Example-standard contents

These cover sheets provide information, reference names, and commonly-used text for use by other files within a containing FrameMaker® document. These pages are not expected to be included in the final document; they can be discarded after printing or the book can be modified to exclude this *StdCover* file from the printing process.

These files and the associated book are templates for drafts of IEEE Standards (or for other documents that choose to mimic this style). Editors of an IEEE document are encouraged to use these files as an initial basis for generation of a publishable-quality IEEE standards document. Since formatting styles are subject to change, the IEEE should be consulted for the most recent style changes.

The reference book contains several files, as follows:

StdCover	— the contents of this file
StdFront	— the cover page file, with acknowledgments and background
StdTOC	— table of contents
StdLOF	— list of figures
StdLOT	— list of figures
StdSec1	— first section of text
StdSec2	— second (or additional) section of text
AnnexA	— annex section

The StdSec1 and StdSec2 files share styles, but the first section has a unique first-page header (white space only) to correspond to the final page of the preceding introductory material.

The annex section is different, in that different styles are used for section headings and figure numbering. Thus, sections begin with "Annex A: Extra material" rather than "5. Extra material." The full set of paragraph styles should not be blindly copied between sections and the Annex documents, since the following have been customized for the Annex:

H1...H2	Section headings
D2...D5	Definition headings
TableTitle	Table title
FigTitle	Figure title

Common styles, which may be used by the main sections as well as the Annex, should be located within this StdCover document. These styles should have their common home within this reference. They can be freely imported into other affected body and Annex sections.

To avoid style pollution, we have only included the known-functionality styles within this file. Other styles may be added after their functionality has been understood and documented.

If you intend to use this template for a standards-like document, your primary task is to update the text sections on the following reference sheet. Other changes to text within the StdFront document are also in order, since it contains IEEE copyright and procedure statements. Of course, the text is also likely to change.

The final page in the StdLOT list of tables document may be blank or may be filled with part of the list of tables. If it is blank, manually assign the LastLeftEmpty master page to it.

Reference names

Locating reference names on one printable page simplifies resolution of inconsistent definitions. Items that are commonly referenced in the remainder of the document are included on this page. A unique style name, *RefTarget*, allows these names to be readily crossreferenced in other book-affiliated files. Each reference name follows a descriptive line, whichis in italics.

Draft information, which is not included within the final draft:

Following "Draft n.nn" string should be a blank space in final draft:
Draft 0.36
Following "Month dd, 199x" should be a blank space in final draft:
November 18, 1996

Title information, which runs across the top of adjacent pages:

Following left-page title, continues across two pages:
IEEE STANDARD FOR CACHE OPTIMIZATIONS FOR LARGE NUMBERS
Following right-page title, continues across two pages:
OF PROCESSORS USING THE SCALABLE COHERENT INTERFACE (SCI)

Standard number, included on the outside top of each page:

Following is standards-body label, top of each page:
IEEE
Following is standard's number, top of each page:
P1596.2-1996

Standards board reference lead-in, change to "Approved by the" when completed:

Following line precedes statement of "IEEE Standards Board".
Not Yet Approved by the
ISBN number, appears on the cover sheet, and should be replaced with proper number.
1-55937-xxxx

Copyright notice, printed at the bottom of each page:

Following is 1st line of copyright notice, included at bottom of each page.
Copyright 1994-1996 IEEE. All rights reserved.
Following is 2nd line of copyright notice, included at bottom of each page.
This is an unapproved IEEE Standards Draft, subject to change.

Reference text

Frame recommends that commonly used references be included within the master pages, so they can be readily copied into new documents. We do not recommend this practice, as it increases the number of master pages. The propagation of multiple copies of master pages as the number of sections and document files increases makes it harder to maintain consistent update to such material. Instead, we recommend that the common references be included within this cover document, as is done here.

AEIC publications are available from the Association of Edison Illuminating Companies, 600 N. 18th Street, P. O. Box 2641, Birmingham, AL 35291-0992, USA.

ANSI publications are available from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

API historical materials can be obtained (for a fee) from the American Petroleum Institute Library, 1200 L Street NW, Washington, DC 20005, USA.

API publications are available from the Publications Section, American Petroleum Institute, 1200 L Street NW, Washington, DC 20005, USA.

ASME publications are available from the American Society of Mechanical Engineers, 22 Law Drive, Fairfield, NJ 07007, USA.

ASTM publications are available from the Customer Service Department, American Society for Testing and Materials, 1916 Race Street, Philadelphia, PA 19103, USA.

AWEA publications are available from the American Wind Energy Association, Standards Program, 777 North Capital Street NE, #805, Washington, DC 20002, USA.

CSA publications are available from the Canadian Standards Association (Standards Sales), 178 Rexdale Blvd., Rexdale, Ontario, Canada M9W 193.

CCITT publications are available from the CCITT General Secretariat, International Telecommunications Union, Sales Section, Place des Nations, CH-1211, Genève 20, Switzerland/Suisse.

CFR publications are available from the Superintendent of Documents, US Government Printing Office, P.O. Box 37082, Washington, DC 20013-7082, USA.

CISPR documents are available from the International Electrotechnical Commission, 3 rue de Varembé, Case Postale 131, CH 1211, Genève 20, Switzerland/Suisse. CISPR documents are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

EIA publications are available from Global Engineering, 1990 M Street NW, Suite 400, Washington, DC, 20036, USA.

ICEA publications are available from ICEA, P.O. Box 411, South Yarmouth, MA 02664, USA.

ICRU publications are available from the National Council on Radiation Protection and Measurements, 7910 Woodmont Avenue, Suite 800, Bethesda, MD 20814, USA.

IEC publications are available from IEC Sales Department, Case Postale 131, 3 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse. IEC publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

IEEE publications are available from the Institute of Electrical and Electronics Engineers, Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

ISO publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse. ISO publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

JEDEC publications are available from JEDEC, 2001 I Street NW, Washington, DC 20006, USA.

MIL publications are available from the Director, U.S. Navy Publications and Printing Service, Eastern Division, 700 Robbins Avenue, Philadelphia, PA 19111, USA.

NBS publications are available from the Superintendent of Documents, US Government Printing Office, P.O. Box 37082, Washington, DC 20013-7082, USA.

NCRP publications are available from the National Council on Radiation Protection and Measurements, 7910 Woodmont Avenue, Suite 800, Bethesda, MD 20814, USA.

NEMA publications are available from the National Electrical Manufacturers Association, 2101 L Street NW, Washington, DC 20037, USA.

NFPA publications are available from Publications Sales, National Fire Protection Association, 1 Batterymarch Park, P.O. Box 9101, Quincy, MA 02269-9101, USA.

NUREG publications are available from the Superintendent of Documents, US Government Printing Office, P.O. Box 37082, Washington, DC 20013-7082, USA.

UL publications are available from Underwriters Laboratories, Inc., 333 Pfingsten Road, Northbrook, IL 60062-2096, USA.

US Regulatory Guides are available from the Superintendent of Documents, US Government Printing Office, P.O. Box 37082, Washington, DC 20013-7082, USA.

This authorized standards project was not approved by the IEEE Standards Board at the time this went to press. It is available from the IEEE Service Center. [NOTE: the reference must include the P-number, title, revision number, and date.]

ANSI XXX-19XX has been withdrawn; however, copies can be obtained from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

The numbers in brackets, when preceded by the letter B, correspond to those in the bibliography in Section XX.

The numbers in brackets correspond to those of the references in XX.

IEEE Std XXX-19XX has been withdrawn; however, copies can be obtained from the IEEE Standards Department, IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

As this standard goes to press, IEEE Std SSS-199X is not yet published. It is, however, available in manuscript form from the IEEE Standards Department, (908) 562-3800. Anticipated publication date is XXX 199X, at which point IEEE Std XXX-199X will be available from the IEEE Service Center, 1-800-678-4333.

This standard will be available from the Institute of Electrical and Electronics Engineers, Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA, in early 199X.

IEEE Standard for Cache Optimization for Large Numbers of Processors using the Scalable Coherent Interface (SCI)

Draft 0.36
November 18, 1996

Sponsor

Microprocessor and Microcomputer Standards Subcommittee of the IEEE Computer Society

Not Yet Approved by the

IEEE Standards Board

Abstract: With the ANSI/IEEE Std 1596-1992 Scalable Coherent Interface (SCI), a few or many processors can share cached data in a coherent fashion (i.e. their caches are transparent to software). Scalability constrains the protocols to rely on simple request-response protocols, rather than the eavesdrop or 3-party transactions assumed by (unscalable) bus-based systems.

The linear nature of the base SCI protocols limits their performance when data is being actively shared by large numbers of processors. To meet these needs, this standard defines a compatible set of extensions based on distributed binary trees. Scalability includes optionality: simple and/or specialized noncoherent systems are not affected by the costs of extended coherence protocols.
Keywords: Scalable Coherent Interface, cache, coherence, scalable, multiprocessors

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

1-55937-xxxx

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

IEEE Standards documents are adopted by the Institute of Electrical and Electronics Engineers without regard to whether their adoption may involve patents on articles, materials, or processes. Such adoption does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the standards documents.

Introduction

(This introduction is not a part of IEEE P1596.2-1996, IEEE STANDARD FOR CACHE OPTIMIZATIONS FOR LARGE NUMBERS OF PROCESSORS USING THE SCALABLE COHERENT INTERFACE (SCI).)

Special thanks to Dave Gustavson (the SCI Chair), whose hard work and integrity has been a role model for us all. Guri Sohi proposed coherent combining and the concept of sharing trees. Ross Johnson was the P1596.2 (SCI Coherence Extensions) Chair and defined their subtree-merge, coherent fetch&add, and cache-line replacement protocols. Stein Gjessing (the current P1596.2 Chair) and Jim Goodman provided academic insights and graduate-student assistance throughout the SCI development. Ernst Kristiansen and Mike Chastain's experiences with commercial systems helped us focus on practical solutions to real-world problems. A large number of unnamed volunteers also contributed to the P1596.2 development.

The Sponsor of the IEEE P1596.2 Working Group, the Microcomputer Standards Committee, had the courage to sponsor leading-edge, as well as pragmatic, standards developments. Their sponsorship of the SCI standard provided the catalyst for bringing together scientists and engineers from the international university and industry communities.

Comments on this document or questions on editing of standards drafts should be addressed to the IEEE Standards Editor:

Kristin Dittmann
IEEE Standards Office
P.O.Box 1331
445 Hoes Lane
Piscataway, NJ 08855-1331
Phone: 1-908-562-3830
Fax: 1-908-908-562-1571
Email: kdittmann@stdsmail.ieee.org

Committee Membership

The following is a list of participants in the IEEE Project 1234.5 Working Group. Voting members at the time of publication are marked with an asterisk (*).

George X. Washington, *Chair**
John Q. Adams, *Vice Chair**

A	B	C

The following persons were on the balloting committee that approved this document for submission to the IEEE Standards Board:

A	B	C

When the IEEE Standards Board approved this standard on Month dd, 199x, it had the following membership:

Also included are the following nonvoting IEEE Standards Board liaisons:

Kristin M. Dittmann
IEEE Standards Project Editor

Contents

Contents

CLAUSE PAGE

IEEE Standard for Cache Optimization for Large Numbers of Processors using the Scalable Coherent Interface (SCI)

1. Introduction

1.1 Scope and purpose

The IEEE-1596-1992 (SCI) standard specifies a coherence protocol that works with large numbers of processors. However, these protocols have performance limitations when the number of processors actively sharing the data becomes very large. There is a need to develop compatible extensions to the SCI coherence protocols that reduce the data-access latencies from order N to order log(N), where N is the number of participating processors.

This proposed standard defines protocols for reducing the latency of accessing cached data shared by large numbers of processors (referred to as widely shared data). Two sets of extensions are defined, each optimized for different environments.

The first set (STEM), optimized for tightly coupled Massively Parallel Processor (MPP) computers, involves combining multiple coherent requests into one before they reach a shared memory controller. These protocols will generate tree-like sharing-list structures compatible with the linear structures defined by SCI. These protocols will also support efficient distribution of data (when many processors read shared data) and purging of stale copies (when the previously shared data is written). The data distribution protocols will include support for combinable operations (such as Fetch&add).

The second set of extensions (GLOW) is designed to support large loosely coupled Networks Of Workstations (NOW) interconnected by bridged SCI rings. The extensions of this set are implemented entirely in the bridges (called GLOW agents) and fully support the SCI protocols as defined in IEEE-1596-1992. GLOW extensions provide for scalable reads and writes by building sharing trees out of SCI-lists.

SCI is an invalidation based cache coherence protocol. This document specifies how a sharing tree can be used as a basis for a write-update extension. This write update mechanism can be used when such a coherence protocol is wanted, and more specific for special purpose operations, specifically barrier synchronization. With this addition, SCI has very efficient hardware support for both mutual exclusion (the QOLB mechanism) and barrier synchronization.

1.2 Problem Constraints

When many processors read the same data at the same time, list creation and data distribution is sequential. When one processor writes data that is actively shared, list purging is sequential. The objective of the performance extension is to parallelize these sharing-list operations to reduce the latency of reads and writes while limiting the interconnect traffic.

Tree creation, data distribution, and tree purging must be efficient for the following operations:

a) Reads. Many processors nearly-simultaneously read the same line of data.
b) Writes. One processor writes the data that is read-shared by many others.
c) Rollouts. One processor invalidates its own copy of the data. This should not significantly affect the performance of later reads or writes by other processors.
d) Associative Updates. Many processors update a common address in a combinable fashion (such as fetch-and-add).
e) Barrier Synchronization. Many processors reach a barrier, past which none can advance until all have reached the barrier.
f) Write Update. The Read/Write scenario is such that write update is more efficient than invalidate on write.

For sharing list of size N, the extensions intend to reduce the O(N) latency (measured in number of transactions on the critical path) of the above operations to O(log N). This latency reduction is subject to the following constraints:

a) Traffic. The total number of transactions should not increase by more than a small constant factor. No finite queue in the system should simultaneously hold more than a small constant number of combinable subactions (similar operations, like reads, on the same cache line or memory address).
b) Complexity. The amount of additional memory/cache tag storage should be minimized.
c) Verifiability. The performance extensions leverage and extend the proofs of correctness developed for the basic SCI cache-coherence specification. In particular, the performance extensions must work correctly, guarantee forward progress (in spite of finite queues), and be able to recover from arbitrary transmission failures.
d) Compatibility. Packet sizes and formats should remain the same. New processors that implement these extensions should be compatible with old processors or memory that do not.

1.3 Scalability

A scalable architecture remains stable over a wide range of design parameters, such as cost, performance, distance, or time. Scalable systems eliminate the need to redesign components to meet minor or dramatic changes in product requirements. Unfortunately, the term “scalable” has become an industry buzz-word; it’s included in marketing literature but has had minimal impact on product designs.

The IEEE Std 1596-1992 Scalable Coherent Interface (SCI) working group had the charter and organizational support to develop a scalable system interconnect. The lifetime of a scalable system was defined as 10-20 years, a design lifetime which exceeded the maturation time of our young children. Working-group members could thus avoid the “Daddy, why did you ...” telephone questions from our children taking their first graduate-school courses.

Within any scalable system design, there is usually an overriding objective that cannot be compromised. For example, on the Serial Bus P1394 working group [Serial], the objective was low cost. Within the SCI standard [SciStd], *high performance* was our primary objective. We believed the highest-performance systems would not be uniprocessors, but would be constructed from large numbers of multiprocessors sharing distributed memory and using caches to hide latency.

We assumed that, for software convenience, these multiprocessors would be tightly coupled: memory can be shared and the caches are transparent to software. Tightly-coupled multiprocessors mandated the development of cache coherence protocols for massively-parallel-processor (MPP) configurations.

We felt that MPP systems would be based on the workstation of today and (perhaps) the video-game components of tomorrow. Thus, the SCI protocols should be cost-effective for noncoherent uniprocessor and small coherent multiprocessor systems as well as for MPPs.

1.4 Eavesdrop coherence protocols

Traditional cache-coherence protocols are based on *eavesdropping* bus transactions. When a data value is read from memory, other processors eavesdrop and (if necessary) provide the most up-to-date copy. When a data value is stored into CPU_C's cache, the read-exclusive transaction (a read with an intent to modify) is broadcast. Other processors are required to invalidate stale copies and (if memory has a stale copy) may also be required to provide the new data, as illustrated in figure 1.

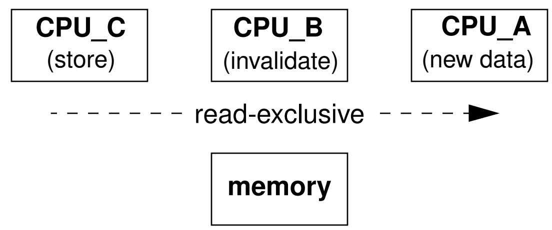

Figure 1—Eavesdrop coherence protocols

Eavesdrop-based protocols are cost-effective on unified-response buses, since the coherence checks can be performed while the data is fetched from memory. Eavesdropping is harder on high-speed split-response buses, where the transaction-issue rates are larger, since they are no longer constrained by memory-access-time latencies.

1.5 Directory protocols

Since each processor can consume a significant fraction of interconnect-link bandwidth, any MPP system would have multiple concurrently active data paths, as illustrated in figure 2. Fully-connected cross-bar topologies scale poorly (from a cost perspective), so the interconnect was assumed to be more generally switch based.

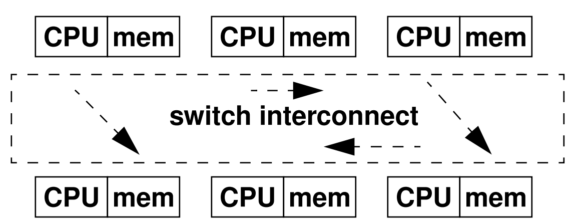

Figure 2—Switch-based systems

Eavesdropping is not possible within switch-based systems, since concurrently active transactions cannot be observed by all. Recent research [Caches] has therefore focussed on the development of directory-based protocols.

Central-directory coherence protocols supplement each line of memory with a tag, where the tag identifies all processors with cached copies. The memory and cache lines are typically the same size, between 32 and 256 bytes. When data is written, memory is responsible for updating or invalidating the previously shared (and now stale) copies.

Central directories have scaling limitations, because the size of the total system is limited by the size of the memory-line tag. Some systems[DashDir][DashSys] propose to use multicast or broadcast transactions when their directories overflow. Other systems[Limit] propose using memory-resident software to handle overflow, eliminating the need to support these special transactions.

Central-directory schemes serialize read and write transactions at memory. SCI avoids tag-storage overflow and memory bottlenecks by distributing the directories: the memory tags contains state and a pointer to the first processor; the cache tags contain state and pointers to other processors.

1.6 Transaction-set constraints

1.6.1 Basic SCI transactions

To simplify the design of high-performance switches, communication between SCI components uses a simple request/response transaction protocol. A transaction consists of request and response subactions. The request subaction transfers an address and command; the response subaction returns status.

For a write transaction, data is included within the request packet. For a read transaction, data is included within the response packet. For a compound transaction (such as fetch&add), data is included within the request and response packets.

Each subaction consists of a send and an echo packet. Information (including commands, status, and data) is transferred within the send packet; flow control information (busy retry) is returned within the echo packet, as illustrated in figure 3.

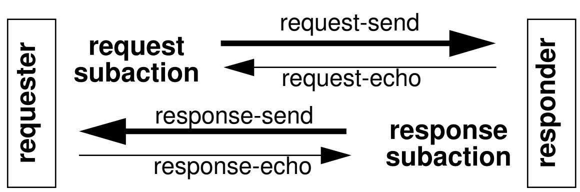

Figure 3—Request/response transactions

These are similar to the phases observed on a split-response backplane: the echo is a packetized equivalent of a busy-retry backplane signal.

Most of the noncoherent and all of the coherent transactions have these components. We avoided using special coherence transactions (such as multicast or three-way transactions), since these would have complicated the design (thereby reducing the performance) of the interconnect.

1.6.2 Coherent SCI transactions

For read and write transactions, the coherence protocols add a few bits in the request-send packet and a few bytes in the response-send packet. Since alignment constraints fix the size of send packets, the coherence protocols have no impact on the basic packet sizes.

However, an extended request is needed to support cache-to-cache transactions, since two addresses are involved (the memory address of the data and the routing address of the cache). The additional information is contained within an extension to the normal packet header.

Coherence bits within packets are typically ignored by the interconnect and have no impact on the design of basic SCI switches. Thus, the coherence protocols are scalable in the sense that the same switch components can be used within coherent and noncoherent SCI systems.

1.7 Distributed directory structures

1.7.1 Linear sharing lists

To support a (nearly) arbitrary number of processors, the SCI coherence protocols are based on distributed directories[SciDir]. By distributing the directories among the caching processors, the potential capacity of the directory grows as additional caches are added and directory updates need not be serialized at the memory controller.

The base coherence protocols are based on linear lists; the extended coherence protocols provide compatible support for binary trees. Linear lists are scalable, in the sense that thousands of processors can share read-only data. Thus, instructions and (mostly) read-only data can be efficiently shared by large numbers of processors.

Memory provides tags, so that each memory line has associated state and (if the data is shared) identifies the cache at the head of the sharing list. The caches also have tags, which identify the previous and next elements within the sharing list, as illustrated in figure 4. The doubly-linked structure simplifies sharing-list deletions (which are caused by cache-line replacements).

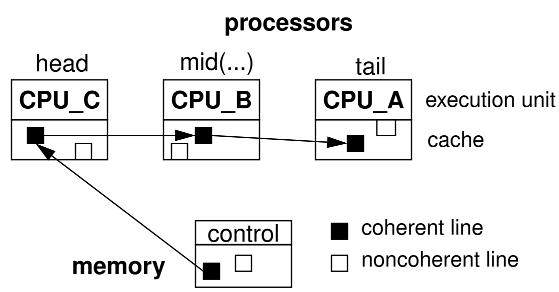

Figure 4—Linear-list directories

Although illustrated separately, a single SCI component may contain processor(s) and memory. However, from the perspective of the coherence protocols, these are logically separate components. Note that each memory-line address may have a distinct sharing list, and these sharing lists change dynamically depending on the processor's load/store behavior.

1.7.2 Sharing-list updates

Updates of the sharing list involve additions (new entries are always prepended to the head), deletions (caused by cache-line replacement), and purges (the sharing list collapses to a single entry). Additions occur when the data is shared by a new processor; purges occur when shared data is modified. Except for the special case of pairwise-sharing, all modifications are performed by the sharing-list head, which purges the remaining sharing-list entries.

The sharing-list update protocols are similar to those used by software to doubly-linked data structures, subject to the following constraints:

a) Concurrent updates. Sharing-list additions, deletions, and purges may be performed concurrently—precedence rules and compare&swap-like updates ensure list integrity (central semaphores are not used).
b) Recoverable faults. The update sequences support data recovery after an arbitrary number of transmission faults—exclusive data are copied before being purged.

c) Forward progress. Forward progress is ensured (i.e. each cache eventually gets a copy)—new caches are prepended to the head, and deletions from the tail have precedence.

1.7.3 Pairwise sharing

The linear-list structures are efficient if shared data is infrequently written, or if frequently-written data is shared by only a few processors. As an option, one of these cases, pairwise-sharing, can be optimized. The pairwise-sharing option is intended to support producer/consumer applications, where exactly two caches are sharing data.

When implemented, the pairwise-sharing option maintains the sharing-list structure, rather than purging the other sharing-list entry, when data is modified. The tail entry, as well as the head entry, is allowed to modify the data.

Thus, pairwise sharing allows processors to perform cache-to-cache transfers when fetching recently modified data. This minimizes access latencies and avoids a potential memory bottleneck.

1.8 Weak ordering

The coherence protocols allow the sharing-list head to proceed beyond a write operation before the remaining sharing-list entries have been invalidated. Stores into the same cache line by other processors require changes in the sharing-list structure and are deferred until the confirmation phase completes. This means that the SCI protocol can fully exploit the potential parallelism of a weakly ordered memory model.

We have divorced the details of the processor's memory-consistency model from the details of the SCI coherence protocols. Depending on the processor's memory-consistency model, some or all of following load and/or store instructions may be stalled while the invalidations are being performed.

For the strongest consistency models, these confirmation-completion interlocks delay the execution of the following memory-access instruction. For the weakest consistency models, these confirmation-completion interlocks are postponed until a dependent synchronization point is reached.

1.9 Extensions for large scale sharing

1.9.1 Overview

The base SCI coherence protocols are limited, in that sharing-list additions and deletions are serialized, at the memory and the sharing-list head respectively. Simulations indicate this is sufficient when frequently-written data is shared by a small number (10's) of processors [ExtJohn].

Although linear lists are a cost-effective solution for many applications, our scalability objectives also mandated a solution for (nearly) arbitrary applications on large multiprocessor systems. Thus, we are developing compatible extensions [ExtStd] to the base coherence protocols. The compatible extensions use combining and tree structures to reduce latencies from linear to logarithmic.

Sharing-list structures are created dynamically and may be destroyed immediately after creation. We therefore focussed on minimizing the latencies of sharing-list creation, utilization, and destruction, as described in the following sections.

1.9.2 STEM and GLOW extensions

Two sets of extensions are defined in this document referred to as STEM and GLOW extensions. To enable scalability of programs, both scalable reads and scalable writes for widely-shared data are required.

— Scalable Reads: Request combining, originally proposed for the NYU Ultracomputer, is the main vehicle for achieving scalable reads. The effect of request combining is achieved efficiently in GLOW because of the nature of the protocol itself, which caches certain information in the network. The request combining that STEM uses does not require information to be stored in the network.

— Scalable writes: Sharing trees are used instead of sharing lists as in SCI. GLOW is based on *k*-ary-sharing trees, while STEM defines binary sharing trees.

1.9.3 Target environments

GLOW and STEM are optimized for different types of target system. Each has its own advantages that may be better realized in a specific class of SCI environments.

1.9.3.1 Tightly coupled MPPs

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

1.9.3.2 Networks of Workstations (NOW)

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

1.9.4 Resource Requirements

1.9.4.1 STEM

To support the STEM performance extensions, processors have three pointers, instead of two. Having the additional pointers increases the tag sizes of the caches. In addition to the third pointer (**treeId**), a 5-bit height field (**height**) represents the height of the tree associated with the third pointer. The additional tag fields are shown in figure 5.

SCI requires 4 percent and 7 percent additional space at the memories and caches respectively to store the SCI-specific tag information. The majority of this additional space is used for pointers. The performance extensions require another 4 percent additional space at the caches, bringing the total increases to 4 percent and 11 percent respectively. Note that smaller systems can be implemented with smaller pointers and, therefore, less tag space. For 1024 nodes the performance extensions require only 3 percent and 7 percent additional space at the memories and caches respectively.

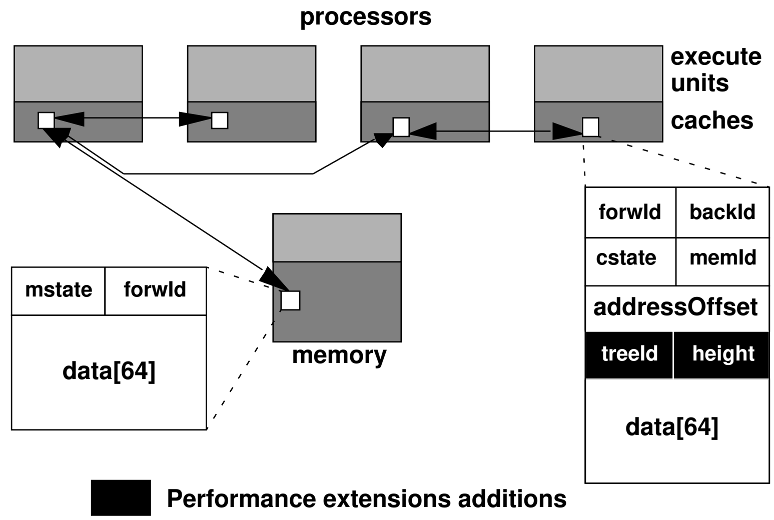

Figure 5—Extended cache tags

1.9.4.2 GLOW

To support the GLOW performance extensions, bridges that implement GLOW caching are required. These bridges are called GLOW agents and implement GLOW extensions transparently to the rest of the system. The standard SCI sharing list protocols are used for all transactions, while the GLOW agents supervise the creation and deletion of the GLOW sharing trees..

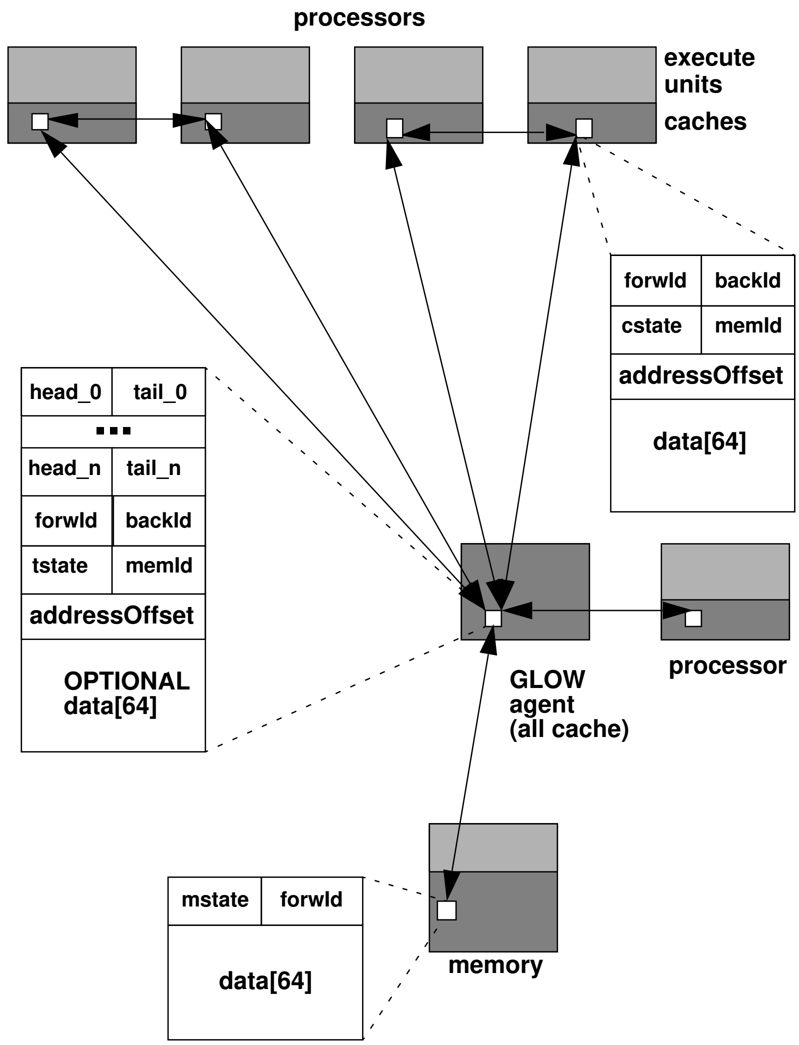

Figure 6—GLOW agents

In their tag storage GLOW agents have a forwid and a backid pointer that enables them to join standard SCI lists with other agents or caches. They also have an implementation dependent number of head and tail pointer pairs so they can manipulate SCI lists. Each tag has a state (tstate) and the associated data can be cached optionally

1.10 C code specification

The detailed SCI transport and coherence specifications are defined by executable C code [Code]; a multi-threaded execution environment is also provided. The original intent of the C code was to avoid ambiguous interpretations of the specification.

We soon found that the C code provided yet another form of scalability—the same specification could be used for clarity, for simulation purposes, or to verify hardware implementations. While creating the technical specification, bugs were often found and eliminated while running performance simulations.

1.11 Surprises

Designing for scalability has its share of surprises. Finding the scalable solution is hard, but, when compared with traditional solutions, it is often good for non-scalable implementations as well.

Our high-performance “backplane” objective forced us to use point-to-point signaling techniques. To our surprise, these techniques provided us with media independence as well.

Support of (nearly) arbitrary large switches forced us to use directory-based coherence protocols. Surprisingly, we found these techniques yielded superior pairwise-sharing performance, reduced sensitivities to tag-lookup-delay, and fault tolerance capabilities.

Supporting large numbers of processors forced us to distribute our coherence directories. We discovered that distributed directories could be updated concurrently, reducing latencies to O(log(n)) rather than O(n).

The most pleasant surprise came from observing the consensus process within the SCI working group. Because difficult problems have only a few (sometimes only one) scalable solutions, the requirements for scalability makes it easier to select which solutions should be used!

1.12 Summary

The coherence protocols defined within the base ANSI/IEEE Std 1596-1992 Scalable Coherent Interface specification and the P1596.2 Coherence Extensions are scalable in many ways.

When compared to unified-response backplane buses, packet-based split-response request/response protocols are scalable—they can be implemented on a wide variety of transport media and distances.

When compared to eavesdrop protocols, directory-based coherence protocols are scalable—they eliminate the need for specialized broadcast and eavesdrop capabilities.

When compared to central directories, distributed directories are scalable—large sharing lists can be supported with fixed-size memory tags.

When compared to linear sharing lists, sharing trees are scalable—large sharing lists can be efficiently created, utilized, and purged with logarithmic (rather than linear) latencies.

When compared to Fetch&Add, Accumulate is scalable—separate Accumulate operations can be readily combined within the interconnect.

SCI options are scalable as well. An implementation may elect to use only the noncoherent portions of the SCI protocols. Coherent implementations may elect to use the base coherence protocols (with their linear-list constraints) or the coherence extensions (with their binary-tree improvements).

Options are designed to interoperate, so that hybrid configurations perform correctly, although their performance typically degrades to that of the lesser implementation.

Although the cache-tag update algorithms may appear to be complex, SCI's directory-based protocols are relatively insensitive to response-time delays. Thus, the protocols can be implemented in firmware, hardware, or (most likely) a cost-effective combination of the two.

2. References and glossary

2.1 References

ANSI/IEEE Std 1596-1992, Scalable Coherent Interface (SCI).[1]

ANSI X3.159–1989, Programming Language—C.[2]

2.2 Glossary

TBD - describe shall, should, may, and expected.

TBD - separate glossaries for STEM and GLOW.

2.2.1 agent, GLOW: Bridge that connects one or more SCI rings and implements the GLOW extensions.

2.2.2 branch: A sharing-tree node that has one parent and two children.

2.2.3 callback: The process sending a return transaction when a previously-requested activity has been performed. The base SCI protocols use a less efficient polling mechanism, which requires sending additional transactions until the desired activity has been performed.

2.2.4 child: For a sharing-tree node *N*, this refers to the node pointed to by the *down* pointer from node N or the node pointed to by the *forw* pointer from node *N*.

2.2.5 child-list: An SCI list, handled by a GLOW agent.

2.2.6 child pointers: A pair of pointers (head_n and tail_n) that a GLOW agent uses to hold an SCI child-list. These pointers are also the forwid and backid of the agent whet it appears a virtual head or virtual head respectively.

2.2.7 children: For a sharing-tree node *N*, this refers to the two nodes pointed to by the *down* pointer and the *forw* pointer from node *N*.

2.2.8 head_n: The pointer that a GLOW agent uses to point to the head of a child list. It becomes forwid when the agent appears as a virtual head in the child-list.

2.2.9 leaf: A sharing-tree node that has one parent no children.

2.2.10 limb: A sharing-tree node that has one parent and one child.

2.2.11 parent: For a sharing-tree node N, this refers to the node pointed to by the back pointer from node N.

2.2.12 request forwarding: The process of accepting a request transaction, when the request transaction has the side effect of generating one (or two) additional request transactions. Request forwarding improves system performance, but is constrained to avoid creation of system deadlocks.

[1]IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, 800-678-4333.

[2]ANSI publications are available from the American National Standards Institute, Sales Department, 11 West 42nd St., 13th Floor, New York, NY 10036, 212-642-4900.

2.2.13 tail_n: A GLOW agent always appears as a virtual tail and this pointer represents the backid pointer of the virtual node.

2.2.14 virtual head: The case when a GLOW agent appears to be the head in one of its SCI child-lists. The head_n pointer of the GLOW agent becomes the forwid pointer of the virtual head node.

2.2.15 virtual tail: AGLOW agent always appears to be the tail in its SCI child-lists. The tail_n pointer of the GLOW agent is the backid of the virtual tail node.

2.2.16 walkdown: The process of traversing a tree to find a leaf node. The walkdown process is performed by a branch node, whose deletions requires the substitution of a leaf node at the branch location.

3. STEM extensions: Linear-list extensions

3.1 Delayed callback

TBD - Complete this section.

On the base protocol, responder can minimizing polling by waiting until half of the timeout period has expired.

The preferred solution is to use callback. When callback is used, busy-retry polling is never used. In the case of a prepend, the back pointer is immediately updated and a response is generated, even if this node is still in the CS_PENDING state. When the data arrives at the head of the old list, a callback transaction transfers the data to the new head.

TBD - Consider how callback is used with memory, when deletions are delayed while the deleting node is not the owner. With the focus on processor-to-processor callback, this one should not be overlooked.

3.2 Request forwarding

TBD - Complete this section.

Its not clear that request forwarding with purges yields any performance improvement, unless the tail can be given the return address for the callback.

StoreUpdate may benefit from request forwarding, since any node can send update requests to its adjacent neighbors.

4. Construction and destruction of STEM sharing tree lists

4.1 Request combining

4.1.1 Combining operations

Within some tightly-coupled shared-memory applications, data is concurrently accessed by large numbers of processors, typically when a synchronization point has been reached. If these read requests were serialized at the memory controller, access-time latencies would be linear.

To achieve logarithmic latencies, the read requests may be (optionally) combined within the interconnect. We assume that combining only occurs under heavy loading conditions, when queuing delays provide the time to compare request-transaction addresses for packets queued within switch components. Although combining rarely occurs, Guri Sohi noted that it occurs when it is most needed.

When at least two combinable requests (such as loads to the same cache line) are waiting in the same request queue (such as in a bridge or switch between SCI rings), two requests can be combined into one. The combined request is forwarded towards the memory controller and one early response is sent immediately. The early response contains a neighbor pointer for the sharing list (pointing to the head of another list) and the recipient no longer requires a response from the memory controller. When a request finally reaches the memory controller, it returns a pointer and changes its notion of the sharing-list head.

For example, consider how three requests (to the same memory address) may be combined during transit. Two requests (req B and req C) can be combined into one, which generates a modified request (req B-C) and an immediate response (res C). In a similar fashion, the combined request (req C-B) can be combined with another request (req A), as illustrated in figure 27.

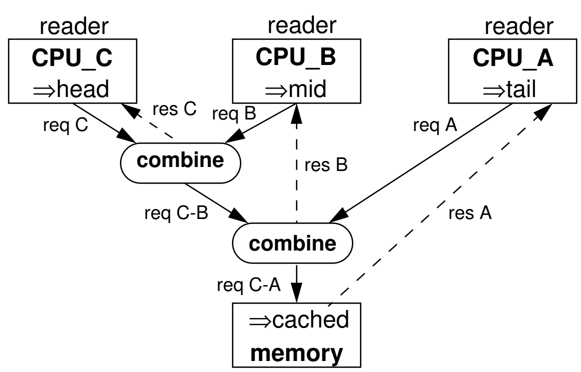

Figure 7—Combining within the interconnect

Since combining is most useful at the “hottest” spots, we expect to see the initial implementation on the front-end of the memory controller (to improve the effective memory-access bandwidth). Depending on performance benchmarking results, combining hardware may eventually migrate into SCI switches as well.

We use coherent combining to reduce the latency of sharing-list creation; combining only returns list-pointer information (and not data, which is returned later). We expect this to be much simpler than other approaches [NYU] which leave residual state within the interconnect, for modifying the responses when they return.

Because basic SCI nodes are accustomed to receiving responses without data, no complications are introduced by the data-not-returned nature of the combining mechanism. Thus, the basic coherent nodes function correctly when connected with a combining interconnect. Also note that combining the requests tends to clump nodes together that are relatively close to each other in the interconnect. This will make some communication between neighboring tree nodes more efficient.

4.1.2 Combined request

A combinable request represents a list segment, giving the node IDs of the two end points of the segment. A single node (segment of size one) can be represented by endpoints that are both equal to the given node, as illustrated in figure 8.

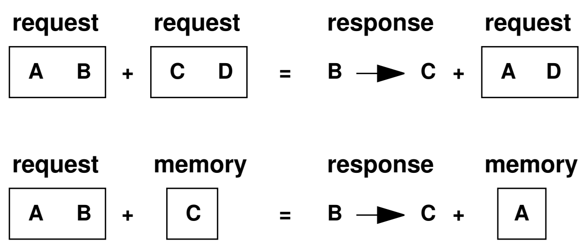

Figure 8—Combined request packets

When there is congestion and two combinable requests are waiting in the same request queue, they can be combined as shown (first equation). One response is returned immediately (no state is saved) that connects the two segments into one larger one. That is, the rightmost node of one segment is informed (on behalf of the memory controller) that its forward neighbor is the leftmost node of the other segment. The two endpoints of the resulting (larger) list segment form one request that replaces the previous two in the queue. This combined request is delivered as soon as congestion allows. When the memory controller receives a request (second equation), it responds to the rightmost node according to the SCI protocol, except that the new sharing-list head is recorded to be the leftmost node of the segment.

TBD: Is the following needed? A special transaction-command bit is used to distinguish the combinable read requests from fetch&add and their otherwise equivalent counterparts (which are generated by processors for memories that do not support combining).

4.2 Sharing-tree creation

After the linear prepend list (or portions thereof) is available, a merge process converts the linear-list structure into a list-of-trees structure. The latencies of the merge (and following) processes are O(log(n)), where n is the number of sharing-list entries, a pragmatic limit of most “scalable” protocols.

The merge process is distributed and performed concurrently by the sharing caches. Data distribution (for fetch&add, as well as load accesses) is performed during the merging process. Multiple steps are involved, one for each level of the binary tree which is created. However, some of these merge-steps can be overlapped in time.

Although each cache entry has three cache-location pointers, one of these pointers is NULL within the initial prepend-list entries. For continuity with the base SCI coherence protocols, two of these pointers are called back and forw; the third pointer (which is not assumed by the base coherence protocols) is called down. Within this standard, cache nodes are illustrated as triangles; each vertex corresponds to one of these three pointer values. The three vertices have unique markings, to indicate which of these pointer values is used, as illustrated in figure 9.

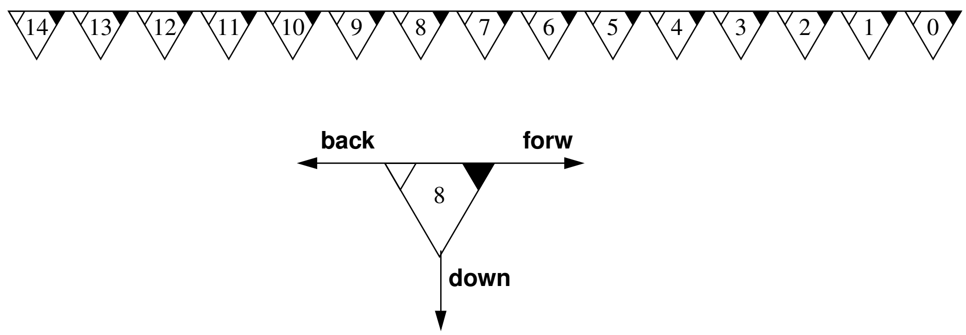

Figure 9—Initial prepend-list structure

Under ideal conditions, the first step of the merge process generates a list of 1-high sub-trees, the second generates a list of 2-high sub-trees, etc. The process continues until the sharing list has reached a stable state, as illustrated in figure 10.

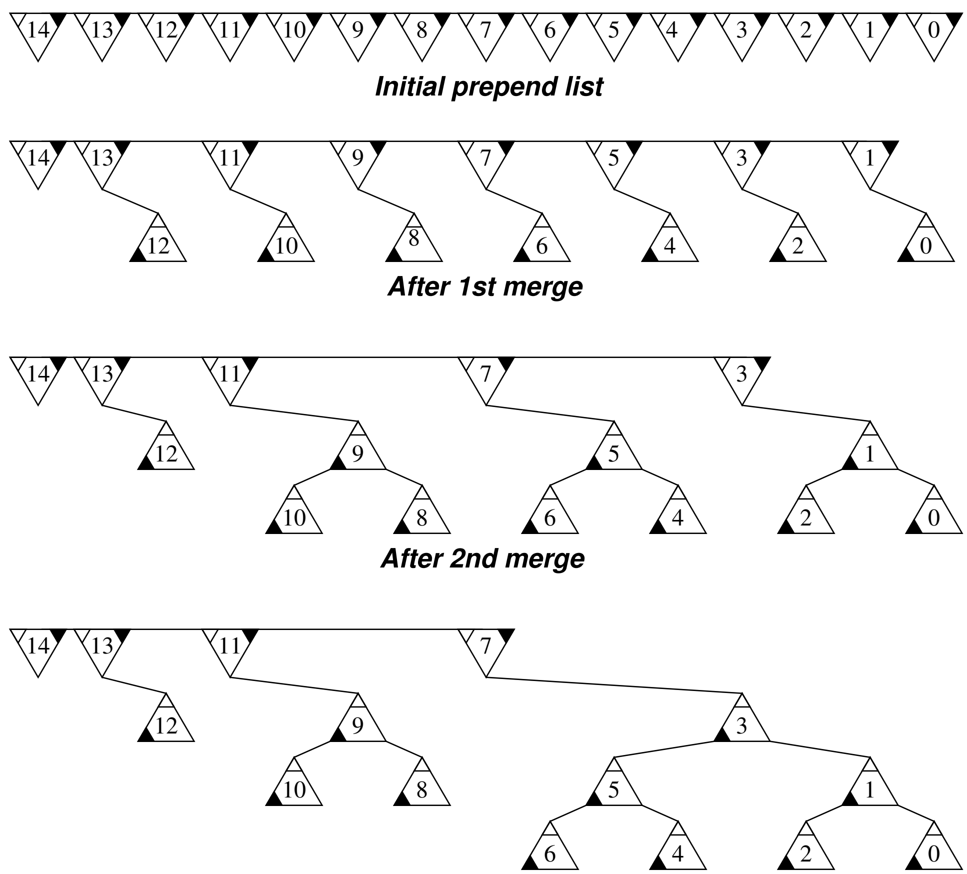

Figure 10—Creating sharing tree structures

When stable, the sharing list is a linear list of subtrees, where the larger subtree heights are located near the tail. Incremental additions are supported; adjacent equal-height subtrees merge to form a single one-height-larger subtree, until a stable subtree structure is formed. Up to three transactions are needed to merge subtree pairs.

During the subtree-merge process, a distributed algorithm selects which pairs of subtrees are merged. To avoid update conflicts, entries 2-and-1 can't be merged if 1-and-0 are being merged. To generate balanced trees, entries 3-and-2 should be merged if entries 1-and-0 are being merged. Note that the usual software algorithms for creating balanced trees can not be used, because the process must be distributed and only localized information can be used. Techniques for selecting the merged list entries are discussed in Section 5.

4.3 Sharing-tree destruction

Sharing tree destruction is initiated by a store instruction on the head of the list. Initiation-phase transactions distribute deletion commands to other sharing-tree entries. The deletion commands start with the head, which sends the deletion command to its children. After accepting the deletion command, intermediate nodes send additional deletion commands to their children, as illustrated in figure 11.

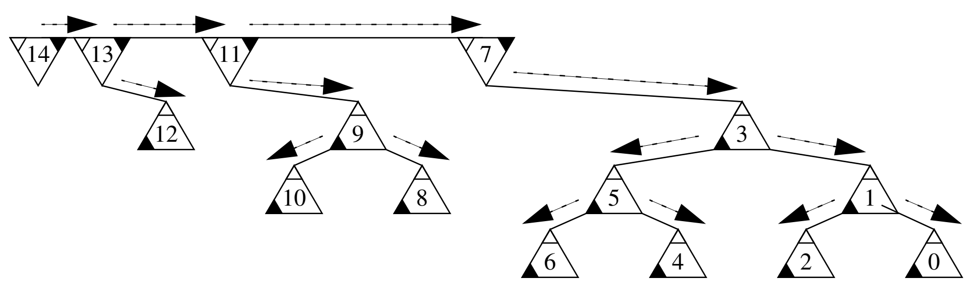

Figure 11—Sharing tree changes (initiation phase)

The forwarding of deletion commands stops with the leaf nodes. These nodes invalidate themselves and (in their returned response subaction) indicate that has been done. Intermediate nodes update their status (i.e. set their child pointer to NULL) when a child-was-leaf response is returned. This trims the leaf nodes from the sharing tree, as illustrated in figure 12. Shading is used to identify the newly-created leaf nodes.

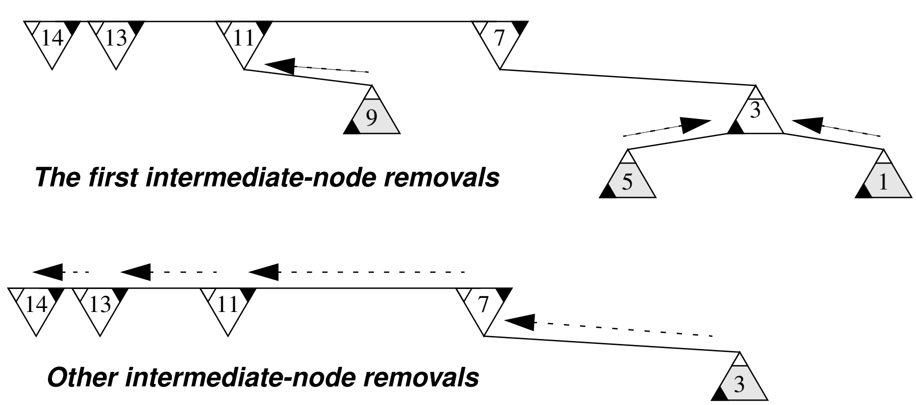

Figure 12—Sharing-tree destruction: intermediate-node removal

The (previously saved) status within newly-created leaves indicates that they should delete themselves as well. When their children's pointers are NULL, these nodes sent transactions to their parents and remove themselves from the list as well, as indicated in the first half of figure 12. This process continues, as illustrated in the second half of figure 12, until only the head entry remains.

4.4 StoreUpdate transactions

4.4.1 StoreUpdate generated by the head

Once formed, the sharing tree may also used to quickly distribute new data values, using a StoreUpdate (as opposed to the standard Store) instruction. This distributes the new data values to other sharing tree processors, rather than invalidating their (now stale) copies

When executed by the processor at the head of the sharing tree, initiation-phase transactions distribute update commands to other sharing-tree entries. The update commands start with the head, which sends the update command to its children. After accepting the update command, intermediate nodes send additional update commands to their children, as previously illustrated in figure 11.

The forwarding of update commands stops with the leaf nodes. These nodes update themselves and (in their returned response subaction) indicate that has been done. Intermediate nodes generate additional transactions when both child-was-leaf responses have been returned. These *confirmation-phase* transactions propagate towards the head, whose returned transaction indicates that all updates have been performed, as illustrated in figure 13.

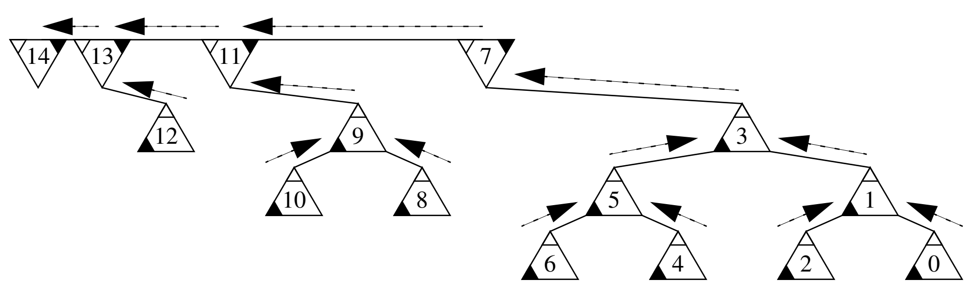

Figure 13—Sharing tree updates (confirmation phase)

TBD - comment a bit about how processors with weak ordering may execute other instructions while waiting for the final confirmation transaction to be returned.

4.4.2 StoreUpdates from non-head nodes

TBD - We considered performing StoreUpdate from non-head locations within the sharing tree. These non-head update protocols were not implemented because of their disadvantages, including but not limited to the following:

a) Latency. On large sharing trees, its faster to propagate StoreUpdates through the head than from the average sharing-list position.

b) Concurrency. Its hard to ensure correctness and forward progress when StoreUpdates from multiple sharing-tree nodes are concurrently active.

c) Starvation. Its hard to ensure that a StoreUpdate will ever finish, since its propagation towards the head by be delayed by additional prepending. Note that prepending can continue indefinitely, since thrashing can cause deletion and rejoining of other sharing-tree entries.

TBD -Talk about how StoreUpdate is done, when the node is not the head.

a) The modifying node has a schizophrenic behavior. While maintaining its position within the sharing tree (one of its personalities), it also prepends itself to the head.
b) As the new sharing-tree head (its other personality), the modifying node distributes a StoreUpdate transaction to the old sharing-tree head.
c) The modifying node deletes itself from the transient sharing list (its second personality is lost).

4.5 Deadlock-free request forwarding

TBD - comment on the fact that request forwarding can deadlock, unless an infinite input-request queue is provided. We therefore assume that the cache lines themselves are used as the pending request queues; linked together with the bus interface processing from the head of the list.

The initiation and confirmation transactions are special, in that they are forwarded through sharing-tree entries. The processing of input requests generates queued output requests. If a full output-request queue inhibits processing of incoming requests a system deadlocks could be generated.

To avoid these system deadlocks, the size of the outgoing request-queue must be sufficient to hold the number of cache-line entries within the node. Implementations are not expected to provide duplicate storage for large output queues. Instead, cache nodes are expected to implement their output-request queues as an localized linked-list of cache-line entries.

If there are no processes available at the child, then the child can not forward a purge immediately. Note that retry is not sufficient to guarantee forward progress without making a reservation to guarantee future acceptance. And, it can be shown that these reservations may lead to deadlock. Therefore, another method is required to guarantee forward progress.

When a node is unable to immediately handle a purge request, the request is acknowledged and then placed on an internal queue. The internal queue is a linked list of cache lines that still require purging. Note that the data space in the cache is no longer needed to store data, so a portion of this space can be used to store the pointers for the internal queue. The purging algorithm resumes when resources become available to handle the waiting purges. These waiting purges are given priority over new processor requests so that the queue is guaranteed to eventually empty. Note that processor requests will eventually be serviced because the processors can only wait for the purging of a finite number of cache lines in the local cache.

4.6 Sharing-tree deletions

Nodes (cache lines) may remove themselves from the tree prior to a purge, often as the victim of cache-line replacement or as a writer that need to become the owner. Removals are more difficult for trees than for lists when the removing node has two children. In this case, the removing node *walks* down the tree to a leaf node, removes the leaf from the tree, and finally replaces itself with the leaf node. Note that about half of the nodes in a tree are leaves, even if the tree is not balanced.

4.6.1 Leaf and limb deletions

If a node has zero or one child, then the basic SCI deletion protocols can be used. Deletion of a leaf node (which has no child) takes one transaction, like the tail-deletion within the base SCI coherence protocols. Deletion of a limb node (which has one child) takes two transactions, like the mid-entry deletion within the base SCI coherence protocols. These two deletion operations are illustrated in figure 14.

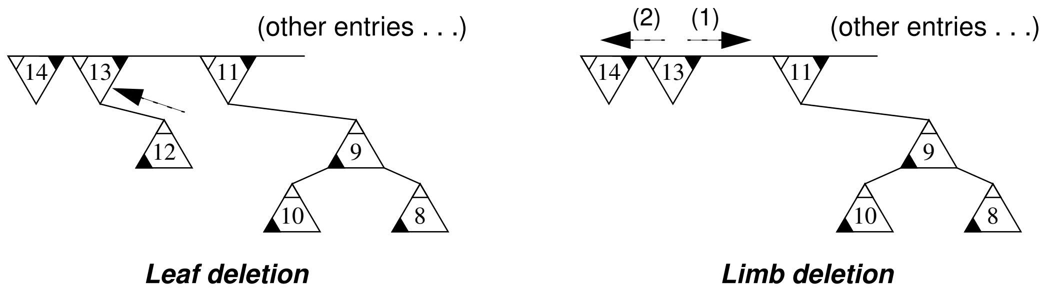

Figure 14—Leaf and limb deletions

4.6.2 Finding replacement nodes

If a rollout node has two children, it must find a replacement for itself in order to keep the tree structure intact. The leaf node that is used to replace the rollout node is found by first using the down pointer and then the forw pointer, as shown in figure 15. This search procedure creates a one-to-one correspondence between nodes with two children and nodes with zero children and there is no overlap between paths that connect pairs of nodes. Within figure 15, the leaf node which is effectively assigned to anode is listed next to the node.

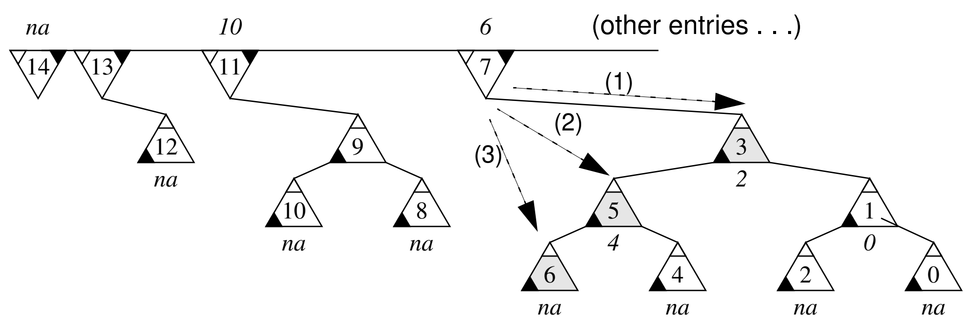

Figure 15—Replacement node values

During the walkdown, the removing node follows the down pointer once and then follows the forw pointer (or down pointer, if forw pointer is NULL) thereafter. When several nodes remove themselves simultaneously, two properties are maintained. First, at most one non-leaf node visits each node. Second, at most one non-leaf node removes each leaf node. These properties allow many of the deletions to be performed concurrently.

4.6.3 Replacement node substitution

After a leaf node has been located with the walkdown, the leaf must be swapped with the rollout node, as shown in figure 16. The first transaction detaches (4) the leaf from its parent. The parent of the rollout node is informed (5) that its new child is the leaf node. Concurrent transactions (6a and 6b) to the two children of the rollout node inform them of their new-parent (this parent was previously the leaf node). Finally, the leaf node is informed (7) of its new parent and children. The order of these non-overlapping steps is important to guarantee forward progress in the event that several neighboring nodes are trying to rollout at the same time. As was true with the base SCI coherence protocols, children have priority over parents.

The walkdown and node replacement protocols can introduce up to eight transactions per deleted node. However, half of the nodes in any binary tree are leaf nodes, meaning that the walkdown is only performed

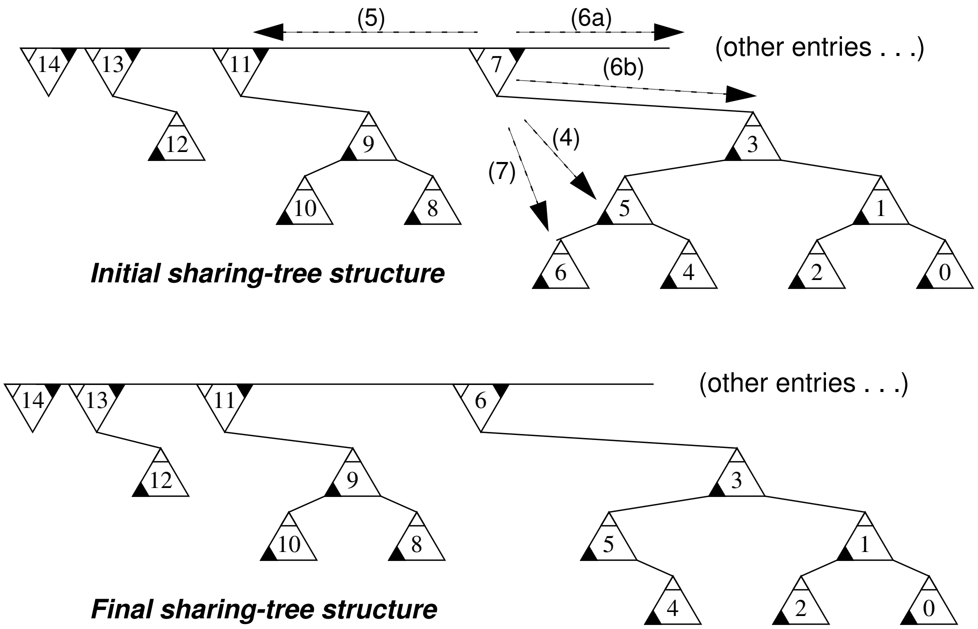

Figure 16—Rollout-node replacement

half of the time. Furthermore, one is the average number of walkdown steps required for nodes in a balanced binary tree and this average becomes even smaller as the tree becomes unbalanced.

4.6.4 Concurrent tree deletions

A problem with simultaneous node removals can be illustrated in figure 17. The walkdown from node 7 starts be getting a pointer to node 5 from node 3, and then a pointer to node 6 from node 5. Problems could arise if node 7 and node 3 attempt to concurrently delete themselves.

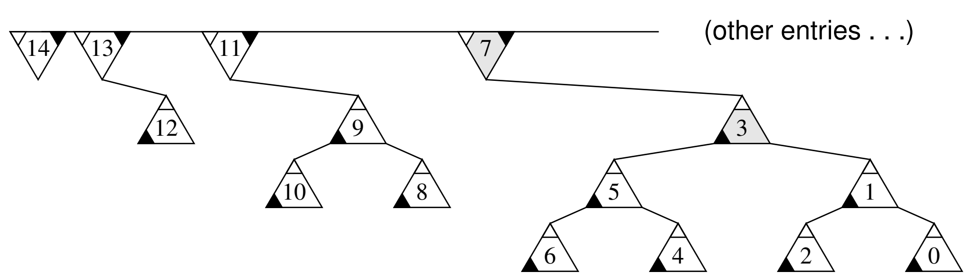

Figure 17—Concurrent tree deletions

The walkdown from node7 checks for a concurrent deletion from node3, its initial down-side node. When this condition is detected within node3, the returned status aborts node7's deletion. The node7 deletion is restarted after the node3 deletion completes.

The node3 deletion could start immediately after the initial walkdown from node7. In this case, the initial walkdown from node7 leaves node3 in an inactive state. The inactive state within node3 is changed when the replacement completes, whereupon the replacement process is restarted.

The deletion restarts are not expected to be a performance problem, since the node deletions are expected to be relatively infrequent and occur at different times. It is theoretically possible for nearly all nodes to delete themselves at the same time, since multiple threads may be executing in a lock-step fashion. Since this is rarely expected to occur, the restarts are not expected to be a performance concern.

Although performance may be impacted, restarts never cause starvation. The deletion precedence ensures that leaf-side nodes have precedence and are deleted first. Although branch-node deletions may be temporarily inhibited, the branch node is eventually changed into a leaf or limb node, at which point the restarted deletion process succeeds.

TBD - We should encourage distinct has codes within caches, based on the nodeId values. Otherwise, concurrent deletions are likely to occur when a conflicting data value is concurrently accessed.

4.6.5 Concurrent leaf conflicts

Another problem with simultaneous node removals is illustrated in figure 18. First, node7 gets a pointer to node5 from node3, and then a pointer to node6 from node5. Second, node6 removes itself, having no knowledge of node7's walkdown. Third, node7 contacts node6 and discovers that node6 is no longer in the tree (node6 has a different cache-line address or its back pointer doesn't point to node5). Note that it is not sufficient to back up to node5 because node5 might also remove itself in the interim. Therefore, when this situation occurs, the walkdown is simply restarted. Note that restarting does not create a forward-progress problem because at least one node is removed for each restart and the restarting node will eventually become a leaf node (or be successful in the walkdown).

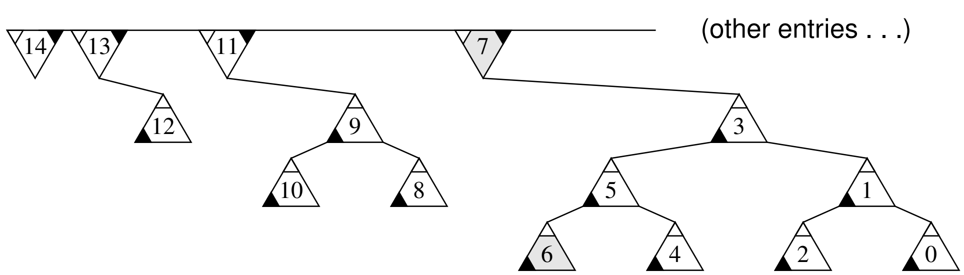

Figure 18—Concurrent leaf deletions

Note that two of the nodes (node5 and node6) could have moved to other places within the sharing tree during the walkdown process. Although node6 may not be the most-current replacement node, it can still be used correctly as a replacement node.

However, restart has a problem with performance. Although it is assumed that random node removals are relatively infrequent, it is possible for nearly all nodes to remove themselves simultaneously due to software that is running somewhat synchronously. Since it is not assumed that synchronous node removals are unlikely, there is a performance problem. It is desirable to recognize this condition and then orderly remove the nodes, beginning with the leaves. The following rules do just that.

4.6.6 Purge and deletion conflicts

TBD - talk about these some more.

4.7 Data distribution

After the merging is completed, the data is distributed. Unlike the base SCI protocol, where nodes continuously request the data until it is available (polling), the performance extensions make only one request for the data, which is then forwarded when it becomes available.

4.7.1 Load distributions

The object of data distribution is to get the data from the last writer (or another node) to all requesting nodes. To avoid extra transactions, data is propagated by cache-to-cache writes, as illustrated in figure 19. Request forwarding is not necessarily a problem, because one request buffer can be reserved until the data arrives and is forwarded. If a second buffer is available when the data arrives, then it can be used to forward the data to both nodes in parallel.

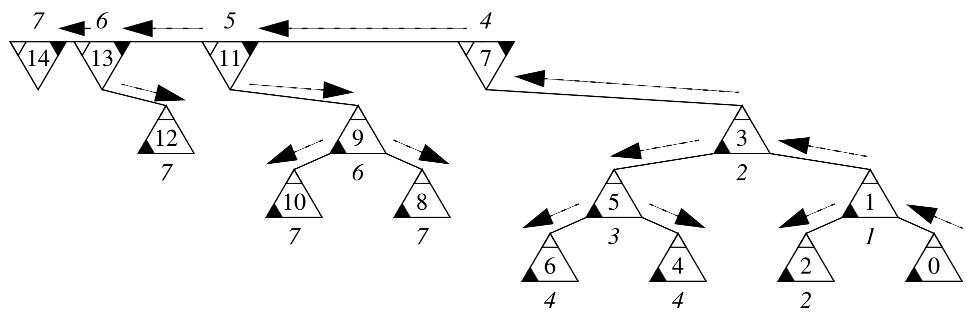

Figure 19—Load data distribution

The numbers shown next to each node indicate the number of subaction delays required to distribute the data from the sharing-tree tail to the other sharing-tree nodes. Actual delays are not as bad as illustrated, since some of these transactions would be sent during the sharing-tree creation process.

4.7.2 Fetch&add distributions

Instead of simply copying the read data in the data distribution phase, it is possible to specify a fetch&add operation instead. It is then important that all prepended cache-lines get different (and correct) values, and that the root of the tree gets the largest (final) value. The algorithm for doing this takes care of this, and in fact returns increasing values specified by the order in which cache-lines are added to the prepend list.

TBD - revise and revise as needed. During the conversion of a prepend list to a sharing tree, partial fetch&add contributions are saved within the created nodes. When the fetch&add data is distributed, these partial values are added to the distributed data, as illustrated in figure 20.

TBD - fetch&add nodes pass a delete-yourself command as well as modified data to their children. When data is received by a fetch&add node, a deletion of data-source node is also initiated.

TBD - integrate this text. Many read-modify-write operations, like fetch-and-add, can be combined. Other combinable operations include reads, integer writes, test-and-set, and part of barrier synchronization (with fetch-and-decrement). The fundamental characteristic that allows a group of operations to be combinable is that they are associative (not commutative). Note that if integer overflow is trapped, then integer arithmetic is not combinable (because it is not associative). However, by combining only positive (or negative) increments, overflow is also associative and combinable. Floating-point addition is not associative, also, due to floating-point round-off errors. However, it can be treated as associative if the magnitude of the round-off error is not critical. Finally, compare-and-swap is not associative and not combinable.

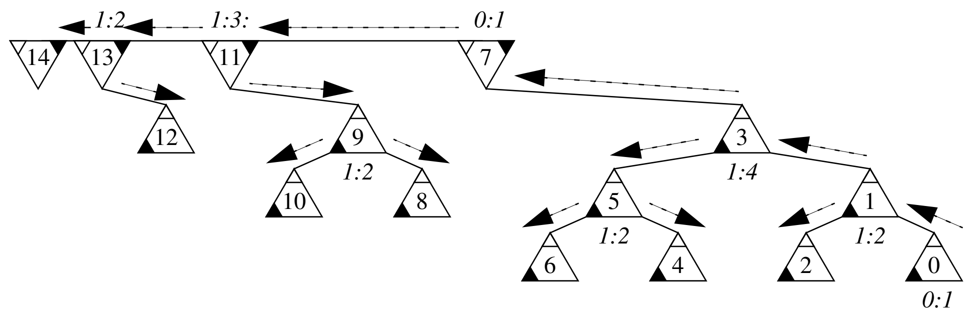

Figure 20—Fetch&add data distribution

To combine associative updates, the sharing list is first created normally. Then, updates are accumulated during tree merging. The update is finally completed during data distribution. After the data has been distributed and the value has been given to the processor, the nodes are left in an QUAD state.

5. Merge-selection protocols for STEM

5.1 Top-side labels

The specification of the distributed merge-selection protocol is an active topic of working-group discussion. The goal is to label alternate equal-height subtrees with '[' and ']' labels, allowing combining between '[' and ']' subtree entries. Distinct label assignments are hard, because the sharing-list entries are "nearly" identical and their list positions are unknown.

These final '[' and ']' labels can be viewed as crystals, grown from initially seeded '[' and ']' values.

We assume that the label assignment process starts with '[', ']', and '*' labels for prepend-list nodes. Thus, the typical prepend list starts with state labels as shown below.

```
[ * * * [ ] [ * * * ] ... ]
```

The meanings of these labels is specified within table 1. Note that the tail-most top-side node always has a ']' label state.

Table 1—Label state values

Label	Description
[	Merges with tailside node.
]	Merges with headside node.
*	Merges with headside or tailside node.
{	Is a '[' label, transitioning to a ']' state.
}	Is a ']' label transitioning to a '[' state.

From previous communications with adjacent nodes, top nodes are also aware of the heights of their adjacent head-side and tail-side nodes. Thus, the *complete label* consists of there characters: "HST". The H component is the relative height of the headsize neighbor, the S component is the label state (as described within table 1), and the T component is the relative height of the tailside neighbor. The values of H and T components are described within table 2.

Table 2—Label side (H and T) values

Label	Description
+	Neighbor's subtree height is larger.
=	Neighbor's subtree height is the same.
–	Neighbor's subtree height is less.

5.2 Merge conditions

Top-side nodes are required to update their list neighbors with their current subtree-height values. This starts with the transactions used to convert the singly-linked prepend list into a doubly-linked sharing list. Some of the new subtree-height values are distributed as a side-effect of the merge process; in other cases, explicit transactions are required to distribute the new subtree-height values.

The merging process is initiated by nodes with the '[' and ']' label-state values, which attempt to merge with their headside and tailside neighbors respectively. The merge process succeeds when merge attempts discover any of the complete-label pairs listed in table 3.

Table 3—Merged complete-label values

Label pairs		Description
`?[=`	`=]?`	Compatible merging initiated by both nodes.
`?[=`	`=*?`	Headside node merges with uncommitted tailside node.
`?*=`	`=]?`	Tailside node merges with uncommitted headside node.

where:
"?" represents any of the "+", "−", or "=" relative-height values.

A node may also merge with itself, if both of its list neighbors have a higher subtree height, i.e. the "`+?+`" complete label value. When this occurs, the node increases its subtree height by one and continues with the subtree-merge process. This node is responsible for communicating its new subtree height to its neighbors, either as part of the an active merge attempt or by generating an additional transaction.

5.3 Merge collisions

5.3.1 Upslope collisions

Nodes can only merge with equal-height subtrees. An adjacent larger-height subtree inhibits merging in that direction and initiates merging in the opposite direction, as specified in table 4.

Table 4—Revised labels based on subtree-height collisions

Old label	New label	Description
`+*~`	`$[$`	Headside cliff discovered; merge with tailside node.
`+]~`		
`~[+`	`$]$`	Tailside cliff discovered; merge with headside node.
`~*+`		

where:
"~" represents either a "−" or "=" relative-height label.
"$" indicates an unchanged height label.

5.3.2 Concurrent-merge collisions

Collisions may also occur between adjacent nodes of with the same subtree height. When this occurs, one of the labels is uncommitted and the other undergoes a deferred change, as illustrated in table 5. In the absence of a successful merge (which could be concurrently active), a deferred change initiates a merge in the opposite direction.

Table 5—Revised labels based on concurrent-merge collisions

Old label pairs		New label pairs		Description
`?[=`	`=[?`	`$*$`	`${$`	Concurrent tailside merges
`?]=`	`=]?`	`$}$`	`$*$`	Concurrent headside merges.

Until the deferred changes occur, the '`{`' and '`}`' nodes have the behavior of the '`[`' and '`]`' nodes respectively. The deferred changes (if they occur) change the label states as specified in table 6.

Table 6—Deferred complete-label changes

Old label	New label	Description
`?{?`	`$]$`	Tailside merge stops, headside merge begins
`?}?`	`$[$`	Headside merge stops, tailside merge begins

The deferred change takes place immediately after a pending response is returned, if the response indicates that the requested merge has failed due to a larger-subtree height conflict.

TBD - clearly state that only the last of successive conflicts has a deferred state change.

5.3.3 Inactive merge states

Collisions may also occur with subtrees of a lower height, labelled '−'. These nodes enter a waiting state (no active merges are attempted), since merge decisions cannot be resolved until the subtree height has increased. These inactive waiting states are listed in table 7.

5.3.4 Adjacent-height storage

Since the subtree heights are bounded, 5 bits is sufficient to hold the neighbor's subtree height. However, to reduce tag-storage costs, a 2-bit storage option for the relative height is also supported. The encoding of these two-bit subtree height values is specified in table 8.

The 2-bit storage is less costly, but saturates on large (>= myHeight+3) values. Extra transactions (based on inaccurate saturated neighbor-height values) are sometimes generated, reducing the overall transaction-bandwidth efficiency.

Table 7—Inactive waiting states

Label	Description
`?[-`	Tailside merging delayed by lower-height tailside neighbor.
`-]?`	Tailside merging delayed by lower-height tailside neighbor.
`-*-`	Uncommitted peak, merge initiated by neighbors.
`-*=`	Uncommitted left edge, merge initiated by neighbors.
`=*-`	Uncommitted right edge, merge initiated by neighbors.
`=*=`	Uncommitted plateau, merge initiated by neighbors.

Table 8—2-bit relative-height label values

Value	Description
0	neighborHeight <= myHeight
1	neighborHeight == myHeight
2	neighborHeight == myHeight+1
3	neighborHeight >= myHeight+2

5.4 Merge-label seeding

TBD - use sequence number first, then miniId, then reversed miniId, then random number. Prioritized boundary resolution is defaulted to the highest priority. That way, perfect seeding has precedence, finite runs are ensured, and good statistical behavior is possible. Discuss how priority affects the merge decisions.

5.4.1 List-position seeding

TBD. Possible when the list position is saved within the memory controller and returned within the response. This requires additional memory-tag storage (or associative tags, for frequent accesses), but yields perfect sharing-tree structures.

5.4.2 Extreme miniId seeding

TBD - clarify this discussion in more detail.

Calculate the 5-bit neighbor-unique miniId values for each top-side node, based on its own nodeId value and the nodeId value of its tailside neighbor. Label-state values are assigned based on the node's miniId value and those of its two adjacent neighbors. For peaks in the miniId values, a '`[`' label is assigned. For valleys in the miniId values, a '`]`' label-state is assigned.

An ideal miniId value is small, easily calculable based on limited nearest-neighbor information, and is neighbor unique. Neighbor unique means that the miniId values for a topsize node differs from those of its headside-neighbor and tailside neighbors, although these adjacent neighbors could have identical miniId. An algorithm which approximates these goals is specified in table 9.

Table 9—Calculating miniId values

```
unsigned Encode16To20(Doublet);

unsigned
MakeMiniId(Doublet thisNodeId, nextNodeId)
{
  unsigned thisId, miniId;

  thisId = Encode16To20(thisNodeId);        /* encoding of your nodeId */
  nextId = Encode16To20(nextNodeId);        /* encoding of next nodeId */
  miniId = FirstOne(~thisId & nextId, 20);  /* The LSB that goes 0-to-1 */
  return(miniId);
}
/* Generate code values, where for any two distinct code values there is a
 * 0-to-
1 transitions in one of the bit positions within the two code words. */
unsigned
Encode16To20(Doublet x)
{
  unsigned data:16, zeros:4, code:20;

  data = x;
  zeros = ZerosCount(x);                    /* Count number of zero bits */
  code = (zeros << 16) | data;              /* Prepend zero-count to x */
  return(code);
}

/* Return one less than the number of zero bits within the input data value.
 * The returned value is always positive, since the all 1s value is not used.
 */
unsigned
ZerosCount(unsigned x)
{
  int i, count;

  assert(x < 0XFFFF);                       /* The encode-able values */
  count = -1;                               /* Count starts from 0 */
  for (i = 1; i <= 0X8000; i <<= 1)         /* and is incremented for */
    count += ((x & i) == 0);                /* each of the zero bits */
  return(count);
}
/* Find the first one bits within the input data value. */
unsigned
FirstOne(unsigned x, unsigned bits)
{
  int i, count;

  mask = 1;                                 /* Check starts with the LSB */
  for (i = 0; i < bits; i+= 1, mask <<= 1)  /* Other bits are checked until */
    if (x & mask)                           /* the first zero bit is found */
      break;
  }
  return(i);
}
```

We have identified two *Encode16To20()* functions, as listed here and in annexxx. The encoding described within the annex is technically superior but would have been much more complex to implement.

The *FirstOne()* routine identifies the least-significant bit that is 0 within *thisId* and 1 within *nextId*, where *thisId* and *nextId* are encoded forms of the *thisNodeId* and *nextNodeId* argument values. The returned *miniId* value is a positive integer and should not be confused with labels normally assigned to bits within data formats. We propose that miniId is based on the least-significant 0-to-1 transition, so that only the smaller count values are generated when a subset of the node identifiers are used.

Note that only one *Encode16To20()* computation is needed if the *Encode16To20(thisNodeId)* value is precomputed when the system is initialized. Implementation options for the miniId computations are as follows:

a) ROM lookup. The "y = Encode16To20(x)" operation is performed using a table lookup, i.e. y = table[x]. A 64K by 20-bit ROM is required.
b) RAM lookup. The "y = MakeMiniId(x)" operation is performed using a table lookup, i.e. y = table[x]. A 64K by 6-bit RAM is required; the values in this RAM are initialized after the node's nodeId value is assigned.
c) Combinational logic. The encoded values are generated by prepending four bits to x, where the 4 bits represent one less than the count of zero bits within the binary representation of x.

5.4.3 Reversed miniId seeding

Bit reverse and complement the original 5-bit miniId values, then pick the peaks for ‘`[`’ and the valleys for ‘`]`’.

5.4.4 Random number seeding

TBD. Flip a coin concept, using random or pseudo-random number generator. The pseudo-random number generators are seeded with the node identifier, to decorrelate the behavior between (otherwise identical) nodes. The definition of this pseudo-random number generator is TBD.

6. GLOW extensions: Topology-dependent sharing trees

GLOW extensions for widely shared data are based on building *k*-ary sharing trees that map well onto the network topology of a system. STEM captures *temporal locality* in the sharing tree: requesting nodes, in close proximity in *time*, end up neighbors in the sharing tree. GLOW captures *geographical locality*, because the structure of its sharing trees is not dependent on the timing of the requests. By *geographical locality* we mean that nodes in physical proximity become neighbors in the sharing tree, regardless of the timing of their requests. Such locality in the tree leads to protocols where messages do not travel very fare .GLOW captures the geographical locality by mapping the sharing trees on top of the trees formed from the natural traffic patterns.

6.1 GLOW agents

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

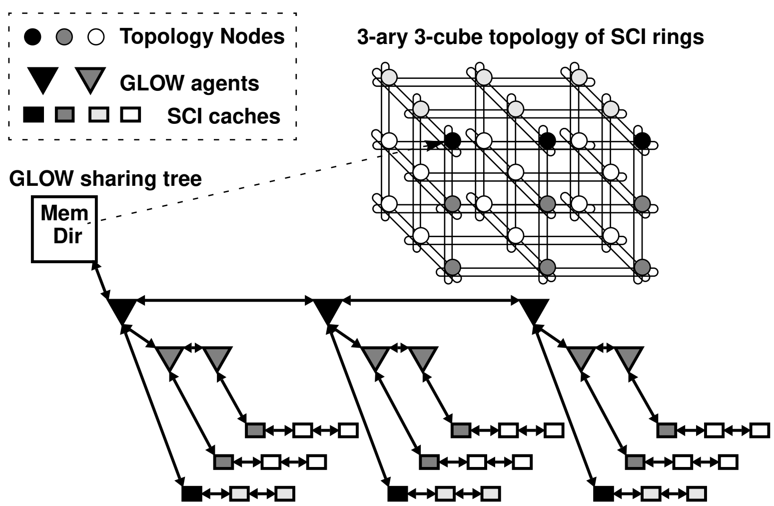

Figure 21—GLOW sharing tree on a *3*-ary *3*-cube

The GLOW sharing tree comprises the SCI caches that have a copy of the data block and the GLOW agents that hold the relevant directory information. All the SCi caches in a GLOW sharing tree have a copy of the data in state FRESH. The sharing trees are built out of small, linear SCI sharing lists. In figure 21 a full, perfectly ordered, sharing tree on a 3-ary 3-cube topology made of SCI rings is shown. The agents are represented by triangles and the SCI caches by small rectangles. The shading of the agents and the SCI caches shows their position in the topology. An agent and the cache directly connected to it having the same color, coincide in the same node.

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

A GLOW agent has a dual personality: towards its children it behaves as if it were the SCI memory directory; towards its parent agent (or towards the memory directory itself) it behaves as if it were an ordinary SCI cache. For example, in figure 21, as far as the memory directory is concerned, it points to a list of SCI caches, whereas in reality it points to the first level of agents holding the rest of the sharing tree. Similarly as far as the SCI caches (the leaves of the sharing tree) are concerned, they have been serviced directly by memory where in fact they were serviced by the agents impersonating memory.

Although the GLOW agents can behave like memory directories there are two differences in the way GLOW agents and memory directories hold SCI lists. The first difference is in the number of SCI lists that the agents can hold. In contrast to the SCI memory directory which can only hold one SCI list (per data block), the agents have a number of pairs of pointers (head_n and tail_n), called child pointers, so they can hold an SCI list (called child-list) in any of the rings they service[3]. The actual number of child-lists is not specified, but is considered an implementation parameter, reflecting both topology and cost considerations.

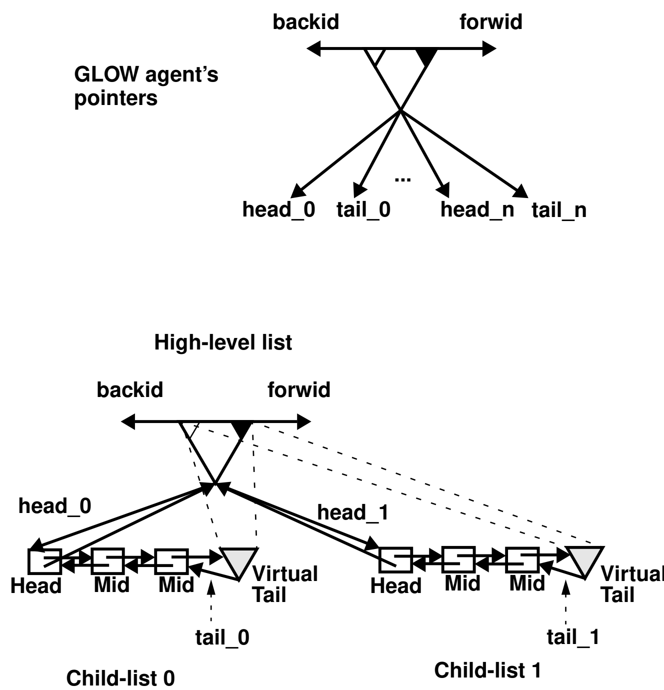

Figure 22—GLOW agent holding 2 child-lists

The second difference is in the way the GLOW agents hold the child-lists. In contrast to the memory directories which only point to the head of a list the GLOW agents hold lists from both ends. This is achieved, in an SCI compatible way, by the agent presenting itself as a *virtual tail* to the first node that joins each child-list. Virtual nodes are a convenient way for the agent to manipulate an SCI list. When the agent appears a a vir-

[3]One less than the total number of rings they are connected to.

tual tail in one SCI list, in effect it has a pointer (tail_n pointer) to the node that would be the actual tail. This pointer is updated correctly with deletions from the list and it always points to the node that would be the actual tail. In figure 27 the agent is represented by a triangle and the SCI caches by rectangles. For every child-list the GLOW agent has a *head* and a *tail* pointer (see figure 27). The agents also have a *forward* and a *backward* pointer, permitting them to join SCI lists and act like ordinary SCI caches.

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

6.2 Wide-sharing read requests

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

Because of the overhead, GLOW is used only for widely shared data. A special request for such data allows the agents to intercept it. This request is a variant of the FRESH request. This is not a change in the SCI protocol for non-GLOW nodes. The special request need only be recognized by the GLOW agents but it can interpreted as an ordinary FRESH request at anytime, by any node in the system.

TBD. Exact format of the request for widely shared data.

7. Construction and destruction of GLOW sharing trees

7.1 Construction

The GLOW sharing trees are constructed with the involvement of the GLOW agents that impersonate the memory directory in various places in the topology. A node uses a special request to access a line of widely-shared data. A normal request would go to the memory directory and the node would be instructed to attach to an SCI list and get the data from the head (if the list was stable) or the last node that joined the list if the data were still in transit (as in figure 27). However, the special request is intercepted at the first agent (where the request would change rings), *at the agent's discretion*. The intercept causes a lookup in the agent's directory storage to find information (a tree tag) about the requested line which will result in a *hit* if there is an allocated tag or a *miss* otherwise:

If the lookup results in a miss, the agent sends its own special request for the data block towards the home node (figure 27). The agent instructs the requesting node to attach to a child list comprised only of the virtual tail (the agent itself) and wait for the data[4]. As soon as the agent gets a copy of the line it will pass it to the child-lists through the virtual tails. Through the use of the virtual tail the SCI node cannot tell the difference between a GLOW agent and the actual memory directory.

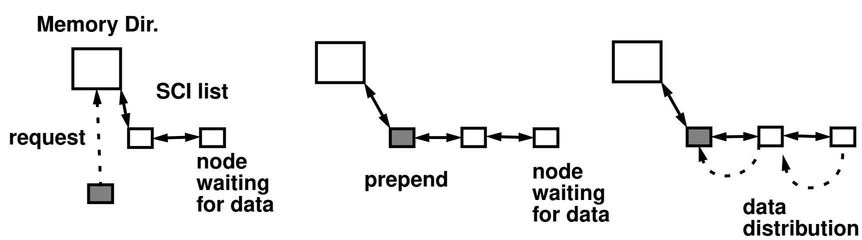

Figure 23—Node attaching to a prepend SCI list

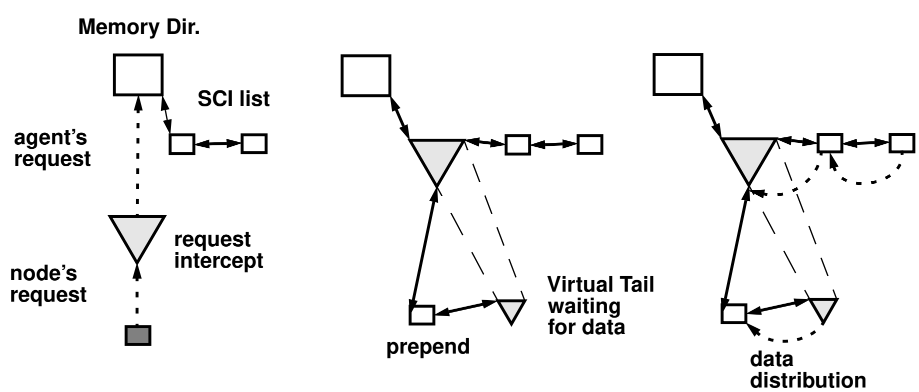

Figure 24—GLOW agent intercepts node's request and generates its own request (Miss in the agent's directory)

If the lookup results in a hit, the requesting node is instructed to attach to the appropriate child-list. It will get the data from either the agent (if it caches data) or the previous head of the child-list.

[4]Unlike the base SCI protocol, where nodes continuously request the data until they are available (polling), the GLOW extensions assume that nodes make only one request for the data, which are then forwarded from a sibling when they become available.

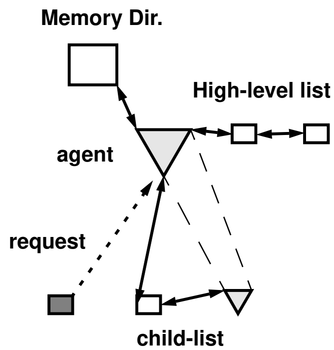

Figure 25—Hit in agent; node joins a pre-existing child list

Since a GLOW agent can have multiple child lists the requesting node might be the first in a child list while the agent has other child-lists. In this case, i.e., when the requesting node attaches directly to the virtual tail in a new child list the agent has to *fetch* the data from one of its other child-lists. Notice that the agent cannot repeat its request to get the data, since this would result in joining the sharing tree twice.

The fetching of data is achieved by the agent attaching as a virtual head in front of one of its populated child lists and reads the data. Fetching data can be applied recursively (descending the sharing tree) until a node that actually has the data is found.

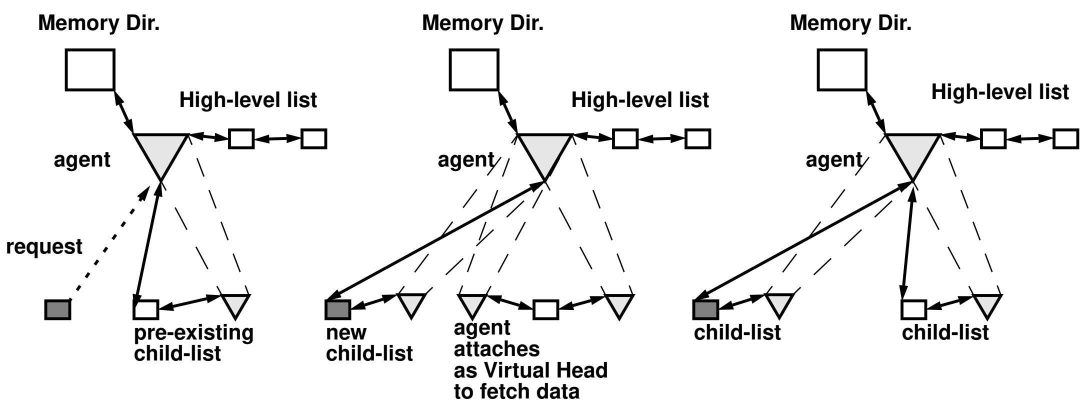

Figure 26—Hit in agent, node joins a new child-list

In the case of a miss in the agent, its own request will be treated the same way in the next agent. Eventually the last agent will get the data from memory and pass it to its child-lists. When all nodes simultaneously read a cache line (for example, after a global barrier) all the linear lists can be constructed in parallel. Copies of the cache line are distributed down the tree concurrently with list construction.

Request combining occurs because the GLOW agents generally do not let intercepted requests pass them by. Instead when the agent is invoked, it generates its own request and sends it towards the remote memory directory. The memory directory sees only the requests of the first level of agents. Such behavior has two effects: First it eliminates memory hot spots, the situation where a node receives a disproportionately large number of requests in a short time compared to other nodes in the system. Second, because the requests are satisfied locally, messages travel only short distances, thus decreasing the load on the network.

7.2 Agent deadlock avoidance

An agent may choose to ignore a request completely. This does not affect the correctness of the protocols at all, but it may have a negative impact on performance. When an agent ignores a request the request is passed to the next hierarchical level where hopefully it will be serviced by the higher level agent. In this case the node that issued the original request joins a linear list in a higher hierarchical level than normally. The request may be ignored by multiple agents all the way to the home node where eventually it will be serviced by the memory directory. The agents may ignore requests to avoid deadlocks due to storage conflicts.

Specifically, a request is always ignored when it requires storage for a tree tag that is currently taken by another tree tag in a waiting state (not yet part of a stable tree). A deadlock can occur when two different sharing trees in construction, require the same tree tags in two different GLOW agents. Assume that one sharing tree occupies a tree tag (in a transient state since the tree is still being constructed) in the first agent, while the other sharing tree occupies a tree tag in the second agent. If each of the trees requires the tree tag in the agent occupied by the other tree in order to complete construction then there is an unavoidable deadlock. By letting the agent ignore requests that need a transient tree tag occupied by another sharing tree this problem is avoided. If a tree tag is in a stable state (in a stable part of a sharing tree) then it can be deleted (rolled out) to be used for new tree.

7.3 GLOW tree deletions

7.3.1 SCI cache rollout from a GLOW sharing tree

An ordinary SCI cache leaves the tree (rolls out) because of a replacement or as a prerequisite for writing the data. The SCI caches follow the standard SCI protocol to rollout from the sharing tree.

7.3.2 GLOW agent rollout from a GLOW sharing tree

Agents roll out because of conflicts in their directory (or data) storage or because they are left childless.

Childless agents are not allowed in the tree. As soon as the last child rolls out the agent rolls out too. This is necessary since a childless and dataless agent would have to repeat its request for data in order to service new children. As mentioned above, agents are not allowed to repeat their requests.

TBD. Other options may exist. Can an agent repeat its request and not join an SCI list again ?

When an agent finds itself childless it is only connected to the tree with its *forward* and *backward* pointers, just like an SCI cache. In this case the rollout is the standard SCI rollout.

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

In figure 27 A we depict a segment of a sharing tree where the agent in the middle of the high level list is about to rollout. The agent becomes *virtual head* in all its child-lists with *attach* requests (figure 27 A and B). The previous heads of the child-lists are now mid nodes. All the virtual nodes are in reality only one entity (the agent itself) and there is no change in the pointers of any node. The only change is in the perception of the nodes that were heads of the child lists. Before the attach these nodes believed the agent to be the memory directory, while after the attach they believe the agent is just another node in from of them..

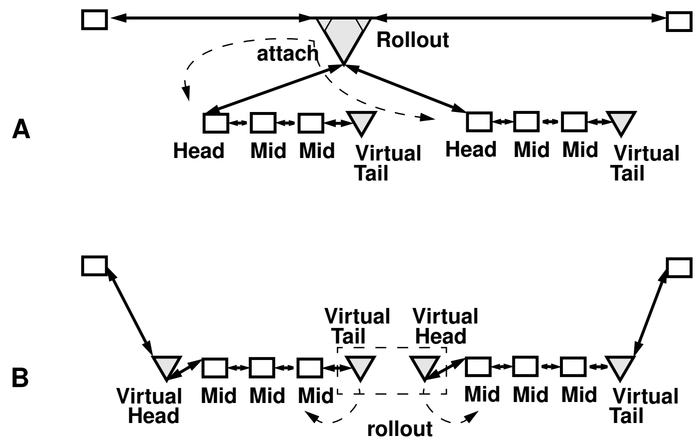

Figure 27—GLOW Agent as virtual head in front of all its child-lists

Since the virtual tail of the leftmost child-list and the virtual head of the rightmost child-list are the same node (the agent) they can rollout as one (in one atomic rollout) and leave the two child-lists connected into one (figure 27 A). In this way any number of child-lists can be concatenated into one list in one step. Concurrently, the agent rolls out as virtual head and virtual tail of the concatenated child-lists (figure 27 A). The sharing tree after the agent rollout is shown in figure 27 B.

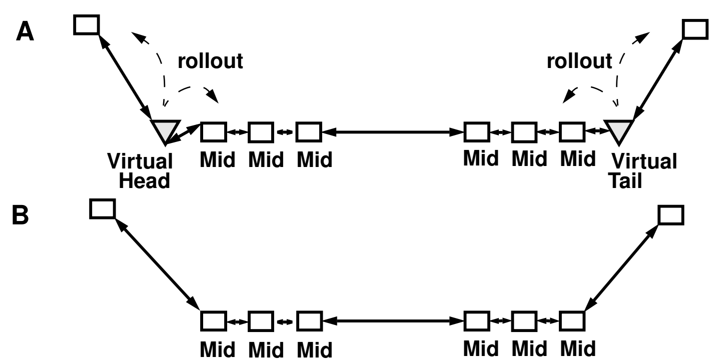

Figure 28—Child-list concatenation and agent rollout

Although the sharing tree degrades over time as agents rollout this method is potentially more effective than the alternative of invalidating all the agent's child-lists (a method we call Destructive Rollout). Two arguments support this claim: First, there is minimal interference to other nodes: while in the Destructive Rollout many shared copies that are potentially in use, are invalidated, this scheme requires only the participation of the heads and tails in the concatenation of the child-lists. Second, the latency of invalidating a subtree can be higher than the latency of chaining the child-lists together and substituting the agent. Therefore replacements in the agent's directory storage can be much faster.

7.4 GLOW sharing tree destruction

In order to write a cache line a node must first become the head of the top-level list connected directly to the memory directory in the home node (figure 27 A,B). In this position the node is the *root* of the sharing tree and it is the only node that has write permission to the cache line. A node has to rollout from the sharing tree before becoming root.

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

— Ignores the node that send the invalidation message pretending that it is about to rollout (figure 27 B). TBD- a negative acknowledgment to the initial invalidation request may be required to deal with time-outs.
— Forwards the invalidation to its downstream neighbor; if the downstream node happens to be an SCI node and responds with a new pointer the agent proceeds with invalidating the next node (figure 27 B).
— Attaches to its child-lists and as a virtual head starts invalidating them using the SCI invalidation algorithm (figure 27 C).

When the agent is done invalidating its child-lists it waits until it becomes tail in its list. This will happen because it will either invalidate all its downstream nodes or they will rollout by themselves (if they are GLOW agents). When the agent finds itself childless and tail in its list it will invalidate and rollout from its upstream neighbor, freeing it to invalidate itself (figure 27 D).

.

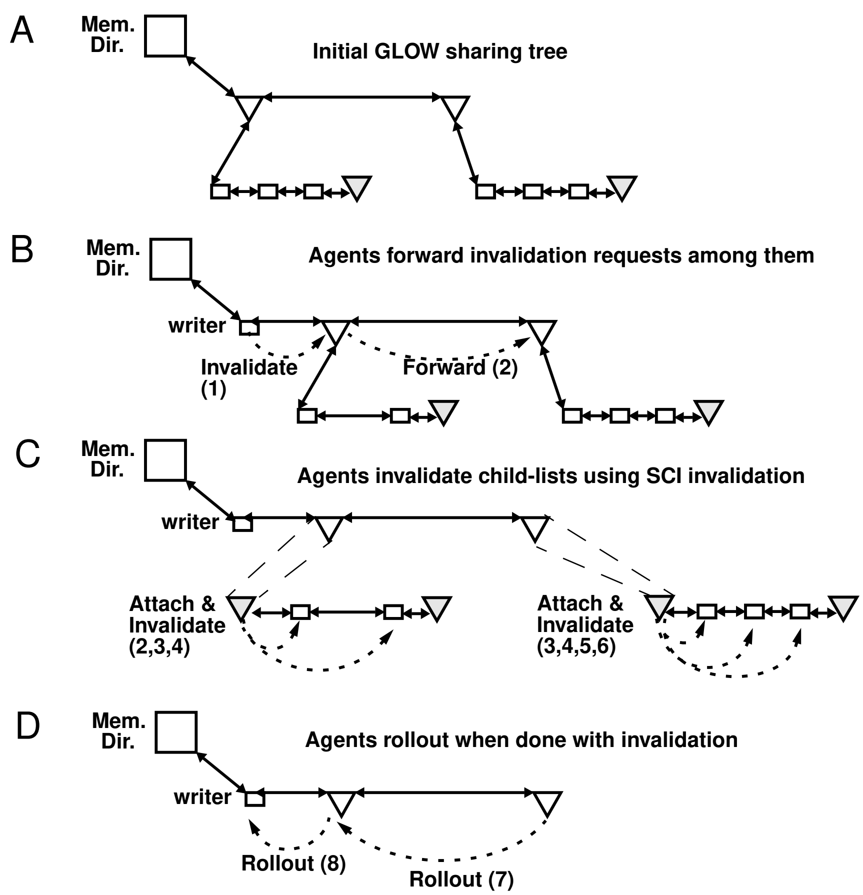

Figure 29—Invalidation of GLOW sharing Tree

7.5 GLOW Update (StoreUpdate)

TBD. This section needs further development.

With a GLOW Update protocol a sharing tree is constructed and it is subsequently used to distribute updates. There are several issues in implementing such a scheme. For sequential consistent memory models, a central place is needed where updates can be serialized. This is either the memory directory itself or the root of the sharing tree if it is unique as is the case for GLOW. It is desired for writer nodes not to leave their place in the tree so the tree structure remains unaffected.

7.5.1 Full GLOW tree update

A writer node becomes the root of the tree and sends updates down the tree. Updates are forwarded by all nodes in the tree (GLOW agents and SCI nodes). When the updates reach the leaves of the tree acknowledgments are returned to the root. The root node prohibits any other node of becoming root until it receives all the required acknowledgments. Although, currently the writer nodes leave their place in the tree to do updates we do have a solution for the nodes to do the updates in place. This update protocol does not assume sequentially consistent memory.

This form of update requires changes in the SCI protocols: update of SCI nodes and request forwarding.

TBD. Status of the SCI Update protocol.

7.5.2 Partial Update

Partial Update protocol is when only the GLOW agents are updated. This protocol is exactly the same as GLOW invalidation except that only SCI nodes are actually invalidated and deleted from the tree. GLOW nodes are updated and remain in the tree.

GLOW agents forward the updates among them. Intervening SCI nodes are invalidated (the GLOW update message is equivalent to the SCI invalidate message). The rest nodes in the sharing tree are invalidated and they have to request the data again. In this way we decrease the update traffic by updating only a few GLOW agents and we bring the data closer to nodes that might potentially access them in the future.

Partial Update requires no change in the SCI nodes.

7.5.3 Pipelined Updates

Pipelined Update tries to solve the problem of delaying any further updates until the writer node gets all acknowledgments back. It still guarantees that all nodes of the same tree will observe the same order of updates. The updates are pipelined and the pipeline stages correspond to the hierarchical levels of the tree. Each hierarchical level signals the receipt of the update immediately to its parent level, at which point the next update is allowed to proceed. Depending on whether we care the final acknowledgments of the updates (that ascend the tree to the root) must be observed in order or not, we may impose pipelining to the returning acknowledgments. As of yet, the details of this scheme have not been carefully examined.

7.5.3.1 Pipelined Updates in SCI

TBD. Pipelined Updates are applicable in SCI. Describe.

7.6 Combinable Fetch&Φ in GLOW

GLOW performs combining of reads (independently of their timing) thus speeding up accesses to widely shared data. An opportunity arises here to leverage the cost of having GLOW agents for more complex operations such as combinable Fetch&Add or Fetch&Increment and combinable Test&Set. This would expand the use of GLOW to synchronization objects such as high contention locks and barrier counters, and it would help in the implementation of various algorithms such as reductions or centralized work queues. Even without these enhancements GLOW Update can be rather useful in implementing barrier synchronization in systems that do not provide for it in hardware. The details of these enhancements have not been examined closely. It should be noted however that implementing Fetch&Add in GLOW is rather difficult.

7.7 Adaptive GLOW

GLOW is used only for widely shared data. For this a special request (a variant of the FRESH) request is employed. Widely shared data can be defined by the user or compiler. The special requests are generated by the processor when it accesses such data or alternatively a special processor instructions (e.g., favored or colored loads) or a sequence of instructions generate such requests.

GLOW can be extended to discover widely shared data dynamically. This is useful when special requests cannot be generated easily or the widely shared data are unknown. In this section, enhancements to the GLOW extensions are discussed. The adaptive GLOW discovers widely shared data at run time. The full responsibility for handling widely shared data is transferred entirely to the coherence protocols. However, adaptive GLOW may require small changes in the SCI nodes.

7.7.1 Discovery of widely shared data

There is a single point in the shared memory machine where the history of data block can be observed. This is the memory directory at the home node of the data block. Widely shared data can be discovered at run time with a counter in the memory directory that counts write runs (the number of reads in-between writes). The counter is incremented for every read request and it is reset to zero with a write request. This assumes that the writer must notify the memory directory in order to write the cache line (in accordance to the FRESH state of the nodes in the sharing tree). If the counter exceeds a certain threshold the data are deemed widely shared. The counter does not need to be very large to detect widely shared data and its size does not need to be proportional to the size of the system. This is because, most shared memory programs either share data among very few nodes or (in the case of widely shared data) among all (or almost all) of the nodes. This bi-modal behavior of shared-memory programs has been observed in many benchmarks programs.

TBD - Extend discussion on the counter size.

After the detection of widely shared data, the objective is to either set up the GLOW agents in the topology to handle the widely shared data transparently or notify all nodes about the nature of the data block so they can use GLOW for further accesses (by issuing special requests).

TBD- Detecting widely shared data requires changes in the SCI memory directories. Discuss specific changes and their implementation.

7.7.2 Top-down trees

In this scheme the sharing trees are build top-down. Ordinary read requests arrive at the memory directory which can detect widely shared data. When the memory directory detects widely shared data it starts sending back special responses. On their way back to the requesting nodes, the special responses build the sharing trees by invoking the GLOW agents.

In detail this scheme works as follows: Nodes read a data block by sending ordinary requests to the memory directory. A counter per data block in the memory directory counts the read requests between the writes. The counter is reset with every write. While the value of the counter is low, the memory directory services the nodes by returning ordinary responses to the read requests. If the counter exceeds a certain value, the data block is considered to be widely shared. All read requests beyond this point are serviced by special responses. These special responses have to return from the same path the corresponding requests took and they invoke every GLOW agent on their way. The GLOW agents are instructed (by the special responses) to install the appropriate information in their caches and start intercepting any further requests for the relevant data block that go toward the memory. The sharing trees are build top-down from the memory directory toward the requesting nodes. A few of the first requests as well as a few requests that will escape the agents will not be handled as widely shared. This scheme however, has some disadvantages:

1) It is very sensitive to timing in the network: whether the agents (after they are invoked by the special responses) will be able to stop a burst of requests going toward memory depends on the exact timing of the requests and the latencies of the system components.
2) It is topology dependent since it requires that the special responses take the reverse path of the corresponding requests.
3) It requires that the GLOW agents snoop at everything, possibly slowing down every transaction in the system. After the GLOW agents are instructed to install the appropriate information about a data block in their caches they have to snoop at every passing request in case it has to be intercepted

7.7.3 Direct notification

The notification of which data block is widely shared, goes to the nodes instead of the GLOW agents. This information is then stored in a table in each node. On subsequent cache misses, the nodes consult the table before they send out any request. If, according to the table, the requested data are widely shared, then a special GLOW request is sent and the building of the sharing tree proceeds as described in previous sections. The notifications are sent directly by the memory directory to all nodes (broadcast) at the point when the memory directory decides that the data block is widely shared. This method also has some drawbacks:

1) If the broadcast of the notification arrives at the nodes after they have sent their requests to memory (*i.e.,* too late) the opportunity to build a sharing tree is lost.
2) If the data change from widely to non-widely shared, a scheme is needed to detect this change and notify the nodes again. Unfortunately, since the nodes believe that it is widely shared they will use the GLOW agents for their accesses. The GLOW agents will mask requests from the memory directory which means that it no longer has a correct picture of the access behavior in the system.

7.7.4 Hot tags

Notification about which data block is widely shared is transferred to the nodes by overloading the invalidation messages. The notification is sent when a writer invalidates the readers of widely shared data. The writer notifies the memory directory before writing the data. The memory directory informs the writer that the data block was widely shared (based on the value of the counter). The writer overloads its invalidation messages with the value of the counter which is then received by all readers (but potentially not all nodes in the system). The invalidated readers can keep the information about the widely-shared nature of the block in special tables as mentioned before or in the invalidated cache blocks themselves, called *Hot Tags.*

Since we do not need the data of an invalidated cache block, we can store the counter value that was carried by the invalidation messages in the data area so we do not need to increase the size of the tag itself. A new tag state ("Invalid_with_counter") is required to distinguish between the invalidated data and the counter value. If the counter value is not used at all to make local decisions regarding the nature of the data, a new state ("Invalid_widley_shared") suffices. When the node references again the cache line (assuming it was not replaced) it finds it in an extended invalid state. This forces the node to use a special GLOW request to access the data block. The building of the sharing tree proceeds bottom-up as described in previous sections. There are two reasons why this scheme could be successful:

1) Many programs that have widely shared data are bimodal in their degree of sharing, *i.e.,* they have low and high degrees of sharing but not much in between. This means that with the invalidation of a widely shared data block most nodes are notified about its nature.
2) In the case where widely shared data are frequently written (*e.g.*, inside a loop), it is unlikely that the hot tags will be replaced, thereby losing the information, by the next time the nodes will reference the same block. This case is very important because frequently written widely shared data are a serious performance concern.

The reason why such a scheme could not be very successful for some programs is that some data blocks are widely shared only once. This scheme fails to do anything for the first time the data are widely accessed. Furthermore this scheme also does not adapt instantaneously to a change in the nature of data from widely to non-widely shared. However, given enough time unused hot tags are likely to be replaced, returning the nodes to a state where they do not have any knowledge about the nature of the data.

8. Integration of STEM and GLOW

TBD - This work is in early stage.

GLOW and STEM can be integrated in one protocol. GLOW can be used to provide topology specific request combining. If the GLOW agents provide very few tree tags, replacements in the GLOW tree are going to convert it to a liner list. STEM can be used to convert these lists to trees. In this way STEM trees inherit the geographical locality of the GLOW trees. The exact point when STEM can kick-in and start converting lists into trees is not clear at this point. STEM can be used on the GLOW tree itself, to convert child-lists into trees.

The existence of GLOW agents has serious implications for STEM's merging algorithms. GLOW can provide local merge-labeling.

TBD - Implications of using GLOW agents to provide request combining for STEM trees.

9. SCI issues

9.1 Local memory tags

9.1.1 Partitioned DRAM storage

TBD - Complete this section. Be careful to ensure emulation of SCI behavior and ensure forward progress.

We expect that many nodes will be shared processor/memory/cache nodes. In these cases, the node's dynamic RAM can be partitioned into a portion that is mapped as global memory (and appears directly in the node address offset) and a portion that is used to cache remote accesses. When accessing the global memory from the local processor, there is no point in copying the data into DRAM cache, since the speed of the DRAM cache and DRAM memory are approximately the same.

Within these environments, the local processor should participate in the SCI coherence protocols when remote processors share or modify the data. However, costs should be minimized by eliminating the need to provide extra cache tags for the local processor.

To do this, the node implements the FRESH memory state. Other nodes can join a FRESH sharing list, but cannot modify the data while it is in the FRESH state. Thus, the local processor can have read-only access to the FRESH state. The local hardware can detect remotely-invoked FRESH-to-GONE changes in the memory state and invalidate the processor's copy.

9.1.2 Transient cache-line behavior

The local processor can generate conflicts with remotely-cached data in the following circumstances:

a) Writing shared data. The local processor modifies data, when its memory state is FRESH.
b) Sharing modifiable data. The local processor reads data, when its memory state is GONE.
c) Writing modifiable data. The local processor writes data, when its memory state is GONE.

In all of these cases, the node allocates a transient cache tag, to emulate the behavior of a caching processor. Only a few of these cache tags are needed, since the cache tag is only used while converting between memory states.

In case (a), the node joins the old sharing list head, invalidates the other entries, and deletes itself from the sharing list (converting it to the HOME state). The existing SCI coherence protocols efficiently support this behavior, allowing the use of the memory-provided data while the other sharing-list entries are being purged.

In case (b), the node fetches the data from the old sharing list head, converts it into the FRESH state, and deletes itself from the sharing list. To improve system performance, the transaction which prepends to the old sharing list is combined with the transaction which deletes the transient cache entry from the newly formed sharing list.

In case (c), the node joins the old sharing list head, copies the data from the list (leaving it in the STALE state), modifies the data, and purges the remaining sharing-list entry. The existing SCI coherence protocols efficiently support this behavior, allowing the use of the list-provided data before the stale entry is purged. Although one might be tempted to invalidate the old sharing-list directly, the two-transaction access of dirty data is needed to support fault-recovery protocols after transmission failures.

Note that this transient-caching behavior also improves system performance when shared data is accessed by a remote node. The locally cached data is returned to memory and the local node behaves as though the data were in a FRESH memory state. The node-local cache interrogation delays request-transaction processing, but is otherwise architecturally transparent to the remote node which joins the sharing list.

Annexes

Annex A
Bibliography

(informative)

[Serial] P1394, SerialBus.[1]

[Caches] Michael Dubois and Shreekant Thakkar (editors), "Cache Architectures in Tightly Coupled Multiprocessors," *Computer*, June 1990, Volume 23, No 6.(special issue).

[DashDir] Daniel Lenoski, et. al., "The Directory Based Cache-Coherence Protocol for the DASH Multiprocessor," Proceedings of the 17th International Symposium on Computer Architecture, May 1990, 148-159.

[DashSys] Daniel Lenoski, et. al., "The Stanford Dash Multiprocessor," *Computer*, March 1992, Volume 25, No 3, pp. 63-80.

[Limit] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS directories: a scalable cache coherence scheme", presented at the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, CA, April 8-11, 1991.

[SciDir] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurinday S. Sohi, "Scalable Coherent Interface," *IEEE Computer*, June 1990, Volume 23, No 6, 74-77.

[ExtJohn] Ross Evan Johnson, "Extending the Scalable Coherent Interface for Large-Scale Shared-Memory Multiprocessors," PhD Thesis, Computer Sciences Department, University of Wisconsin-Madison, February 1993. Computer Sciences Technical Report #1136.

[NYU] Gottlieb, A. et. al., "The NYU Ultracomputer — Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. on Computers*, February 1983, Volume C-32, No 2, 175-189.

[1]P1394 is an authorized standards project that is unapproved at the time of this document's publication. The latest version of this document is available from the IEEE Computer Society.

Annex B
Instructions set requirements

TBD - complete the clause.

This clause describes the features within a processor instruction set which allow heterogeneous multiprocessors to communicate using the features of this standard.

B.1 Read and write operations

TBD - describe how these are defined within the CSR Architecture and shared-data-formats standards.

read1,2,4,8,16,64
write1,2,4,8,16,64
writesb

Note that write64 is useful for message passing.

TBD - should we really encourage indivisible accesses of 64 byte linelets?

Note that writesb is a performance optimization.

B.2 Basic lock operations

TBD - describe how these are defined within the CSR Architecture and shared-data-formats standards.

mask&swap, compare&swap, fetch&addBig, fetch&addLittle
4- and 8-byte data sizes

B.3 Extended lock operations

TBD - describe how accumulate8 is an optimization of the fetch&add8 instruction.

accumulate8

B.4 Transactional memory support

TBD - We need to think about this one some more. Recent article from Harold Stone should provide some guidance.

B.5 Store update

StoreUpdate8 shall be implemented (for barrier synchronization). StoreUpdatesb and StoreUpdate64 should also be implemented.

B.6 Interrupts

B.6.1 Memory-mapped interrupts

Interrupts can generate queue-dependency deadlocks if processor nodes delay the acceptance of interrupt requests until other transactions complete. An example is a processor that delays the acceptance of interrupts (these write4 transactions are busied) while higher-priority interrupts are being serviced.

To illustrate such an interrupt-queue deadlock, consider the execution of high-priority interrupt service routines on physically separate processors. If each interrupt service routine sends lower-priority interrupts to the other processor, both interrupt transactions would be continually busied, waiting for the other processor to lower its interrupt-service priority, as illustrated in figure B.1.

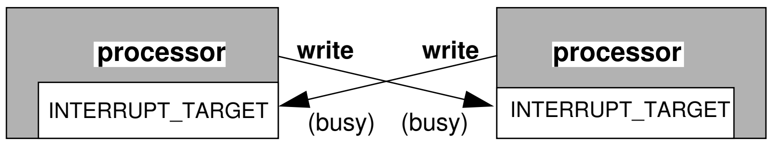

Figure B.1—Deadlocked multiprocessor interrupts

Processors can avoid this type of deadlock by providing sufficient storage to queue all outstanding interrupts or by not using interrupt-related resources when processing queued interrupts.

B.6.2 Maskable interrupt bits

To immediately queue all memory-mapped interrupts, processors are expected to provide external write-access to the least-significant half of a 64-bit interrupt-bits register. Special interrupt-set and interrupt-clear control registers provide local software access, and a mask selectively enables higher-priority interrupts, as illustrated in figure 2.

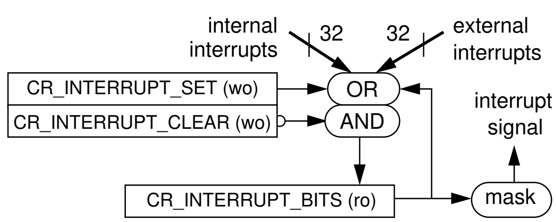

Figure 2: Processor interrupt model

Interrupt-queue bits are preferred to a finite-sized interrupt-vector queue, since the interrupt-queue bits never overflow. Each interrupt-queue bit typically corresponds to a completion-status queue in memory, which can be directly accessed by processors and I/O devices.

B.6.3 Prioritized interrupt bits

TBD - Illustrate a 256-level fixed-priority implementation as well.

B.7 Clock synchronization

TBD - discuss global clock format (integer seconds and fractions-of-a-second components). Illustrate implications of the clock-strobe packet.

To support real-time applications, processors are expected to support 64-bit synchronized clocks which represent time in seconds and fractions of a second, as illustrated in figure B.3.

integer seconds	**fractions of a second**
32 bits	32 bits

Figure B.3—Globally-synchronized time

Synchronization of these clocks requires an adjustable clock-tick register (to control clock speed), a clock-sample register (to sample clock value on an interconnect-provided clock-strobe signal), and a clock-alarm register for generating interrupts after alarm times are reached, as illustrated in figure B.3.

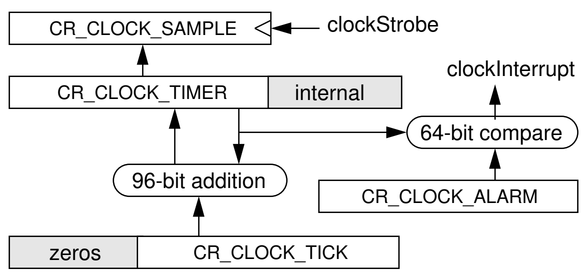

Figure B.4—Processor clock model

B.8 Endian ordering

TBD - discuss the usefulness of a byte-swap instruction, as illustrated in figure B.5.

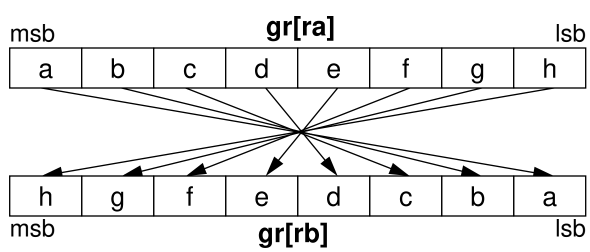

Figure B.5—Byte-swap instruction capability

B.9 Cache hashing

TBD - nodeId-dependent hashing of virtual addresses to cache-index values may be desired to minimize concurrent cache-line rollouts caused by cache-line conflicts.

B.10 Page protection

TBD - discuss 64 byte and 4K protection models. A 64-byte protection model is useful when providing user-level access to I/O devices.

Annex C
Special benchmark considerations

TBD - complete the clause.

This clause describes the common operations, whose performance requirements have influenced the design of this standard.

C.1 Barrier synchronization

C.1.1 Simple barrier synchronization

With confidence that the basic sharing-tree structures handled many of the coherent sharing applications, we proceeded to examine “worst-case” scenarios. In particular, we considered synchronization of multiprocessors at barrier points within code (barrier synchronization).

For a design model, we assumed that barrier synchronization involved multiple processes, each of which calls a Barrier() routine, as illustrated in table 1, when its task completes. For n processors, the value of *next* is n larger than the value used on the previous barrier.

Table 1—Simple barrier-synchronization code

```
Barrier(int *sum, int next) {
  int old;

  FetchAdd(sum, 1);
  /* Wait for barrier to be reached */
  while (*sum != next);
}
```

Analysis of this algorithm found potential performance problems. In typical applications, the *FetchAdd()* and *while()* accesses will occur concurrently. Sharing-tree thrashing can occur: each *FetchAdd()* operation destroys the existing sharing-tree structure; each *while()* access rebuilds a portion of the final sharing-tree structure. In addition to generation of unnecessary interconnect traffic, thrashing increases barrier-synchronization delays.

C.1.2 Two-phase barriers

To improve barrier-synchronization performance, we assume that different addresses (not in the same cache line) are used during the *accumulate* and *notify* phases of barrier synchronization. Processes increment the *add* value before the barrier has been reached and then poll the *sum* value to determine when the barrier has been reached, as illustrated in table 2.

This code avoids the thrashing problem; the sharing tree for the `sum` location is unaffected by the arithmetic operations performed on the `add` value.

To improve the performance of repetitive barrier synchronization operations, a StoreUpdate instruction distributes the `sum` value. The `sum` value is written into the sharing list head and then distributed to other sharing-tree entries. By using StoreUpdate, rather than Store, the sharing tree structure is preserved for use during the notify phase of following barrier-synchronization operations.

Table 2—Two-phase barrier code

```
Barrier(int *add, int *sum, int next) {
  int old;

  old = Accumulate(add, 1);
  if (old == (next-1))
    /* Notify others, barrier was reached */
    StoreUpdate(sum, next);
  else
    /* Wait for barrier to be reached */
    while (*sum != next);
}
```

C.1.3 Accumulate vs. FetchAdd

To improve performance, we have replaced the *FetchAdd()* operation with an *Accumulate()*. Accumulate differs from FetchAdd in that a NULL (zero) value (rather than the previous data value) may be returned for intermediate (but never the final) Accumulate calculations.

This distinction allows Accumulate operations to be combined within a stateless interconnect. When combined, an immediate NULL response is returned.

Annex D M
ulti-linc nodes

TBD - describe the structure of a multi-linc node, as described within the SCI C code.

Annex E
DC free encoding

(informative)

This standard uses a sub-optimal 20-bit encoding for intermediate miniId calculations. This annex illustrates how its theoretically possible to use a smaller 19-bit encoding with the same properties, i.e. any to distinct code words have a 0-to-1 transition in one of their bit values. Although fewer miniId values would have been generated (19 values instead of 20), the complexity of this encoding didn't justify its additional complexity.

We describe a technique for generation of a monotonically-increasing function for mapping a range of unsigned integers into their DC-free code-word values. By DC-free, we mean that a binary form of the code word contains an equal number of 1's and 0's. For a 2*N-bit encoding, the total number of values that can be encoded is as follows:

```
           2N * (2N-1) * ... * (N+1)
total =  -------------------------
             N * (N-1) * ... * 2
```

This represents the number of ways that N objects (in this case 1 bits) can be picked from a set of 2*N objects (in this case, encoded bits). For this application, interesting values for N and M are as follows:

```
20 values can be encoded in 6 bits
184,756 values can be encoded in 20 bits
```

The algorithm for generating these codes is referenced in:

J. P. M. Schalkwijk, "An algorithm for source coding," IEEE Trans. Inform. Theory, vol. IT-18, pp. 395-399, May 1972.

As discovered (by Dave James) in the following reference:

Richard F. Lyon, "Two-Level Block Encoding for Digital Transmission," IEEE Transactions on Communications, Vol COM-21, No. 12, December 1973.

Lyon's description of this algorithm is as follows:

"Schalkwijk's algorithm involves a walk through a matrix whose elements are binomial coefficients (i.e., elements of Pascal's triangle)." The following table shows the matrix used to generate 6-bit codewords"

```
| start here
v
* 10 *  6 *  3 *
*  4 *  3 *  2 *
*  1 *  1 *  1 *
*  0 *  0 *  0 *
                ^
                | finish here
```

"The coding is accomplished by defining a walk through the matrix, starting at the upper left asterisk (*). Cross a number to the right if it can be subtracted from the value to be coded with zero or positive remainder; otherwise, move down one position. If a subtraction can be performed the difference becomes the new value to be coded. Each move right gives a code letter 1, and each move down gives a code letter 0." After three 1's and 0's, the path ends at the lower right corner.

The matrix that is used to encode numbers into DC-free 20-bit codes is larger, but the same code-word-generation protocol is used. This matrix is shown below:

```
| start here
v
* 92378 * 48620 * 24310 * 11440 * 05005 * 02002 * 00715 * 00220 * 00055 * 00010 *
* 43758 * 24310 * 12870 * 06435 * 03003 * 01287 * 00495 * 00165 * 00045 * 00009 *
* 19448 * 11440 * 06435 * 03432 * 01716 * 00792 * 00330 * 00120 * 00036 * 00008 *
* 08008 * 05005 * 03003 * 01716 * 00924 * 00462 * 00210 * 00084 * 00028 * 00007 *
* 03003 * 02002 * 01287 * 00792 * 00462 * 00252 * 00126 * 00056 * 00021 * 00006 *
* 01001 * 00715 * 00495 * 00330 * 00210 * 00126 * 00070 * 00035 * 00015 * 00005 *
* 00286 * 00220 * 00165 * 00120 * 00084 * 00056 * 00035 * 00020 * 00010 * 00004 *
* 00066 * 00055 * 00045 * 00036 * 00028 * 00021 * 00015 * 00010 * 00006 * 00003 *
* 00011 * 00010 * 00009 * 00008 * 00007 * 00006 * 00005 * 00004 * 00003 * 00002 *
* 00001 * 00001 * 00001 * 00001 * 00001 * 00001 * 00001 * 00001 * 00001 * 00001 *
* 00000 * 00000 * 00000 * 00000 * 00000 * 00000 * 00000 * 00000 * 00000 * 00000 *
                                                                                ^
                                                                    finish here |
```