issuing of the completion response or forwarding the invalidate request), a bit mask is set identifying that an invalidate is currently in progress. When an invalidate request completes, the pending invalidate bit is cleared. When a read response is received from the SCI network, the currently pending invalidate register is read and copied to a unique register, specifically for this read response. The currently set bits of the private copy represent the writes from other processors that must be completed to prevent the cases:

- Read Passing a Write Scenario
- Write Passing a Write Scenario

As invalidates complete, the pending invalidate bits are cleared. When all posted invalidate bits are cleared for a particular read response, it is issued to the bus fulfilling the read request. Note there is a “unique” pending invalidate register for all read responses. The bit vector of the pending invalidates is captured when the read response is received. When all bits have been retired the read completes. This methodology provides a new extension to the SCI protocol to aid in decreasing the time to invalidate a sharing list. The decrease is realized by the parallelization of the invalidation of the local copy of the cache line and the flight time of the SCI response packet.

This very same mechanism is used also for write-backs and interrupts. Again these accesses must be delayed also because they can potentially contain the “Completion Signal” as Line “B” does in table 5.1. The scenario that is being prevented in the case of the Write-back is as follows:

The processor, immediately following the write of B, experiences a capacity miss and line “B” is selected to be evicted. Since the line is modified, it must be written back to into memory location. If the home of the memory location is on not on the same node as the processor, the write-back is issued over the SCI network. If “B” gets installed in its home node and a local processor of that node reads the local copy of B prior to the completion of the invalidation of its copy of line “A” then “a write passed a write”. Therefore, to prevent this situation from occurring, a write-back must be delayed until the completion of all currently active posted invalidate requests are completed.

5.1 Comparison of the Two Invalidation Methods

Portrayed in figure 5.1 are the steps taken with the standard SCI protocol to purge a sharing list. In the figure the length of the sharing list is two.

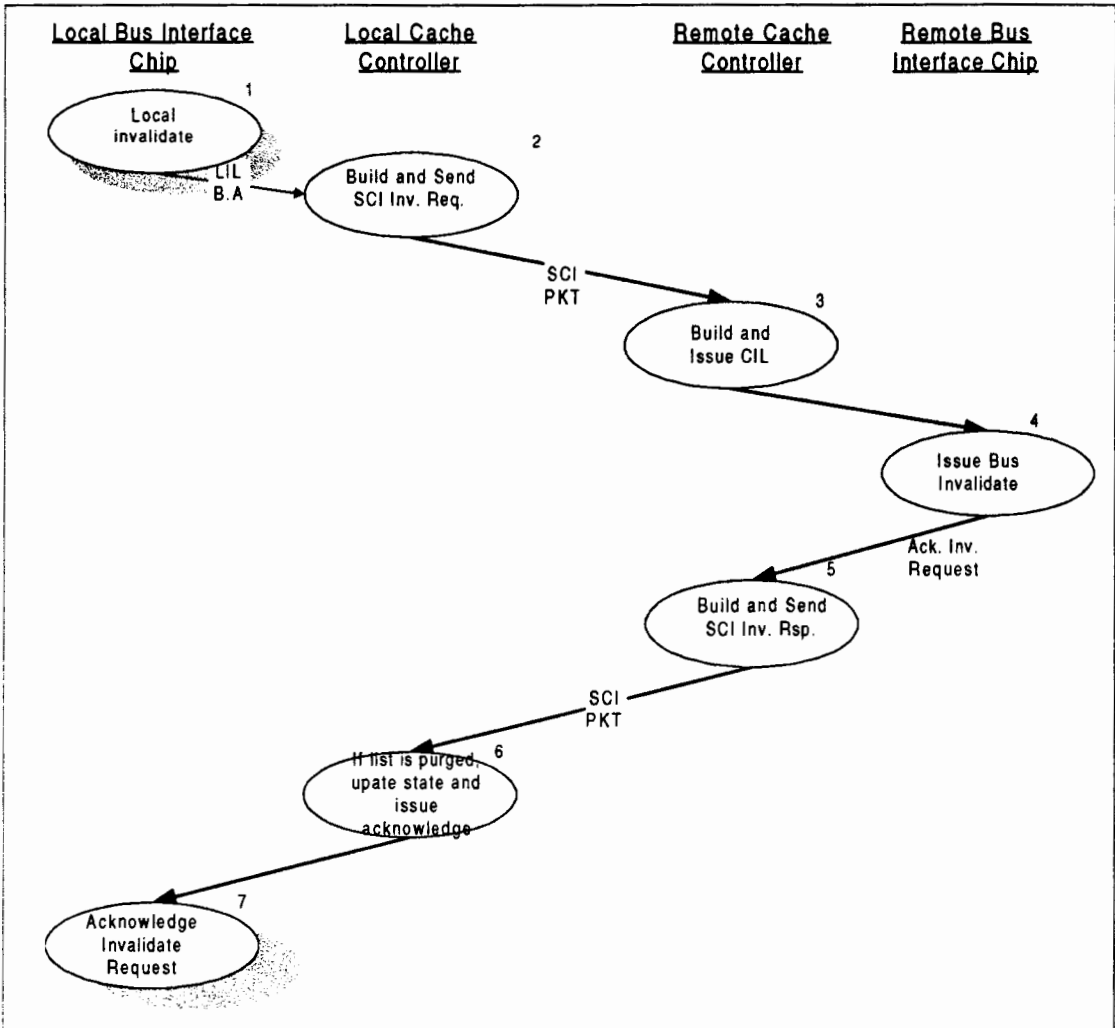


Figure 5.1 Standard SCI Invalidation Flow of Events

In figure 5.1 the bubbles represent the steps of the invalidation sequence.

The steps are as follows:

1. Bus interface issues to the cache controller an invalidate request.
2. Cache Controller looks up the line, a SCI Invalidate Request Packet is built and is targeted to the node stored as the “Forward Pointer”.

3. Remote Cache Controller receives Invalidate Request, checks the state of the line and in most cases issues invalidate to the bus.
4. Remote Bus Controller issues request on node's system bus and when complete acknowledges invalidate request.
5. Remote Cache Controller updates state to invalid and issues SCI response.
6. Local Cache Controller processes response, detects that the SCI list is completely purged, updates its state and issues acknowledge to its bus controller.
7. Local Bus Controller acknowledges the invalidate request of the bus and the transaction is complete.

If the remote node is not the last element of the SCI sharing list, then the status sent back to the invalidating node would have reflected this fact. In that case steps two through six would be repeated for each node on the sharing list. Only when the "Tail" issues a response to the invalidating node is the sharing list completely purged. An example of the standard SCI invalidation methodology of a list with two additional elements is portrayed in figure 5-1a.

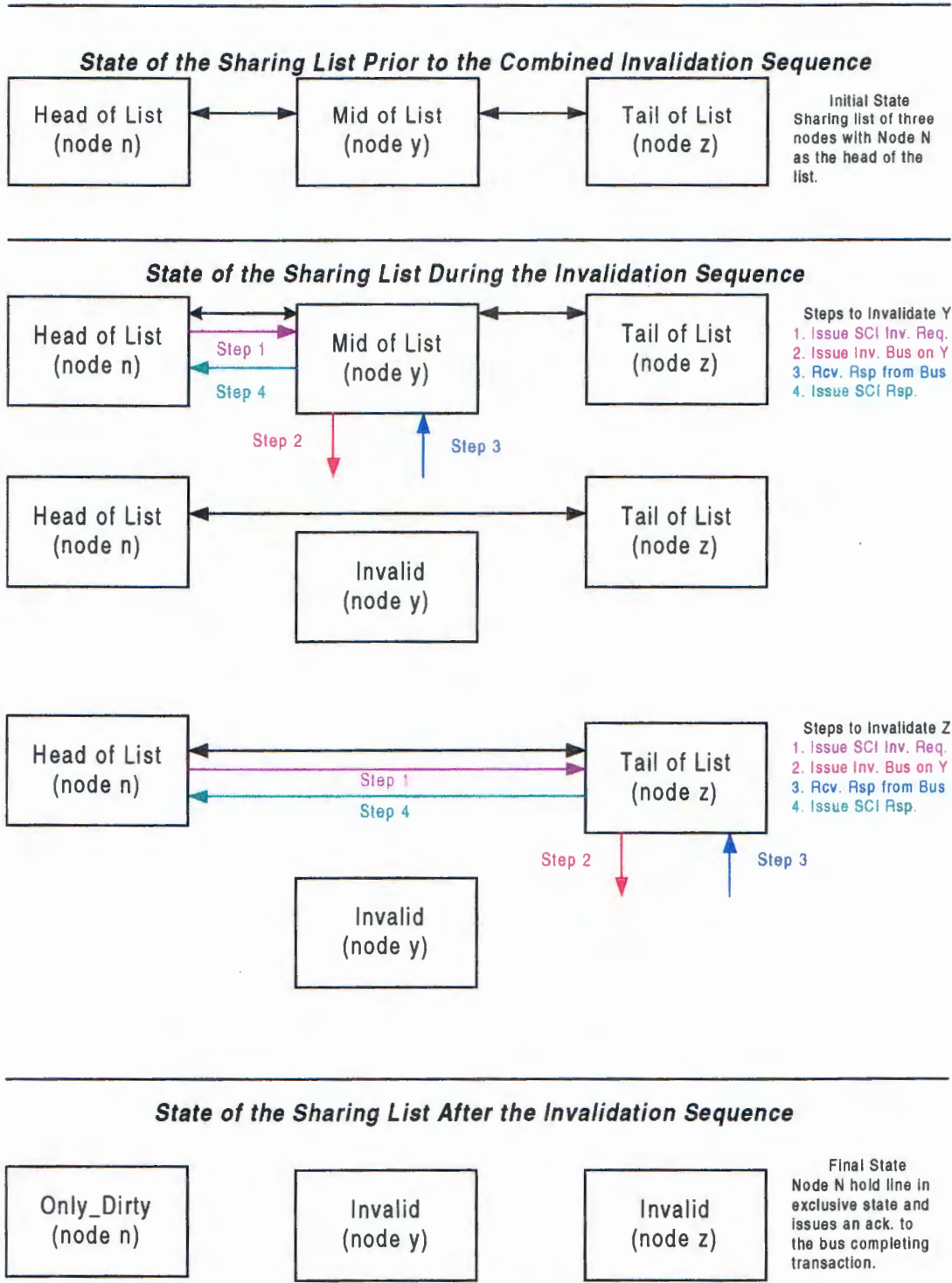


Figure 5.1A - Standard SCI List Invalidation

Figure 5.2 represents the steps the Reduce List Invalidation extension employs to invalidate a sharing list. Again like in 5.1, this example has only one additional node of the sharing list.

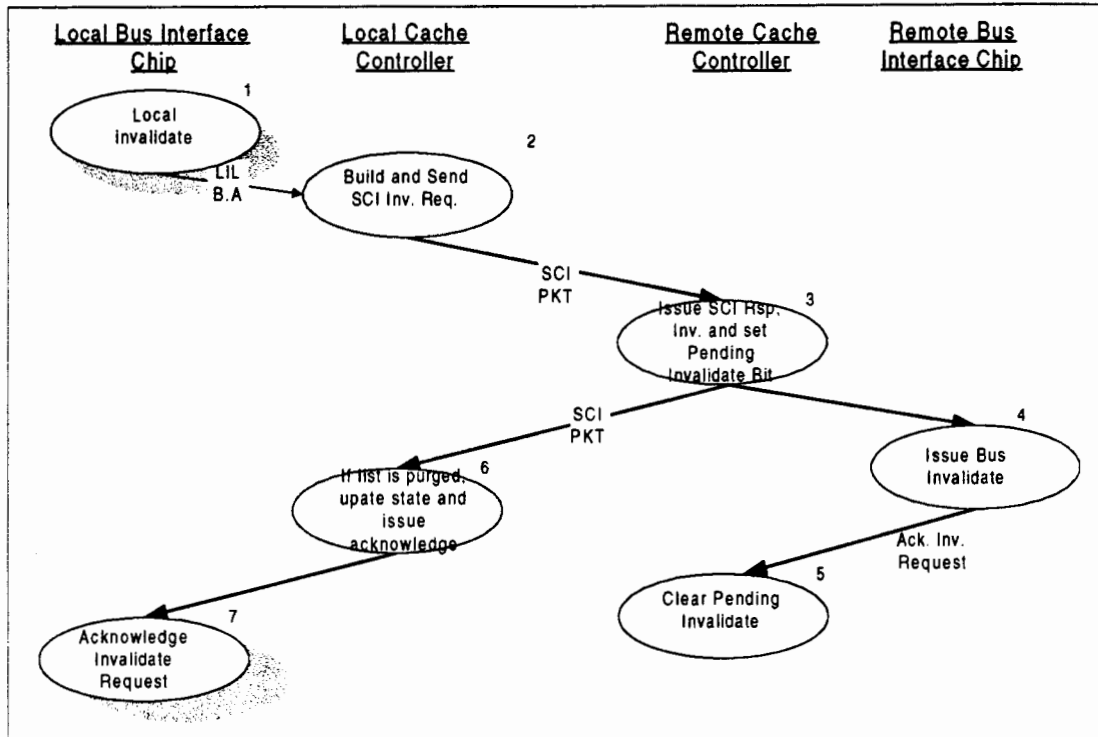


Figure 5.2- Reduced List Invalidation Flow of Events

The steps taken for the Reduced List Invalidation extension is very similar to the standard SCI methodology. The steps for this extension to invalidate a sharing list are as follows:

1. Bus interface issues to the cache controller a invalidate request.
2. Cache Controller looks up the line, an SCI Invalidate Request Packet is built and is targeted to the node stored as the “Forward Pointer”.

3. Remote Cache Controller receives Invalidate Request, checks the state of the line and in most cases issues an invalidate request to the bus. In parallel it issues the SCI response packet signaling that the invalidate request is complete. Also the “pending invalidate” bit is set, signaling an invalidate request has been acknowledged “early”.
4. Remote Bus Controller issues request on node’s system bus and when complete acknowledges invalidate request.
5. Remote Cache Controller updates state to invalid and issues SCI response and clears the “pending invalidate” bit corresponding to this request. All resources for this request are released for a new SCI command.
6. In parallel with the invalidation on the remote node, the Local Cache Controller processes the SCI response, sees the SCI list is completely purged, updates its state and issues acknowledge to its bus controller.
7. Local Bus Controller acknowledges the invalidate request of the bus and the transaction is complete and then releases all resources for this request for a new request from the system bus.

The fundamental difference is parallelization of the SCI response and the actual invalidation on the remote node. Again, if the sharing list consisted of additional nodes, steps two through six would repeat for each node on the sharing list.

With the Reduce List Invalidation extension, the completion of the remote node's invalidation and the invalidating node's acknowledging the completion of the purging of the list are now completely asynchronous. The breaking of the connection between these steps is represented by separate branches of steps 4 & 5 and steps 6 & 7 in figure 5.2. The time saved at each node through the invalidation time is additive. The longer the sharing list, the greater the reduction of time for invalidating the list. An example of the reduced list invalidation methodology of a list with two additional elements is portrayed in figure 5-3.

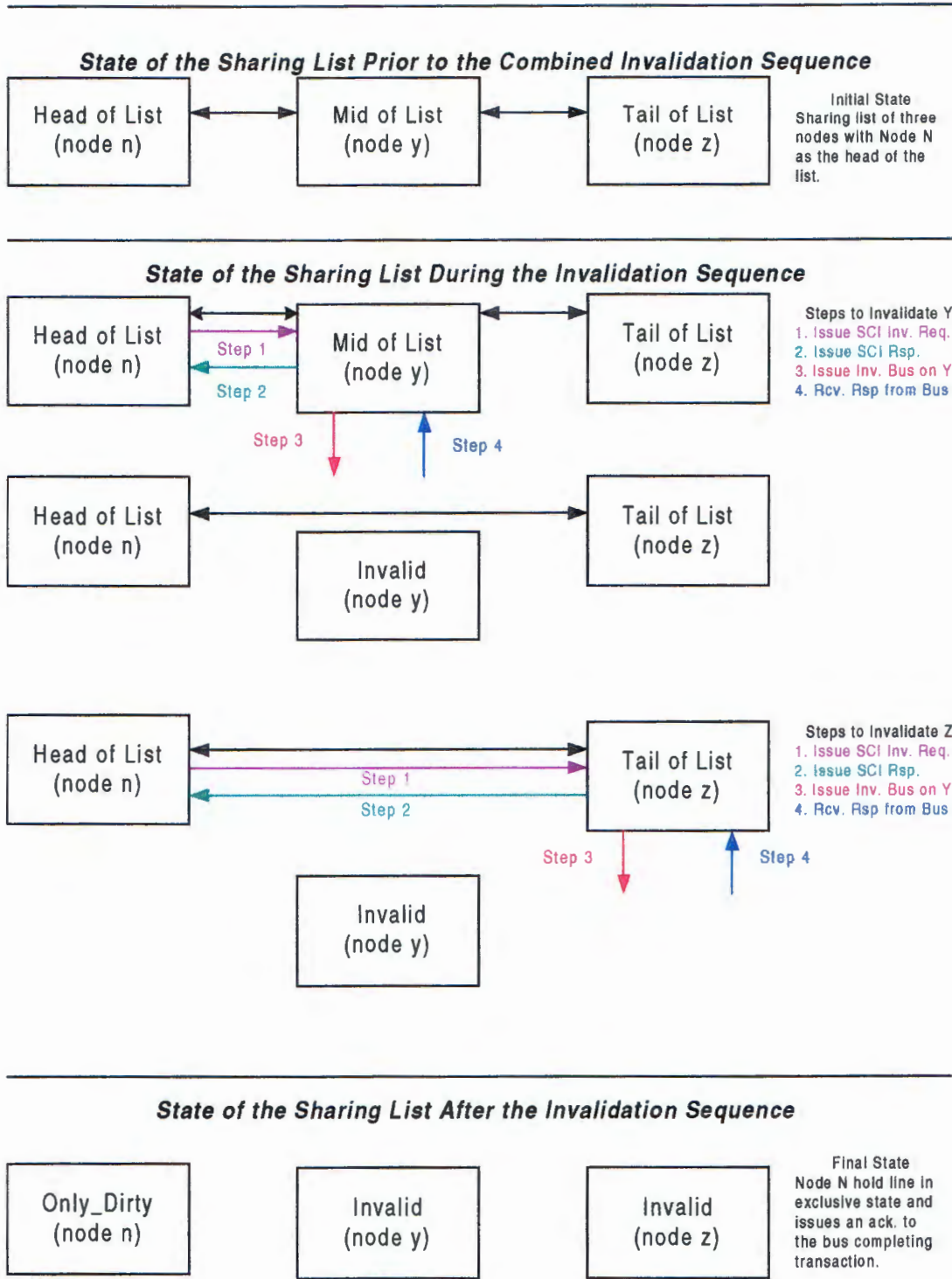


Figure 5.3 - Reduced List Invalidation Sequence

The penalty for acknowledging the invalidate request early is, of course, complexity. With the early acknowledgement comes the responsibility to maintain ordering to ensure the correct memory consistency model. To maintain ordering the “Pending Invalidation” logic is employed. This logic provides the ability to “queue” events on the completion of previously posted events that are currently in progress. Figure 5.4 is a representation of this queuing process.

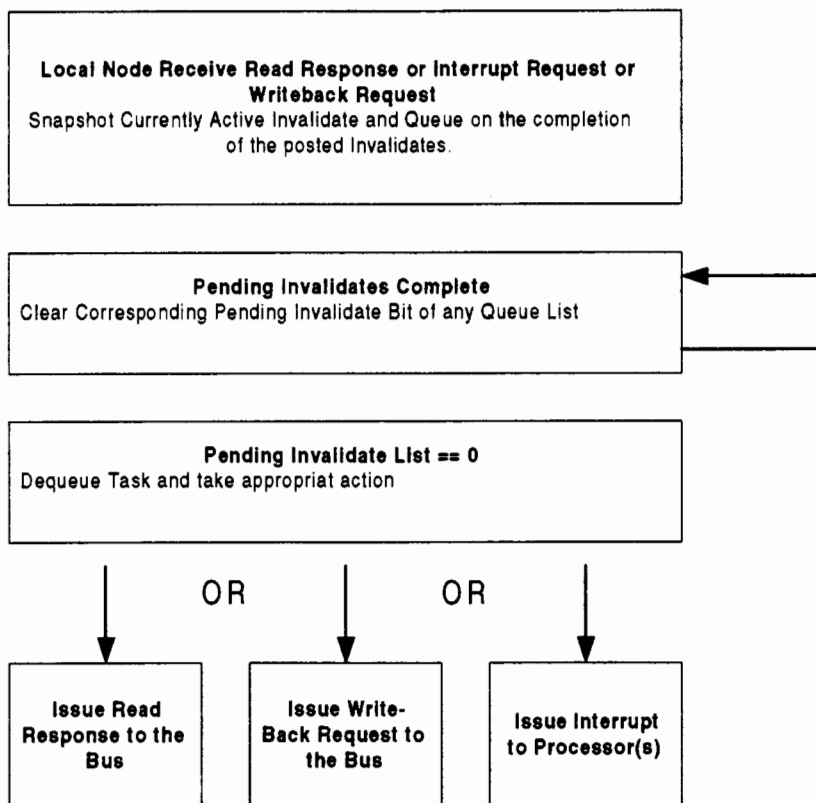


Figure 5.4 Pending Invalidation Flow of Events

The programmable protocol engine of the cache controller executes a specified group of instructions for a particular “event”. In the case of read responses, write-backs, and interrupts the specified instructions are to capture a copy of the currently “pending invalidates” and queue on the completion of these tasks. The protocol engine is completely free to work on other requests or responses in the interim. When the “Pending Invalidate List” is equal to 0 and the protocol engine is “Idle”, this particular thread is de-queued and continues to sequence through the protocol engine’s program to complete the specified routine. This specified routine issues the appropriate response or request to complete that specific transaction.

Via the “Pending Invalidate” and queuing logic the cache controller is able to guarantee the ordering requirements for a processor consistency model. The processor consistency model is guaranteed by preventing any processor local to a specific node:

- To have its cacheable read response passing a previously posted invalidate.
- To be able to view the writes (via reads) of another processor out of the order the writes were issued.

5.2 Comparison of the Invalidation Methods of the Standard and New Protocols

The flow of events between the two invalidation methodologies is represented in the comparison of figures 5.1 and 5.2. The primary difference is the overlapping of steps 4 and 5 with steps 6 and 7 in figure 5.2. It is this overlap which provides the performance increase by parallelizing the sending of the SCI response packet with the node's invalidation sequence.

However, acknowledging the completion of the invalidate early forces the cache controller to maintain ordering of events observed by this node. This is to ensure that a processor on this node does not observe writes from a distant processor in an order different from the order were issued. If the cache controller does not maintain the ordering of events, then the processor consistency model will be violated. The scenarios that must be prevented are the classical

- Read passing a write scenario.
- Write passing a write scenario.

To prevent these scenarios the Pending Invalidate bits and queuing logic of the protocol engines are used. Processor consistency is maintained by delaying all read responses, write-back and interrupt requests behind all currently posted invalidate requests. Queuing read responses behind posted invalidates ensures that a read would never pass a write. Also queuing write-

backs and interrupts behind the posted invalidates ensures that a write would never pass another write. The mechanism used to clear the pending invalidate bits is the acknowledge response from bus signaling that all copies of the cache line have been invalidated. As the acknowledge responses are received the corresponding pending invalidate bits are cleared. When all previously set pending invalidate bits are cleared the event is de-queued and issued to the bus.

5.3 Merging of Reduced List Invalidation with List Invalidation Method

The next logical step in decreasing the invalidation time is to develop a method of forwarding the invalidation request down the list and thus eliminating the intermediate SCI response packets. Work was previously done in this area. This work is referred to as the “Fast Invalidate Extension for SCI” [52]. Logically, a complete protocol was developed based on this forwarding concept. However, due to hardware limitations of the SCI physical interface chip, this extension is not currently realizable. It should be noted that the two invalidation methods are not mutually exclusive. If changes could be made to the SCI physical layer definition, the optimum invalidation methodology would be the merging of these two extensions. This concept is addressed in further detail in section 7.

6. Testing and Measured Results

A primary goal of this research is to develop a “realizable” cache coherency protocol that could be demonstrated on the NUMA-Q system of Sequent Computer System Inc. To that end, the developmental and debug strategies of this company were followed. In general, the development of the cache coherence protocol’s basic structure (or the simplest cases) was exercised in a simulation environment. The complete protocol (all end cases, roll out strategy, hardware imposed limitations and race conditions) was debugged in a system environment. The system environment was initially based on a “two node” configuration (8 processors). After the “two node” configuration was stable, the system environment grew to three nodes and finally four nodes.

6.1 Development Strategy

The exploitation of the programmability of the cache controller is the key to the development phase of this research project. The cache controller’s RAM based protocol engine thread based architecture executes the instructions stored for a particular event. An example of the instructions of the protocol engine is shown in the following figure.

```

;*****
;* Entry Point: DP Response for LRL
;*****
LRLRsp:
  CMPI R_M_JX  RspHdr0h,Resp64, 0, 7, DPSENDREQ,NOOP ;
  MOVA R_M_JE  RspHdr1,RspHdr1,16,15,LRLDONE ; Extract MemID
  MOVA R_M     Hdr0, Hdr0, 0, 15 ; Save TransID
  OR R_M      RspHdr1, B_ALUResult, Hdr0, 16, 31
  MOVI       DPPostReq, PostVec ; post send req.
  MOVA      RspHdr1, LclDirPtr ; Update MemID
  MOVI JMP   0xffffffff, RspHdr0, IDLE ; Set RspHdr0 for debug.

```

Figure 6.1 – Example of Protocol Engines Program Language

This code segment is executed when a response packet has been received by the local cache controller for a sharing list which is being converted from “Dirty” to “Fresh”. The first instruction is a “Compare Immediate Instruction with a rotate and mask extension”. It is comparing the Response Header 0 to see if this response contains 64 bytes (the size of a cache line). Also, this instruction has a jump operand appended to it. All instructions can jump on a previously set “jump condition codes”. The “JX” prefix is for an extended jump condition flag, in particular the “NOOP” bit. If the NOOP bit was set, the protocol engine would have immediately “jumped” to the “DPSendReq” label. The following instructions are examples of Move Instructions from the “A-Side” of the ALU with “rotate/mask operands.

The code was developed, and with the use a table based assembler, compiled. The binary files created can be either brought into a simulation environment or downloaded into the cache controller during initialization.

6.1.1 Simulation Environment

The initial testing of the two cache extensions was done in a chip standalone simulation environment. This environment entails the RTL of the cache controller (written in Verilog) and the standalone jig that emulates the bus interface as well as the interface to the SCI. This environment had the ability to generate requests or responses from the bus or SCI interface and exactly predict the behavior of an individual cache controller for a specific case.

This environment initially tested and isolated implementation cases for the Guaranteed Forward Progress and Reduce List Invalidation extensions. The environment simulated the cache controller's ability to issue a read request to the bus for a "Fresh" list, as well as test the queuing of responses on pending invalidates. It was never the intention to modify the complete set of tests in the standalone environment to provide comprehensive testing of the new coherency protocol. The comprehensive testing would be done in the development environment.

6.1.2 Development Environment

Once the extensions passed the initial simulation tests and a multi-node system was available, the research migrated to a development environment.

The development environment consisted of a multi-node system capable of running either standalone diagnostic tests or operating system diagnostic tests. Using actual hardware, the time to uncover a design flaw in the new protocol was greatly accelerated. Initially, a “two node” system (an eight processor system) was used. Once that system was stable, the development process migrated to a “four node” system (sixteen processor system).

The basic steps for both system configurations were to first run diagnostics on the system, boot the system, and then run more stringent tests under the operating system control. These basic tests run in a simple system environment and were intended to proof and debug hardware. These tests were never intended to provide a cache coherency validation suite of tests and were not a very good debug mechanism.

Once the diagnostic tests passed, the system was then booted. The system boot process is where most implementation problems were uncovered. A large portion of the cache coherency protocol is tested during the boot process. A major problem with debugging a cache coherency protocol

through the boot process is recovering the system after a crash due to a coherency bug.

6.2 Debug Environment

The basic environment used to debug a cache protocol extension is provided in figure 6.2. In addition to the “four node” system, a four channel logic analyzer monitored each cache controller’s connection to its node system bus, as well as the program counters of the protocol engines.

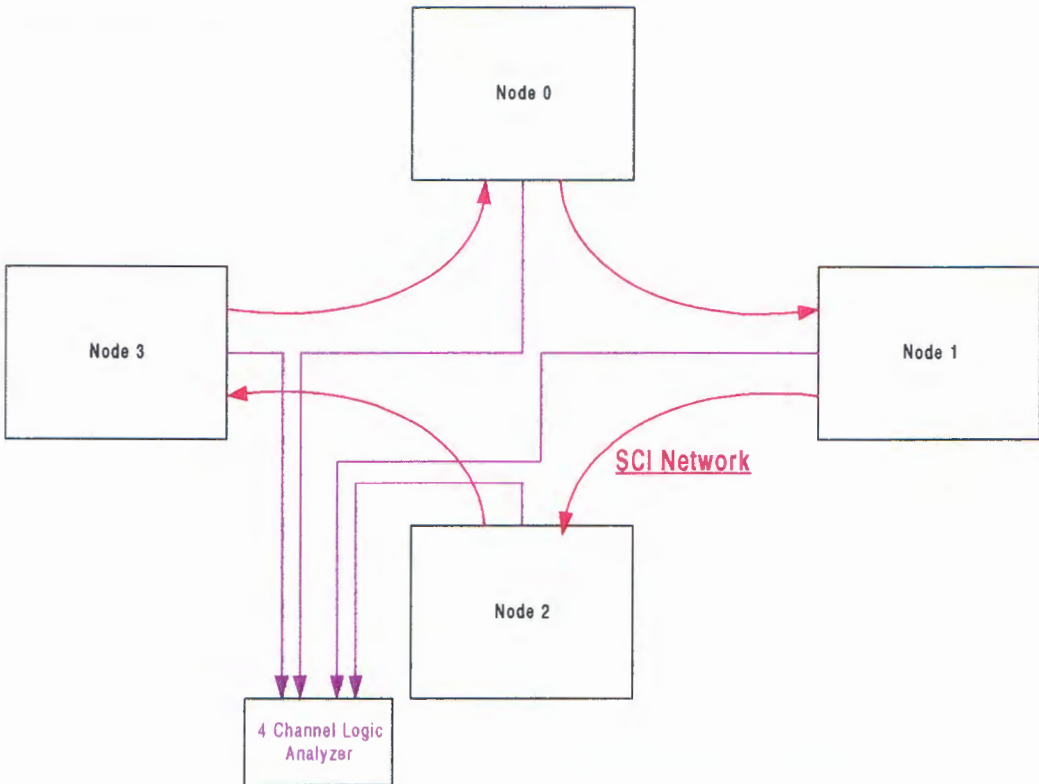


Figure 6.1 – System Debug Environment

- The debug environment, even with the correlated traces of the channels from the logic analyzer, provided only a limited view of what is actually happening

on each node of the system. The SCI interconnect, system bus, I/O busses, and memory of each node was not instrumented.

6.3 Measured Results

The performance metrics used to measure protocol extensions took four forms. At the lowest level, logic analyzer traces were taken as the first low level metric. Following this, some initial “Read” measurements were taken on a system under moderate load. Invalidation tests were made next. The final test was to see how the extensions performed under a “real” load. The final metric is based on a system running a database benchmark.

6.3.1 Logic Analyzer Traces

Logic analyzer traces were used initially to debug implementation bugs of the cache coherency protocol extensions. The traces were also collected as an initial metric to see if the extensions were performing as they were intended to perform. An explanation of the trace is as follows:

- Node Column indicates which Node the data is coming from. The entries could be from Q0 – Q3 in a “four node” system.
- CMD Column Indicates the Command (or Response) which will be ultimately issued on the node’s system bus. This interface, like the node’s system bus, is based on a split transaction architecture.

- ID Column identifies which request or response is being issued. The interface supports up to 32 outstanding requests in each direction.
- Debug Port of the Cache Controller – This port indicates the Thread # and program counter for the Dual protocol engines of the cache controller.
- Timestamp Column provides the elapsed time since the previous sample. It should be noted that the nodes base frequency is 90MHz (or a 11.11nsec duty cycle) and the resolution of the logic analyzer is 0.5nsec.

The first trace provided is of Q0 issuing a read response for a previously issued request. This trace portrays the “ten clock” penalty of the cache controller due to a hardware design flaw of the device. The second is a trace of a sharing list invalidation sequence.

6.3.1.1 Logic Analyzer Read Trace

The following figure contains a logic analyzer trace of the read response timing. This trace shows the unloaded latency addition to every read (or in this case "ACK") for queuing on the posted invalidation.

NODE	CMD	ID	DEBUG	Timestamp
Q0	IDLE	1F	F001F192	11.000 ns - Start Overhead
Q0	IDLE	1F	F001F193	11.000 ns
Q0	IDLE	1F	F001F001	11.000 ns
Q0	IDLE	1F	F001F001	11.000 ns
Q0	IDLE	1F	F001F001	11.000 ns
Q0	IDLE	1F	F001F001	11.000 ns
Q0	IDLE	1F	7001F00A	11.000 ns
Q0	IDLE	1F	7001F3AC	11.000 ns
Q0	IDLE	1F	7001F3AD	11.000 ns
Q0	IDLE	1F	7001F3AE	11.500 ns - End Overhead
Q0	IDLE	1F	7001F194	11.000 ns - Code Seq. for either
Q0	IDLE	1F	7001F195	11.000 ns
Q0	IDLE	1F	7001F196	11.000 ns
Q0	IDLE	1F	7001F197	11.500 ns
Q0	IDLE	1F	7001F198	11.000 ns
Q0	IDLE	1F	7001F199	11.000 ns - Code Seq. for either
Q0	S_NULL	1F	7001F001	55.500 ns
Q0	S_ACK	0F	7001F001	11.000 ns

Figure 6.3 Logic Analyzer Trace of a Read Response

The NUMA-Q system cache controller is a dual engine implementation. This is to say that each cache controller contains two complete protocol engines.

There is a protocol engine for "even" cache lines and one for "odd". A design mistake was identified with the dual protocol in the area of the Queuing Logic.

In the current implementation, a protocol engine can only check to see if it has anything posted (not if either engine has anything posted). This oversight

forces the queuing of all cacheable read responses. In most cases, there

aren't any invalidates posted and therefore, the "ten clock" overhead of

queuing and de-queuing is incurred for no reason. This "ten clock" penalty is

an implementation issue and not an architecture issue. However this penalty

is contained in all the data collected.

6.3.1.2 Logic Analyzer Invalidate Trace

The following figure was trace collected from a “four node” system. This trace is of the Standard SCI protocol. The local node issues a request to invalidate a line. The sharing list consists of three other nodes (1, 2, and 3).

Node	CMD	ID	DEBUG	Timestamp
Q0	O_LIL	05	F001F001	11.000 ns - Local Node issues an Inv.
Q0	O_NULL	14	F001F001	11.000 ns
Q1	S_NULL	1F	70017001	969.000 ns
Q1	S_CIL	07	70017001	11.000 ns - 1 st Node issues request
Q1	O_NULL	1F	70017001	389.000 ns
Q1	O_ACK	07	70017001	11.000 ns - 1 st Node issues SCI Resp.
Q2	S_NULL	1F	70017001	1.569,500 us
Q2	S_CIL	07	70017001	11.000 ns - 2 nd Node issues request
Q2	O_ACK	07	70017001	277.500 ns- 2nd Node issues SCI Resp.
Q3	S_NULL	1F	70017001	1.273,500 us
Q3	S_CIL	07	70017001	10.500 ns - 3rd Node issues request
Q3	O_ACK	07	70017001	278.000 ns- 3rd Node issues SCI Resp.
Q0	S_NULL	1F	F001F001	754.500 ns
Q0	S_ACK	05	F001F001	11.000 ns - Local Node issues Response

Figure 6.4 - Trace of Std. Protocol Invalidation Sequence

The first Invalidate request is targeted to Q1. When Q1 receives the acknowledgement from the bus, it then issues the response back Q0. This scenario is repeated for Q2 and Q3. When Q0 receives the last SCI response, it then issues the acknowledgement to the local bus. The elapsed time of this transaction is the sum of the Timestamps, 5586 nanoseconds.

The next trace is of a similar scenario, but instead of the Standard SCI Protocol the new protocol was used. As in the previous trace, the list being

invalidated requires issuing 3 SCI Invalidate requests. A difference between the previous trace is the order of the sharing list. Instead of an order {Q0, Q1, Q2, Q3}, the order for this trace is {Q0, Q3, Q2, Q1}.

Node	CMD	ID	DEBUG	Timestamp
Q0	O_LIL	04	7001F001	11.000 ns
Q3	S_NULL	1F	F0017360	760.500 ns
Q3	S_CIL	07	F0017001	11.000 ns
Q3	O_ACK	07	F0017001	278.000 ns
Q3	O_ACK	17	F0017001	11.000 ns
Q2	S_NULL	1F	F0017360	784.500 ns
Q2	S_CIL	07	F0017001	10.500 ns
Q2	O_ACK	07	F0017001	278.000 ns
Q1	S_NULL	1F	70017358	826.000 ns
Q1	S_CIL	07	70017001	11.000 ns
Q1	O_ACK	07	70017001	278.000 ns
Q0	S_NULL	1F	7001F001	174.000 ns
Q0	S_ACK	04	7001F001	11.500 ns

Figure 6.5 - Trace of New Protocol Invalidation Sequence

The time to invalidate this list is 3444 nanoseconds. The reason for the reduction in time is due to the overlapping of the SCI response and issuing the request down to the local bus on each remote node.

The difference in time to invalidate three other nodes of a sharing list from the logic analyzer traces is 1142 nanoseconds, a decrease in list invalidation time of twenty percent. It should be noted that these measurements were taken

during the boot cycle of the system. The system load during this time is very low.

6.3.2 Lock and Invalidate Tests

The following tests were developed to measure the worst case scenario to invalidate an SCI sharing list. There are four specific versions of the test and the versions are referred to as “Share List”, “Share List – Atomic”, “Share List – List” and “Share List – List Atomic”. The tests were initially developed by Paul McKenney of Sequent to measure different attributes of cache based locks in a CC-NUMA environment. The basic premise of the tests is to have a processor of a node write to update a list structure and have each “reading” processor read the updated structure. This structure consists of 64 elements. Each element is contained in a cache line. The Element consists of a pointer to the next cache line and a count. The list structure resides in a contiguous address range. The pool of processors for the test consists of processor 1 writing the “List”, processors 2 to “n-1” reading the list, and processor (n) controlling the activity. The actual steps of the test are described below. The description is based on a pool of 60 processors. The test has two parallel threads of activity. The first is the activity performed by the “control” processor. The flow of this activity is as follows:

- Control processor writes control cache line.
- All other processors read the control cache line.

- Selected processor writes the cache line when it is done.
- All Processors read the control cache line.
- Control processor writes control cache line.
- All other processors read the control cache line.
- Next selected processor writes the cache line when it is done.
- Process continues for all the processors in the list.

Again the control of the test is contained in a single cache, while the other thread of activity involves the manipulation of the 64 cache line structure. The other thread's flow of events is as follows:

- Processor 1 writes the structure when instructed.
- Processor "n" reads the structure (when instructed).
- Processor n+1 reads the structure (when instructed).
- Process continues for all the reading processors in the list.
- Processor 1 writes the structure when instructed (invalidating all the sharing lists of the cache lines in the process of updating the individual elements).
- Processor "n" reads the updated structure (when instructed).
- Processor n+1 reads the updated structure (when instructed).
- Process continues for all the reading processors in the list.

The environment created by the "Share List" tests gives the user the ability to grow a sharing list in a guaranteed order and length as well as invalidate the list on command. Using the "Share List" tests, a user can measure Read

Latency numbers of cache lines under contention, as well as list invalidation times.

Differences in the four versions of the “Share List” tests are based on how the elements are updated and how the list of elements of the structure is traversed. The differences between the tests are as follows:

- **Share List:** The writing processor uses a simple increment instruction to update the count and does not use the chained link structure of the List to traverse it. Since a simple increment instruction is used, the processor is allowed to issue multiple write instructions.
- **Share List – Atomic:** The writing processor uses a lock increment instruction to update the count and does not use the chained link structure of the List to traverse it. The lock increment prevents the processor from issuing multiple writes at any given instant.
- **Share List – List:** The writing processor uses a simple increment instruction to update the count, but uses the chained link list embedded in the element to traverse the list. Since a simple increment instruction is used, the processor is allowed to issue multiple write instructions. But in this test the writing processor must also read the pointer embedded in the line.

- **Share List – List Atomic:** The writing processor uses a “locked” increment instruction to update the count and uses the chained link list embedded in the element to traverse the list. The locked increment instruction prevents the processor from issuing multiple writes at any given instant, as well as forcing the writing processor to read the pointer embedded in the line.

6.3.2.1 Share List Performance Measurements from a Four Node System

The following measurements were taken on a “four node” system, each with a bus frequency of 90MHz. Each node contained four 360MHz Intel XEON processors. The tests were set up to have four reading processors, 1 writing processor, and the control processor. It should be noted that the four reading processors are physically located on different nodes. The following four bar graphs compare the results of the Share List tests. These comparisons are of the Standard SCI protocol to the new protocol with both the extensions enabled.

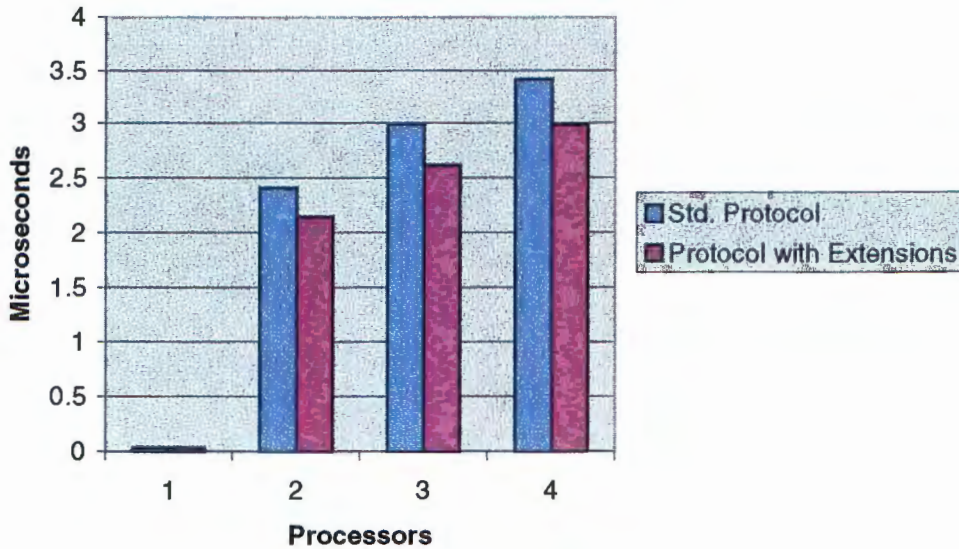


Figure 6.6 – Share List Invalidation Time

In figure 6.6, the time to invalidate the first processor is only gated by the bus invalidation time of the local node, since the writer and the first reader are resident on the same node. The other processors (processor 2 – 4) are located on remote nodes and require the SCI Sharing List to be invalidated.

6.3.2.2 Share List - Atomic Performance Measurements from a Four Node System

In general for the “simple” Share List case, where there are four processors and three remote nodes, the new protocol extension provides a list invalidation time reduction of 432nsec. It should be noted that the invalidation time theoretically continues to decrease the longer the sharing list for the Share List Test. This fact is represented in figure 6.3.

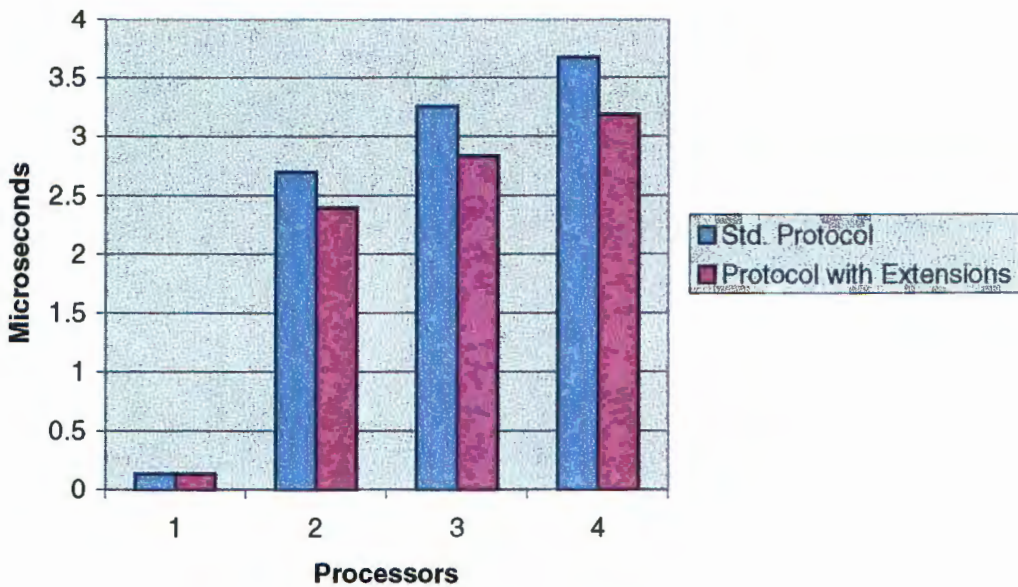


Figure 6.7 – Share List - Atomic Invalidation Time

Figure 6.7 represents the results of the Share List –Atomic test. Unlike the simple “Share List” case the writing processor is using a “locked increment” instruction. This eliminates the chance of any parallelization to happen due to the posting of multiple writes by the writing processor. As expected, the invalidation time increases due to the serialization of the writes. This also amplifies the differences between the two cache protocols. In the “four readers” case, the invalidation time difference grew to over 480nsec. As in the previous case, the differences between the two invalidation methods grow with the length of the sharing list. Refer to the graph of figure 6.6 to view the representation of this fact.

6.3.2.3 Share List - List Performance Measurements from a Four Node System

The “Share List – List” test requires the writing processor to actually extract information from the cache line that is being written. This simple act adds overhead and negates some of the benefit of the new invalidation scheme.

Figure 6.8 reflects the difference in invalidation time between the two cache protocols.

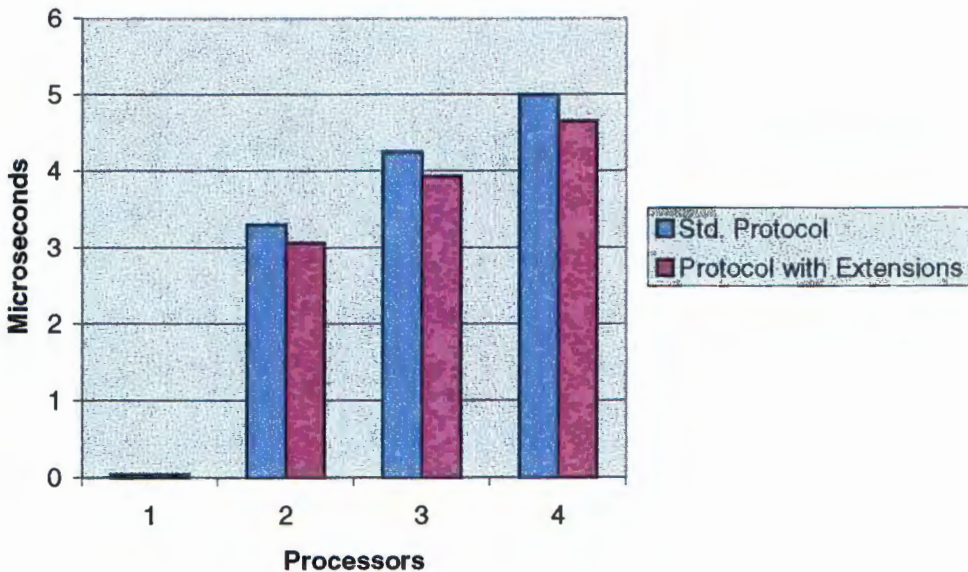


Figure 6.8 – Share List - List Invalidation Time

The time difference between the two protocols in the Share List – List case is approximately 340nsec in the three remote node cases (2, 3, and 4 processors).

6.3.2.4 Share List - List Atomic Performance Measurements from a Four Node System

The “Share List – List Atomic” test requires the writing processor to actually extract information from the cache line that is being written, as well as use a “locked” increment instruction to perform the update. This simple act adds overhead and negates some of the benefit of the new invalidation scheme. This is clearly the worst case scenario of the four sharing list tests. But even this case shows that the new cache protocol with the invalidation extension is still higher performant than the standard SCI protocol. As in the previous three cases, figure 6.9 shows that the new cache protocol provides a consistently shorter time to invalidate the sharing list.

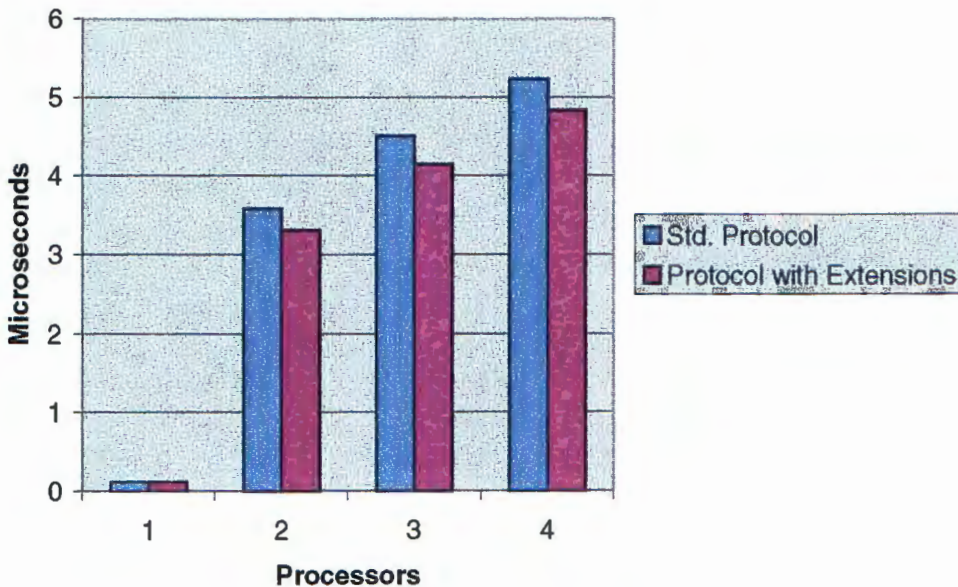


Figure 6.9 – Share List – List Atomic Invalidation Time

Figure 6.10 is a comparison of the two cache protocols for each of the four Share List tests. This figure shows the differences in time it takes the writing processor to update a single element of the structure. The measurements taken are the averages of three runs. Each run performed the specific test 100,000 times. In all four cases (Share List, Share List – Atomic, Share List – List, and Share List – List Atomic) the “Reduced List Invalidation” methodology provides for a shorter period of time to purge a sharing list in all four versions of the test.

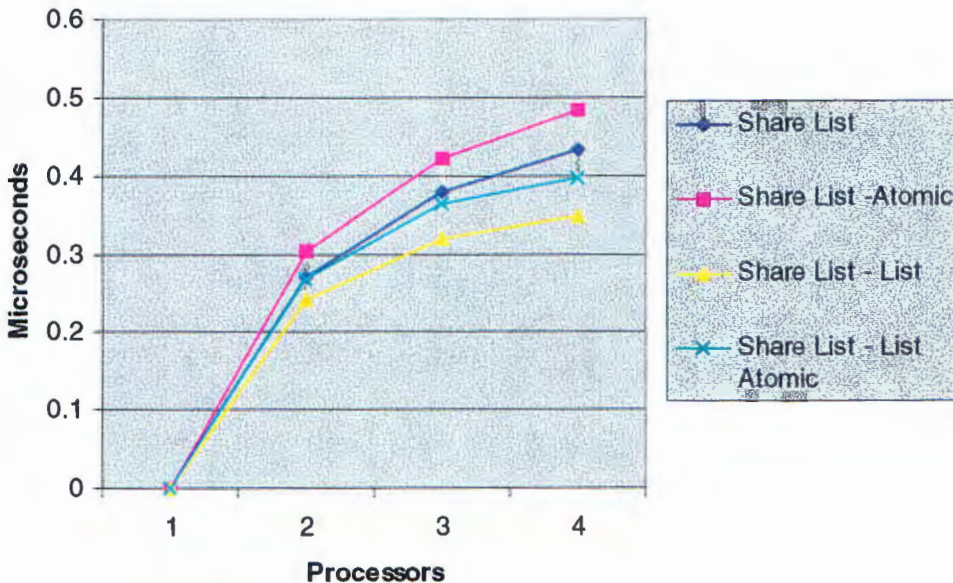


Figure 6.10 – Invalidation Time Differences between Standard and New Cache Protocols

Another way to view this data in figure 6.10 is to look at the percentage decrease in time between the two cache protocols. Figure 6.11 shows the percentage change for the four Share List tests. In the case of only

invalidating the line (the Share List and Share List – Atomic cases) the time to purge the sharing list was reduced by approximately thirteen percent. The worst case latency reduction for the new protocol in the Share List tests was a seven percent.

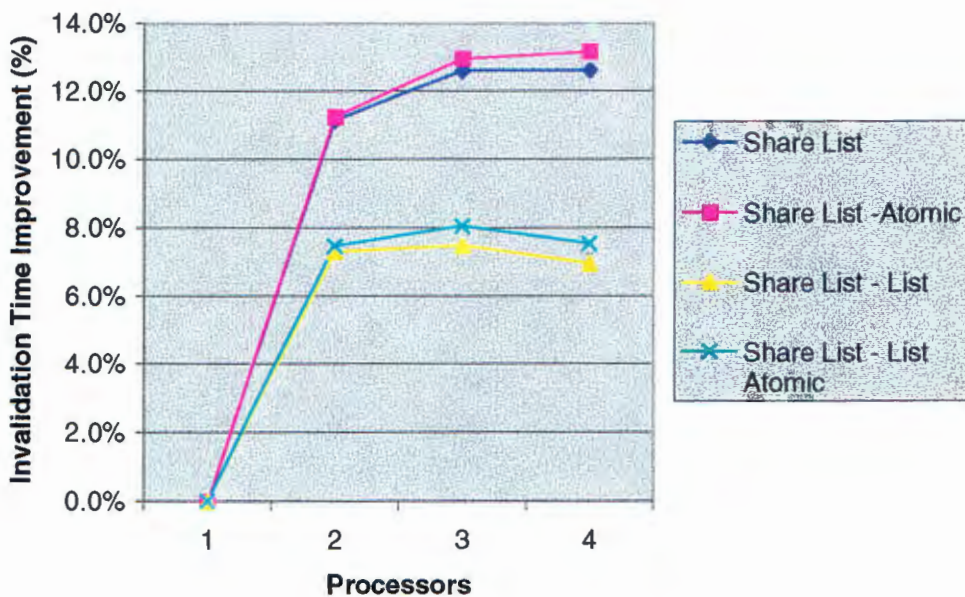


Figure 6.11 – Percentage change Between the Two Protocols

6.3.3 Read Measurements

As shown in the previous section, the time to purge a sharing list is reduced by the new cache protocol. These measurements were made on the same system, with exactly the same configuration (same OS, memory size, number of disk drives, same background load, etc.). The only difference between the runs was the cache protocol. The penalty of the new cache protocol is in the read latency. The reason read measurements are a concern because of the

issue that the Reduce List Invalidation Extension negatively impacts read responses. Are the gains of the Reduce List Invalidation Extension negated by the read response penalty? Also, what has to be taken into account for this generation of cache controller, is that all read responses incur at minimum a “ten clock” penalty due to a limitation in the hardware. In most cases no invalidates are posted and the read response is delayed by the queuing – de-queuing time of the cache controller.

The data provided in the figure 6.12 is based on the average time it took to completely install a cache line at the remote node. These measurements were taken using the “Share List - List Atomic” test. In all cases the first remote read is to a line that is “home”. All other remote reads are prepending to a “Fresh” sharing list. In prepending to a “Fresh” sharing list, the remote access is burdened with the additional SCI request/response transaction to the “old” head to notify it that it is no longer head of the list. Figure 6.7 consists of four latency measurements. These measurements include:

- Read latency for the standard protocol.
- Read latency for the new protocol.
- List invalidation time for the standard protocol.
- List invalidation time for the new protocol.

Due to limited access time to a larger system (64 processors or 16 node system), the invalidation time for sharing lists of four through lists of thirteen could not be measured.

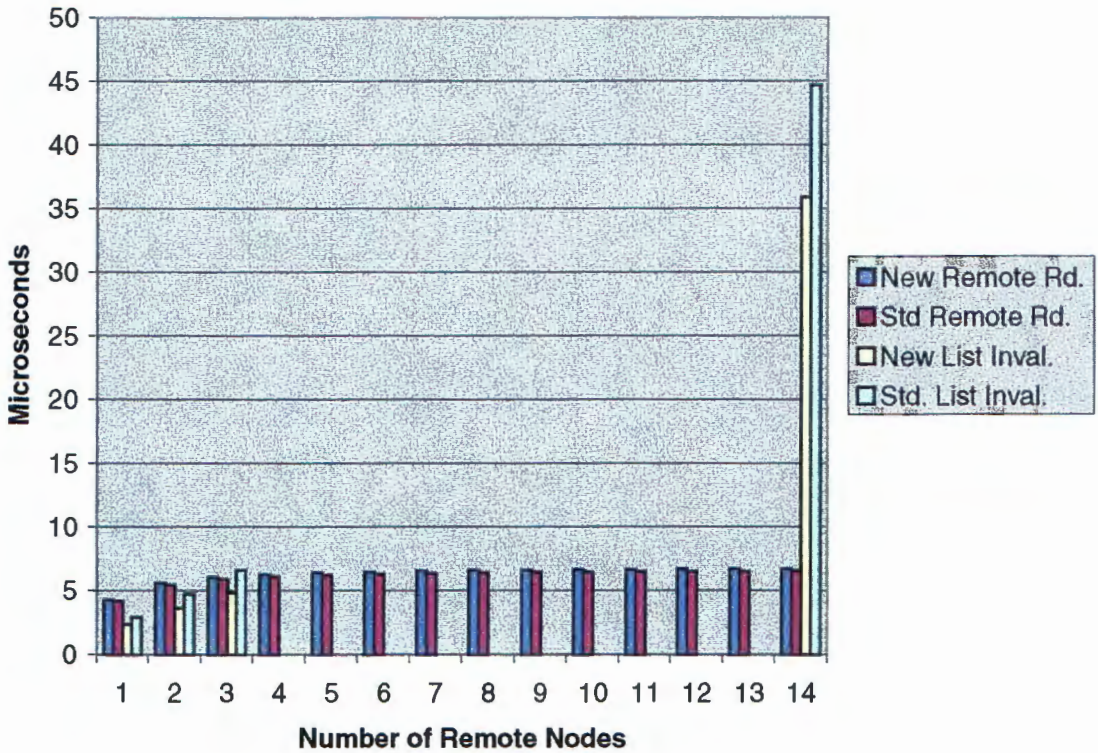


Figure 6.12 – Read Response versus List Invalidation Time

Figure 6.12 portrays a major issue with the SCI protocol. As a sharing list grows the time to purge the list also grows. In contrast the time to read a cache line approaches a consistent number. It should be noted that these measurements were taken in a system under load. During these tests all

processors in the system were measured to have a greater than 90% utilization.

The average time to purge a sharing list with 14 additional nodes for the two cache coherency protocols is as follows:

- For the Standard Protocol – 44.716 microseconds.
- For the Protocol with Extensions – 35.891 microseconds.

The resulting reduction in latency for the write is a 19.7% decrease. In comparison to the write time, the average latency for reads for the “Share List – List Atomic” test in a larger system is as follows:

- For the Standard Protocol – 6.539 microseconds.
- For the Protocol with Extensions – 6.728 microseconds.

As stated earlier, the increase in read latency is due to two components. The first is a limitation of the cache controller which adds an additional ten clocks (~0.111 microseconds) to remote accesses. The second issue is the overhead due to queuing read responses behind posted invalidates.

6.3.4 Database Measurements

The next step in comparing the two protocols was to measure the system performance running a real application. The application chosen was that of a relational database. To generate system load, an OLTP warehouse

benchmark was used. The load used was modeled similarly to the host side of a TPC-C database benchmark.

The database transaction workload used is intended to be a representative workload of a database application managing the inventory for a company spread across a number of sites. The workload is intended to be representative of a database for a “typical” warehouse application. A description of this type of workload is provided by the TPC-C benchmark.

The following is a description of the actual benchmark. “As an on-line transaction processing (OLTP) system benchmark, TPC-C simulates an environment in which a population of terminal operators executes transactions against a database. Given that its context is centered on an order-entry environment, the benchmark includes the activities of entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. However, it should be stressed that TPC-C is not designed to specify how best to implement an order-entry system. The benchmark portrays the activity of a wholesale supplier, but is not limited to the activity of any particular business segment; rather, it is designed to represent any industry in which one must manage, sell, or distribute a product or service” [9].

The following is the list describing the system hardware configuration used in this database test.

- System was configured as a 2, 3 or 4 node system.
- Database was “tuned” using the “Standard” SCI cache protocol. No additional tuning was done on the new cache protocol.
- Memory of the system
 1. 2 Node System – 8Gbytes.
 2. 3 Node System – 12Gbytes.
 3. 4 Node System – 16Gbytes.
- Size of Database is 820 “Warehouses”
 1. Database striped across 384 4Gbyte Disks.
 2. Approx. Size of Database is 100GBytes.
- System Processors – Intel XEON processor (360MHz).
- System Node Frequency – 90MHz.

The database was tuned for a system using the “Standard” SCI protocol. Data was collected and the system rebooted running the “Protocol with Extensions. No additional tuning to achieve an optimal performance number was done to produce a higher transaction number. This was done specifically to create, as best as one can, the exact circumstances to measure the differences between the cache coherency protocol. Three different system configurations were measured, a two, three and four node configuration. A high-level block diagram of the “four node” system is provided is in figure 6.13.

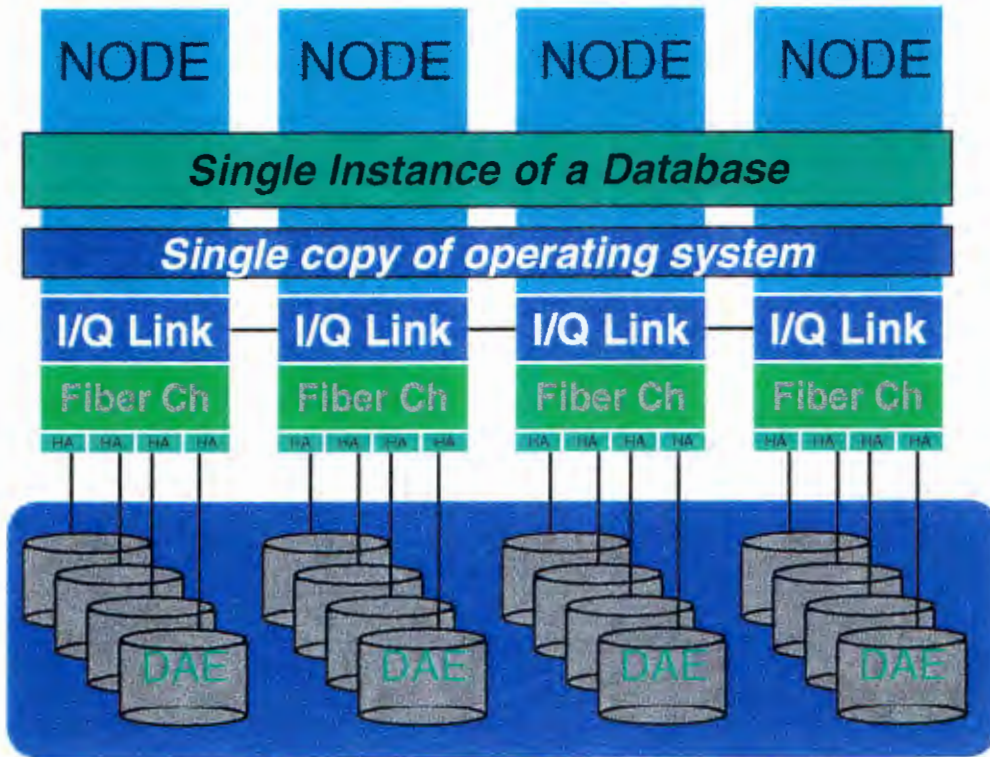


Figure 6.13 – Four Node Database System

The versions of operating system and database used for these runs were:

- Operating System - DYNIX/ptx(R) V4.4.4
- Database - Oracle's Ver. 8.0.4.1

The database configuration was changed between the system configurations (2,3, and 4 node configurations) in the attempt to better match with the system hardware configurations. (Primarily, the number of database engines was increased as nodes were added to the system.)

The system's database performance is represented in figure 6.14. It should be noted that tuning a database is a very complex and time-consuming endeavor

that was considered outside the scope of this research project. The database measurements were taken under severe time and resource constraints and are not representative of the systems real capability. The results can be used to compare the two protocols.

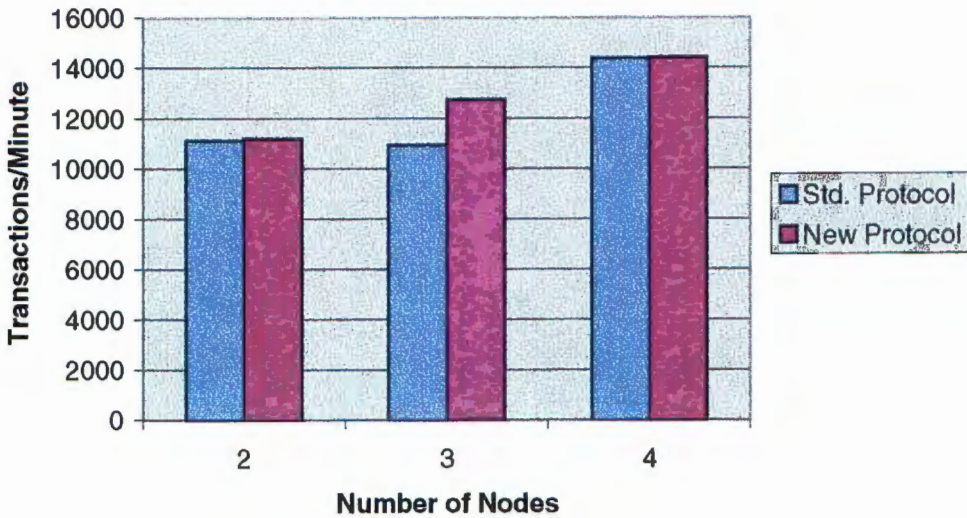


Figure 6.14 – Number of Database Transactions

The results were somewhat inconclusive in that the number of transactions per minute between the two different protocols were very similar and that the “tuning” of the database turned out to be much more complex in the three and four node systems. In just comparing the number of transactions per minute one would conclude:

- The penalty incurred in the read response time due to queuing on invalidates has no effect on the system’s overall performance.

- The Reduced List Invalidation time does not positively effect the system's overall performance.

Also in comparing the overall system performance, one should review the cache accesses of the system. For this system the cacheable accesses can be broken down into two major categories, local accesses and remote accesses. For each of these categories one must take into account reads, invalidates, and "read/invalidates".

Figure 6.15 is a comparison of the local nodes cacheable access patterns for the three system configurations. As expected, the figure 6-10 shows that local read and "read/invalidates" accesses are slightly slower (approx. ten clocks) and local invalidates are slightly faster (approx. thirty to forty clocks faster).

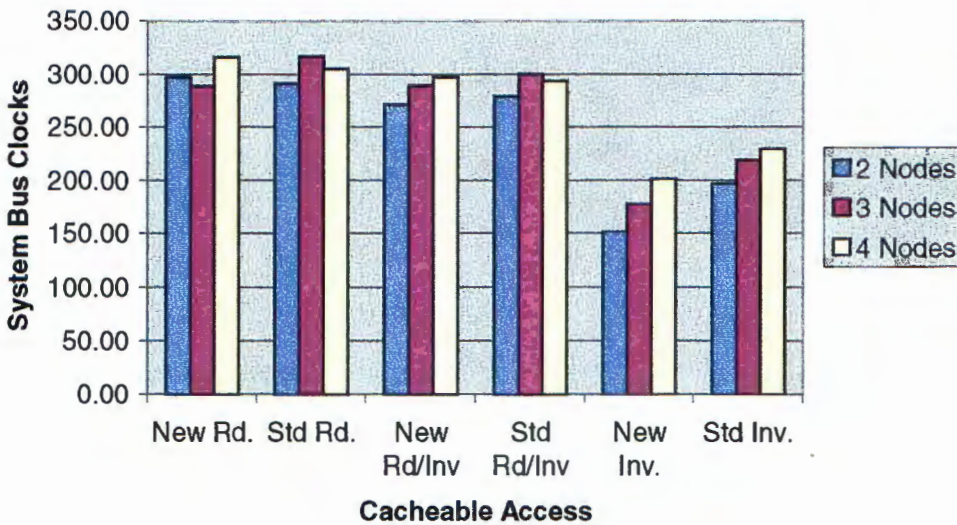


Figure 6.15 – Local Cacheable Accesses

Figure 6.16 is a comparison of the remote nodes cacheable access patterns for the three system configurations. Again as with the local accesses, the remote reads and “read/invalidates” were slightly slower (approx. ten to fifteen clocks) and invalidates were slightly faster (approx. thirty to forty clocks in the 3 and 4 node configurations).

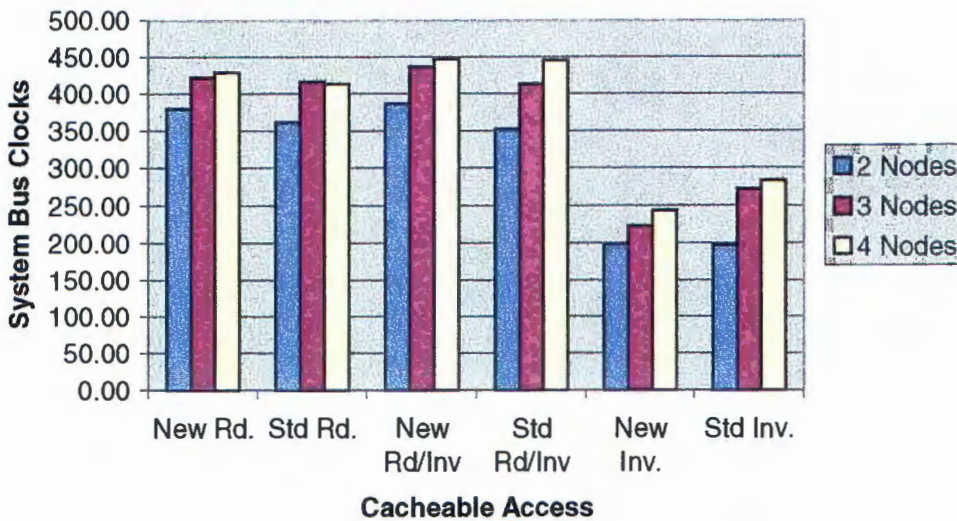


Figure 6.16 – Remote Cacheable Accesses

In analyzing the results presented between the figure 6.14 through 6.16, one can see that the difference between the two protocols from a database performance is less than one percent. Also as expected, the read latencies are slightly higher for the new protocol and the time to invalidate a sharing list is less. Analyzing the “read/invalidate” case for both the remote and local accesses, the latency for the new protocol begins to cross over (i.e. the

latency is less in the case of the new protocol). This is due to the fact that for a “read/invalidate”, there is the potential for a sharing list to exist. In that case the invalidation sequence to purge the sharing list must be performed.

Other key questions that must be analyzed are the following:

- Over a given period of time, what is the ratio of reads being issued versus invalidates being received?
- What is the duration of the active posted invalidate at a node (i.e. How long must a read response be delayed for a posted invalidate?)?

Figure 6.17 shows the contrast between the average time of SCI invalidate requests and the average time between remote cacheable read requests (this is the combined remote/local read and “read/invalidate” requests).

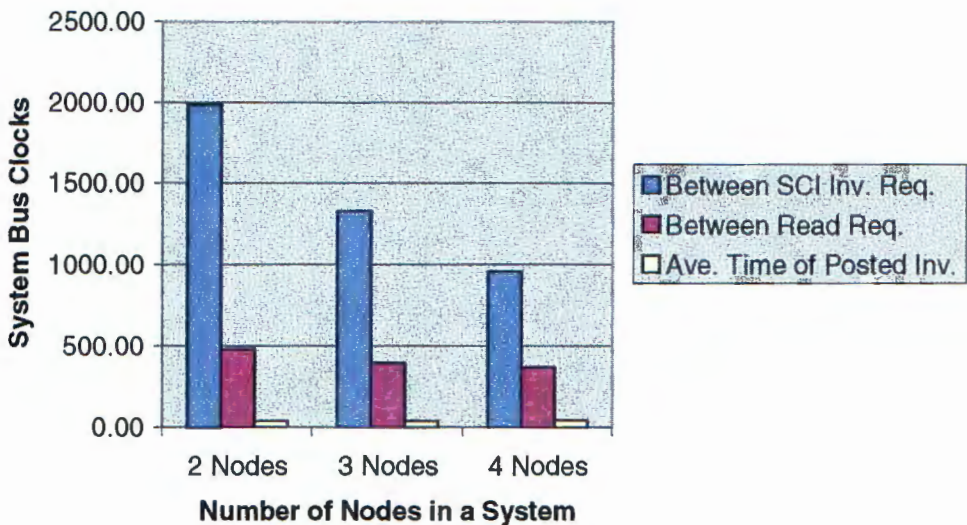


Figure 6.17 – Rd. Requests versus SCI Inval. Requests

Figure 6.17 shows that as the number of nodes increase, the average time between remote reads and SCI invalidate requests decrease (i.e. there are more of them). Also, the average time of a posted invalidate is fairly constant across all three configurations and that time is significantly shorter than the time between cacheable reads (about a factor of ten in the 4 node case).

7. Observation Section

This section contains a collection of observations and opinions that were made during the course of this research project. These observations address the issues of:

- Should this work be incorporated in future products?
- What are the issues of developing cache protocols?
- Are there any other additional areas where the Std. SCI cache protocol could be improved to reduce latency?

7.1 Performance Gains and Drawbacks

7.1.1 Ideal versus Real Performance Gain for Invalidation Extension

An observation that should be pointed out is the difference between the “ideal” performance gain of the reduced invalidate extension (represented by the logic analyzer traces in section 6.1.3.2) and the “Share List” test results (figures 6.2-6.7). The difference between the “ideal” and the “realized” is due to many causes. Some of these items that affect list invalidation time are bus utilization, cache controller utilization, memory bandwidth, remote cache tag bandwidth, and whether the cache line is highly contested. Even taking these items into account, the purging of a sharing list is consistently faster with the new protocol.

7.1.2 Elimination of Additional Ten Clock Penalty

As stated earlier, an additional read latency timing penalty of ten clocks for all transactions that are required to check on pending invalidates had to be incurred due to a flaw in the hardware of the cache controller. A description of the design flaw is as follows:

The architecture of the cache controller of the NUMA-Q system is not based on a single protocol engine, but actually two complete protocol engines. Each protocol engine has its own directory and remote cache tags. One engine only works on requests for “even” cache lines, the other on “odd”. Since the system cache line size is 64 bytes, even and odd cache lines are determined by address bit 6. The protocol engines can check to see if that particular engine has a previously set “pending Invalidate” bit(s) before queuing. However, it does not have visibility into the engine “pending invalidate” bits. The queuing logic for the protocol engine does take the “pending invalidate” bits from both protocol engines. The time for a protocol engine to queue on “nothing” and then de-queue itself is ten clocks.

In looking at the ratio of reads issued by a node and invalidates issued to a node, it is easy to see that in most cases the read response is queuing on a list of zero elements.

Since the “reduced list invalidate” extension positively affects invalidation time, the extension will be incorporated in the product line.

The hardware design flaw will be corrected in the next generation of the product to minimize the read latency penalty incurred due to this extension. The “ten clock” penalty should be reduced to “one clock” in the case of an empty queue and the time until the posted invalidates complete in the nonempty case.

7.2 Cache Coherency Validation Techniques

A great deal of time in this research project was consumed in the validation and debugging of the cache coherency protocol. Methods that were attempted were formal verification, developing tests in a simulation environment, low level diagnostics for system hardware and operation tests under an operating system.

7.2.1 Formal Verification

In developing the extensions to the SCl protocol no formal verification was done. The new protocol was validated by “inspection” only. No formal proof was developed to ensure that the extensions were deadlock free and completely coherent. An attempt was made to use the Symbolic State Model (SSM) to formally validate the cache coherency protocol [54]. The SSM methodology was specifically developed to validate complex coherency protocols that have a centrally located directory. The SSM methodology for centrally located directory structures does an excellent job in avoiding the classical validation problem of “state explosion”. However, the distributed

nature of the SCI list could not be handled via the constructs of the SSM verification tool. The resulting outcome in attempting to use the SSM tool to verify the SCI cache protocol was the classical “state explosion” (i.e. the application core dumps).

7.2.2 Simulation Environment

The initial validation for this research was done via inspection and initially a behavioral RTL simulation environment was used to test the new protocol. This environment was based around the actual “RTL” of the NUMA-Q cache controller. This was a very accurate environment, and was the method used to debug the basic attributes of the extensions. This environment identified fundamental mistakes in the implementation of the cache coherency extensions. The major drawback to this debug environment was the time to develop the simulation tests.

7.2.3 System Level Cache Coherency Tests

When the time to develop and run simulation tests became too long, the debug environment migrated to actual hardware. Initially, a “two node” system was used as the debug environment, then a three, and finally a “four node” system. Most of the implementation problems were identified during the “booting” process of a system. During the boot process, multiple processors were coming online and contending for cache base locks, capacity misses in caches were occurring in the third level cache (i.e. write-backs are occurring), and I/O devices were writing into memory and generating interrupts.

Most of the actual system level debug time was spent just attempting to boot the system. Initially, the errors in the cache protocols caused hard failures and were identified quite easily. These problems were fairly easy to identify via logic analyzer traces. As problems were identified and corrected the remaining problems became more and more obscure. As the problems became more obscure the system failures became more catastrophic during the boot process. On several occasions the system disk was unrecoverable and the entire operating system had to be re-installed (using the standard SCI protocol).

Once the new protocol survived booting a “four node” system, only two other end cases were uncovered. These end cases were uncovered by running disk, LAN, memory, and processor tests in parallel.

7.3 Additional Areas of Research Uncovered with the SCI Protocol

In developing these extensions, several other areas for performance enhancements were uncovered. Several of these areas are:

- To merge the Reduced List Invalidation extension with some “request forwarding” technique, thus eliminating the intermediate SCI response packets.
- Developing other read algorithms to provide a data response prior to completely prepping to the sharing list.
- In the area of capacity misses, to parallelize the rollout and install

operations (or develop an algorithm to do the roll out operation after the install has completed).

7.3.1 Merging the two Invalidate Extensions

Work was done in August of 1996 to develop an extension for SCI to forward the SCI Invalidate down the sharing list. This extension was referred to as the “Fast Invalidate Extension for SCI” [52]. The basic premise for the “fast invalidate” extension is to have the sharing list invalidate itself. At the highest level, all that the “fast invalidate” extension does differently than the standard SCI protocol is to forward the invalidation request packet down the sharing list, thus eliminating the sending and receiving of the intermediate response packets. In the development of this protocol an “undesirable feature” was uncovered with the hardware device that provides the physical SCI interface. The problem had to do with the part’s inability to read the status of the send queues due to a synchronization problem inside the part. If a solution can be found to this hardware limitation, it is this author’s belief that the combination of these two extensions provides the “minimum” time to invalidate an SCI. The combination of these two extensions is represented in figure 7.1

The basic flow of events in the “ideal” SCI list invalidation would be to have a node forward a request to the next node (its “forward pointer”). If the node was at the “Tail” of the list, it would issue the response packet (signaling the list is completely purged). In the case of figure 7.1, “Node Y” would immediately

forward the request to the next node on the sharing list, issue the invalidate request to its bus, and set the corresponding “pending invalidate” bit. The invalidate request on bus “Y” would overlap the SCI request to node “Z”. When node “Z” received the forward request, being the “Tail” of the list, it would issue the SCI response to node “N”, issue the invalidate to its bus, and set the corresponding “pending invalidate” bit.

Because of the “pending invalidate” bits on their corresponding nodes, the actual invalidates on “Y” and “Z” can actually happen after the acknowledgement of the invalidate request on “N” and still have the entire system remain cache coherent.

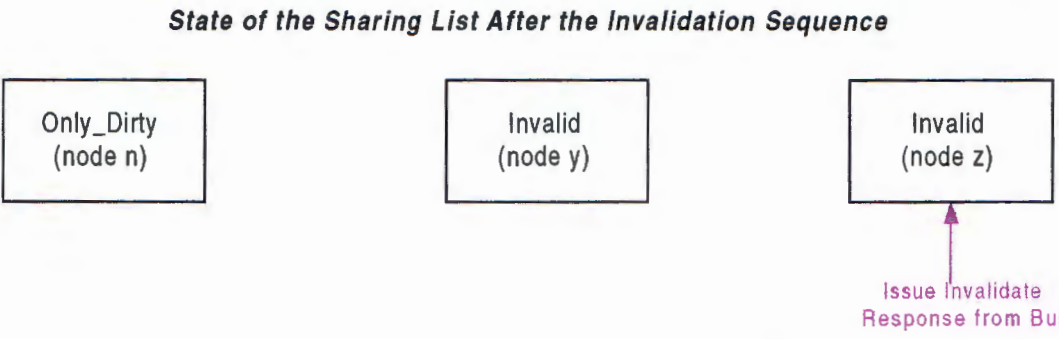
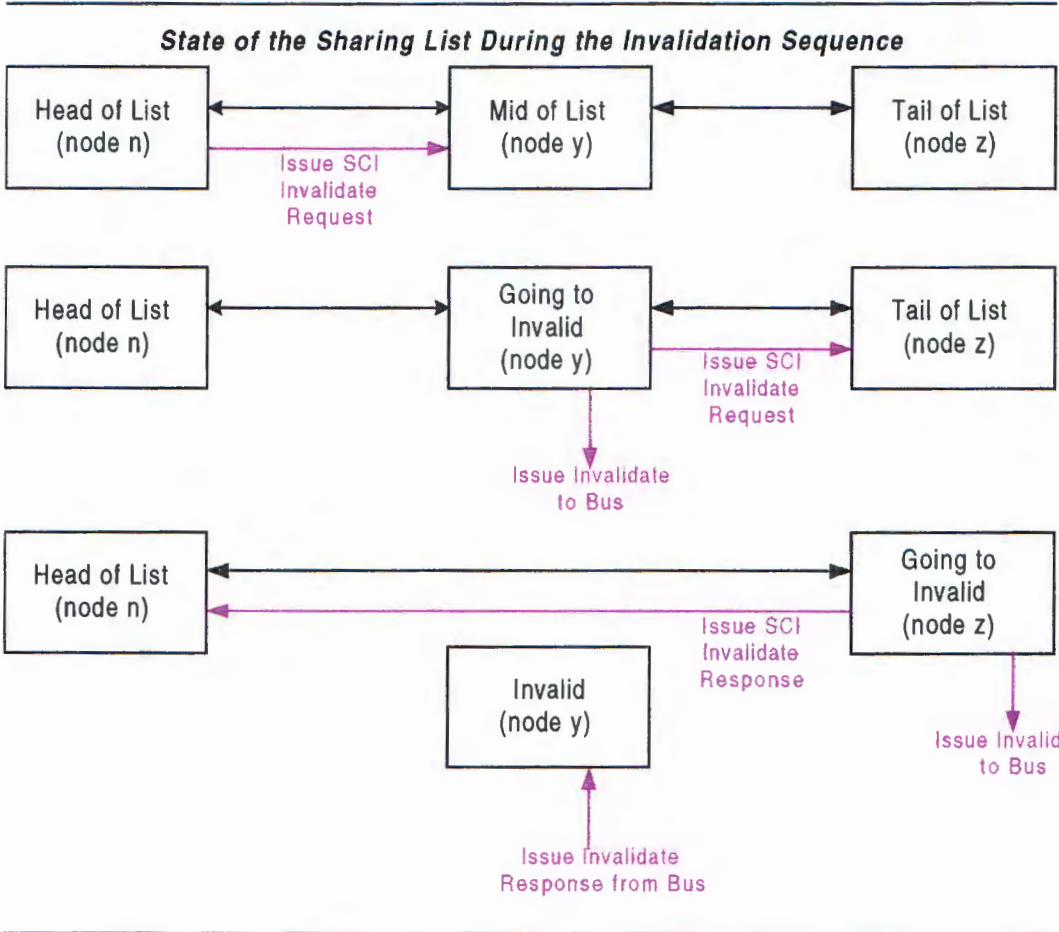
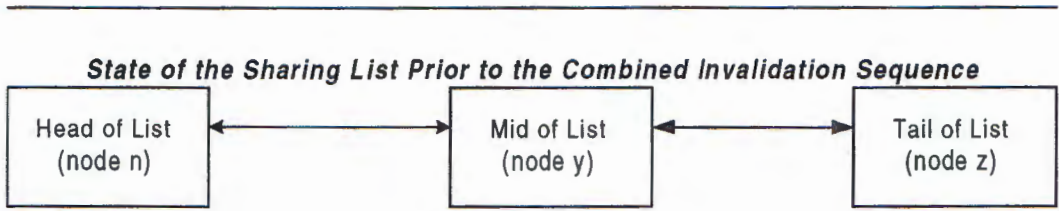


Figure 7.1 Ideal SCI List Invalidation Method

7.3.2 Reducing Processor Read Latency when prepending to a Fresh List

A possible performance enhancement in prepending to a “Fresh” List is to allow the read response to continue prior to the completion of prepending of the sharing list. It should be noted that this approach would:

- Require developing a completely different memory directory protocol.
- Require a more complex cache protocol to handle the issues of “roll out” requests prior to completely prepending to the sharing list and SCI invalidation requests during this process.
- Be mutually exclusive with the guaranteed forward progress extension.

The primary benefit of this change (if it was realizable) is actually represented in figure 6.7. This figure shows the time for the average read response with a single SCI transaction (approx. 4.5 microseconds with a “two node” system under load) and with two SCI transactions (approx. 6 microseconds with a four-node system under load). Remember in the case of a “Fresh” list, the remote node first issues a read request to the “Home” node followed by a request to the “old head” of the list to complete the list prepending process. The response for the cache line request is not issued until the prepend process is completed.

7.3.3 Parallelizing the Rollout/Installation Process

In reviewing the performance data an observation of the number of capacity misses for the remote cache were much higher than in the previous generation of the NUMA-Q system. A capacity miss currently forces the cache controller to first “roll out” the line currently in the remote cache prior to going through the installation process. A possible improvement in this area is to develop a protocol where the installation happens first and then allow for the “roll out” to happen second.

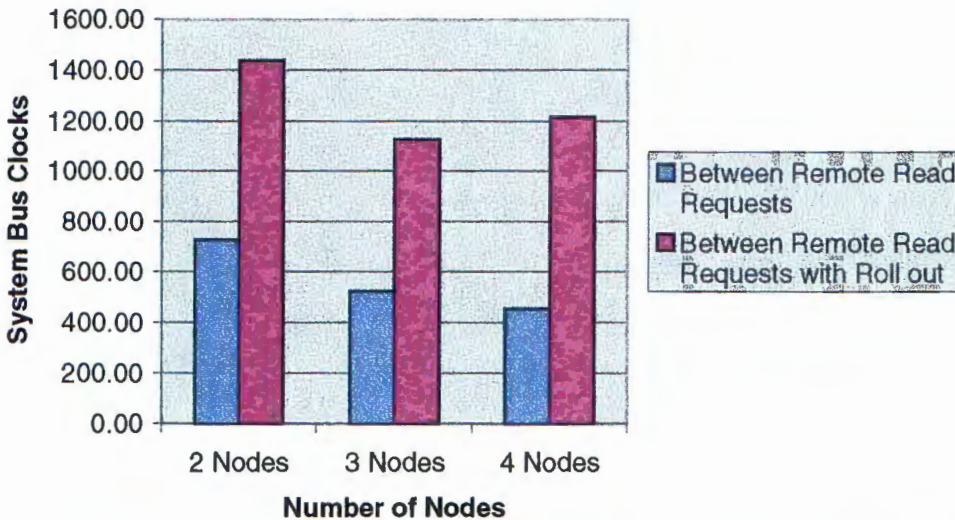


Figure 7.2 - Total Remote Cache Read Requests versus Rollout Requests

Figure 7.2 shows the average time between remote cache read (and “read/invalidate”) requests compared to the remote cache read requests that initially require a roll out prior to start of the install. In the case of the database application, every second or third remote read request on average was a

capacity miss, forcing the read to be stalled prior to the completion of the “roll out” request.

8. Summary and Conclusion

Fundamentally, this research exploits the ability of the NUMA-Q's cacheable interconnect to be "reprogrammed" resulting in the realization of a completely new CC-NUMA cache protocol. The resulting protocol has its origins in SCI but addresses several shortcomings of SCI's basic cache protocol. This research's primary focus is in the area of protocol extensions to the SCI cache protocol. Specifically, the standard SCI cache protocol was enhanced in the areas of:

- Guaranteed Forward Progress.
- Reduced List Invalidation Time.

It is the hope that these enhancements provide better characteristics for linear behavior as the system grows in the number of processors.

The primary purpose of the Guaranteed Forward Progress extension was to simplify the SCI protocol in the area of a remote node attaching to a sharing list. The flow of events with this extension is the same for either prepending to a "Fresh" or "Dirty" list. In addition to the simplification in SCI, this extension distributes the bandwidth requirements for providing the cache line equally among the "old heads". This is in comparison to the standard SCI method that stipulates that the "home" node commits the resources to provide the cache line for all requests to prepend to a "Fresh" list.

The Reduced List Invalidation Time extension does add complexity to the standard SCI protocol, but as the length of the lists grows the benefits of this approach also grow. In the case of invalidating a list of fourteen nodes the

invalidation time at the processor was reduced by almost ten microseconds (a twenty percent decrease in list invalidation time). The key component of the Reduce List Invalidation extension is the ability of the protocol engine to queue an action (like a read response) behind the completion of some specific currently active events. This ability to queue and de-queue events is employed for all cache line read responses, interrupt requests, and write-back requests. With this logic a processor of a node is prevented from:

- Observing Writes from a given processor in the wrong order.
- Having a Read passing a Write.

In preventing these situations from occurring, this extension is able to decrease the invalidation time of a sharing list while maintaining a processor consistency model.

Several methods were employed in determining the “worth” of the extensions, these being:

- An accurate Behavioral – RTL simulation environment. This environment provided an excellent debug facility, as well as accurately predicted the “ideal” performance gain (or loss) of the extensions.
- Multi-Node System with correlated logic analyzer traces. This environment provided the ability to completely debug the new cache protocol, as well as provided key insight to refine the extensions implementations.
- Performance Counters and Software. With performance counters built into all processors, memory controllers, cache controllers, and

operating system, it is technically feasible to monitor all dimensions of a system's performance while under operation. The major drawback is, of course, the magnitude of the data collected. The data must be organized in a manner so that information describing the system operation can be correctly extracted.

Using these three methods provides an excellent vehicle in determining the worth of the extensions. However, the time required to use a relational database to measure system performance was greatly underestimated. The job of "tuning" a database is a very complex and time-consuming endeavor. There are many possible reasons why a database scales poorly, other than the cache protocol of the system. Using a database that is not completely tuned to the system is not the best application metric in measuring system performance differences between cache protocols.

A considerable amount of time in of this project was spent in the area of debugging and validating that the new cache protocol was coherent and provided the correct consistency model. As stated earlier, some of this validation work was done via a simulation model of the system, some via standalone diagnostics of the system, but most was done via the boot process of the system.

The simulation environment provided the best environment to debug problems but a multi-node system simulation image is extremely large. The time to simulate a second of system run time for a four-node system would take multiple weeks to complete. The simulation environment is a very good tool to

validate the basic operations, as well as, some of the obvious end cases.

As stated earlier, the majority of the cache coherency protocol is validated during the process of booting a multi-node system. However, this mechanism is extremely poor in identifying or isolating the failing scenario. Also the negative side effect of corrupting the boot image of the system disk must be taken into account.

This work points out a need for an additional system diagnostic test in the area of testing a system's cache coherency. Unlike most diagnostic tests that focus on an individual unit's ability to perform a list of specific functions, this test would validate that the entire system performs in a cache coherent fashion.

This test, by definition, would be a multiprocessor and distributed I/O test.

This test would have the following characteristics:

- All processors and I/O devices reading and writing to local and remote memories under contention and no contention circumstances.
- All processors exercising different locking mechanisms.
- Processor and node caches (L1, L2, and L3) experiencing communication, as well as capacity misses.
- Interrupts being used during this time.

Ideally this system cache coherency test would be self-checking as the system is being exercised. This level of system testing would elevate the diagnostic step of debugging the system cache coherence protocol under the boot process. This system test would greatly accelerate the validation process of

any NUMA based cache coherency protocol and thus shortening the system development cycle time.

The cache coherence protocols are a crucial element of CC-NUMA system architecture and SCI is a common protocol used in these architectures. As the number of systems that are based on this type of architecture increase, the value in research in this area also increases.

The major value of the Guaranteed Forward Progress extension is in the area of simplification. The SCI cache protocol is not a simple protocol. The standard protocol consists of approximately thirty states. Any reduction in the number of states and commands (without loss to system performance) is a beneficial enhancement.

The major value of the Reduced Invalidation Time extension is the ability to acknowledge an invalidate request early while maintaining the system's desired memory consistency model. It must be pointed out that this extension is not limited to the SCI protocol, but is applicable to any cache protocol used in a NUMA architecture.

Currently this research has spawned two U.S. Patent Applications based this research invalidation methodology. This research is still being reviewed for other patentable ideas. These cache coherency protocol extensions are currently being considered as part of next generation computer systems developed by Sequent Computer Systems Inc. It should be noted that all new

ideas presented in this research are considered the intellectual property of
Sequent Computer Systems Inc.

References

- [1] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiawicz, B.-H. Lim, K. Mackenzie, D. Yeung. The MIT Alewife machine: Architecture and performance. In Proceedings of the 22nd International Symposium on Computer Architecture, pages 2-13, June 1995
- [2] ANSI/IEEE Std. 1596-1992, Standard for Scalable Coherent Interface (SCI) Specification, New York, New York, August, 1993.
- [3] J. Chapin, S. A. Herrod, M. Rosenblum, and A. Gupta. Memory system performance of UNIX on CC-NUMA multiprocessors. In Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, pages 1-13, May 1995.
- [4] D. Chaiken and A. Agarwal. "Software-Extended Coherent Shared Memory: Performance and Cost. Proceedings of the 21st Annual Symposium on Computer Architecture, pages 314-324, April 1994.
- [5] D. Chaiken J. Kubiawicz, and A. Agarwal. "LimitLESS Directories: A Scalable Coherence Scheme". Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 224-234, April 1991.
- [6] D. Chaiken. "Cache Coherence Protocols for Large Scale Multiprocessors". Master's thesis, M.I.T., Dept. of E.E and Computer Science, Sept. 1990.
- [7] S. Frank, H. Burkhardt III, and J. Rothnie. The KSR1: Bridging the gap between shared memory and MPPs. In Proceedings of the 38th IEEE Computer Society International Conference (Spring Comcon), pages 285 - 294, February 1993.
- [8] B. Gallagher and M. Jonikas. SQL Server 6.0: tough to top. PCWeek, page 83, vol. 12, no. 36, September 11, 1995.
- [9] J. Gray, Editor. The Benchmark Handbook for Database and Transaction Processing Systems, Morgan Kaufmann Publishers, San Mateo, CA, 1991.
- [10] E. Hagersten, A. Landin, and S. Haridi. DDM - A cache-only memory architecture. IEEE Computer, pages 44-54, vol. 25, no. 9,

September 1992.

- [11] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J. P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, J. Hennessy. The performance impact of flexibility in the Stanford FLASH multiprocessor. In Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), pages 274-285, October 1994.
- [12] C. Holt, M. Heinrich, J. P. Singh, E. Rothberg, and J. Hennessy. The performance effects of latency, occupancy and bandwidth in cache-coherent DSM multiprocessors. Presentation at The Fifth Workshop on Scalable Shared Memory Multiprocessors, Santa Margherita Ligure, Italy, June 1995.
- [13] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In Proc. of the 10th Int. Sym. on Computer Architecture, pages 124-130, (May 1983).
- [14] J. R. Goodman. Cache Consistency and Sequential Consistency, University of Wisconsin-Madison, (March, 1989).
- [15] Intel Corporation. Intel ParagonÆ Supercomputer Product Brochure. <http://www.ssd.intel.com/paragon.html>
- [16] Intel Corporation. P6 Processor - P6 Bus Analogy: Intel Web server, (<http://www.intel.com>).
- [17] Intel Corporation. P6 Processor - Supporting Greater than 4 P6s: Intel Web server, (<http://www.intel.com>).
- [18] Intel Corporation. P6 Processor - Supporting Multiple Processors: Intel Web server, (<http://www.intel.com>).
- [19] Intel Corporation. P6 Processor - Bus Protocol : Intel Web server, (<http://www.intel.com>).
- [20] Intel Corporation. P6 Processor - P6 Bus Summary: Intel Web server, (<http://www.intel.com>).
- [21] D. James. "The Scalable Coherent Interface: Scaling to High Performance Systems".
- [22] D. James. "Coherent SCI Memory", P1596.2 working-group

activity, (March 1994).

- [23] D. A. Kranz, D. Chaiken and A. Agarwal. "Multiprocessor Address Tracing and Performance Analysis". MIT VLSI Memor No. 91-624, Sept. 1990.
- [24] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In Proceedings of the 21st International Symposium on Computer Architecture, pages 302-313.
- [25] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Logic overhead and performance. IEEE Transactions of Parallel and Distributed Systems, pages 41-61, vol. 4, no. 1, January 1993.
- [26] T. Lovett and S. Thakkar. The Symmetry multiprocessor system. In Proceedings of the 1988 International Conference on Parallel Processing, pages 303-310, August 1988.
- [27] Pyramid Technology. ReliantÆ 1000,
<http://ra.pyramid.com/products/1.1.1.1.html>
- [28] R. J. Safranek. "Considerations in Implementing a System Based on SCI", In The Fourth International Workshop on SCI-based High Performance Low-Cost Computing, pages 12-22, (October 1995).
- [29] H. Sandu, B. Gamsa and S. Zhou. "The Shared Regions Approach to Software Cache Coherence on Multiprocessors", 1993 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, May 1993.
- [30] A. Saulsbury and A. Nowatzky. Implementing simple COMA on S3-MP. Presentation at The Fifth Workshop on Scalable Shared Memory Multiprocessors, Santa Margherita Ligure, Italy, June 1995.
<http://playground.sun.com:80/pub/S3.mp/simple-coma/isca-95/present.html>
- [31] P. Stenstrom, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In Proceedings of the 19th International Symposium on Computer Architecture, pages 80-91, May 1992.
- [32] Sequent Computer Systems, Inc. Symmetry 5000 Series.

[http://www.sequent.com:
80/public/mktg/ds/symmetry/s5000.html#topdoc](http://www.sequent.com:80/public/mktg/ds/symmetry/s5000.html#topdoc)

- [33] Transaction Processing Performance Council. TPC Benchmark Specifications. [ftp.dg.com:tpc/benchmark_specifications/TPC_A, TPC_B, TPC_C, TPC_D](ftp.dg.com:tpc/benchmark_specifications/TPC_A,TPC_B,TPC_C,TPC_D).
- [34] Tving, Ivan. "Multiprocessor Interconnections Using SCI", Technical University of Denmark, (Feb. 2, 1994).
- [35] A. W. Wilson, Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In Proceedings of the 14th International Symposium on Computer Architecture, pages 244-253, June 1987.
- [36] X. Zang and Y. Yan. "Comparative Modeling and Evaluation of CC-NUMA and COMA on Hierarchical Ring Architectures", IEEE Transaction on Parallel and Distributed Systems.
- [37] A. Agarwal, D. Chaiken, G. D'Souza, et al. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. Technical Report MIT/LCS Memo TM-454, Laboratory for Computer Science, Massachusetts Institute of Technology, 1991.
- [38] R. Bianchini, M. E. Crovella, L. Kontothanassis, and T. J. LeBlanc. Memory contention in scalable cache-coherent multiprocessors. Technical Report 448, Computer Science Department, University of Rochester, 1993.
- [39] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott. Simple but effective techniques for NUMA memory management. In Proc. of the 12th ACM Symp. on Operating System Principles, pages 19-31, 1989.
- [40] D. Chaiken, J. Kubiawicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In Proc. of the Fourth Int'l Conf. on ASPLOS, pages 224-234, New York, April 1991.
- [41] Convex Computer Corporation. Convex Exemplar Systems Overview, 1994.
- [42] K. Farkas, Z. Vranesic, and M. Stumm. Scalable cache consistency for hierarchically-structured multiprocessors. Journal of Supercomputing, 1995. in press.
- [43] Kendall Square Research. KSR1 Technical Summary, 1992.

- [44] J. Kuskin, D. Ofelt, M. Heinrich, et al. The Stanford FLASH multiprocessor. In Proc. of the 21st Annual ISCA, pages 302-313, Chicago, Illinois, April 1994.
- [45] R. P. LaRowe Jr. and C. S. Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. ACM Transactions on Computer Systems, 9(4):319-363, Nov. 1991.
- [46] D. Lenoski, J. Laudon, K. Gharachorloo, et al. The Stanford DASH multiprocessor. Computer, 25(3):63-79, March 1992.
- [47] D. E. Lenoski. The design and analysis of DASH: A scaledirectory-based multiprocessor. Technical Report CSL-TR-92-507, Stanford University, January 1992.
- [48] S. Mori, H. Saito, M. Goshima, et al. A distributed shared memory multiprocessor: ASURA - memory and cache architectures -. In Supercomputing '93, pages 740-749, Portland, Oregon, November 1993.
- [49] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In Proc. of the 21st Annual ISCA, pages 325-336, Chicago, Illinois, April 1994.
- [50] University of Toronto .NUMAchine,
<http://www.eecg.toronto.edu/EECG/RESEARCH/ParallelSys/numachine.html>
- [51] SV. Soundararajan, M. Heinrich, B. Verghese, K. Gharachorloo, A. Gupta, and J. Hennessy. Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors.
- [52] R. J. Safranek. "Fast Invalidate Extension for the Scalable Coherent Interface" a paper developed for PSU credit for EE-501, Portland State University, August 1996.
- [53] R. J. Safranek. "Considerations in Implementing a System Based on SCI", The Sixth International Workshop on SCI-based High-Performance Low-Cost Computing, pages 25-30, Santa Clara California, September 1996.
- [54] Fong Pong. Symbolic State Model A new Approach for the Verification of Cache Coherence Protocols. Doctoral Thesis of the University of Southern California, 1995.

Appendix - Glossary of SCI Terminology

List - The mechanism to track which nodes of a system have a particular cache line. Each node contains two pointers for each cache line. One points toward the tail (forward pointer), the other points toward the head of the list (backward pointer).

Dirty List - A List of nodes that share a line in which the memory (or home) node does not hold a valid copy.

Fresh List - A List of nodes that share a line in which the memory (or home) node contains a valid copy.

SCI Memory States - Every node that contains cacheable memory must manage the state and pointer of each line. The memory states for SCI are as follows:

Home - Memory is home and there is no sharing list.

Fresh - Memory contains a valid copy and the line is shared with at least one other node.

Gone - Memory does not contain a valid copy of the line. To get a valid copy, a node prepending to the list must get a copy from the head of the dirty list.

Busy - Memory is in the process of having a line updated to the fresh state and immediately back to a Gone State.

SCI Memory Commands - To access and/or change the memory state at the home node, SCI memory commands are used. The following is the

subset of commands that are necessary in merging SCI and MESI:

Cache_Fresh - Remote node is requesting a readable copy of a line.

Cache_Dirty - Remote node is requesting an exclusive copy of a line.

Fresh_To_Home - Remote node, which was in Only_Fresh, is rolling out a line and sending it home.

List_To_Home - Remote node, which was in Only_Dirty, is rolling out a line and sending it home.

List_To_Gone - Remote node, which is Head of a Fresh list, is requesting the line transition to "Gone". The Home node also invalidates its copy of the line.

Pass_Head - Remote node, which is Head of a list (either Dirty or Fresh), is rolling out its copy of the line and assigning another remote node as the new "Head of the List".

SCI Cache States - Every node that holds or manipulates a cache line must manage the cache state and its forward and backward pointers. The following is the common subset of SCI cache states:

Invalid - Line at this place in the Remote Cache is Invalid.

To_Invalid - Line at this place in the Remote Cache is Invalid, but currently has a request with no response.

Pending - Remote node has issued a Cache_Fresh or Cache_Dirty to the Home node.

Queued_Dirty - Remote node has issued a Cache_Dirty for a line that currently has a sharing list that must be invalidated. This state is left when the sharing list is completely invalidated.

Queued_Fresh - Remote node issued a Cache_Fresh for a line that

currently has a fresh list and, therefore, the node must prepend to the head of the list.

Queued_Junk - Remote node issued either a Cache_Fresh or Cache_Dirty for a line that is currently dirty (modified). It must issue a Copy_Valid command to the head of the list to get a valid copy of the line and prepend to the head of the list.

Only_Dirty - Remote node has the only valid copy of the line which has been modified.

Only_Fresh - Remote node, as well the home node, have a shared copy of an unmodified line.

OF_Mods_OD - Remote node (which was Only_Fresh) intends to modify the line and sends a List_To_Gone command to the home node so that copy of the line can be invalidated.

OF_Retrn_In - Remote node is either rolling out a line while it was in the Only_Fresh state or it was in OF_Mods_OD and its List_To_Gone command was NOOP'ed by the home node.

OD_Retrn_In - A Remote node is rolling out a line that was in an Only_Dirty state.

OD_Spin_In - A Remote node is getting off a list and is waiting for the new head to prepend.

Head_Dirty - Remote node is head of list where the home node does not have a valid copy of the line.

Head_Fresh - Remote node is head of list where the home node does have a valid copy of the line.

HD_Inval_OD - Remote node is head of a Dirty List and intends to

modify the line again, so it is invalidating the sharing list.

HF_Mods_HD - A Remote node (which was Head_Fresh) intends to modify the line and sends a List_To_Gone to the home node so that copy of the line will be invalidated.

HX_Forw_HX - Remote node that was the head of the list is getting off the list.

HX_Forw_OX - Remote node that was the head of the list is getting off a list which is collapsing.

HX_Retrn_In - Remote node that was the head of the list is getting off the list and is waiting for the new head (which is in a queued state) to prepend.

Mid (Mid_Valid, Mid_Copy) - Remote node is in a sharing list below the head and above the tail of the list.

MV_Forw_MV and MV_Back_In - Remote node that was in the middle of the list is getting off the list (either because the node is rolling out the line, or because the node intends to become head of the list).

Tail (Tail_Valid, Tail_Copy) - Remote node is the last node on a sharing list.

TV_Back_In - Remote node, which was tail of the list, is getting off the list (either because the node is rolling out the line, or because the node intends to become head of the list to modify the line).

SCI Cache Commands - To gain access to a cached line and/or to change state of a cached line at a node, SCI cache commands are used. The following is the standard subset of commands for SCI:

Pend_Valid - Remote node command requesting to prepend to a fresh sharing list.

Copy_Valid - Remote node command requesting to prepend to a dirty sharing list and a valid copy of the line.

Valid_Invalid - Remote node (that is head) requesting other remote nodes to invalidate their copy of the line.

Prev_VTail - Remote node (that was tail) notifying the node immediately preceding that it is the new tail.

Prev_VMid - Remote node (that was Mid) notifying the node immediately ahead to update its Back_ID to maintain the linked list.

Next_VMid - Remote node (that was MV_Forw_MV) notifying the node immediately following to update its Forw_ID to maintain the linked list.

Next_DHead - Remote node (which was Head_Dirty) notifying the preceding node that it is the new head of the dirty list.

Next_FHead - Remote node (which was Head_Fresh) notifying the preceding node that it is the new head of the fresh list.

NOOP - An SCI term for what the rest of the computer world refers to as a NACK.