# EXTENDING THE SCALABLE COHERENT INTERFACE FOR LARGE-SCALE SHARED-MEMORY MULTIPROCESSORS

by

**ROSS EVAN JOHNSON**

Under the supervision of Professor James R. Goodman

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

Doctor of Philosophy
(Computer Sciences)

at the

**UNIVERSITY OF WISCONSIN-MADISON**

1993

# Abstract

Massively parallel machines promise to provide enormous computing power using an amalgamation of low-cost parts. We believe many of these will be shared-memory machines, since they do not burden the programmer with data placement and nonuniform access semantics. However, an efficient kiloprocessor solution for the shared-memory paradigm has proven elusive due to bottlenecks associated with parallel accesses to rapidly changing data.

The Scalable Coherent Interface (SCI) is an IEEE and ANSI standard for multiprocessors, specifying a topology-independent network and a cache-coherence protocol. The goal of this dissertation is to investigate ways to efficiently share frequently changing data among thousands of processors. SCI is the platform in which these methods are investigated.

Before investigating cache-coherence protocols, we demonstrate that an arbitrary topology can be constructed from a set of interwoven rings, such as SCI rings. This result is important because it would be impossible for SCI to realize the performance advantages of our new cache-coherence protocols without an efficient network. Our investigation of rings leads to a new scheme for deadlock avoidance that does not require resource partitioning. We also compare the performance of various topologies.

Next, we investigate two new cache-coherence protocols that employ trees of cache lines. The first adds shortcuts, called temporary pointers, to a list created by SCI. We investigate several protocol variations for request combining and temporary-pointer structures. We give lower bounds on latency for numerous sets of assumptions and compare these against the protocol variations. The second protocol transforms an SCI list into a probabilistically balanced, binary tree. Again, we investigate several protocol variations, comparing the performance to lower bounds that we derive, and we extend this protocol to support combining fetch-and-add without the severe network constraints of other combining mechanisms. We show that both protocols are compatible with SCI networks and correctly interoperate with SCI's cache-coherence protocol.

We compare the performance of our new protocols against SCI's protocol. We demonstrate that our protocols have a small increase in network traffic. They both have similar or lower latency than SCI, significantly lower when there is even a small amount of global sharing.

## Acknowledgements

I would first like to thank my family – especially Beth, who caught my interest on Sept. 30th, our common birth date, and married me during my quest for a PhD. Beth has provided me with constant encouragement, excellent lunches, and motivation to finish. Without this love, I would have taken longer. Our four parents, George and Bonnie Johnson and George and Mary English, surpassed all parenting requirements. They provided much love during our tender years, planned ahead so that we could both graduate without debt, and stayed together in the face of a rampant divorce rate. If Beth and I can do as well for our future children, I will be satisfied with my life. Our parents and three siblings, Alan Johnson, Lynn Johnson, and Dave English, have all brought joy to my life.

Next, I would like to thank those who have contributed to my technical growth – especially my advisor, Jim Goodman, who gave me a chance to choose my direction and prove myself. Jim's feedback and advice during my earlier graduate years have been invaluable. Members of the SCI working group, such as Dave James, Stein Gjessing, and Dave Gustavson, have been very supportive of my work, providing constructive criticism on preliminary drafts and work-in-progress presentations. I am also indebted to those who appointed me to chair the Kiloprocessors Extensions to SCI, an experience from which I gained valuable perspectives on leadership, group dynamics, and administration.

In addition to my advisor, many UW-Madison faculty and students have aided my technical and personal development. Steve Scott, now at Cray Research, gave his friendship and an example of technical excellence. Office mates Phil Woest, now at Northwestern University, and Nagi Aboulenein were willing to discuss both technical and personal issues. Mark Hill, Guri Sohi, Eric Bach, Mary Vernon, David Wood, Stuart Friedberg (now at Sequent), Parameswaran Ramanathan, Sarita and Vikram Adve, Alain Kagi, Rick Kessler (now at Cray Research), and others donated time to help my work.

Udi Manber was my best instructor in graduate school, giving me excitement for algorithms and the analysis of their complexity. Jim Holmes was my best instructor at Bethel College, giving me a love for higher mathematics. Unfortunately, too many professors sacrifice instruction to the god of research, where self publication is more important than the learning of others. I remember these, my worst instructors, equally well.

Outside of family and the ‘‘computer-geek’’ world, there were many who were important to me. God created opportunities for me, even though I would often take the credit myself. Members of Bible study groups, InterVarsity Christian Fellowship, and High Point Church helped me keep my perspective on life. Many roommates and friends, both during graduate and undergraduate years, created a safe environment for me to be myself. Mike Macdonald has been a recent encouragement. The junior-high youth at church gave Beth and me a common ministry in which we could meet.

Mary English, Dave Gustavson, and Dave James proof-read entire versions of this dissertation, without payment or obligation. The designated readers were Jim Goodman, Mark Hill, and Guri Sohi. Existing errors were later added by myself. This work was supported in part by a departmental teaching position, a research position through NSF Grant CCR-892766, and an equipment grant from Apple Computer, Inc.

# Table of Contents

# Table of Figures

# Chapter 1

# Introduction

## 1.1. Motivation

Exploiting parallelism is one way to make a faster computer. One way to exploit parallelism is by connecting together multiple processors to work on the same problem, called *multiprocessing*. Multiprocessing is becoming more and more attractive as the cost-performance ratio of microprocessors continues to plummet. As the number of processors per machine grows into the thousands and beyond, sequential execution of anything substantial must be avoided.

Our conceptual model of a multiprocessor is shown in Figure 1.1, where *nodes* are connected by a communication network and each node contains at most one processor, a memory hierarchy, and some I/O capabilities. Since processors are not required at every node, this model subsumes *dance-hall architectures*, where processors and memories are on opposite sides of the network. A multiprocessor network must efficiently support communication between nodes.

As processors cooperate to solve a problem, they share data. Since remote accesses to data are slow and use network resources, shared data should be replicated in the nodes where processors are using the data. However, data replication creates the problem of *data coherence:* when one processor changes its copy, nodes with other copies must be notified. The difficulty of maintaining coherence depends on the type of sharing that occurs. For example, data that is never changed is always coherent, but maintaining coherence of rapidly changing and widely used data is more difficult. We define *massively-shared data* to be any data that is rapidly changing and widely shared.

There are several mechanisms for replicating data. The choice of mechanism determines the range of other mechanisms available for maintaining data coherence. In a message-passing machine, messages are the primary mechanism for communication, relying on software to

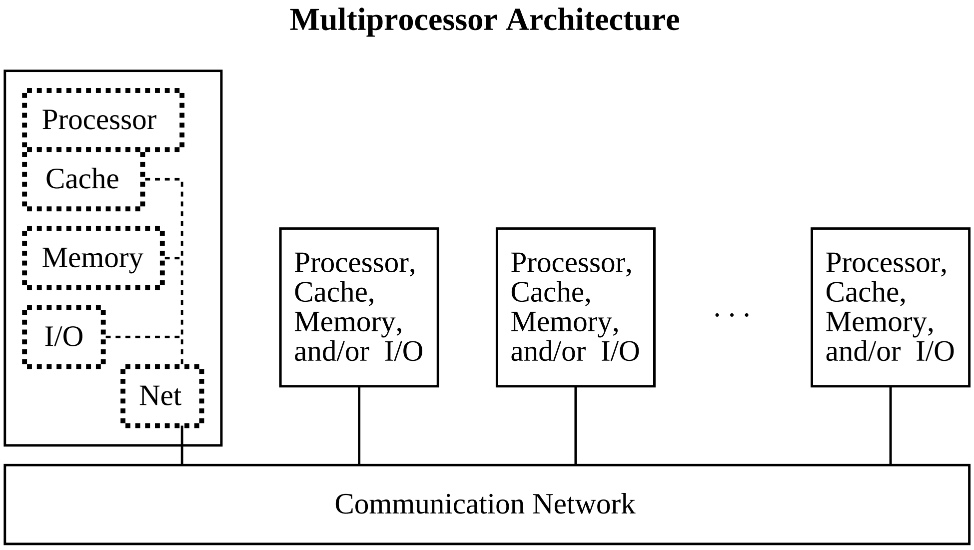



**Fig. 1.1:** This figure show our model of a multiprocessor architecture, where nodes are connected by a communication network and each node contains at most one processor, a memory hierarchy, and some I/O capabilities.

---

manage data replication and coherence. These machines are attractive to build because the communication hardware is relatively simple. However, these machines are difficult to program for some applications. The access semantics for local and remote data are different and it is not trivial for a programmer to efficiently and correctly manage the coherence of massively-shared data. As an alternative, a shared-memory multiprocessor provides uniform access semantics and the possibility of automatic data replication and coherence. It is possible to do everything with message passing, but it is more convenient to do everything with shared memory. We believe that multiprocessors should implement both paradigms, but we focus on shared memory because it is more difficult to implement[1].

[1]Shared memory implementations normally make use of an underlying message-passing mechanism. Simply providing a bare communication mechanism would be easier than adding a level of complexity to support shared memory.

When implementing shared memory in hardware, *caches* are often[2] used to replicate the most recently used data, grouped in *lines* (often 64 bytes) of nonoverlapping, contiguous memory. An instance of a line at a cache is called a *cache line*. Caches work well when programs exhibit temporal and spatial locality of reference [Smit82]. Caches not only reduce the time to access shared data, but they reduce network demands by avoiding remote accesses [Good83]. Data coherence between caches is known as *cache coherence*. We say that a multiprocessor system is *cache coherent* if a read access to any line always returns the most recently written value of that line [ArBa84]. A cache-coherence protocol uses network messages to replicate data and manage coherence in a cache coherent multiprocessor.

An efficient coherence protocol minimizes the number of network messages on the critical path, called *latency*, and minimizes the total number of network messages, called *traffic*. Unless otherwise stated, *average* latency and traffic refer to the arithmetic means. We use the standard notations of $O()$ and $\Omega()$ to respectively represent asymptotic upper and lower bounds[3]; $\Theta()$ implies both $O()$ and $\Omega()$. Data coherence in this dissertation is explored in the context of hardware cache coherence. However, our results are relevant to any system that must manage data coherence, such as message-passing operating systems, compilers, and/or application software.

Coherence protocols can update or invalidate stale copies to maintain coherence. Invalidate protocols appear to be in widespread favor and some [EgKa88, Scot92] have put forth data and/or arguments in favor of invalidate over update. Updates perform well when 1) a cached copy is reread after an update and 2) temporally close updates to the same cache line are collected before sending. The problem with updates is that all caches with stale copies are modified, even when the new data is not wanted. Since the bandwidth used per update grows

---

[2]Some multiprocessors, such as the NYU Ultracomputer [GGKM83] and the BBN Butterfly [ReTh86, LeSB88], do not use caches.

[3]A function $g(n)$ is $O(f(n))$ if constants $c$ and $N$ exist such that $g(n) \le c\, f(n)$ for all $n \ge N$. Similarly, $g(n)$ is $\Omega(f(n))$ if constants $c$ and $N$ exist such that $g(n) \ge c\, f(n)$ for all $n \ge N$ [Manb89].

with the size of the system, so does the waste. Therefore, we limit this dissertation to mechanisms for invalidation of stale copies.

A program's execution is sequentially consistent when all memory accesses appear to execute one at a time and in program order [Lamp79]. Naive programmers expect sequential consistency because it is implied by a single monolithic memory. Advanced programmers prefer to reason with sequential consistency because it is relatively simple. However, cache coherence by itself is not sufficient to guarantee sequential consistency. It may be violated in a cache-coherent multiprocessor if one processor is allowed to use the newer version of some data during the time it takes to update or invalidate stale copies and if other processors are allowed to use stale copies during that time. This is another argument against write update: it is difficult to implement updating writes that obey sequentially consistency over general networks.

There exist weaker memory models [DuSB86, DuSB88, AdHi90, GLLG90, DuSc90, Good91, GAGH92, Mosb93] that may provide better performance than sequential consistency. A complete discussion of these is beyond the scope of this dissertation. It suffices to say that many models make use of an mechanism called a *weak write.* A weak write allows a processor to continue execution after starting an invalidation and before all stale copies of the data have been invalidated. This is not allowed when the hardware implements sequential consistency [ScDu87]. The weaker memory models usually require a mechanism to test or signal when all copies are invalidated. We mention weak writes because this mechanism should be incorporated into a cache-coherence protocol.

An efficient cache-coherence protocol for thousands of processors must be scalable. Our notion of scalability for either hardware or software is that there is no sequential execution of anything substantial and that system cost grows slower than the square of the number of processors[4]. There is no widely accepted definition of scalability [Hill90] and so we review a sampling

---

[4]Concerning networks, our scalability notion allows multistage omega networks and disallows completely connected networks, both reviewed by Feng [Feng81]. Restricting the cost to $O(P \cdot polylog(P))$ would yield the same distinction for $P$ processors and $O(polylog(P)) = O(\log^k(P))$ for constant $k$. However, we would entertain a cost of $O(P^{1.5})$, although we know of no such networks.

of notions. On one hand, Amdahl [Amda67] observes that every program has a serial part. No matter how many processors are used, speedup of a program is limited by sequential execution of its serial part. This implies that scalability is useless. On the other hand, Gustafson [Gust88] observes that the time a person is willing to wait for an answer is relatively constant and, therefore, the amount of work to be completed can increase with the number of processors. This makes hardware scalability feasible for some programs because the parallel portion of a program can increase. Nussbaum and Agarwal [NuAg91] define hardware scalability as a comparison to a theoretical EREW PRAM [FoWy78] for a given application. This dissertation demonstrates that their exclusiveness of reads and writes is unnecessary. Scott [Scot92] gives an informal definition of scalability based on cost, delays, and bandwidth. His definition is essentially the same as ours except that he separates network scalability into bandwidth and message delay. We informally include bandwidth and message delay in our assertion of no sequential execution. Although scalability arguments are not a substitute for performance studies with real programs and thousands of processors, scalability is a useful notion for anticipating performance in the absence of such studies.

Section 1.2 explains the numbering, citation, and directive conventions in this dissertation. Section 1.3 reviews related work, including a number of proposed cache-coherence protocols. These protocols are classified according to how they store a *sharing set,* which is the set of caches that have copies of a particular line. We show why many protocols are unacceptable for kiloprocessor systems. Of particular interest is the cache-coherence protocol of the Scalable Coherent Interface (SCI), which this dissertation extends. Section 1.4 gives the history of SCI and the history of its Kiloprocessor Extensions, where this dissertation is immediately applicable. Section 1.5 summarizes the contents of later chapters and appendices.

## 1.2. Conventions

The numbering, citation, and directive conventions in this dissertation are as follows. Pages, footnotes, and chapters are each numbered sequentially, starting at Chapter 1. Equations and figures are each numbered sequentially, restarting at each chapter, and preceded by the chapter number. Section numbers give a hierarchy, including the chapter number. A citation string is constructed from authors' last names and the date of publication. Complete citations and an index of citation strings appear at the end of the dissertation. Throughout the dissertation, special reading directives are given to the nonmathematical reader. These directives help the first-time reader to skip equation derivations without missing other significant content. Later, the reader can return to these sections and study the equations.

## 1.3. Related Work

In this section, we review related work and describe a number of cache-coherence protocols, focusing on their usefulness for kiloprocessor systems. A coherence protocol maintains a sharing set so that particular copies can be invalidated when data is changed. This set is managed whenever a cache is *inserted* into the set (for reading) or *deleted* from the set (for cache-line replacement), as well as when the set is *purged* (reduced to size one for writing). We next describe four classes of hardware cache-coherence protocols: snooping, hierarchical, directory, and distributed-pointer protocols. After that, we discuss related work in data structures.

### 1.3.1. Snooping Protocols

Snooping protocols [Good83, RuSe84, PaPa84, KEWP85, SwSm86, ArBa86, ThSt87] rely on the broadcast capability of a shared bus. Each cache remembers to which sharing sets it belongs (implicit by the lines that are stored), as well as the notions of data validity, exclusiveness, and ownership for each cache line. All cache controllers observe every bus transaction, called snooping, and take the appropriate action(s) when necessary. For example, the owner cache supplies requested data when the memory is not valid. All nonrequesting caches purge their copies when there is a request for exclusive access. The simplicity of a snooping protocol is facilitated by a shared bus. However, the one-at-a-time nature of a shared bus is a bottleneck for large multiprocessors. Actual systems, like the Sequent Symmetry [LoTh88], Sequent Balance, Encore Multimax [Enco86], DEC Firefly [ThSt87], Xerox Dragon [McCr84], Berkeley SPUR, and SGI Power Series [Sili89], have been limited to at most 32 processors and often much less.

### 1.3.2. Hierarchical Protocols

Hierarchical protocols make use of the cache hierarchy implied by a tree of buses [Wils87, BaWa88, VeJS89, ChGB91, YaTB92]. A cache hierarchy on a *k*-ary *n*-cube, implemented as either a grid of buses [GoWo88] or a set of rings [Scot91, Scot92], has a different tree per memory module so that the tree root is not a network bottleneck. The network and the data's home location determine the tree's structure, but the coherence protocol determines how the sharing set is stored in the given tree.

One naive way to store the sharing set is to maintain a vector per line per node. A vector contains *k* bits, where *k* is the branching factor of the tree, and each bit indicates whether or not there is a copy in the corresponding subtree. Invalidates are multicast from the tree root, entering only those subtrees that have copies. Requests for data can be combined in the tree if requests visit every cache on the way to the root [Scot92]. The storage for this naive scheme is prohibitive, $O(k N^2)$ bits for $N$ nodes, and two alternatives with less storage have been proposed, discussed later.

Tree neighbors in a hierarchical protocol are also network neighbors. This improves performance by reducing the average message latency. In sparsely populated trees, however, purge messages must still exit the network and do a lookup in every cache on the way to the leaves, even when an intermediate cache does not have a copy to be purged. This extra delay is on the critical path for the common cases of small sharing sets. Furthermore, extra lookups are a potential bottleneck as vector activity increases with system size.

Deadlock avoidance is not trivial in a hierarchical protocol on a *k*-ary *n*-cube. When forwarding purges (or combining insertions), storage is required at each node in the hierarchy and this storage is not released until a response is returned from the leaves (or root). When storage is exhausted, additional requests must wait. Since trees have different roots, circular wait and deadlock can result. Scott [Scot92] proposes the use of an *outstanding invalidation buffer* (OIB) at each node and allocates storage based on a cache line's level in the line's tree. This is a variant of the standard solution for deadlock avoidance in store-and-forward networks [Tane81], where the storage is divided into classes and the assignment of space is restricted, providing a partial order on the use of the classes.

### 1.3.2.1. Multilevel Inclusion

One way to avoid the storage overhead of the naive solution is to maintain multilevel inclusion (MLI) [LaMu79, Wils87, BaWa88], where a cache stores a superset of all the lines stored in descendant caches. When a cache line is purged, the purge is propagated to only those children that have copies. To save space, a parent need only store a vector if children have copies. To avoid storing data for every inclusion, these vectors can be stored in a separate *inclusion cache.* If it is infeasible to guarantee that there will always be space in the inclusion cache [BaWa88], as argued by Scott [Scot92] for grids of buses, then older children can be purged to make room for newer children [Wils87].

#### 1.3.2.2. Pruning Caches

A second way to avoid the storage overhead of the naive solution is to use *pruning caches* [GoWo88, Scot91, Scot92], which are inclusion caches that do not maintain MLI. Whereas a missing vector in an inclusion cache implies that no subtree has a copy, a missing vector in a pruning cache makes the conservative assumption that all subtrees have copies. The performance of pruning caches depends on the hit rate; a low hit rate results in extra traffic. Whereas an inclusion cache requires MLI for correctness, a pruning cache relies on MLI only for good performance. Using comparably sized caches, Scott's simulations [Scot92] show that systems with pruning caches perform better than systems with inclusion caches. This is because the inclusion caches must purge useful copies in order to maintain MLI.

### 1.3.3. Directory Protocols

A directory protocol stores each line's sharing set in a *directory* at memory. A directory protocol stores an entire sharing set at the memory; later, in Section 1.3.4, we will discuss sharing sets that are distributed throughout the system. Caches insert and delete themselves from the sharing set so that later purges can be limited to the caches that share data. A directory protocol can be implemented on top of a point-to-point network with any topology, thereby avoiding the bottleneck of a centralized resource (except for the central-directory protocol of Tang [Tang76]). Censier and Feautrier [CeFe78] store the sharing set as a bit map per directory, where a bit for each cache indicates whether the cache has a copy of the line. A version of this is implemented in the Stanford DASH [LLGG90].

Directory protocols have a performance problem in that each directory serializes insertions and deletions and the network connection serializes the messages for purges. This problem has not been highlighted in current performance studies [ASHH88, EgKa88, OKNe90, CFKA90, GHGM91, GuWe92] because they have focused on the performance of small machines with at least one study [GHGM91] admitting the assumption that shared instructions hit in the cache. With thousands of processors, anything sequential is unacceptable, unless the case is extremely infrequent. Matloff [Matl91] claims that scalability is not important, arguing that most software can be rewritten to group the sharing. Other studies [CFKA90, GuWe92] also suggest rewriting software that does not scale. However, the point of the shared-memory paradigm is to make it easier to generate and reuse correct and efficient programs.

The storage for a bit-map implementation of each directory [CeFe78, LLGG90] grows in proportion to the number of caches in the system, making this implementation too costly for kiloprocessor systems. A number of alternatives have been proposed.

#### 1.3.3.1. Limited Directories

*Limited directories* reduce the directory storage by maintaining only $i$ cache pointers for the sharing set. If the sharing set overflows on an insertion, then either an overflow bit is set or else one of the elements is purged to make room for a new one. If the overflow bit is set, then future purges are broadcast to all caches in the system. Agarwal et al. [ASHH88] classify directory protocols with $Dir_iX$, where $i$ is the number of pointers and $X$ is either $B$ or $NB$ for broadcast or no broadcast respectively. Full-map protocols [Tang76, CeFe78, LLGG90] function as $\mathrm{Dir}_n\mathrm{NB}$, where $n$ is the maximum number of data locations. The two-bit solution of Archibald and Baer [ArBa84] is $\mathrm{Dir}_0\mathrm{B}$. One disadvantage of limited directories is that they do not allow all caches to share the same data, such as shared instructions.

#### 1.3.3.2. Directory Groupings

Three directory schemes reduce the storage requirements by grouping the sharing sets for different lines. *Sectored directories*, proposed by O'Krafka and Newton [OKNe90], reduce the directory storage by grouping lines of the same memory module into *superlines* (our term). Although a state machine is maintained per line, a sharing set per superline is stored as the union of the lines' bit maps plus one owner pointer per line. This reduces the storage overhead by a constant factor of $x$ for large systems, where $x$ is the number of lines per superline. An earlier and more complex version of sectored directories is proposed by Censier and Feautrier [CeFe78], where each cache stores one counter per superline so that a cache's last cache-line replacement of the given superline can reset the corresponding bit in the directory. However, Censier and Feautrier do not mention the directory's need for an owner pointer per superline, limiting their proposal to networks that allow snooping.

*Coarse vectors*, independently proposed by Gupta et al. [GuWM90] and Brooks and Hoag [BrHo90], group storage by caches. Instead of maintaining a bit map of size proportional to the number of caches, each bit represents a group of caches. The bit is set whenever any cache in the group has a copy of the data. Gupta et al. recommend the use of limited pointers until the directory overflows, after which coarse vectors are used.

The *superset scheme*, proposed by Agarwal et al. [ASHH88], stores one pointer and one bit mask of the same size. The first set insertion determines the pointer and sets all bits in the mask. Each subsequent insertion unsets the corresponding mask bit for every bit of the insertion pointer that is different from the directory pointer. A purge is sent to all caches such that the bitwise-and of the mask and the cache pointer is the same as the bitwise-and of the mask and the directory pointer. The pointer and mask represent a size-$2^i$ superset of the caches that have a copy, where $i$ is the number of unset bits in the mask. Gupta et al. [GuWM90] claim that the superset scheme performs poorly compared to a full bit map.

#### 1.3.3.3. Software Support

Two directory schemes reduce directory storage by invoking a trap to software for problem cases. The LimitLESS Directories of Chaiken et al. [ChKA91], implemented in the MIT Alewife, use limited directories of size $i$ and trap to software when a pointer overflow occurs. LimitLESS stands for limited directories that are Locally Extended through Software Support. Software manages large sharing sets correctly with the hope that the common cases will be smaller sharing sets and, therefore, handled in faster hardware. We classify LimitLESS directories as $\mathrm{Dir}_i\mathrm{SW}$, where *SW* stands for SoftWare support.

*Cooperative shared memory* of Hill et al. [HLRW92] suggests using a cooperation between software hints and very simple directory hardware. The directory hardware is classified as

$\mathrm{Dir}_1\mathrm{SW}$, where one integer stores either a pointer to the exclusive copy or the number of shared copies. Software uses hints, called CICO (Check-In and Check-Out shared or exclusive), to help the directory hardware manage the common cases. For better performance, the software can specify prefetch hints for shared or exclusive data. With accurate hints, the single pointer/counter is sufficient to efficiently maintain coherence. For incorrect and missing hints, the directory traps to software to correctly handle the problem. The authors claim that inserting the CICO hints is a programming advantage because the hints help the programmer reason about the sharing that occurs. CICO hints have the advantage over software coherence in that they do not affect memory semantics and are not required for correctness. The authors show that careful insertion of hints by the programmer can nearly eliminate traps in the SPLASH benchmarks [SiWG92].

#### 1.3.3.4. Pointer Caches

Two directory schemes reduce storage by sharing a pointer cache for sharing sets in the same memory module. *Dynamic pointer allocation* of Simoni and Horowitz [SiHo91] uses a pointer cache that is shared by all directories at the same memory module. The pointers in the cache are organized as a circular linked list. Insertions, deletions, and purges are simple list functions, although deletions and purges must sequentially search/traverse the list. If an insertion finds a full pointer cache, then one (or some) of the cache lines is (are) purged to make room. To avoid frequent overflow, the authors suggest a 4 to 1 ratio of pointers to lines for 16-byte lines and an 8 to 1 ratio for 32-byte lines. Dynamic pointer allocation is classified as $\mathrm{Dir}_n\mathrm{NB}$.

The *tag cache* scheme of O'Krafka and Newton [OKNe90] uses two associative pointer caches. One cache has many entries, each storing a small number of pointers. The other cache has few entries, each storing a full bit map. When a line is first cached, it is allocated an entry of pointers. When the pointers overflow, it is allocated a bit-map entry and the pointer entry is freed. When one of the caches is full, an entry is purged to make room. Gupta et al. [GuWM90] independently discover a pointer cache, which they call a *sparse directory*, in combination with coarse vectors.

### 1.3.4. Distributed-Pointer Protocols

Instead of maintaining lots of cache pointers (or a bit map) at the memory, the sharing set can be distributed as a structure of cache lines that point to each other [JLGS90, ThDe90a, NiSt92]. Storage for these *distributed-pointer protocols* (DP protocols) is dynamically allocated as caches are inserted and deleted, avoiding static allocation based on worst-case assumptions. The directory (per line) stores one pointer at the memory and each cache line stores one or more pointers, depending on the specific protocol.

DP protocols have a number of advantages over directory and hierarchical protocols. Primarily, a DP directory can be very simple and completely ignorant of the particular DP protocol, requiring no counters, no traps, no request forwarding, and only one pointer and four states. The pointer indicates the owning cache and the four states, described later in Section 5.2, record whether the data is cached and whether the memory's data is valid. Therefore, a DP directory can be simpler than the directories of all previously proposed directory and hierarchical protocols, including the $Dir_0B$ and $Dir_1X$ varieties [ArBa84, ASHH88, HLRW92]. Nearly all of the complexity of a DP protocol is in the cache controllers. By using a programmable entity for a cache controller, DP protocols can be debugged and enhanced without affecting the fast and simple directories that are implemented in hardware.

The simple directory hardware results in four advantages: interoperability, network independence, combining simplicity, and pairwise sharing. First, since all pure DP protocols have the same directory hardware, it is possible to construct numerous DP protocols that can correctly interoperate on the same multiprocessor. This is useful, for example, when adding new processor-cache boards to an existing multiprocessor; nonprogrammable hardware need not be discarded.

Second, since the directory only responds to requests (neither forwards nor generates requests), it is possible to construct DP protocols that use only request-response transactions. Such a protocol is decoupled from the network design because special *forward-progress* (deadlock- and starvation-avoidance) mechanisms are not required, other than for two-phase transactions. For example, recall that Scott [Scot92] briefly addresses deadlock for hierarchical protocols and recommends partitioning of queue resources [Tane81]. This solution required approximately $2h$ partitions, where $h$ is the height of the hierarchy, because a separate partition is required for every time a message is forwarded during a purge. In contrast, a protocol that obeys the request-response paradigm requires only two partitions, one for requests and one for responses. The guarantee of forward progress, not just deadlock avoidance, is necessary for real-time scheduling and/or any claims that an algorithm will terminate in a given number of steps.

Third, there exists a request-combining mechanism for DP protocols [JLGS90] that is much simpler and more general than previously proposed mechanisms for hardware combining. Software combining mechanisms have been proposed [YeTL87, YeTa90] and the first software-combining algorithm for fetch-and-add is given by Goodman et al. [GoVW89]. Previous attempts at hardware-combining fetch-and-add [GGKM83, PBGH85] are cumbersome and expensive[5]. These hardware mechanisms saved state in the network, constrained the return path of responses, and required separate combining networks to avoid slowing normal memory accesses in the RP3 [PBGH85]. Scott's read combining [Scot92] does not require a separate combining network, but combinable reads are slower than normal reads. Furthermore, these mechanisms are built for only one primitive, fetch-and-add or read. Combining other primitives would require additional network hardware per primitive. In contrast, a DP directory need only return a pointer to the owner cache (after which the requester gets the data from another cache), so a combining pointer swap can be used to avoid congestion at the memory. One response is returned immediately at the point of combining, creating a list of would-be owners: the combining accepts two requests and outputs one combined request and one response. More details are given in Sections 3.2.1 and 5.3.2.1. This mechanism does not have any of the above problems (network state, constrained response paths, and separate networks) and the same mechanism can be used for any combinable memory operation without additional modifications to the network.

---

[5]Pfister and Norton [PfNo85] claim that combining increases switch cost by a factor of 6 to 32 in the RP3 [PBGH85] and it was not completed [SoSG89]. However, Dickey and Kenner [DiKe92] argue that the costs are more modest.

Fourth, caches can cooperate to avoid consulting the directory, such as for pairwise sharing. Pairwise sharing is when only two caches share data, which is a special case of single-invalidate data. When sharing data between two caches, a writer leaves one stale copy in the sharing set, marked as stale. Later, the stale-copy cache requests the data directly from the valid-copy cache, thereby avoiding a directory access. This cuts the latency approximately in half for both reading and writing. Pairwise sharing is not possible for directory and hierarchical protocols that do not have cache-to-cache pointers.

#### 1.3.4.1. Scalable Coherent Interface

Scalable Coherent Interface (SCI) [JLGS90, Gust92a] is an ANSI (American National Standards Institute) and IEEE (Institute of Electrical and Electronics Engineers) standard (IEEE Std 1596-1992) [Comm91]. It specifies a topology-independent network and a cache-coherence protocol[6]. SCI distributes a sharing set as a noncircular, doubly-linked list of cache lines, requiring two pointers per cache line. New insertions become the owner by contacting the directory and then contacting the previous owner, its *forward neighbor*, thereby becoming the next owner or *head.* The oldest cache line in the list is the *tail* and the second neighbor (closer to the head) is the *backward neighbor.* A deletion from the list connects two neighbors to each other. Purges are sequential after ownership is acquired. The correctness of SCI's protocols is being tested through time-intensive random testing at independent locations[7] and formal verification of the coherence protocols has begun at the University of Oslo in Norway [GjKM89, GjKM90a, GjKM90b]. SCI is robust in that it guarantees forward progress (no deadlock and no starvation for any processor) and software is able to recover from arbitrary transmission failures. Although SCI has good performance for small sharing levels, the sequential nature of its linked lists is a bottleneck for large sharing levels.

---

[6]SCI also specifies a message passing mechanism, but its existence is of limited interest for this dissertation due to the relative simplicity of implementing message passing.

[7]Some testing cites include, but are not limited to, Apple Computer Inc. (David James: dvj@apple.com), The University of Wisconsin-Madison (Alain Kagi: alain@cs.wisc.edu), and The University of Oslo in Norway (Sverre Johansen: sj@ifi.uio.no).

As part of every SCI cache controller, there is at least one request controller and at least one response controller. A *request controller* accepts a command from a processor (or other entity, like the network) and generates the required requests according to the cache-coherence protocol. The SCI network guarantees that the request will eventually be delivered (forward progress) and so the request controller can wait for the eventual response. Several request-response transactions can be sequentially performed by a request controller. When the protocol requirements are satisfied, the request controller is free to accept more processor commands. Similarly, a *response controller* accepts a request from the network and generates the required response; the SCI network guarantees that a response will eventually be delivered.

In general, request forwarding is forbidden by SCI networks because forwarding resources are necessarily finite. For example, if a request would generate another request in the same node and if there are no more resources available for sending requests, then the response controller must wait for request-controller resources to become available. Suppose that the given request controller is waiting for a response controller in another node, which is also waiting. Continuing with this reasoning, we can construct an example of circular wait and deadlock. Therefore, request forwarding without proper precautions can lead to deadlock in SCI. One way to avoid this deadlock is to create a finite queue of waiting requests such that the queue never overflows. Preventing queue overflow is discussed in Section 5.5.2.

#### 1.3.4.2. Stanford Distributed-Directory Protocol

In contrast to SCI, the Stanford Distributed-Directory Protocol (SDD) of Thapar and Delagi [ThDe90a, ThDe90b] distributes a sharing set as a singly linked list, requiring only one pointer per cache line. Deletions are more costly than for SCI because half of the list, on average, must be purged. Unlike SCI, SDD insertions are forwarded in a triangle: reader, directory, previous owner, reader. Although this reduces the latency of reads, this three-phase transaction requires additional network support to avoid deadlock. SDD does not currently support pairwise sharing and it does not perform well for large sharing levels.

### 1.3.4.3. Scalable Tree Protocol

The Scalable Tree Protocol (STP) of Nilsson and Stenström [NiSt92] distributes a sharing set as a tree of cache lines. STP maintains optimal height trees for efficient purging. Although trees suggest the potential of parallel insertions and deletions, STP serializes insertions and deletions. Currently, STP does not support pairwise sharing. STP requires five pointers per cache line and, unlike SCI and SDD, requires five pointers per directory. STP is also different from SCI and SDD because insertion requests always get the data from the memory. If the memory data is not valid, the directory gets data from the previous writer before responding to the insertion request. This makes STP incompatible with the DP combining mechanism. Furthermore, numerous multi-phase transactions are used in the protocol, requiring additional network support to avoid deadlock, as discussed later in Section 6.2.1.2.

### 1.3.4.4. Wisconsin STEM

The Wisconsin STEM (permuted acronym for Tree Merging Extensions to SCI) of this dissertation is a DP protocol that distributes a sharing set as a tree. Each cache line stores three pointers and one 5-bit height[8]. Unlike STP, insertions and deletions can proceed in parallel and a network with nondeadlocking two-phase transactions is sufficient to avoid deadlock. The DP combining mechanism is useful for performance, but not required for correctness. STEM correctly interoperates with nodes that implement the SCI protocol so that STEM nodes can be added to existing SCI systems. As shown later in Sections 6.3.2 and 6.4.3, STEM's latency is less than or equal to SCI's latency with no network load. Because of these advantages, STEM is being considered by the SCI working group [Gust91] as an extension to SCI. Unlike hierarchical protocols, tree neighbors in STEM are not necessarily network neighbors. This is a performance disadvantage for large sharing sets. However, STEM (and SCI) has (have) simple directories, network independence, interoperability with SCI, a pairwise-sharing optimization, and a simple and general combining mechanism.

---

[8]A 5-bit height is sufficient for 64K nodes, as discussed in Section 5.6.5.

### 1.3.5. Data Structures

Balanced trees have been widely researched as a structure for storing information, beginning with AVL trees [AdLa62]. However, the standard uses of trees lead to tree protocols that are inappropriate for cache coherence. The parallelizing of tree operations has been spear-headed by the database community [BaSc77, Elli80a, Elli80b, KuLe80, LeYa81, FoCa84, Sagi85]. For database applications, a *search tree* must support efficient insertion, deletion, and searching. Due to the sorted nature of a search tree, all insertions and searches invariably start at the root. The hot spot at the root makes all of these algorithms inappropriate for efficient cache-coherence protocols.

The theory community [MoIy85, DePI86, AKLM89, KiPr90] constructs various optimal and near optimal search trees for various applications, such as Huffman encoding. However, these theoretical algorithms make unrealistic assumptions about communication through shared memory, with at least two [MoIy85, DePI86] assuming that the data structure is stored in an array. A cache-coherence protocol can not assume shared memory, since that is what the protocol is trying to implement.

Pugh [Pugh90] shows how lists can be augmented with additional pointers so that short cuts across the list can be used to access data more quickly. Each node in the list can be limited to at most $\lceil \log_2(N) \rceil$ pointers, where $N$ is the maximum size of the list; randomness is used to determine the actual number of pointer at each node as it is inserted. Pugh shows how to do sequential insertions, deletions, and searches in $O(log(n))$ expected time, where $n \le N$ is the current number of nodes in the list. However, Pugh's *skip lists* are not easily modified for extending SCI because 1) each node can not be limited to a constant number of pointers without giving $O(n)$ latency and 2) there is no obvious way to parallelize insertions and neighboring deletions.

Whereas search trees (and heaps) store sorted information, a *coherence tree* stores multiple copies of the same information, so there is no need for searching and sorting. The purpose of a coherence tree is to efficiently distribute and purge copies of the same data. The required operations on a coherence tree are insertion, deletion, and purging. Insertion adds one node to the tree and obtains a copy from another node in the tree. Deletion removes one node from the tree. Purging deletes all but one node in the tree so that the data can be changed. Balanced heights are still important for efficient tree operations, but the restrictions of searching and sorting are discarded. In order to avoid an insertion hot spot, a combining mechanism is required.

### 1.4. SCI History

SCI specifies a topology-independent network and a cache-coherence protocol. The network is made of point-to-point links that each support 1 gigabyte. The links may be connected to form rings (to support modest-performance, high-volume applications), connected through switches (for more expensive high-performance applications), or may be formed into meshes of rings (for intermediate applications). A single ring can efficiently simulate a bus[9]. SCI's cache-coherence protocol is fully distributed, using linked lists of cache lines to keep track of cached copies. Using lists is efficient when the sharing level is low. SCI is ''industrial strength'' in that forward progress is guaranteed and it is possible to recover from an arbitrary number of transmission failures. The standard is specified in executable C code [KeRi88] to reduce ambiguity, simplify testing, and enable accurate simulation. The information contained in the following history has been taken from Gustavson [Gust92a] (chair of the SCI working group), the foreword to the SCI standard [Comm91], and the minutes of the SCI meetings [Gust91]. In a few cases, information has been acquired through informal interviews of meeting participants and checked against the memories of other participants.

#### 1.4.1. Standardization of SCI

Dave Gustavson [Gust92b] states:

> Standardization in the computer field has changed radically since the mid-1970's. Standardization used to collect industrial practice into a publicly maintained accessible document, compromising among conflicting practices and requirements to achieve interchangeability in future products. That mechanism is of little use in the modern computer industry, however: by

[9]A register ring [Stal84] holds promise for higher throughput than a backplane bus because the sequential broadcast of a bus is a bottleneck and the cycle time of a bus is limited by distributed capacitances and the speed of light. Given reasonable cycle times, Scott et al. [ScGV92] show that a single ring does indeed have better performance than a single bus, both for throughput and message delay. The KSR1 uses what its designers [Burk92] call an insertion-modification token ring.

> the time such a standard is complete, the technology it standardized is usually obsolete and irrelevant.
>
> Therefore it has become necessary to be more pro-active, creating and inventing in the standardization process, in order to have a chance at relevance. This is not always successful, of course. Serial Bus (P1394, nearly finished now) and Futurebus (IEEE Std 896-1991) restarted several times during their long evolution, and may have missed their market windows; FastBus (IEEE Std 960-1986) took about 8 years and developed a very limited market acceptance; SCI (IEEE Std 1596-1992) took 4 years and seems likely to be ready with the needed infrastructure as its market window opens.
>
> Though both traditional and pro-active work is called ‘‘standardization,’’ the nature of the work is very different. The latter has many aspects of invention and research, and sits at the leading edge of technology. Some researchers are still unaware of this new kind of standardization, and categorically refuse to participate in standardization work, since they erroneously believe it to be ‘‘boring and tedious.’’

We agree with Gustavson. Some standards work for computers has changed, containing more innovative ideas and becoming more aligned with traditional research methods. It is a pity that some researchers refuse to accept the changing personality of standards work.

The SCI standards project grew out of the Futurebus attempts, which started in the late 1970’s, to provide the ultimate bus for 32-bit processors. The SuperBus Study Group first met in November 1987 to study buses that could sustain a data bandwidth of 1 gigabyte per second. Paul Sweazey, the coordinator of the Futurebus cache-coherence task group, lead this group under the IEEE Computer Society’s Microprocessor Standards Committee (MSC). In less than a year, this new study group realized that there was something better than a bus, namely a register ring [Stal84], as suggested by Manolis Katevenis. In October 1988, P1596’s Project Authorization Request (PAR) was approved, with David Gustavson as working-group chair, and SCI was officially born.

The SCI standards project converged rapidly due to a small number of hard-working individuals, especially Dave James, who generated documents at an incredible rate and single-handedly created the bulk of the SCI text by June 1989. Up-to-date documents and small meetings increased productivity and the meetings' monthly frequency, as well as substantial between-meeting work, prevented the loss of momentum. The group worked entirely by consensus, always choosing a technically superior solution, rather than being forced, by industrial politics, into sub-optimal choices. This working format, as well as the technically interesting topics, attracted a large number of experts in SCI-related fields, adding credibility to the resulting SCI specification.

In January 1991, the working group held its only vote, voting unanimously to submit the draft specification to the MSC for forwarding to the balloting body. The ballot passed, 92 percent affirmative, and all but one objection was later resolved (the working group refused to change the C code to Pascal). After minor fixes and a recirculation to the balloting body, final approval by the IEEE Standards Board was secured in March 1992. SCI was completed, from PAR to IEEE standard, in less than four years, a noteworthy achievement, considering that previous standards of this type generally evolved in eight to twelve years. SCI became an approved American National Standards Institute (ANSI) standard in October 1992 and international standardization by the International Electrotechnical Commission (IEC)[10] is in progress.

During SCI's push towards completion, a number of SCI-related standards projects were spawned. In April 1989, the CSR Architecture (P1212), chaired by Dave James, was started to specify a unified Control and Status Register and I/O Architecture. The P1212 project was shared by Futurebus+ (IEEE 896.x), SerialBus (P1394), and others. P1212 began its balloting in September 1990. In March 1989, a Fiber Optic Task Group (SCI-FI) was started, chaired by Hans Wiggers, and an SCI/Futurebus+ Bridge Task Group was also started, led by Mark Williams. In November 1990, the first meeting for an SCI/VME Bridge Architecture (P1596.1) was held, chaired by Ernst Kristiansen.

In June 1991, three new PARs were approved. First, the Kiloprocessor Extensions to SCI, P1596.2, was approved to define protocols that would reduce SCI's memory-access latency when large numbers of processors share data. This project, described in more detail later, was

[10]In French, it is the Commission Electrotechnique Internationale (CEI).

initially chaired by Ross Johnson and is now chaired by Stein Gjessing. Second, the Low-Voltage Differential Signals project (P1596.3) was approved to specify low-voltage differential signals for CMOS, GaAs, and BiCMOS logic arrays, chaired by Gary Murdock. Third, RAM Link (P1596.4) was approved, initially chaired by Hans Wiggers. Later, the Shared-Data Formats project (P1596.5) was started, chaired by Dave James; it is in the balloting processes as of August 1992. Overall, SCI has generated significant interest in and contributions by the academic and standards communities.

The rapid acceptance of SCI and the activity on related standards has generated significant interest in the industrial community. Due to the competitive nature of the computer industry, a number of SCI related products are secretly under development. However, some industrial developments are public knowledge: In November 1990, Hewlett-Packard decided to release its gigabit serial link specification for use by SCI-FI, followed by an SCI-compatible product, G-Link, which became available in July 1992 and has already seen extensive use in non-SCI applications. Dolphin SCI Technology of Oslo, Norway promises an SCI chip set in GaAs, with availability expected in early 1993. In May 1992, an international supercomputer company, Convex Computer, announced an agreement to use Dolphin's SCI technology in their next generation multiprocessors. In December 1992, LSI Logic announced CMOS support for SCI, to be developed with Dolphin SCI Technology. LSI Logic promises to provide both stand-alone chips and SCI "core-ware" libraries for VLSI, with availability of some products expected as early as second quarter of 1993. Contact SCI's working-group chairman, David Gustavson (dbg@slac.stanford.edu), for more complete and up-to-date developments and vendor-contact information.

### 1.4.2. Kiloprocessor Extensions to SCI

As mentioned above, the working group for the Kiloprocessor Extensions to SCI (P1596.2) was officially started in June 1991. However, work on these extensions has continued throughout the development of the SCI standard. The official purpose and scope of these extensions are given in Figure 1.2.

In September 1989, the working-group first began to admit that SCI systems are not scalable to 64K nodes, due to the linear performance of the sharing list. In October 1989, Guri Sohi

**P1596.2 Charter**

Standard for
Cache Optimizations for Large Numbers of Processors
using the Scalable Coherent Interface

Purpose of Proposed Standard:
The IEEE-P1596 (SCI) draft 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 processors.

Scope of Proposed Standard:
[The standard will] define protocols for reducing the latency of accessing cached data shared by large numbers of processors. This involves combining multiple coherent requests into one before they reach a shared memory controller. These protocols will generate tree-like sharing structures compatible with the linear structures defined by IEEE-P1596. 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-and-add).

**Fig. 1.2:** quoted from the IEEE Standards Project Authorization Request for P1596.2.

---

suggested the use of Recursive Doubling for $N$-node distribution of data with $O(\log(N))$ latency [Lebe91, LeSo92], as described by Hillis and Steele [HiSt86] and earlier by Stone [Ston73]. At the same time Sohi proposed the combining mechanism [JLGS90] for creating the SCI list structure (and possibly fetch-and-add), thereby avoiding congestion at the memory controller. These contributions, recursive doubling and combining, became the backbone of a changing solution. Sohi also suggested that combining fetch-and-add may be possible and, if so, possibly simpler than in the NYU's Ultracomputer. In November, Jim Goodman showed how fetch-and-add could be integrated with combining. In December 1989, the working group decided to separate the Kiloprocessor Extensions from the SCI standard so that SCI would not be delayed by the research needed to complete the Kiloprocessor Extensions.

In March 1990, Jim Goodman showed the group, on behalf of Ross Johnson, how temporary pointers could be created during combining. At the same meeting Stein Gjessing showed how temporary pointers could be created at the memory controller. In September 1990, Kurt Baty and Jim Goodman bet dinner on who could create the best topology for large systems, using processors with one SCI port and switches with four ports. In April 1991, Baty bought dinner for Ross Johnson, one of the late additions to the Baty-Goodman contest.

For one and a half years, since the fall of 1989, Ross Johnson, Dave James, Jim Goodman, Stein Gjessing, and other working-group members contributed to the refinement of Recursive Doubling, reducing its message traffic and making it compatible with SCI. In the process, Dave James and Ross Johnson proposed various tree structures that were compatible with Recursive Doubling. In April 1991, Ross Johnson presented the total cache-coherence solution for kiloprocessor SCI systems, using patient combining, recursive doubling, and combining of fetch-and-add operations. This presentation is expanded in Chapters 3 and 4. The PAR for the Kiloprocessor Extensions to SCI was submitted to the MSC in May. However, the modified version of Recursive Doubling greatly complicates the combining mechanisms and its performance becomes bad as trees age, so it was discarded in favor of maintaining nonaging binary trees.

Tom Knight first proposed nonaging binary trees for cache coherence at the June 1988 SCI meeting, officially a SuperBus Study Group meeting. Knight [Knig92] says:

> Indeed, I started thinking about linked list cache protocols long before SCI existed; I would attribute most of the novel ideas in linked caches originally to the work of Jeff Boughton and others at the S-1 project at LLNL [Lawrence Livermore National Laboratory]. They wrote some initial ideas down in their proposal for the S-1 Mark II-B, which was, of course, nothing like either the S-1, or the S-1 Mark II. In their scheme, they had a high speed ring network and a simple (by SCI standards) cache protocol, using linear linked lists. I got excited about using those protocols in circumstances with more general communications, and made what I thought was a pretty obvious extension to a binary tree. The major difficulty was in handling the case of deletion from the tree in maintaining balance. Fortunately, by looking up in an algorithm book (I think it was Sedgewick [Sedg83]), I found the trick of maintaining balance by flipping heavy/light bits at each node on insertion and deletion. You follow the light branches for insert and the heavy ones for delete. To delete a node interior to the tree, you splice out the interior node and replace it with the heaviest node.

Knight's ideas are not published and his tree insertions are sequential. Nilsson and Stenström [NiSt92] also proposed the maintaining of nonaging binary trees, but these algorithms employ locks that force sequential insertions and deletions.

In August 1991, Ross Johnson proposed a total solution (insertions, deletions, and purges) for cache coherence, using tree merging, an efficient way to maintain binary trees without using locks, without significant aging problems, without complicating the combining mechanism, and without adding much overhead. Johnson's simulations showed that randomness could be used to get good tree-merging performance. At the same meeting Dave James proposed the use of a local purge queue, composed of local cache lines, as an alternative to deadlock avoidance during purge forwarding. Johnson's proposal was enthusiastically received by the working group as the currently best solution. James's enhancement was incorporated a few meetings later. The details of tree merging are given in Chapter 5.

In May 1992, Ross Johnson resigned the chairmanship of the Kiloprocessor Extensions in order to get married and write his PhD dissertation. Stein Gjessing became the new chairman immediately, providing a smooth transition. Ross Johnson provided the first untested C code in June. While writing a paper in October, Johnson discovered that performance could be improved by decoupling tree creation and data distribution. At the time of this writing the working-group document needs to be updated and the C code still needs to be debugged.

### 1.5. Chapter Outline

This section summarizes the contents of later chapters.

SCI specifies low-delay and high-bandwidth networks for shared-memory multiprocessors. The basic building block for communication is a pair of unidirectional, point-to-point links. These always form part of a ring, because any feedback related to sent packets must return later through the incoming link. In some systems, the two links form a trivial ring, connecting a node to a switch interface. In other systems, a node may have to pass along packets it receives that are destined for other nodes. Systems may range from single rings, where all the nodes share the bandwidth of one ring, to interconnected meshes of smaller rings or combinations of rings and switches. It is important for this dissertation that the bandwidth of a network increases with the number of processors. Otherwise, network congestion would negate the benefits of our efficient cache-coherence protocols.

Chapter 2 explores the use of rings (including, but not limited to, SCI rings) to construct several classical topologies and the complications arising from the ring mapping and routing constraints. Mapping is constrained by the closure requirement of a ring and the desire to keep rings small for performance reasons. Routing is complicated by deadlock concerns and limitations of multiple-ring paths. Performance based on message delay and throughput is compared for five topologies. Ultimately, this chapter demonstrates that the use of multiple-ring networks need not limit the scalability of shared-memory multiprocessors, specifically the scalability of their cache-coherence protocols.

SCI's cache-coherence protocol stores its sharing set as a doubly-linked list of cache lines. This is bad for massively-shared data because purges are sequential. Chapter 3 shows how list shortcuts, called *temporary pointers*, are constructed in parallel, using a method called *recursive doubling*, to provide coherence operations with logarithmic latency and constant traffic per node. The choice of which temporary pointers to create is nontrivial; this chapter gives two sets and compares their performance. We show when *request forwarding* can be used without network deadlock, but suggest that its performance improvement does not justify its complexity. We show how *request reserving* can always be used in SCI as an alternative to spin waiting. Finally, this chapter shows how a simple form of request combining can be used to avoid congestion, without the complexity of other combining mechanisms. Chapter 4 extends the results of Chapter 3 by comparing protocol performance against theoretical lower bounds and by exploring the theoretical usefulness of several protocol alternatives.

Chapter 5 introduces a new cache-coherence protocol, called *STEM*, which uses a binary tree to store the sharing set. STEM is the first cache-coherence protocol that helps thousands of processors to efficiently share data, especially massively-shared data. STEM is the first protocol to support fetch-and-add combining without mandating special network support. STEM guarantees forward progress without relying on special network capabilities (such as broadcasts, in-order delivery, or request forwarding), allowing it to be implemented on a wide variety of networks, including SCI. Chapter 6 compares the performance of SCI to the recursive-doubling extensions (Chapter 3) and STEM (Chapter 5). STEM's performance is about the same as the recursive-doubling extensions. STEM has equal or better latency than the SCI protocol and we show the sharing levels when this advantage becomes significant. Finally, STEM has been accepted by the SCI working group as a compatible extension to SCI's coherence protocol for efficiently supporting hundreds/thousands of processors.

Chapter 7 contains conclusions and directions for future work. Appendix A introduces swap-if-zero, a new read-modify-write primitive that is both universal and combinable.

# Chapter 2

# Constructing Topologies with Rings

SCI [Comm91, Gust92a] specifies a topology-independent communications network with the goals of low-delay and high-bandwidth for shared-memory multiprocessors. The basic building block for communication is a pair of unidirectional, point-to-point links. These always form part of a ring, because any feedback related to sent packets must return later through the incoming link. In some systems, the two links form a trivial ring, connecting a node to a switch interface. In other systems, a node may have to pass along packets it receives that are destined for other nodes. Systems may range from single rings, where all the nodes share the bandwidth of one ring, to interconnected meshes of smaller rings or combinations of rings and switches. It is important for this thesis that the bandwidth of the SCI network scales as the number of processors increases. Otherwise, it would be impossible to fully realize the benefits of the extended cache-coherence protocols that are presented in later chapters.

This chapter explores the use of rings (including, but not limited to, SCI rings) to construct several classical topologies and the complications arising from the ring mapping and routing constraints. Mapping is constrained by the closure requirement of a ring and the desire to keep rings small for performance reasons. Routing is complicated by deadlock concerns and limitations of multiple-ring paths. Performance based on message delay and throughput is compared for five topologies. Ultimately, this chapter demonstrates that the use of multiple-ring networks need not limit the scalability of shared-memory multiprocessors, specifically the scalability of their cache-coherence protocols.

## 2.1. Introduction

SCI [Comm91, Gust92a] specifies bus-like functions to a collection of computing nodes. It is intended to scale to thousands of high-bandwidth links, up to 1 gigabyte per second per link, while providing a low-delay, cache-coherent, shared-memory interface. Achieving these objectives precluded the use of a true bus. The basic building block for communication in a ring is a sequence of unidirectional, point-to-point links. The links are truly unidirectional in that *all* signals flow in the same direction, i.e., there are no separate handshaking or flow-control signals.

The basic interface consists of a single input link and a single output link. Packets passing through the node normally experience very low message delay, a few nanoseconds. In the absence of contention, the progress of the packet around the ring is largely limited by speed-of-light transmission delays and the number of intermediate nodes. Routing information is provided in the first few bytes of each packet.

An SCI node usually contains a single interface, zero or more processors, possible memory, and possible connections to other communication media (I/O). The node can send and receive packets. As a receiver, it is capable of recognizing a packet addressed to it and it truncates the packet on the ring, turning it into an echo packet which allows the sender to recognize that the original packet has been received.

A switch is a node containing more than a single interface, with the capability for routing packets from the input of one interface to the output of another. Then, the basic ring topology can be enhanced to include any topology that can be constructed from a set of rings. This chapter explores some candidate topologies (for a large-scale, regular multiprocessor) constructed from a set of SCI rings. While it is not obvious that most of the classical network topologies can be constructed from such rings, we demonstrate that this is possible and explore several well-known topologies that seem promising. The exclusive use of rings presents new constraints that limit the options for some topologies or change the trade-offs. Three constraints are particularly significant.

(1) *Favored switch setting.* A switch can be constructed to have $f$ input links and $f$ output links, but each input link has a unique preferred output link. Because of the ring structure, routing to the corresponding output link will be significantly faster than routing to the others. Thus a topology that minimizes the number of ring changes will provide better performance, other things being equal. Note that this constraint is not unique to rings or SCI implementations, but is rather a characteristic of high-speed switches.

(2) *Different size rings.* Any architectures that can be conceived of as a cylinder, i.e., a "dance-hall" architecture where the two sides are connected, can be constructed out of rings. However, some of the connections may require two iterations or more through the network to arrive back at the starting nodes, while others may not. The optimal size of a ring is strongly affected by the degree of favored switch setting, that is, the relative penalty for taking the unfavored path.

(3) *Deadlock avoidance.* SCI inherently pipelines transfers through a link and routinely permits a packet to emerge from a node before it has been received completely; this is like the wormhole routing of Dally and Seitz [DaSe87]. It also provides sufficient buffering in a node to store an entire packet, like store-and-forward networks (described by Tanenbaum [Tane81]). However, routing in SCI topologies is less restricted than in simple store-and-forward networks because packets are only stored at nodes where packets change rings. This permits solutions for deadlock avoidance that are superior to those appropriate for either wormhole routing or simple store-and-forward networks.

A simple ring exhibits many of the properties of a bus. Like a bus, all processors can quickly witness an event. Goodman and Woest [GoWo88] examined the Wisconsin Multicube, a large-scale, shared-memory multiprocessor architecture that employs a snooping cache protocol over a grid of buses. In a sense this is a continuation of that work, investigating the best topologies for large-scale systems. However, the focus has shifted from buses, which exhibit severe physical and electrical limitations, to rings, which can provide much better performance (both throughput and delay), as shown by Scott et al. [ScGV92].

Traditional metrics of a topology include the average and worst-case distance between nodes, though in practical networks an additional consideration is the bandwidth limitations of the busiest links and other resources. While these metrics have been well studied for the common topologies, the construction through rings adds a new factor to the evaluation: the number of rings traversed. If topology A provides generally shorter paths between nodes than topology B, but requires the traversal of more rings, is it better or worse? For purposes of this chapter, we consider the following metrics: (1) total distance (number of links traversed), (2) number of rings traversed, (3) amount of uniform traffic through the busiest link, and (4) amount of uniform traffic through busiest queue. We then demonstrate how these metrics can be combined to determine the best topology for a given set of nodes, comparing them with constant hardware per processor (including fixed size switches).

The remainder of this chapter is divided as follows. Section 2.2 discusses the constraints on the construction of topologies. Section 2.3 constructs five candidate topologies, plus a common topology that can not be limited to constant hardware per processor. Section 2.4 describes the performance metrics. Section 2.5 derives some of the performance equations. Section 2.6 compares the five candidate topologies with the given metrics. Section 2.7 summarizes this work. Earlier versions of this chapter appear as a technical report [JoGo91a] and as a conference paper [JoGo92].

## 2.2. Topology Construction

In this section we show how various topologies and routing algorithms can be defined for point-to-point networks. Many of the traditional topologies can be constructed with SCI rings and this topology construction requires the definition of ring mapping and routing functions. Also, deadlock avoidance is discussed in the context of routing. In the figures, except where noted otherwise, the nodes are represented as numbered rectangles and the input and output links are represented as labeled lines.

### 2.2.1. Ring Mapping

The goal of ring mapping is to find a one-to-one mapping between a topology's links and the links in a set of rings. We number the nodes of a topology from 0 to $N-1$, where $N$ is the number of nodes. For each node we number the input and output links each from 0 to $f-1$, where $f$ is the node fanout, such that each link has the same number on both ends. Then, ring mapping is formally defined by the function that returns the node connected to a given node via a given output link of the given node.

Figure 2.1 shows one way to map a topology onto rings. The link numbering defines the rings by following the same numbered links around a sequence of nodes. For every node, each link is numbered twice (input and output); distinct letters will be added later for discussion of deadlock. Note that the link numbering does not define uniquely numbered rings. For example,

**Example Torus Topology**

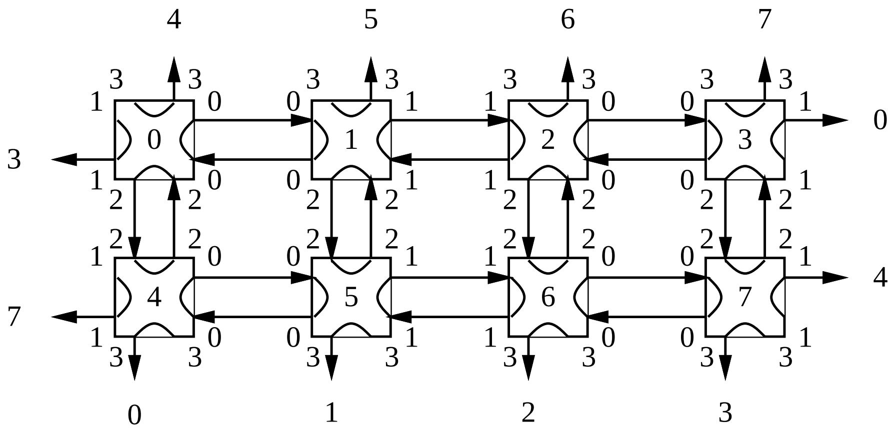



**Fig. 2.1:** In this 2D torus topology, 8 nodes are connected by 16 rings, where each node is connected to 4 rings. Each two-node ring simulates a bidirectional link. Rings are composed of links with the same link number, but not every link with the same number is on the same ring. The twelve numbers around the perimeter are node numbers.

---

link 2 defines four two-node rings. This nonuniqueness is a result of our consecutive link numbering for each node.

The figure indicates the naive way to map network topologies onto rings, by converting every bidirectional link into a two node ring. While entirely general, this method is inferior to other mapping strategies that are able to use the output links of a node to achieve a richer interconnection pattern. When simulating bidirectional links, a node with fanout $f$ ($f$ interfaces) can only reach $O((f-1)^k)$ nodes in $k$ hops while other organizations can reach $\Omega(f^k)$ nodes. For example, no interesting topologies (only doubly-linked lists[11], possibly circular) can be built the naive way with a fanout of two. Connections to more adjacent nodes mean lower diameter graphs, which is important because the bandwidth required of the network is directly proportional to the average path length. A second argument against simulating bidirectional links is that changing rings takes more time than continuing along the same ring. A third argument

[11] This is a line topology, not necessarily a ring and having nothing to do with SCI's cache coherence.

against this is that it is more difficult to avoid deadlock while routing, described later in Section 2.2.3. Therefore, we usually want rings to have more than two nodes.

We also want ring sizes to be small because echo packets must continue the remaining distance around every ring that is visited. (In SCI, echo packets are part of the mechanism to guarantee forward progress and detect packet loss; a new echo packet is created on every ring visited by the send packet.) For each send packet that travels a short distance on the ring, the echo packets waste significant bandwidth. Also arguing for smaller rings, the returning echo packet (acknowledgement) allows release of the buffer resources on that ring, so smaller rings free up the resources faster. Therefore, very large rings are also bad.

This work is *not* an exhaustive study of ring sizes. It is a sample of reasonable ring sizes. As such, this work demonstrates that construction of topologies from rings is possible and reasonable and that communication in large-scale multiprocessors can be supported reasonably by a network of interwoven rings.

### 2.2.2. Routing Functions

Given a mapping function, it remains to define a routing function that avoids deadlock. Routing is formally defined by a function that returns the next link for a packet, given (1) the node that originally generated the packet, (2) the current node through which the packet is traveling, (3) the node that is the target of the packet, and (4) the last link that was used to reach the current node. Note that the routing function defines the next link, not the next node; the mapping and routing functions together define the next node. This means that the two functions collectively provide more information than the traditional nonring routing functions. Our function definition includes the source of the packet, unlike Dally and Seitz [DaSe87], so that it is easier to specify the routing for some topologies, including a general routing function.

A routing constraint imposed by SCI is that a packet does not visit the same ring twice, illustrated by Figure 2.2. It would seem that node 2 could send a packet to node 8 by using the inner ring (entering at 3 and exiting at 6). This generates two echo packets on the outer ring (at 3 and 8) because one echo packet is generated every time a send packet leaves a ring [Comm91]. Unfortunately, there is no way for nodes 2 and 6 to determine which echo to

**Example Clock Topology**

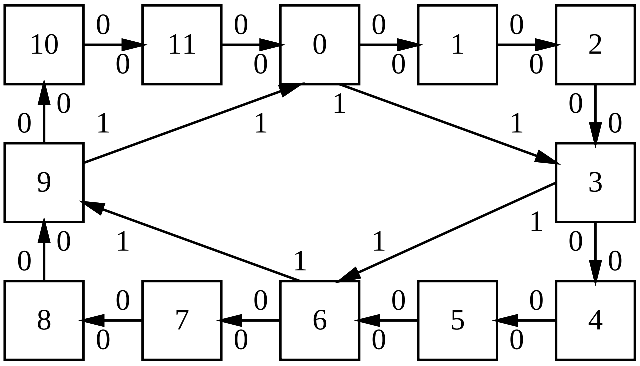



**Fig. 2.2:** In this clock topology, similar to a chordal ring of Feng [Feng81], 12 nodes are connected by 2 rings. This topology illustrates that the same packet can not visit the same ring twice.

---

consume because there is no ordering guaranteed by the ring and the echos can be identical. (An echo contains the original source and final target nodes, but not the intermediate node that generated the echo.) Therefore, the inner ring can not be used in this way.

A general routing function is useful when creating a new topology; the mapping function can be developed using a general routing function before a more concise routing function is developed. It is also useful when a concise function is elusive. Our general router is available via ftp [JoGo91b], uses breadth-first search, and prevents packets from visiting the same ring twice. However, it does *not* avoid deadlock.

### 2.2.3. Deadlock Avoidance

SCI is carefully defined so that forward progress is guaranteed on a single ring, but SCI does not prevent deadlock for multiple rings. Since entire send packets are potentially stored in queues at nodes where they change rings, this is the classical deadlock problem for store-and-forward networks. The standard solution (described by Tanenbaum [Tane81]) to avoid deadlock is to use *queue partitioning*, where the storage is divided into classes and the assignment of space is restricted to provide a partial ordering on the use of the classes. It should be stressed that there is nothing in SCI to prevent packets from being forwarded before they have arrived completely, as is done for the wormhole routing of Dally and Seitz [DaSe87]. However, SCI reserves sufficient space to store entire packets, so deadlock is easier to avoid by viewing SCI as a store-and-forward network.

Partitioning queues to avoid deadlock is costly for SCI because space for an entire packet must be reserved in each partition. Alternately, we can restrict the routing algorithm so that queue partitioning is kept to a minimum. Restricting routing to help avoid deadlock was explored by Dally and Seitz [DaSe87] for wormhole routing, but we explore the concept for store-and-forward networks that are based on rings. If the routing function requires no more than $n$ partitions in any queue to avoid deadlock, then we say that the routing is *deadlock-n*. When routing alone prevents deadlock, the routing is *deadlock*-1, not to be confused with the deadlock-free routing of Dally and Seitz. Deadlock-free routing uses queue partitioning (virtual channels) to avoid deadlock, whereas deadlock-1 routing does not[12].

Figure 2.3 shows a topology and ring mapping for which there exists a simple deadlock-1 routing. The letters relate to deadlock avoidance and will be explained shortly. The link numbers (following the letters) define the rings, as usual. A routing algorithm is straightforward: first send packets on link 1 (if needed) and then on link 0 (if needed). For example, a packet sent from node 3 to 2 goes through nodes 3, 6, 0, 1, and 2 in that order. Then, the packet visits only the queues in nodes 3, 0, and 2 because the packet changes rings at most once. In order to prove that the routing is deadlock-1, we name every input and output queue with a letter. It is easy to show that there are no cycles and no deadlocks because packets are always transferred to a queue with a name that is strictly greater, alphabetically, than the previous queue. In the given example the following queues are visited in this order: queue A in node 3, queues B and C in node 0, and queue D in node 2. Note that this deadlock-1 routing generalizes to arbitrary dimensions and ring sizes by always traversing the highest (or lowest) dimensions first. Also note that we are able to find a deadlock-1 routing in this $r$-ary $f$-cube, unlike Dally and Seitz, because queues are visited only when changing rings.

---

[12]Actually, SCI's network protocol partitions each input and output queue into request and response partitions. Deadlock-1 routing requires no additional partitioning.

**Example Topology with Deadlock-1 Routing**

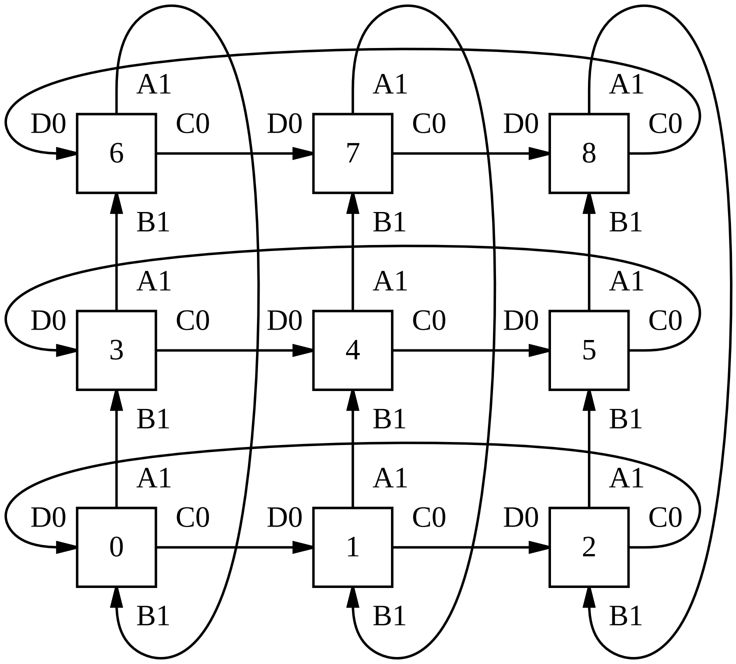



**Fig. 2.3:** In this 2D torus topology, $r^2$ nodes are connected by $2r$ rings, where $r = 3$ for this figure. Each node is connected to 2 rings and each ring has $r$ nodes. The letters show that the routing is deadlock-1, described in the text. Rings are composed of links with the same link number (following the letters). However, not every link with the same number is on the same ring.

---

Of course, not all topologies allow deadlock-1 routings. Figure 2.4 shows a topology with a ring mapping for which there is no deadlock-1 routing function. The problem is that many of the communication routes must use two queues that belong to the center nodes. No matter how the routing is chosen, these routes form queue cycles that can lead to deadlock. For example, trying to simulate a single ring (not using the four internal links) creates a cycle of queues through nodes 3, 4, 8, and 7. Note that the ring cycles have nothing to do with deadlock; the important resources are the queues in the nodes where packets change rings.

## 2.3. Topology Candidates

Given the constraints placed on the mapping and the routing functions, we consider the construction of six (of many possible) topologies: an $r$-ary $f$-cube proposed by Wittie [Witt76], a

**Example Deadlocked Topology**

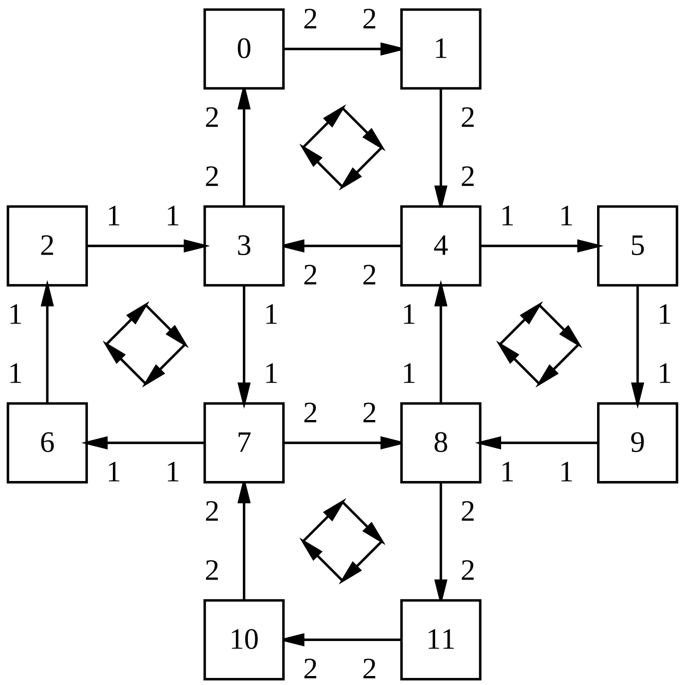



**Fig. 2.4:** This 12-node topology has no deadlock-1 routing. There are 12 nodes and 4 rings. Rings are composed of links with the same link number, but not every link with the same number is on the same ring. That is, the 4 rings of the topology all go clockwise; the center 4 nodes are not on the same ring. Deadlock can occur because there are cycles between the store-and-forward queues, not to be confused with ring cycles.

---

single-stage shuffle-exchange network proposed by Stone [Ston71], three versions of a multistage omega network [Lawr75] reviewed by Feng [Feng81], and the completely connected network reviewed by Feng [Feng81]. We call these six topologies the Multicube [GoWo88], Shuffle, Butterfly, Deadfly, Livefly, and Crossbar respectively. In this section, we discuss the construction of each topology in turn, giving pictorial examples of the mapping functions and outlining the routing functions. In the pictorial examples, the nodes are represented as numbered rectangles and the input and output links are represented as labeled lines. We also address the issue of deadlock avoidance for each topology. Formal C definitions [KeRi88] of all these tested functions, including our general router, are available via ftp [JoGo91b].

### 2.3.1. Multicube Construction

A Multicube is an $r$-ary $f$-cube with $r^f$ nodes, where fanout $f$ is the number of dimensions and $r$ is the size of the rings. We use $r$ and $f$, rather than $n$ and $k$ respectively, to maintain consistency between discussions of different topologies. Although the term Multicube is first used by Goodman and Woest [GoWo88] for grids of buses, we expand its meaning to include $r$-ary $f$-cubes made from point-to-point links. Ring mapping and deadlock-1 routing are explained earlier in conjunction with the Multicube in Figure 2.3. A three-dimensional Multicube is shown in Figure 2.5.

### 2.3.2. Shuffle Construction

The second ring-based topology, called the Shuffle, is the single-stage shuffle-exchange network of Stone [Ston71]. However, the nodes are numbered differently (because we constructed the topology before consulting the literature). The size of the Shuffle is $f^r$, where $f$ is the fanout and $r+1$ is the maximum ring size. One example of link numbering (the mapping function) is shown in Figure 2.6. We are not able to derive a concise routing function and, therefore, we use the general routing function described above. This means that we do not have a deadlock-1 routing for the Shuffle and so the Shuffle must use queue partitioning (or another mechanism) to avoid deadlock. It is likely that we would have found a concise routing function if we had used the node numbering as defined by Stone. However, we do not believe that a deadlock-1 routing exists for the Shuffle due to its asymmetry and multiple ring sizes.

### 2.3.3. Butterfly Construction

Consider the next topology. When designing a multiprocessor with the multistage omega network, there are $\log_f(N)$ switching elements per processor. This violates our constraint of constant hardware per processor and we propose two way to fix this. The first way is to replace each processor with $\log_f(N)$ processors so that there is one switching element per processor; we call this the Butterfly. This solution creates two types of nodes, one with zero processors and one with clusters of processors. Note that clustering the processors does not exactly meet our hardware constraint because there still needs to be some multiplexing at the clusters that is not needed for other topologies. However, we will still analyze this topology in spite of the

**Example Multicube Topology**

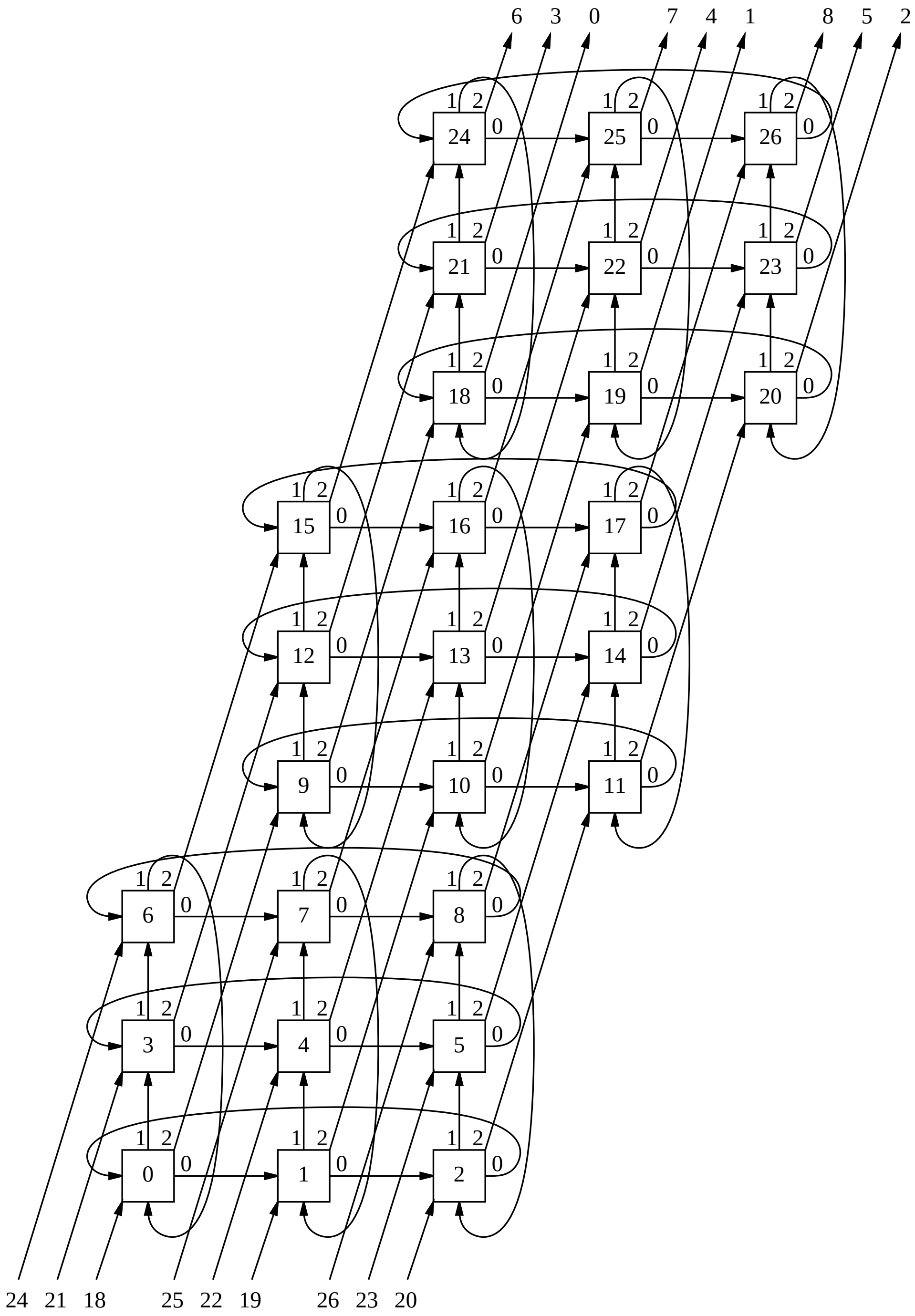



**Fig. 2.5:** The 3D torus with $r^3$ nodes is shown for $r = 3$. It is constructed using $3\,r^2$ rings of size $r$. Each node is on 3 rings, shown by output links with the same link number. However, not every output link with the same number is on the same ring. The numbering of input links is similar, but not shown.

## Example Shuffle Topology

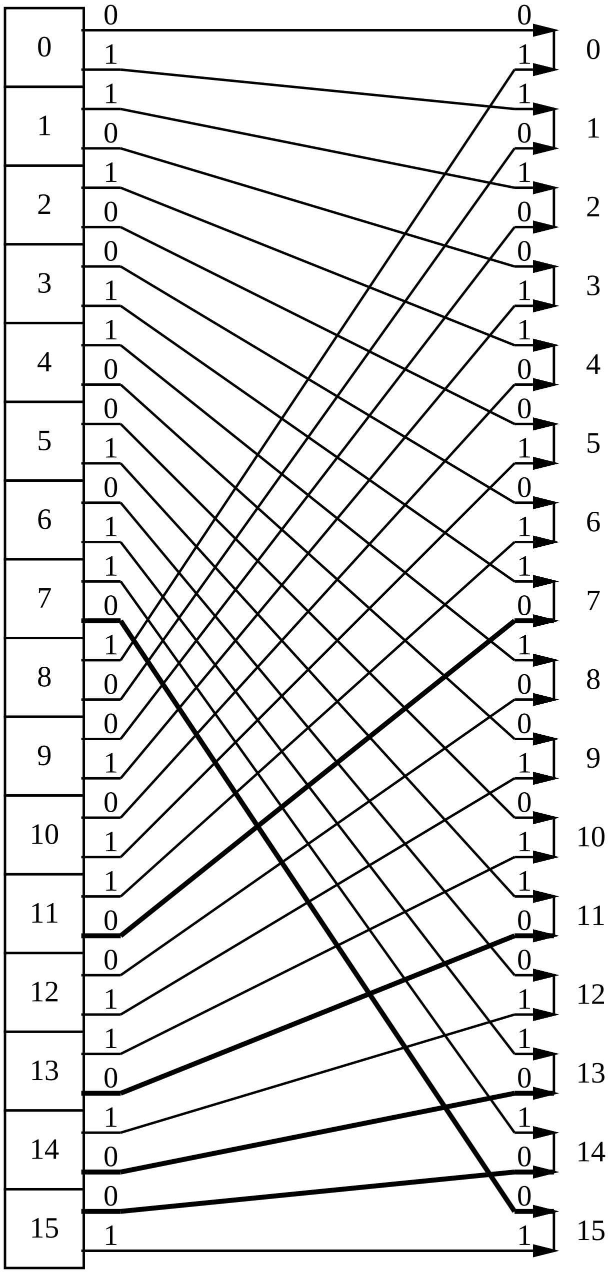



**Fig. 2.6:** An example of the single-stage shuffle topology is shown with $f^r$ nodes; $f = 2$ and $r = 4$. Each node is connected to $f$ rings and the rings have maximum size of $r + 1$. Numbers on the right are node numbers. Nodes 0 and 15 each has one link to itself, making a 1-node ring. One of the longest rings is highlighted with bold lines, going through nodes 15, 14, 13, 11, 7, and back to node 15.

unfairness. The second way to satisfy the constant-hardware constraint is to embed the processors in the network, but we will discuss this later.

The size of the Butterfly is $r \cdot f^r$, where $f$ is the fanout and $r$ is the size of the shortest rings (the cylinder size or the number of network stages). Constructing the ring-based Butterfly topology is not as simple as it might first appear. There are two significant problems to solve.

The first problem is to determine the mapping function. Note that the links must be numbered so that (a) no two input links of the same node are numbered the same, (b) no two output links of the same node are numbered the same, and (c) the size of all rings is minimized. Our goal is to have ring sizes that are no larger than two times the number of stages in the network because we can show this is a lower bound for one-way round-trip traffic around the cylinder. The key to proving the lower bound is that each network stage changes one bit of the node number and only that same stage can change it back.

The mapping function is illustrated by the link numbering in Figure 2.7. Some rings cycle once, straight through on one row around the cylinder. Other rings always jump rows and cycle twice before completing. The definition of the mapping function is relatively simple for binary fanout. But, the general case is complex, relative to the Multicube, because it requires divide and modulus operations to mask parts of the node number.

The second problem is to determine a deadlock-1 routing such that no packet visits the same ring twice. It is important that we have picked the correct mapping function, otherwise this routing might be impossible. Our routing for the Butterfly modifies a traditional routing, where the next node is found by changing part of the current node number to match the target node number (masked substitution). This routing is modified to determine the next link rather than the next node. The proof-sketch that this routing is deadlock-1 is that there exists a partial ordering of the queues (no cycles of packet buffers) such that packets visit queues in strictly increasing order. One partial ordering corresponds to the stages of the network, shown by the letters in the figure.

**Example Butterfly Topology**

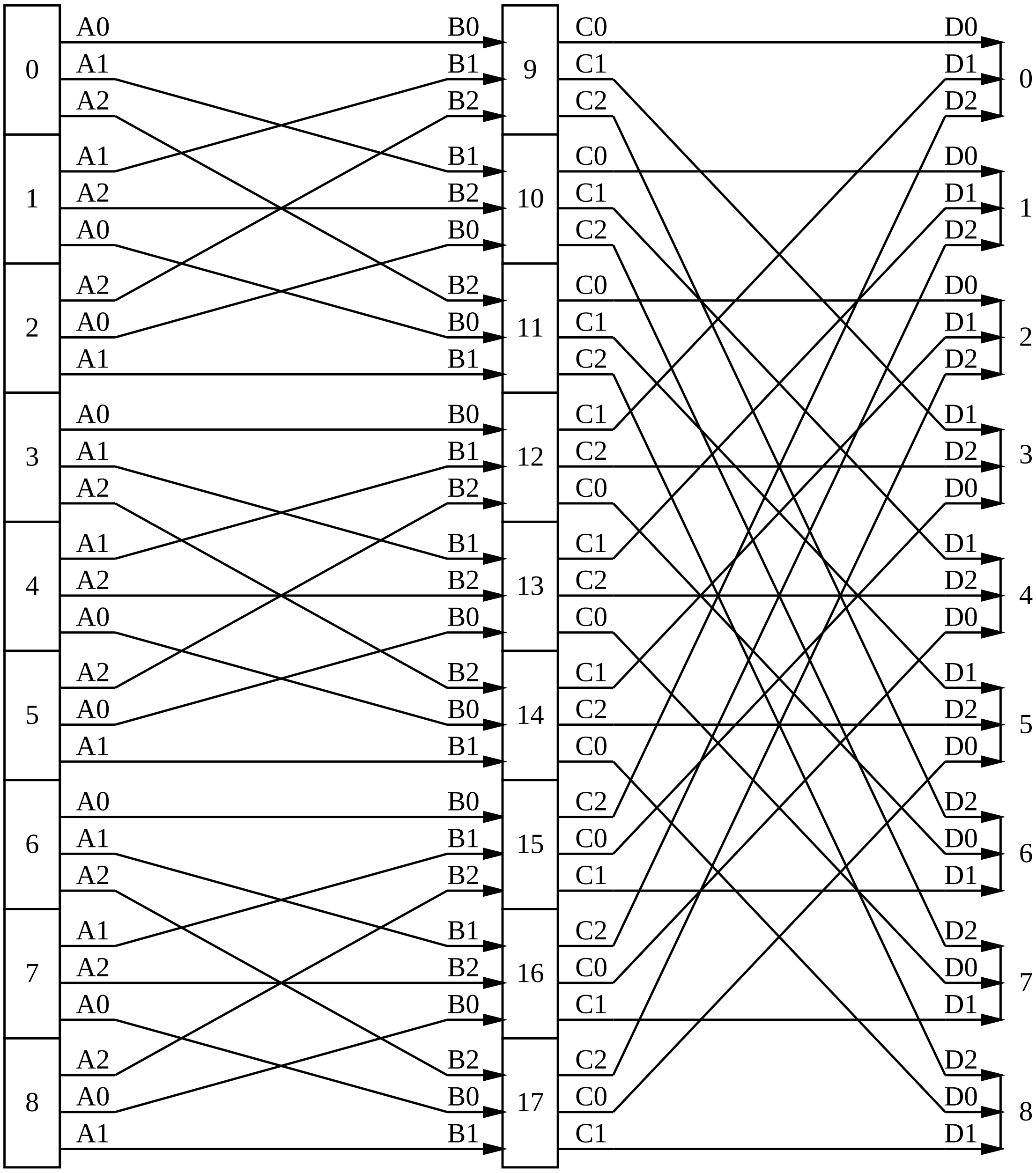



**Fig. 2.7:** This is the Butterfly with $r \cdot f^r$ nodes, where $r = 2$ and $f = 3$. Each node is connected to $f$ rings and each of the $\frac{f+1}{2} f^r$ rings is connected to either $r$ or $2r$ nodes. The short rings follow the straight links once around the cylinder, but the other links make rings after two cycles. Numbers on the right are node numbers.

### 2.3.4. Deadfly Construction

The Butterfly makes the processor-switch ratio equal to 1 by replacing each processor by $\log_f(N)$ processors. Another way to make this ratio equal to 1 is to place a processor at every switch in the topology. However, when we embed processors in the stages and use the same routing, we get the possibility of deadlock. This is because packets can start at an arbitrary stage and end at an arbitrary stage; we can prove that at least one cycle of queues exists for at least one size of this topology. For this reason we call this topology the Deadfly and, like the Shuffle, the Deadfly must use queue partitioning (or another mechanism) to avoid deadlock. Note, we have not proven that no deadlock-1 routing exists; we merely state that the given routing is not deadlock-1. However, we *believe* that no deadlock-1 routing exists.

### 2.3.5. Livefly Construction

In order to find a deadlock-1 routing for the Deadfly, we modify the topology so that there are two short-ring links instead of one, shown in Figure 2.8. The link numbering illustrates the mapping function and we call this the Livefly. Deadlock-1 routing can then be accomplished in three phases. First, the packet is sent to the first stage of the network via one set of the short-ring links; call them the secondary links. Second, the packet is routed through the topology (around the cylinder) as normal, using the other short-ring links as needed; call them the primary links. Third, the packet is delivered to the correct network stage via the secondary links. Any of these three phases is skipped when the phase would make no progress towards delivery of the packet.

The letters in the figure are an example of the proof-sketch that the routing for Livefly is deadlock-1. In particular, each input and output queue is given a name so that the given routing moves packets between queues of alphabetically increasing names. (Recall that queues are visited only when a packet changes rings.) Notice that the input queues of nodes 0 through 7 are labeled alphabetically greater than the output queues and also notice that the input and output queues on the secondary links are labeled differently. These differences result from the three-phase routing. Finally, note that this topology has a performance disadvantage because one of the links is dedicated to solving the deadlock problem. Essentially, two short-ring links are used instead of partitioning the one short-ring queue into two parts. Incidentally, a corollary to this

**Example Livefly Topology**

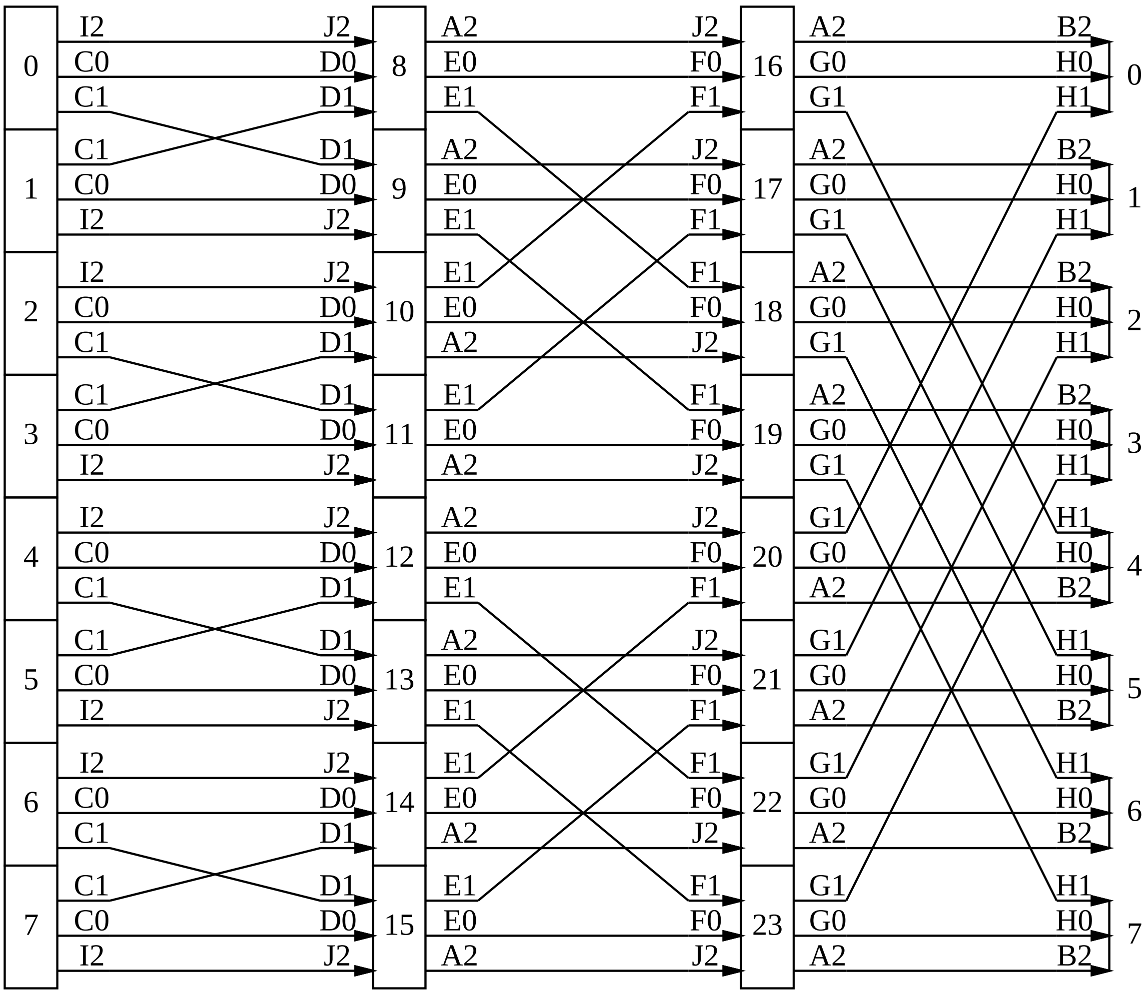



**Fig. 2.8:** Here we show the Livefly topology with 2 short-ring links per node and $r\cdot(f-1)^r$ nodes, where $f=3$ and $r=3$ for this picture. Each node is connected to $f$ rings. There are a total of $\frac{f+2}{2}(f-1)^r$ rings of size $r$ or $2r$. The letters demonstrate that the routing is deadlock-1, discussed in the text. The short rings follow the straight links once around the cylinder, but the other links make rings after two cycles. Numbers on the right are node numbers.

---

proof-sketch shows that there exists a deadlock-2 routing for the Deadfly, but we chose not to explore deadlock-2 routings.

### 2.3.6. Crossbar Construction

Figure 2.9 shows a ring mapping for a Crossbar with four nodes. Each ring has one or two nodes and each node is on four rings. The dotted lines show connections that are not needed since they form rings that have only one node. Nodes are drawn as being attached to $f$ rings, rather than $f-1$ rings. This is because it is impossible, in general, to number the links from 0 to $f-2$ such that 1) no node has two input links and no two output links of the same number and 2) an input and an output link of the same number on the same node form a ring with another node. For example, it is not possible to use only two links for $f=3$ (not shown in the figure). Shortest-path routing is deadlock-1 because a packet never visits an input queue before an output queue. Of course, the completely connected topology is not a good topology for large systems because the hardware (specifically links) per node grows in proportion to the number of nodes. Since the constant-hardware assumption would necessitate the multiplexing of $\Theta(N)$ processors in at least one node, the Crossbar is not a candidate topology for performance analysis in later sections.

## 2.4. Metrics and Methods

Now we consider the metrics for evaluating the performance of the five candidate topologies in the context of SCI. It is important to know how a topology supports efficient communication for both light and heavy loads. For light loads we compare the worst-case delay to deliver one packet, assuming no congestion. For heavy loads we compare the worst-case bandwidth requirements to deliver packets between every pair of nodes.

For light loads we are mainly interested in the maximum number of links visited by one packet. We call this the topology *distance*. The maximum number of visited rings, called the *ring hops*, is also important for two reasons. First, moving packets between nodes on different rings takes longer than moving packets between nodes on the same ring. Second, some topologies may use queue partitioning, which requires queue resources to hold at least ring-hops packets.

Given the distance and the ring hops, we can combine these two metrics into one metric of delay under light load. Preliminary studies suggest that under a variety of lightly loaded conditions, the time to pass through a node and change rings is approximately four times the time to

**Example Crossbar Topology**

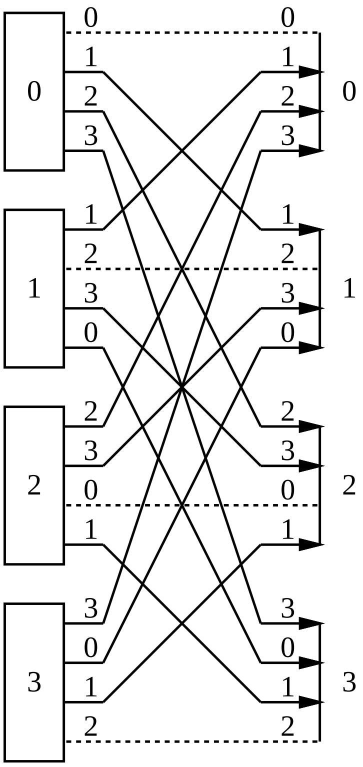


**Fig. 2.9:** In the Crossbar, $f$ nodes are connected by $f\,(f+1)\,/\,2$ rings, where $f = 4$ for this figure. Dotted lines indicate unneeded links. Ignoring the one unneeded link per node, each node is connected to $f - 1$ rings of size 2.

---

simply pass through a node. Assuming this, delay is the weighted sum of the distance plus three times the ring hops. We multiply this by 10ns, the time for an SCI symbol (or flit) to pass through a node for a prototype ECL (Emitter-coupled logic) implementation [Kris91]. Note that delay under light load is not affected by SCI packet lengths because SCI packets contain all the required routing information in the first few bytes of each packet [Comm91].

For heavy loads we consider the maximum bandwidth required per resource to deliver average-size packets between every pair of nodes in the topology. We do not consider waits or retries after collisions. We merely determine the total bandwidth used per resource to complete all packet deliveries, giving an upper bound on the capacity of each topology under uniform load. This will show the topology bottlenecks. We determine the bandwidth used for each link and each queue to deliver the $\Theta(N^2)$ packets. We call these traffic metrics *hot link* and *hot queue* respectively. The maximum rate at which processors can send and receive packets, as limited by the hot link for uniform traffic, is called the *throughput*. For both the hot link and throughput we use the SCI echo size of 10 bytes, including one prepended idle symbol [Comm91], and estimate that the average send-packet size is 50 bytes. Throughput is inversely proportional to the hot-link traffic.

Figure 2.10 summarizes the metrics used. We used both simulation and analytical methods to determine the values of the various metrics. Simulation involves tracing every packet through the topology and counting the resources used. For the lightly loaded scenario we can use analytical methods because the math is tractable. For the heavily loaded scenario we also use analytical methods (checked by simulation for the Multicube and Deadfly), but only for symmetric topologies. We are unable to find analytical solutions for heavily-loaded asymmetric topologies, so we rely entirely on deterministic simulations for these.

### 2.5. Equation Derivations

In this section we present the derivations of the equations for message delay and throughput, including the metrics of distance, ring hops, hot link, and hot queue, all of which are summarized in Figure 2.10. These equations are derived for the five candidate topologies, discussed in Section 2.3. The delay equations and their components are summarized later in Figure 2.11 and the components of the throughput equations are summarized near the end of this section in Figure 2.12. Except for the Butterfly, the throughput equations are checked by simulation. The terms *short ring* and *long ring* refer to rings of size $r$ (cylinder circumference) and $2r$ respectively in the Butterfly, Deadfly, and Livefly topologies. The nonmathematical reader should skip to Section 2.6.

#### 2.5.1. Delay Analysis

Consider the Multicube with $N = r^f$ nodes, where $r$ is the ring size and $f$ is the fanout at each node. What is the distance of the topology? In the worst case a packet must visit all nodes in a ring (all links minus one) before going to the ring in the next dimension. Since it must do this in all $f$ dimensions, the distance is $(r-1)f$ and the ring hops is $f$. Assuming a ring change penalty

**Comparison Metrics**

| | |
|---|---|
| One packet between *one* pair of nodes: | |
| *Distance* | maximum visited links |
| *Ring Hops* | maximum visited rings |
| *Delay* | weighted sum of above |
| One packet between *every* pair of nodes: | |
| *Hot-Link Traffic* | bandwidth used per link |
| *Hot-Queue Traffic* | packet visits per queue |
| *Throughput* | peek processor send rate for uniform traffic |

**Fig. 2.10:** This is a listing of the comparison metrics.

of $c$ (the time to pass through a node and change rings divided by the time to pass through a node without changing rings), the delay is then $(r-1)\,f+(c-1)\,f$ or $f{\cdot}r+(c-2)\,f$.

Consider the Shuffle with $N = f^r$ nodes, where $r+1$ is the maximum ring size and $f$ is the fanout at each node. The worst case is illustrated by a packet sent between nodes 0 and $N-1$, so the distance and the ring hops are both $\log_f(N)$ or $r$. This communication path is the worst case because no other path is longer and a packet on this path changes rings at every node. Assuming a ring change penalty of $c$, the delay is then $r+(c-1)\,r$ or $c{\cdot}r$.

Consider the Butterfly with $N = r{\cdot}f^r$ nodes, where $r$ is the size of the short rings and $f$ is the fanout at each node. Recall that $r$ processors are clustered at each node in the first network stage. Notice that a packet must travel either zero or one time around the Butterfly to get to the correct node, so the distance (maximum visited links) is $r$. The packet can change rings at each hop, so the ring hops is also $r$. Assuming a ring change penalty of $c$, the delay is then $r+(c-1)\,r$ or $c{\cdot}r$.

Consider the Deadfly with $N = r{\cdot}f^r$ nodes, where $r$ is the size of the short rings and $f$ is the fanout at each node. Recall that processors are embedded in the topology. In the worst case a send packet must travel once around the Deadfly to get on the correct row and then almost another time around to get to the target in the correct network stage. So, the distance is $2\,r-1$. The packet can change rings at each of the $r$ hops for the first time around the Deadfly, but stays on the same ring after entering on the correct row. So, the ring hops is $r+1$. Assuming a ring change penalty of $c$, the delay is then $(2\,r-1)+(c-1)\,(r+1)$ or $(c+1)\,r+(c-2)$.

Consider the Livefly with $N = r \cdot (f-1)^r$ nodes, where $r$ is the size of the short rings and $f$ is the fanout at each node. Recall that processors are embedded in the topology. In the worst case a send packet must travel almost once around the Livefly to get to the first network stage, once around the Livefly to get on the correct row, and then almost another time around to get to the target in the correct network stage. So, the distance is $3r-2$. The packet can change rings at each of the $r$ hops for the second time around the Livefly, but stays on the same ring after entering on the correct row for the first and third phases. So, the ring hops is $r+2$. Assuming a ring change penalty of $c$, the delay is then $(3r-2)+(c-1)(r+2)$ or $(c+2)r+(2c-4)$.

The delays for the five topologies are summarized in Figure 2.11. Note that the ring sizes for these topologies are not exactly the same for identical values of $N$ and $f$, illustrated by the different subscripts. In particular, $r_s = \log_f(N)$ for Shuffles, $r_m = N^{1/f}$ for Multicubes (which is $r_m \le \log_f(N)$ when $r \le f$), approximately $r_b \approx \log_f(N) - \log_f(\log_f(N)) + \dfrac{\ln(\log_f(N))}{\log_f(N)}$ for the Butterfly and Deadfly, and approximately $r_l \approx \log_{f-1}(N) - \log_{f-1}(\log_{f-1}(N)) + \dfrac{\ln(\log_{f-1}(N))}{\log_{f-1}(N)}$ for the Livefly, where $\ln(x)$ is the natural log.

From the equations in Figure 2.11 we can estimate the delay in nanoseconds by multiplying the given delay by 10ns, the time for an SCI symbol (or flit) to pass through a node for a prototype ECL implementation [Kris91]. We also substitute $c=4$, which is suggested by preliminary studies.

### 2.5.2. Throughput Analysis

Next, we consider the scenario of packets being sent between every pair of nodes. We derive the equations for the hot link and the hot queue in the Multicube, Deadfly, and Butterfly. The equations for the Multicube and Deadfly are checked by simulation. Since the equations for Shuffle do not seem tractable, we get results by simulation alone. Also, we use simulation for the Livefly because we do not derive those equations either.

**Topology Delay**

| Topology | Ring Hops *rings* | Distance *links* | Delay $(c-1)\cdot rings + links$ |
|---|---|---|---|
| Multicube | $f$ | $f\cdot r_m - f$ | $f\cdot r_m + (c-2)\,f$ |
| Shuffle | $r_s$ | $r_s$ | $c\cdot r_s$ |
| Butterfly | $r_b$ | $r_b$ | $c\cdot r_b$ |
| Deadfly | $r_b + 1$ | $2\,r_b - 1$ | $(c+1)\,r_b + (c-2)$ |
| Livefly | $r_l + 2$ | $3\,r_l - 2$ | $(c+2)\,r_l + (2\,c-4)$ |

**Fig. 2.11:** Here we summarize the delay and its components, where $f$ is the fanout and $r$ is related to the ring size, described in the text. Subscripted variables are used for $r$ to indicate that $r$ is not necessarily the same for different topologies with the same fanout $f$ and total number of nodes. The hop penalty, $c$, is the ratio of delays through a node, depending on whether or not the packet changes rings. See Figure 2.10 for other definitions.

---

#### 2.5.2.1. Multicube Throughput

Consider the Multicube with $N = r^f$ nodes, where $r$ is the ring size and $f$ is the fanout at each node. What is the hot-link traffic, the most bandwidth required of any one link? Since the Multicube is symmetric, we can add up the traffic over all links and divide by the total number of links. There are $N^2$ packets to deliver, counting the zero-bandwidth packets that have the same source and target. On average, each send packet traverses $\frac{r-1}{2}$ links (range between 0 and $r-1$) in each of $f$ dimensions, giving a total traffic of $N^2\,\frac{r-1}{2}\,f$ send packets. Since there are $f\cdot N$ total links, the individual link traffic for send packets is $\frac{r-1}{2}\,N$. When echo packets are added, the entire rings are traversed (collectively by send and echo packets), using $r$ links in each dimension traversed. However, not every dimension is always used. The average links used in each of $f$ dimensions is $r-1$ because one out of every $r$ packets does not need to be routed in the given dimension. This yields a total traffic of $N^2\cdot(r-1)\,f$ packets (both send and echo) and an individual link traffic of $(r-1)\,N$ packets. Normalizing to the size of the average send packet, the general formula for the hot-link traffic is then

$$(1 + e_s)\,\frac{r-1}{2}\,N, \tag{2.1}$$

where $e_s$ is the ratio of the average send packet to the size of an echo packet.

To find the hot-queue metric we first determine the average number of rings visited by each packet. Since 1 out of every $r$ packets (on average) is not routed in a given dimension and since there are $f$ dimensions, the average number of rings visited by each packet is $\frac{r-1}{r}\,f$. Multiplying by the number of messages, $N^2$, and dividing by the number of nodes, $N$, gives the hot-queue metric of

$$\frac{r-1}{r}\,f \cdot N. \tag{2.2}$$

#### 2.5.2.2. Deadfly Throughput

Consider the Deadfly with $N = r \cdot f^r$ nodes, where $r$ is the size of the short rings and $f$ is the fanout at each node. What is the hot-link traffic? We will determine this in four cases: long-ring links for $e_s = 0$ (send-packet traffic), long-ring links for $e_s = 1$ (total traffic, ignoring packet size), short-ring links for $e_s = 0$ (send-packet traffic), and short-ring links for $e_s = 1$ (total traffic, ignoring packet size). In the derivations of the four cases we make use of the fact that the topology is symmetric. In particular, each node has 1 link that belongs to a ring of size $r$ and $f - 1$ links that belong to rings of size $2\,r$. After deriving equations for the four cases, we combine them into a long-ring case and a short-ring case for arbitrary $e_s$. Then, the general formula for the hot-link traffic is the maximum of these two combined cases. After finishing with the hot-link derivations for the Deadfly, we derive the hot-queue equations for the Deadfly.

First, find the hot-link traffic in long-ring links for $e_s = 0$. Note that the long-ring links are only used until the packet is lined up on the correct row of the cylinder, after which only the short-ring links are used, if any. Therefore, we can consider each of the $N$ nodes to be the root of a balanced $f$-ary tree for sending $N$ packets, the targets of which are evenly distributed among the leaves. Each of the $N^2$ packets traverses $r$ of the $f{\cdot}N$ links. So, each long-ring link supports transmission of $\frac{N^2\,r}{f{\cdot}N}$ or

$$(r{\cdot}N)\,\frac{1}{f} \tag{2.3}$$

send packets.

Second, find the hot-link traffic in long-ring links for $e_s = 1$. Consider the average number of long rings visited by one packet. The first traversed link has probability $\frac{f-1}{f}$ that it is a long-ring link. Each subsequent link has conditional probability of $\frac{1}{f}\,\frac{f-1}{f} + \frac{f-1}{f}\,\frac{f-2}{f}$ or $\frac{(f-1)^2}{f^2}$ that it is a new long-ring link (depending on the previous link type) and this occurs $r-1$ times. Adding these together gives the average number of long rings touched by a packet,

$$\frac{f-1}{f} + (r-1)\frac{(f-1)^2}{f^2}. \tag{2.4}$$

Multiplying this sum by the total number of packets, $N^2$, dividing by the total number of long-ring links, $(f-1)\,N$, and multiplying by the size of a long ring[13], $2\,r$, gives the total number of packets traversing any one long-ring link, $\left(\frac{f-1}{f} + (r-1)\frac{(f-1)^2}{f^2}\right)\frac{N^2\,2\,r}{(f-1)\,N}$ or

$$(r{\cdot}N)\left(\frac{2}{f} + 2\,(r-1)\,\frac{f-1}{f^2}\right). \tag{2.5}$$

To determine the hot-link traffic in short-ring links (the third and fourth cases), we must recall that packets are delivered in two rounds of the topology: the first round lines up the packet on the correct row (short ring) by hopping from ring to ring; the second finishes the delivery using only the given short ring. Of course, some packets find their target before using both rounds. To determine the hot-link traffic in short-ring links, we first find the traffic during each

[13]Recall that if a send packet enters a ring, then all links in the ring are touched once by a packet, either by the given send packet or its echo packet.

round, making a simplifying assumption about when packets find their targets. Second, we subtract the error due to this simplifying assumption.

For the third case, find the hot-link traffic in short-ring links for $e_s = 0$. For simplicity let us assume that no packet reaches its target before $r$ hops (the source stage in the network). Under this simplifying assumption, each short-ring link supports the same amount of traffic during the first round as do the long-ring links, namely

$$(r \cdot N)\,\frac{1}{f} \tag{2.6}$$

send packets from Equation 2.3. In the second round the distances traveled by a packet are evenly distributed between 0 and $r-1$, so the total traffic in the second round is $N^2 \sum_{k=0}^{r-1} \frac{k}{r}$ or $N^2\,\frac{r-1}{2}$. Dividing this by $N$, the number of short-ring links (because no long-ring links are used in the second round), gives $N\,\frac{r-1}{2}$ or

$$(r \cdot N)\,(r-1)\,\frac{1}{2\,r} \tag{2.7}$$

hot-link traffic in short-ring links for $e_s = 0$ and the simplifying assumption.

Now, consider the error induced by the simplifying assumption in the third case. The number of erroneous packets is the number of targets that can be reached in distance $r-1$ or less (including when the target is the same as the source). If $k$ is the network-stage number, then the number of erroneous packets per node is $\sum_{k=0}^{r-1} f^k$ or $f^r \left( \frac{1}{f-1} - \frac{1}{(f-1)\,f^r} \right)$ or $N \left( \frac{1}{(f-1)\,r} - \frac{1}{(f-1)\,N} \right)$, because $N = r\,f^r$. Note that for each erroneous packet the distance error is exactly one full short ring. Multiplying the number of erroneous packets per node by the error per packet, $-r$, and the number of sources, $N$, and then dividing by the total number of short-ring links, *N*, gives the error term

$$-r\,N\left[\frac{1}{(f-1)\,r}-\frac{1}{(f-1)\,N}\right]. \tag{2.8}$$

So, the hot-link traffic in short-ring links for $e_s = 0$ is the sum of Equations 2.6, 2.7, and 2.8 or $(r \cdot N)\left[\frac{1}{f}+(r-1)\,\frac{1}{2\,r}-\frac{1}{(f-1)\,r}+\frac{1}{(f-1)\,N}\right]$ or

$$\frac{r \cdot N}{f-1}\left[\frac{f+1}{2}-\frac{1}{f}-\frac{f+1}{2\,r}+\frac{1}{N}\right]. \tag{2.9}$$

For the fourth case, find the hot-link traffic in short-ring links for $e_s = 1$. For simplicity let us assume that all packets finally reach their targets via a short-ring link. Consider the average number of short rings visited by one packet, using this simplifying assumption. The first traversed link has probability $\frac{1}{f}$ that it is a short-ring link. Each subsequent link has conditional probability of $\frac{1}{f}\,0+\frac{f-1}{f}\,\frac{1}{f}$ or $\frac{f-1}{f^2}$ that it is a new short-ring link (depending on the previous link type) and this occurs $r-1$ times. Adding these together gives the average number of short rings touched by a packet, $\frac{1}{f}+(r-1)\,\frac{f-1}{f^2}$. Multiplying this sum by the total number of packets, $N^2$, dividing by the total number of short-ring links, *N*, and multiplying by the size of a short ring, *r*, gives the total number of packets traversing any one short-ring link in the first round, namely

$$(r \cdot N)\left[\frac{1}{f}+(r-1)\,\frac{f-1}{f^2}\right]. \tag{2.10}$$

For the second round the probability is $\frac{f-1}{f}$ that a packet arrives on a long-ring link, after which it starts (touches) a new short ring. So, using the same analysis, the second-round traffic contribution per link is $\frac{f-1}{f}\frac{N^2\,r}{N}$ or

$$(r\cdot N)\,\frac{f-1}{f}. \tag{2.11}$$

Now, consider the error induced by the simplifying assumption in the fourth case. The number of erroneous packets is the number of packets with the same source and target plus the number of packets that reached their target nodes via a long-ring link (in the first round). The number of erroneous packets per node is therefore $1+\sum_{k=1}^{r} f^k\,\frac{f-1}{f}$, where $f^k$ is the number of nodes that are reachable in exactly $k$ hops and $\frac{f-1}{f}$ is the probability that the packet arrived via a long-ring link. This reduces to $f^r$ or $\frac{N}{r}$ erroneous packets per node, because $N = r\,f^r$. Note that for each erroneous packet the distance error is exactly one full short ring. To get the error term we multiply the number of erroneous packets per node by the number of sources, $N$, divide by the number of short-ring links, $N$, and multiply by the distance error, $-r$. This is $\frac{-N}{r}\frac{N\,r}{N}$ or

$$-(r\cdot N)\,\frac{1}{r}. \tag{2.12}$$

So, the hot-link traffic on the short-ring links for $e_s = 1$ is the sum of Equations 2.10, 2.11, and 2.12 or

$$(r\cdot N)\left(\frac{1}{f}+(r-1)\,\frac{f-1}{f^2}+\frac{f-1}{f}-\frac{1}{r}\right) \tag{2.13}$$

or

$$\frac{r\cdot N}{f-1}\left(\frac{2\,r+2\,f-4}{2}-\frac{2\,r-2}{f}-\frac{2\,f-2}{2\,r}+\frac{r-1}{f^2}\right). \tag{2.14}$$

Now that we have the four cases derived, we combine the cases of similar links. This is done by noting that the echo-packet traffic is the total traffic minus the send-packet traffic. Then, the echo-packet traffic can be weighted by $e_s$, the ratio of sizes, and added to the send-packet traffic. Therefore, from Equations 2.3 and 2.5, the hot-link metric for the long-ring links of the Deadfly is

$$(r \cdot N)\left[\frac{1}{f} + e_s\left[\frac{1}{f} + 2\,(r-1)\,\frac{f-1}{f^2}\right]\right] \tag{2.15}$$

and, from Equations 2.9 and 2.14, the hot-link metric for the short-ring links of the Deadfly is

$$\frac{r \cdot N}{f-1}\left\{\begin{array}{l} \dfrac{f+1}{2} - \dfrac{1}{f} - \dfrac{f+1}{2\,r} + \dfrac{1}{N} \\ + e_s\left[\dfrac{2\,r+f-5}{2} - \dfrac{2\,r-3}{f} - \dfrac{f-3}{2\,r} + \dfrac{r-1}{f^2} - \dfrac{1}{N}\right] \end{array}\right\}. \tag{2.16}$$

Finally, we find the hot-queue metric, which can be rephrased (since the Deadfly is symmetric) as the maximum number of times the $N^2$ packets transfer rings at any one node. To get the per-node number of transfers to long rings, we take the hot-link traffic in long rings for $e_s = 1$, given in Equation 2.5, divide by the long-ring size, $2\,r$, and multiply by the number of long-ring links per node, $f-1$, yielding $r \cdot N\left[\frac{2}{f} + 2\,(r-1)\,\frac{f-1}{f^2}\right]\frac{f-1}{2\,r}$ or

$$N\left[r - \frac{2\,r-1}{f} + \frac{r-1}{f^2}\right]. \tag{2.17}$$

To get the per-node number of transfers to short rings, we take the hot-link traffic in short rings for $e_s = 1$, given in Equation 2.13, divide by the short-ring size, $r$, and multiply by the number of short-ring links per node, 1, yielding $r \cdot N\left[\frac{1}{f} + (r-1)\,\frac{f-1}{f^2} + \frac{f-1}{f} - \frac{1}{r}\right]\frac{1}{r}$ or

$$N\left[1 + \frac{r-1}{f} - \frac{r-1}{f^2} - \frac{1}{r}\right]. \tag{2.18}$$

Adding together the long-ring transfers, Equation 2.17, and the short-ring transfers, Equation 2.18, gives the total number of ring transfers per node, namely

$$N\left[1 + r - \frac{r}{f} - \frac{1}{r}\right]. \tag{2.19}$$

#### 2.5.2.3. Butterfly Throughput

Now consider the hot link and hot queue of the Butterfly. Since exactly one full round is needed for the Butterfly, the short-ring links support almost the same number of send packets as do the long-ring links; the short-ring links actually support less because packets with the same source and target never need to be sent through the network. However, the short-ring links must support fewer echo packets since the short-ring size is smaller. Therefore, the hot links in the Butterfly are the long-ring links and this hot-link metric is already derived for the Deadfly, given in Equation 2.15.

Now consider the hot-queue traffic. From Equation 2.4 we know that each packet visits an average of $\frac{f-1}{f} + (r-1)\,\frac{(f-1)^2}{f^2}$ long-ring links. So, each packet visits an average of $\frac{f}{f-1}\left(\frac{f-1}{f} + (r-1)\,\frac{(f-1)^2}{f^2}\right)$ rings. Multiplying this by the number of packets, $N^2$, dividing by the number of nodes, $N$, and simplifying, gives the hot-queue metric

$$N\left(1 + (r-1)\,\frac{f-1}{f}\right). \tag{2.20}$$

#### 2.5.2.4. Throughput Summary

Figure 2.12 summarizes the equations for the hot-link and hot-queue metrics. The hot-link metric for the Deadfly is the maximum of the metrics for the long-ring links and the short-ring links. Also, recall that the values of $r$ are not exactly the same between topologies. In particular, $r_s = \log_f(N)$ for Shuffles, $r_m = N^{1/f}$ for Multicubes (which is $r_m \le \log_f(N)$ when $r \le f$), and approximately $r_b \approx \log_f(N) - \log_f(\log_f(N)) + \frac{\ln(\log_f(N))}{\log_f(N)}$ for the Butterfly and Deadfly, where $\ln(x)$ is the natural log.

In order to use the equations in the figure, we need to know $e_s$, the ratio between the sizes of send packets and echo packets. For the common cache-coherence transactions in SCI, half of the send packets are 16-byte requests and the others are 80-byte data packets [Comm91]. Since each packet is preceded by a 2-byte idle symbol [Comm91], the expected (common-case) send

**Topology Hot Links and Hot Queues**

| Topology | Hot Link<br>*send packets* $+ e_s \cdot$*echo packets* | Hot Queue<br>*send packets* |
|---|---|---|
| Multicube | $(1+e_s)\,\frac{r_m-1}{2}\,N$ | $\frac{r_m-1}{r_m}\,f \cdot N$ |
| Shuffle | simulation | simulation |
| Butterfly | $\frac{r_b \cdot N}{f}\left\{1+e_s\left(1+2\,(r_b-1)\,\frac{f-1}{f}\right)\right.$ | $N\left(1+(r_b-1)\,\frac{f-1}{f}\right)$ |
| Deadfly (long ring) | $\frac{r_b \cdot N}{f}\left\{1+e_s\left(1+2\,(r_b-1)\,\frac{f-1}{f}\right)\right.$ | $N\left(1+r_b-\frac{r_b}{f}-\frac{1}{r_b}\right)$ |
| Deadfly (short ring) | $\frac{r_b \cdot N}{f-1}\left\{\frac{f+1}{2}-\frac{1}{f}-\frac{f+1}{2\,r_b}+\frac{1}{N}\quad +e_s \cdot\right.$<br>$\left(\frac{2\,r_b+f-5}{2}-\frac{2\,r_b-3}{f}-\frac{f-3}{2\,r_b}+\frac{r_b-1}{f^2}-\frac{1}{N}\right)$ | |
| Livefly | simulation | simulation |

**Fig. 2.12:** Here we summarize the hot link and hot queue traffic, where $N$ is the topology size, $f$ is the fanout, and $r$ is the ring size. Subscripted variables are used for the ring sizes to indicate that $r$ is not necessarily the same for different topologies with the same $f$ and $N$. The ratio between the sizes of send packets and echo packets is $e_s$. See Figure 2.10 for other definitions. Unclosed parentheses extend throughout the remainder of an equation. The ‘‘$\cdot$’’ in ‘‘$e_s\cdot$’’ means that $e_s$ is multiplied by the quantity on the next line. Except for the Butterfly, these equations are checked by simulation.

---

packet consumes 50 bytes of bandwidth. Echo packets are 8 bytes plus the 2-byte idle, so the expected ratio is close to $e_s = 5$. This is also an upper bound for all packets related to cache-coherent data. A lower bound is $e_s = 1.8$, when all send packets are 16-byte request (or response) packets. To determine the bytes on the hot link we substitute $e_s = 5$ and multiply the formula by the average send-packet size of 50.

Furthermore, we can use these equations to determine the maximum issue rate, in bytes per second per processor, by taking the $N-1$ packets issued per processor, dividing by the hot-link traffic given in Figure 2.12, and multiplying by the SCI bandwidth of 1 gigabyte per second per link. Since each packet in the network is preceded by a 2-byte idle symbol, we also multiply by the ratio of processor bandwidth to network bandwidth, namely $48/50$. The result is maximum throughput in gigabytes per second per processor, as limited by the hot link for uniform traffic.

## 2.6. Discussion

In this section we present the performance of the five selected topologies: Multicube, Shuffle, Butterfly, Deadfly, and Livefly. Recall that we assumed constant hardware per processor, but have violated this assumption for two cases. First, the Shuffle and Deadfly require queue partitioning (or another mechanism) to guarantee forward progress. Queue partitioning requires additional information (a hop count) per packet and more control complexity per node. Second, the Butterfly must multiplex $\log_f(N)$ processors at each processor cluster, adding complexity. In spite of these violations, we first compare these topologies in terms of message delay (under light load), and second in terms of throughput (under heavy load), recalling the violations as appropriate.

For all graphs shown, the keys order the topologies with respect to system sizes of 1024. Some graphs have missing data points for large sizes of the shuffle because the simulations required too much time to complete. The graphs do not show the Livefly for fanout 2 because it is no better than a single ring.

### 2.6.1. Delay Graphs

We first compare the delay of the topologies under light load. One aspect of delay is ring hops, characterized by Figure 2.13. The ring hops for the Multicube is constant, making the shape of the curve significantly different than the others that have logarithmic shapes. The other aspect of delay is the distance, characterized by Figure 2.14. Except for small fanouts, both metrics agree on the ordering of the topologies.

Delay is shown in Figure 2.15. Clearly, Butterfly has the lowest delay for all configurations, but it must multiplex many processors per node. Furthermore, its routing

**Ring Hops**

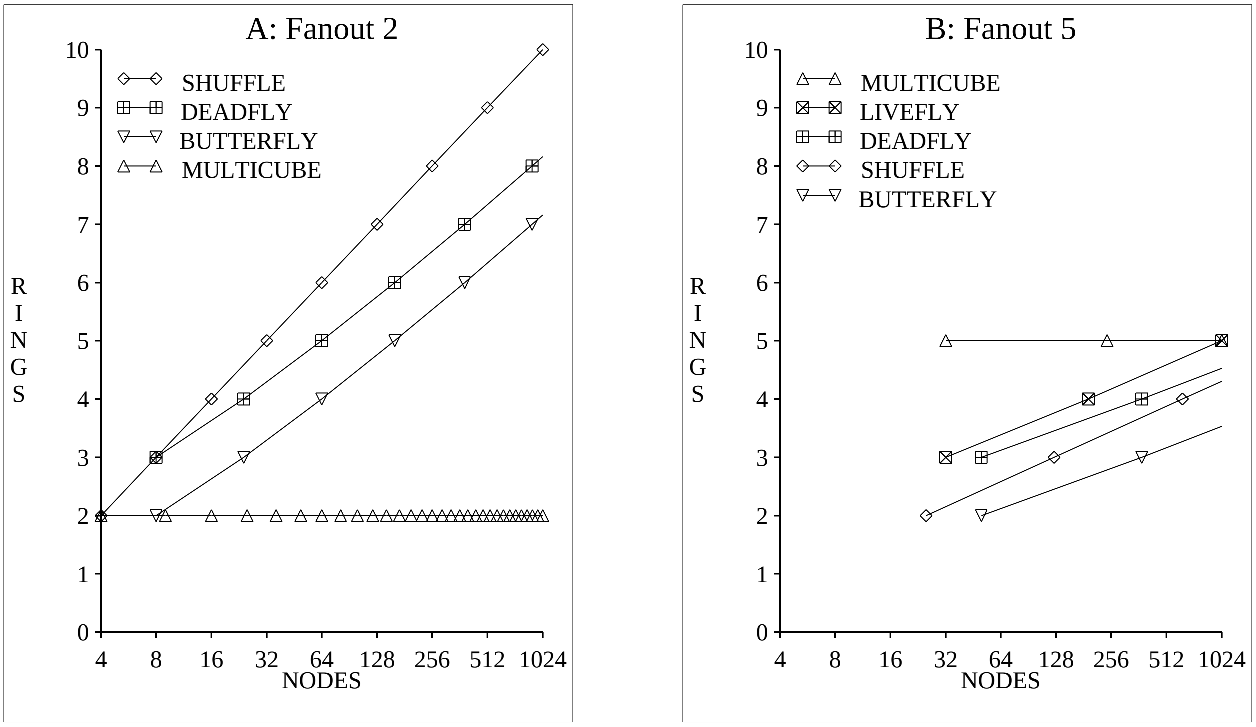



**Fig. 2.13:** These semi-log graphs characterize the maximum number of *rings* visited by any one packet as the fanout is varied from 2 to 8. Livefly, Deadfly, and Butterfly are always ordered as given for fanout 5. Multicube crosses the others at fanouts 3 and 4 and is lowest for fanout 2. Shuffle crosses Deadfly at fanout 3 (at about 80 nodes).

---

function is more complex than Multicube's routing function if the fanout is not a power of two [JoGo91b]. For low fanout and small systems, Multicube is competitive because it has fewer ring hops. The graphs also show that Multicube size is more flexible, especially for small fanout. Multicube size becomes even more flexible when the topology is not limited to a single ring size, not shown here.

One difference between the Multicube and the others is most accented when the fanout is 2, where the distance grows as square root for Multicube, rather than as log for the others. This is because the Multicube's dimensionality (fanout) is held constant, rather than the radix (ring size), due to the constant-hardware assumption. The crossover point in Figure 2.15D is explained by the same crossover point for ring hops with fanout 4; as the hop penalty increases, the ring-hops metric dominates.

**Distance**

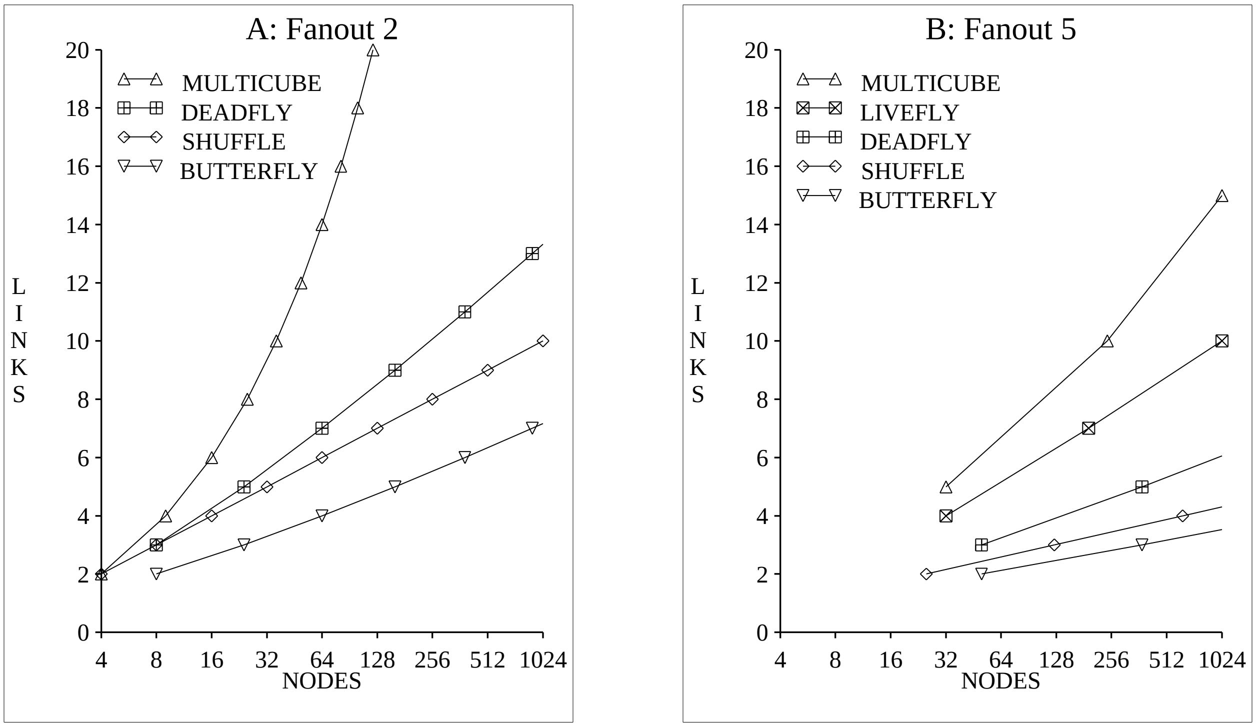



**Fig. 2.14:** These semi-log graphs characterize the maximum number of *links* visited by any one packet as the fanout is varied from 2 to 8. The topologies are always ordered as given for fanout 5, except that Multicube is lower than Livefly for fanout 3 and up to 128 nodes.

---

As the topology size increases, so does the physical size of the multiprocessor. Bianchini [Bian89] and Dally [Dall90] have considered the effects of three-space on the performance of topologies, but this is beyond our scope. Essentially, the physical layout of the topology becomes increasingly important for increasing system size, affecting the accuracy of our results on delay under light load. On the other hand, three-space does not need to affect throughput because signals can be pipelined on the wire, as shown by Scott and Goodman [ScGo91, Scot92]. SCI exploits this pipelining.

### 2.6.2. Throughput Graphs

One indicator of throughput is the hot link, shown in Figure 2.16. Generally, Butterfly is best by this metric. It is interesting to note that the Shuffle is better than Multicube, even though it is not

**Delay**

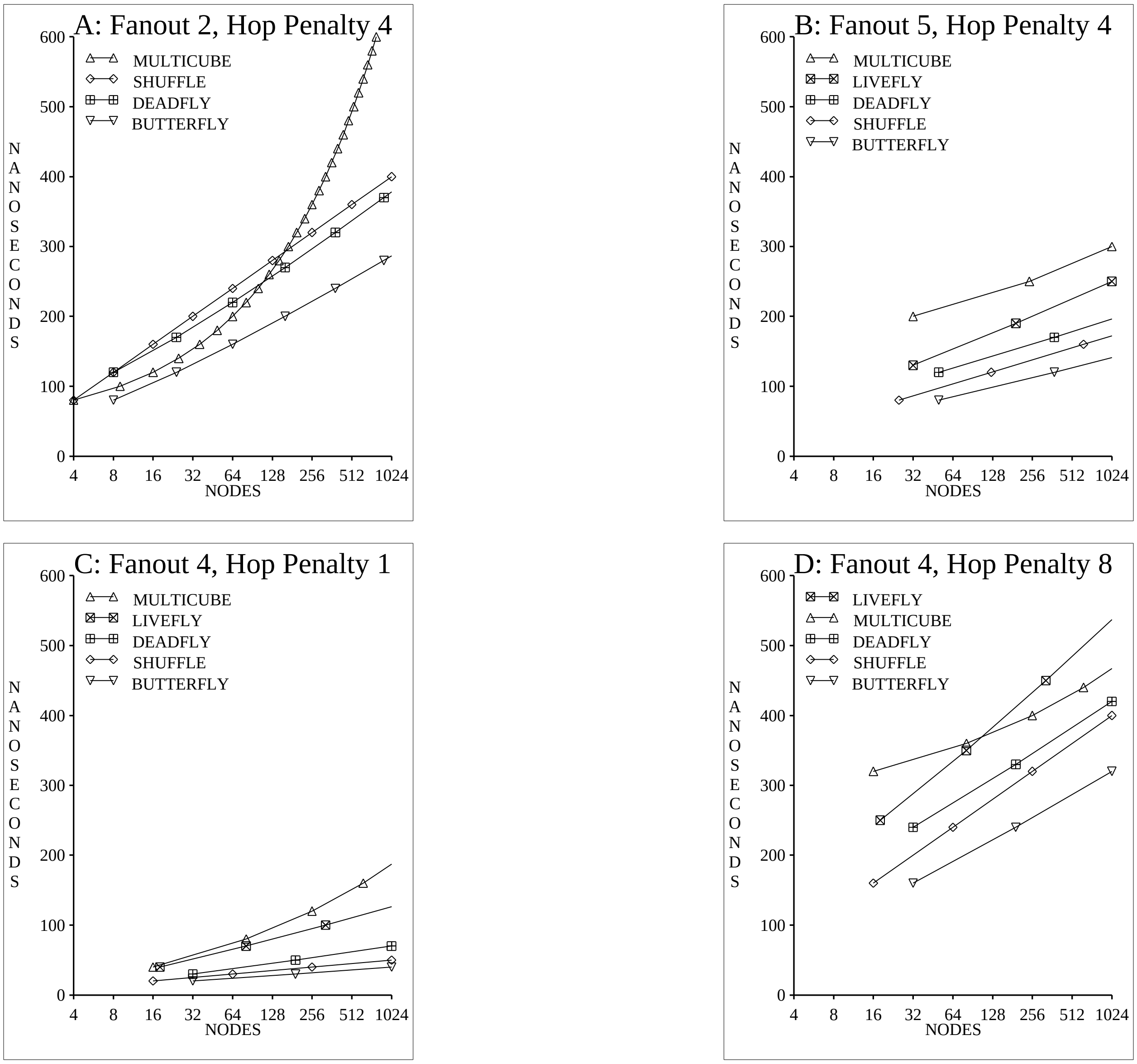



**Fig. 2.15:** These semi-log graphs characterize the worst-case delay of packet delivery under light loading. The hop penalty is the the ratio of delays through a node, depending on whether or not the packet changes rings (changing rings takes longer).

As the fanout varies from 2 to 8 with fixed hop penalty of 4 (the top two graphs), the topologies usually remain ordered as shown for fanout 5. Fanout 2 is an exception, as shown, and Multicube is lower than Livefly for fanout 3. As the hop penalty varies from 1 to 16 with fixed fanout of 4 (the bottom two graphs), the topologies are ordered as shown. The crossover point shown for hop penalty 8 begins at hop penalty 5 and stays fixed around 100 nodes as the hop penalty increases.

**Hot Link**

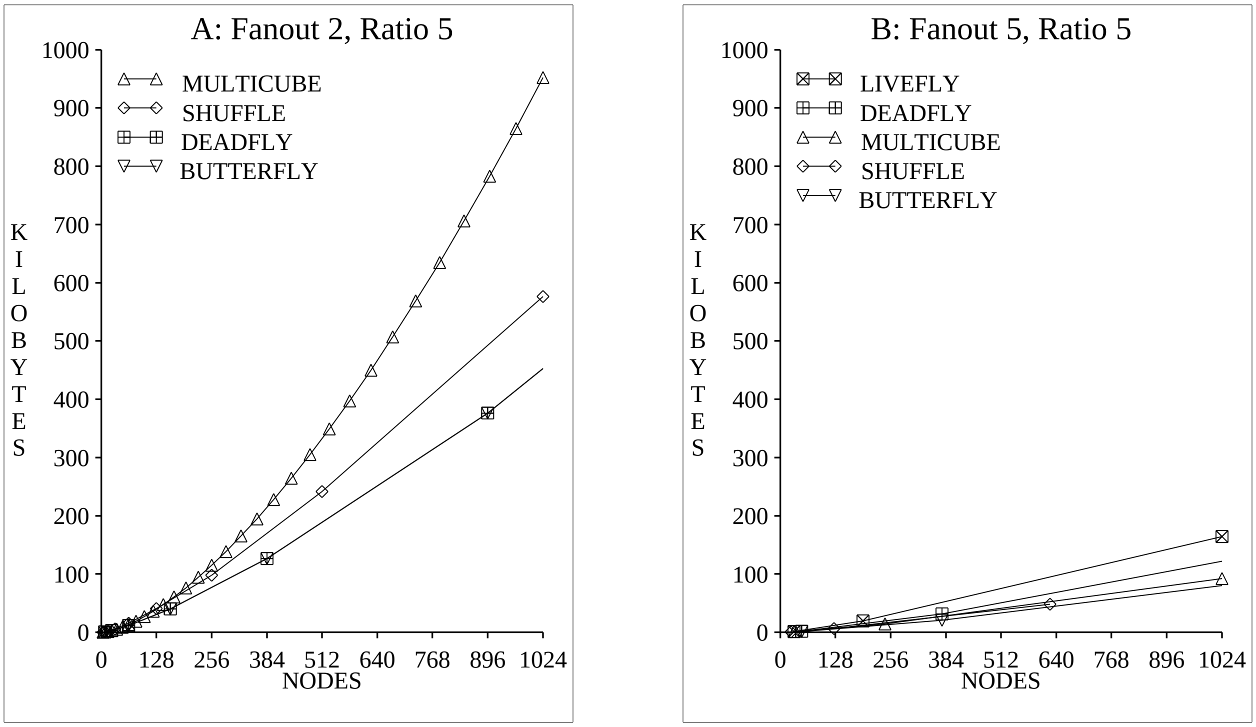



**Fig. 2.16:** These graphs characterize the maximum number of *bytes* passing through any one *link* when all pairs of nodes communicate once. The ratio of sizes between send and echo packets is called *ratio*. Generally, for fanouts of 2 to 8, the topologies are ordered as shown in the legend for fanout 5. But for fanouts of 2 to 4, Multicube is higher than Deadfly. At fanout 2, Deadfly and Butterfly are equal. Shuffle is always slightly lower than Multicube (higher than Deadfly for fanouts of 2 and 3) and it is about equal with Butterfly for fanouts 6 to 8. Recall that the keys order the topologies with respect to system sizes of 1024, so Butterfly is the lowest for fanout 5.

---

completely symmetric. The reason for this is that the Shuffle's asymmetry can be better characterized by its few cold links rather than its many near-average hot links.

Figure 2.17 shows the hot-queue traffic. The hotter the queue, the more likely that the finite queue will fill up. When the queue fills up, the tree saturation of Pfister and Norton [PfNo85] can occur, causing large losses in performance. In other words, the hot queue approximates the topology's tolerance to fluctuations in traffic flow (temporary hot spots). Since the Shuffle and Deadfly require queue partitioning (or another mechanism) to avoid deadlock, the graph lines for these topologies reflect a multiplier corresponding to the number of required partitions. This

**Hot Queue**

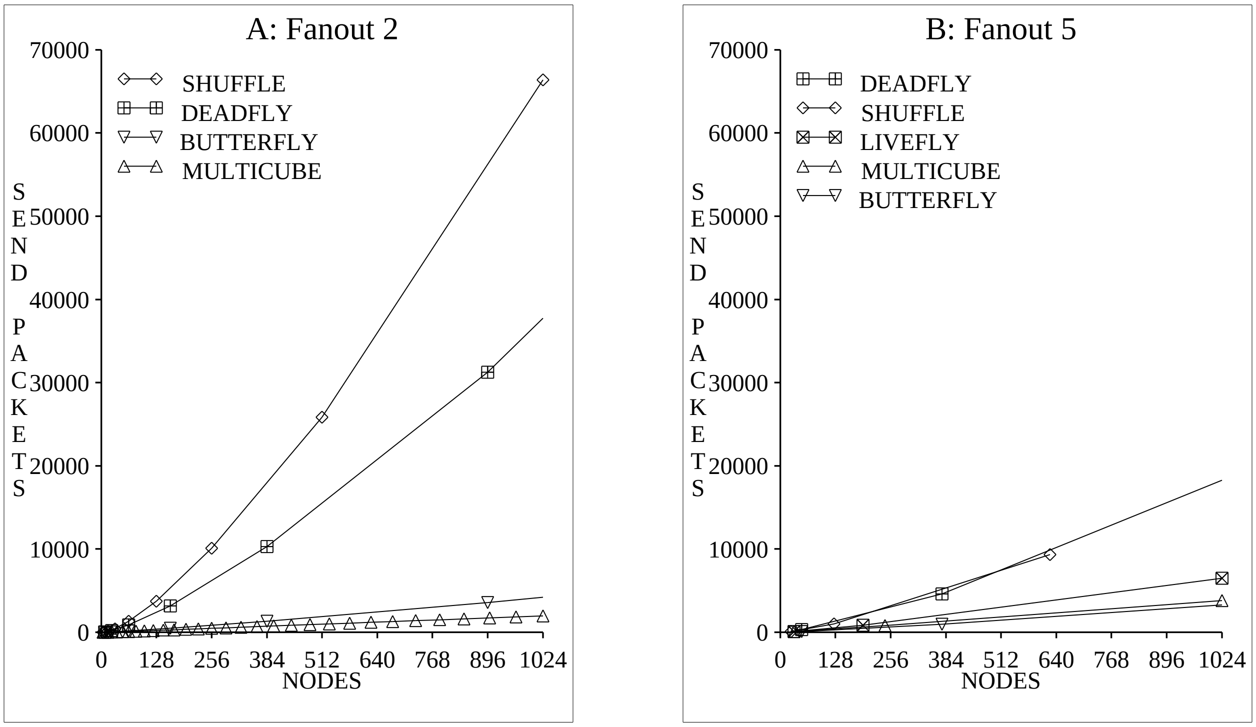



**Fig. 2.17:** These graphs characterize the maximum number of *packets* passing through any one *queue* when all pairs of nodes communicate once. Generally, for fanouts of 2 to 8, the topologies are ordered as shown by the legend of fanout 5. However, Multicube is the lowest for fanouts 2 and 3 and Livefly is lower than Multicube at fanout 8. Shuffle is the highest for fanouts 2 to 4 and slightly lower than Deadfly for fanout 8.

---

makes them significantly worse, especially for low fanouts. For the hot queue overall, the Butterfly is best, except for fanouts 2 and 3, when the Multicube is best.

The graphs of throughput, as limited by the hot link, are shown in Figure 2.18. As predicted by the hot-link graph, the Butterfly is best. However, Figure 2.18A points out that Multicube is best for small fanouts and very small system sizes. Figures 2.18C and 2.18D also show that relatively small send packets are favorable for Multicube. This is because packets in Multicube visit fewer rings, causing fewer echo packets to be generated.

**Throughput**

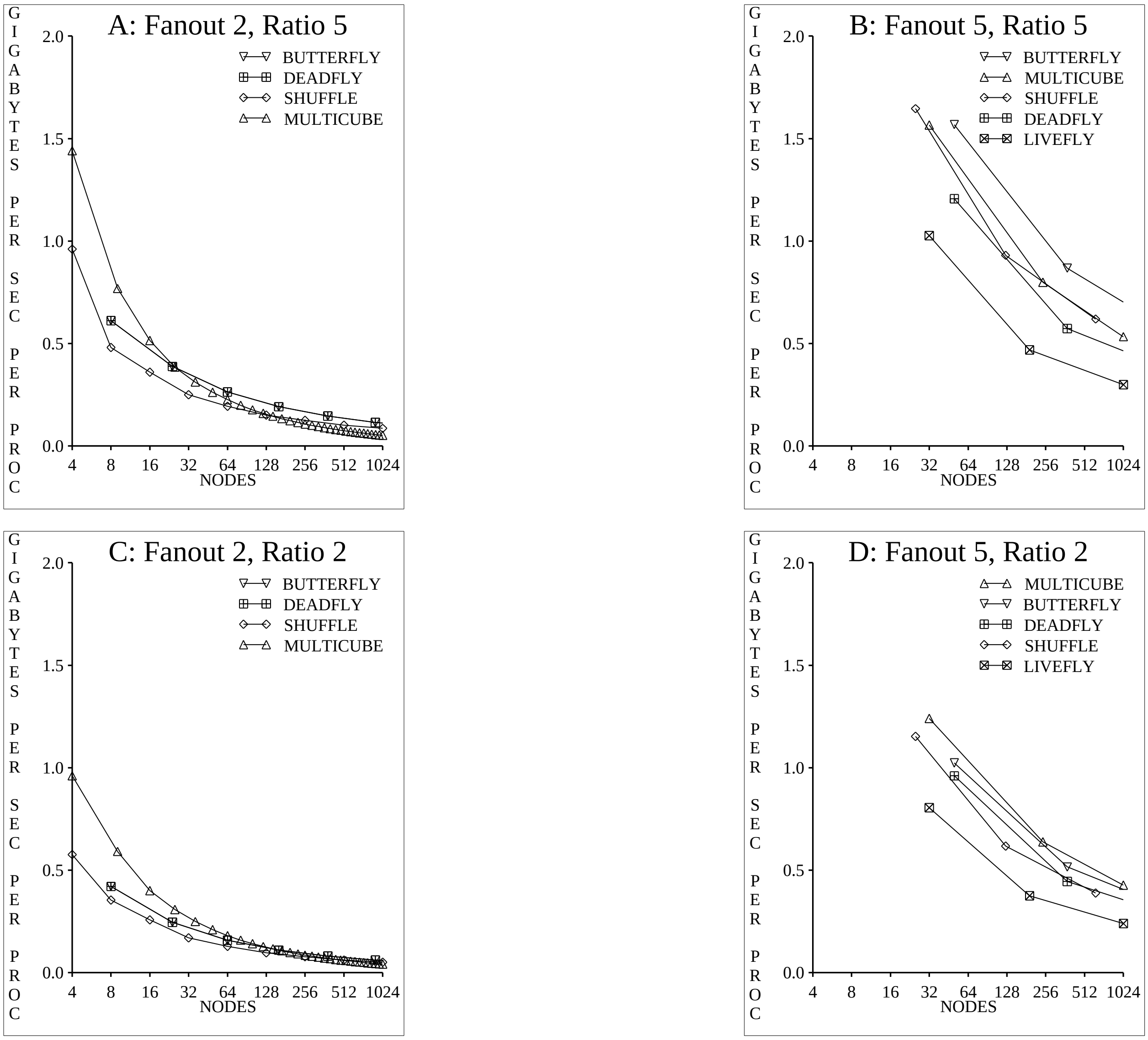



**Fig. 2.18:** These semi-log graphs characterize the maximum throughput (gigabytes per second per processor) as limited by the hot link. The ratio of sizes between send and echo packets is called *ratio*. Generally, for fanouts of 2 to 8 and a ratio of 5 (the top two graphs), the topology ordering is as shown for fanout 5. However, Deadfly is higher than Multicube for fanouts 2 and 3 and larger system sizes; they are about equal for fanout 4. Shuffle crosses Deadfly at fanout 4.

Generally, for fanouts of 2 to 8 and a ratio of 2 (the bottom two graphs), the topology ordering is as shown for fanout 5. However, Multicube is the lowest for fanouts 2 and 3 for more than about 100 and 500 nodes respectively. Also, the Shuffle is higher than Deadfly for fanouts 6 to 8, but the Shuffle is the lowest for fanout 2 and up to 128 nodes.

## 2.7. Summary

SCI [Comm91, Gust92a] specifies a topology-independent communications network with the goals of low-delay and high-bandwidth for shared-memory multiprocessors. The basic building block for communication is a pair of unidirectional, point-to-point links that may be connected in a sequence to form rings. Many classical topologies can be constructed with rings, but rings add additional constraints that complicate the construction. This construction includes both the ring mapping and routing functions. Ultimately, this chapter demonstrates that the use of multiple-ring networks need not limit the scalability of shared-memory multiprocessors, specifically the scalability of their cache-coherence protocols.

In this chapter we discussed the construction constraints and compared five topologies that can be constructed with rings. Ring mapping is not trivial because rings must be closed, the ring size must be kept small, and the mapping must lead to a good routing function. Routing is complicated by the fact that no ring may be visited twice and deadlock between finite queues must be avoided. Also, the required routing functions are not trivial modifications of standard routing functions. This is because the ring mapping and routing functions collectively provide more information than the standard topologies and associated nonring routing functions. Concerning deadlock, we showed how some store-and-forward topologies can be constructed without deadlock and without partitioning the queues; this routing is called deadlock-1 routing. Deadlock-1 routing is important because queue partitioning complicates the implementation and makes queues more vulnerable to tree saturation.

After the topologies were constructed, they were compared with constant hardware per processor. Four metrics were used: ring hops, distance, hot-link traffic, and hot-queue traffic. The first two were converted into a combined metric of delay and the second two were discussed in terms of throughput and temporary hot-spot tolerance. Where possible, analytical solutions and simulation data validated each other.

In summary of the graphs, we have compared Multicube, Shuffle, Butterfly, Deadfly, and Livefly. In general, Butterfly performed better by both metrics of delay and throughput. However, to implement the multiplexing of the processor clusters, more hardware is necessary. Furthermore, it is not clear how the layout in three-space will affect the wire lengths and, subsequently, the delay for large systems. As an alternative, Multicube is competitive for small system sizes and small fanouts, especially for relatively small send packets and large penalties for changing rings. Also, Multicube has more choices of system size for a given fanout.

# Chapter 3

# Recursive Doubling

## 3.1. Introduction

SCI [Comm91, Gust92a] includes a cache-coherence protocol for shared-memory multiprocessors. The cache-coherence protocol is a distributed-pointer scheme [TDLS90] where the directory[14] maintains a pointer to the head of a doubly-linked list of cache lines. Due to the sequential nature of these *sharing lists*, SCI is not scalable for applications that have a high degree of sharing (long lists) and frequent writes to these lines. Both reads and writes are inefficient for high degrees of sharing, but this is important only when the writes are frequent. Therefore, extensions to SCI's cache-coherence protocol are needed to provide for scalability (logarithmic time) when the degree of sharing is high and writes to these lines are frequent.

In September 1989 [Gust91], the SCI working group first decided to address the sequential problem of lists. In October 1989, Guri Sohi proposed a combining mechanism [JLGS90] for creating the sharing list, thereby avoiding sequential access to the directory. This mechanism, called *request combining,* reduces the effects of hot spots and tree saturation [PfNo85] when the same line is requested almost simultaneously by many caches. Simultaneous requests for the same instructions or data are likely to happen when all processors enter a new phase of a program after a barrier. At the same time Sohi presented the use of *recursive doubling* [Lebe91, LeSo92], an asynchronous version of the synchronous recursive doubling presented by Hillis and Steele [HiSt86] and earlier by Stone [Ston73], for simultaneously distributing data to $N$

---

[14]Recall that we define the *directory* to be at memory and use *sharing set* to refer to the set of caches with copies.

requesting caches. This recursive doubling with request combining was the first proposal to provide logarithmic-time reading of shared data without sequential access to the directory.

However, this recursive doubling (and that of the others) has two major deficiencies. First, this recursive doubling generates a total of $\Theta(N \log(N))$ messages. Second, this recursive doubling provides no mechanism for logarithmic-time writing of shared data when a large number of cached copies are invalidated. In this chapter, we show how this recursive doubling and request combining can be modified to provide logarithmic-time reading of shared data with only $\Theta(N)$ total messages. Furthermore, our recursive doubling leaves behind a tree structure that can be used for logarithmic-time writing of shared data. Since these extensions are based on recursive doubling, we call these extensions the *recursive-doubling extensions to SCI*. This is necessary to distinguish these extensions from the competing extensions in other chapters. Only one of these sets of extensions will become the standard, if any.

In addition to describing our protocols for logarithmic-time reads and writes, we consider the performance related to four protocol issues.

(1) Our recursive doubling is parameterized by the tree structure to be created. Given the limitation of fanout, the tree with the lowest height can be constructed. However, this is not the optimal tree for a number of reasons. First, data distribution begins on one side of the structure and invalidate distribution begins on the other side. This means that the optimal structure must minimize the height of both trees, rooted at opposite ends of the structure. The problem is illustrated in Figure 3.1. Second, the time to create the structure is dependent on the particular structure to be created. Third, the complexity of the hardware implementation must be minimized and this complexity also depends on the choice of structure. What is a good structure that minimizes all of these constraints, yet maintains the embedded list required by SCI?

(2) The request-response paradigm of SCI transactions can limit performance. If a responder knows the next request that the requester will make, then why not make that request on behalf of the requester? This is called *request forwarding*. If forwarded requests are on the critical path, then the number of messages on the critical path is reduced by one. However, request forwarding destroys SCI's arguments for forward

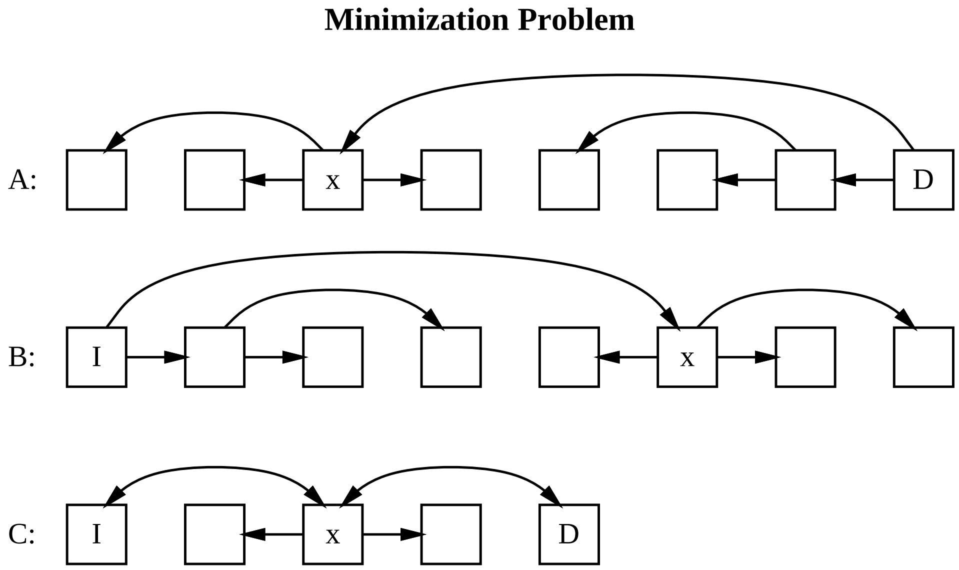



**Fig. 3.1:** This figure illustrates the minimization problem. All nodes are connected in a doubly-linked list and short cuts across the list, with at most one short cut per node. The directions of the short cuts are opposite for data and invalidate distribution, giving the appearance of two short cuts for node x in each list. List A shows the maximum number of nodes to which the data can be distributed in 2 steps, starting at node D. List B shows the maximum number of nodes to which the invalidates can be distributed in 2 steps, starting at node I. If each of data and invalidates are to be distributed in 2 steps, then the maximum number of nodes is shown in list C, although alternate routings are possible.

---

progress, so request forwarding must be defined carefully and new arguments must be presented to uphold the forward-progress guarantee. Is the performance improvement worth the added complexity?

(3) Spin waiting can significantly increase the total number of messages in the network. In a number of places in the SCI protocol, a node must repeatedly request information from another node until the desired information becomes available. Likewise, the other node may be spin waiting on the same information. Can this spin waiting can be avoided by making a reservation for the information (and immediately returning a negative response)? This is called *request reserving.* When the requested information becomes available, it is forwarded to the node that made the reservation. As for request forwarding, request reserving must be defined carefully so that forward progress is guaranteed. Note that request reserving is different from request forwarding because request

reserving assumes a need to wait. Can all spin waiting in SCI be transformed into request reserving?

(4) The performance of one part of the protocol depends on what request combining occurs (distinct from request reserving and request forwarding) in the network. By waiting to combine requests, more information may become available to make better decisions about how and when to combine requests, discussed later in Section 3.3. This is called *patient combining*. We ask four questions about patient combining. What requests should be combined? How should they be combined? How long should be waited before combining them? Does the performance justify the complexity of waiting?

In this chapter, we answer all of the above questions. We define two simple tree structures and compare their performance. We also obtain additional performance data for comparison with data in other chapters. In this chapter, we show some cases when request forwarding can be safely used and give the percent improvement for the performance. We also show how request reserving can always be used in SCI as an alternative to spin waiting. Related to request combining, we show one way to do patient combining and demonstrate that this method gives substantial performance improvement over not waiting, both in the average and worst cases.

Our conclusions are guided by research into the anticipated performance, measured in terms of latency and traffic. *Latency* is defined to be the number of network messages on the critical path, where each message takes unit time. It is reasonable to compare our protocol alternatives with the assumption of unit-time messages because none of the alternatives tries to take advantage of network locality. Furthermore, none of the asynchronous alternatives is tuned to give better performance when network behavior is synchronous, so no alternative will obtain a significant advantage over another in this synchronous analysis. *Traffic* is defined to be the total number of messages required. Larger traffic is bad in that it increases the average message delay due to network congestion.

We present the latency and traffic for sharing structures with $N$ nodes. Although this will not directly predict the performance of any parallel program, it provides a framework for the comparison of different cache-coherence protocols, paying special attention to results that affect programs with high degrees of sharing. This will also demonstrate the scalability of the Scalable

Coherent Interface with these extensions. Although it is always possible to tune a program to avoid high degrees of sharing, this tuning can be time consuming and tends to increase the complexity of a program. By providing efficient hardware support for programs with high degrees of sharing, we believe that significantly less time will be needed to get good performance out of shared-memory multiprocessors with thousands of processors.

For comparisons that are internal to this chapter, we consider the performance for each of the protocol phases, described later. For comparisons to data in other chapters, as well as for internal comparisons, we consider the performance of $N$ simultaneous reads, one write after $N$ reads, single and multiple cache-line rollouts (also known as cache-line replacements), and the producer-consumer scenario with one fixed writer and $N$ readers. For most of the protocol alternatives, we find the latency for the best case, the median case, the average case, and the worst case. Then, we compare the median cases for a number of reasons that are described later in Section 3.6, including the fact that the average is shown to be almost identical to the median. We demonstrate scalability by showing that latency of these operations with these extensions is $O(\log(N))$ and that the average traffic per node is $O(1)$. The one exception with these extensions is that the performance of multiple cache-line rollouts is not so bounded.

This concludes the introduction and motivation. Section 3.2 gives an overview for these extensions to SCI, including request combining and the optimizations of request reserving and request forwarding. Section 3.3 describes the details of request combining and gives some protocol alternatives, including patient combining. Section 3.4 discusses the performance of patient combining. Section 3.5 describes two tree structures and discusses some optimization techniques, including request forwarding. Section 3.6 discusses the performance of the two tree structures and the usefulness of request forwarding. Section 3.7 summarizes this chapter.

### 3.2. Protocol Overview

In this section, we give an overview for the recursive-doubling extensions to SCI. They are one proposed set of extensions to SCI's cache-coherence protocol, but they are not currently in vogue. These extensions are based on recursive doubling for creating the sharing structure. These extensions add one 16-bit pointer and one 16-bit counter per cache line, guarantee forward progress with only 2-phase network transactions, and have a simple combining mechanism. To simplify the discussion, we consider one line in isolation.

Consider the scenario of many readers with one writer. Before beginning, space in the cache is secured via the standard SCI protocol. If processors simultaneously read a shared line and miss in their caches, then four overlapping phases are required, bringing the data to the processors. First, the sharing list is created (or appended), generating the *list pointers,* as well as other pointers. The other pointers, called *early pointers*, enhance the list structure, to form a tree structure, by providing a variety of shortcuts across the list. Each pointer is a 16-bit node number (representing one of up to 64K nodes) that can be used to send messages. Second, the nodes work together, using these pointers, to determine each node's 16-bit list position, called a *pcount* (pronounced pee-count). Third, the pcounts are used to generate new early pointers, called *temporary pointers,* that form a better tree structure. Nodes that are connected by early pointers or temporary pointers are called *temporary neighbors.* Fourth, the tree structure is used to distribute the data to the waiting caches. It is important for good performance that the early pointers are replaced by temporary pointers, as discussed later in Section 3.2.3.

When a processor purges a sharing list, as is required for writing shared data, the *purge initiator* must first become the head (owner) of the sharing list, using the standard SCI protocol. Then, the list nodes work together to complete the purge in two overlapping phases. First, the invalidate is propagated to all nodes in the list, using the tree structure. Second, the list is purged by removing all nodes in parallel, using the bit-reversed pcounts, called *rcounts,* as priorities. When the purge initiator (the head) becomes the tail too, the purge is completed. To maintain a view of sequential consistency [Lamp79], the writing processor stalls and the cache does not release the new data to other caches until the purge is completed. However, some weaker memory models [DuSB86, AdHi90, GLLG90] allow processors to continue and data to be released before the purge is completed, a topic for future work.

Next, we discuss each of the six overlapping phases in detail, four phases for reading data and two phases for writing data. We also discuss request reserving and request forwarding.

Finally, we discuss cache-line rollout and the effects of stale temporary pointers. In Sections 3.3 and 3.5, we elaborate on possible variations for two of the more complex phases.

### 3.2.1. List Creation

The goal of efficient list creation is to create a sharing list without congesting the cache-coherence directory or parts of the network. After the list is created, the data is distributed as described later in Section 3.2.4. Hot spots not only degrade the performance of the requesting processor, but they degrade the performance of other processors that share portions of the congested path [PfNo85]. Efficient list creation is accomplished through the combining of requests in the network, as shown in Figure 3.2. Each request contains a *combinable bit*, that indicates if the request should be combined in the face of congestion. Since combining prevents all but one cache line from becoming the head, caches that intend to write the data will inhibit combining by clearing the combinable bit.

If at least two combinable requests, such as reads to the same line, are waiting in the same request queue, such as in a bridge or switch between SCI rings, then two requests can be combined into one. At the point of combining, a response to one request is immediately generated, which is sometimes delayed as described later in Section 3.3, and the second response will be generated by the directory or a future combining. The response contains at least a pointer to a neighboring node in the list, as described later in Section 3.3. A combinable request contains the two endpoints of a list segment. The endpoints are called the *segment head* and *segment tail.* Segments of size one have the same segment head and tail. When combining, the response is sent to the tail of one segment and this response contains an early pointer to the head of the other segment. The as yet unused head and tail of the segments become the endpoints of the combined request. When a request (possibly combined) finally reaches the directory, the new list head is changed to the segment head and the generated response is sent to the segment tail; the response contains the old list head and the memory's data, if valid. Note that no state is saved in the network. This makes the hardware implementation simpler than other combining mechanisms [GGKM83], avoiding contention for finite result queues and allowing a response to return via different network path. Also note that request combining tends to clump nodes together that are

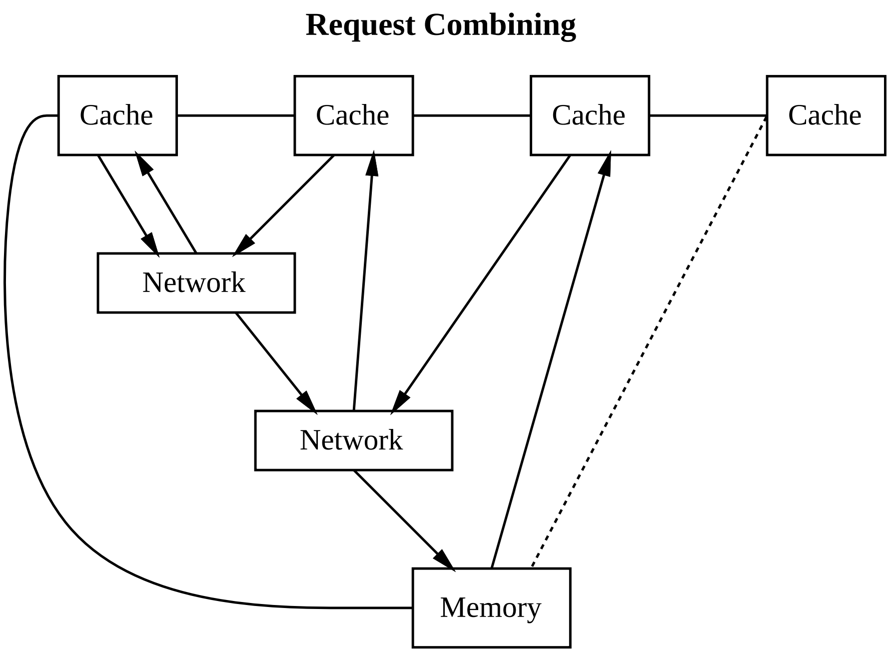



**Fig. 3.2:** This figure outlines the combining mechanism and illustrates why no state needs to be saved in the network. Dotted lines indicate previous list pointers. Arrows indicate messages, both requests and responses. Solid lines indicate newly created list pointers. In summary, two requests are combined while waiting in network queues, generating one combined request and one response.

---

relatively close to each other in the network topology, so this tends to decrease the average message delay for communication with list neighbors.

### 3.2.2. Pcount Distribution

In order for a node to know what temporary pointers to create, described later in Section 3.2.3, it is necessary for each node to know its position in the list. For each line of 64 bytes, the associated directory maintains a 16-bit wrap-around count of the number of nodes that are added to the sharing list, restarting at 1 with each new writer. If requests are not combined, each node obtains its list position, or pcount, in the directory's response. When combining occurs, segment sizes are accumulated and propagated with the requests, requiring adders in the network. Then, when a request reaches the directory, a local fetch-and-add updates the pcount and returns the correct value to be sent to the segment tail.

Acquiring the correct pcount is more complex when a response is returned from the network during combining. In this case, a node could obtain its correct pcount by getting the pcount of its forward neighbor, waiting if necessary, and adding one. However, the latency is not acceptable when a node is the head of a long segment because the pcount is sequentially distributed, starting at the tail. Therefore, early pointers are created during request combining to reduce the latency of pcount distribution. The creation of early pointers is summarized here and detailed later in Section 3.3. Essentially, each early pointer is a shortcut along the list and is accompanied by the length of the segment that it spans. These early pointers, along with the list pointers, form a tree structure for efficient distribution of pcounts. If a node receives an early pointer in its response from network combining, then it requests the pcount via the early pointer. Otherwise, it asks its forward neighbor. Each node can efficiently compute its own pcount by adding the length of the segment associated with the pointer by which the pcount is obtained. One scheme is summarized in Figure 3.3.

Note that it is impossible to efficiently determine the pcounts in a list for which the associated directory does not implement pcounts. This may be the case in mixed systems where standard SCI nodes coexist with nodes that implement these extensions. In order to interoperate correctly, extension nodes must be able to determine when this case occurs and then use the standard SCI protocols. One way to make this determination is to set a bit, perhaps in the address translation hardware, indicating when the associated directory implements pcounts. This bit should then be propagated with the requests, in the unused space that is ignored by standard SCI nodes, so that combining can be prevented when necessary. Note that requests generated by standard SCI nodes will never be combined because unused bits are always zero.

### 3.2.3. Recursive Doubling

The early pointers described above are not suitable for data and invalidate distribution (described in Sections 3.2.4 and 3.2.5), for two reasons. First, if no combining occurs in the network, then no early pointers (shortcuts) will be created. The result is that invalidate distribution will occur sequentially. This may then temporally space out future requests, thereby preventing future combining and repeating the problem. Second, if a lot of combining occurs, then the worst-case height is linear in the number of participating nodes. This will cause large latencies for both

**Pcount Distribution**

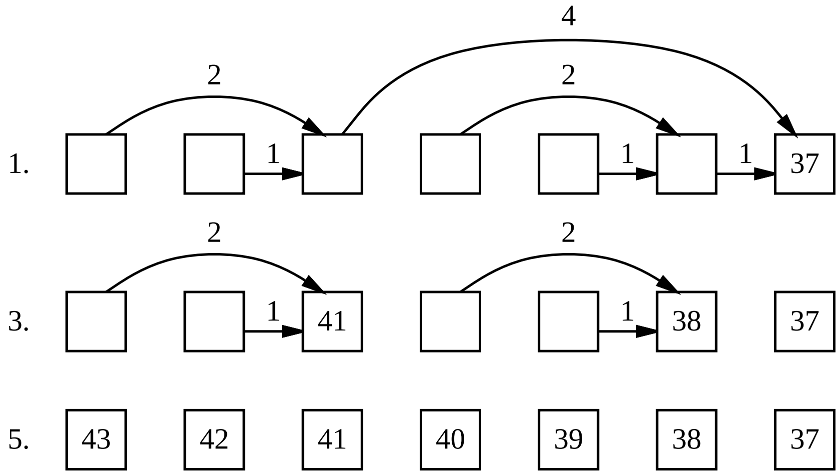



**Fig. 3.3:** This figure gives an example of pcounts distribution with early pointers that are generated during combining. Time progresses down the figure with time steps shown on the left. In one scheme, a node requests a pcount via its earlier pointer, if any. Otherwise, it requests a pcount via its forward list pointer. A numbered arrow indicates an unsatisfied request and the number that is to be added to the pcount in the expected response.

---

data and invalidate distribution, if the given early pointers are used. Trying to massage these early pointers into better pointers is too difficult. Simpler results can be obtained by discarding the early pointers and creating new pointers, called temporary pointers.

An example of temporary-pointer creation, called recursive doubling, is shown in Figure 3.4. Before beginning, each node computes the pcount of the node to which it desires a temporary pointer. We will show later, in Section 3.5, how this computation is made. In short, the computation is a function of the node's pcount and the maximum list size. After making this computation, each node gets its next temporary pointer from its current temporary neighbor (initially its forward neighbor), replacing the old pointer with the new one. This is repeated until the desired temporary pointer is acquired. The process of acquiring temporary pointers is called recursive doubling, similar to the recursive doubling of Hillis and Steele [HiSt86] and Stone [Ston73]. Use of recursive doubling for SCI was first presented by Guri Sohi in October 1989 [Gust91]. Our recursive doubling is different than Sohi's recursive doubling because the termination condition is a function of list position, rather than a function of data acquisition.

**Recursive Doubling**

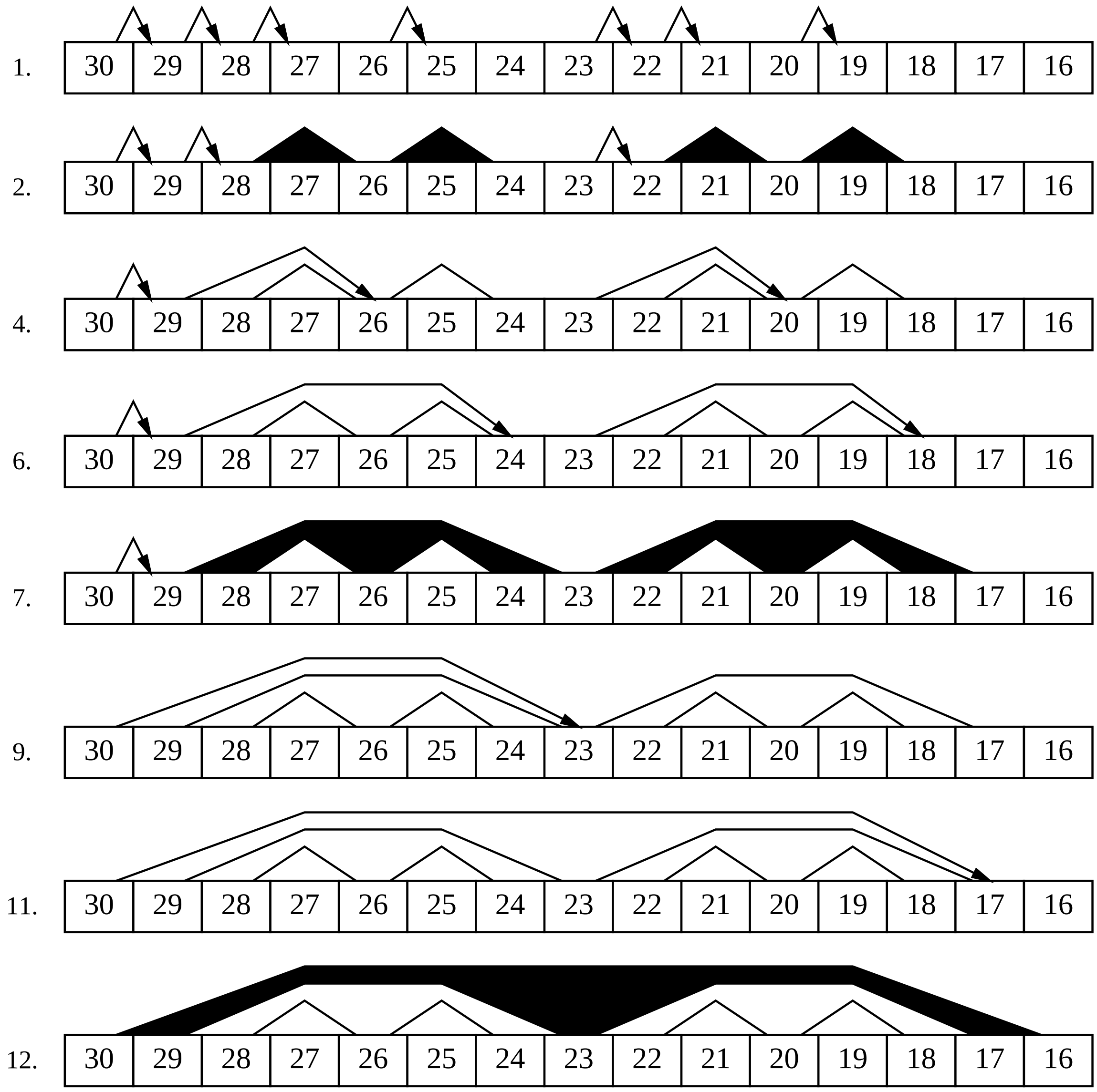



**Fig. 3.4:** This figure shows the recursive doubling mechanism with time progressing down the figure. Numbers on the left designate time steps (assuming unit time per message), some steps not shown. Arrows represent requests and reservations. Lines, including edges of shaded areas, represent final temporary pointers. The recently completed temporary pointers are highlighted by the shading.

---

If a node receives a temporary-pointer request before it is finished with recursive doubling, then the node sets a reservation flag, returns a negative response immediately, and, after finishing, sends the requested temporary pointer in a new request. This is called request reserving.

For an example, consider node 29 in the figure. It desires a temporary pointer to node 23. First, it asks node 28 for its temporary pointer (step 1). Note that at the same time, node 28 is asking node 27, but node 27 does not do the same. After 27 responds to 28, 28 sends to 29. Node 29 then asks node 26 for its temporary pointer (step 4) and 26 responds with a pointer to node 24. Finally, a request to node 24 (step 6) returns a pointer to 23 and then node 29 is done with its recursive doubling. Assuming node 30 has been waiting all this time, node 29 also sends its final temporary pointer to node 30 so that node 30 can continue. The forward progress concerns related to request reserving will be discussed with data distribution, which is discussed next.

### 3.2.4. Data Distribution and Request Reserving

After the temporary pointers are created, efficient data distribution is straightforward, as shown in Figure 3.5. If a node has a temporary pointer, it requests the data from that node. Otherwise it requests the data from its forward neighbor. The figure shows which nodes receive the data as time progresses down the figure. If a node is given the data while generating its temporary pointer, then there is no need to request the data. However, the node must still complete the recursive-doubling phase so that its temporary pointer is available for the subsequent invalidate distribution.

The request-response paradigm can be used for pcount distribution, recursive doubling, and data distribution. However, a simple optimization exists, called request reserving, that will significantly improve the latency and reduce the traffic. If information (pcount, temporary pointer, or data) is requested and the information is not yet available, then a reservation bit is set in the responder's cache line, a response is immediately returned, and the information is forwarded when it arrives. This forwarding generates another request.

In SCI in general, it is forbidden for one request to generate another. This is because waiting for finite resources might lead to circular wait and deadlock, as described in Section 1.3.4.1. However, in the special case of request reserving, circular wait can be avoided because there is already a request controller that is prepared to do the forwarding when the information arrives, eliminating the need to wait for one. A *request controller* is an entity in a cache controller that executes the cache-coherence protocol. Lower layers of the SCI protocols guarantee that the request controller will eventually be able to send its next request.

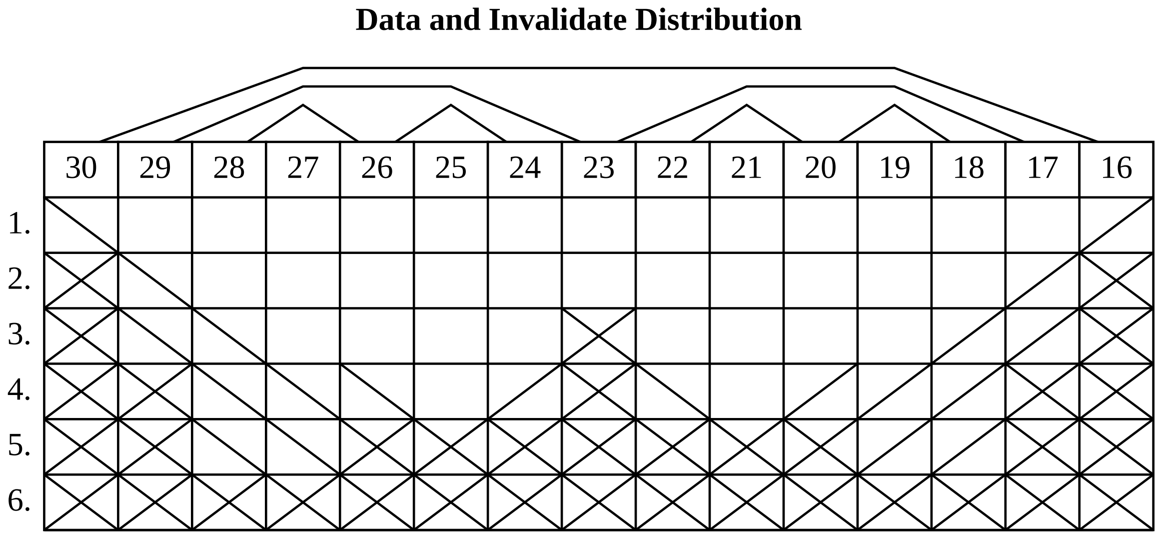



**Fig. 3.5:** This figure shows the separate distributions of data and invalidates, using the temporary and list pointers. Temporary pointers are shown above the nodes. List pointers are implied by touching nodes. Time is shown progressing down the figure with time steps shown on the left. Nodes that have received the data have a slash. Nodes that have received an invalidate have a back slash. Note that this figure illustrates two separate distributions that are generally *not* concurrent, one distribution for data and one for invalidates.

---

Since temporary pointers are nested, at most two reservation bits are required (encoded in the tag state), one for the list pointer and one for the request that comes via a temporary pointer. When a reservation is made via a temporary pointer, then it is necessary to save a pointer to the requester, called a *request pointer,* so that the information can be sent later. This request pointer is saved in the global space of the request controller. If a cache controller has more than one request controller, then a small translation table is also required for the cache so that an incoming reservation can find the correct request controller. The request pointer is not saved in the cache line, so the additional space is proportional to the number of request controllers rather than proportional to the (larger) number of cache lines. Note that, in general, it is impossible to save the request pointer in any of the cache-line locations that are reserved for unavailable information. This is because the request pointer is overwritten when the information arrives.

### 3.2.5. Invalidate Distribution and Request Forwarding

Invalidate distribution is similar to data distribution, as also shown in Figure 3.5. The main difference is that invalidates are forwarded (data-driven) rather than requested (demand-driven).

In other words, nodes do not know when to expect an invalidate and we must use request forwarding rather than request reserving. Another difference is that invalidates are forwarded via both temporary and forward pointers, rather than just one pointer. Since invalidate distribution uses request forwarding, the potential unavailability of a request controller, as described in Section 1.3.4.1, complicates the guarantee of forward progress.

Forward progress for invalidate distribution is guaranteed as follows. If an invalidate request can be forwarded (there is an available request controller), then there is no problem. If an invalidate request can not be forwarded by a node and the invalidate came from the node's backward neighbor, then the problem node removes itself from the list by returning its forward pointer in the response to its backward neighbor. The requester can then send an invalidate to its new forward neighbor (similar to the standard SCI protocol). If an invalidate can not be forwarded and the invalidate came via a temporary pointer, then the invalidate is ignored and invalidate distribution will correctly complete via the list pointers.

### 3.2.6. List Purging

After the invalidate has been distributed, the list is collapsed. Purging is actually started as soon as a node receives an invalidate, but for simplicity we describe purging as a distinct phase. Note that it is not efficient for nodes to simply try to remove themselves from the list. If neighboring nodes try to remove themselves simultaneously, the standard SCI protocol gives priority to the most forward node. This causes sequential purging in the worst (and common) case.

To prevent sequential purging, nodes use their bit-reversed pcounts, called rcounts, as priorities. This is shown in Figure 3.6. A node sequentially purges its forward neighbors until its rcount is greater than the rcount of its forward neighbor. Then, it stops and waits to be purged by a backward neighbor. If a node is busy purging when it receives a purge request, it tells the requester to try again. However, if the node's rcount is not greater than the requester's rcount, then it tells the requester to stop. There is one exception to the stopping rule. The purge initiator (usually a writer) never stops purging its forward neighbors, until it has no more forward neighbors. When this happens, all stale copies are invalidated and the purging process is complete.

**Purge Removal Ordering**

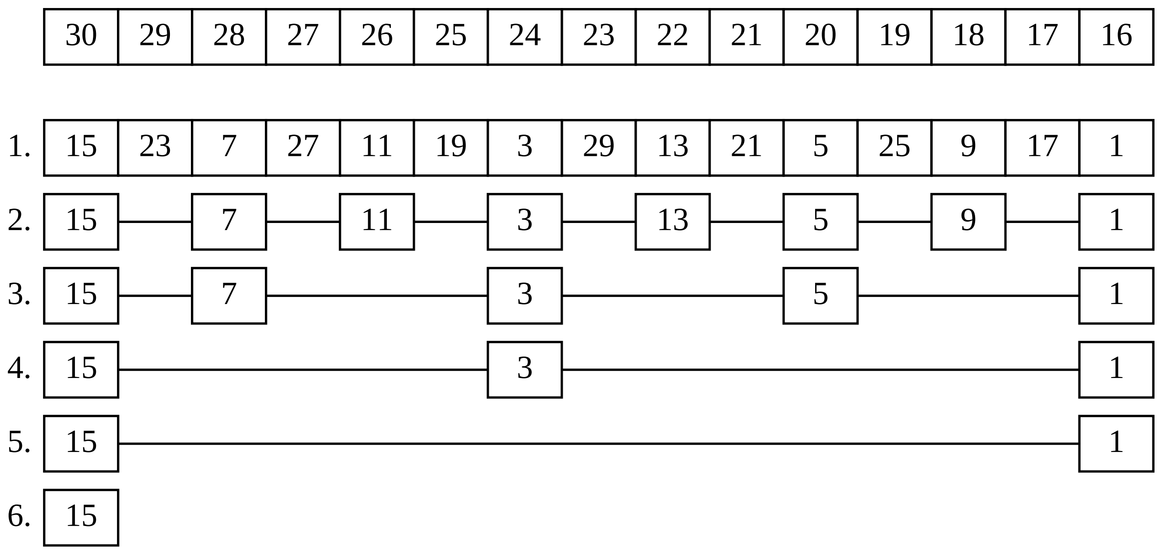



**Fig. 3.6:** This figure shows the purging of a list after all nodes have received an invalidate. All numbers are rcounts, except for the pcounts on the top row. Node 30, with rcount 15, is the purge initiator. Lines represent list connectivity, some connectivity being represented by touching nodes. Time progresses down the figure with time steps shown on the left. The absence of a node indicates that it has been purged.

---

With synchronous purging and the absence of early node removals (due to cache-line rollout), a node can predict the comparison between its own rcount and its forward neighbor's rcount[15]. This prediction is based on the fact that in a fully populated list a neighbor's rcount is smaller when either 1) the node's own rcount is greater than half of the maximum or 2) the difference between the node's rcount and the rcount of the previous neighbor is a power of two. Then, in the absence of early node removals, a node can avoid its last request by stopping early, saving 2 messages. If there are some early node removals, then a node other than the purge initiator may stop early or late, possibly causing a change in performance. The rcount comparisons

[15] Some of the simulations, discussed in Section 6.3.1.2, incorrectly predict the comparison due to the overlap of invalidate distribution and purging. This is not a problem for correctness, as discussed in the text.

are not actually performed and purge completion is guaranteed by the repeated purge requests of the purge initiator.

When purging a node's forward neighbors, the latency of purging each node is 2 messages, one request and one response. If a node can forward a purge request, instead of simply responding, then purge latency can be reduced. This request forwarding is called *fast purging* [ThDe90a]. The purge request is forwarded towards the tail of the list and then the tail responds directly to the head. Fast purging was first presented by Manu Thapar for use with simple lists and his idea was finally accepted by the SCI working group in April 1990 [Gust91] for use with these extensions. In order to modify Thapar's idea to work with rcounts, the rcount of the requester is included in the purge request. When a node receives a purge request, it compares the requester's rcount to its own. If the difference is not power of two, then the node forwards the request and is done purging. If a node would forward the request, but it is currently busy, then request reserving is used. Otherwise, if the difference is a power of two, then the request should not be forwarded. Forward progress is guaranteed if a node, after purging, does not release its request controller until it is purged. This essentially reserves the request controller for the future possibility of forwarding.

### 3.2.7. Cache-Line Rollout and Stale Temporary Pointers

When nodes are removed from the sharing list, as is required for cache-line rollout, the temporary pointers are not updated. This reduces protocol complexity for the extensions because the existing SCI protocols for cache-line rollout can be used exactly. To summarize, a node connects its forward neighbor to its backward neighbor and then it connects its backward neighbor to its forward neighbor. When adjacent nodes are removed from the list simultaneously, priority is given to the most forward nodes, resulting in sequential rollout. Although sequential rollout is a performance disadvantage, there is no known way to easily improve the protocol. Note that using rcounts as priorities is messy because a node must be able to undo its request to connect its forward neighbor to its backward neighbor if it finds out that its backward neighbor has priority. This undo may also require an undo on the part of the node's forward neighbor, and so on. Also note that a node can not check the rcount of its backward neighbor before beginning the rollout because this increases the latency of a single rollout, which is the most common case.

Therefore, the SCI protocol for cache-line rollout is used exactly without resolving the problem of sequential rollouts.

Another problem with the rollout of cache lines is that a rollout can cause a temporary pointer to become stale, since temporary pointers are not updated during rollout. The existence of stale temporary pointers means that the staleness of a temporary pointer must be checked when it is used. Otherwise, an invalidate may purge unrelated cache lines. Or worse, recursive doubling may connect two (or more) distinct sharing lists, causing a mixing of the associated data. The primary check for a stale temporary pointer is a simple comparison between the address in the request packet and the address in the cache-line tag of the responder. If the addresses do not match, the request is nullified and the requester resorts to the standard SCI protocol.

It is possible that the use of a stale temporary pointer will not be detected by a simple comparison of the memory addresses. This can occur when a node reenters the list after rollout. During reading, if an attempt is made to set a reservation flag for the second time, then the problem is detected. In this case, the request is nullified and the requester resorts to the standard SCI protocol. If the wrong node sets a reservation, correct data is still distributed to all readers. During writing, if an invalidate is received at a node that does not have the data, then the problem is detected. In this case, the responder nullifies the request and the requester does nothing special. Invalidates from stale temporary pointers may go undetected, but spurious invalidates only affect performance, not correctness.

Recall that during recursive doubling, temporary pointers are created between nodes with particular pcounts. If the destination node of a temporary pointer does not exist, because it was removed from the list due to cache-line rollout, then it is impossible to create the desired temporary pointer. In this case, the requester does not generate any temporary pointer, using the standard SCI protocol to get the data from its forward neighbor.

Handling stale pointers necessarily complicates the protocol. In addition, stale temporary pointers degrade the protocol's performance. Latency is increased because some of the list shortcuts cease to exist. Latency is also increased because time is wasted as a stale pointer is discovered. Furthermore, network bandwidth is wasted as a stale pointer is discovered.

### 3.2.8. Overview Summary

To summarize, there are four overlapping phases for processor reads and two overlapping phases for writes. For reads, the reader must create space in the cache before beginning. Then, the list is created first, using combining to avoid congestion at the directory. Second, the pcounts are distributed using the early pointers and segment lengths that were generated during combining. Third, recursive doubling creates the temporary pointers that are determined by the pcounts. Fourth, data is distributed using the tree formed by the temporary and list pointers.

For writes, the writer must first become the head of the list. Then, invalidates are propagated through the list using the temporary and list pointers. After that, list purging is completed, in parallel, with the aid of rcounts (bit-reversed pcounts) as priorities. If stale temporary pointers exist, a byproduct of cache-line rollouts, then the performance of writing (and reading) will be degraded.

Temporary pointers increase the cache-tag size by 16 bits per cache line. In addition, pcounts increase the cache-tag size by 16 bits per cache line and they also increase the directory size by the same amount. Furthermore, a few more bits are required for additional cache and directory states, as well as reservations in each cache line. Overall, in comparison to the standard SCI cache-coherence protocol, the recursive-doubling extensions to SCI require about six percent additional cache-line space and about three percent additional memory space, considering the directory and related memory data as a single unit.

## 3.3. Request Combining

When many processors simultaneously miss on the same line, the directory becomes a hot spot. Simultaneous requests for the same instructions or data are likely to happen when all processors enter a new program phase after a barrier. In order to prevent a directory hot spot, requests are combined. Any set of read operations on the same line can be combined, assuming the combinable bit is set. It is also possible to combine similar associative update operations (like fetch-and-add), but this is a topic for future work. Combining is expected to take place in the network queues between SCI rings. It is unlikely that combining will be implemented in the FIFO queues of a single ring due to the aggressive timing constraints. There are two main questions to answer about combining: how to combine requests and when to combine them.

The primary performance objective of combining is to minimize network congestion, specifically at the directory. The functional objectives of combining are 1) to create a sharing list and 2) to create a tree structure for pcount distribution. As a result, another performance objective is to minimize the tree height for pcount distribution. The two methods of combining are now described, free-node combining and two-pointer combining, answering the question of how to combine. After that, patient combining is described, answering the question of when to combine.

### 3.3.1. Free-Node Combining

For free-node combining, we first discuss the combining of previously combined requests and then we discuss the cases for combining single requests. Each combined request contains three nodes, as shown in Figure 3.7, and the length of the combined segment. Two of the nodes are the segment head and tail and the third node is an unconnected node, called the *free node.* Before the combining, both the segment tails and the free nodes have not yet been sent a response, indicated by the circles in the figure, but responses have already been sent to the segment heads and all the other nodes in the segments.

When two previously combined segments combine, the first head (node A) becomes the new head, the first free node (node C) becomes the new tail, and the second free node (node F) remains a free node for later combining. Two responses are sent, one to each of the segment tails (nodes B and E), that connect them to either the head of the second segment (node D) or the first free node (node C) respectively. In addition, the response to the first tail (node B) also contains an early pointer to the first free node (node C) and the length of the second segment. This early pointer is later used for pcount distribution, equating the tail of the new list (node C) to the root of a probabilistically balanced binary tree.

When a combined request reaches the directory, two responses are generated, one to the segment tail (linking it to the free node) and one to the free node (linking it to the old head of the sharing list). Recall that a response from the directory also returns the pcount before the pcount

**Free-Node Combining**

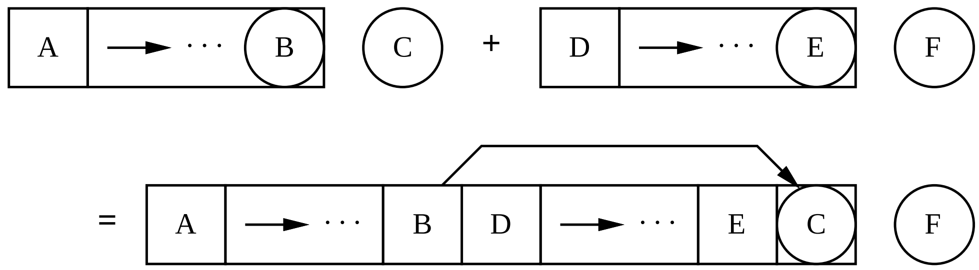



**Fig. 3.7:** This figure illustrates the combining of two previously combined requests, using free-node combining. Letters represent node identifiers, boxes represent list segments, and circles represent nodes that have not yet been sent a response. The two original requests on the top are separated by a plus sign. The combined request on the bottom shows the early pointer that is created by the response that is sent to node B.

---

is updated (local fetch-and-add). Since an SCI directory is located with memory, data is also included in the response if the data is valid.

The base cases for free-node combining are straightforward and are shown in Figure 3.8. In case A, two uncombined requests combine to form a combined request with one segment of size one and one free node. The left node in the combined request is both the head and tail of a segment. It has not yet been sent a response. Responses are delayed until further combining or until the combined request reaches the directory. In case B, one uncombined request becomes the free node when combining with an already combined request. The free node of the old combined request becomes the tail of the new segment and one response (with no early pointer) is sent to the tail of the old segment (the leftmost node for case B). Two nodes have not yet been sent responses. Case F is the same as given in Figure 3.7 and cases C, D, and E follow the pattern.

Free-node combining appears to work well. Each node receives exactly one response with at most one early pointer. Also, the request-response paradigm can be used for pcount distribution, unlike two-pointer combining (described later in Section 3.3.2). The major disadvantage of free-node combining is that responses are delayed. This means that the directory must return two responses per combined request, rather than one, and up to half of the combinings in the network must also return two responses. Furthermore, the directory must implement pcounts. Due

**Combining Free Nodes and Segments**

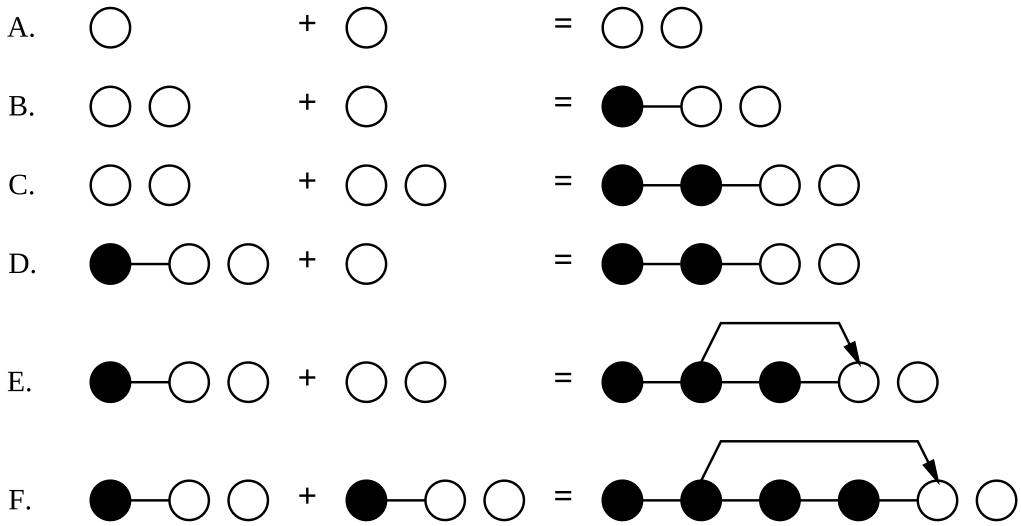



**Fig. 3.8:** This figure shows all the cases for free-node combining. Unattached circles represent the free nodes. Black and white circles indicate whether or not, respectively, a node has been sent a response. Lines and arrows indicate list pointers and early pointers, respectively, that have been (or will be) established by the already sent responses. The equal signs separate the original requests from the resulting request. The plus signs separate the two original requests. The letters label the cases for reference by the text.

---

to these requirements, free-node combining is not compatible with standard SCI directories. If free-node combining is used in mixed systems, then these extensions must avoid combining of requests destined for standard SCI directories, as described in Section 3.2.2.

### 3.3.2. Two-Pointer Combining

For two-pointer combining, each combined request contains three nodes, as shown in Figure 3.9, and one length for each of the two segment halves. Two of the nodes represent segment end-points and the third node represents an approximate midpoint. When two segments combine, the first segment head (node A) becomes the new segment head, the second segment tail (node F) becomes the new segment tail, and the first segment tail (node C) becomes the midpoint of the new segment. To connect the new midpoint (node C) to the second segment head (node D), one response is sent (to the new midpoint, node C). This response contains two early pointers to the midpoints of the old segments (nodes B and E) and the corresponding segment lengths. These early pointers make the midpoint of the new list (node C) act like the root of a binary tree that

**Two-Pointer Combining**

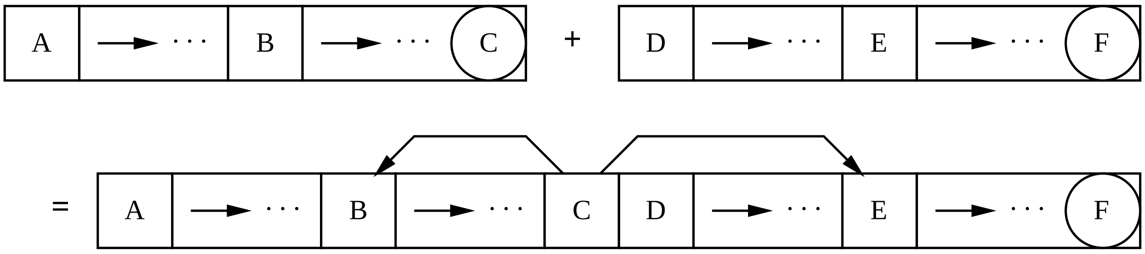



**Fig. 3.9:** This figure illustrates the combining of two previously combined requests, using two-pointer combining. Letters represent node identifiers, boxes represent list segments, and circles represent nodes that have not yet been sent a response. The two original requests on the top are separated by a plus sign. The combined request on the bottom shows the two early pointers that are created by the response that is sent to node C.

---

will be used later for pcount distribution. After combining, only the new segment tail (node F) has not yet been sent a response, indicated by a circle in the figure.

When a combined request reaches the directory, a response is returned to the segment tail. This response contains a pointer to the head of the sharing list and one early pointer to the mid-point of the segment in the current request (with the appropriate length). Recall that a response from the directory also returns the pcount before the pcount is updated at the directory. Since an SCI directory is located with memory, data is also included in the response if the data is valid.

The base cases for two-pointer combining are straightforward and are shown in Figure 3.10. Cases A through F show the combinings for various sizes. There is a distinction between single nodes (segments of size one), short segments (segments of size two), and long segments (segments of size at least three). Early pointers are generated only for combining pairs that contain no single nodes and at least one long segment, as shown in cases E and F.

Pcount distribution is much different for two-pointer combining than for free-node combining because the early pointers point to the destinations of the pcounts rather than to the sources. This means that nodes must wait for their pcounts and then forward them, rather than use a request-response paradigm. The guarantee of forward progress is not compromised by this forwarding because a request controller is already active (and available) for the given cache line.

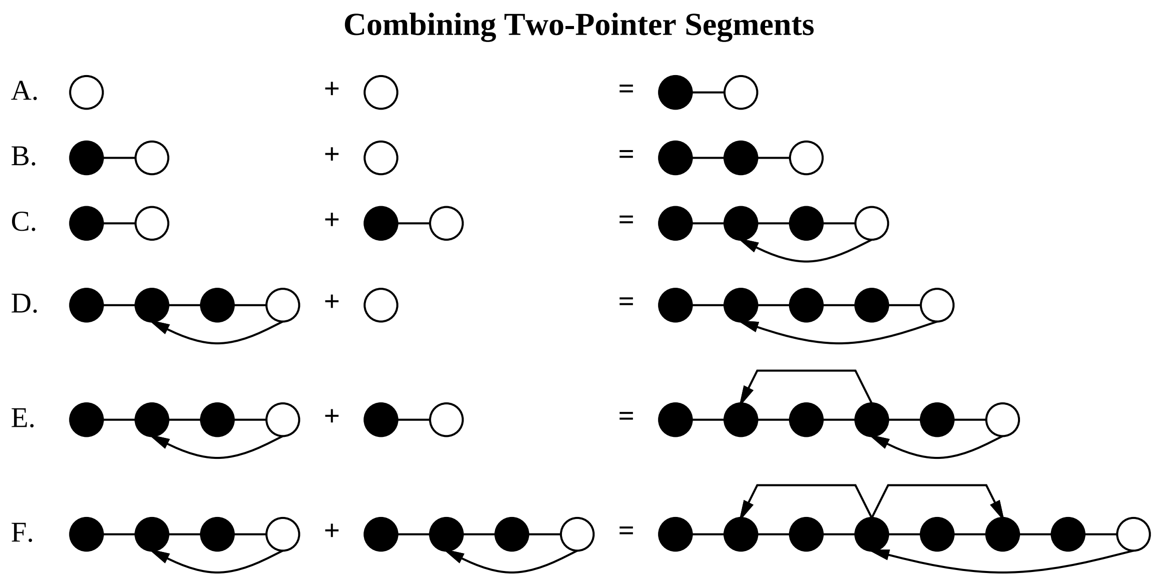



**Fig. 3.10:** This figure shows all the cases for two-pointer combining. Unattached circles represent the free nodes. Black and white circles indicate whether or not, respectively, a node has been sent a response. An arrow under a list represents the pointer that would be returned in a response from the directory, also indicating the midpoint. Other lines and arrows indicate list pointers and early pointers, respectively, that have been (or will be) established by the already sent responses. The equal signs separate the original requests from the resulting request. The plus signs separate the two original requests. The letters label the cases for reference by the text.

---

Two-pointer combining has the advantage over free-node combining in that only one response is returned by the directory and by the combining in the network.

Two-pointer combining has two main disadvantages. First, although the directory returns only one response, it is now responsible for returning an early pointer with the associated length. Due to this requirement, two-pointer combining is not compatible with standard SCI directories. If two-pointer combining is used in mixed systems, then these extensions must avoid combining of requests destined for standard SCI directories, as described in Section 3.2.2. Second, up to two early pointers and associated lengths are returned per response, either requiring more storage space in the cache tag or in the temporary space of the request controller.

### 3.3.3. Patient Combining

Free-node and two-pointer combining are two alternatives for how to implement combining, but they do not describe when to combine. Here we describe two philosophies, and their associated algorithms, for deciding when to combine. The philosophy of *greedy combining* states that two requests should be combined whenever possible, thereby minimizing the network traffic. Then, according to the philosophy of greedy combining, combining takes place whenever two combinable requests coexist in the same network queue. Greedy combining is good in that it reduces the number of unsatisfied requests as soon as possible, resulting in the use of less network bandwidth.

However, greedy combining can create an early-pointer tree that is unbalanced, resulting in poor latency for pcount distribution. In the worst case, all combinings could contain at least one noncombined request. This results in sequential distribution of pcounts because there are no early pointers. Both the order of the worst case and the magnitude of the average case can be improved.

The philosophy of *patient combining* states that segments of near-equal tree heights should be combined, thereby minimizing the latency of pcount distribution. With patient combining, the average- and worst-case latencies are reduced significantly, as shown later in Section 3.4. In order to determine the difference of tree heights, each combined request must also contain the associated height. This does not increase the size of a request packet because there is unused space in the extended header[16] and the extended header is already required by a combined request. Non-combined requests have an implied height of one. Given the heights for every combinable request in a network queue, the algorithm for patient combining is given in Figure 3.11 and described below.

When a new request enters a combining queue, the queue first checks for combinable requests (the first `if`). If a set of $X \geq 2$ combinable requests exists, where $X$ is an implementation parameter, then the queue combines the two of them with the nearest tree heights (possibly the same tree height). Ties are broken by choosing the tree heights that are the smallest and then by choosing the request that will be first to leave the queue. If $X$ is two, we have greedy combining. If $X$ is greater than two, then we have patient combining of degree $X$. Note that this first

[16]Normally, an SCI packet has a 16-byte header. An extended header is the optional 16-bytes of additional header, yielding a 32-byte header.

**Patient Combining**

```
for (each new request entering the combining queue)
{
  if (there are X combinable requests)         /* X is an impl. parameter */
    combine two with nearest tree heights,     /* avoid traffic; combine  */
      choosing smallest heights to break ties;
  while (two combinable requests have same tree height)
    combine that pair immediately;             /* optimal combining       */
  if (queue is near full and two combinable requests exist)
    combine any nearest tree heights,          /* avoid tree saturation   */
      choosing smallest heights to break ties;
}
```

**Fig. 3.11:** This figure shows the algorithm for patient combining. A *tree height* is given (or implied) in each request. Two requests are *combinable* if they are read requests for the same line (or if they are the same fetch-and-Φ request for the same line, a topic for future work). The pair with the *nearest* tree heights is the pair of combinable requests with the smallest difference of heights. A queue is *near full* when it can not hold another request of maximum size.

---

step need not be repeated because the queue is always left with combinable sets of size less than $X$. Also note that the queue need only check requests that are combinable with the new request.

Second, the queue combines any combinable pair of equal heights (the `while` loop). This reduces network traffic, as does greedy combining. If two requests have the same height, there is no performance advantage to being patient. Note that this step can occur at most $X-2$ times, at most once if $X=3$. Also note that the queue need only check for equal heights between the most recently combined request and other requests that are (were) combinable with the new request.

Third, if the queue can not hold another request of maximum size, then the queue combines any two combinable requests, if possible, choosing a pair with the nearest heights (the second `if`). Ties are again broken by choosing the tree heights that are the smallest and then by choosing the first to leave. This third step is a panic mode to avoid tree saturation. In this case, the queue can not limit its search to requests that are combinable with the new request. All requests must be considered. Of course, this third step is not required for correct behavior. However, it may help with performance under heavy load and it guarantees that the performance of patient combining is never worse than greedy combining.

Patient combining is not much more complicated than greedy combining. Both require an associative lookup in the queues between SCI rings, where combining occurs. Since atomicity of the lookup is not required for correctness[17], the lookup can be implemented as a sequential search through the queue. Note that a lookup need not delay a request from exiting a queue because the lookup can be aborted and the combining attempt repeated. Given a sequential implementation of the lookup, it is relatively simple to implement the algorithm for patient combining with at most one sequential scan per conditional expression in Figure 3.11.

### 3.4. Patient-Combining Performance

In order to avoid congestion at a heavily-used directory, read requests for the same line are combined in the network. Recall that recursive doubling requires that every node knows its position in the list, called a pcount. Although a response from a directory contains the needed pcount, combining in the network means that some nodes will not initially receive a pcount. Therefore, during the combining, early pointers are generated to help efficient distribution of the pcounts. These early pointers, in collaboration with the list pointers, form a tree structure. Without the early pointers, if only list pointers are used, this distribution would be linear in the size of the combined segment, as large as the whole sharing list in the worst case.

In Section 3.3, we discussed two ways to generate early pointers, free-node combining and two-pointer combining. Free-node and two-pointer combining have the same traffic: exactly one request and one response per node. They also have the same latency; the proof-sketch is as follows. First, take any two segments such that if they reached the directory, then they would have tree heights of $g$ and $h$. Second, combine these two segments, noting the combined height. If the new segment was to reach the directory, then the combined height is $1 + \max(g, h)$ for both combining methods. Therefore, the two mechanisms have the same performance.

If we consider the fact that free-node combining delays responses and assume that this means that the average response must travel a greater distance, then the performance of

[17] Combining can be aborted if a participating request exits the queue. However, an atomic update is required.

two-pointer combining is slightly better. However, if we maintain the given granularity of our analysis, assuming unit time per message, then there is no performance rationale for choosing one mechanism over the other. Since these two solutions have the same performance, the better of the two is the one with the simpler implementation. Based on the advantages and disadvantages of the two methods, discussed in Section 3.3, free-node combining is best.

In Section 3.3, we also discussed greedy and patient combining. Recall that greedy combining always combines two requests whenever possible. However, if a network queue waits for $X \geq 3$ combinable requests to enter it, where $X$ is an implementation parameter, then there is a choice of which pair to combine. In particular, waiting to combine segments with near-equal tree heights yields significantly better latency for pcount distribution than for greedy combining, as we now show. We also show that the latency improvement for going from $X$ to $X+1$ decreases rapidly with increasing $X$. This is important because implementations with large $X$ are significantly more expensive to build. First, we discuss the simulator for the average cases. Second, we derive the equations for the worst cases, showing that the worst-case latency for pcount distribution can be reduced from $\Theta(N)$ to $O\left(N^{1/2}\right)$. Third, we discuss the simulator results for the average cases.

### 3.4.1. Simulations

Now we discuss the simulator that generates the data for the latency of pcount distribution. As noted earlier, the latency depends on the implementation parameter $X$, which is the maximum number of combinable requests that coexist, without combining, in any one queue. The results are identical for free-node combining and two-pointer combining, as also noted earlier, because two arbitrary trees combine into one tree that is the same height under both combining mechanisms.

The network simulator begins with a pool of $N$ segments, where $N$ is the list size. Each segment is of size one and is combinable with any other segment in the pool. This initial pool of segments represents $N$ simultaneous read requests. At each iteration the simulator takes a random set of $X$ segments, combines the pair of segments with the smallest height difference, and returns the $X-1$ segments to the pool. When there are less than $X$ segments left in the pool, the simulation stops. During this combining, the directory is simulated by draining one segment from the pool after every $\max\left(\frac{P}{8}, 1\right)$ combinings, where $P$ is the number of segments left in the pool. The data point for one simulation is the maximum height of the drained and remaining segments, giving the early-pointer tree height for a list of $N$ nodes. The average case is based on 1000 simulations.

We also simulate two flavors of combining without draining. No drain means that the segments are left to combine in the network until there are $X-1$ (or less) segments left. One simulation modifies the combining so that all equal-height segments in the randomly chosen set are combined before the set is returned to the pool. The other simulation combines exactly one segment per iteration. Then, the difference in the output of these two simulations will show the usefulness (or uselessness) of greedily combining all of the equal-height segments, as specified by the `while` loop in Figure 3.11 on page 91. Also, the results of these extra simulations, without a drain, will act as a check on the results for the primary simulator, with a drain. It will be shown that all simulators support the same conclusions.

Before giving the performance data, let us intuitively argue why these simulations provide a good way to compare patient and greedy combining. The main reason why these simulations are a good comparison measurement is that the probability of finding the optimal mate for a given segment is strongly related to the percentage of possible mates that are optimal. In other words, the probability of combining a segment of size $S$ with another segment of size $S$ is related to the percentage of segments in the pool that have size $S$. Intuitively, this seems to approximate the behavior of a real network as follows. Any request is available to combine with any other request. Small segments may combine with large segments and equal sized segments may combine with each other. The probability of each combining depends on the percentage of requests of the given sizes. In a real network, many small segments will exist, but there will be few large ones. In the simulators, small segments abound and the percentage of large segments is small until the very end. Furthermore, large segments do not magically appear in real networks. They must be built from smaller segments. Likewise, the simulator builds large segments from small ones.

The slow drain on the pool simulates the directory that removes segments from the network at a constant rate; the drain rate is equivalent to the rate at which each queue can combine requests. Without this drain, the other simulators find inflated values for the tree height. Consider the rate at which segments are drained from the pool. When a large number of requests are headed for the directory, the network queues near the directory become full and tree saturation occurs. The number of saturated queues is proportional to the number of unsatisfied requests divided by the number of requests that each queue can hold, an implementation parameter that is perhaps between 4 and 8 for large systems. In addition, there are a large number of partially saturated queues at the leaves of the saturated tree, where the root is the node with the directory. These leaf queues are, on average, about half full, so the number of affected queues may be up to doubled, depending on the structure of the saturated tree and the network topology. Also, more queues may be affected due to the presence of uniform traffic, other requests that use queue space on the congested paths. On the other hand, the probability of combining in the leaf queues is reduced with increasing $X$ and the probability of combining in any queue is also reduced with increasing uniform traffic. Therefore, we estimate that the number of segments that are combined per directory response is $\frac{P}{8}$, which is approximately the number of affected queues.

Finally, in argument for their usefulness, these simulations do not consider the network topology. Measuring performance in a topology-independent way is the preferred method when studying a standard and/or a proposed extension that is suppose to be topology independent, such as SCI. Although actual performance will vary from topology to topology, patient combining will be shown to be better in general, which is the appropriate criterion in this case. Of course, algorithms that exploit the topology may do better than topology-independent algorithms and improve performance more than patient combining. However, exploration of topology-dependent algorithms is beyond our scope.

### 3.4.2. Worst Cases

For greedy combining, the worst case is when one segment of the two combinable segments is always size one. In this case, no early pointers are created and the tree height for pcount distribution is $N$, assuming no drain on the pool. In the worst case with pool draining, the directory always drains a segment of size 1, leaving the bigger segments to continue combining. The results for simulations of the worst case with draining will be given later in Section 3.4.3.

If patient combining with $X = 3$ is used without draining, then the worst-case tree height for pcount distribution is

$$h = \left(4N + \frac{9}{2} \pm \frac{1}{2}\right)^{1/2} - 2,\ N \geq 11, \tag{3.1}$$

where $N$ is the number of nodes in the sharing list. This is approximately $h = 2\sqrt{N} - 2$ for large $N$. The hard part for the derivation of this equation is discovering and solving the recurrence relation, $N(h) = N(h-2) + h + 1$, where $N$ is the number of nodes in the sharing list and $h$ is the worst-case tree height for a segment without draining. The recurrence relation is derived from a worst-case combining that is shown in Figure 3.12. The example is a worst case because the difference in tree heights for two combined segments is minimized. Solving the recurrence relation yields

$$N(h) = \frac{2h^2 + 8h - 1 + (-1)^h}{8},\ h \geq 5. \tag{3.2}$$

Replacing $(-1)^h$ with $\pm 1$ and solving for $h$ gives Equation 3.1, the maximum tree height without draining in terms of the number of nodes in the sharing list.

When the worst case with draining is considered, the directory always removes a segment of size 1. Then, the number of nodes required for a given height is increased by the required number of nodes to feed the drain. The number of feedings can be easily computed by simulation, given the $N - 2$ combinings that occur. The number of feedings is then added to $N(h)$, given in Equation 3.2, before solving for $h$. However, the number of feedings is negligible for the worst case with large $N$.

Equation 3.1 shows that the worst-case tree height for pcount distribution is improved from $\Theta(N)$ to $\Theta\left(N^{1/2}\right)$ for $X = 3$. Furthermore, we hypothesize that the recurrence relation for arbitrary $X$ is $N(h) = N(h - X + 1) + \Theta\left(h^{X-2}\right)$. If this holds true, then the latency for pcount distribution can be reduced to $\Theta\left(N^{\frac{1}{X-1}}\right)$ for constant $X$ or $\Theta(\log(N))$ for $X = \Theta(\log(N))$. In other words,

**Worst-Case Patient-3 Combining**

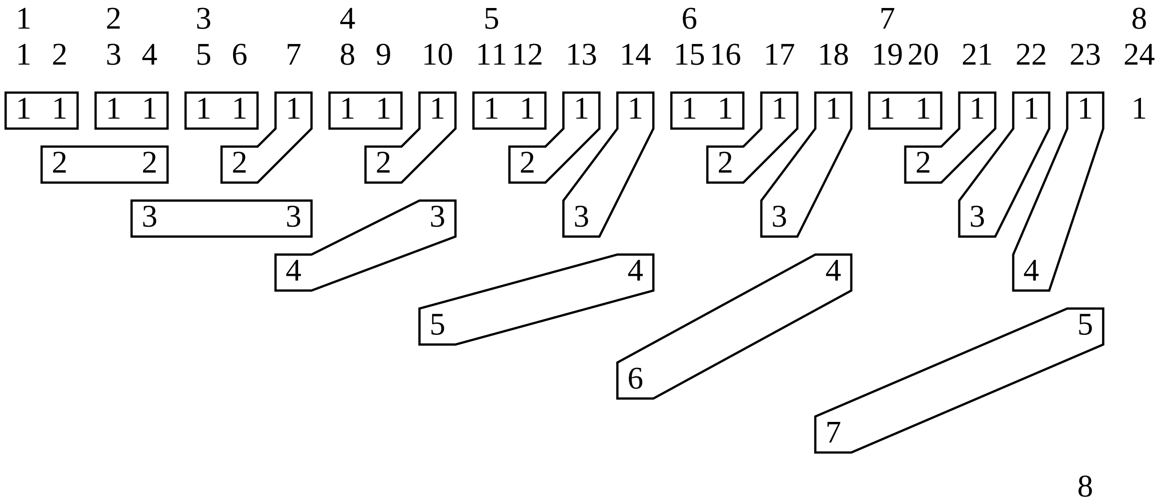



**Fig. 3.12:** This figure shows a worst-case patient combining for $X = 3$, illustrating the recurrence relation for the worst-case tree height. The numbers in the top row are heights. The second row shows the required minimum number of nodes to reach these heights, given patient combining of degree 3. The rest of the numbers are tree heights at different points in the combining, time progressing down the figure. Polygons group the heights of two segments that are combined in the worst-case way, resulting in a segment of the height given directly below the rightmost height. Initially, all segments start at size 1, shown in the third to top row. As an example, a segment with height 8 is generated by combining segments with heights 7 and 5 in a queue that also contains a segment with height 1.

---

for a system with 64K nodes, logarithmic latency can be achieved with $X = 16$. However, we next show, through simulation, that the latency improvement for going from $X$ to $X + 1$ decreases rapidly with increasing $X$. This means that implementations with large $X$ are not cost effective.

### 3.4.3. Results

Figure 3.13 summarizes the simulator results and the worst-case analysis from above. Recall that the workload is $N$ simultaneous read-misses. The worst cases for greedy combining ($X = 2$) and patient combining ($X = 3$) are linear and square root respectively. Other lines in these semi-log graphs show that the average cases are logarithmic, which is one of the goals of these extensions.

With and without draining, patient combining for $X = 3$ is shown to be a clear winner over greedy combining. Not only is the worst case reduced from linear to square root, but the average

**Patient versus Greedy Combining**

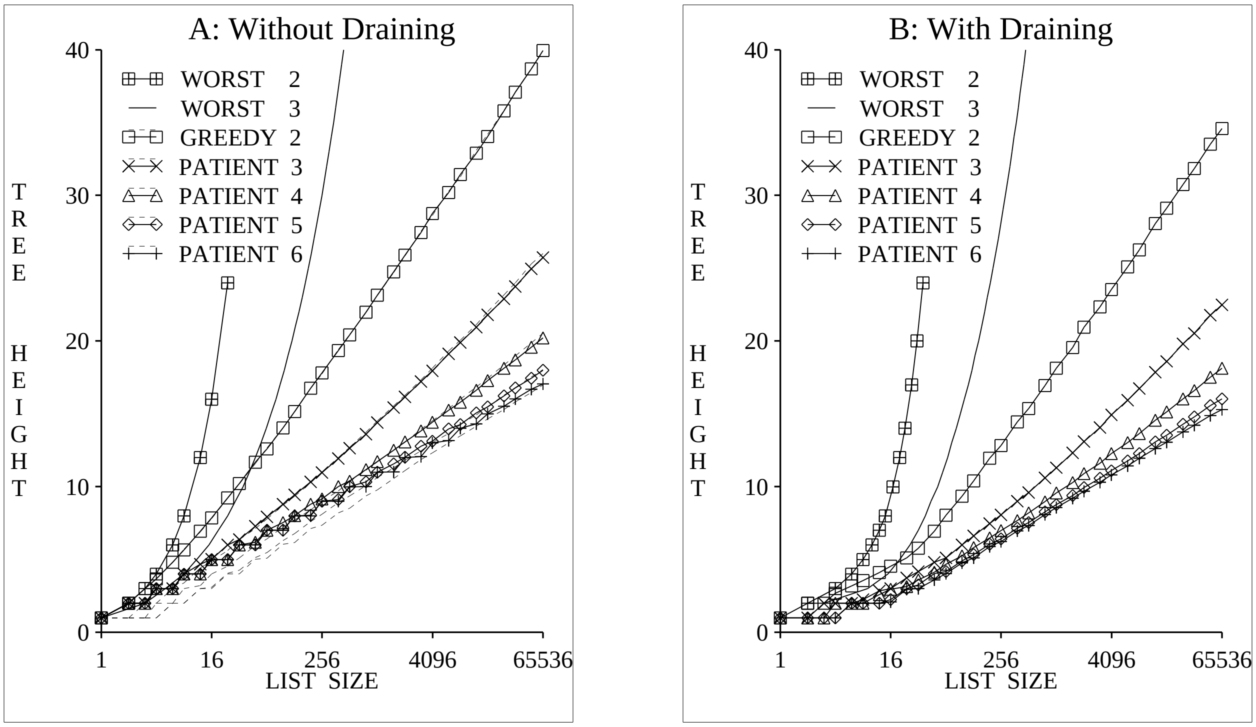



**Fig. 3.13:** These semi-log graphs summarize the simulator results (average case) and the worst-case analysis for the early-pointer tree height after combining. The left graph is for simulations without draining and the right is for ones with draining. The lines in the keys are ordered by their tree heights for 64K nodes. The number following each line name is the value of *X*, the implementation parameter described in the text. The GREEDY and PATIENT lines, acquired by simulation, are average cases for the given *X*. The WORST lines are the worst cases for the given *X*. These lines are derived analytically, except for the drain component in the right graph. The dotted lines are average cases where only one pair of segments is combined in a network queue, even though other pairs of segments with equal-height trees may exist.

Each data point for the simulation data is averaged from 1000 simulations. For all simulation data, the standard deviations for the 1000 average cases ranged from 1 to 6 percent for 64K nodes. The standard deviations in the right graph ranged from 3 to 6 percent for 64K nodes. The standard deviations for the dotted lines in the left graph ranged from 1 to 6 percent for 64K nodes. The standard deviations for the solid lines in the left graph ranged from 3 to 4 percent for 64K nodes.

---

case with draining is reduced by 35 percent for 64K nodes. Furthermore, even the worst case of patient combining is no worse than the average case of greedy combining for small numbers of nodes. However, increasing *X* past 3 quickly reaches a point of diminishing returns. Going to

$X = 4$ reduces the latency by another 19 percent, but going from $X = 5$ to $X = 6$ with draining reduces the tree height by only another 5 percent.

In the real world, it is reasonable to increase the size of a system incrementally, possibly resulting in a system where nodes have different values of $X$. In this case, the system will still work correctly and the performance of the system is at least as good as if all nodes were implemented with the smallest $X$ that is currently in the system.

In the graph without draining, the dotted lines (one combining per queue entry) versus the solid lines (multiple combinings) show that the combining of additional equal-height segments often increases the tree height of the resulting segments, especially for small list sizes. This is because the extra combining sometimes causes the combining pool to stop with less than $X - 1$ segments, resulting in larger segments on average. For small $X$, however, the difference is not significant.

One might think that the directory must service two or three times as many requests for patient combining because the number of noncombined requests is increased. This is not true, however. As the directory takes longer to service the first few noncombined requests, other requests are backed-up and *forced* to combine. The penalty for patient combining, in terms of requests serviced by the directory, resembles an additive constant rather than a multiplicative constant. The effects of tree saturation are reduced by forcing the combining when a queue is near full (defaulting to greedy combining). Another way to look at this is that ‘‘you get combining exactly when you need it,’’ as Jim Goodman likes to say. That is, when congestion is high and combining is needed, the probability is increased that multiple combinable requests will coexist in the same queue and combining will occur. When congestion is low, combining is improbable and unneeded.

It is also interesting to note that increasing the size of the network’s queues will increase, not decrease, the time of the average hot-spot access. This is because an increase in queue size reduces the number of saturated queues, thereby reducing the combining rate relative to the service rate of the directory. Because requests are combined slower, the directory must service more hot-spot requests. This implies that it takes longer to satisfy all of these requests and increases the time of the average hot-spot access. It might seem that the network queues should be as large as possible in order to avoid tree saturation. Although larger queues decrease the number of saturated queues, the above arguments imply that larger queues increase the time that a given queue is saturated. This result leads to the following three conclusions. First, there exists a crossover point for the optimal network queue size, based on the hot-spot traffic for a particular application. This is a topic for future work. Second, large network queues should be implemented so that multiple combinings can occur simultaneously. Third, it is difficult to specify and/or build hardware that reduces hot-spot latency. Software hot spots should be avoided whenever possible.

In summary, patient combining with $X = 3$ is best for use with recursive doubling because the performance of the worst and average cases are substantially improved. The protection from a sequential worst case, with a slight improvement for the average case, is what justifies the complexity of patient combining. However, since combining of additional equal-height segments makes the implementation more difficult and does not reduce latency, patient combining without this optimization is best. Extra request combining may help reduce traffic in some topologies, but this optimization has not been studied in this context.

### 3.5. Temporary Pointers

The object of recursive doubling is to efficiently create temporary pointers that will be useful for efficient distribution of data and invalidates. In this section, we consider two sets of temporary pointers, symmetric and binary, and look at a number of ways to enhance recursive doubling. Before discussing the two pointer sets, we review recursive doubling, detailed in Section 3.2.3.

Before recursive doubling, each cache line determines the pcount of the node to which it wants a temporary pointer, if any. This is a function $t$ of the system size $M$ and the cache line's own pcount $p$ such that temporary pointers make shortcuts across the list towards the tail and pointers are nested. Formally, $p \geq t_M(p)$ and if $q > p > t_M(q)$, then $t_M(p) > t_M(q)$. If no temporary pointer is desired, then $t_M(p) = p$.

Given this function $t$, recursive doubling is summarized as follows. Each node iteratively asks its temporary neighbor (initially its forward neighbor) for that neighbor's temporary (or neighbor) pointer and pcount. A node replaces the old temporary pointer with the new one at each iteration. A node stops iterating when the pcount for its current pointer is less than or equal to the pcount for the desired pointer. After the temporary pointers are created, the data and invalidates can be efficiently distributed.

### 3.5.1. Pointer Structures

There are at least five important issues for choosing which pointers to create, that is, for choosing the function $t_M$. First, it is important that the structure provide for low latency data and invalidate distribution. Second, the structure should still be efficient for invalidate distribution after a small number of nodes are removed from the list. Recall that temporary pointers are not updated during cache-line rollout. Third, the structure must be easy to generate. That is, recursive doubling must generate the structure fast and with minimal traffic. Fourth, each temporary pointer should be a simple (nonrecursive) function of the given node's pcount. Otherwise, the hardware implementation might be too expensive. Fifth, the structure should be efficient for small lists and it should be easy to incrementally add nodes. Next, we describe two possible structures that are reasonable; many different structures are possible, but we have found only two that seem reasonable.

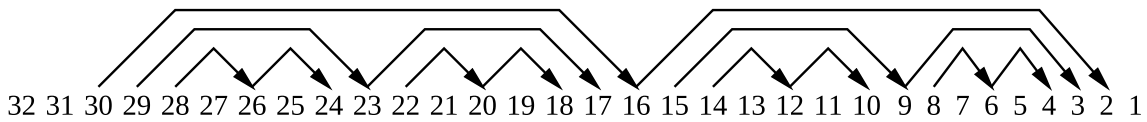
#### 3.5.1.1. Symmetric Structure

One candidate structure is the symmetric structure, conceived by James [Jame90, Gust91] and shown on the top in Figure 3.14. This structure was constructed so that each list segment is divided in half. The figure also shows how the distance (the pcount difference) of the temporary pointers is determined for the symmetric structure. The maximum list size is needed, in addition to the pcount, in order to determine the distance. Note that the entire structure for $M = 16$, not shown, is left shifted by one to make the right half of the structure for $M = 32$, where $M$ is the maximum number of nodes in the list. Given the distance, the pcount of the desired node can be found by subtraction from the current node's pcount.

#### 3.5.1.2. Binary Structure

Another candidate structure is the binary structure, shown in Figure 3.15. This structure is constructed so that the temporary pointers and neighbor pointers together form a binary tree. Actually, it forms a list of binary trees with the smallest trees nearer the tail of the list. The C code in

**Symmetric Structure**




```
int distance(int size, int pcount) {
/* Returns distance of temporary pointer, always power of two minus two
 * Assumes 1 <= pcount <= size == power of two
 */
  if (pcount <= 2 || size-1 <= pcount)          /* catch unusual cases      */
    return (0);
  while (pcount != size-2 && pcount != size/2 && pcount != 2) {
                                                /* stop when on boundary    */
    size = size / 2;                            /* split in half            */
    if (pcount > size)                          /* if in upper half         */
      pcount = pcount - (size - 1 );            /*   move to lower half     */
    else                                        /* else                     */
      pcount = pcount - 1;                      /*   keep in lower half     */
  }
  if (pcount == 2)                              /* if no pointer            */
    return (0);                                 /*   return zero            */
  return (size/2 - 2);                          /* else return distance     */
}
```

**Fig. 3.14:** This figure shows the symmetric structure for a maximum list size of $M = 32$ and gives the C code, for arbitrary $M$, that determines the distance spanned by a temporary pointer. Temporary pointers are shown by arrows and list pointers are implied by sequential node numbers.

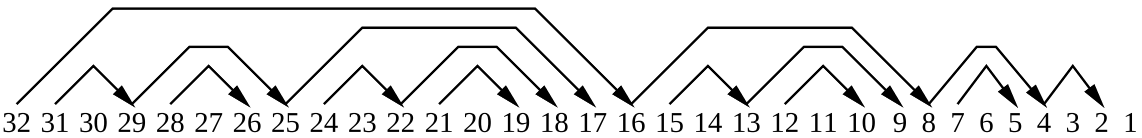
---

the figure shows how the distances (the pcount differences) of the temporary pointers are determined for the binary structure. For the binary structure, only the pcount of a node is needed to determine the distance of its temporary pointer. This means that the structure for $N = 16$ is the same as the right half of the structure for $N = 32$, where $N$ is the number of nodes in the list.

### 3.5.2. Optimization Techniques

Three techniques can be used to reduce the latency and traffic requirements associated with the candidate structures, two of which are specific to the binary structure. Request forwarding helps the latency associated with temporary-pointer creation for both structures, symmetric and binary. Pointer reversing helps the binary structure reduce the latency of invalidate distribution. Removal insurance helps the binary structure mitigate the performance problems related to stale temporary pointers. These three techniques are discussed next.

**Basic Binary Structure**




```
int distance(int pcount) {
  int two;                                   /* find least power of two  */
  two = power2(pcount);                      /*   such that two >= pcount*/
  if (pcount == two) {                       /* quit if barrier          */
    two = two / 2;
  } else {
    pcount = two + 1 - pcount;               /* find binary-tree size    */
    for (two /= 4; pcount != two && two > 1; two /= 2)
                                             /* stop on boundary         */
      if (pcount > two)                      /* if in upper half         */
        pcount = pcount - two + 1;           /*   move to lower half     */
  }
  two = (two == 1 ? 0 : two);
  return (two);
}
```

**Fig. 3.15:** This figure shows the basic binary structure and gives the C code for determining the distance spanned by a temporary pointer. Temporary pointers are shown by arrows and list pointers are implied by sequential node numbers.

---

### 3.5.2.1. Request Forwarding

During recursive doubling (and data distribution), one node iteratively requests temporary pointers from several nodes (and then requests the data). If the desired pcount is included in the request, then the responding node can determine whether or not the requester will make another request to the new temporary neighbor. If another request will be made, then the responding node should forward the request on behalf of the requester. Request forwarding helps reduce latency, overlapping the response with the next request.

Three cases must be considered when request forwarding is used during recursive doubling. First, if the forward neighbor is still doing recursive doubling, then the request pointer and the pcount of the desired temporary pointer are stored in the global space of the request controller. Also, some sort of state flag is required so that the request controller knows that it has something to forward. Second, if the forward neighbor has completed recursive doubling and there is an available request controller, then the new request controller is started and given the request pointer and the pcount. However, if there is no available request controller, the third case, then the node reverts to request reserving, described in Section 3.2.4.

#### 3.5.2.2. Pointer Reversing

The binary structure, as describe above, has the bad property that invalidate distribution requires $\Theta(\log^2(N))$ messages on the critical path. The problem is that invalidates propagate in the opposite direction from data and so the binary structure is not effectively used. One way to fix this problem is to make the temporary pointers bidirectional. That is, if $A$ has a temporary pointer to $B$, then $B$ has a temporary pointer to $A$. However, bidirectional pointers require more cache-tag space for the second pointer.

A better way of fixing the poor invalidate performance is by reversing the direction of some of the temporary pointers. This is accomplished during data distribution. The resulting pointers are shown in Figure 3.16. Notice the direction of the arrows. By changing the temporary pointers to point to the destinations of the data, rather than the sources, each selected node prepares to propagate the invalidates backward *instead* of forward. A node is selected to reverse its temporary pointer if and only if its temporary pointer is requested during recursive doubling. In other words, all nested temporary pointers are reversed as described.

Pointer reversing necessarily changes the protocol for invalidate distribution since invalidates no longer always propagate in the forward direction. If an invalidate arrives via the back pointer, then the protocol is the same: send the invalidate to the forward and temporary neighbors, if any. However, if the invalidate arrives via any other pointer, the protocol is different. If the invalidate arrives via the forward pointer, then send it to the backward and temporary neighbors. If it arrives via the temporary pointer, then compare against the pcount of the requester to determine if the invalidate came from a forward temporary neighbor or a backward temporary neighbor. If it came from a forward temporary neighbor, send it to the backward and temporary neighbors because the node's temporary pointer is reversed. If it came from a backward temporary neighbor, then send it to all neighbors, backward, forward, and temporary. Sending it to the forward neighbor is required for correctness in the event of stale temporary pointers and also as a special case for node 2.

**Binary Structure with Pointer Reversing**

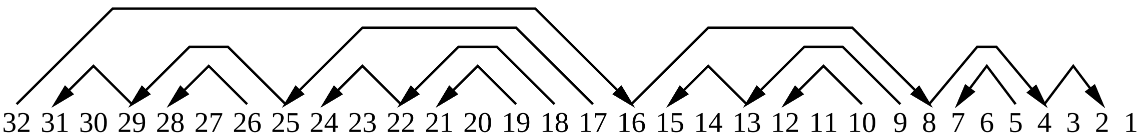



**Fig. 3.16:** This figure shows the binary structure after the nested temporary pointers have been reversed. Temporary pointers are shown by arrows and list pointers are implied by sequential node numbers.

---

To summarize the technique of pointer reversing, each node first generates its temporary pointer based on its pcount. Then, it requests the data from its temporary pointer, if any. When the data is received and forwarded to the next node and if the node's temporary pointer is nested, then the node remembers the destination of the forwarded data as its new temporary pointer. The resulting pointer structure can also be drawn as a tree, shown in Figure 3.17. This tree is later used for efficient invalidate distribution.

### 3.5.2.3. Removal Insurance

In order be insured against the worst-case performance after nodes are removed from the binary structure, additional temporary pointers can be added. The removal-insurance pointers are shown in Figure 3.18. Essentially, each insurance pointer is created immediately after creating a long temporary pointer (distance of at least 4). For example, after pcount $A$ creates a temporary pointer to pcount $B$ (and if $(A-B) \geq 4$), $A$ tells $A-1$ to keep a pointer to $B+1$.

The main advantage of removal insurance is that the latency for invalidate distribution degrades more gracefully as nodes are removed and temporary pointers become stale. The main disadvantage is that there is increased traffic. However, this additional traffic can be decreased to a negligible amount by using only the longer insurance pointers, perhaps only insurance pointers of distance 6 or greater.

## 3.6. Temporary-Pointer Performance

In this section, we compare the performance of the symmetric and binary structures and consider the performance improvement for request forwarding with each structure. Since the

**Binary Structure for Purging**

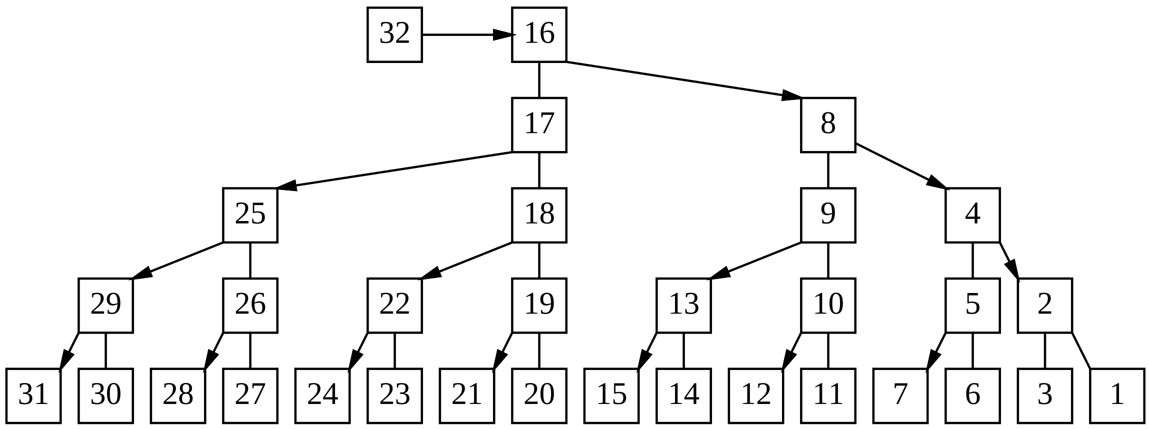



**Fig. 3.17:** This figure shows the binary structure after the nested temporary pointers have been reversed. Temporary pointers are shown by arrows and list pointers are shown explicitly by lines and implicitly by sequential node numbers.

---

performance of many protocols can be inflated by a particular choice of $N$, such as a power of two, we present the best, median, and worst cases for list sizes ranging between $\frac{N}{2}+1$ and $N$, where $N$ is a power of two. Since we are trying to give the performance for specific list sizes, rather than a range of list sizes, the median is a better measure than the average. However, for the given data it will be shown that the median is close to the average and, in many cases, exactly the average. Although not all of the performance data will be discussed here in detail, the generated data will be useful for comparisons to data in other chapters and in future work.

A number of scenarios are considered, both separately and collectively. Separately, we consider the latency and traffic associated with each of the different protocol phases. This includes directory access (with combining), pointer creation, data distribution, invalidate distribution, purging, fast purging, single cache-line rollouts, and multiple cache-line rollouts. Pcount distribution is considered in Section 3.4. Collectively we consider the latency and traffic associated with combinations of these phases and their interactions. For comparisons to lower bounds in other chapters, we consider *creation distribution,* the combination of pointer creation with data distribution and request reserving. For comparison to data in other chapters, we consider three combinations of reading and writing shared data. First, for reading sharing data, we

**Binary Structure with Full Removal Insurance**

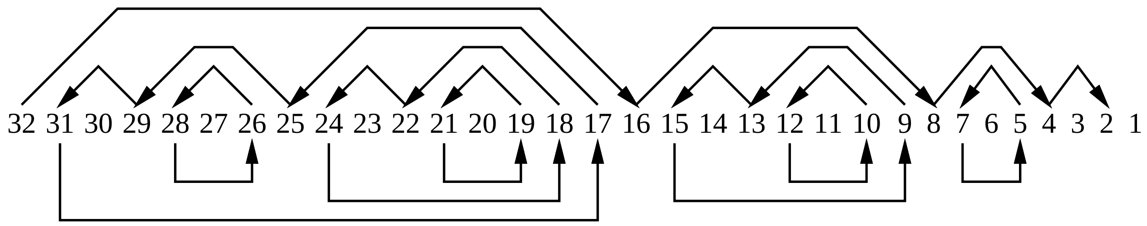



**Fig. 3.18:** This figure shows the binary structure after the nested temporary pointers have been reversed and insurance pointer have been added. Temporary pointers are shown by arrows and list pointers are implied by sequential node numbers.

---

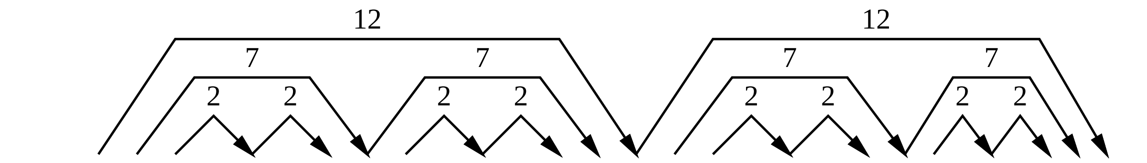
consider the combination of directory access, pcount distribution, and creation distribution. Second, for writing shared data, we consider the combination of single rollouts, directory access, invalidate distribution, and purging. Third, to give a total picture of one writer with many readers, we consider the combination of reading and writing.

Next, we derive the equations for the latency and traffic for the two sets of temporary pointers, with and without request forwarding. As stated earlier, we consider the best, median, and worst cases for list sizes ranging between $\frac{N}{2}+1$ and $N$, where $N$ is a power of two. We also show that the average performance is close to the median performance. We begin with the equations that are specific to the symmetric and binary structures, in that order, and then cover the equations that are common to both structures. The nonmathematical reader should skip to Section 3.6.4.

### 3.6.1. Symmetric-Structure Equations

The symmetric structure is shown again in Figure 3.19. The generating code is shown in Figure 3.14 on page 102. First, we will derive the latency equations for pointer creation *without* request forwarding, data distribution (with request reserving), and creation distribution *without* request forwarding. Second, we will derive the latency equations for pointer creation *with* request forwarding and creation distribution *with* request forwarding. Third, we will derive the latency equations for invalidate distribution. Fourth, we will consider the traffic for all of the above scenarios. In the derivations, we will make reference to Figure 3.19 and the next one as

**Symmetric Structure Latencies (without request forwarding)**




| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 5 | 4 | 3 | 5 | 7 | 7 | 6 | 6 | 5 | 4 | 6 | 6 | 5 | 5 | 4 | 3 | 2 | 4 | 6 | 6 | 5 | 5 | 4 | 3 | 5 | 5 | 4 | 4 | 3 | 2 | 1 | 0 |
| 14 | 13 | 12 | 13 | 15 | 15 | 14 | 14 | 13 | 12 | 14 | 14 | 13 | 13 | 12 | 11 | 10 | 9 | 10 | 10 | 9 | 9 | 8 | 7 | 6 | 6 | 5 | 5 | 4 | 3 | 2 | 0 |

**Fig. 3.19:** This figure shows the symmetric structure for a maximum list size of $M = 32$. Temporary pointers are shown by arrows and list pointers are implied by sequential node numbers in the third to last row, immediately below the pointers. A number above a pointer indicates the latency to create that pointer without request forwarding. A number in the second to last row, immediately below the node numbers, indicates the latency of data distribution to the given node. These numbers assume that the data starts at node 1, all temporary pointers are created, and all reservations are set. A number in the last row indicates the latency for creation distribution.

---

examples for $M = 32$, but we will give the equations for arbitrary $M$. We are using $M$ instead of $N$ to indicate that the equations assume that the number of nodes is the maximum. This assumption makes the analysis easier because the symmetric structure depends on the maximum list size. We will discuss this disadvantage later when we compare the performance data in Section 3.6.4.

#### 3.6.1.1. Equations: Symmetric Creation without Forwarding

First, consider the latency of pointer creation *without* request forwarding, shown by the numbers above the pointers in the figure. Creation of each of the least-distance pointers requires 2 time units for 2 messages, one request and one response. Assuming the least-distance pointers are already created, the next least-distance pointers, distance six, require 6 time units: 2 to get the first pointer, 2 to get the second pointer, and 2 to get the final pointer. However, the least-distance pointers are not available immediately and request reserving is used. This causes a wait of 1 time unit after the first request and before the least-distance pointers can be forwarded. So, creation of the next least-distance pointers actually requires 7 time units. Continuing with this analysis yields the general equation for the time to create the *greatest-distance* temporary pointer (the temporary pointer that spans the most nodes),

$$5\log_2(M) - 13,\ M \geq 8, \tag{3.3}$$

where $M$ is the (maximum) list size. Since the right half of the structure always has one of the greatest-distance temporary pointers, the given equation is the best, median, average, and worst cases.

Consider the latency of data distribution, assuming the pointers are already created and the requests for data have already been received and reserved. This latency is shown by the numbers in the second to last row of the figure. Since the data must make two hops per pointer nesting, the latency equations for data distribution are related to $2\log_2(N)$. In the worst case, the data arrives last at node $N - \log_2(N)$, yielding

$$2\log_2(M) - 3,\ M \geq 16. \tag{3.4}$$

The worst cases are nodes 27 and 28 in the figure. In the best case, only the right half of the structure is sent the data, yielding

$$2\log_2(M) - 4,\ M \geq 16. \tag{3.5}$$

The median is the same as the best case because one of the critical paths is always contained in the right half, except for the worst cases. The average asymptotically approaches the median case for the same reasons, always within 1 time unit of the median case.

Consider the latency of creation distribution without request forwarding. This latency is shown by the numbers in the last row of the figure. Sometimes a node receives the data while it is still generating its temporary pointer. For example, node 16 in the figure receives the data from node 9 while trying to generate a temporary pointer to node 2. This is its second new pointer during recursive doubling. For list sizes of at least 64, the middle node receives the data with its first new pointer during recursive doubling. Therefore, the latency to deliver the data to the middle node is 4 less than the time to create the greatest-distance pointer, $5\log_2(M) - 13$ from Equation 3.3. This gives

$$5\log_2(M) - 17,\ M \geq 64. \tag{3.6}$$

Then, add this to the worst-case data-distribution latency for the second half, which is 2 less than the total of $2\log_2(M) - 3$ from Equation 3.4. This gives

$$7\log_2(M) - 22,\ M \geq 64, \tag{3.7}$$

the worst-case latency for creation distribution without request forwarding. The critical path for the best case is in the right half. When $M$ is at least 128, the middle node of the right half also receives the data with its first new pointer. So, by the same analysis, the best-case latency for creation data distribution without request forwarding is

$$7\log_2(M) - 29,\ M \geq 128. \tag{3.8}$$

Consider the median latency for creation distribution without forwarding. The latency to the right quarter of the left half is the latency to the middle node, Equation 3.6, plus the latency to distribute the data through that quarter, which is 3 less than $2\log_2(M) - 3$ from Equation 3.4. Since this sum is greater than the best case, the median latency is

$$7\log_2(M) - 23,\ M \geq 64. \tag{3.9}$$

The average is close to the median because the best and worst cases are not far apart and the frequency of values is skewed towards the larger values.

#### 3.6.1.2. Equations: Symmetric Creation with Forwarding

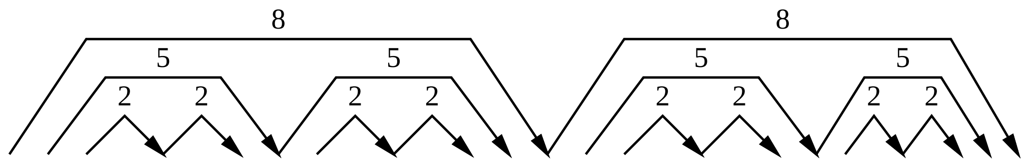
Second, consider the latency of pointer creation *with* request forwarding, shown by the numbers above the pointers in the next figure, Figure 3.20. After the least-distance pointers are created in 2 messages, the requests are forwarded two hops before being sent to the original requester, making a latency of 5 messages. Continuing with this analysis yields a latency of

$$3\log_2(M) - 7,\ M \geq 8 \tag{3.10}$$

to create the greatest-distance temporary pointers. Since one of the two greatest-distance pointers is located in the right half of the structure, this is the best, median, average and worst cases.

Consider the latency of creation distribution with request forwarding, shown by the second to last row in the figure. The middle node, for $M \geq 16$, always gets the data with its final

**Symmetric Structure Latencies (with request forwarding)**




| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| 5 | 4 | 3 | 5 | 7 | 7 | 6 | 6 | 5 | 4 | 6 | 6 | 5 | 5 | 4 | 3 | 2 | 4 | 6 | 6 | 5 | 5 | 4 | 3 | 5 | 5 | 4 | 4 | 3 | 2 | 1 | 0 |
| 11 | 10 | 9 | 11 | 13 | 13 | 12 | 12 | 11 | 10 | 12 | 12 | 11 | 11 | 10 | 9 | 8 | 6 | 8 | 8 | 7 | 7 | 6 | 5 | 6 | 6 | 5 | 5 | 4 | 3 | 2 | 0 |
| 8 | 7 | 6 | 8 | 10 | 10 | 9 | 9 | 8 | 7 | 9 | 9 | 8 | 8 | 7 | 6 | 5 | 4 | 6 | 6 | 5 | 5 | 4 | 3 | 5 | 5 | 4 | 4 | 3 | 2 | 1 | 0 |

**Fig. 3.20:** This figure shows the symmetric structure for a maximum list size of $M = 32$. Temporary pointers are shown by arrows and list pointers are implied by sequential node numbers in the fourth to last row, immediately below the pointers. A number above a pointer indicates the latency to create that pointer with request forwarding. A number in the third to last row, immediately below the node numbers, indicates the latency of data distribution to the given node. These numbers assume that the data starts at node 1, all temporary pointers are created, and all reservations are set. This row of numbers is repeated from Figure 3.19. A number in the second to last row indicates the latency for creation distribution. A number in the last row indicates the latency for invalidate distribution to all lesser numbered nodes, starting at the given node.

---

temporary pointer in $3\log_2(N) - 7$ messages. Note that if valid data is discovered during the forwarding, it is not returned to the original requester until the forwarding is done. Data distribution then takes $2\log_2(N) - 5$ more messages to reach the rest of the nodes in the worst case, so the worst-case latency for creation distribution with request forwarding is

$$5\log_2(M) - 12,\ M \geq 16. \tag{3.11}$$

The best case is

$$5\log_2(M) - 17,\ M \geq 32 \tag{3.12}$$

because the critical path for the best case is in the right half. The median case is

$$5\log_2(M) - 13,\ M \geq 16 \tag{3.13}$$

because this is the worst-case latency to the right quarter of the left half. The average is close to the given median because the best and worst cases are not far apart and the frequency of values is skewed towards the larger values.

### 3.6.1.3. Equations: Symmetric Invalidate Distribution

Third, consider the latency for invalidate distribution (with request forwarding), shown by the numbers in the last row of Figure 3.20. The best case for invalidate distribution is

$$2\log_2(M)-4,\ M\geq 16. \tag{3.14}$$

This is the same as Equation 3.5, due to symmetry, as shown by the right halves of the third to last and last rows in the figure. The worst case is

$$4\log_2(M)-10,\ M\geq 16 \tag{3.15}$$

because the greatest-distance pointers, see nodes 30 and 29 in the figure, do not exist for incomplete structures. For example, a 27-node structure from the figure acts like a balanced tree with node 16 as the root and node 27 as a leaf.

Consider the median latency for invalidate distribution. The latencies for nodes 30 through 17 in the figure range from 10 to 6. The frequencies are 2, 4, 4, 2, and 2 respectively. For $M=16$, the frequencies are 2, 2, and 2. For $M=64$, the frequencies are 2, 6, 8, 6, 4, 2, and 2. Notice the pattern, shown in Figure 3.21 for $16\leq M\leq 256$. Adding the sequence for $M$ to itself after shifting one of the copies yields the sequence for $2M$, except for the trailing 2 that must be appended. To find the median frequency, remove the leftmost $x$ elements from the sequence such that sum of these elements is maximized, but not greater than $\frac{M}{4}$. The leftmost remaining one is shown in bold. Then, the number of remaining frequencies is the median latency for an invalidate to reach the middle node of the structure. From the pattern in the figure we deduce that this function is $\lceil 1.5\log_2(M)\rceil-5$ and have checked this by calculation for $16\leq M\leq 65536$. Adding this to the latency of the middle node, which is one less than the best case given in Equation 3.14, yields

$$\lceil 3.5\log_2(M)\rceil-10,\ M\geq 16. \tag{3.16}$$

The median is closer to the worst case than the best case because the frequency of values is skewed towards the larger values.

**Invalidate-Latency Frequency Sequences**

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $M=16$ | | | | | | 2 | 2 | **2** | | | | |
| | | | | | 2 | 2 | 2 | | | | | |
| $M=32$ | | | | | 2 | 4 | **4** | 2 | 2 | | | |
| | | | | 2 | 4 | 4 | 2 | 2 | | | | |
| $M=64$ | | | | 2 | 6 | 8 | **6** | 4 | 2 | 2 | | |
| | | | 2 | 6 | 8 | 6 | 4 | 2 | 2 | | | |
| $M=128$ | | | 2 | 8 | 14 | **14** | 10 | 6 | 4 | 2 | 2 | |
| | | 2 | 8 | 14 | 14 | 10 | 6 | 4 | 2 | 2 | | |
| $M=256$ | | 2 | 10 | 22 | 28 | **24** | 16 | 10 | 6 | 4 | 2 | 2 |
| | 2 | 10 | 22 | 28 | 24 | 16 | 10 | 6 | 4 | 2 | 2 | |

**Fig. 3.21:** This figure shows the pattern associated with the frequencies for the latencies of invalidate distribution. Each pair of sequences represents $M=16$ to $M=256$, from top to bottom. The $i^{\text{th}}$ number from the right represents the frequency, number of nodes between $M-2$ and $\frac{M}{2}-1$, that can send an invalidate to the middle node with latency $i$. The number in bold represents what would be the median frequency if latencies for nodes $M$ and $M-1$ were included in the frequencies. Notice that element $f_i$ for sequence $M$ is $f_{i-1}+f_{i-2}$ for sequence $\frac{M}{2}$, except for element $f_1$, the rightmost element of sequence $M$.

---

The average asymptotically approaches

$$3.5\log_2(M)-10,\ M\geq 16 \tag{3.17}$$

for the same reasons, always within 1 time unit of the median case. The hard part in deriving the average latency is discovering and solving the recurrence relation $R(x)=2\cdot R(x-1)+3{\cdot}2^{x-2}-4$, where $x$ is $\log_2(M)$ and $R(x)$ is the sum of the latencies, from node $M-2$ to $\frac{M}{2}-1$, to reach the middle node.

### 3.6.1.4. Equations: Symmetric Traffic

Fourth, consider the traffic associated with the symmetric structure. Without request forwarding, the required number of messages to create the symmetric structure is a total of about $2.5M$ messages, one request and one response for each of the $M/4$ least-distance temporary pointers and 8 messages for each of the remaining $M/4-2$ temporary pointers. Although the latency is reduced by request forwarding, the traffic with request forwarding is the same as without request forwarding. Afterwards, data distribution requires up to $4M$ total messages, two for every reservation and two to send the data. Since it will be shown later in Section 3.6.4 that the latency results are more interesting than the traffic results, we will not consider the small traffic reduction for nodes that receive the data during recursive doubling or request reserving. Finally, invalidate distribution requires about $3M$ total messages, 2 messages for each of the $\frac{3M}{2} - 2$ forward pointers, both list and temporary pointers.

### 3.6.2. Binary-Structure Equations

The binary structure is shown again in Figure 3.22. The generating code is shown in Figure 3.15 on page 103. First, we will derive the latency equations for pointer creation *without* request forwarding, data distribution (with request reserving), and creation distribution *without* request forwarding. Second, we will derive the latency equations for pointer creation *with* request forwarding and creation distribution *with* request forwarding. Third, we will derive the latency equations for invalidate distribution. Fourth, we will consider the traffic for all of the above scenarios. In the derivations, we will make reference to this figure and the next one as examples of $N = 32$, but we will give the equations for arbitrary $N$. Note that we use the variable $N$ because the binary structure is not dependent on the maximum list size.

#### 3.6.2.1. Equations: Binary Creation without Forwarding

Consider the latency of pointer creation without request forwarding, as shown by the numbers above the pointers in Figure 3.22. Using the same derivation method as for the symmetric structure, the worst-case latency to create the temporary pointers is

$$3\log_2(N) - 4,\ N \geq 4. \tag{3.18}$$

Since the largest pointer is only created when all $N$ nodes are present, the best and median cases have latency of

$$3\log_2(N) - 7,\ N \geq 8. \tag{3.19}$$

The average case approaches the median for large $N$, always within 1 for $N \geq 8$.

Consider the latency of data distribution, assuming the pointers are already created and the requests for data have already been received and reserved. This latency is shown by the

**Binary Structure Latencies (without request forwarding)**

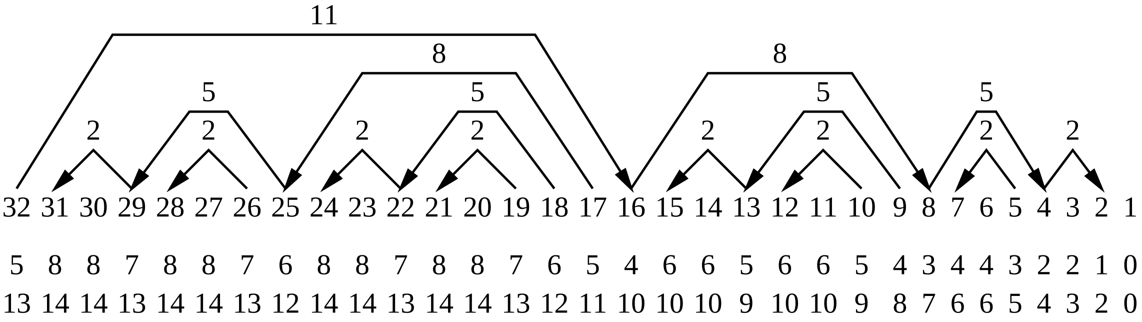



**Fig. 3.22:** This figure shows the binary structure. Temporary pointers are shown by arrows and list pointers are implied by sequential node numbers in the third to last row, immediately below the pointers. A number above a pointer indicates the latency to create that pointer without request forwarding. A number in the second to last row, immediately below the node numbers, indicates the latency of data distribution to the given node. These numbers assume that the data starts at node 1, all temporary pointers are created, and all reservations are set. A number in the last row indicates the latency for creation distribution.

---

numbers in the second to last row of the figure. The best-case latency is

$$2\log_2(N) - 4,\ N \geq 16. \tag{3.20}$$

The worst-case latency is

$$2\log_2(N) - 2,\ N \geq 4. \tag{3.21}$$

The worst case is also the median case because one of the worst cases is for $N/2 + \log_2(N) - 1$ nodes, which is in the right quarter of the left half. The average-case latency for data distribution approaches the median case for large $N$, always within 1 for $N \geq 4$.

Consider the latency of creation distribution without request forwarding. This latency is shown by the numbers in the last row of the figure. For $N \geq 8$, the middle node of the structure never receives the data until immediately after the node asks for the data. Therefore, the worst-case for creation distribution is the best case for pointer creation, given in Equation 3.20, plus the latency to distribute the data through the right half. So, the worst-case latency is

$$4\log_2(N) - 6,\ N \geq 16. \tag{3.22}$$

The critical path for the best case is in the right half of the structure for $N \geq 64$, so the best-case latency is

$$4\log_2(N) - 10,\ N \geq 64. \tag{3.23}$$

The median latency is

$$4\log_2(N) - 6,\ N \geq 8. \tag{3.24}$$

This is the latency to reach the right quarter of the left half of the structure. The average latency approaches the median for large $N$, always within 1 for $N \geq 8$.

#### 3.6.2.2. Equations: Binary Creation with Forwarding

Second, consider the latency of pointer creation *with* request forwarding. This is shown by the numbers above the pointers in the next figure, Figure 3.23. The worst-case latency is

$$2\log_2(N) - 2,\ N \geq 2. \tag{3.25}$$

However, the worst case only occurs when all $N$ nodes are present, so the best case is

$$2\log_2(N) - 4,\ N \geq 4. \tag{3.26}$$

The median is the same as the best case and the average case approaches the median for large $N$, within 1 for $N \geq 4$.

Consider the latency of creation distribution with request forwarding, shown by the second to last row in the figure. The best-case latency is

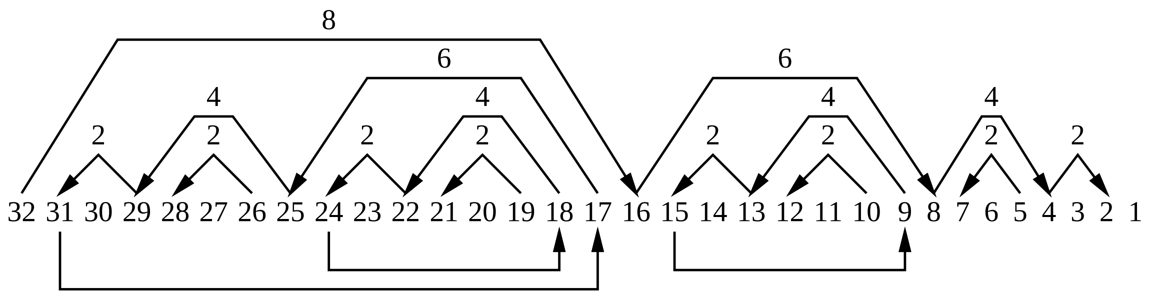
$$3\log_2(N) - 6,\ N \geq 32 \tag{3.27}$$

to reach all nodes in the right half. The worst-case latency is

$$3\log_2(N) - 4,\ N \geq 8 \tag{3.28}$$

to reach node $\frac{N}{2} + \log_2(N) - 1$. The median is the same as the worst case and the average case approaches the median for large $N$, within 1 for $N \geq 8$.

**Binary Structure Latencies (with request forwarding)**




| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 8 | 8 | 7 | 8 | 8 | 7 | 6 | 8 | 8 | 7 | 8 | 8 | 7 | 6 | 5 | 4 | 6 | 6 | 5 | 6 | 6 | 5 | 4 | 3 | 4 | 4 | 3 | 2 | 2 | 1 | 0 |
| 9 | 11 | 11 | 10 | 11 | 11 | 10 | 9 | 11 | 11 | 10 | 11 | 11 | 10 | 9 | 8 | 7 | 8 | 8 | 7 | 8 | 8 | 7 | 6 | 5 | 5 | 5 | 4 | 3 | 3 | 2 | 0 |
| 5 | 8 | 8 | 7 | 8 | 8 | 7 | 6 | 8 | 8 | 7 | 8 | 8 | 7 | 6 | 5 | 4 | 6 | 6 | 5 | 6 | 6 | 5 | 4 | 3 | 4 | 4 | 3 | 2 | 2 | 1 | 0 |

**Fig. 3.23:** This figure shows the binary structure. Temporary pointers are shown by arrows and list pointers are implied by sequential node numbers in the fourth to last row, immediately below the top pointers. A number above a top pointer indicates the latency to create that pointer with request forwarding. The bottom pointers are the insurance pointers described in Section 3.5.2.3. These are used during invalidate distribution to mitigate the performance problems associated with stale temporary pointers. A number in the third to last row, immediately below the insurance pointers, indicates the latency of data distribution to the given node. These numbers assume that the data starts at node 1, all temporary pointers are created, and all reservations are set. This row of numbers is repeated from Figure 3.22. A number in the second to last row indicates the latency for creation distribution. A number in the last row indicates the latency for invalidate distribution to all lesser numbered nodes, starting at the given node.

---

### 3.6.2.3. Equations: Binary Invalidate Distribution

Third, consider the latency for invalidate distribution (with request forwarding), shown by the numbers in the last row of the figure. Since this row is the same as the third to last row, the worst-case latency is

$$2\log_2(N) - 2,\ N \geq 4, \tag{3.29}$$

from Equation 3.21. The best-case latency is

$$\log_2(N),\ N \geq 1, \tag{3.30}$$

starting from node $\frac{N}{2}+1$. The median case is

$$2\log_2(N)-3,\ N\geq 32 \tag{3.31}$$

and the average case approaches the median for large $N$, within 1 for $N \geq 32$.

#### 3.6.2.4. Equations: Binary Traffic

Fourth, consider the traffic associated with the binary structure. Without request forwarding, the required number of messages to create the binary structure is 2 messages for each of the $\frac{N}{4}$ least-distance temporary pointers, plus 6 messages for each of the $\frac{N}{8}$ next least-distance temporary pointer, plus 10 messages for each of the $\frac{N}{16}$ next least-distance temporary pointers, etc, giving

$$\sum_{i=2}^{\log_2(N)} (4\,i-6)\,\frac{N}{2^i} \;=\; 3\,N-2\log_2(N)+2 \tag{3.32}$$

or about $3\,N$ messages. Afterwards, data distribution usually requires $4\,N$ total messages, 2 for every reservation and 2 to send the data. However, node 2 and each of the $\log_2(N)-1$ unnested pointers do not need request reserving, so the data-distribution traffic is

$$4\,N-2\log_2(N). \tag{3.33}$$

The traffic with request forwarding is slightly less because pointers of distance eight are generated with fewer messages: 2 messages for each of the $\frac{N}{4}$ least-distance temporary pointers, plus 6 messages for each of the $\frac{N}{8}$ next least-distance temporary pointer, plus 8 messages for each of the $\frac{N}{16}$ next least-distance temporary pointers, etc, giving

$$2\,\frac{N}{4}+\sum_{i=3}^{\log_2(N)} 2\,i\,\frac{N}{2^i} \;=\; 2.5\,N-2\log_2(N)-2, \tag{3.34}$$

or about $2.5\,N$ messages. Afterwards, data distribution requires up to $4\,N$ total messages, 2 for every reservation and 2 to send the data. Since it will be shown later in Section 3.6.4 that the latency results are more interesting than the traffic results, we will not consider the small traffic reduction for nodes that receive the data during recursive doubling or request reserving.

Finally, invalidate distribution requires about $3.75N$ total messages. There are 2 messages for each of the $N-2$ backward list pointers and 2 messages for each of the $\frac{N}{2}-1$ noninsurance temporary pointers. Each of the $\frac{N}{8}-1$ insurance pointers adds 2 messages after recursive doubling to create the insurance pointer and each adds 4 more messages during invalidate distribution because messages received via forward temporary pointers are also forwarded to the forward neighbors. Each of the $\log_2(N)-1$ nonreversed temporary pointers also adds 2 messages for the forward pointers. Therefore, the total number of messages for invalidate distribution is

$$\frac{15N}{4}+2\log_2(N)-14. \tag{3.35}$$

### 3.6.3. Common Equations

The binary and symmetric structures have the same performance for directory access, pcount distribution, cache-line rollout, and post-invalidate purging. First, consider the directory access. Independent of the amount of combining, which occurs exactly when needed, the latency and traffic are both 2 messages. These are the best, median, average, and worst cases.

Second, the traffic for pcount distribution depends on the amount of combining. We assume that all requests combine into a small constant number of requests. This assumption is reasonable for traffic comparisons in this chapter because all protocol variations experience the same traffic for pcount distribution and because the comparisons are made with sharing sets of size 65536. Therefore, the traffic is about 4 messages per node: 2 messages (request and response) to request the pcount (and set the reservation) and 2 more to forward the pcount when it arrives.

Third, consider cache-line rollout for a single rollout and for multiple rollouts. The latency and traffic of a single rollout is 4 messages, one request and one response to connect the forward neighbor to the backward neighbor and two more messages to connect the backward neighbor to the forward neighbor. These are the best, median, average, and worst cases. Note that these two transactions can not be overlapped since the forward neighbor has priority and may rollout first, as discussed in Section 3.2.7. If all nodes in the list try to rollout simultaneously, the nodes are removed sequentially and the head of the list must wait $\Theta(N)$ messages. This is the best, median, average, and worst case. The total traffic is $\Theta(N^2)$ and the average traffic per node is $\Theta(N)$.

Fourth, consider the purging latency after the invalidates have been distributed. This purging is shown in Figure 3.24, repeated from Figure 3.6 on page 81. Recall that each node can predict when to stop purging its forward neighbor, based on the rcounts, so no messages are wasted. Using the request-response paradigm for purging, each time step shown in the figure requires 2 messages. So the latency is

$$2\log_2(N),\ N \geq 1 \tag{3.36}$$

and the average traffic per node is

$$2 - \frac{2}{N}. \tag{3.37}$$

These are the best, median, average, and worst cases. When using fast purging (request forwarding), each time step shown requires 1 message. So, the latency is

$$\log_2(N) + 1,\ N \geq 2, \tag{3.38}$$

which is the best, median, average, and worst case. However, the traffic per node is about 2 messages since each forwarded request still requires a response.

### 3.6.4. Performance Comparisons

In order to compare the given pointer structures, we consider the scenario of one producer and many consumers. The performance considered is the latency and traffic required for one writer to become the head and invalidate all $N-1$ readers and then for the $N-1$ readers to acquire the new data. We also consider reading and writing separately. The binary structure is shown to have significantly better latency than the symmetric structure, but the traffic for both is about the same.

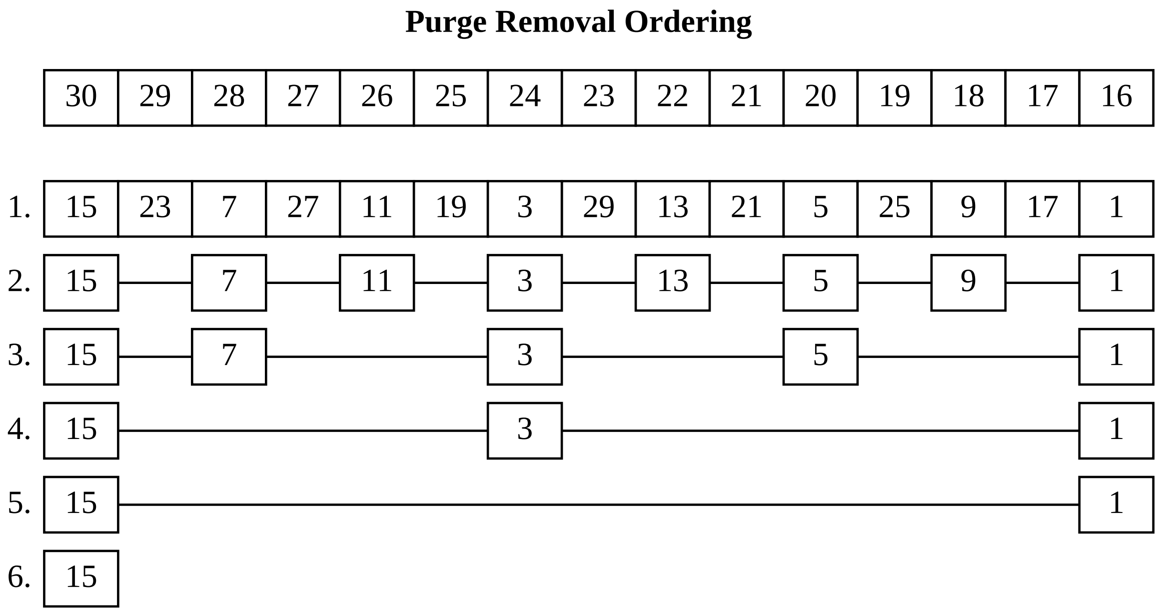



**Fig. 3.24:** This figure shows the purging of a list after all nodes have received an invalidate. All numbers are rcounts, except for the pcounts on the top row. Node 30, with rcount 15, is the purge initiator. Lines represent list connectivity, some connectivity being represented by touching nodes. Time progresses down the figure with time steps shown on the left. The absence of a node indicates that it has been purged.

---

To approximate the latency for writes, we add the latencies for a single rollout, directory access, 2 messages, invalidate distribution for the given pointer structure, and purging. It is reasonable to include the rollout and directory-access latencies for a write because the most recent writer is never the head of the sharing structure, unless there are no readers. It is also reasonable to simply add the invalidate-distribution latency to the purging latency. This is a reasonable approximation (plus or minus a small constant) because the purging tree is balanced. That is, the purge distances are the same throughout the right half of the structure. Therefore, the critical path of invalidate distribution, which is in the rightmost half, will be the first part of the critical path for writing. Finally, we add the 2 extra messages because the writer must send the invalidate to the previous list head and receive the completion message.

To approximate the latency for reads, we add the latency for directory access, creation distribution, and pcount distribution, which is given by simulation in Section 3.4. We do not add rollouts because in our producer-consumer scenario we can assume that read misses are caused by invalidates. The latency overlap is considered for pointer creation and data distribution, which are both included in creation distribution. However, we do not consider this overlap for pcount distribution and creation distribution. This is a reasonable approximation because it is highly probable that there will be several leaves on the critical path for pcount distribution (for a partially balanced tree) and many of these leaves will be nodes that create no temporary pointers. Therefore, during recursive doubling, many nodes will have to wait for the pcount leaves, losing most of the time gained in the overlap.

The bar graph in Figure 3.25 shows the traffic for the symmetric and binary structures, with and without request forwarding. The equations for traffic to not take into account the overlap of phases. For example, sometimes a purge request to a forward neighbor is sent in the same message as an invalidate request, thereby reducing the total message count. However, this inaccuracy should similarly affect the binary and symmetric structures, so the given comparison is still reasonable. Since there is no traffic difference between using request forwarding and not using request forwarding in large symmetric structures, only one bar is given for the symmetric structure. The bar graph shows that there is no significant traffic difference between the symmetric and binary structures when a sharing list is large. It also shows that there is no significant traffic difference for using or not using request forwarding when a sharing list is large. For these reasons, our results for latency will determine which structure is better and whether or not request forwarding helps performance. The structure that wins will be the one that can best overlap the delivery of messages.

Note that the standard SCI cache-coherence protocol generates a tremendous amount of traffic per node for long lists. This is because nodes must spin-wait for the data; no reservations are used. Since the latency of data distribution is $\Theta(N)$ in a single list, the average traffic is also $\Theta(N)$ per node. This makes the traffic of both structures, which each have $\Theta(1)$ average traffic per node, tremendously superior to the simple list of SCI.

The median-latency equations for the two structures, with and without request forwarding, are summarized in Figure 3.26. Recall that the median and average are about the same, as shown in Sections 3.6.1 and 3.6.2. Clearly, request forwarding is useful for invalidate distribution in order to give logarithmic latency. Otherwise, it can be seen from the table that request

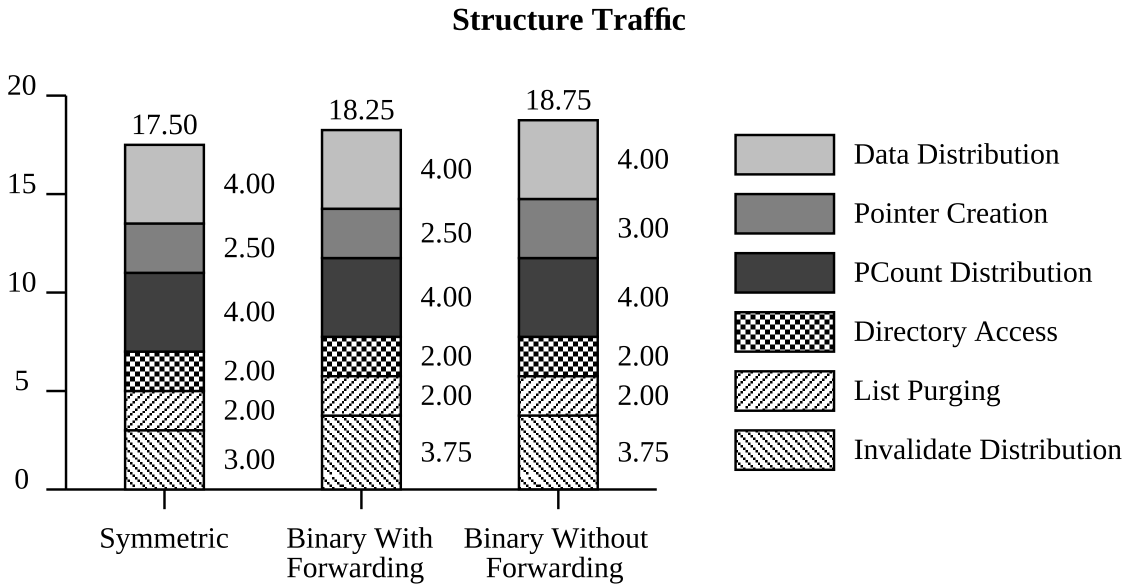



**Fig. 3.25:** This bar graph shows the average traffic per node, accumulated over several protocol phases. The given numbers are approached for large sharing lists. The number for each phase is given to the right of the corresponding bar component and the sum of the components is given above the bar. Each phase is coded by different shading and labeled to the right of the graph. Both the symmetric structure and the binary structure are represented. However, with respect to request forwarding, the traffic significantly differs only for the binary structure. Traffic comparisons to other protocols are given in Sections 6.3.2 and 6.4.2.

---

forwarding reduces the latency of purging by about 50 percent for both structures and it reduces the latency of pointer creation by about 40 and 33 percent for the symmetric and binary structures, respectively. The table also shows that, for pointer creation without request forwarding, the latency for the binary structure is about 40 percent less than the latency for the symmetric structure. However, the improvement is reduced to about 33 percent when request forwarding is used. This is because, during recursive doubling, any given node must visit more nodes, on average, in the symmetric structure than in the binary structure. The latency for creating the greatest-distance pointers is better for the symmetric structure, but latencies for creating the three least-distance pointers are equal or worse for the symmetric structure than for the binary structure.

The median latencies for reads, writes, and their sum are shown in Figure 3.27. Since the median latency is taken from the range $\frac{N}{2}+1 \leq \textit{list size} \leq N$, each data point is plotted at

**Median-Latency Table**

| Protocol | Without Request Forwarding | | With Request Forwarding | |
|---|---|---|---|---|
| Phase | Symmetric | Binary | Symmetric | Binary |
| Pointer Create | $5\log_2(M)-13$ | $3\log_2(N)-7$ | $3\log_2(M)-7$ | $2\log_2(N)-4$ |
| Data Distrib | $2\log_2(M)-4$ | $2\log_2(N)-2$ | $2\log_2(M)-4$ | $2\log_2(N)-2$ |
| Create and Data | $7\log_2(M)-23$ | $4\log_2(N)-6$ | $5\log_2(M)-13$ | $3\log_2(N)-4$ |
| Inval Distrib | $\Theta(M)$ | $\Theta(N)$ | $\lceil 3.5\log_2(M)-10 \rceil$ | $2\log_2(N)-3$ |
| Purging | $2\log_2(M)$ | $2\log_2(N)$ | $\log_2(M)+1$ | $\log_2(N)+1$ |
| Single Rollout | 4 | 4 | 4 | 4 |
| Multi-Rollouts | $\Theta(N)$ | $\Theta(N)$ | $\Theta(N)$ | $\Theta(N)$ |

**Fig. 3.26:** This table summaries the median latencies for the symmetric and binary structures. $N$ is the number of nodes in the sharing list. $M$ is the number of nodes in the sharing list and the number of nodes in the system, the maximum list size. Both $N$ and $M$ are powers of two.

---

*list size* $= \frac{3N}{4}$. Concerning the benefit of request forwarding, the latency of both structures is significantly helped for both reads and writes. The total latency for both structures is reduced by about 20 percent for 64K nodes and about 14 percent for 64 nodes. Although the symmetric structure is helped equally or more for the individual phases, as stated earlier, the binary structure is helped about equally for total latency. This is because the latency of purging is helped the most and purging is a greater percentage of the total latency for the binary structure.

Figure 3.26 and Figure 3.27 (table and graphs) clearly show that the binary structure has lower latency than the symmetric structure. Without request forwarding, the binary-structure latency with 64K nodes is about 28 percent better for reads, about 20 percent better for writes (always using request forwarding for invalidate distribution), and about 24 percent better for the sum. With request forwarding, the binary-structure latencies for 64K nodes are about 24 percent better for all three measures. It is also interesting to note that if the list size is larger than 256

**Median-Latency Graphs**

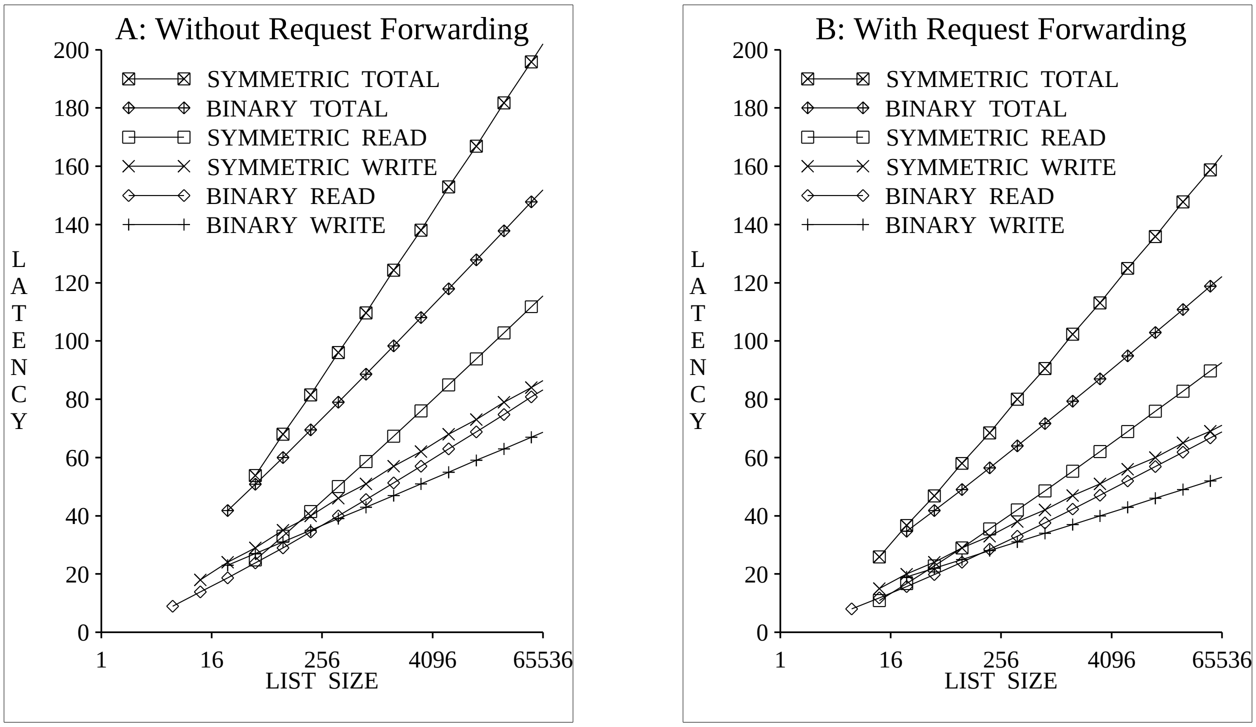



**Fig. 3.27:** These semi-log graphs show the median latency for reads, writes, and their sum. Both the symmetric and binary structures are fully represented. The key orders the latencies with respect to list sizes of 64K nodes.

---

nodes, then the binary structure *without* request forwarding does better than the symmetric structure *with* request forwarding.

Furthermore, the performance of the symmetric structure is based on a list size that is maximal, the size of the system. When the list size is small compared to the system size, the symmetric structure will *not* get the performance indicated by the data in this section. This is because many of the pointers that reach to the rightmost $\log_2(M) - 2$ nodes may not be created. For this reason, the latencies of data and invalidate distribution may each be increased by another $\log_2(M) - 2$ messages in the worst case. For example, no symmetric structure of size $N \le \log_2(M)$ will have any temporary pointers. This increase is unacceptable because the common cases are short lists, not long ones. On the other hand, the performance of the binary structure is independent of the system size and temporary pointers are created with lists as small as 4 nodes.

Overall, the binary structure with request forwarding gives the best performance. The binary structure is best for use with recursive doubling because the binary structure gives better performance than the symmetric structure, especially for short lists (the common case). Request reserving is useful to prevent spin-waiting and the wasting of network bandwidth.

Request forwarding is necessary for good performance of invalidate distribution. However, the choice of whether or not to use request forwarding for other operations is a more difficult matter. This is because the choice is sensitive to the frequency of long lists in real programs. If lists are frequently large, request forwarding yields significant performance improvements, as much as 20 percent. However, if lists are not frequently large, the improvements are small or negligible. Since we believe that small lists are much more common in real programs, such as for private data and pairwise sharing, and since request forwarding does not reduce the order of the worst or median cases, request forwarding is not useful in general.

### 3.7. Summary

SCI [Comm91, Gust92a] includes a cache-coherence protocol for shared-memory multiprocessors. The cache-coherence protocol is a distributed-pointer scheme [TDLS90] where the directory maintains a pointer to the head of a doubly-linked list of cache lines. Due to the sequential nature of these sharing lists, SCI is not scalable for applications that have a high degree of sharing (long lists) and frequent writes to these lines. Therefore, extensions to the cache-coherence protocols of SCI are needed to provide for scalability (logarithmic latency) when the degree of sharing is high.

In October 1989, Guri Sohi [JLGS90, Gust91] proposed a new combining mechanism and the use of recursive doubling [Ston73, HiSt86], in order to obtain logarithmic latency for $N$ simultaneous reads to shared data. In this chapter, we show how Sohi's proposal can be modified to 1) reduce the total number of messages from $\Theta(N \log(N))$ to $\Theta(N)$ and 2) leave behind a cache-coherence structure that can be used for writes with latency that is logarithmic in the number of previous readers. We call these extensions the recursive-doubling extensions to

SCI. Our total solution has seven overlapping phases: list creation, pcount distribution, pointer creation, data distribution, cache-line rollout, invalidate distribution, and purging.

For three reasons, it is not trivial to determine a good structure to replace the SCI sharing list. First, the distribution of information (data and invalidates) begins from opposite sides of the structure and both latencies must be minimized. Second, the latency to create the structure must be minimized. Third, the complexity of the structure's hardware implementation must be minimized. In this chapter, we present two simple candidate structures, the symmetric and binary structures, and show that the latency (number of messages on the critical path) for the binary structure is superior. Furthermore, the binary structure is shown to be better for small degrees of sharing, which are the more common cases. We also show that the traffic (the total number of messages) for both structures is about the same.

There are several places in the protocol of the SCI standard where one node must spin wait for information and this spinning generates a significant number of unnecessary messages, thereby congesting the network. In this chapter, we show how request reserving (asking for the information to be forwarded when available) can be used to avoid spin waiting in all cases in the SCI protocol and in these proposed extensions. We also show how the guarantee of forward progress, including deadlock avoidance, can be upheld during request reserving.

Request forwarding (the sending of a request on behalf of another node) can also be used to improve latency. In this chapter, we show how request forwarding can be used in a limited number of cases. In each case, we show the performance improvement and gave arguments to uphold the guarantee of forward progress. Note, however, that forward progress can not be guaranteed in all cases of request forwarding. Except for the use of request forwarding with invalidate distribution, the complexity of request forwarding is not worth the performance improvement of at most 20 percent.

Finally, the performance of pcount distribution depends on the early pointers that are created during combining. In this chapter, we show how patient combining (waiting for more congestion) can be used to generate better early pointers. The worst-case latency is shown to improve from $\Theta(N)$ to $O(N^{1/2})$ and the average-case latency is shown to improve by at least 35 percent. Furthermore, it is argued that patient combining increases the directory congestion by only an additive constant, rather than a multiplicative constant, because you get combining when you need it. We also show two ways to do the combining, free-node combining and two-pointer combining, and argue that free-node combining has a lower complexity of implementation.

There are two unexpected results on combining in Section 3.4.3. First, greedily combining equal-height segments does not decrease the latency and sometimes even increases it. Second, increasing the size of network queues can increase the latency of hot-spot accesses for these extensions. Of course, increasing the queue sizes decreases the number of queues involved in tree saturation for any one hot spot.

Overall, the binary structure, with request reserving and patient free-node combining, is best for use with recursive doubling. Furthermore, request forwarding is necessary only for efficient invalidate distribution with these extensions.

# Chapter 4

# Potential Latencies for Recursive Doubling

## 4.1. Introduction

Cache-coherence protocols are used in multiprocessors to help maintain a consistent shared-memory model for programming. Some cache-coherence protocols, like SCI [Gust92a], use linked lists of cache lines to maintain hardware cache coherence and prevent system-wide broadcasts. As processors read and write data, the cache-coherence protocols create lists, distribute data, and destroy lists of cache lines. It is important that these list operations be efficient as the system size (number of caches) increases. However, the standard SCI cache-coherence protocol is not efficient for long lists. We believe that additional hardware support for these operations will significantly reduce the programming time required for good performance on systems with thousands of coherent caches.

In Chapter 3, we add temporary pointers to the list of cache lines to make trees that are efficient for the two multicasts that are required for reading and writing shared data. These trees help reduce the latency of reads and writes from $O(N)$ to $O(\log(N))$, where $N$ is the number of participating cache lines or *nodes.* We propose two candidate structures and study their performance in terms of latency and traffic. *Latency* is defined to be the number of network messages on the critical path, where each message takes unit time. *Traffic* is defined to be the total number of messages required. Two pointer structures, symmetric and binary, are created with *recursive doubling,* which will be reviewed later in Section 4.4.1. We call these extensions the *recursive-doubling extensions to SCI.*

Since we show in Section 3.6.4 that the traffic difference for the two structures is insignificant, this chapter focuses on the performance aspect of latency. We ask a number of questions about the optimality of the given pointer structures and the optimality of other possible structures. Comparing the given structures against theoretical lower bounds on latency helps us to determine useful topics for future work, if any. These comparisons also provide a level of confidence as to the optimalness of the given structures. We explore four topics related to lower bounds.

(1) Recall from Section 3.1 that the optimal latency is influenced by three constraints. First, a doubly-linked list must be embedded in the structure and connect all nodes in the structure, as defined by SCI. Second, the latency of both data and invalidate distribution must be minimized, each of which begins on opposite sides of the structure. Third, the latency to create the pointer structure must be minimized. How much do these constraints interact? Can a single pointer structure yield a near-optimal solution for every constraint, or does this combination of constraints necessitate a significant compromise? How do the symmetric and binary structures compare to the optimal structure for these constraints, taking the constraints both separately and collectively?

(2) Recall that a pointer structure is constrained to at most one temporary pointer per node, in addition to the two list pointers. This constraint is necessary for the structure to be compatible with recursive doubling and to minimize the cache-tag size. Is one temporary pointer a reasonable limitation? What is the best latency that can be achieved with multiple temporary pointers?

(3) Recall that node removals (cache-line rollouts) can cause temporary pointers to become stale because they are not updated. Since there is no efficient way for a node to determine what temporary pointers address it, there is no efficient way to update or delete a temporary pointer. However, it is possible to force temporary pointers to be *bidirectional,* such that a temporary pointer from *X* to *Y* implies another temporary pointer from *Y* to *X*. Then, a node would know all the pointers that address it so that it can update them when it is removed and avoid stale temporary pointers. Is this useful? In particular, how much connectivity is lost if two temporary pointers always simulate a bidirectional pointer?

(4) Recall that the symmetric structure limits the flow of invalidates towards the tail; backward list pointers are never used to propagate invalidates. This is called

*unidirectional information flow*. However, the binary structure with pointer reversing allows invalidates to flow in both directions, using both forward and backward list (and temporary) pointers. This provides a richer connectivity that can be used to advantage in the binary structure. What is the potential disadvantage of unidirectional information flow for other pointer structures?

We define *potential latency* to be the lower bound on latency for a given set of constraints. We call this the potential latency, rather than the optimal latency, in order to signify that the details for achieving this latency are not yet in place and, therefore, may be a topic for future work. In this chapter we determine potential latencies for different parts of a cache-coherence protocol, given that the protocol uses a structure that is embedded with a doubly-linked list. A study of potential latencies has three purposes. First, it gives an optimal against which to compare proposed solutions. Note that it does not suffice to merely determine complexity orders since multiplying constants affect which structures are candidates for hardware implementation. Second, it gives insight into what characteristics of a problem are difficult and which constraints make it difficult. In particular, if a constraint causes a significant increase in the potential latency, then there is a tendency for the given constraint to increase the median and average latencies of a proposed structure with that constraint. Third, the exploration of numerous constraints suggests promising variations for future work.

In studying potential latency, we discover some important results. First, we show that the potential advantage of additional temporary pointers decreases rapidly as more pointers per node are added. This is reassuring since space and algorithmic limitations probably restrict the number of temporary pointers to one per node. We also show that the number of temporary pointers is more important than the restrictions of bidirectional pointers, unidirectional information flow, and *noncircular lists,* where the list head and tail do not have list pointers to each other. Second, we show that using the same temporary pointers for both data and invalidate distribution necessarily increases the sum of their latencies, assuming one temporary pointer per node. Third, we show that recursive doubling contributes significantly to latency and can not be easily overlapped with data distribution. Simultaneous pointer creation and data distribution is called *creation distribution.* In all cases, we compare the potential latencies to the latencies of the symmetric and binary structures.

In most cases, we solve the equivalent problem of maximizing the number of nodes for a given tree height (or latency) instead of minimizing the tree height for a given number of nodes. An approximation for the number of nodes in terms of tree height is

$$h(N) = \frac{\log_2(N)}{\log_2(r)} - \frac{\ln(k)}{\ln(r)} \pm \frac{|c| + \sum_{i=2}^{t} |k_i r_i|}{\ln(r)\left(N - |c| - \sum_{i=2}^{t} |k_i r_i|\right)}, \quad N > |c| + \sum_{i=2}^{t} |k_i r_i|, \tag{4.1}$$

where $N$ is the number of nodes, $h$ is the tree height, $\ln(x)$ is the natural log, and

$$N = k r^h + c + \sum_{i=2}^{t} k_i r_i^h, \quad h \geq 1, \quad t \geq 1, \quad |r_i| < 1 \; \forall \; 2 \leq i \leq t. \tag{4.2}$$

This result will be derived later in Section 4.2.1 and used repeatedly. Note that we assume throughout this chapter that

$$\sum_{x=y}^{z} w = 0 \;\; \forall \;\; z < y. \tag{4.3}$$

Exact analytical solutions have been computed for all recurrence relations in this chapter and have then been checked by Vaxima, a symbolic manipulator. In addition, exhaustive searches of all pointer combinations for small numbers of nodes have tested the equations, where noted. In order to aid understanding, numerical approximations are presented in addition to the exact solutions.

Concerning related work, the multicasting of information in the recursive-doubling extensions to SCI is fundamentally different than most communication problems. This is because no node knows all of the desired destinations of the multicast and nodes are allowed to change which destinations they know, effectively changing the logical communication topology. Furthermore, no node can remember more than a constant number of destinations at any one time (probably three) and SCI messages are limited to a small constant size, 96 bytes (32 bytes without data). Broadcasting variants of the gossiping problem [HeHL88] are not multicasting and do not allow modification of the communication topology. For multicasting in networks

[FrWB85, Deer88], including the Steiner problem in networks [Wint87], algorithms either create new nodes or make use of nodes that are not in the destination set. These operations are not allowed by the recursive-doubling extensions to SCI.

Throughout this chapter we derive the potential latency for each of a number of constraints. Section 4.2 gives the potential latency for one multicast in isolation. It also considers the restrictions of noncircular lists, bidirectional temporary pointers, and unidirectional information flow. Section 4.3 gives the potential latency for both multicasts (data and invalidates), given that they must use the same temporary pointers. Sections 4.2 and 4.3 are relevant to any pointer structure with embedded lists, not necessarily ones created with recursive doubling. Section 4.4 reviews recursive doubling and gives its potential latency. Section 4.5 gives the potential latency of creation distribution, which includes recursive doubling and the multicast for data. Section 4.6 discusses the above performance data and compares it to the data in Section 3.6. Section 4.7 concludes with a summary of the results. An earlier version of this chapter appears as a technical report [John91], substantially refined here.

## 4.2. Single Distribution

In this section we derive the equations for one multicast in isolation, preparing to consider the effectiveness of multiple temporary pointers per node in doubly-linked lists. We also derive the equations for the constraints of unidirectional information flow, where information propagates away from the originator, bidirectional temporary pointers, where two pointers simulate an undirected edge in the graph, and noncircular lists, where the list head and tail do not have list pointers to each other. Note that lower bounds for doubly-linked lists are also lower bounds for singly linked lists because the extra list pointer can be simulated with an additional temporary pointer.

First, we find an approximation for a lower bound on tree height, given an upper bound on the number of nodes in a list. Second, we find a lower bound on the tree height for one multicast without the limitations of either bidirectional temporary pointers or unidirectional information flow. These lower bounds are lower bounds on all other bounds given in this chapter. Third, we find a lower bound with exactly one of these constraints, either bidirectional temporary pointers or unidirectional information flow. Fourth, we find a lower bound on the tree height for one multicast with both constraints. For all cases in this section, we find a lower bound for noncircular and circular lists. The nonmathematical reader should skip to Section 4.3.

### 4.2.1. Equations: Height Approximation

First, find an approximation for a lower bound on the tree height, given an upper bound on the number of nodes in a list. Equations for the upper bound on the number of nodes usually have the form

$$N = k\,r^h + c + \sum_{i=2}^{t} k_i\,r_i^{\,h},\quad h \ge 1,\quad t \ge 1,\quad |r_i| < 1\ \forall\ 2 \le i \le t, \tag{4.4}$$

where $t = 1$ implies that the summation term is zero. Therefore, solving for $h$, a lower bound on height, is not straightforward algebra. However, the size of the summation term is no larger than $\sum_{i=2}^{t} |k_i\,r_i|$ because $|r_i| < 1$. This allows us to bound the error for an approximation of $h(N)$ for large $N$, which is given in Equation 4.1.

Solving for the first $h$ (the power of $r$) in Equation 4.4 yields

$$h = -\frac{\ln(k)}{\ln(r)} + \frac{\ln(N - c - \sum_{i=2}^{t} k_i\,r_i^{\,h})}{\ln(r)}. \tag{4.5}$$

Recall that $\ln(N) = -\ln(1/N)$, so extracting $\ln(N)$ from the rightmost term gives

$$h = \frac{\ln(N)}{\ln(r)} - \frac{\ln(k)}{\ln(r)} + \frac{\ln\left(1 + \frac{-c - \sum_{i=2}^{t} k_i\,r_i^{\,h}}{N}\right)}{\ln(r)}. \tag{4.6}$$

Then, the rightmost term becomes the error term. Series expansions show that

$$|\ln(1+x)| = \left| \sum_{k=0}^{\infty} (-1)^k \frac{x^{k+1}}{k+1} \right| \le \sum_{k=0}^{\infty} \frac{|x|^{k+1}}{k+1} < \sum_{k=0}^{\infty} |x|^{k+1} = \frac{|x|}{1-|x|} \tag{4.7}$$

for all $-1 < x < 1$. Using this inequality on the error term from Equation 4.6 gives

$$h = \frac{\ln(N)}{\ln(r)} - \frac{\ln(k)}{\ln(r)} \pm \left[ \frac{\left| -c - \sum_{i=2}^{t} k_i r_i^h \right|}{\ln(r) \left( N - \left| -c - \sum_{i=2}^{t} k_i r_i^h \right| \right)} \right]. \tag{4.8}$$

Recall that

$$\frac{|\alpha|}{N - |\alpha|} \le \frac{|\beta|}{N - |\beta|} \tag{4.9}$$

for all $|\alpha| \le |\beta| < N$ and note that

$$\left| -c - \sum_{i=2}^{t} k_i r_i^h \right| \le |c| + \sum_{i=2}^{t} \left| k_i r_i^h \right| \le |c| + \sum_{i=2}^{t} |k_i r_i| . \tag{4.10}$$

These two inequalities can be used for the error term in Equation 4.8 to get

$$h(N) = \frac{\ln(N)}{\ln(r)} - \frac{\ln(k)}{\ln(r)} \pm \left[ \frac{|c| + \sum_{i=2}^{t} |k_i r_i|}{\ln(r) \left( N - \left( |c| + \sum_{i=2}^{t} |k_i r_i| \right) \right)} \right], \quad N > |c| + \sum_{i=2}^{t} |k_i r_i|, \tag{4.11}$$

which is equivalent to Equation 4.1. Although a smaller error term could be found by more detailed analysis, the given one is sufficient and approaches zero for large $N$.

### 4.2.2. Equations: Neither Constraint

Second, given $p$ temporary pointers per node, find a lower bound on the tree height for one multicast without the limitations of bidirectional temporary pointers and unidirectional information flow. We can define two interdependent recurrence relations by identifying the two types of nodes in a spanning tree for the list and noting the maximum branching factor for each node. See Figure 4.1. If a node is connected to its parent by a temporary pointer, then the node can have $p+2$ children: $p$ by temporary pointers and 2 by list pointers. However, if a node is connected to its parent by a list pointer, then the node can have only $p+1$ children because only one list pointer can reach a new node. This leads us to the following recurrence relations, where $h$ is

**Maximum Nodes with One Temporary Pointer**

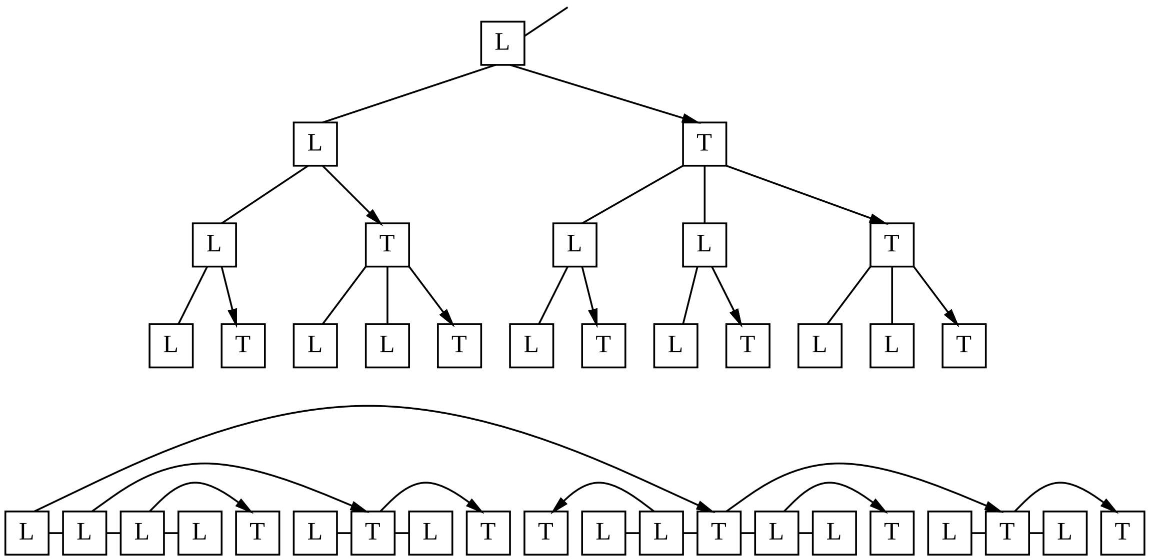



**Fig. 4.1:** This figure illustrates a minimal height spanning tree, given one temporary pointer per node. The same structure is shown in tree form on the top and in list form on the bottom. Temporary pointers are denoted by arrows and list pointers are denoted as lines without arrows. Unused list pointers are not shown. Nodes are represented by labeled boxes. Nodes labeled L are reached via a list pointer and nodes labeled T are reached via a temporary pointer.

---

the height of a subtree and $T(h)$ and $L(h)$ are the maximum number of nodes in a subtree whose root is visited by a temporary pointer and a list pointer respectively.

$$\begin{aligned} &T(h) = 1 + p\,T(h-1) + 2\,L(h-1),\ \ T(1) = 1,\ \ T(0) = 0, \\ &L(h) = 1 + p\,T(h-1) + L(h-1),\ \ L(1) = 1,\ \text{ and }\ L(0) = 0. \end{aligned} \tag{4.12}$$

$L(h)$ is the desired function for noncircular lists because the end of a noncircular list can use only one list pointer. Standard methods for solving recurrence relations [PuBr85] yield

$$\begin{aligned} &N(h,p) = K_1\left(\frac{p+1+\sqrt{D_1}}{2}\right)^h - \frac{1}{2p} + K_2\left(\frac{p+1-\sqrt{D_1}}{2}\right)^h \approx K_1\left(\frac{p+1+\sqrt{D_1}}{2}\right)^h - \frac{1}{2p}, \\ &K_1 = \frac{D_1 + (3p+1)\sqrt{D_1}}{4p\,D_1},\ \ K_2 = \frac{D_1 - (3p+1)\sqrt{D_1}}{4p\,D_1},\ \ D_1 = p^2 + 6p + 1,\ \ h \ge 0, \end{aligned} \tag{4.13}$$

where each node has at most $p \geq 1$ temporary pointers and $h$ is the height of the shortest-path spanning tree that is rooted at one end of the list. Approximating for $p = 1$ yields

$$N(h) = 0.6036\,(2.4142)^h - 0.5000 - 0.1036\,(-0.4142)^h, \;\; h \geq 0. \tag{4.14}$$

Note that the contribution of one root becomes negligible for large $h$. Using Equation 4.1 yields

$$h(N) = 0.7864 \log_2(N) + 0.5729 \pm \frac{0.6160}{N - 0.5429}, \;\; N \geq 1. \tag{4.15}$$

$A(h)$ from Equation 4.13 is the desired function for circular lists because the ‘‘end’’ of a circular list can use two list pointers. Standard methods for solving recurrence relations [PuBr85] yield

$$N(h,p) = K_3 \left( \frac{p+1+\sqrt{D_1}}{2} \right)^h - \frac{1}{p} + K_4 \left( \frac{p+1-\sqrt{D_1}}{2} \right)^h \approx K_3 \left( \frac{p+1+\sqrt{D_1}}{2} \right)^h - \frac{1}{p},$$
$$K_3 = \frac{D_1 + (p+1)\sqrt{D_1}}{2\,p\,D_1}, \;\; K_4 = \frac{D_1 - (p+1)\sqrt{D_1}}{2\,p\,D_1}, \;\; D_1 = p^2 + 6p + 1, \;\; h \geq 0, \tag{4.16}$$

where each node has at most $p \geq 1$ temporary pointers and $h$ is the height of the shortest-path spanning tree that is rooted at one end of the list. Approximating for $p = 1$ yields

$$N(h) = 0.6768\,(2.4142)^h - 1 - 0.1464\,(-0.4142)^h, \;\; h \geq 0. \tag{4.17}$$

Note that the contribution of one root becomes negligible for large $h$. Using Equation 4.1 yields

$$h(N) = 0.7864 \log_2(N) + 0.1797 \pm \frac{1.2034}{N - 1.0607}, \;\; N \geq 2. \tag{4.18}$$

### 4.2.3. Equations: Either Constraint

Third, given $p$ temporary pointers per node, find a lower bound on the tree height for one multicast with exactly one of the limitations of either bidirectional temporary pointers or unidirectional information flow. Consider noncircular lists. Since the maximum branching factor is $p + 1$ for all nonleaf nodes in the spanning tree, the maximum number of nodes is the geometric sum

$$N(h,p) = \sum_{k=0}^{h-1} (p+1)^k, \;\; h \geq 0. \tag{4.19}$$

Algebraic manipulation yields

$$N(h,p) = \frac{1}{p}(p+1)^h - \frac{1}{p}, \;\; h \geq 0. \tag{4.20}$$

Simplifying for $p = 1$ yields

$$N(h) = 2^h - 1, \;\; h \geq 0. \tag{4.21}$$

Using Equation 4.1 yields

$$h(N) = \log_2(N) + 0 \pm \frac{1.4427}{N-1}, \;\; N \geq 2. \tag{4.22}$$

Consider circular lists. If information flow is unidirectional, then the maximum number of nodes is the same as for a noncircular list. This is because the ‘‘end’’ of the circular list can not use one of its list pointers, due to the flow limitation. However, if the temporary pointers are bidirectional, then the ‘‘end’’ of the circular list has two usable list pointers. Except for the root node, the maximum branching factor is then $p+1$, yielding the geometric sum

$$N(h,p) = 1 + (p+2)\sum_{i=0}^{h-2} (p+1)^i, \;\; h \geq 2. \tag{4.23}$$

Algebraic manipulation yields

$$N(h,p) = \frac{p+2}{p}(p+1)^{h-1} - \frac{2}{p}, \;\; h \geq 2. \tag{4.24}$$

Simplifying for $p = 1$ yields

$$3 \cdot 2^{h-1} - 2, \;\; h \geq 2. \tag{4.25}$$

Using Equation 4.1 yields

$$h(N) = \log_2(N) - 0.5850 \pm \frac{2.8854}{N-2}, \;\; N \geq 3. \tag{4.26}$$

### 4.2.4. Equations: Both Constraints

Fourth, given $p$ temporary pointers per node, find a lower bound on the tree height for one multicast with both limitations of bidirectional temporary pointers and unidirectional information flow. The derivation is similar to that of Equation 4.13. Again, we can define two interdependent recurrence relations by identifying the two types of nodes in a spanning tree for the list and noting the maximum branching factor for each node. If a node is connected to its parent by a list pointer, then the cache line can have $p+1$ children: $p$ by temporary pointers and 1 by the list pointer of the appropriate direction. However, if a node is connected to its parent by a temporary pointer, then the cache line can have only $p$ children because one of its temporary pointers connects to the parent. This leads us to the following recurrence relations, where $h$ is the height of a subtree and $L(h)$ and $T(h)$ are the maximum number of cache lines in a subtree whose root is visited by a list pointer and a temporary pointer respectively.

$$\begin{aligned} &L(h) = 1 + L(h-1) + p\,T(h-1),\ \ L(1) = 1,\ \ L(0) = 0, \\ &T(h) = 1 + L(h-1) + (p-1)\,T(h-1),\ \ T(1) = 1,\ \text{ and }\ T(0) = 0. \end{aligned} \tag{4.27}$$

$L(h)$ is the desired function for noncircular lists because $L(h) \geq T(h)$. Standard methods for solving recurrence relations [PuBr85] yield

$$\begin{aligned} &N(h,p) = K_7\left(\frac{p+\sqrt{D_2}}{2}\right)^h - \frac{2}{p} + K_8\left(\frac{p-\sqrt{D_2}}{2}\right)^h \approx K_7\left(\frac{p+\sqrt{D_2}}{2}\right)^h - \frac{2}{p}, \\ &K_7 = \frac{D_2 + 2\sqrt{D_2}}{p\,D_2},\ \ K_8 = \frac{D_2 - 2\sqrt{D_2}}{p\,D_2},\ \ D_2 = p^2 + 4,\ \ h \geq 0. \end{aligned} \tag{4.28}$$

Approximating for $p=1$ yields

$$N(h) = 1.1708\,(1.6180)^h - 2 - 0.1708\,(-0.6180)^h,\ \ h \geq 0. \tag{4.29}$$

Note that the contribution of one root becomes negligible for large $h$. Using Equation 4.1 yields

$$h(N) = 1.4404\log_2(N) - 1.3277 \pm \frac{4.2918}{N-2.0652},\ \ N \geq 3. \tag{4.30}$$

Since the flow is unidirectional, one of the list pointers at the ‘‘end’’ of a circular list can not be used. Therefore, the results for a circular list with both constraints is the same as for a noncircular list.

### 4.3. Double Distribution

In this section, we give the optimal pointer structure for minimizing the tree heights of two multicasts that begin at opposite sides of the list. In the process, we derive the sum of these tree heights. Note that there is a fundamental difference between the two multicasts. Temporary pointers that are used for data distribution are stored in nodes that *need* the information, but temporary pointers that are used for invalidate distribution are stored in nodes that *have* the information.

Recall from Section 3.5.2.2 that pointer reversing is used in the binary structure to reduce the tree height for invalidate distribution. On the other hand, the symmetric structure does not use pointer reversing. To provide a fair comparison to the potential latency for each, we derive two lower bounds. First, we derive the lower bound on the sum of the tree heights for the two multicasts where information flow is unidirectional for both, as is true for the symmetric structure. Second, to account for pointer reversing in the binary structure, we derive the lower bound where data distribution is unidirectional, but invalidate distribution is not so limited. The non-mathematical reader should skip to Section 4.4.

#### 4.3.1. Equations: Without Pointer Reversing

Without pointer reversing, information flow is unidirectional for both data and invalidate distributions. However, the flow is in the opposite direction for each because the information starts at opposite ends of the list. Figure 4.2 shows the recursively constructed set of pointers that is optimal for these two multicasts.

Consider $T(l,d)$, the maximum number of nodes in a structure where $l$ is the height of the minimal spanning tree rooted on the left ($l$ for left) and $d$ is the height of the minimal spanning tree rooted on the right ($d$ for data). Let $j$ be the number of pointers in a structure that are not nested. A pointer between nodes $x$ and $y$ in a structure is *nested* if and only if there exists another pointer between nodes $w$ and $z$ in that structure such that $w$ is to the left of $x$ and $y$ and $z$ is to the right of $x$ and $y$. Intuitively, the concept of nesting divides $T(l,d)$ into two groups of nodes: 1) $j+1$ nodes that are sequentially connected by temporary pointers and 2) the $j$ substructures of nodes that are spanned by the $j$ pointers. For simplicity, temporary pointers in this

**Construction for Two Multicasts**

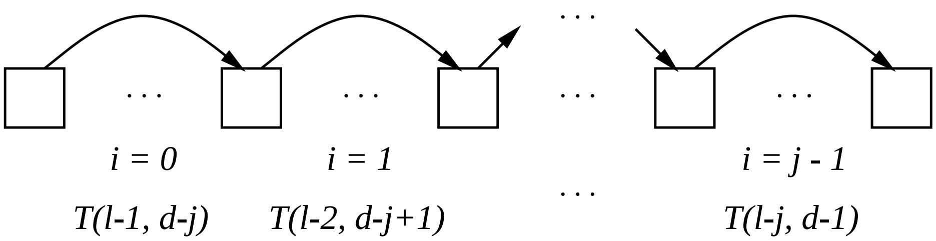



**Fig. 4.2:** This figure shows the construction for the maximum number of nodes, given the heights of the two minimal spanning trees rooted on opposite sides of the list. Nodes are represented by boxes and temporary pointers are represented by arrows. The first row of text below the nodes indicates that the structure is divided into $j$ substructures, numbered consecutively from $i=0$ to $i=j-1$ (left to right). The second row shows the maximum number of nodes in substructure $i$, between the given endpoints.

---

construction may span substructures of size zero. In other words, every node has exactly one temporary pointer. Note that it takes $i+1$ hops for invalidates to reach and enter the $i^{\text{th}}$ substructure ($0 \le i < j$). So, the spanning tree from the left endpoint of the $i^{\text{th}}$ substructure has $l-(i+1)$ hops remaining. Likewise, it takes $j-i$ hops for data to reach and enter the substructure, leaving $d-(j-i)$ hops. Therefore, the $i^{\text{th}}$ substructure contains exactly $T(l-i-1,\, d-j+i)$ nodes and $T(l,d)$ contains the nodes with unnested pointers plus the sum of the substructure sizes, which is

$$1+j+\sum_{i=0}^{j-1} T(l-i-1,\, d-j+i). \tag{4.31}$$

It remains to determine $j$. There can be at most $\min(l,d)-1$ substructures because a spanning tree in a substructure can not have negative height. For the same reason, $j$ can not be less than zero. Therefore,

$$\begin{aligned} T(l,d) &= \max_{0 \le j < \min(l,d)} \left\{ 1+j+\sum_{i=0}^{j-1} T(l-i-1,\, d-j+i) \right\}, \\ T(0,0) &= T(0,d) = T(l,\,0) = 0, \end{aligned} \tag{4.32}$$

where $l \ge 1$ and $d \ge 1$ are the heights of the two spanning trees.

As a check on our work, we have observed that

$$\lim_{l \to \infty} T(l,d) = \lim_{d \to \infty} T(l,d) = 2^{\min(l,d)} - 1 \quad \forall \quad l,d \le 40, \tag{4.33}$$

which agrees with Equation 4.21. Experimentally, we have found that $j$ is not always $\min(l,d)-1$ as $l$ and $d$ vary. Due to this, we have been unable to find a closed-form solution for $T(l,d)$.

It seems reasonable to consider minimization of the sum of the two tree heights,

$$N(h) = \max\{T(l,d) : l + d = h\}. \tag{4.34}$$

Using linear regression analysis for $1 < \log_2(N(h)) < 16$ and integer $h$ yields

$$h(N) \approx 2.608 \log_2(N) - 0.208 \tag{4.35}$$

with correlation coefficient 0.9998. Solving for $N$ yields

$$N(h) \approx 1.057\,(1.304)^h. \tag{4.36}$$

### 4.3.2. Equations: With Pointer Reversing

We relax the constraints of the previous equations for the next set of equations. When pointer reversing is used, information flow for invalidate distribution is not unidirectional. Figure 4.3 shows the recursively constructed set of pointers that is optimal for these two multicasts.

Consider $T(l,r,d)$, the maximum number of nodes in a structure. The variable $d$ is the height of a minimal spanning tree rooted on the right. It represents the tree height allowed for data, starting on the right side of the substructure. The variables $l$ and $r$ are the heights of two invalidate trees, rooted on the left and right respectively. They collectively make a spanning forest of two trees that is used for invalidate distribution. Two variables, $l$ and $r$, are required for invalidates in order to represent the propagation of invalidates in each direction. These variables represent the two tree heights allowed for invalidates, starting with the given height on each side of the substructure. For example, with $l = 3$ and $r = 12$, the number of nodes in the substructure is limited so that an invalidate can reach any node in either 3 steps from the left side or 12 steps from the right side, counting the 1 step required on each side to enter the substructure.

**Construction for Pointer Reversing**

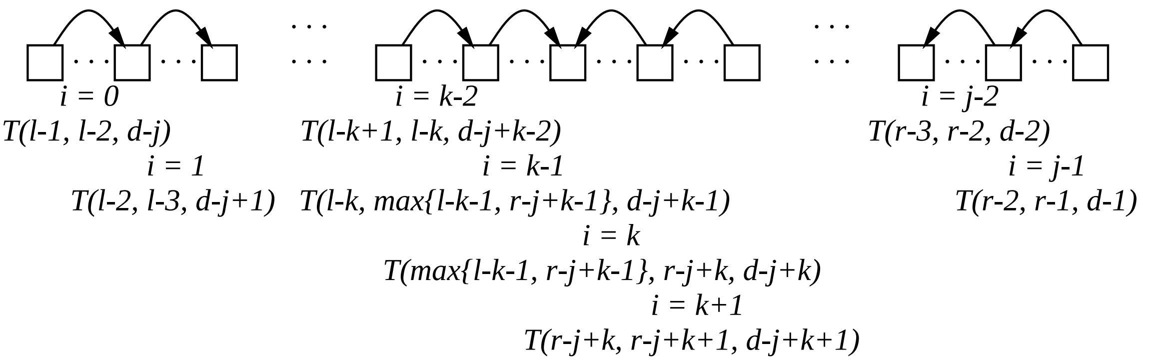



**Fig. 4.3:** This figure shows the construction for the maximum number of nodes, given two multicasts and pointer reversing for one of them. Nodes are represented by boxes and temporary pointers are represented by arrows. The structure is divided into *j* substructures. Each substructure is numbered consecutively from $i = 0$ to $i = j - 1$ (left to right). Substructure $i = k$ is a pivot point in the structure such that $j - k$ temporary pointers are reversed, where $i \geq k$ . The maximum number of nodes in substructure *i*, not counting the endpoints, is centered below each *i*.

---

Let *j* be the number of temporary pointers in a structure that are not nested. Intuitively, this divides $T(l, r, d)$ into two groups of nodes: 1) $j + 1$ nodes that are sequentially connected by temporary pointers and 2) the *j* substructures of nodes that are spanned by the *j* pointers. For simplicity, temporary pointers in this construction may span substructures of size zero. In other words, every node has exactly one temporary pointer. Note that it takes $j - i$ hops for data to reach and enter the $i^{\text{th}}$ substructure ($0 \leq i < j$). So, the spanning tree from the right endpoint of the $i^{\text{th}}$ substructure has $d - (j - i)$ hops remaining.

Invalidate distribution is more complicated than data distribution because pointers are reversed and invalidates can enter a substructure from either side. The problem is made easier by noting that there exists a pivot point *k* such that a structure reverses its $i^{\text{th}}$ temporary pointer if and only if $i \geq k$. This existence can be proved by showing that more than one pivot point per structure can not increase the number of nodes in the structure. This is because invalidates start at the endpoints of a structure and propagate toward the center substructure. Extra pivots mean that it can only take longer for invalidates to reach the center substructures because some temporary pointers will be pointing the wrong direction. Furthermore, if it can be shown that the invalidate propagates through substructure *i* before the substructure's unnested temporary pointer is used, then pivot point $k$ has been placed on the wrong side of $i$. Therefore, if $k$ is chosen correctly (to maximize the number of nodes), then invalidates will never propagate through a substructure before the invalidate reaches the other endpoint via the unnested pointer.

Invalidates might reach and enter the $i^{\text{th}}$ substructure from either of the two sides of the structure. If $i$ is less than $k-1$, then the invalidate from the left side of the structure will be first to reach substructure $i$. This is because each substructure $i < k$ has its unnested pointer reversed. Likewise, if $i$ is greater than $k$, then the invalidate from the right side of the structure will be first to reach substructure $i$. Also, the left side of substructure $k-1$ and the right side of substructure $k$ will be reached first by invalidates from the left and right sides of the structure respectively. However, there is a race to the center node of the structure, which is both the left side of substructure $k$ and the right side of substructure $k-1$. This means that we must take the maximum number of substructure nodes allowed by either origin.

The following four equations summarize the values of $T(l,r,d)$ in Figure 4.3 for a given substructure.

$$T(l-i-1,\ l-i-2,\ d-j+i),\ \ i < k-1. \tag{4.37}$$

$$T(l-i-1,\ \max\{l-i-2,\ r-j+i\},\ d-j+i),\ \ i = k-1. \tag{4.38}$$

$$T(\max\{l-i-1,\ r-j+i-1\},\ r-j+i,\ d-j+i),\ \ i = k. \tag{4.39}$$

$$T(r-j+i-1,\ r-j+i,\ d-j+i),\ \ i > k. \tag{4.40}$$

Adding the $j+1$ substructure endpoints to the sizes of the substructures yields

$$1 + j + \sum_{i=0}^{j-1} T\left(G(k-0,\ i,\ l-i-1,\ r-j+i-1),\ G(k-1,\ i,\ l-i-2,\ r-j+i-0),\ d-j+i\right), \tag{4.41}$$

where

$$G(u,v,x,y) = \begin{cases} x & \text{if } u > v \\ y & \text{if } u < v \\ \max(x,\ y) & \text{if } u = v. \end{cases} \tag{4.42}$$

In order to determine the maximum number of nodes, we maximize for all $j$ such that $1 \le j < \min(l+r,\ d)$ and for all $k$ such that $\max(0,\ j-r) \le k \le \min(j,l)$. This gives

$$F(l,r,d) = \max_{\substack{1 \le j < \min(l+r,\ d) \\ \max(0,\ j-r) \le k \le \min(j,l)}} \left\{ 1 + j + \sum_{i=0}^{j-1} T \begin{bmatrix} G(k-0,\ i,\ l-i-1,\ r-j+i-1), \\ G(k-1,\ i,\ l-i-2,\ r-j+i-0), \\ d-j+i \end{bmatrix} \right\}, \quad (4.43)$$

where $G(u,v,x,y)$ is as given in Equation 4.42. The base cases are when $\min(l+r,\ d) \le 1$, when there are no temporary pointers, so

$$T(l,r,d) = \begin{cases} \min(l+r,\ d) & \text{if } \min(l+r,\ d) \le 1 \\ F(l,r,d) & \text{otherwise.} \end{cases} \quad (4.44)$$

Since invalidate distribution begins on the left, $T(l,\ 0,d)$ gives the desired value. The top level tree heights are $d$ for data with unidirectional information flow and $l$ for invalidate distribution with pointer reversing.

It seems reasonable to consider minimization of the sum of data and invalidate distribution,

$$N(h) = \max\{T(l,\ 0,d) : l + d = h\}. \quad (4.45)$$

Using linear regression analysis for $1 < \log_2(N(h)) < 16$ and integer $h$ yields

$$h(N) \approx 2.038 \log_2(N) + 1.536 \quad (4.46)$$

with correlation coefficient 0.9997 Solving for $N$ yields

$$N(h) \approx 1.057\,(1.304)^h. \quad (4.47)$$

### 4.4. Pointer Creation

In this section we consider the potential latency for pointer creation. This is an important part of the total latency. Note that it is not enough to find the optimal structure because it must be possible to create the structure efficiently. Here, we consider pointer creation in isolation, but consider creation distribution (pointer creation with data distribution) later in Section 4.5. Before deriving the potential latency for pointer creation, we review recursive doubling, which is described in Section 3.2.3. The nonmathematical reader should read Section 4.4.1 and then skip to Section 4.5.

### 4.4.1. Recursive Doubling

The object of recursive doubling is to efficiently create temporary pointers that will be useful for efficient distribution of information. We start by assuming that each node knows its own list position, called a pcount. First, each node determines the pcount of the node to which it wants a temporary pointer, if any. This is a function $f$ of the system size $M$ and the node's own pcount $x$ such that a temporary pointer addresses a smaller pcount and all temporary pointers are nested. Formally, $x \geq f_M(x)$ (smaller pcounts) and if $y > x > f_M(y)$, then $f_M(x) > f_M(y)$ (nested temporary pointers). It remains to show how to create the temporary pointers for some function $f_M$, shown in Figure 4.4. For all nodes $u$ such that $u \neq f_M(u)$, node $u$ requests a pointer to node $f_M(u)$ from its forward neighbor, node $u-1$. When a node $v = f_M(v)$ receives a request for a pointer to $f_M(u) = v-1$, it returns pointer $v-1$ to the requester, shown in Figure 4.4A. Otherwise, node $v$ forwards the request to $v-1$, shown in Figure 4.4B. When node $u$ receives a request while waiting, shown in Figure 4.4C, it delays the request until it receives the pointer to $f_M(u)$. Finally, $u$ forwards the delayed request to $f_M(u)$.

After the temporary pointers are created, the data and invalidates can be efficiently multicast. If a node $u$ has a temporary pointer, then it receives the data from node $f_M(u)$ and later forwards an invalidate to that node and node $u-1$. Otherwise, node $u$ requests data and forwards invalidates with respect to only node $u-1$. Note that node $f_M(u)-1$ requests the data from node $f_M(u)$ on behalf of node $u$. This request forwarding prevents an extra message delay during data distribution.

### 4.4.2. Equations: Pointer Creation

Next, we determine the greatest-distance pointer (the pointer that spans the most nodes) that can be created in $h$ steps. Figure 4.5 illustrates the construction. In order to get the greatest-distance pointer, recursive doubling must finish creating each temporary pointer by the time it is needed. The number of nodes spanned by a temporary pointer is represented by the distance function $D(x)$, where $x$ is the required number of steps to reach the left endpoint of that temporary pointer. Note that $D(x)$ is *one plus* the number of nodes between the endpoints.

**Examples of Recursive Doubling**

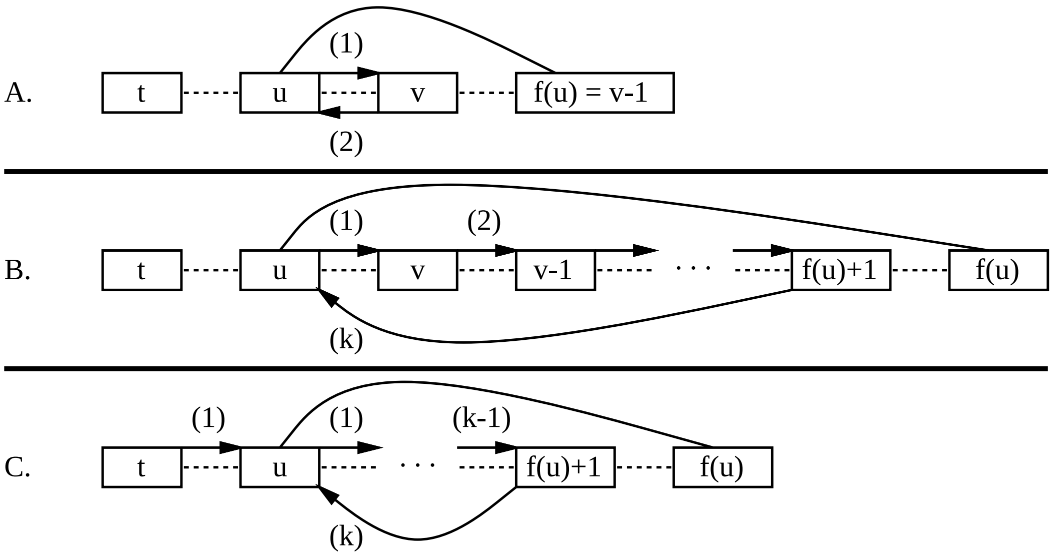



**Fig. 4.4:** This figure gives three examples of recursive doubling, separated by thick horizontal lines. Labeled boxes are nodes, where the label represents the pcount. Labeled arrows are messages, where the label represents the time step. Dotted lines represent list connections. A curved line above a list indicates the desired temporary pointer.

---

The figure leads us to the following recurrence relation.

$$D(h) = 3 + \sum_{i=2}^{h-2} D(i), \;\; h \geq 3, \;\; D(3) = 3, \;\; D(2) = 2. \tag{4.48}$$

Substitution of $h+2$ and $h+1$ for $h$ and then subtraction yield

$$D(h+2) - D(h+1) = D(h), \;\; h \geq 2, \;\; D(3) = 3, \;\; D(2) = 2, \tag{4.49}$$

which is the standard Fibonacci sequence for $h \geq 2$. Standard methods for solving recurrence relations [PuBr85] yield

$$D(h) = \left(\frac{5+\sqrt{5}}{10}\right)\left(\frac{1+\sqrt{5}}{2}\right)^h + \left(\frac{5-\sqrt{5}}{10}\right)\left(\frac{1-\sqrt{5}}{2}\right)^h, \;\; h \geq 2. \tag{4.50}$$

**Construction for Greatest-Distance Temporary Pointers**

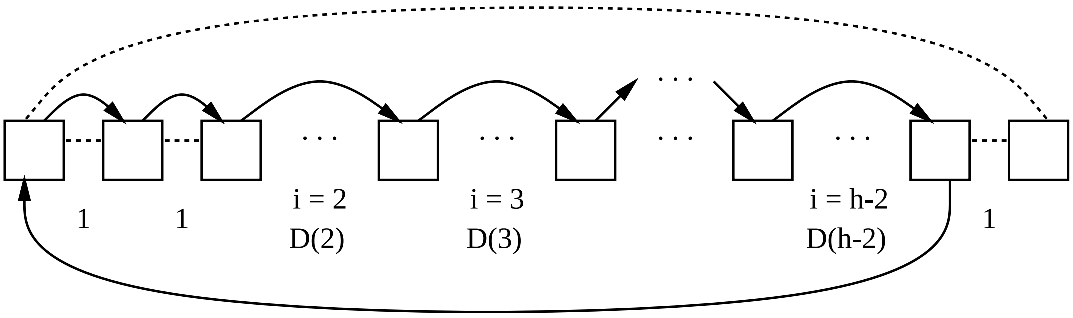



**Fig. 4.5:** This figure shows the construction for creating the greatest-distance temporary pointer in $h$ steps. Boxes represent nodes. Dotted lines between nodes indicate list pointers. Arrows represent messages. Arrows above the nodes also represent temporary pointers. The dotted line above the nodes represents the greatest-distance temporary pointer that can be created in $h$ steps. The function $D(x)$ represents the distance of the temporary pointer above it, where it takes $x$ steps to reach the left endpoint of that temporary pointer. The numbers below the list pointers indicate no temporary pointer for distance 1.

---

This is equivalent to

$$D(h) = (0.7236)\,(1.6180)^h + (0.2764)\,(-0.6180)^h \approx (0.7236)\,(1.6180)^h, \ \ h \geq 2. \qquad (4.51)$$

Note that the contribution of one root becomes negligible for large $h$. Using Equation 4.1 yields

$$h(N) = 1.4404 \log_2(N) + 0.6723 \pm \frac{0.3550}{N - 0.1708}, \ \ N \geq 2. \qquad (4.52)$$

### 4.5. Creation Distribution

In this section we give the lower bound on the latency for creation distribution, which simultaneously creates temporary pointers and distributes data. Before deriving the equation for the maximum number of nodes for a given latency, we derive a preliminary equation that will be useful in deriving the desired equation. The nonmathematical reader should skip to Section 4.6.

#### 4.5.1. Equations: Preliminary

As a preliminary step, we derive the equation for $T(r,d)$, which is the maximum number of nodes that can be spanned by a temporary pointer in $r$ steps (using recursive doubling) and can then be sent the data in $d$ steps, using the temporary pointers so created. Note that in this function there is no overlap of work. That is, recursive doubling completes before data is distributed. In order to maximize the number of nodes in a structure, the temporary pointers are constructed for the structure as follows and as shown in Figure 4.6.

Partition the structure into $M$ substructures, where $M = \min(r-2, d-1)$ and the right endpoint of one substructure is the same as the left endpoint of another (except for the rightmost and leftmost endpoints). Number these substructures from $z = 1$ to $z = M$, from right to left. Then, create a temporary pointer across each substructure from the left endpoint to the right. Recursively construct the temporary pointers for each substructure with new rules for $r$ and $d$, as follows. Since data reaches and enters the $z^{\text{th}}$ substructure in $z$ message delays, the data must be distributed throughout that substructure in $d - z$ message delays. Consider the *spanning pointer,* the least-distance temporary pointer that spans the whole structure and does not have an endpoint in the structure. Since recursive doubling for the spanning pointer must complete in $r$ message delays, the $z^{\text{th}}$ substructure must finish recursive doubling in $r - z - 1$ message delays. Note that recursive doubling can not complete in less than 2 message delays (request and response) and that data can not be forwarded in less than 1 message delay. These observations lead to the following formula.

$$
\begin{aligned}
&T(r,d) = M + 1 + \sum_{z=1}^{M} T(r-z-1,\, d-z), \;\; d \geq 1, \;\; r \geq 2, \\
&T(x,\, 0) = T(0,x) = T(1,x) = 0, \;\; x \geq 0.
\end{aligned}
\tag{4.53}
$$

The value $M + 1$ represents the number of overlapping endpoints of these substructures, sequentially connected by temporary pointers. Each term of the summation represents one less than the distance of the temporary pointer that spans the $z^{\text{th}}$ substructure. Since this construction satisfies the constraints of $r$ and $d$, it remains to show that it is maximal and to simplify the summation.

Three observations show that $T(r,d)$ is maximal. First, $M$ is the largest number of substructures that are possible for the given values of $r$ and $d$. Second, decreasing the length of any substructure would not allow an increase in the length of any other substructure because the constraints are independent. Third, decreasing the number of substructures decreases the total

**First Construction for One Multicast with Recursive Doubling**

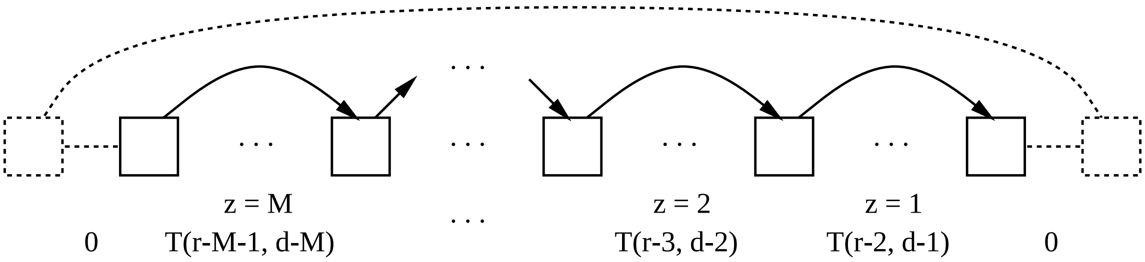



**Fig. 4.6:** This figure shows the construction for $T(r,d)$, which is the maximum number of nodes that can be spanned by a temporary pointer in $r$ steps and can then be sent the data in $d$ steps, using the temporary pointers so created. Boxes represent nodes and arrows represent temporary pointers. Dotted nodes and lines represent the superstructure, not part of $T(r,d)$. The longest pointer in the figure is the temporary pointer that is created in $r$ steps. Each substructure is numbered from $z=1$ (right) to $z=M$ (left), directly below the substructure. The number of nodes in the substructure, not counting the shown endpoints, is the function shown below the substructure number.

---

length of the structure because removal of the last $i$ substructures does not relax the constraints (on $r$ or $d$) for the other $M-i$ substructures. Therefore, $T(r,d)$ is maximal.

In order to simplify Equation 4.53, note that

$$T(r-1, d-1) = M + \sum_{z=1}^{M-1} T(r-z-2, d-z-1), \tag{4.54}$$

where $M = \min(r-2, d-1)$ as defined above. Changing the index $z$ and then substituting $T(r,d)$ from Equation 4.53 gives

$$T(r-1, d-1) = M + \sum_{z=2}^{M} T(r-z-1, d-z) = T(r,d) - 1 - T(r-2, d-1) \tag{4.55}$$

and

$$\begin{aligned} &T(r,d) = 1 + T(r-1, d-1) + T(r-2, d-1), \quad d \ge 1, \quad r \ge 2, \\ &T(x, 0) = T(0,x) = T(1,x) = 0, \quad x \ge 0. \end{aligned} \tag{4.56}$$

Since it can be shown[18] that

$$T(r,\ d) - T(r,\ d-1) = \sum_{i=0}^{r-d-1} \binom{i}{d-1} \ \ \forall\ \ d+1 \le r \le 2\,d, \tag{4.57}$$

and since there is no known closed-form solution for partial sums of Pascal's triangle, we believe that $T(r,d)$ does not have a closed-form solution.

### 4.5.2. Equations: Creation Distribution

Finally, we determine $N(h)$, the latency of creation distribution. Construct the temporary pointers for the list as follows and as shown in Figure 4.7. Partition the list into $h-1$ substructures, where the right endpoint of one substructure is the same as the left endpoint of another (except for the rightmost and leftmost endpoints). Number these substructures from $z=1$ to $z=h-1$, from right to left. Then, create a temporary pointer across each substructure from its left endpoint to its right endpoint, except for $z=1$. Recursively construct the temporary pointers for each substructure with rules for $T(r,d)$ (Equation 4.56) as follows. First, recursive doubling must complete in $r=z$ message delays, $z \ge 2$, so that data can be forwarded immediately when it reaches the $z^{\text{th}}$ substructure after $z$ message delays. Note that the first reader receives the data after 2 message delays because it must request the data from the last writer. So, $T(r,d)=0$ for $z=1$. Second, the data must be distributed throughout the $z^{\text{th}}$ substructure in $d=h-z$ message delays so that the latency is at most $h$ message delays.

Using Equation 4.56 with these rules leads to the following formula.

$$N(h) = h + \sum_{z=2}^{h-1} T(z,\ h-z), \ \ h \ge 1. \tag{4.58}$$

---

[18]This result follows from an iterative expansion of the recurrence relation for $T(r,d)$. Each iteration decreases $d$ by one, leaving behind a row of Pascal's triangle and generating a new row of terms such that the coefficients form the next row of Pascal's triangle. Terms with $r \le 1$ generate zero with no successors, masking out a complete wedge of the triangle and leaving a quadrilateral. When $d=1$, the next iteration generates all zero terms. The key observation is that starting with $T(r,d-1)$ results in the same quadrilateral without the final (partial) row. Therefore, $T(r,d) - T(r,d-1)$ is as stated.

**Second Construction for One Multicast with Recursive Doubling**

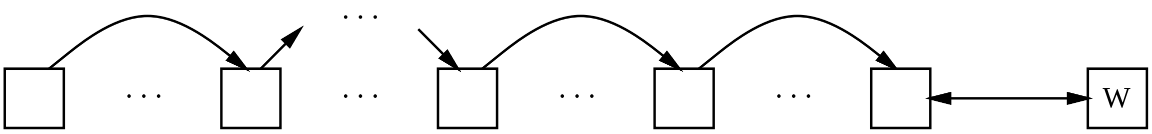



**Fig. 4.7:** This figure shows the construction for $N(h)$, which is the maximum number of nodes in a structure when $h$ is the latency for creation distribution. Boxes represent nodes and curved arrows represent temporary pointers. The last node on the right is the last writer and is connected to the next reader via a list pointer. Each substructure is numbered from $z=1$ (right) to $z=h-1$ (left), directly below the substructure. The number of nodes in the substructure, not counting the shown endpoints, is the function shown below the substructure number. $T(r,d)$ is given in Equation 4.56.

---

The $h$ term represents the number of overlapping endpoints of these substructures, sequentially connected by temporary pointers. Each term of the summation represents one less than the distance of the temporary pointer that spans the $z^{\text{th}}$ substructure. Since this construction satisfies the constraints of $h$, it remains to simplify the summation and show that it is maximal for given $h$.

First, simplify the summation. Substituting $h+1$ and $h+3$ for $h$ yields

$$N(h+1) = h+1+\sum_{z=2}^{h} T(z,\ h+1-z) \tag{4.59}$$

and

$$N(h+3) = h+3+\sum_{z=2}^{h+2} T(z,\ h+3-z) \tag{4.60}$$

respectively. Using the recurrence relation of Equation 4.56 in the last equation gives

$$N(h+3) = h+3+(h+1)+\sum_{z=2}^{h+2} T(z-1,\ h+2-z)+\sum_{z=2}^{h+2} T(z-2,\ h+2-z). \tag{4.61}$$

Note that the $(h+1)$ term comes from the +1 in Equation 4.56. Changing the index $z$ of the summations yields

$$N(h+3) = 2h + 4 + \sum_{z=1}^{h+1} T(z,\ h+1-z) + \sum_{z=0}^{h} T(z,\ h-z). \tag{4.62}$$

Removing the zero terms in the summations (the base cases from Equation 4.56) yields

$$N(h+3) = 2h + 4 + \sum_{z=2}^{h} T(z,\ h+1-z) + \sum_{z=2}^{h-1} T(z,\ h-z). \tag{4.63}$$

Subtracting Equation 4.58 and Equation 4.59 from the last equation yields

$$N(h+3) - N(h+1) - N(h) = 3,\ \ N(3) = 4,\ \ N(2) = 2,\ \ N(1) = 1. \tag{4.64}$$

Standard methods for solving recurrence relations [PuBr85] yield

$$\begin{aligned} N(h) &= K_{13}\left(\frac{D_3^{2/3} + 2^{2/3}}{D_3^{1/3}\ 2^{1/3}\sqrt{3}}\right)^h - 3 \\ &+ K_{14}\left(\frac{D_3^{2/3}\ \overline{\omega} + 2^{2/3}\ \omega}{D_3^{1/3}\ 2^{1/3}\sqrt{3}}\right)^h + K_{15}\left(\frac{D_3^{2/3}\ \omega + 2^{2/3}\ \overline{\omega}}{D_3^{1/3}\ 2^{1/3}\sqrt{3}}\right)^h, \end{aligned} \tag{4.65}$$

where

$$\begin{aligned} &K_{13} = 1 + \frac{E_3 + F_3}{276},\ \ K_{14} = 1 + \frac{E_3\ \omega + F_3\ \overline{\omega}}{276},\ \ K_{15} = 1 + \frac{E_3\ \overline{\omega} + F_3\ \omega}{276}, \\ &E_3 = D_3^{2/3}\ 2^{1/3}\left(207 - 19\sqrt{3}\sqrt{23}\right),\ \ F_3 = D_3^{1/3}\ 2^{2/3}\left(92\sqrt{3} - 18\sqrt{23}\right), \\ &D_3 = \sqrt{23} + 3\sqrt{3},\ \text{ and }\ \omega = \frac{-1+\sqrt{-3}}{2}. \end{aligned} \tag{4.66}$$

Note that $\omega$ is a primitive cubed root of unity and $\bar{\omega}$ is its complex conjugate. Equation 4.65 is equivalent to

$$\begin{aligned} N(h) &= 2.9460\,(1.3247)^h - 3 \\ &+ \left(0.0270 + 0.1184\sqrt{-1}\right)\left(-0.6624 - 0.5623\sqrt{-1}\right)^h \\ &+ \left(0.0270 - 0.1184\sqrt{-1}\right)\left(-0.6624 + 0.5623\sqrt{-1}\right)^h. \end{aligned} \tag{4.67}$$

Note that the contributions of two roots become negligible for large $h$. Using Equation 4.1 yields

$$h(N) = 2.4650\log_2(N) - 3.8423 \pm \frac{11.4193}{N - 3.2111}, \quad N \geq 4. \tag{4.68}$$

Three observations show that $N(h)$ is maximal. First, $h - 2$ is an upper bound on the number of nonzero substructures that are possible for the given value of $h$; the rightmost substructure is always size zero. This is because it takes two message delays for data to reach the first substructure (one request and one response) and then the remaining message delays to reach the remaining endpoints at the rate of one message delay per endpoint. Second, keeping the same number of substructures and decreasing the length of any substructure would not allow an increase in the length of any other substructure because the constraints are independent. Third, we *believe* that decreasing the number of substructures could not increase the total length of the list. Since Equation 4.65 is tested by exhaustive search for $h < 12$ and $N \leq 72$, $N(h)$ and $h(N)$ are probably correct as stated.

### 4.6. Comparisons

In this chapter we study lower bounds on latency for parts of the cache-coherence protocol that depend on recursive doubling. In this section, we discuss this chapter's results and compare them against the results of Section 3.6. The comparison to potential latency enables us to decide whether or not a given solution is near optimal. A study of the potential latency also yields intuition about what is the hard part of a problem. That is, if a particular constraint causes a significant increase in the potential latency, then there is a tendency for the given constraint to increase the median and average latencies of proposed solutions.

Before comparing the potential latencies to the results of Section 3.6, we discuss the usefulness of having multiple temporary pointers per node. We also consider the significance of the limitations for noncircular lists, unidirectional information flow, and bidirectional temporary pointers. After that, we compare the potential latencies to the results in Section 3.6 for four cases: one multicast in isolation, two opposite-rooted multicasts that use the same temporary pointers, recursive doubling in isolation, and creation distribution (recursive doubling with data distribution). Recall that the results of Sections 4.2 and 4.3 in this chapter represent tree heights, rather than latencies. Therefore, either 1 or 2 is subtracted before a height from Sections 4.2 or 4.3 respectively is presented in a graph of potential latency.

### 4.6.1. Multiple Temporary Pointers

Figure 4.8 shows the potential latencies for one multicast, given $1 \le p \le 9$ temporary pointers. Although recursive doubling, as currently defined, can not work with multiple temporary pointers per node, it is interesting to note the potential. These graphs show that the potential benefit of additional temporary pointers quickly reaches a point of diminishing returns. Going from zero to one temporary pointer is a substantial potential improvement because the potential latency changes from $\Theta(N)$ to $\Theta(\log(N))$. The worst-case latency also improves by the same order. Going from one to two temporary pointers still adds significant potential, resulting in a 31 to 43 percent reduction in the best-case latency for 64K nodes. However, going from two to three results in only an 11 to 25 percent reduction and the potential reduction quickly becomes insignificant as the number of temporary pointers increases beyond three. Therefore, it is not worth the effort of trying to extend recursive doubling to work with multiple temporary pointers. This result is reassuring because it is expensive to implement additional temporary pointers, each of which requires an additional 16 bits per 64-byte cache line in SCI.

The graphs also show that the potential latency is more affected by the number of temporary pointers than by the constraints of unidirectional information flow and/or bidirectional temporary pointers. This conclusion is further supported by the fact that the equations in

**Multiple Temporary Pointers**

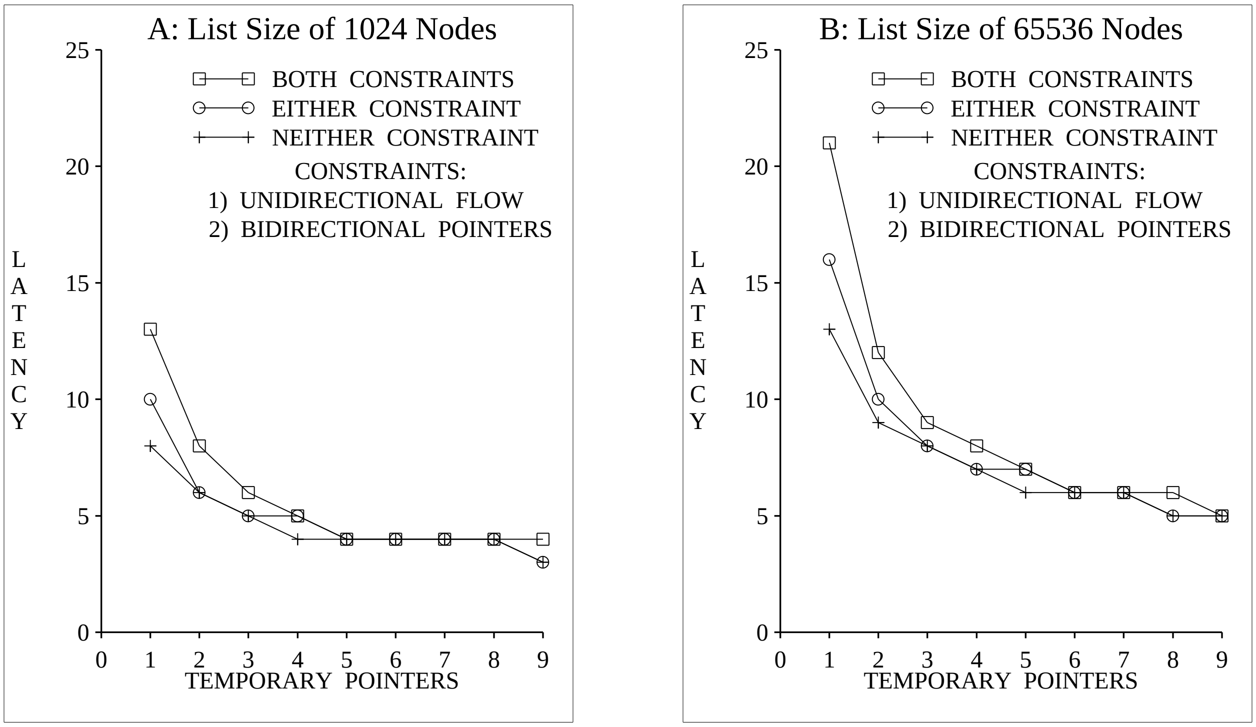



**Fig. 4.8:** These graphs show the potential latencies for multicasts with the given number of temporary pointers per node. The keys are ordered by the latencies for one temporary pointer per node. Results are shown for noncircular lists only. The two constraints are unidirectional information flow and bidirectional temporary pointers.

---

Section 4.2 are related by

$$p < \frac{p+\sqrt{D_2}}{2} < p+1 < \frac{p+1+\sqrt{D_1}}{2} < p+2. \tag{4.69}$$

By Equation 4.1, this means that the potential latency of one multicast is approximately $\dfrac{\log_2(N)}{\log_2(1+p\pm 1)}$ for large $N$, regardless of these two constraints and the constraint of noncircular lists. Therefore, the issues of circular lists, unidirectional information flow, and bidirectional temporary pointers are less important than the number of temporary pointers. For future work, this means that it is not useful to look for a different pointer structure or another pointer-creation algorithm that specifically endeavors to exploit these properties.

### 4.6.2. Single Comparisons

Figure 4.9A compares the potential latencies among themselves for one multicast with one temporary pointer. Clearly, circular lists (the dashed lines) give insignificant advantage over noncircular lists for potential latency. This indicates that it is not worth the extra complexity to add circularity to the standard SCI sharing lists. Adding just one of the constraints of either bidirectional temporary pointers or unidirectional information flow does not add much to the potential latency, about 23 percent for 64K nodes. However, adding both constraints can add up to 60 percent to the potential latency. This indicates for future work that some latency improvement might be gleaned by not enforcing both limitations. For example, both the symmetric and binary structures of Section 3.5.1 avoid the constraint of bidirectional temporary pointers. The latency of the binary structure is reduced by avoiding unidirectional information flow, whereas the latency of the symmetric structure is not harmed by this constraint. In other words, it is useful for the binary structure to avoid both constraints, but it is useful for the symmetric structure to avoid only one particular constraint.

Figure 4.9B compares the relevant potential latencies against the median-case latencies of the symmetric and binary pointer structures. BINARY INVALIDATE should be compared against ZERO CONSTRAINTS because it uses pointer reversing. However, the other three median-case lines should be compared against ONE CONSTRAINT because their information flows are unidirectional. This graph shows that the given structures are not close to optimal. However, these potential latencies consider only one multicast in isolation. They consider neither the conflict for minimizing the latencies of the multicasts from opposite directions nor the conflict for minimizing the latency to create the temporary pointers.

### 4.6.3. Double Multicast Latency

Figure 4.10 compares two times the potential latency of one multicast (the two lowest lines) to the potential latency of two opposite-rooted multicasts that must use the same temporary pointers (the two lines with triangles). The comparisons are made for both with and without pointer reversing (described in Section 3.5.2.2), where no pointer reversing implies unidirectional information flow for exactly one multicast. The slope of the dotted line is not smooth because 1) data points are plotted for every change in discrete latency and 2) the discrete latency

**Single Multicast Latency**

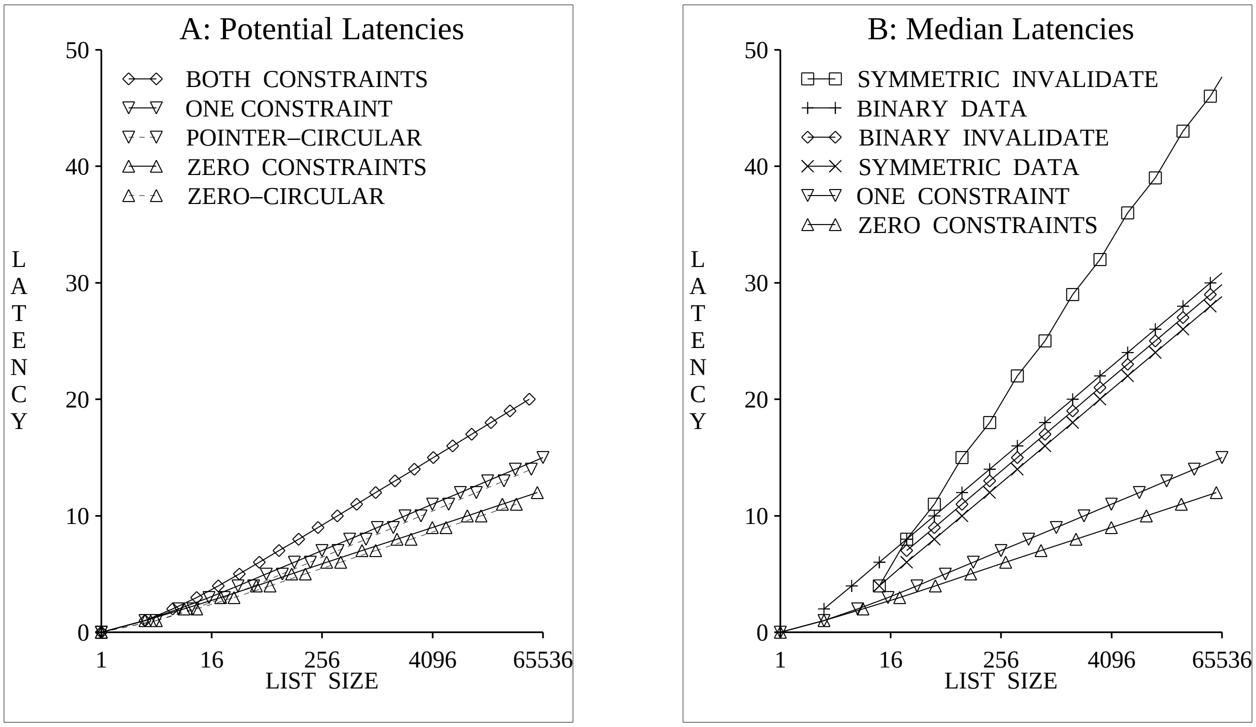



**Fig. 4.9:** These semi-log graphs show the latency for one multicast with one temporary pointer. The left graph shows the potential (best-case) latencies for every combination of constraints given in this section: bidirectional temporary pointers, unidirectional information flow, and non-circular lists. BOTH CONSTRAINTS, ONE CONSTRAINT, and ZERO CONSTRAINTS refer to the noncircular combinations of bidirectional temporary pointers and/or unidirectional information flow. POINTER–CIRCULAR refers to circular lists with bidirectional temporary pointers. ZERO–CIRCULAR is the same as ZERO CONSTRAINTS, except that it is for circular lists. Circular lists with unidirectional information flow are not shown because they are the same as for noncircular lists.

The right graph compares the best cases with the median cases from Section 3.6. SYMMETRIC and BINARY refer to the symmetric and binary structures respectively. DATA and INVALIDATE refer to the multicast for data and invalidate distribution respectively. BINARY INVALIDATE should be compared against ZERO CONSTRAINTS because it uses pointer reversing. However, the other three median-case lines should be compared against ONE CONSTRAINT because their information flows are unidirectional.

---

changes at different list sizes for the different modes of pointer reversing (with and without). Based on this graph, we conclude that using the same temporary pointers for data and invalidate distribution necessarily and significantly increases the sum of their potential latencies. In other words, the potential latencies shown in Figure 4.9 are not realizable.

**Double Multicast Latency**

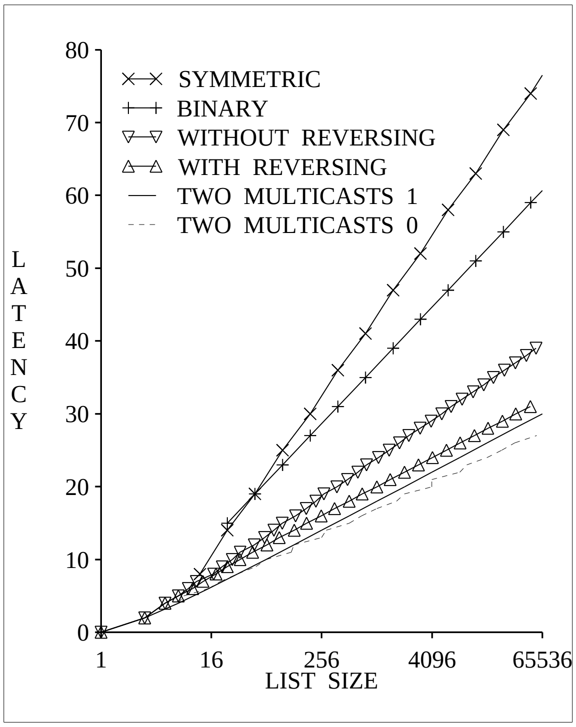



**Fig. 4.10:** This semi-log graphs compares the sum of the latencies for two multicasts that begin on opposite sides of a list with one temporary pointer per node. The key is ordered with respect to latency for the largest list sizes. SYMMETRIC is the sum of the data and invalidate latencies for the symmetric structure. BINARY is the same sum for the binary structure. The other four lines are potential latencies. WITHOUT REVERSING and WITH REVERSING pertain to the potential latencies for any structure, given that information flow is or is not unidirectional for invalidate distribution, respectively. Information distribution is always unidirectional for data distribution. These two lines should be compared to SYMMETRIC and BINARY respectively. The last two lines (the lowest) represent twice the potential latencies for one multicast. TWO MULTICASTS 1 represents twice the potential latency for one multicast with unidirectional information flow. TWO MULTICASTS 0 represents the sum of the two potential latencies for with and without unidirectional information flow, using a different pointer structure for each multicast.

---

Figure 4.10 also compares the potential and median-case latencies for two multicasts, given that the two multicasts use the same temporary pointers. To be fair, the binary structure should be compared to the potential latency with pointer reversing and the symmetric structure should be compared to the potential latency without pointer reversing. Similar to Figure 4.9B, this graph shows that the symmetric and binary structures are not optimal. However, this graph does not consider the time to create the temporary pointers.

### 4.6.4. Pointer Comparisons

Figure 4.11 compares the potential and median-case latencies for recursive doubling, the mechanism for creating temporary pointers. Since potential latency with request forwarding provides a lower bound on potential latency without request forwarding, we assume request forwarding for all data presented in this graph. Note that we compare the latencies for creating a pointer of a given distance rather than the latencies for creating all pointers in a list of a given size. This is because the best-case latency for a given list size is when there are no temporary pointers to create, which is zero latency. This result is not interesting because the multicast latencies would then be $\Theta(N)$. Comparing the latency with respect to distance shows that the binary structure is fair, about 35 percent more than the potential latency for 32K nodes, and that the symmetric structure is poor, about 84 percent more for 32K nodes. However, this comparison does not consider the multicast latency, which is the primary motivation for temporary pointers.

### 4.6.5. Creation Comparisons

Figure 4.12 compares the potential and median-case latencies for creation distribution. Again, request forwarding is assumed. This graph shows that the symmetric structure is poor, about 94 percent more than the potential latency for about 48K nodes. However, the binary structure is good, about 27 percent more. The binary structure is especially good when we note that the symmetric structure is optimized for a given list size, whereas the binary structure is good for all list sizes. Therefore, the binary structure can not be significantly improved. Furthermore, the optimal structure for creation distribution is not necessarily well suited for invalidate distribution. It is a topic for future work to find the optimal structure for the combination of creation distribution and invalidate distribution.

The lowest line in Figure 4.12 is the latency of one multicast in isolation. Note that the potential latency of this multicast is about 42 percent of the potential latency for creation distribution for about 32K nodes. Therefore, the latency of recursive doubling contributes

**Pointer-Creation Latency**

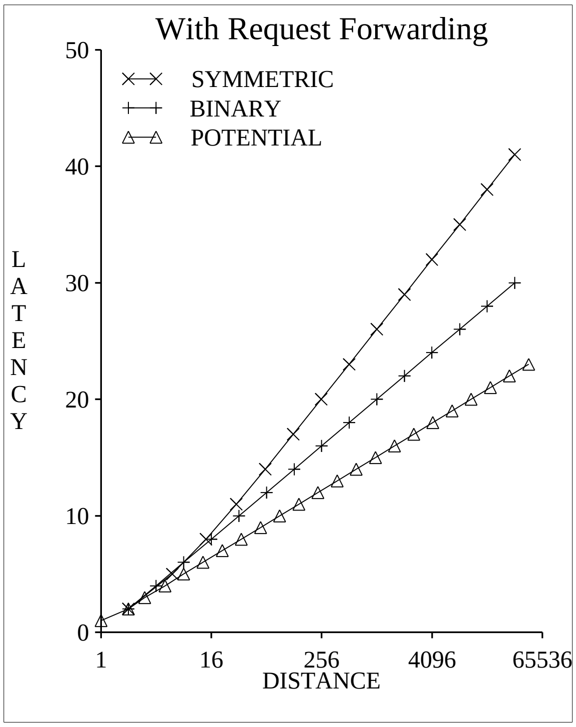



**Fig. 4.11:** This semi-log graph shows the latency for creating a temporary pointer of the given distance. Request forwarding, as described in Section 3.5.2.1, is assumed. SYMMETRIC and BINARY refer to the symmetric and binary structures from Section 3.5.1. POTENTIAL gives the potential latency for any pointer structure. The key is ordered with respect to latency for the largest distances.

---

significantly to the latency of reads and can not be easily overlapped with data distribution. In other words, the potential latencies shown in Figure 4.9 are not realizable, as previously noted, and the latency of creation distribution can not be ignored: it is impossible to construct a structure that is optimal for data distribution without paying a significant cost for its construction.

## 4.7. Summary

Cache-coherence protocols are used in multiprocessors to help maintain a consistent shared-memory model for programming. Some cache-coherence protocols, like SCI [Gust92a], use linked lists of nodes (cache lines) to maintain hardware cache coherence and avoid system-wide broadcasts. As processors read and write data, the cache-coherence protocols create lists,

**Creation-Distribution Latency**

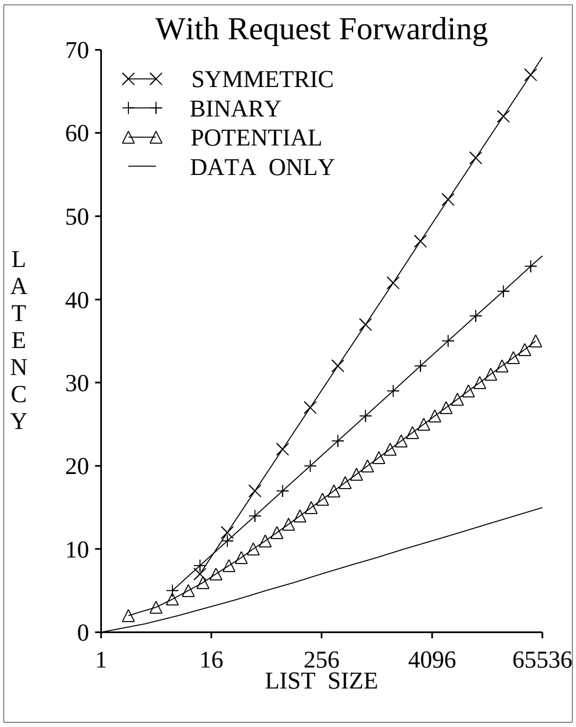



**Fig. 4.12:** This semi-log graph shows the latency for creation distribution with one temporary pointer per node. Request forwarding, as described in Section 3.5.2.1, is assumed. SYMMETRIC and BINARY refer to the symmetric and binary structures from Section 3.5.1. POTENTIAL gives the potential latency for any pointer structure. DATA ONLY gives the potential latency for one multicast in isolation, not considering the latency of recursive doubling. The key is ordered with respect to latency for the largest list sizes.

---

distribute data, and destroy lists of nodes. It is important that these list operations be efficient as the system size (number of caches) increases. However, the standard SCI cache-coherence protocol is not efficient for long lists. We believe that additional hardware support for these operations will significantly reduce the programming time required for good performance on systems with thousands of coherent caches.

In this chapter we study potential (lower bounds on) latency for the parts of the recursive-doubling extensions to SCI that depend on recursive doubling. We give the potential latencies for creating temporary pointers and then using them for two separate multicasts. We show that the issues of circular lists, unidirectional information flow, and bidirectional temporary pointers are less important for one multicast than the number of temporary pointers. Therefore, it is not useful for future work to consider pointer structures that exploit these features. We also show that the potential advantage of additional temporary pointers decreases rapidly as more temporary pointers per node are added. This result is reassuring since space and algorithmic complexity increase with the number of temporary pointers.

We examine the pairwise interactions between recursive doubling and the two required multicasts, assuming one temporary pointer per node. We show that using the same temporary pointers for data and invalidate distribution necessarily increases the latency of both. We review an algorithm for creating temporary pointers, called recursive doubling, and show that creation of temporary pointers contributes significantly to the latency of reads and can not be easily overlapped with data distribution. Therefore, in any future work, it is important to consider the latencies of all three aspects: finding a pointer structure that is optimal for one aspect means that it is not optimal for another.

Finally, we compare the potential latencies, derived in this chapter, with the median latencies for the symmetric and binary structures, derived in Section 3.6. Two of these comparisons are especially significant. First, we compare these two structures against the potential latencies for two opposite-rooted multicasts that use the same temporary pointers. These comparisons are not favorable, showing that a better pointer structure may exist. However, these comparisons do not consider the latency for pointer creation. Second, we compare these two structures against the potential latency for creation distribution. These comparisons show that the symmetric structure is about double its potential, indicating that there exists a better pointer structure (such as the binary structure). The binary structure is shown to be near its potential and that that significant improvements are impossible without changing the assumptions. Not only is the latency of the binary structure lower than the latency of the symmetric structure, but the binary structure is closer to its potential latency. This result is especially good in that potential latency and symmetric-structure latency are optimized for a given list size, whereas the latency of the binary structure is good for all list sizes.

# Chapter 5

# Tree Merging

## 5.1. Introduction

STEM (permuted acronym for Tree Merging Extensions to SCI), is a new coherence protocol that provides efficient support for multiple and coherent accesses to the same data. This binary-tree protocol allows parallel tree insertion and deletion, while maintaining a reasonably balanced tree for write invalidations, called *purges*. On average, single insertions and deletions are constant latency, while multiple insertions and deletions, as well as single purges, are logarithmic in the number of participating caches. STEM currently has the consensus of the SCI (Scalable-Coherent-Interface) working group [Gust91], officially IEEE P1596.2, for use in the Kiloprocessor Extensions to SCI[19]. The recursive-doubling extensions to SCI, outlined in Chapter 3, are not currently in vogue.

In addition to efficient reads, writes, and cache-line replacements, it is conceivable that massively parallel machines will need nonserial synchronization mechanisms, such as combining fetch-and-add [Ston84, AlGo89, SoSG89]. On one hand, it is unclear whether software techniques [YeTL87, GoVW89, YeTa90] can compete with hardware speeds. On the other hand, current hardware mechanisms [GGKM83, PBGH85] are expensive[20], save state in the network, and constrain the return path of responses. As a result, separate combining networks are used in the RP3 [PBGH85] to avoid slowing normal accesses. Instead, we use the simpler combining

---

[19]We can not resist mentioning the SCI-STEM (system) pun.

[20]Pfister and Norton [PfNo85] claim that combining increases switch cost by a factor of 6 to 32 in the RP3 [PBGH85] and it was not completed [SoSG89]. However, Dickey and Kenner [DiKe92] argue that the costs are more modest.

mechanism of the Scalable Coherent Interface (SCI) [JLGS90][21] to extend our coherence protocol and provide combining fetch-and-add capabilities. This is the first hardware mechanism for combining fetch-and-add with logarithmic latency, constant traffic per participating cache[22], no network state, no separate networks, and no limitations on response paths. Furthermore, our combining fetch-and-add is compatible with sequential consistency [Lamp79].

Cache-coherence protocols in massively parallel architectures should avoid deadlock. The networks for SCI [Gust92a] and Stanford DASH [LLGW92] ensure forward progress (no starvation of any processor and no deadlock) of two-phase request-response transactions. Many coherence protocols use other multi-phase transactions, which are more complex to implement and do not ensure forward progress. For example, a DASH directory forwards a writer's invalidate request to all sharing caches, which then respond directly to the writer. The DASH network (a dual 2-D mesh with wormhole routing) breaks deadlocks by using exponential backoff, but it can still have starvation problems. The STEM protocol ensures forward progress within the paradigm of request-response transactions, making it compatible with a wide range of networks, including SCI and dual meshes.

The first draft of STEM's specification is written in C [KeRi88] and testing of STEM's correctness is beginning. Some may argue that tree protocols are complex. However, there are no other viable coherence structures that have been proposed for thousands of processors; sequential execution of anything substantial is unacceptable. Furthermore, STEM's directory is very simple because it uses an SCI controller, which is a response-only controller that is ignorant of the tree protocol, requiring no counters, no traps, no request forwarding, four states, and only one pointer. Therefore, STEM's directory is simpler than previously proposed distributed-directory protocols, including ones of the $\mathrm{Dir}_0\mathrm{B}$ and $\mathrm{Dir}_1\mathrm{X}$ varieties [ArBa84, ASHH88, HLRW92]. We expect a first implementation of STEM to use programmable cache controllers to substantially reduce the design time for producing a working prototype and allow minor

---

[21] This published mechanism is compatible with the memory controller defined in SCI [Comm91]. However, it is not explicitly described in the standard because it is expected to be described in the extensions, highlighted by Section 1.4.2.

[22] The recursive-doubling solution of Lebeck and Sohi [Lebe91, LeSo92] uses $O(\log(N))$ traffic per participating cache for $N$ participating caches.

protocol changes after the hardware arrives. A programmable prototype will provide a platform for studying performance before completely implementing STEM in hardware.

We describe the protocol in sufficient detail to be simulated and provide a performance study to show that STEM is efficient for thousands of processors. Section 5.2 gives an overview of our protocol and defines some terms. Section 5.3 describes how to build a binary tree for reads so that the data can be distributed with logarithmic latency. We also describe how to extend tree merging to do combining fetch-and-add in hardware. Sections 5.4 and 5.5 describe how efficient cache-line replacements and writes are performed. Section 5.6 investigates numerous variations on the protocol and examines its performance. Section 5.7 concludes with a summary of this chapter.

## 5.2. Overview and Definitions

STEM is a write-invalidate protocol for cache coherence. The SCI working group (P1596.2) is currently exploring ways to extend STEM to support write-broadcast, but this is beyond our scope. We also limit our scope to the coherence of caches and memories that are connected to the network, assuming that the intracluster coherence for TLB and cache hierarchies can be managed by another mechanism, such as multilevel inclusion [Wils87, BaWa88] or pruning caches [GoWo88, Scot91].

The description of a write-invalidate protocol must specify how the set of sharing caches is stored, how caches are inserted, deleted, and purged with respect to the set, and how the notions of data validity and ownership are managed. In STEM, the structure for each line is distributed as a binary tree, where a *tag* in a cache stores 16-bit pointers[23] to at most three other tags in the set and the *directory* at memory stores a 16-bit pointer to one tag in the sharing set. The tag specified by the directory is called the *insertion point*, which is usually the most recently inserted tag. For simplicity, we consider one line in isolation, except where noted otherwise.

---

[23]All pointers are 16 bits in SCI, providing support for up to 64K clusters.

Tree creation is the most important part of the protocol because the height of the resulting tree affects the latency for reading, writing, and cache-line replacement and the latency of tree creation also contributes to the latency for reading. Tree-creation is complex because it must be flexible: it must create a new tree from an existing (possibly empty) tree and $N$ new insertions, where $N$ varies from 1 to the size of the system, while minimizing the resulting tree height, the latency, and the traffic. It must minimize these metrics when tags are inserted sequentially, in parallel, or in some degree in between. Each of the inserting tags must make its own decisions, consulting only with its current neighbors, at most three. The resulting tree must be dynamic in that additional insertions must be permitted to start at any time and that multiple deleting tags are also permitted to proceed in parallel with inserting tags that are not neighbors.

The constraint of parallel insertions prohibits starting each insertion at the root and traversing the tree to the leaves. Otherwise, the root would be a bottleneck. Furthermore, fetch-and-add requires nodes to be sorted according to their insertion order, not some arbitrary key, so starting each insertion at the root does not make sense, except for force of habit. Instead, beginning an insertion at a leaf makes sense because this is where the tag will most likely come to rest. Furthermore, if appending a single tag to a leaf creates an unbalanced tree, then the familiar AVL rotations [AdLa62, Knut73] can be used to balance the tree in $h$ rotations for tree-height $h$. Simultaneous appends to the same leaf are also possible, using a combining mechanism [JLGS90] to avoid the directory hot spot, and then the familiar AVL rotations can be used in parallel, with some care. The details of these cooperative rotations are explained in Section 5.3. Overall, this mechanism meets our performance objectives for sequential and parallel insertions, allowing insertions (and deletions) to start at any time.

Deletions must keep the tree intact. If a deleting tag has zero or one children, deletion is a simple matter of connecting the parent to the child, if any. However, if a deleting tag has two children, it must find a replacement (a leaf) and then swap itself with that replacement. Purging is accomplished by forwarding a message throughout the tree from the insertion point to the leaves and then collecting the responses from the leaves to the insertion point.

The height of the tree is balanced during insertions by the simplest AVL rotation of Adel’son-Vel’skiĭ and Landis [AdLa62, Knut73]. The AVL balancing property is that a node’s subtree heights differ by at most one. In contrast, our tree maintains a new balancing property, called *strictly growing*, which allows parallelization of the AVL rotations. We define this property after defining some terms, shown in Figure 5.1. Each tag has three pointers, which reference a *forward neighbor*, a *backward neighbor,* and a *downward neighbor*. The tree has two roots: the *deletion root* is the insertion point and the *insertion root* is found from the insertion point by traversing forward neighbors. Intuitively, the insertion root is usually the root of a spanning tree with minimal height, where the deletion root is one of its leaves. A tag on the inclusive path between the two roots is called a *list tag.* We define the *left child*, *parent* and *right child* to be respectively the forward neighbor, backward neighbor, and downward neighbor, with one exception. The definitions for left child and parent are swapped for each list tag in the context of the insertion root (see Figure 5.1). The *right height (downward height)* of a tag is the nondecreasing height of the tag's right subtree, not considering deletions in that subtree.

Finally, the tree is *strictly growing* if and only if the right height of every list tag, except for the insertion point[24], is strictly greater than the right height of its backward neighbor. Intuitively, this forms a list of trees with strictly increasing heights. This property is important in order to facilitate insertions that result in rotations between neighboring list tags of similar right heights. Rotating list tags of equal right heights maximizes the number of tags in a tree of given height, intuitively maximizing the balance of the tree. Since list tags have small right heights when they starting inserting, they should rotate with list tags near the leaves of the tree, not near the root of the tree. Therefore, the strictly-growing property helps keep the tree balanced, defining the insertion point to be one of the leaves. Unfortunately, this definition seems to double the latency for purging because insertions and purges both start at the insertion point, a leaf. However, our experiments with unbalanced alternatives lead us to believe that rotating list tags of unequal right heights would more than double the tree height, resulting in even worse purge latencies.

---

[24]The insertion point has no backward neighbor.

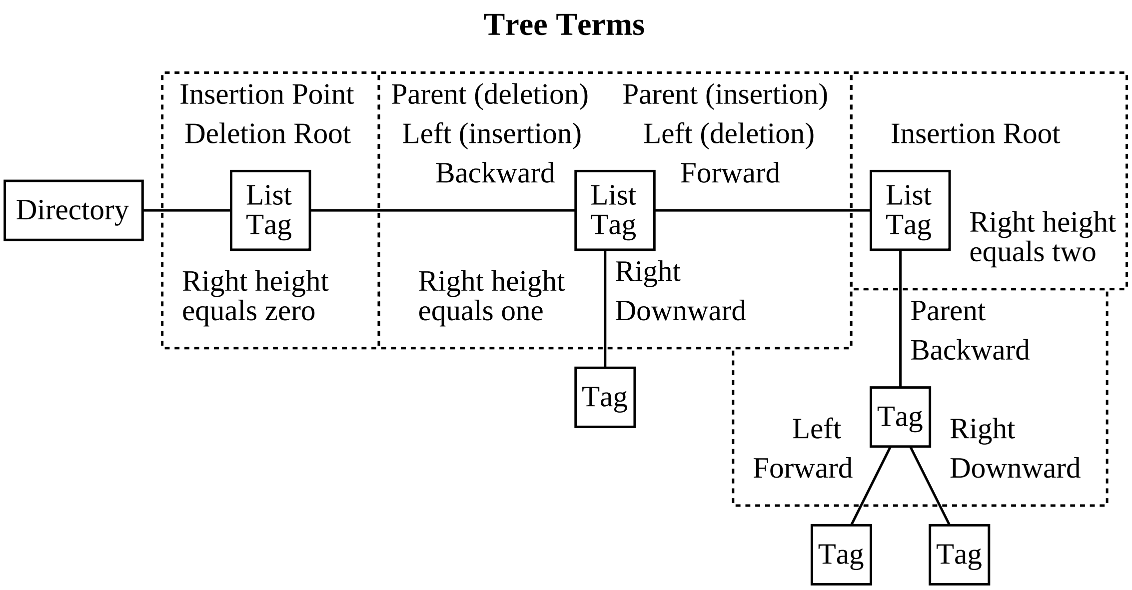



**Fig. 5.1:** This figure summarizes some terms defined in the text.

---

Continuing with the definitions, the directory has four states: HOME, GONE, FRESH, and WASH. HOME means that the sharing set is empty, otherwise it is not empty. HOME and FRESH each mean that the memory's data is valid, otherwise it is not valid. GONE and WASH both mean that the sharing set is not empty and the memory's data is not valid. WASH is a transition state between GONE and FRESH and it will be discussed later in Section 5.3.3. The directory controller only responds to requests; it never forwards requests and never traps to software, leaving most of the complexity to the cache controllers.

In addition to the three pointers, each cache tag stores a 58-bit address (assuming a 64-byte line) minus $x$ bits for set associativity, one 5-bit height, and a 10-bit state. A 5-bit height is sufficient for 64K nodes, as discussed in Section 5.6.5. The above tag fields may have a more compact encoding, but we will consider them separately for conceptual clarity. In order to manage the state-machine complexity, each state is a product of three orthogonal attributes: position, permission, and transition. Position describes the validity of each of the three pointers, indicating the tag's position in the tree. Permission indicates the validity of the data and access permissions, including coherence with memory and ownership. Transition indicates if the state is stable or what is happening otherwise. Since the number of nonstable transition attributes is very large, we group them into four phases (insertion, distribution, deletion, and purging) and limit their interactions. In particular, interphase cycles in the state machine do not exist and interphase arcs, which serve to order the phases as listed, are relatively few. Intraphase cycles are also relatively few, except for one-state cycles. Some nonstable transition attributes have additional information associated with them and this will be discussed when appropriate. Except for fetch-and-add, however, responding to requests from other caches does not require this additional information.

## 5.3. Tree Insertions

When a cache needs some data it inserts a tag into the appropriate tree. First, we describe the specific case for how single tags can be inserted one by one to build a perfectly balanced tree. We define *merging* as the sequence of rotations that are requested for one tag. Second, we show the general case for how multiple insertions can merge in parallel, called *tree merging,* while avoiding network congestion near the directory. In the degenerate case of one insertion, tree merging is the same as merging. Third, we show how data is distributed in parallel with tree merging. Fourth, we describe how our protocols can be extended to combine associative updates [GoLR83], like fetch-and-add, without complicating the network.

### 5.3.1. Sequential Insertions

If a single cache needs the data, then it inserts a tag into the tree, if any, and performs rotations until the tree becomes strictly growing. In particular, a list tag requests a rotate with its parent whenever it is possible that the tag and its parent violate the strictly growing property, including whenever the right height of the parent is unknown. Recall that the forward neighbor of a list tag is the parent during insertion. The right height of the parent is stored in the state machine (controller) that is shared by all tags in the same cache. This allocates space per state machine rather than space per tag, resulting in less overall storage space. One per state machine is sufficient because the right height of the parent is needed only during merging, during which time a state machine is allocated to the merging tag.

Figure 5.2 illustrates sequential merging for three tags. Assuming the tree is not empty, the new tag gets the insertion point from the directory and the directory points to the new tag. Then, the new tag requests a rotation with the previous tag. If the right heights obey the strictly

**Sequential Merging**

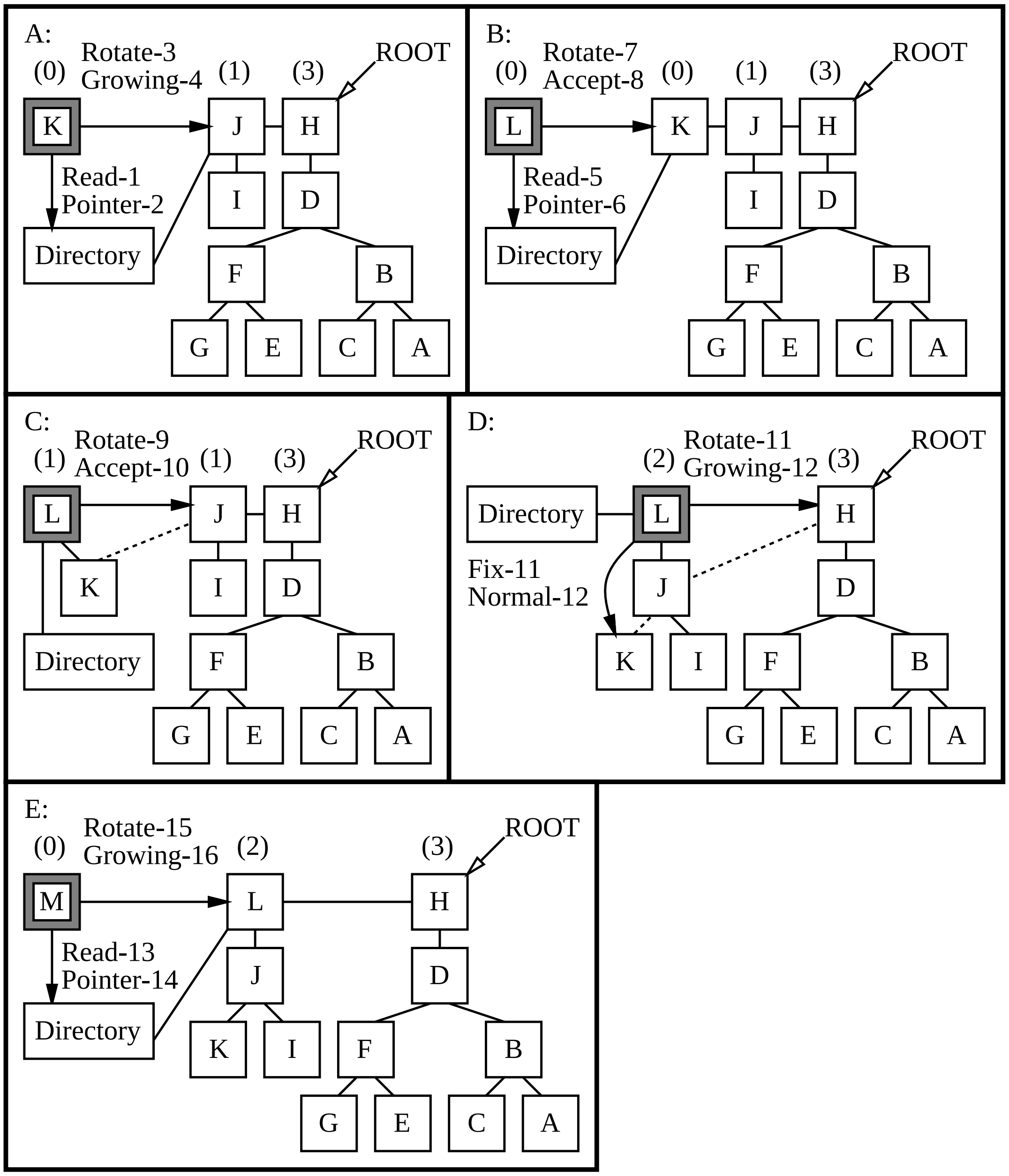



**Fig. 5.2:** These figures show sequential merging of three tags, K, L, and M. The merging of tags K and M are respectively shown in figures A and E and the merging of tag L is shown in consecutive figures B through D. The insertion root is labeled in each figure. Solid arrows represent request-response transactions, each labeled with a name and a time index per message. Dotted lines indicate pointers that require fixing. The number above each list tag represents that tag's right height.

growing property, then the rotate is rejected and merging for the new tag is done. Otherwise, the rotate is accepted and rotations are repeated until the strictly growing property is satisfied. A rotate request also fixes the backward pointer of each new parent. Starting after the second accepted rotation, the parent pointer of the tag’s previous left child is fixed by the message shown in Figure 5.2D. Note that the strictly growing property facilitates the generation of a binary tree with minimal height.

If a list tag finishes merging by obeying the strictly growing property, then it internally swaps the pointers to its parent and left child. When all merging is done, this swap effectively changes the tree’s root to be the deletion root. A root change is necessary to guarantee forward progress for deletions, as discussed later in Section 5.4.2, and it simplifies the later discussions (and implementations) of deletions and purges in Sections 5.4 and 5.5. However, if a rotate request later comes from a parent, which is otherwise impossible, then the pointers are logically unswapped before accepting the rotate.

### 5.3.2. Parallel Insertions

Multiple tags can be inserted in parallel. To avoid congestion and tree saturation [PfNo85] near the directory, directory requests may be combined in the network, resulting in a list of tags. Then, barring the exception discussed later in Section 5.3.2.2, each tag requests a rotate until the strictly growing property is locally satisfied.

#### 5.3.2.1. Request Combining

Each request contains a *combinable bit* that indicates whether the request should be combined in the face of congestion. For this discussion of combining, we assume that all read requests have set the combinable bit and that all requests are reads of the same line. Then, multiple insertion requests to the directory are combined in the network, as proposed by James et al. [JLGS90] and as shown in Figure 5.3. This creates a list of tags, which is later merged into a list of trees with strictly increasing heights. A combined request contains pointers to the two endpoint tags *(head* and *tail)* of a *segment*, which is a fragment of list tags. A noncombined request can be viewed as a combined request where the segment endpoints are equal. When two requests are combined, they are transformed into one response and one new request. The response is sent to the tail (B)

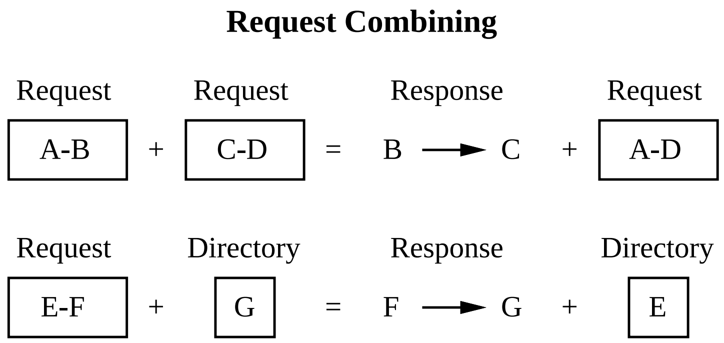



**Fig. 5.3:** This figure shows request combining. The head (tail) of each segment is shown on the left (right).

---

of one segment and this response contains a pointer to the head (C) of the other segment. The new request contains pointers to the remaining segment endpoints (A and D). This combining takes place either at the queue in front of the directory or in other network queues that are part of the switches between SCI rings.

When a request reaches the directory, the head (E) of the segment is swapped with the insertion point (G) and a response, containing the old insertion point (G), is sent to the tail (F) of the segment. All combining responses are as if each request is satisfied by a directory in state GONE, one response per request. This combining does not save any state in the network and does not place limitations on the response path, unlike the NYU Ultracomputer [GGKM83] and the IBM RP3 [PBGH85, PfNo85]. Instead, SCI's combining accepts two requests and outputs one combined request and one response.

#### 5.3.2.2. Parallel Rotations

After combining, the tags in the list merge in parallel, as shown in Figure 5.4. Each tag requests a rotate with its parent, retrying if its parent has not yet received a directory response. Although the first tag (E) may find that the strictly growing property is satisfied, the other rotate requests are rejected because all of the new tags are busy. Continuing to request rotates in this way would result in $\Omega(N / \log(N))$ latency because the maximum amount of parallelism would be limited by the right height of the insertion root.

**Parallel Merge Problem**

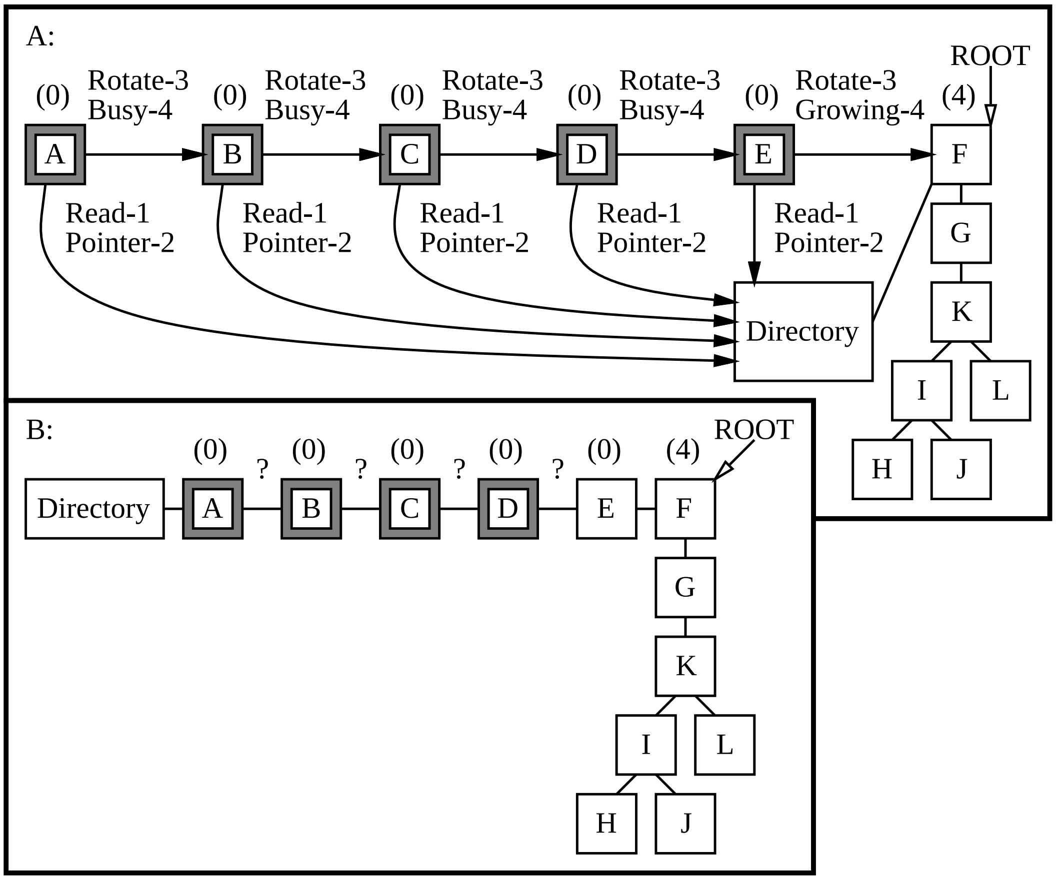



**Fig. 5.4:** These figures illustrate the problem for parallel merging after combining is done. The insertion root is labeled in each figure. Solid arrows represent request-response transactions, each labeled with a name and a time index per message. The number above each list tag represents that tag's right height.

In order to break this sequential waiting, we introduce randomness. Normally, a list tag requests a rotate with its parent whenever it is possible that the tag and its parent violate the strictly growing property. However, whenever the right heights of a list tag, its parent, and the tag's left child are all the same, then the tag requests a rotate with 50 percent probability, a coin flip. This effectively shatters the long list into small fragments that can rotate in parallel, although the resulting tree may cease to have minimal height[25]. For the other 50 percent, the tag sends a request that updates the parent's notion of the tag's right height and returns the parent's current right height. This update effectively inserts a delay before the next coin flip while preventing right heights from becoming very stale. Note that the delaying tag can accept a rotate from its left child (and be done merging) because it has not committed itself to rotate with its parent. In order to reduce the implementation costs, the right height of the left child is stored as an equal bit in the tag's state, rather than as another 5-bit tag field.

A fair coin (percentage of 50) is chosen to maximize the probability that any given node will be able to rotate with either its forward or backward neighbor. It is possible that a pseudo-random number generator may be biased. However, a biased coin of 60-40 will reduce the rotation probability from 50 percent to only 48 percent.

The example of Figure 5.4 is continued in Figure 5.5. In Figure 5.5A, the list tag with no left child (A) requests a rotate, but the remaining tags (B, C, and D) flip coins. B chooses to delay and C and D each choose to request a rotate. Since B is waiting, it accepts A's request to rotate and B becomes done. Since E is done, D's request is also accepted. However, C's request is rejected because D is busy. In Figure 5.5B, C flips a coin, choosing to delay, because it has no knowledge of the other rotates. D and A each request a rotate because the parent's right height is unknown. This request also fixes the stale left-child pointer in the parent. D discovers that it is done merging, but A's rotate is accepted and C becomes done. In Figure 5.5C, A requests a rotate with D and a pointer fix for D at the same time as it requests a pointer fix for B. D accepts the rotate because the strictly growing property is still violated. Finally, in Figure 5.5D, A requests a rotate with F and a pointer fix for F at the same time as it requests a pointer fix for C. Since the strictly growing property is obeyed, A is done merging and tree merging is done. If a right height reaches 31 during merging, then the offending subtree is purged, as described later in Section 5.5. Although a right height of 31 is theoretically possible, the probability is vanishingly small, as argued in Section 5.6.5. Note that the global notion of strictly growing is not needed, so it is not computed.

---

[25]Some deterministic solutions are currently being studied by the P1596.2 working group, showing similar expected-case performance and adding complexity to the combining mechanism.

**Parallel Merge Solution**

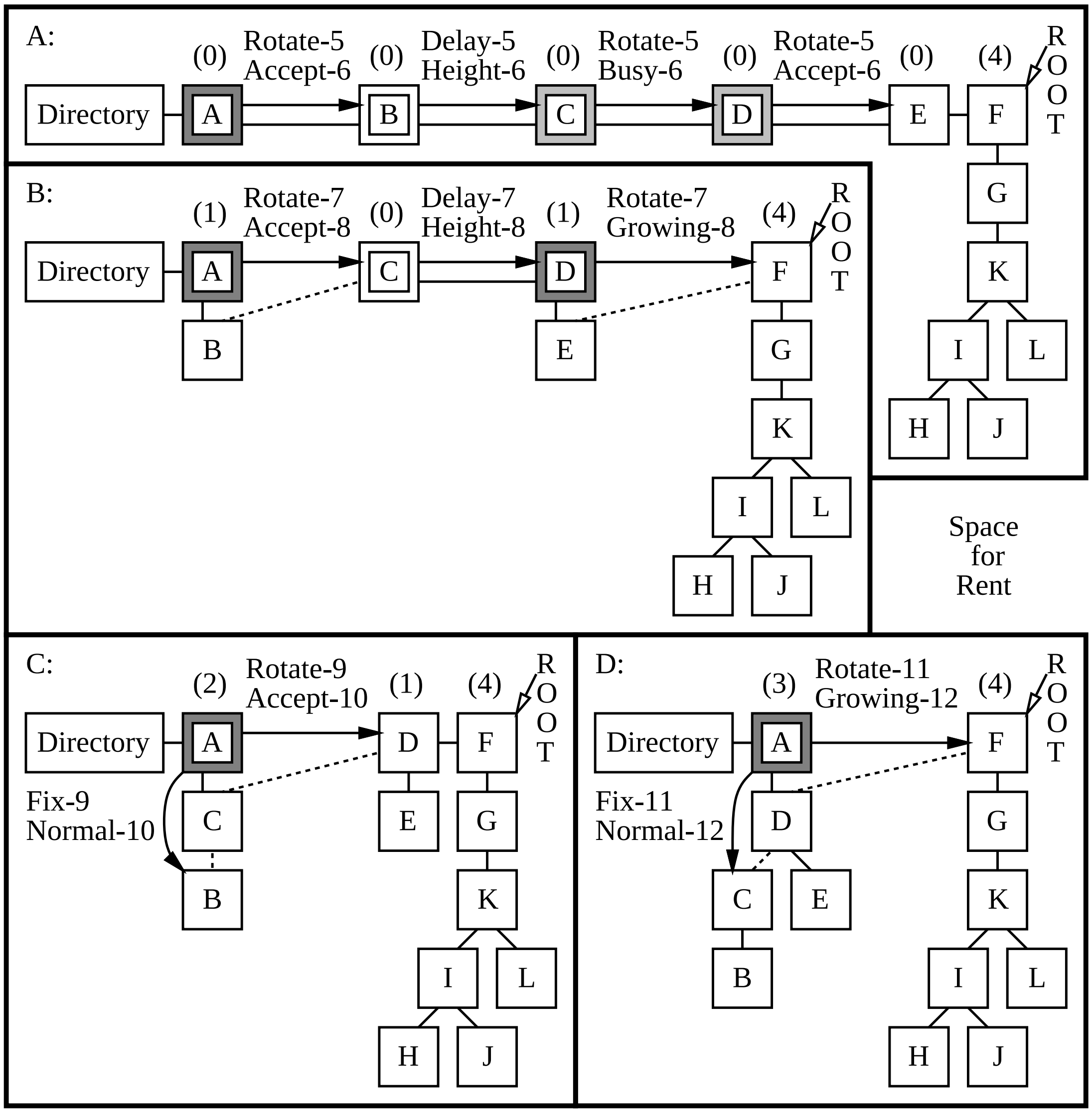



**Fig. 5.5:** These figures illustrate the solution for parallel merging after combining is done. The insertion root is labeled in each figure. Solid arrows represent request-response transactions, each labeled with a name and a time index per message. Dotted lines indicate pointers that require fixing. The number above each list tag represents that tag's right height.

---

### 5.3.2.3. Forward Progress

The proof-sketch of forward progress for asynchronous tree merging is as follows. Consider one tag that has not finished merging, meaning that the tag is a list tag, the strictly growing property is not satisfied, and the tag is not the insertion root. This tag is either waiting for a response to its last rotate request or it is waiting for a rotate request from its backward neighbor. Without loss of generality, we consider only these two conditions. The tag may also be executing an atomic decision between messages, but this will result in one of the given conditions in bounded time.

First, consider the case when the tag is waiting for a response to its rotate request. If the request is accepted, then the distance to the insertion root is decreased by one, called the *primary distance.* If the request is rejected, then either the forward neighbor is busy or the strictly growing property is satisfied and the tag is done. If the forward neighbor is busy, then it too must be requesting a rotate. However, this chain of rotate requests must stop at a finished tag, the distance to which is called the *secondary distance;* note that the secondary distance is bounded by the primary distance. If the last request of the chain is accepted, then the primary distance is decreased by one. If it is rejected, then another list tag is finished and the secondary distance is decreased by one. Therefore, a bounded time after the original rotate request, the time of which depends on the variable cache-controller and network delays, either the primary or secondary distance must be decreased by one. After one of the distances is decreased, a later rotate request from the tag will again reduce one of the distances. This can not always be the secondary distance, implying that the primary distance is eventually decreased to zero, unless the tag finishes merging. Either way, the tag finishes merging. In summary of the first case, a bounded number of rotate requests makes either primary or secondary progress towards the finishing of merging.

Second, consider the case when the tag is waiting for a rotate request from its backward neighbor. Without loss of generality, we assume that the coin flip always causes the tag to wait; otherwise, progress towards finishing is made, from the first case. Since the waiting tag has flipped a coin, the tag's right height is the same as its neighbors' right heights. Consider the length of the chain of tags with equal heights, starting at the given tag and going backwards. The chain length at any point in time is bounded by the system size (maximum of 64K in SCI). The end tag of this chain must request a rotate because the strictly growing property is violated and the end tag's backward neighbor, if any, is not part of the chain, by definition. The only reason that the end tag's rotate can be rejected is if its forward neighbor requests a rotate. Likewise, that rotate can be rejected only if the next tag requests a rotate, and so on. Ultimately, one of the rotate requests in the chain must be accepted because the original tag is waiting to accept one. This results in a shorter chain, where the right height of the end tag's backward neighbor is strictly greater than the right height of the end tag. After this point, it is impossible for the chain size to increase because, without a successful rotate, it is impossible for the end tag's right height to become the same as that of its backward neighbor.

It is possible to increase the chain length of the very first chain: new insertions and/or rotations can increase the length provided that no tag in the chain, including the given tag, accepts a rotate when it comes. However, this implies that the given tag requests a rotate with its forward neighbor, giving progress by the first case in the previous paragraph. Also note that tags already on the chain are each committed to participate in a least one rotate before leaving the chain because a node can not delete itself before insertion and data distribution are completed. So, there can be no new end tag after the maximum chain length is reached. Therefore, progress is definite and each merging tag is guaranteed to finish.

### 5.3.3. Data Distribution

While tags merge into the tree, the data is distributed. If the directory is in state HOME or FRESH, then the data is returned in the directory's response. If the data is not returned in the directory's response, then the data is acquired from another tag in the tree. In general, a tag forwards the data to its neighbors as soon as the data arrives. This forwarding replaces SCI's spin waiting and reduces latency. Also, the data is returned in every response to a rotate request from a needy tag, which is necessary for single insertions. If a tag finishes merging before receiving the data, then the tag waits for the data while retaining the minimal resources required to forward the data[26]. SCI already provides a response timer to detect lost packets and this same timer is used to prevent infinite waiting if an ancestor dies (hardware failure); note that the timer detects system failures and is not required for normal operation. If the data arrives during merging and there are not enough resources to forward the data in parallel with merging, then this condition is

[26]These are the same required resources to request another rotate, which can be reserved without deadlock.

recorded in the tag state so that the data can be forwarded when the merging is done. As an optimization, the downward fix request for merging can be coupled with data distribution, provided that the associated rotate commits before the data arrives. For example, tag C in Figure 5.5C can fix tag B's pointer when it sends the data, eliminating the need for the extra request from tag A.

STEM distinguishes between fresh and nonfresh insertions, where a *fresh* insert desires read-only data. If the directory request is fresh, then HOME is changed to FRESH and other states are unchanged. Otherwise, all states are changed to GONE. If the last tree insertion is fresh and the tag did not receive read-only data, then it is important to write-back the data so that future fresh insertions can get read-only data from the memory. We extend SCI's write-back protocol to work with trees. In the simple case, there are no more insertions during the write-back, leaving the directory in state FRESH. However, if there are more insertions, then we must guard against the possibility that one of these insertions requested ownership to change the data. In this case, the write-back partially completes, leaving the directory in state WASH, and the directory responds to the write-back request with a pointer to the new insertion point, called the *write-back pointer*.

WASH is a transition state between GONE and FRESH. It is required for correct operation of nonatomic write-backs of data and it is specified in the SCI standard [Comm91]. If there are additional insertions after the write-back, the write-back can not immediately change the state from GONE to FRESH. This is because one of the insertions may be a writer, having permission to change the data. If that writer changes the data and then later insertions are requested, then the later insertions will get stale data. Therefore, the write-back requires an intermediate state between GONE and FRESH, called WASH, that persists until it can be verified that there are no intermediate writers. State WASH asserts that 1) no nonfresh insertions have occurred since the receipt of the write-back request, time X, and 2) it is unknown whether a nonfresh insert occurred between time X and the time of the fresh-insert request from the same source. The object of the write-back protocol is to remove the unknown in part 2 of this assertion.

In order to change the directory's state from WASH to FRESH, the tags cooperate to determine if there are any writers. As the read-only data propagates to the newly inserted caches, the write-back pointer is included. If the data passes through a nonfresh inserter, then the data is no longer read-only and the write-back pointer is discarded; the directory may remain in state WASH for a long time. If a fresh inserter receives read-only data with a write-back pointer to itself, it notifies the directory that part two of the above assertion is safe. It is safe because all of the intermediate insertions were fresh. In the mean time, the directory responds to all fresh requests as if in state GONE and changes to state GONE for any request that is not fresh. If the directory receives a write-back notification while still in state WASH, the write-back is completed and the directory's state changes to FRESH. Otherwise, the write-back is not completed, possibly leaving the directory in state WASH until the next write-back attempt, if any.

### 5.3.4. Fetch-and-Add

Tree merging and data distribution can be extended to support combining of fetch-and-add and other associative updates [GoLR83] (including concurrent writes), without extending the existing network combining. During merging, each tag accumulates the operands of its right subtree plus its own, a simple addition for each rotation. During data distribution, each tag normally sends the data plus the accumulation to its left subtree and sends the data to the remaining neighbor. However, if the data arrives from the left child (due to a list-tag pointer swap), then the data plus accumulate is sent to the parent instead. The logical ordering implied by this combining fetch-and-add is an in-order traversal of the tree from right to left, using the insertion root as a reference. Figure 5.6 gives an example of the distribution of fetch-and-add, starting at the bottom right, where tag $x$ adds $2^x$ to an integer that starts at zero. Each triple shows the tag's original operand, its right-subtree accumulation, and the data plus accumulation.

The value returned by a fetch-and-add operation is the data plus the accumulation minus the tag's original operand[27]. If the tag is the insertion point, then the data plus accumulate is stored in the cache line. Otherwise, the data is marked invalid and the tree is deleted, leaves first. To guarantee sequential consistency [Lamp79], 1) each fetch-and-add purges all

[27]If replace-add [GoLR83] is implemented, the original operand need not be remembered. If fetch-and-increment is implemented, the original operand is always one.

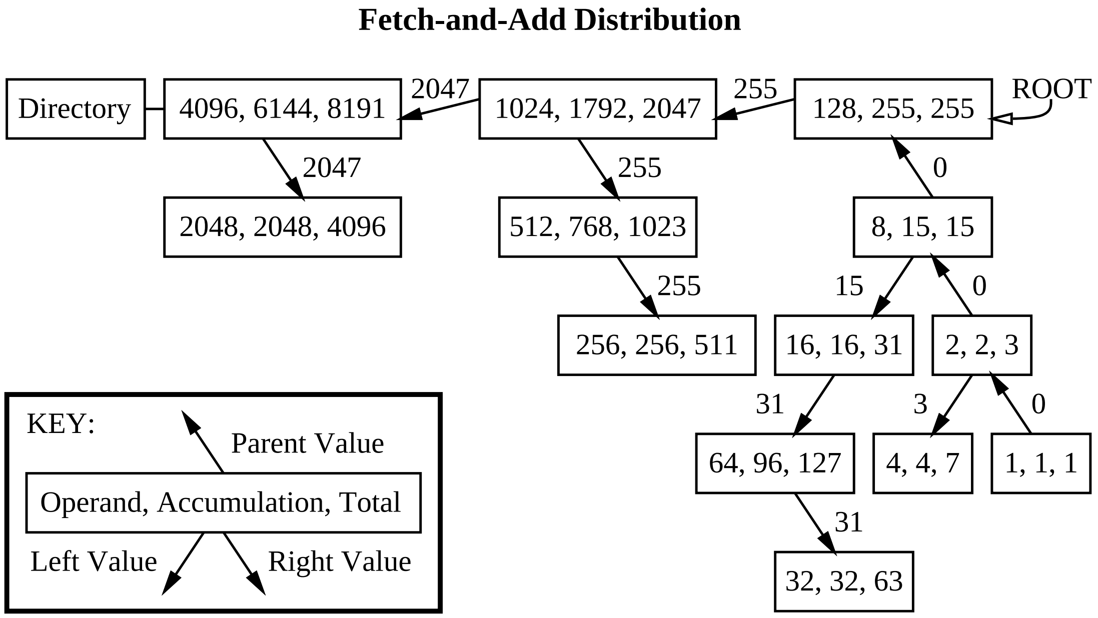



**Fig. 5.6:** This figure shows the distribution of fetch-and-add values. Each tag has a triple that represents the tag's operand, right-subtree accumulation, and its final data plus accumulate. The black arrows are labeled by the data values that are propagated, starting at zero. To aid understanding, each of the operands is a different power of two.

---

non-fetch-and-add copies before it releases the data (to processors and other caches) and 2) each processor blocks for each of its own insertions, purges, and fetch-and-adds to be completed[28]. Sequential consistency is guaranteed (if all processors issue memory accesses in program order) because the protocol defines a partial ordering on all memory accesses such that there exists a global ordering that is consistent with program order. Another proof-sketch is that our combining fetch-and-add meets the sufficient conditions defined by Scheurich and Dubois [ScDu87][29]. For hardware simplicity, we expect combining fetch-and-add to be defined on the first 8-byte integer in a line. However, the mechanism can be extended to combine fetch-and-adds on

[28]It is sufficient to wait for valid data with insertions and nonpurging fetch-and-adds.

[29](A) All processors issue memory accesses in program order. (B) If a processor issues a STORE, then the processor may not issue another memory access until the value written has become *accessible* by *all other processors*. (C) If a processor issues a LOAD, then the processor may not issue another memory access until the value which is to be read has both been *bound* to the LOAD operation and become accessible to all other processors.

multiple nonoverlapping integers in the same line by assuming that unspecified operands are zero.

### 5.4. Tree Deletions

When there is a cache miss and there are no free cache lines that can be used, it becomes necessary to delete a tag from a tree. Recall that the insertion point is the root after merging. If the chosen *victim* in the cache has zero or one children, then SCI's deletion protocol can be used, connecting the child (if any) to the parent (or directory). However, if a tag has two children, then the deletion is more complicated. In this case, a tag finds a replacement for itself to keep the tree structure intact. The victim walks down the tree to a leaf tag, detaches the leaf from its parent, and finally replaces itself with the leaf tag. Note that over half of the tags in any binary tree can use SCI's deletion protocol. Also note that the average distance to a leaf is less than one, assuming distance zero for all tags with zero or one children.

One might think that it would be more efficient to detach the first tag with less than two children. However, this complicates the protocol since detaching a leaf must still be supported and detaching a node with one child is much more involved. Furthermore, in a balanced tree, a tag with only one child implies that its child is a leaf and taking the parent of a leaf would be a small performance improvement.

#### 5.4.1. Walkdown

In order to find a replacement for itself, a tag traverses the tree until a leaf is found, as shown in Figure 5.7. The victim (H) requests a child of its right child (K). This right child (K) responds with a pointer to its only or left child (J). Then, the victim (H) requests a child of the given grandchild (J), which returns its only or left child (I). These requests are repeated until the leaf tag is reached and locked. This repetition of requests is called a *walkdown.* Note that only the victim chooses a right child. For performance (but not correctness) of multiple deletions, it is interesting to note that 1) no two walkdown paths traverse the same tag and 2) no two tags try to lock the same leaf.

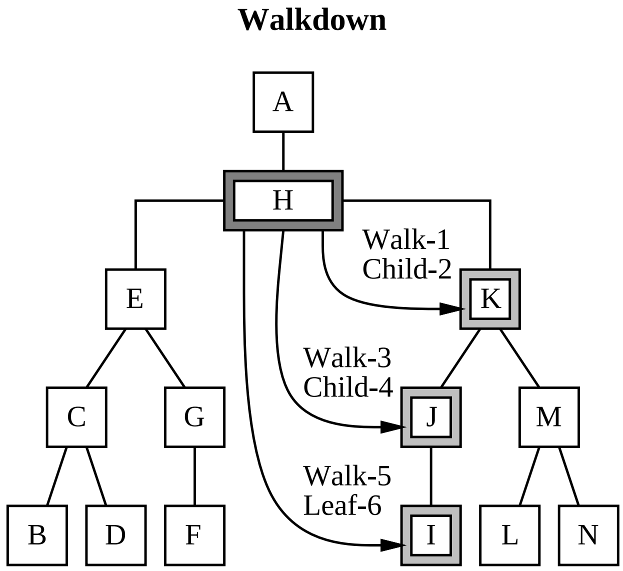



**Fig. 5.7:** This figure shows the walkdown to find a leaf that can be used for a swap. Arrows represent request-response transactions, each labeled with a name and a time index per message.

---

### 5.4.2. Tag Swap

After a leaf tag has been locked with the walkdown, the leaf is detached and swapped with the victim, as shown in Figure 5.8. This is done in three nonoverlapping steps. First, the leaf (I) is detached from its parent (J) by a request from the victim (H) directed to the leaf's parent (J). Second, the two children (E and K) of the victim (H) are informed in parallel that their new parent is the leaf tag (I). Also in parallel, the parent (A) of the victim (H) is informed that its new child is the leaf tag (I). Third, the leaf tag (I) is informed of its new parent (A) and children (E and K). The order of these three nonoverlapping steps is important to guarantee forward progress in the event that several neighboring tags are trying to delete themselves at the same time. In particular, children have priority over parents so that every swap is guaranteed to complete after the leaf has been successfully detached. Since the insertion point is the deletion root (a parent, not a left child), repeated tree insertions can not starve a deletion.

### 5.4.3. Correctness

If a tag on the walkdown path tries to delete itself while the victim is doing the walkdown, then correctness requires special care. It is possible, for example in Figure 5.7, for tag J to respond

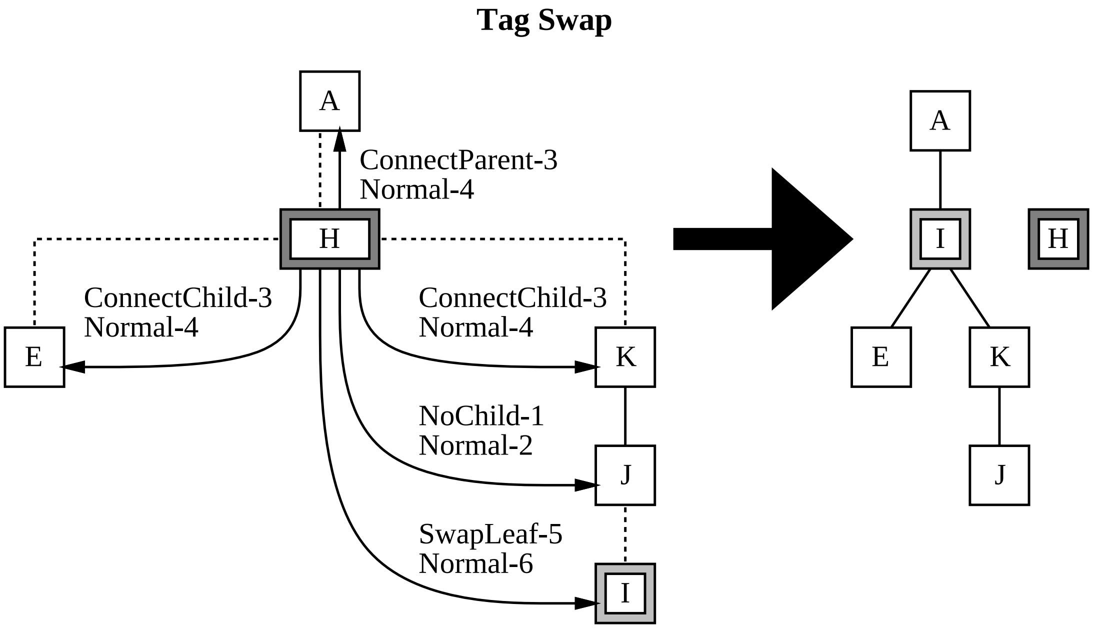



**Fig. 5.8:** This figure shows the swap of a leaf and a victim. Dotted lines indicate pointers that are modified. Small arrows represent request-response transactions, each labeled with a name and a time index per message.

---

with a pointer to tag I and then for tag I to delete itself before it receives the next walkdown request from tag H. When the walkdown request finally reaches tag I, it is possible for tag I to be in any legal state! The problem is compounded if a string of tags is deleted and reappears as the same string somewhere else, possibly in the same tree. To guarantee correctness and forward progress, a set of sufficient conditions are checked. If any condition fails, the walkdown is restarted.

To avoid mixing of tags from different trees, the memory address of each tag on the walkdown path must be the same as the victim's memory address. It is also possible to have problems if a deleted tag re-enters the same tree. In this case, however, it is permissible to swap with any leaf tag as long as it is done correctly. To avoid interactions with other protocol phases, the state of each tag on the walkdown path must be either stable or in the deletion phase. If a detach request is sent and the parent of the leaf is not stable or in a deletion phase, then the detach request is rejected, the leaf is re-attached to its parent, and the walkdown is restarted. To guarantee forward progress, the right height of each tag on the walkdown path must be strictly decreasing[30] and the leaf tag must not be already locked for swapping. If any condition fails, the walkdown is restarted. If the leaf is already locked by the wrong tag, the leaf will be detached after a finite number of restarts, reducing the size of the victim's right subtree. In all other cases, every restart implies a reduction in the right subtree's size. Therefore, because a starving victim can eventually use SCI's deletion protocol, forward progress is guaranteed.

### 5.4.4. Optimization

Walkdown restart has a performance problem in that forward progress is slow. This problem is exacerbated when nearly all of the tags try to delete themselves at approximately the same time. This case can frequently occur after synchronization points, if new information is cached and many caches choose the same line for replacement. Therefore, we add the following optimization. If the right child of the victim is also trying to delete itself, then the victim waits for its right child to complete. Completion is detected when the right child connects the victim to a new tag. If all tags simultaneously delete themselves from the tree, then this optimization efficiently deletes the entire tree, leaves first.

## 5.5. Tree Purging

When a cache needs to change some data, all other copies of that line are invalidated and removed from the tree, called *purging*. Recall that the insertion point is the root after merging. First, the tag becomes the insertion point of the tree, using the deletion protocol and/or insert, if necessary. However, no rotations are requested. Second, the tag purges its subtree(s) in parallel. The purge is forwarded to all the leaves and then reflected back to the purger, removing all tags from the tree except the purger. Depending on the memory consistency model, the purger may be allowed to continue computation before purging completes. If sequential consistency is maintained by the hardware [ScDu87], then the purger must wait for completion.

---

[30]We thank Stein Gjessing for noticing this forward-progress problem. If tags are continually deleting themselves and re-entering, what is the guarantee that a leaf will eventually be found? We thank Alasdair Rawsthorne for the given solution.

### 5.5.1. Basic Mechanism

Figure 5.9 shows an example of purging an entire tree. When a tag (I) with two children receives a purge request, it forwards the request to both children (H and K) and responds to its parent (E). All three messages are sent in parallel. When both subtrees (H and K) are done purging, the tag (I) marks itself invalid and notifies its parent (E). If a tag (D) has one child, then it marks itself invalid and responds with a pointer to its only child (C). Its parent (B) then repeats its request to the new child (C). If a tag has zero children (C), it marks itself invalid and responds that it is purged. When the purger (A) has no children, the purge is done.

### 5.5.2. Deadlock Avoidance

In general, request forwarding with SCI networks can cause deadlock due to finite resources, such as request queues. Making reservations for these resources can also lead to deadlock. In August of 1991, however, David V. James [Gust91] suggested a simple solution to this problem for purging, as described next. If a tag with two children receives a purge request and if that tag does not have the resources to forward the request, then the tag appends itself to an internal purge queue. This queue is composed of different cache lines in the same cache that are waiting to finish purging. Since data is no longer needed after a purge request has been accepted, the data space in the cache can be used to implement a linked-list FIFO. Only a little extra space per cache is required for pointers to the front and back of the queue. After appending itself to the internal purge queue, the tag can respond normally to the purge request, indicating that the parent can wait. Later, when resources become available, the first task on the purge queue is continued. In order to avoid circular wait and deadlock, it is necessary to append waiting purgers on the purge queue when there are other purgers that need to forward requests[31]. Then, resources are guaranteed eventually, provided that new cache requests, such as new processor commands, are not allowed to starve the requests on the purge queue. Note that we expect resources to be available in the common case, so the correctness of purging is the primary concern here, not performance.

[31]We discovered and solved this particular deadlock problem, thereby improving James's solution.

**Tree Purging**

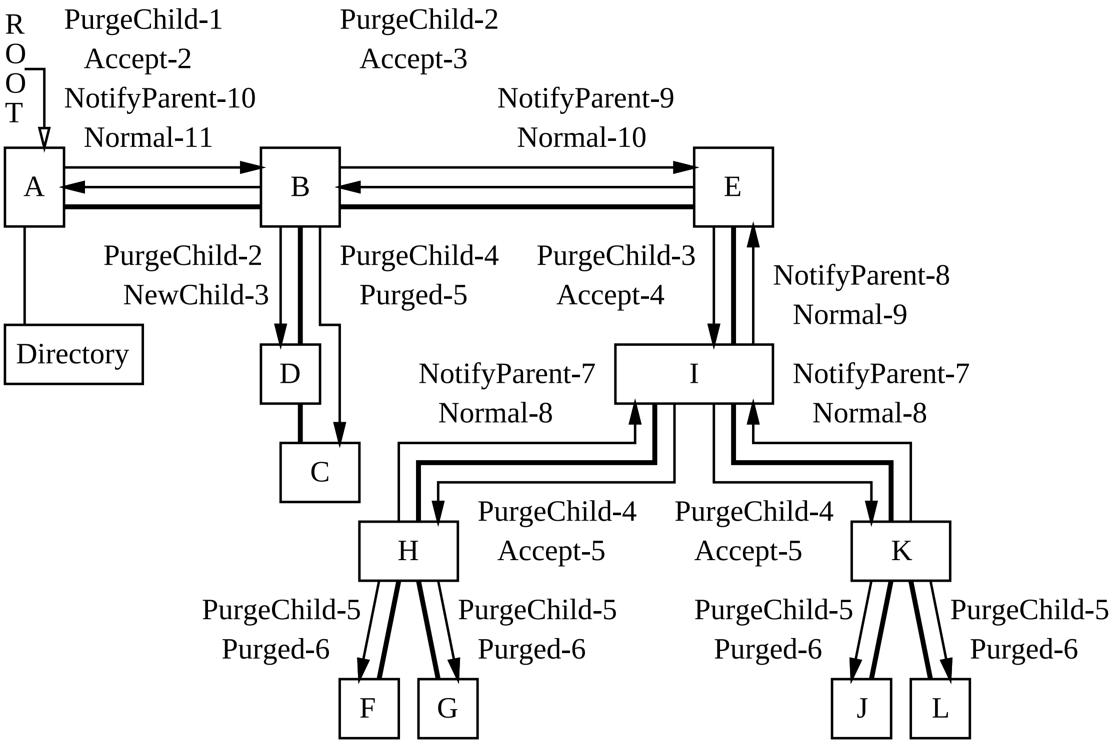



**Fig. 5.9:** This figure shows the purging of an entire tree. The deletion root is labeled on the left. Arrows represent request-response transactions, each labeled with a name and a time index per message.

---

Lockup-free caches [Krof81] and/or allowing multiple outstanding requests are useful for good performance, but they are not required for correctness[32]. Any time that the protocols want to send requests in parallel, they can be sent sequentially. For correctness, caches need only be able to accept purge requests into the internal purge queue so that the purges can be forwarded later.

[32]However, providing sequentially consistent hardware [ScDu87] with lockup-free caches seems difficult.

### 5.5.3. Protocol Interactions

Concerning interactions with other protocol phases, purges have lowest priority. In particular, purge requests are rejected by inserting and deleting tags so that the requests are repeated. On the other hand, a purger always accepts a rotate request as if it has no children, but does not release the data until the purge completes.

Furthermore, these tree-merging extensions to SCI are careful to interoperate correctly with the standard SCI protocol. When an SCI tag requests a purge, the responder forwards the request as usual, but rejects the request so that it is repeated. The responding tag continues to reject the purge requests until it has no children. Then, it responds to the next SCI request in the same way as an SCI tail. A tag in a nonstable state normally rejects an SCI purge until it is prepared to start purging. However, a deleting tag connects itself as the leaf's parent and then accepts the purge. This is needed because an SCI tag rejects a connect-parent request while purging. Note that in these interactions, we are primarily concerned with correctness, not performance.

## 5.6. Protocol Variations

In this section, we investigate two variations of STEM and compare each variation's latency to the theoretical lower bounds for reading, writing, and their sum. First, recall that data is forwarded asynchronously with respect to rotations. An older version of the protocol does not forward the data to a tag until the tag has completed its rotates, although data is included in the response to a rotate request if wanted. Which tree-merging version is better, the new or the old? Second, recall that a tag does not know the right height of its forward neighbor immediately after a rotate. If this height is known, by storing it exactly and returning it when a rotate is accepted, then could a tag make a more intelligent decision about whether or not to request the next rotate? Call this the *recent-height* optimization. These two variations and their interactions are identified by suffixes, *PD* for Piggybacked Data, *RHO* for the Recent-Height Optimization, and *X* for neither. For example, *STEM-PD* is the old version of the protocol without recent height, *STEM-PD-RHO* is the old version with recent height, *STEM-X* is the new version without recent height (described in Section 5.3), and *STEM-RHO* is the new version with recent height.

Concerning lower bounds, we want to compare against the lower bound per variation per metric for 65536 tags, where the three metrics are the latencies of reading and writing and the sum. Recall that a list tag requests a rotate whenever the strictly growing property is violated, except that randomness is used whenever three adjacent list tags have the same right height. Is this the optimal policy? Perhaps it would be useful for a tag to wait. For example, let $x$, $y$, and $z$ be the right heights of three adjacent list tags $X$, $Y$, and $Z$ respectively such that the forward neighbor of $X$ is $Y$ and the forward neighbor of $Y$ is $Z$. If $x > y$ and $x > z$, then maybe $Y$ should always request a rotate with $Z$, even when $y < z$. This policy would decrease the height of the resulting tree, but it may increase the tree-creation latency. Note that this policy requires $Z$ to accept the rotate even when $y < z$, so both $x$ and $y$ are sent in a rotate request. Then, $Z$ can make an accurate determination based on the most current value of $z$.

How should a list tag decide when to request a rotate such that the latencies of reads and writes are minimized? We enumerate the 13 decision cases for $Y$, shown in Figure 5.10, where each case can decide yes, no, or maybe (50 percent chance) to request a rotate. A 13-ply vector of decisions is called a *configuration.* Each configuration has a unique identifier

$$\sum_{n=1}^{13} 3^{13-k} D(n), \quad D(n) = \begin{cases} 0, & \text{case } n \text{ decides "no"} \\ 1, & \text{case } n \text{ decides "maybe"} \\ 2, & \text{case } n \text{ decides "yes"} \end{cases} \tag{5.1}$$

with $n$ as given in Figure 5.10. As an example, the figure gives configuration 372,463 for STEM. Note that the notion of configuration is orthogonal to protocol variation.

In this section, we show that 1) STEM is significantly better than STEM-PD, 2) that the recent height optimization is not significantly useful, and 3) that the latency of the STEM configuration is near the latency of the optimal configuration for reading and writing. Next, we discuss the simulations. After that, we give the latency results for reading and writing and the sum.

**The Thirteen Decision Cases**

| n | Expression | Request Rotate? | n | Expression | Request Rotate? | n | Expression | Request Rotate? |
|---|---|---|---|---|---|---|---|---|
| 1. | $x < y < z$ | no | 7. | $x = y < z$ | no | 13. | $x = y = z$ | maybe |
| 2. | $x < z < y$ | yes | 8. | $z < x = y$ | yes | | | |
| 3. | $y < x < z$ | no | 9. | $x = z < y$ | yes | | | |
| 4. | $y < z < x$ | no | 10. | $y < x = z$ | no | | | |
| 5. | $z < x < y$ | yes | 11. | $y = z < x$ | yes | | | |
| 6. | $z < y < x$ | yes | 12. | $x < y = z$ | yes | | | |

**Fig. 5.10:** This table enumerates the 13 decision cases and illustrates the STEM configuration.

### 5.6.1. Simulation

Simulation is synchronous, assuming one time unit per network message. This time unit includes the cache and memory accesses at either or both ends of a message delivery. We use synchronous simulation because it is simpler than asynchronous simulation. By avoiding variable-latency messages, we miss the more complex state transitions. However, these corner cases are not expected to be frequent and the coherence protocol is not optimized for synchronous operation, so a synchronous simulation is a good approximation of its performance in an asynchronous system. To compare the latency of reads and writes, we use a simple benchmark, where a writer must invalidate all $N-1$ copies and then all the $N-1$ readers must simultaneously acquire new copies. We assume very large caches so that there is a negligible number of cache-line replacements, except when ownership is acquired for writing. For reads, we consider the latency to get the data, starting with an invalid cache line that is prepared to hold the data. This starting point is reasonable for our workload because a cache line will be vacated by the invalidation associated with the previous write. To avoid congestion at a heavily shared directory, we assume combining of read requests. For writes, we consider the latency to completely invalidate all copies and notify the writer, including the messages to obtain permission for writing. We assume that the same tag is the writer in every iteration, which is common for scenarios with one producer and many consumers.

We would like to find the lower bound on average latency per variation per metric over 1000 simulations with 65536 tags. Unfortunately, there are $3^{13}$ (1.6 million) configurations for each lower bound. Timings of early simulation runs suggested that it would take well over 100 years of wall-clock simulation time to compute each lower bound. Therefore, instead of comparing the latency of STEM and its variations to the corresponding lower bounds, we find one configuration per lower bound that is reasonably close to the latency of the lower bound, called a *cogent configuration.* Then, we compare the STEM configuration to the given cogent configuration for each variation and metric.

We find each cogent configuration in three steps. First, we eliminate obviously bad choices for 6 of the 13 cases of Figure 5.10. Case $n = 1$ decides no because this is the strictly growing property. To avoid sequential merging, discussed in Section 5.3.2, case 13 decides maybe. To guarantee that merging does not stop prematurely, cases 2, 5, 9, and 12 ($x < y$ and $y \geq z$) decide either maybe or yes. This limits the number of reasonable configurations to $2^4\ 3^7$ (35 thousand).

Second, we find a set of *preliminary configurations*, all of the configurations that have a median latency that is less than or equal to some cutoff for 10 simulations with $N = 1024$. Using the median instead of the average has two advantages: 1) each simulation can be stopped as soon as it is known that the value will be greater than the cutoff and 2) exploration of that configuration can be stopped when over half of its simulations exceed the cutoff. Using $N = 1024$ helps the simulations complete an order of magnitude faster. The choice of a good cutoff is discussed later in this section.

Third, we find the average latency over 100 simulations with $N = 65536$ for all configurations in the preliminary set. The configuration with the lowest average is the cogent configuration. After finding the cogent configuration, we find its average over 1000 simulations for all $N = 2^i$ and integer $i \geq 1$.

We argue that the average of a cogent configuration is near the average of the optimal configuration (lower bound) for $N = 65536$. If the optimal configuration is in a preliminary set, but it is not chosen by 100 simulations, then its average must be nearly identical to the chosen cogent configuration. There are three reasons why the optimal configuration may not be in a preliminary set: the use of the median instead of the average, the use of $N = 1024$ instead of $N = 65536$, and the use of only 10 simulations. If the median of the optimal configuration for $N = 1024$ is less than the associated cutoff and repeated simulations always produce cogent configurations with approximately the same averages for $N = 65536$, then the given average must approximate the lower bound. Therefore, our confidence that the average of a cogent configuration is near the lower bound increases with 1) our confidence that the cutoff is less than the median of the optimal configuration and 2) the reproducibility of the same cogent configuration.

First, since each of our cutoffs for $N = 1024$ is above the average of the associated cogent configuration, each cutoff must be above the average of the associated optimal configuration. Furthermore, the median is a good estimator of the average. This is because the average percent error of this estimate for $N = 65536$ in all our preliminary sets is $-0.16$ percent (medians are smaller) with a standard deviation of 0.68 percent (less than one percent). Since cutoffs are also above all cogent-configuration medians for $N = 1024$, it is highly probable that cutoffs are above the medians of the associated optimal configurations.

Second, we define a measure of reproducibility. Two preliminary sets form a *pair* if they are generated with the same constraints, except that their cutoffs differ by one (three for writes[33]). The set with the larger cutoff is called the *master set* and the other is called the *slave set*. The *containment set* of a preliminary-set pair is the largest set of configurations such that 1) the containment set is a subset of both the master and the slave sets and 2) the average (for 100 simulations) of each configuration in the containment set is not greater than any average (for 100 simulations) of a configuration in the set difference of the master and containment sets. The *containment* is the size of the containment set and is a measure of reproducibility. If the containment is $s \geq 1$, the cogent configuration is said to be reproduced with degree $s$.

Figure 5.11 gives some preliminary-set statistics for the ten combinations of the four variations and three metrics. Write-X and Write-RHO are respectively identical to Write-PD and Write-PD-RHO (latter two not shown) because these variations differ only in the way that data is distributed for reading. For greater accuracy, the medians and standard deviations are recomputed from 1000 simulations, but the set sizes and containments are based on the original

[33]Writes differ by three because the write latency is computed as three times the tree height plus a constant, discussed later in Section 6.3.1.3.

**Preliminary-Set Statistics**

| Metric Suffix | Cogent Configurations ($N = 1024$) Identifier | Std. Dev. | Average | Median | Cutoff | Master Set Size | Slave Set Size | Containment |
|---|---|---|---|---|---|---|---|---|
| Reading | | | | | | | | |
| PD | 371,959 | 1.551 | 51.323 | 51 | 52 | 209 | 59 | 6 |
| PD-RHO | 391,642 | 1.592 | 51.535 | 51 | 52 | 153 | 36 | 1 |
| X | 371,968 | 1.102 | 46.305 | 46 | 48 | 773 | 214 | 56 |
| RHO | 371,959 | 1.089 | 46.341 | 46 | 48 | 566 | 163 | 58 |
| Writing | | | | | | | | |
| X | 405,322 | 0.982 | 39.342 | 39 | 45 | 411 | 73 | 32 |
| RHO | 411,883 | 1.431 | 39.825 | 39 | 45 | 317 | 65 | 12 |
| R/W Sum | | | | | | | | |
| PD | 404,782 | 3.352 | 98.213 | 98 | 100 | 105 | 31 | 4 |
| PD-RHO | 371,977 | 3.646 | 99.827 | 100 | 101 | 167 | 66 | 7 |
| X | 404,782 | 2.742 | 92.760 | 93 | 96 | 262 | 124 | 16 |
| RHO | 371,977 | 3.174 | 94.581 | 94 | 96 | 207 | 90 | 1 |

**Fig. 5.11:** This table gives some statistics for preliminary sets. Recall that the cutoff difference for preliminary-set pairs is one, except that it is three for writes.

10 simulations that generated the preliminary sets. Note that cutoffs are chosen to be greater than the medians.

The table shows that the cogent configuration is reproduced in all cases. From this, we conclude that the average latency for the cogent configuration is a good approximation of the lower bound. Therefore, we will freely interchange references to lower bounds and references to the latencies of cogent configurations.

### 5.6.2. Reading Variations

The read latencies for the four STEM variations are shown with their respective cogent configurations in Figure 5.12. These graphs demonstrate that the new version of STEM is better than the old one, especially for small sharing sets. The lower bound is achieved for small sharing sets and the read latencies for large sharing sets are within 30 percent of optimal. The recent-height optimization is completely useless for small sharing sets and of marginal usefulness for very large sharing sets. It is interesting to note that the recent-height optimization

**Read Latencies of Variations**

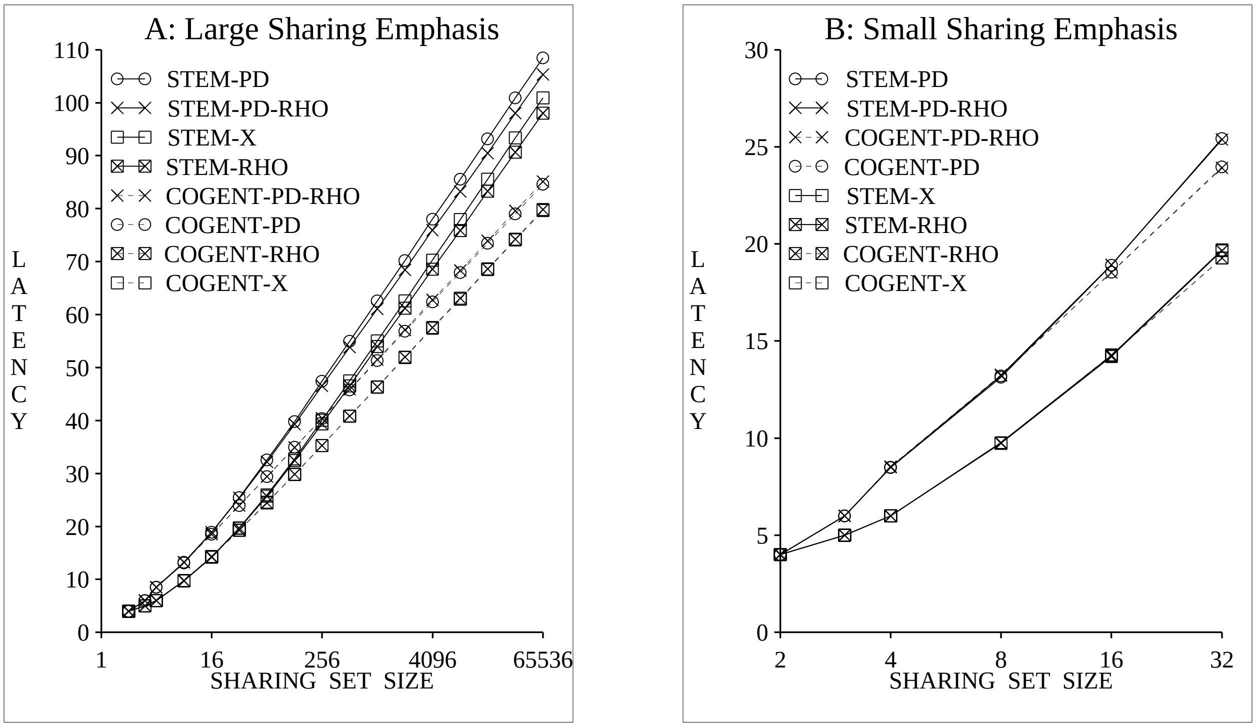



**Fig. 5.12:** These semi-log graphs show the read latencies for the four STEM variations with their respective cogent configurations. The keys are ordered by the latencies for the largest and smallest sizes, depending on the emphasis.

---

slightly increases the lower bound on latency, which is contrary to intuition. The reason for this is described later with the graphs for writing in Section 5.6.3.

### 5.6.3. Writing Variations

The write latencies for the four STEM variations are shown with their respective cogent configurations in Figure 5.13. The latencies of variations PD and PD-RHO are not shown because they differ from the latencies of variations X and RHO only for reading. These graphs again demonstrate that the recent-height optimization is useless. The STEM configuration achieves the lower bound for small sharing sets and is within 15 percent for large sharing sets. Again, the ordering of the lower bounds is inverted with a difference that is more significant than

**Write Latencies of Variations**

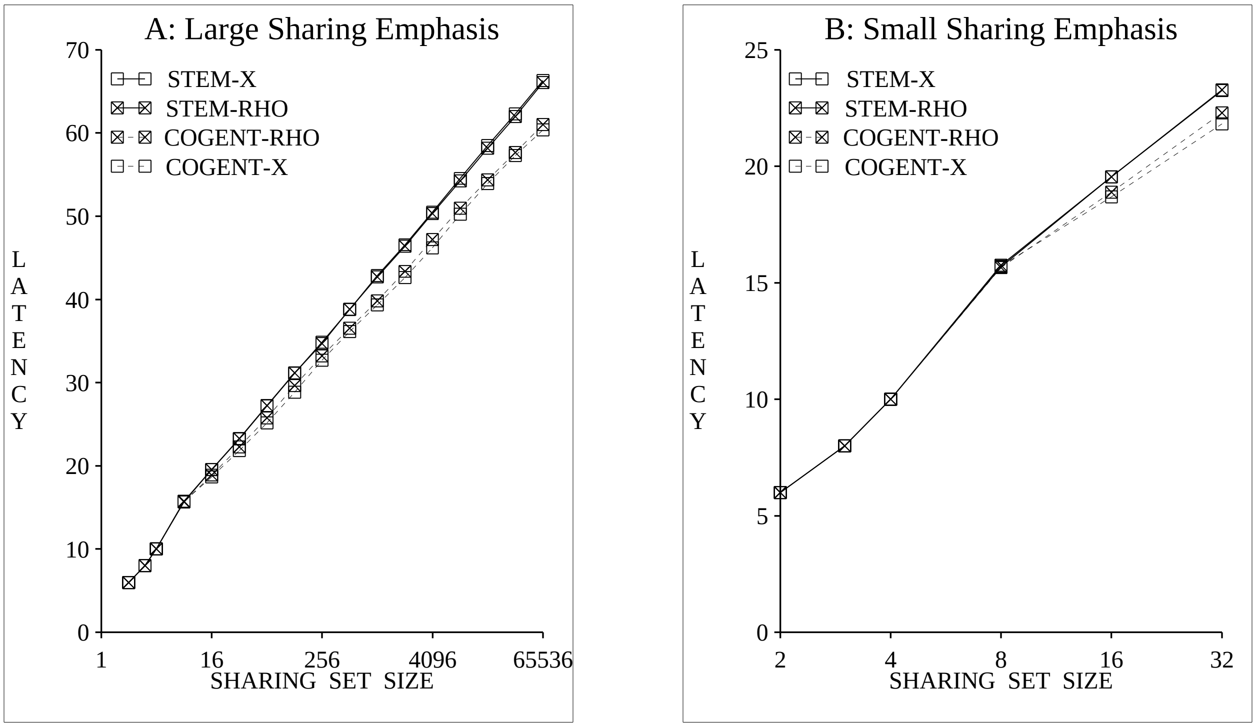



**Fig. 5.13:** These semi-log graphs show the write latencies for the four STEM variations with their respective cogent configurations. The keys are ordered by the latencies for the largest and smallest sizes, depending on the emphasis.

---

for reading. Although it is possible that the optimal configuration is not in the preliminary sets, we believe that this is not the case because the cogent configurations are reproducible.

Instead, we believe that the inversion is due to the complex interactions in the randomization of rotate-request decisions. The two cogent configurations for Write-X and Write-RHO, given in Figure 5.11, differ only in decision case $n = 5$ ($z < x < y$), where Write-X decides maybe and Write-RHO decides yes. By knowing the height $z$ of its forward neighbor $Z$, a tag $Y$ may increase the resulting tree height for Write-X. Tag $Y$ with $z < x < y$ may choose to wait, giving $X$ (its backward neighbor) time to increase its right height and then rotate with $Y$ and $Z$ for $z < y < x$. In configuration Write-RHO, however, $Z$ would first rotate with $Y$, thereby rotating with a tree of closer height and yielding a smaller overall height. Concerning performance, the recent-height optimization increases the average write latency of Write-X by 0.84 (for

100 simulations) and decreases the latency of Write-RHO by 0.57 (for 100 simulations). This results in a change of cogent configurations and a net increase in average latency of 0.45 (100 simulations) or 0.654 (1000 simulations). Note that the STEM configuration also decides yes for case $n = 5$, which is consistent with the fact that the recent-height optimization is marginally helpful for STEM.

More complicated explanations are constructed for reading. Although the performance implications are negligible, the differences are statistically significant[34] and require plausible explanations. Cogent configurations for Read-PD and Read-PD-RHO differ only in case $n = 4$ ($y < z < x$), where Read-PD decides no and Read-PD-RHO decides maybe when $z$ is known. If $Y$ knows $z$ after rotating, then it always waits and allows $X$ to rotate with $Y$ and then $Z$, resulting in a larger tree height than if $z$ is unknown. Since deciding yes is optimal for tree height, Read-PD is more penalized by the recent-height optimization than Read-PD-RHO, giving more weight to the advantage of this optimization in Read-PD-RHO.

Cogent configurations for Read-X and Read-RHO differ only in case $n = 11$ ($y = z < x$), where Read-X decides maybe and Read-RHO decides no. Here, the importance of tree-merging latency increases relative to tree height because $z - x = z - y$, not $z - x > z - y$. Let $W$ with right height $w$ be the backward neighbor of $X$ and let $ZZ$ with right height $zz$ be the forward neighbor of $Z$. $X$ will request a rotate with $Y$ exactly when $w \neq x$. $Z$ will wait for a rotate with $Y$ exactly when $z = zz$ and the coin flip is favorable. Assuming the probability of $w \neq x$ is greater than half the probability of $z = zz$, the probability of a rotation involving $Y$ is greater if $Y$ waits for $X$. Read-X suffers more than Read-RHO because Read-RHO always waits and Read-X waits with probability one-half. Therefore, the advantages of the recent-height optimization are less in Read-X.

---

[34]Read-PD and Read-PD-RHO are each simulated 1000 times with $N = 65536$. Respectively, the average latencies are 84.586 and 85.105 with standard deviations of 1.342 and 1.391. Read-X and Read-RHO are each simulated 1000 times with $N = 65536$. Respectively, the average latencies are 79.630 and 79.826 with standard deviations of 1.108 and 1.142.

### 5.6.4. Reading and Writing Variations

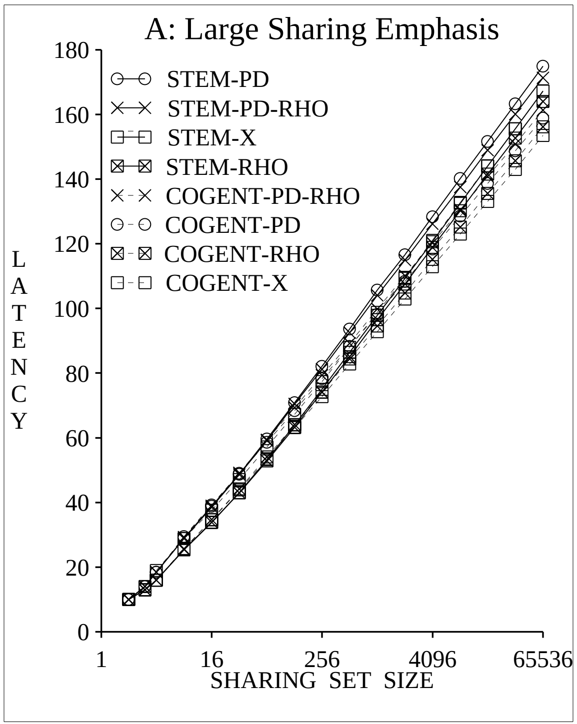
The sum of the read and write latencies for the four STEM variations are shown with their respective cogent configurations in Figure 5.14. The results are qualitatively the same as for reading, requiring only a few points of discussion. First, a lower bound on the sum of read and write latencies is not the sum of their lower bounds. This is because a cogent configuration for reading is not a cogent configuration for writing, as shown in Figure 5.11. Therefore, new cogent configurations are computed for the sum of the read and write latencies. Second, the lower bounds on latency are slightly higher than the latency of the STEM configurations. This does *not* imply that the optimal configurations are absent from the preliminary sets. This apparent anomaly is present because a cogent configuration is found for $N = 65536$ and, as indicated by the graphs, it is not a cogent configuration for small $N$. Since STEM is better than the given cogent configuration for the common cases of small $N$, we do not discard STEM in favor of the cogent configuration. Third, the cogent configurations plotted for small $N$ are not inverted as they are for reading and writing, although the importance of this is negligible. Fourth, Total-PD and Total-PD-RHO differ in decision case $n = 4$ ($y < z < x$), where they respectively decide yes and no, and in decision case $n = 5$ ($z < x < y$), where they respectively decide maybe and yes. Total-X and Total-RHO have the identical differences.

### 5.6.5. Worst-Case Latency

Recall that STEM uses randomness to break ties when a sequence of list tags all have the same right heights. The worst case for reading in STEM occurs when all of the tags choose to rotate, building a minimal-height tree with sequential merging. The worst case for writing is when all tags choose to wait, resulting in a tree that contains a sequential branch with over half the tags. Both worst cases have latency that is over 64,000 for 64,000 tags. However, probability of the worst case is vanishingly small[35], as shown in Figure 5.15. This frequency distribution with 10000 runs shows the STEM latency for 65535 readers to cache the data, merging with one

[35] James [Jame93] suggests that the worst case may be more likely if the pseudo-random number generators get into lock step, resulting long sequences of tags that get the same results for their coin flips. He suggests modifying the generators to include the tag's node identifier so that lock step is less likely.

**Sum Latencies of Variations**



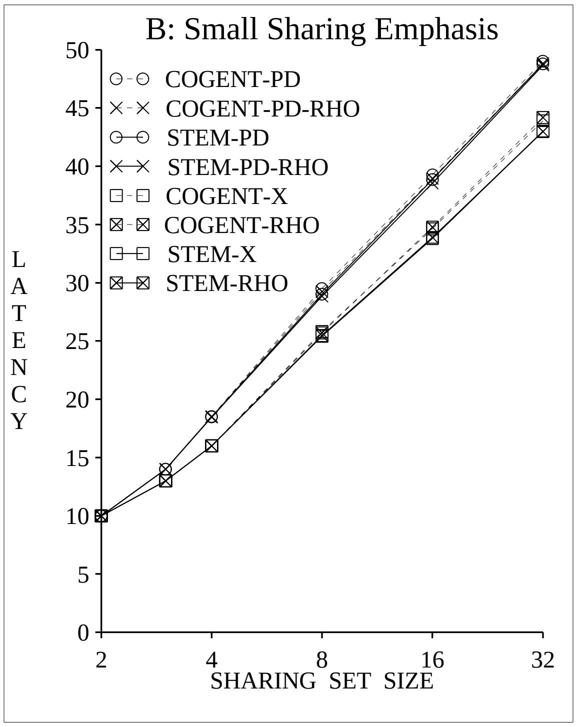




**Fig. 5.14:** These semi-log graphs show the sum of the read and write latencies for the four STEM variations with their respective cogent configurations. The keys are ordered by the latencies for the largest and smallest sizes, depending on the emphasis.

---

previous writer, and then for the writer to invalidate the 65535 readers. Although the data has a slight skew to the right, we can statistically say with 99.99% confidence that the average is $167.249 \pm 0.092$. This is computed by using the standard deviation of the given (large) sample from an infinite population. In other words, the contribution of all the bad cases, including the worst case, is negligible. Furthermore, since the variance is small and the expected tree height is about 20 for 64K nodes, a 5-bit height is sufficient for good performance[36].

[36]Recall from Section 5.3.2.2 that a subtree is purged in the rare case when its height reaches 31.

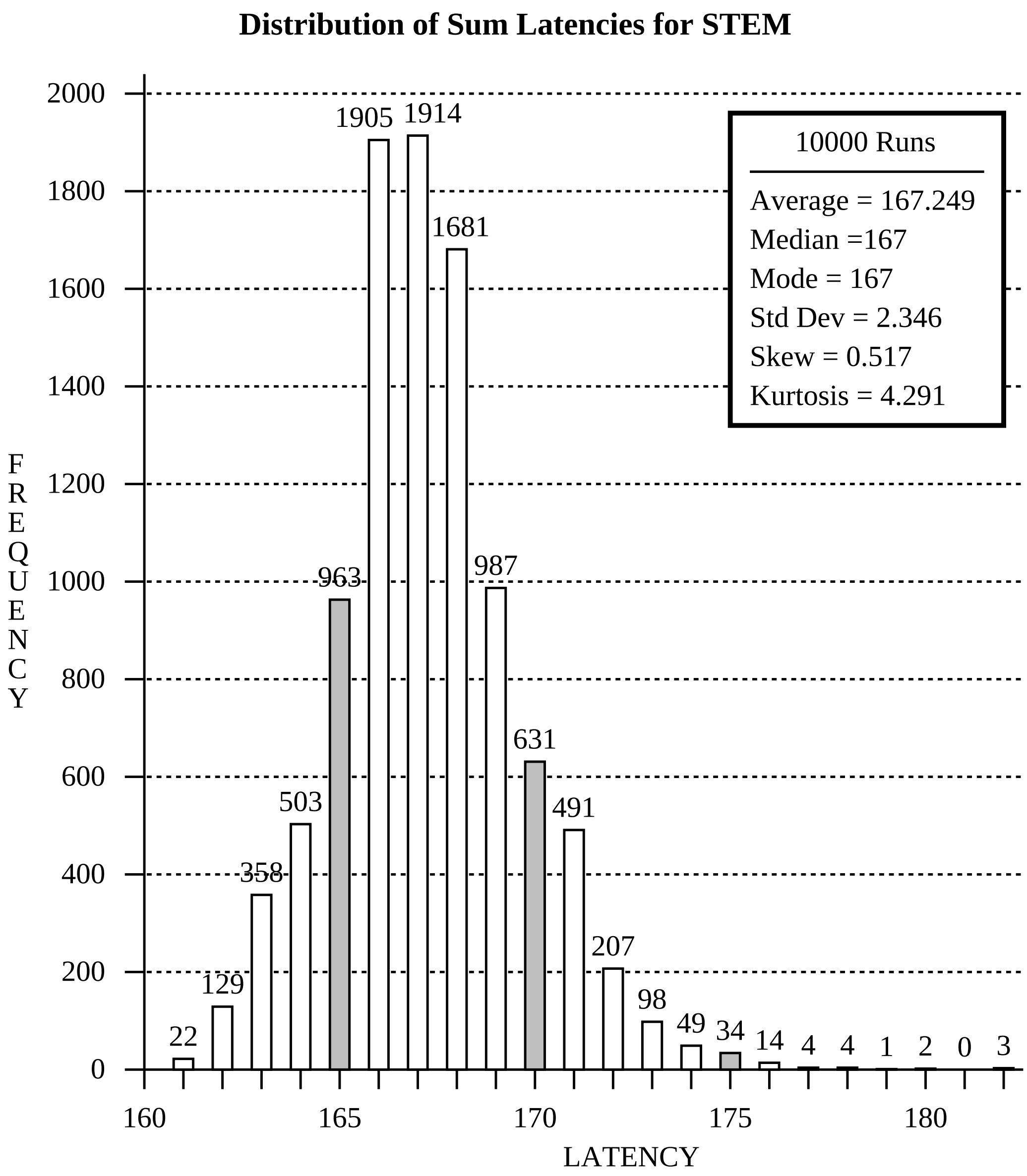



**Fig. 5.15:** This frequency distribution with 10000 runs shows the STEM latency for 65535 readers to cache the data, merging with one previous writer, and then for the writer to invalidate the 65535 readers. Several statistical measures of the sample are also given. All missing frequencies are zero.

---

## 5.7. Summary

We describe STEM, a new cache-coherence protocol that is efficient for massively parallel architectures (thousands of processors). STEM scales logarithmically for single writes, multiple cache-line replacements, and multiple reads to the same memory address. STEM has earned the consensus of the SCI working group (P1596.2) as the way to extend SCI's coherence protocol to efficiently support hundreds/thousands of processors.

Some may view tree structures as complex. However, there are no other viable coherence structures that have been proposed for thousands of processors: it is unacceptable to sequentially perform anything substantial. We manage the state-machine complexity by defining orthogonal state attributes and by limiting the interactions between protocol phases. Furthermore, the memory directory is very simple because it is ignorant of the coherence protocol; the directory only responds to requests.

STEM operates within the network paradigm of request-response transactions. To guarantee forward progress, STEM does not require broadcasting, request forwarding, in-order delivery, or exponential backoff from the network. This makes STEM more network independent than DASH [LLGG90], STP [NiSt92], and many other protocols.

STEM supports hardware-combining fetch-and-add, and/or other associative updates [GoLR83], without saving network state and without constraining the return path of the responses. Latency is logarithmic and the traffic is constant per tag. The complexity of combining is managed in the cache controllers, not in the network, thereby simplifying network implementation. Furthermore, our combining of associative updates, including concurrent writes, is compatible with sequential consistency.

# Chapter 6

# Performance Comparisons

## 6.1. Introduction

We believe that it is useful to provide efficient support for concurrent coherence operations on the same memory address in massively parallel machines. After a synchronization point, for example, it is likely that many processors will concurrently request the same new instructions and data. Although this hot spot is of short duration for small machines, a sequential-access latency will have a significant impact on the performance of massively parallel machines. Current performance studies of coherence protocols [ArBa86, ASHH88, EgKa88, OKNe90, CFKA90, GHGM91, GuWe92] have not addressed this problem because they have focused on the performance of small machines with at least one study [GHGM91] assuming that shared instructions hit in the cache. Matloff [Matl91] claims that scalability is not important because most software can be rewritten to group the sharing. Other studies [CFKA90, GuWe92] also suggest rewriting software that does not scale. However, the point of the shared-memory paradigm is to make it easier to generate and reuse correct and efficient parallel programs.

In this chapter, we compare a representative set of cache-coherence protocols. In Section 6.2, we review five candidate protocols and discuss why it would be unfair to include two of them in the following performance studies. In Section 6.3, we compare the latency and traffic of the three remaining candidates, based on the size of the sharing sets. This analysis is workload independent. In Section 6.4, we make similar comparisons, based on system size. This analysis makes assumptions about the workload. In Section 6.5, we make other comparisons and, in Section 6.6, we summarize this chapter.

## 6.2. Candidate Selection

In this section, we consider five invalidate protocols that are based on directories, where the directory's location is a function of the address. The protocol candidates are the Stanford DASH [LLGG90] of Section 1.3.3, the Scalable Tree Protocol (STP) [NiSt92] of Section 1.3.4.3, the Scalable Coherent Interface (SCI) [Gust92a] of Section 1.3.4.1, the Recursive-Doubling Extensions to SCI (RDE) of Chapter 3, and the Tree-Merging Extensions to SCI (STEM) of Chapter 5. We also consider the optimization of pairwise sharing (PWS) for SCI, STEM, and RDE. Before discussing our two performance studies, we review each protocol and then drop DASH and STP from further consideration.

### 6.2.1. Protocol Review

Here, we describe the five protocol candidates: DASH, STP, SCI, RDE, and STEM. An optimization for pairwise sharing is also described.

#### 6.2.1.1. Stanford DASH

We use the DASH coherence protocol [LLGG90, LLGW92] as a representative for all distributed-directory protocols that do not distribute the storage of each sharing set [Tang76, CeFe78, ArBa84, ASHH88, OKNe90, ChKA91, SiHo91, HLRW92]. DASH maintains the sharing set as a bit map of 64 clusters, one bit per shared-bus cluster of four processors. To simplify the discussion and since all directory protocols could cluster several processors, we reduce DASH's cluster size to one. In DASH, a writer sends an invalidate request to the directory, the directory forwards the request to all caches with copies, and then the caches respond directly to the writer, which counts the responses. The first reader sends a request to the directory, the directory forwards the request to the writer, and then the writer responds to the reader and the directory. Subsequent readers receive the data from the directory, after the directory receives its response from the writer. Replacement requests flip their associated bit in the directory map.

#### 6.2.1.2. Scalable Tree Protocol

The Scalable Tree Protocol (STP) of Nilsson and Stenström [NiSt92] builds and maintains a threaded $x$-ary tree of minimal height by serializing all tree operations. For a binary tree ($x = 2$), each cache line and directory maintains five pointers. For each read, write, and replacement, the tree is kept perfectly balanced. The first reader sends a request to the directory and the directory then forces a write-back of the data before responding to the reader. These nested request-response transactions can cause deadlock on top of an SCI network[37], but Nilsson and Stenström do not discuss this problem. Subsequent readers receive the data from the directory. After receiving the data and the insertion point from the directory, a reader inserts itself into a perfectly balanced tree, waiting for previous readers to finish, if any. A writer locks the tree by consulting the directory, which then forwards the invalidate request throughout the balanced tree. Each cache line is deleted as it responds to its parent in the tree, after receiving responses from all its children. A cache-line replacement locks the directory before replacing itself with another node. The tree is kept perfectly balanced by choosing the insertion point as the node used for swapping.

#### 6.2.1.3. Scalable Coherent Interface

We use the cache-coherence protocol of the Scalable Coherent Interface (SCI) [JLGS90, Comm91, Gust92a] as a representative for all directory protocols that distribute the sharing set as a linked list of cache lines [ThDe90a, ThDe90b, Knig92][38]. The directory maintains a pointer to the most recent insertion (insertion point) and the insertion order determines the list order. Readers append to the list by consulting the directory, getting the data from either the previous insertion point or the memory at the directory. Combining of read requests in the network may be used to avoid congestion at a heavily shared directory. This published mechanism [JLGS90] of Section 5.3.2.1 is compatible with the memory controller defined in SCI [Comm91]. However, it is not explicitly described in the standard because it is expected to be described in the

[37]SCI guarantees forward progress of single request-response transactions, not nested request-response transactions. As discussed in Section 1.3.4.1, general request forwarding is forbidden by SCI and may lead to deadlock. However, it may be possible to use a *cache-internal queue*, similar to the one discussed in Section 5.5.2, but Nilsson and Stenström do not discuss this possibility. Leaving this as a topic for future work, we note that a cache line can not have valid data at the same time as the cache line is on the queue.

[38]See Section 1.4.2 for limited information about the S-1 project [Knig92].

kiloprocessor extensions. Writers sequentially invalidate the list after becoming the insertion point. A cache-line replacement connects its two neighbors in SCI's doubly-linked list. When multiple and adjacent replacements occur, list neighbors are removed sequentially.

SCI can create a lot of traffic due to spin waiting. For example, if a large number of readers request the data at the same time, a long list is created. Each reader repeatedly asks its neighbor for the data until the data arrives. The last reader makes this request $\Theta(p)$ times, where $p$ is the size of the list, resulting in $\Theta(p^2)$ total messages. One way to avoid this spinning is to use request reserving, as discussed in Section 3.2.4. However, adding request reserving to SCI would create a coherence protocol that is not SCI, resulting in misleading performance comparisons. Instead, we assume a careful implementation of SCI that avoids spinning. If a node receives a request such that an immediate response would result in a retry of the request, then the request is held in a wait queue until either the desired response can be made or the queue becomes full. The implementation must be careful to set the response timeout value (between 23ns and 136 years) so that a timeout does not prematurely signal a lost packet. Since SCI's traffic with combining is uniform, small wait queues should be sufficient to avoid request retries and an assumption of infinitely large wait queues is reasonable for our performance studies.

#### 6.2.1.4. Recursive-Doubling Extensions

The recursive-doubling extensions to SCI (RDE), discussed in Chapter 3, are the first coherence extensions that were considered by the SCI working group [Gust91]. RDE extends an SCI list by adding temporary pointers that make shortcuts across the list. Temporary pointers are created, using an asynchronous and enhanced version of the recursive doubling of Hillis and Steele [HiSt86] and Stone [Ston73], after which these pointers can be used for efficient distribution of data and invalidates. These extra pointers are temporary because they are not updated when nodes are removed from the list during replacements, meaning that the temporary pointers may become stale over time. Combining of read requests in the network may be used to avoid congestion at a heavily shared directory, although this combining is more complicated than the combining used for tree merging and SCI. In order to create a set of temporary pointers that is efficient for multicasting to all nodes in the list, each node must know its position in the list, called a *pcount.* This knowledge is propagated, called *pcount distribution,* using a preliminary tree that is created during combining. In order to improve the height of this preliminary tree, combinable requests are not immediately combined, waiting for more optimal pairs of requests.

#### 6.2.1.5. Tree-Merging Extensions

The tree-merging extensions to SCI (STEM), introduced in Chapter 5, have the consensus of the SCI working group [Gust91] for use as extensions to SCI. Multiple readers consult the directory, possibly using combining to avoid congestion, and then cooperate to build a probabilistically balanced tree in logarithmic time. This cooperation is called tree merging. Sequential insertion of readers results in a perfectly balanced tree. A writer first becomes the root of the tree and then multicasts an invalidate throughout the tree, being careful to guarantee forward progress. As the invalidate reaches the leaves, nodes begin to remove themselves, leaves first, until the writer is the only remaining node. If a replacement victim has less than three neighbors, then the SCI replacement routine is used. If the victim has three neighbors, then it finds a leaf node, detaches it, and makes a swap to keep the tree intact. Replacements slowly cause the staleness of tree heights, which are used for tree merging.

#### 6.2.1.6. Pairwise Sharing

SCI, STEM, and RDE are compatible with an optimization for pairwise sharing (PWS). When this optimization is activated, a purger leaves one stale reader in a list/tree of size two. Then, reading and writing continues without consulting the directory because the two nodes communicate directly.

### 6.2.2. Protocol Discussion

For two reasons, we drop two of the above candidates from further consideration. First, DASH and STP would not use SCI's combining mechanism to handle hot spots and modeling the contributions of the different combining mechanisms would be difficult. Second, DASH and STP do not claim forward progress or robustness and it is difficult to modify the protocols to make a fair comparison to SCI. We do not consider the scaling problems associated with DASH's directory storage because reasonable solutions exist, such as pointer caches of Section 1.3.3.4 and LimitLESS Directories [ChKA91].

### 6.2.2.1. Hot Spots and Combining

Pfister and Norton [PfNo85] propose a hot-spot model that incorporates a background of uniform traffic and a percentage of messages destined for a single location (the hot spot). Let $r$ $(0 \le r \le 1)$ be the total rate emitted by each source, let $h$ $(0 \le h \le 1)$ be the percentage of $r$ that is destined for the hot spot, let $p$ be the number of processors (and the number of memory modules), and let $c$ $(0 < c \le 1)$ be the maximum service rate of the sink. For simplicity, Pfister and Norton assume that all messages are the same size, all messages are requests (there are no responses), and that all requests can be serviced at a maximum rate of $c = 1$ (which is why $r \le 1$). The authors explain the phenomenon of tree saturation, where messages back up in the network (with finite queues) because the arrival rate at the hot spot exceeds the service rate. In particular, tree saturation begins to develop when the arrival rate of uniform traffic, $r\,(1 - h)$, plus the arrival rate of hot spot traffic, $r\,h\,p$, equals $c$. The authors [PfNo85] show that combining in RP3 [PBGH85] can prevent hot spots and tree saturation in a 6-stage network.

Kumar and Pfister [KuPf86] later show that the onset of hot spots is very quick, on the order of 10 to 50 instructions, and that recovery is much slower than the onset. Lee et al. [LeKK86] show that RP3's 2-way combining (only providing resources for one combining per switch) is not sufficient to avoid hot spots in a 9-stage network, although their results suggest that 3-way combining is sufficiently close to the optimal $n$-way combining. Dickey and Kenner [DiKe92] propose 2.5-way combining. It allows 3-way combining when requests arrive at particular inputs to the combining switch, but it is not as expensive to implement as 3-way combining. Dickey and Kenner argue that the performance of 2.5-way combining is identical to the performance of 3-way combining.

It is important to note that tree saturation decreases a network's performance for all transactions, including transactions that do not visit the hot spot. This means that one poorly written application can destroy the performance of other applications in a time-shared environment. Therefore, when considering the effects of hot spots and tree saturation, the examination of poorly written applications is more important than the examination of well-behaved applications. In particular, some applications exhibit large degrees of sharing with frequently changing data.

Some studies [CFKA90, Matl91, GuWe92] suggest rewriting applications that have a lot of system-wide sharing and we agree that it is always possible to rewrite software to avoid large sharing levels, using software combining [YeTL87, GoVW89, YeTa90] and/or other techniques. However, the point of the shared-memory paradigm is to make it easier to generate and reuse correct and efficient parallel programs. If a large multiprocessor can efficiently execute old parallel programs, then this is a definite advantage over a multiprocessor that insists on software rewriting before providing acceptable performance. Furthermore, it is still an open research questions as to whether or not a software solution for system-wide sharing can be as efficient as a hardware solution.

Lebeck and Sohi [Lebe91, LeSo92] classify combining mechanisms into four types. Their taxonomy is based on 1) what determines the combining set and 2) what distributes the results. In network-network combining [GGKM83, PBGH85], the network determines the combining set and distributes the results. Processor-processor combining is another term for software combining [YeTL87, GoVW89, YeTa90] and this combining requires *a priori* knowledge of the number of requests that may combine [YeTa90]. In processor-network combining, the processors determine the combining set and then the network distributes the results. As with software combining, this requires *a priori* knowledge of the combinable locations. In the remaining type, network-processor combining [JLGS90, Lebe91, LeSo92], the combining set is determined by the network and the results are distributed by the processors.

Given the need for combining, it is natural to use network-processor combining for SCI, RDE, and STEM because they are designed to be compatible with that combining. However, it is natural to use network-network combining for STP and DASH because the directories, not the cache controllers, are designed to distribute copies of memory. This is not to say that STP and DASH could not be modified to be compatible with network-processor combining, but the details of such modifications are not readily available. Note that the natural network-network combining for DASH needs some way to remember the set of nodes that combined so that the sharing set at the directory can be accurately updated. The straight-forward implementation of this would result in combined requests that have a maximum size proportional to the system size.

In any case, it is important to distinguish between the types of combining used for the five protocols because the type of combining will influence protocol performance and complexity. On one hand, network-network combining implies finite result queues that can be another source of contention and tree saturation and the result queues imply that response paths are limited by the combining. Recall that Lee et al. [LeKK86] report a need for at least 3-way combining to avoid tree saturation and recall that 3-way network-network combining is expensive. Also, DASH's combining possibilities are limited by the need to compress a sharing set into one request packet. On the other hand, network-processor combining implies greater latency of the combining operations. However, as discussed in Section 1.3.4, network-processor combining is simpler and the network components are more general than for network-network combining. Recall that network-processor combining (SCI's combining) can implement *n*-way combining because there is no state saved in the network.

Therefore, a fair comparison of the five protocols would need to include the differences in combining mechanisms, probably including an exact model/simulation of the traffic patterns. Obtaining performance data for such a study would be exceedingly slow, especially for thousands of processors. Nilsson and Stenström [NiSt93] show that STP compares favorably with SCI, but they simulate only 16 processors and avoid tree saturation through assumptions of infinite queues and an infinite-bandwidth network with constant delay. Contention in their simulations occurs only at the intranode buses and combining is not modeled. This comparison would not be valid for systems with thousands of processors.

#### 6.2.2.2. Design Philosophies

The five protocols are also different in the way that they approach forward progress (avoidance of both deadlock and starvation) and robustness (recovery from transmission errors). Some definitions of forward progress require only one process to complete useful work, allowing other processes to starve. Whereas this may be appropriate for some applications, this is not appropriate for cache-coherence protocols. Our definition of forward progress is that all requests are eventually satisfied in bounded time. For example, SCI guarantees forward progress of 2-phase transactions and coherence operations: every request-response transaction is guaranteed to complete and every cache eventually finishes each coherence operation.

The coherence protocols of SCI, RDE, and STEM guarantee forward progress with only 2-phase transactions. However, STP and DASH do not. STP uses nested request-response transactions and Nilsson and Stenström [NiSt92] offer no solution to avoid or break deadlock, which is unacceptable for an extension to SCI. The authors also state that read and write operations are not serviced during cache-line replacement, but they do not specify how starvation is prevented when multiple requests are waiting on a lock. STP's locks are also a problem for robustness: how does software recover from a hardware malfunction that halts a node and leaves locks set throughout the system?

DASH uses multiphase transactions for both an invalidation and a first reader. DASH's multiphase transactions for reading and writing provide a substantial performance improvement over SCI. For example, the first DASH reader obtains the data in 3 message delays, as compared to 4 message delays for SCI. Concerning forward progress, DASH's network uses exponential backoff to break deadlock [LLGW92], which can occur when queues become full. However, DASH does not prevent starvation, which is one of the design goals of SCI.

It would be difficult to modify DASH to meet the design constraints of SCI. Currently, DASH uses two 2-D mesh networks to guarantee forward progress of all request-response transactions, one mesh per transaction phase, but this does not guarantee forward progress of multiphase transactions. It would not suffice to add a third 2-D mesh for DASH's 3-phase invalidation triangle because the directory forwards one invalidation request per cached copy: finite output queues at the directory can cause a backup at the input queues, paving the way towards deadlock. To guarantee forward progress, DASH needs a network mechanism for multicast and this mechanism must guarantee forward progress in the face of finite queues[39]. Furthermore, SCI guarantees detection of any number of lost packets and the ability for software recovery, so DASH's multicast mechanism would need to be reliable. The need for a reliable multicast mechanism with forward progress would make DASH systems very complex to implement, perhaps unrealistic. Although SCI has a reliable broadcast mechanism for one ring [Comm91],

[39]Note that a broadcast mechanism would not be sufficient for scalability.

it was considered too difficult to implement a reliable multicast over several rings, which is what DASH would require. Instead, SCI could be modified to use a 3-phase transaction for reading by either 1) giving up its claim to forward progress or 2) adding a third set of network queues[40]. This would reduce the performance differences between DASH and SCI, but a comparison would still be unfair due to DASH's multicast without forward progress.

Since STP and DASH improve their performance with techniques that violate SCI's design goals (guarantees of robustness and forward progress), direct comparisons to SCI, RDE, and STEM would be unfair. It would be exceedingly difficult to modify these protocols to make a fair comparison. Furthermore, STP's nested request-response transactions can lead to deadlock and it is unclear that there exists any scalable combining mechanism for DASH. Therefore, we drop STP and DASH from further consideration.

### 6.3. Increasing Sharing-Set Size

Ideally, we would like to compare the performance impacts of different coherence protocols for real programs on large machines. However, this is not possible with currently available technology[41]. Instead, we consider the latency and traffic associated with each read, write, and cache-line replacement for a given sharing level. Simulation is synchronous, assuming one time unit per network message. This time unit includes the cache and memory accesses at either or both ends of a message delivery. We use synchronous simulation because it is simpler than asynchronous simulation. By avoiding variable-delay messages, we miss the more complex state transitions. However, these corner cases are not expected to be frequent and the coherence protocol is not optimized for synchronous operation, so a synchronous simulation is a good approximation of a protocol's performance in an asynchronous system. Although analysis of latency and traffic will not directly predict the performance of any parallel program, it gives us a consistent and

---

[40] SCI guarantees forward progress by having a separate set of network queues, one for requests and one for responses. Although requests and responses are multiplexed on the same ring links, their storage queues are distinct. More details are discussed in the SCI standard [Comm91].

[41] Virtual prototyping [HLRW92, RHLL92] shows promise, but the Wisconsin Wind Tunnel [RHLL92] is currently limited to 64 processors.

reasonable framework for comparing various cache-coherence protocols, paying special attention to results that affect programs with high levels of sharing. Since this dissertation is the first study of some new coherence protocols, we do not consider the effects of other latency-reducing techniques [CaKP91, GHGM91].

For cache-line replacements, we compare the average latency to prepare a cache line to hold new data, starting with valid data in a stable state with a different memory address. To compare reads and writes, we use a simple benchmark, where a writer must invalidate all $N-1$ copies and then all the $N-1$ readers must simultaneously acquire new copies. We assume very large caches so that there is a negligible number of cache-line replacements, except when ownership is acquired for writing. For reads, we consider the latency to get the data, starting with an invalid cache line that is prepared to hold the data. This starting point is reasonable for our workload because a cache line will be vacated by the invalidation associated with the previous write. To avoid congestion at a heavily shared directory, we assume combining of read requests. For writes, we consider the latency to completely invalidate all copies and notify the writer, including the messages to obtain permission for writing. We assume that the same node is the writer in every iteration, which is common for scenarios with one producer and many consumers. Note that this benchmark is the same as discussed in Section 5.6.1.

We compare reads, writes, and replacements for three invalidate protocols that are based on distributed directories, where the directory's location is a function of the address. The candidates are SCI, RDE, and STEM. Next, we derive the equations and/or discuss the simulation assumptions. After that, we present the results of this performance study.

### 6.3.1. Equations and Simulations

Here, we derive the equations and/or discuss the simulation assumption for each of the three candidate protocols. Pairwise sharing is discussed later in Section 6.3.2.

Recall that latency is the number of messages on the critical path and traffic is the total number of messages. Later, in Section 6.4.2, we would like to add the traffic results for reading and writing. In order to facilitate this addition, we compute the traffic per node, rather than the total traffic. Note that this metric pertains to the number of messages sent from each node, not the number of messages passing through each node.

#### 6.3.1.1. Scalable Coherent Interface

Consider the latencies in SCI with delayed responses. The latency for $n-1$ simultaneous reads is 2 messages to consult the directory, 2 messages for the first reader to get the data, and 1 more message for each of the $n-2$ remaining readers to get the data, yielding

$$n+2. \tag{6.1}$$

The latency for a writer to purge the $n-1$ readers is 2 messages to detach from the tail of the list, 2 messages to consult the directory and become the head, and $2(n-1)$ messages to purge the list, yielding

$$2n+2. \tag{6.2}$$

Consider the traffic per node in SCI with delayed responses. The traffic per reader for $n-1$ reads is 2 messages to consult the directory and 2 messages to get the data, yielding

$$4\,\frac{n-1}{n}. \tag{6.3}$$

The $(n-1)/n$ factor amortizes the traffic per node, counting the dormant writer. The traffic per node for writing is the same as for latency, $2n+2$ messages, divided by the number of nodes, yielding

$$2+\frac{2}{n}. \tag{6.4}$$

#### 6.3.1.2. Recursive-Doubling Extensions

For RDE, the binary structure and patient combining of degree 3 are assumed, as respectively recommended by Sections 3.6.4 and 3.4.3. Simulation is used to approximate the expected height of the tree that is used for distributing the pcount. This simulation takes a pool of list segments, initially all of size one, randomly selects two, combines them as would be done in the network, and places the combined segment back into the pool. In addition, there is a slow drain on the pool to simulate the processing of the directory. This is repeated until no segments are left. The worst-case segment that was drained gives the tree height that corresponds to the latency of pcount distribution. More details on this simulation can be found in Section 3.4.1.

Note that this simulation initially favors the combining of small segments, as would be true in a real network, due to the high ratio of small segments to large segments. The average of 1000 simulations for each list size is added to the analytically derived latencies of the other reading phases, given in Sections 3.6.2 and 3.6.3. For small lists (2 to 4), we assume that the pcount is returned by the directory, which is true whenever there is no congestion and, therefore, no combining. The latencies for writes are also taken directly from these sections, except for small lists (2 to 4). Small lists are simulated by hand because some of the given equations are not valid for these sizes.

The traffic analysis of Section 3.6.2.4 is not precise enough for a traffic comparison between coherence protocols. This is because the analysis of that section does not consider the overlap of invalidate distribution and purging, as do the analysis of SCI and the simulation of STEM. Therefore, we simulate RDE to determine the amount of traffic generated by the protocol for both reads and writes. The simulation for reads will test the simulation data against the analytical data. The simulation for writes will give the new data for invalidation and purging.

For writes, we simulate invalidate distribution and purging and then add 4 messages for the writer's rollout and directory access. Note that the writer does not generate any temporary pointers. For reads, we assume that each reader begins with its pcount and a pointer to its forward neighbor. The simulation determines the traffic generated by recursive doubling and data distribution, including the traffic generated by the creation of insurance pointers of length 6 or greater. After finding the traffic generated by recursive doubling and data distribution, we add 2 messages per reader for the directory access (and possible combining) and, for pcount distribution, 4 messages per reader over 8. If there are 8 or less readers, then it is unlikely that they will combine in the network and each reader will get its pcount from the directory. However, as the number of readers increases, so does the likelihood of combining and the traffic related to pcount distribution. It is unrealistic to expect the number of combined segments to remain constant (at 8 or any number) as sharing-set size increases. However, our results are not sensitive to this parameter, so the assumption is not a problem.

#### 6.3.1.3. Tree-Merging Extensions

Since STEM uses randomness, it is difficult to obtain analytical performance data. Therefore, we use simulation to determine the average latency and traffic for tree merging and the average insertion-root height of the resulting tree. All results are averaged over 1000 simulations, except when noted otherwise.

Simulations provide data for the latency and traffic of reads and the traffic of writes[42], but additional analysis is used to get the results for write latency. Recall that the tree root for purging is the deletion root, not the insertion root. To approximate the deletion-root height, we use 1.5 times the insertion-root height. It is also possible to find the deletion-root height by simulation. However, we believe that our approximation better represents the deletion-root height in real systems, when there is a mixture of sequential and parallel insertions. In any case, the deletion-root height is bounded between one and two times the insertion-root height. The write latency is then 4 messages for rollout and directory access, 2 messages for contacting the previous head, plus 3 times one less than the insertion-root height. If the simulation determines that the writer's rollout would reduce the insertion-root height, then 2 is subtracted from the write latency. Writes with 3 or fewer readers are simulated by hand, since our scaling of the insertion-root height is valid only for large sharing levels.

### 6.3.2. Performance Results

Here, we present the results of this performance study. In latency graphs for reading and writing, PWS for sharing level two is shown as a separate data point in the graphs that emphasize small levels of sharing. If this optimization is mistakenly activated for larger sharing levels, read latency is slightly decreased and write latency is slightly increased[43]. Next, we discuss the performance results for read misses. After that, we discuss the performance results for write misses.

---

[42]Write traffic is difficult to compute analytically because it depends on the population distribution for nodes with 0, 1, and 2 children.

[43]Average read latency is decreased because one of the readers does not need to consult the directory. Write latency is increased because the writer is not the tail when it becomes time for the next invalidate, requiring 2 extra messages for rollout.

#### 6.3.2.1. Read Misses

Figure 6.1 compares the performance of read misses for the three protocols. Concerning latency for large sharing levels, SCI has $\Theta(N)$ read latency, scaling linearly. This is a noticeable problem as soon as sharing level 16. However, RDE and STEM have logarithmic latencies, thereby mitigating the effects of many simultaneous read-misses on overall execution time. Although RDE's latency is lower than STEM's latency, the small constant factor should not make a significant impact on overall execution time because large sharing levels are expected to be infrequent. Concerning latency for small sharing levels, the three protocols are almost identical.

Concerning traffic for large sharing levels, RDE and STEM have 4 times as much as SCI. This is the price that is paid for logarithmic latency. However, the traffic per node levels off at a constant amount for all three. Note that RDE's read traffic, 13.25 messages per node, is exactly the same as computed in Section 3.6.2.4, when including the traffic generated by directory accesses and the creation of insurance pointers. Concerning traffic for small sharing levels, the three protocols are within a factor of two and they are equal for the common case of pairwise sharing, with and without the optimization. Since the common sharing sets are expected to be small, the three protocols should have similar amounts of read traffic for most applications.

Although RDE and STEM have about the same read traffic, they allocate their traffic budgets differently. On one hand, RDE spends part of its traffic budget determining the pcounts in order to create a deterministic tree structure. More of RDE's traffic budget is allocated to data distribution because it makes reservations to avoid spinning. On the other hand, STEM spends most of its traffic budget on the randomness in tree merging, leaving a minimal amount for data distribution. Furthermore, STEM's reservations are made implicitly during tree merging, so less traffic is generated during data distribution.

#### 6.3.2.2. Write Misses

Figure 6.2 compares the performance of write misses for the three protocols. Concerning latency for large sharing levels, SCI has $\Theta(N)$ write latency, scaling linearly. This is a noticeable problem as soon as sharing level 8. However, RDE and STEM have logarithmic latencies, thereby mitigating the effects of large-scale invalidations on overall execution time. The

**Average Read Miss, Given Sharing-Set Size**

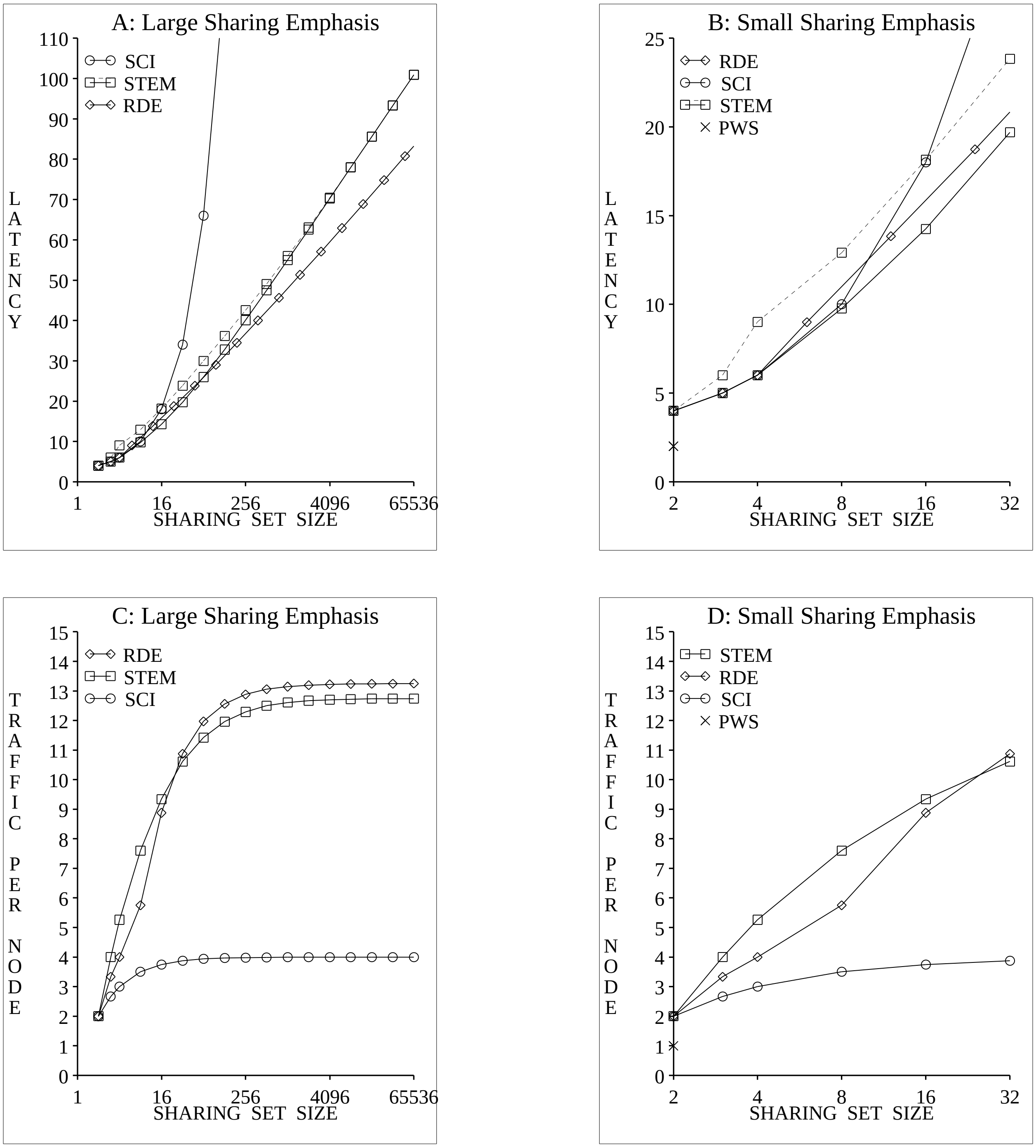



**Fig. 6.1:** These semi-log graphs show the average read-miss performance of the three protocols, given a particular sharing level. The dotted lines for STEM indicate the additional latency to complete tree merging after the data is distributed.

**Average Write Miss, Given Sharing-Set Size**

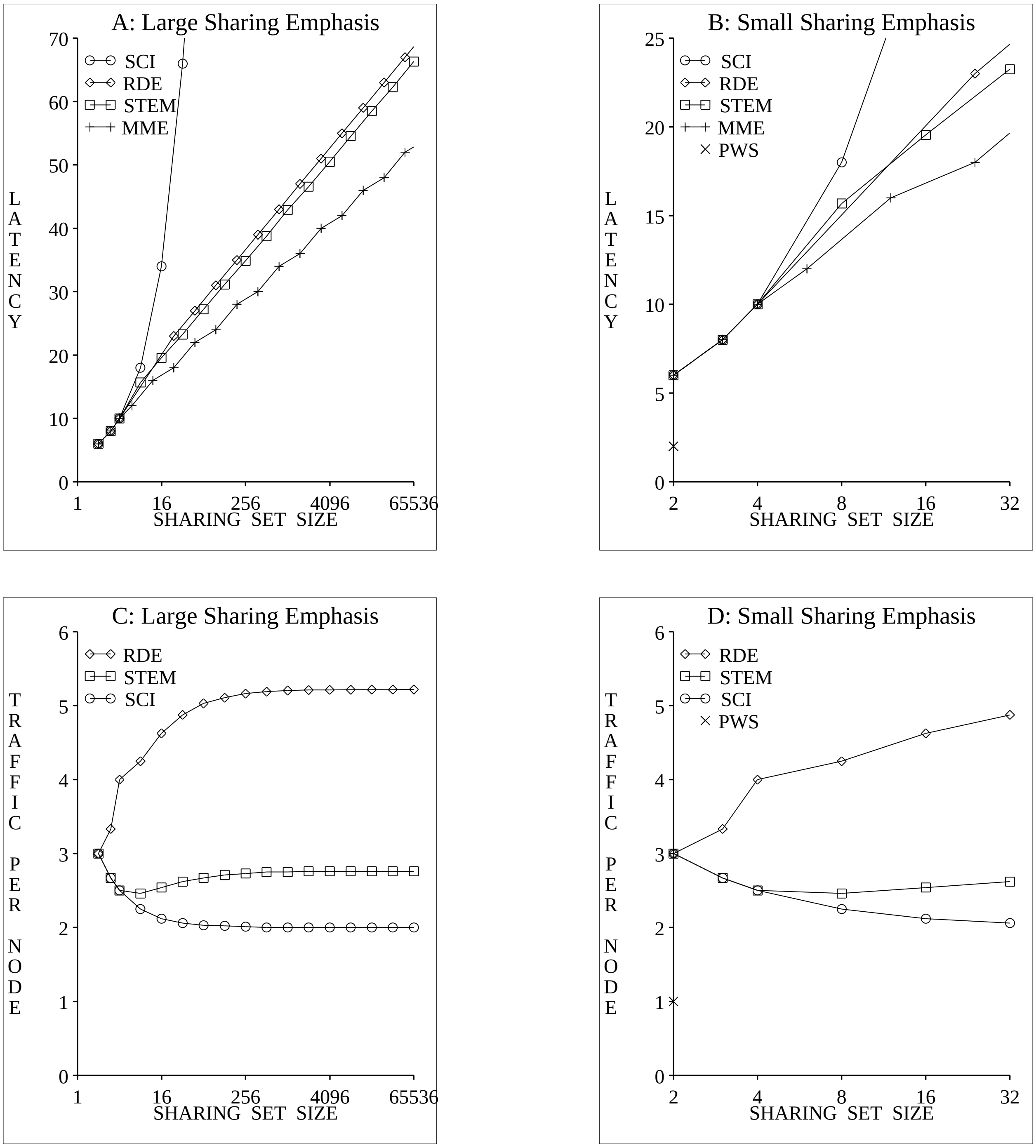



**Fig. 6.2:** These semi-log graphs show the average write-miss performance for the three protocols, given a particular sharing level. The MME lines show the median write latency for STEM when the tree is constructed by sequential insertions.

latencies of RDE and STEM are almost identical. However, when nodes are sequentially inserted, STEM's write latency varies violently between the given latency and about half that much. This is because sequential insertions create a perfectly balanced binary tree from the insertion root, but the distance between the insertion and deletion roots varies between zero and the height of the insertion tree. The median of this range is shown in the graph by MME, which stands for Median Merging Extensions. The difference between MME and the range's lower bound (not shown) is about the same as the difference between STEM and MME. Concerning latency for small sharing levels, SCI, RDE, and STEM are identical for sizes 2, 3, and 4.

Concerning traffic for large sharing levels, RDE has about twice as much as the other two. This is the price that RDE pays for logarithmic latency. However, the traffic per node again levels off at a constant amount for all three. Note that RDE's write traffic, 5.22 messages per node, is less than what is computed in Sections 3.6.2.4 and 3.6.3, 5.50 messages per node (counting invalidation and purging, but not the generation of insurance pointers). This is because the simulation piggybacks simultaneous invalidate and purge requests, whereas the analysis in those sections assumes separate requests. Concerning traffic for small sharing levels, the three protocols are are equal for the common case of pairwise sharing, with and without the optimization. Since the common sharing sets are expected to be small, the three protocols should have similar amounts of write traffic for most applications.

## 6.4. Increasing System Size

In Section 6.3, we consider the latency and traffic while increasing the size of the sharing set. These results are independent of the workload and demonstrate the scalability of STEM and RDE for reads and writes. However, they do not predict how much a coherence protocol affects any parallel program. In this section, we construct a model workload and consider the latency and traffic as the system size increases. These results show that STEM and RDE perform significantly better than SCI for some workloads on large multiprocessors.

### 6.4.1. Workload Model

Our workload model is based on a frequency distribution of the sharing levels, where the frequency distribution is a function of the system size and an application program. We define the latency (traffic) of a workload to be the product of the latency (traffic) for a given sharing-set size and the cache-miss frequency for that size, summed over all sharing-set sizes. This gives the average latency (traffic) per cache miss for the given distribution. By choosing a frequency distribution that matches a given program, the relative performance impact of the cache-coherence protocols (for cache misses) can be approximated for that program. By changing the function of the frequency distribution, we can compare the performance of the coherence protocols for different sharing behaviors and characterize the workloads that perform badly for some coherence protocols.

Rather than designing a frequency distribution for a particular program, we construct a representative distribution that is based on the percentage of invalidating writes for several programs that are analyzed by Gupta and Weber [GuWe92]. It is true that the study of Gupta and Weber is limited to 32 processors. However, there are currently no sharing studies available with thousands of processors and the study of Gupta and Weber serves as a starting point from which we can vary our frequency distribution. Gupta and Weber characterize the invalidation patterns of the following five programs. Maxflow finds the maximum flow in a directed graph. MP3D simulates a three-dimensional wind tunnel using particle-based techniques. Water performs an $N$-body molecular dynamics simulation of the forces and potentials in a system of water molecules. PTHOR is a parallel logic simulator developed at Stanford University. LocusRoute is a global router for VLSI standard cells.

The footprints related to the percentage of invalidating writes for these five programs are considerably different. However, they can be broadly classified by the percentage of invalidating writes that involve over half of the caches, called *global sharing.* MP3D and LocusRoute exhibit no global sharing for large systems: their percentages respectively are 0.7 and 3.1 for system size 8, 0.1 and 0.3 for system size 16, and both 0.0 for system size 32. Maxflow's global sharing becomes vanishingly small for large systems: 2.1 percent for size 8, 0.9 percent for size 16, and 0.1 percent for size 32. Water appears to exhibit a constant percentage of global sharing: 1 percent for all three sizes. PTHOR appears to exhibit an increasing percentage of global sharing: 1.8 percent for size 8, 2.0 percent for size 16, and 2.8 percent for size 32.

The five programs can also be classified by the falloff rates of the remaining sharing-size frequencies. For all five programs, the percentages are steadily decreasing and appear to do so exponentially. After experimenting with several distributions, we discovered that the choice of this distribution does not significantly affect the results of this section, unless frequencies are unrealistically high. This is because the performance of the coherence protocols is approximately the same for small sharing levels. Therefore, it is acceptable to pick any reasonable frequency distribution for nonglobal sharing.

The workload is parameterized and the results are analytically derived from the data given in Section 6.3. Since most cache misses in massive multiprocessors are likely to be for shared data, we assume all of the misses are for shared data. This is a reasonable assumption for infinitely large caches and is probably within a factor of two for large finite caches. The percentage of global data is either 0 percent, 0.1 percent, 1 percent, or 10 percent of the shared data, using separate graphs for each. The remaining sharing frequencies are computed to satisfy the following ratios: the ratio of two-way and three-way sharing is 4 to 1, the ratio of three-way and four-way sharing is 2 to 1, and the ratio of $2^i$-way and $2^{i+1}$-way sharing for integer $i \geq 2$ is 8 to 1. Note that for computational simplicity, there is no $x$-way sharing for $x \geq 5$ and $x \neq 2^j$ for integer $j$. Except for the case of 10 percent global sharing, all of these distribution functions are within the range spanned by the five programs discussed above.

The data points are computed as follows. The average latency is

$$\sum_{s=2}^{N} freq(s) \cdot (read_{lat}(s) + write_{lat}(s)), \tag{6.5}$$

where $N$ is the system size, $freq(s)$ is the frequency of $s$-way sharing, $read_{lat}(s)$ is the latency for $s-1$ readers to get the data from the writer, and $write_{lat}(s)$ is the latency for the writer to invalidate the $s-1$ readers. $Freq(s)$ is given above, where $freq(N)$ is the percentage of global sharing. $Read_{lat}(s)$ and $write_{lat}(s)$ are given in Figures 6.1 and 6.2; if a data point is not available, semi-log interpolation is used. $Read_{lat}(s)$ for STEM uses the solid lines from Figure 6.1, which do not include the latency to finish tree merging after the data is distributed. Using the dotted lines would increase the latency by only about 5 percent for 65536 nodes. Furthermore, it is unlike that this latency would be on a critical path because the additional tree merging does not affect the validity of the available data. $Write_{lat}(s)$ for STEM uses the STEM lines from Figure 6.2, not the MME lines. The optimization for pairwise sharing is ignored for all three protocols and this does not qualitatively affect our results. Equation 6.5 gives the expected latency to share data.

In the execution of real programs, it is possible that not all of the reads for a given address occur simultaneously. This contributes error to our latency calculations. However, the results on read and write latencies in Figures 6.1 and 6.2 show that SCI differs substantially from RDE and STEM, both for reads and writes, so this source of error does not greatly affect our conclusions. Furthermore, if synchronization is used to order readers and writers, then it is very likely that all readers make their accesses at approximately the same time, after the synchronization. This reduces the quantitative error of our results and conclusions.

The average traffic per node is

$$\sum_{s=2}^{N} freq(s) \cdot (read_{traf}(s) + write_{traf}(s)), \tag{6.6}$$

where $N$ is the system size, $freq(s)$ is the frequency of $s$-way sharing, $read_{traf}(s)$ is the traffic per node for $s-1$ readers to get the data from the writer and finish tree merging, and $write_{traf}(s)$ is the traffic per node for the writer to invalidate the $s-1$ readers. $Freq(s)$ is given above, where $freq(N)$ is the percentage of global sharing. $Read_{traf}(s)$ and $write_{traf}(s)$ are given in Figures 6.1 and 6.2; if a data point is not available, semi-log interpolation is used. The optimization for pairwise sharing is ignored for all three protocols and this does not qualitatively affect our results. Equation 6.6 gives the expected traffic per node to share data. The possible lack of simultaneous reads does not affect our conclusions because 1) the traffic disparity for reads is greater than for writes, 2) nonsimultaneous reads would only reduce the traffic disparity for reads, and 3) our results show that the total disparity is minimal.

### 6.4.2. Workload Traffic

Figure 6.3 shows the average traffic per node, given by Equation 6.6. Although the difference in traffic is significant for a given sharing level, as shown in Section 6.3.2, the increase in average traffic per node for tree-based protocols over SCI is marginally significant; the difference

**Average Traffic Per Node, Given System Size**

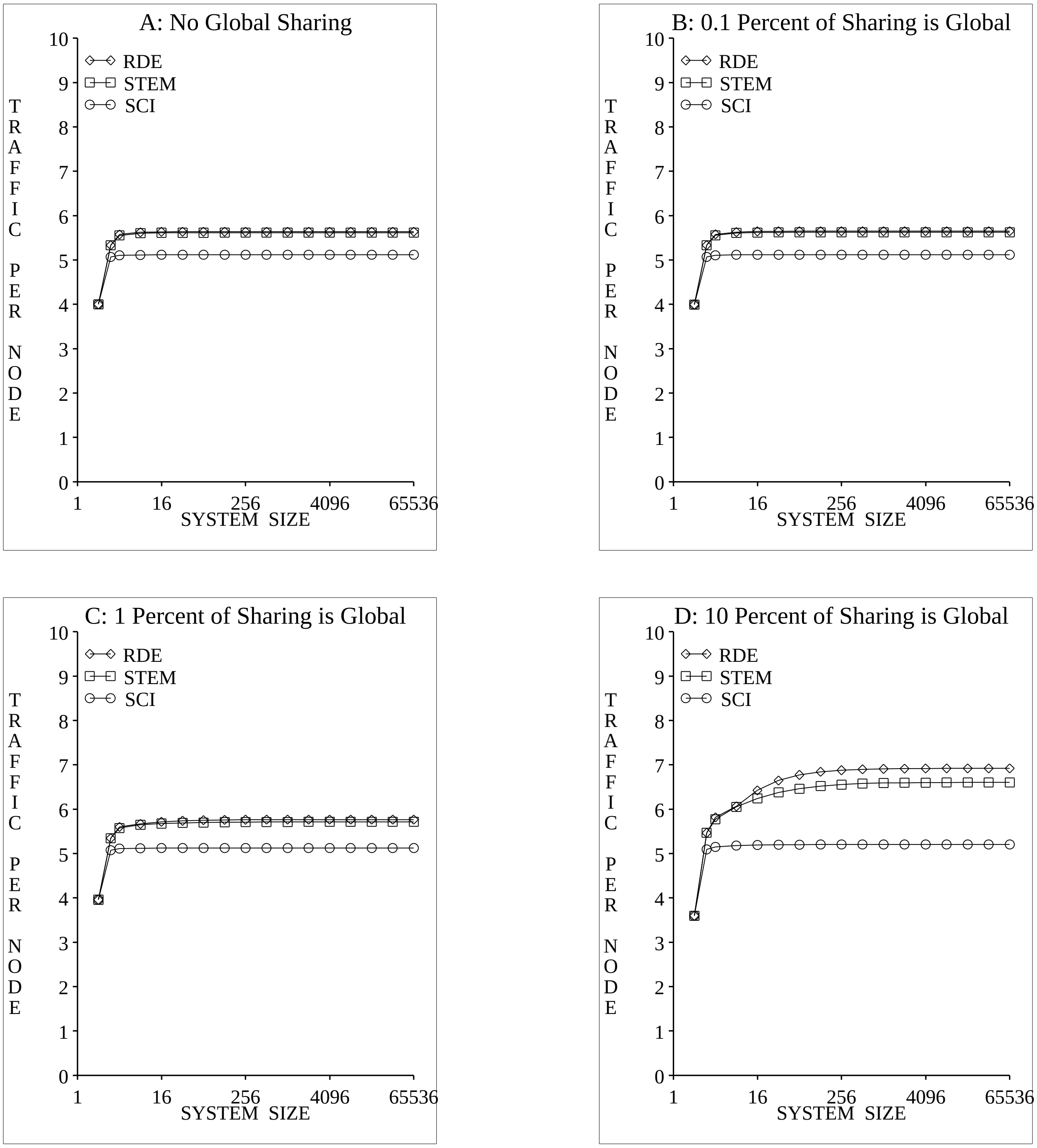



**Fig. 6.3:** These semi-log graphs show the average traffic per node as the amount of global sharing is varied. The keys order the coherence protocols with respect to a system size of 65536.

between RDE and STEM is not significant. This is because the traffic per node for a given sharing level is within a small constant factor (about 4 for reading and much less for writing) for all three coherence protocols, regardless of the system size, and because STEM and RDE have the similar amounts of read traffic. The small increase in traffic per node is the price paid for logarithmic latency.

Recall that the measure of traffic per node pertains to the number of messages sent per node, not the number of message passing through each node. This is why the traffic per node can remain constant and does not increase with system size. If we had multiplied the traffic per node by the average path length per message, then the graphs would have shown the relative growth in bandwidth that must accompany an increase in system size. However, this multiplication would not change the result that there is limited increase in the amount of traffic generated by RDE and STEM.

### 6.4.3. Workload Latency

Since there is a minimal increase in traffic for RDE and STEM and since combining is used to avoid tree saturation, it is reasonable to compare these protocols based on latency, the number of messages on the critical path. Figure 6.4 shows the average latency, given by Equation 6.5. With no global sharing, all latencies are approximately the same. STEM is the lowest by an insignificant amount due to the choice of frequency distribution for small sharing levels. However, if even a small percentage of the sharing is global, then SCI's latency increases linearly with system size, whereas the latencies of RDE and STEM remain relatively constant. With larger percentages of global sharing, the logarithmic factor becomes more important for RDE and STEM, as shown. SCI scales well through hundreds of nodes with applications that have less than 0.1 percent global sharing. However, SCI scales less well with larger amounts of global sharing, having problems with as few as 16 nodes when 10 percent of sharing is global.

In contrast to SCI, STEM and RDE scale well for all frequencies of global sharing. The performance differences between STEM and RDE for reading and writing are not significant. If 10 percent of the sharing is global, then RDE begins to show marginally better latency than STEM for system sizes over 1000. As expected, the complexities of STEM and RDE are not warranted for systems running applications that do not have global sharing. However, STEM

**Average Latency, Given System Size**

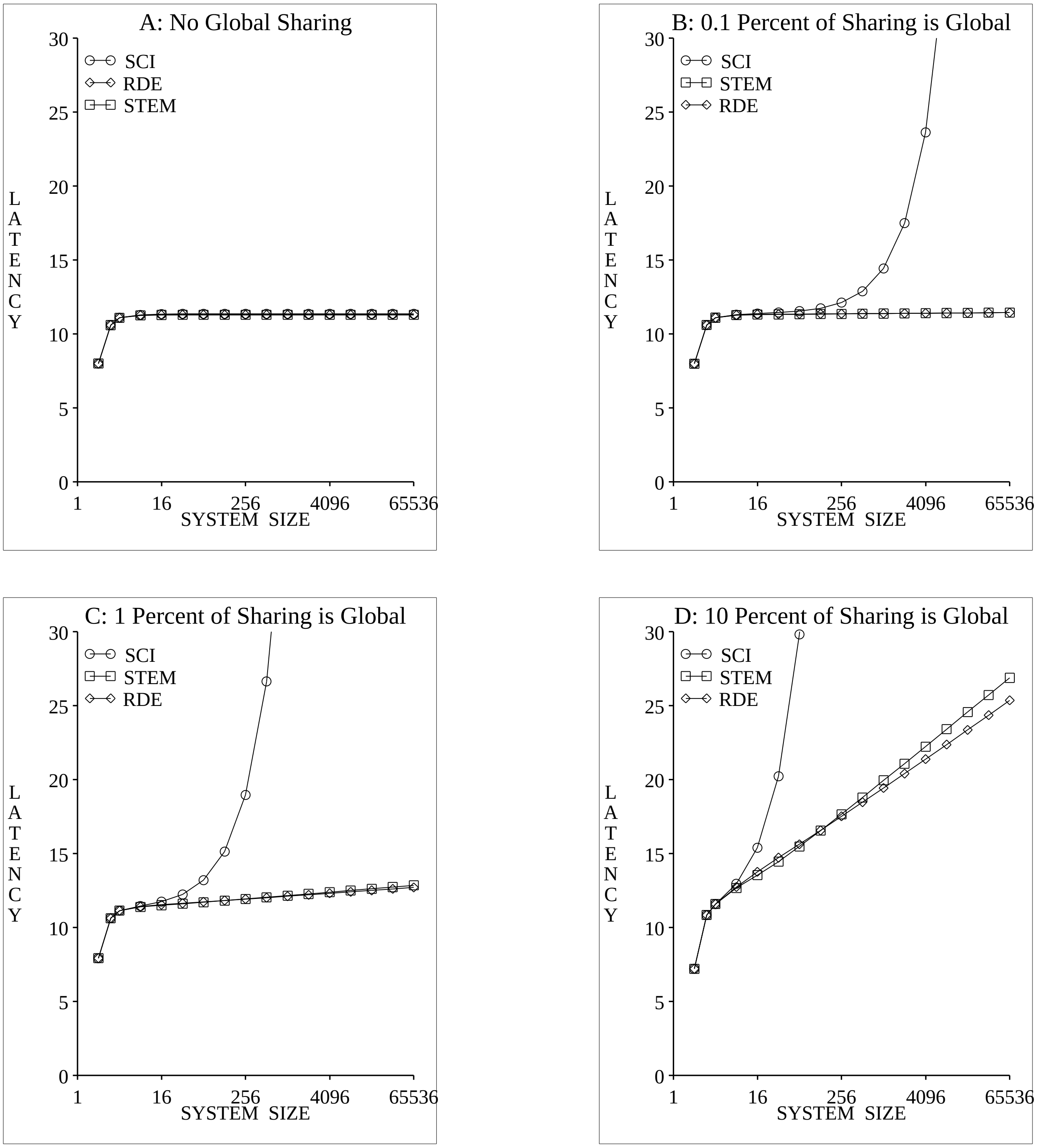



**Fig. 6.4:** These semi-log graphs show the average latency for sharing as the amount of global sharing is varied. The keys order the coherence protocols with respect to a system size of 65536.

and RDE have a significant performance advantage over SCI for kiloprocessor systems running applications that use even a little global sharing.

### 6.4.4. Workload Significance

In Sections 6.4.2 and 6.4.3, we showed that the traffic increases of RDE and STEM over SCI are minimal, but that the latency reductions are large for as little as 0.1 percent global sharing. Figure 6.5A shows the amount of global sharing per 100,000 parts such that SCI's latency is twice the latency of either RDE or STEM. For example, the latencies of RDE and STEM are about half of SCI's latency for 256 nodes and 1 percent global sharing, suggesting that systems with over 256 nodes should consider one of the tree-based protocols. With systems reaching SCI's maximum size of 64K nodes, tree-based protocols will significantly help performance of applications with as little as 0.005 percent global sharing.

Concerning traffic, Figure 6.5B shows the amount of global sharing per 100,000 parts such that SCI's traffic per node is half the traffic of either RDE or STEM. RDE and STEM do not double SCI's traffic until there exists applications with 42 and 57 percent global sharing respectively. Since we believe that these are unreasonable amounts of global traffic for any commercial application, this graph supports our conclusions that the traffic increase is minimal and that latency is the important metric.

In summary of the data in Section 6.4, the ratio of latencies between SCI and the other two grows as $\Theta(N / \log(N))$ for $N$ nodes, which is significant even for applications with diminutive amounts of global sharing. However, the traffic ratio is $\Theta(1)$, asymptotically approaching a small constant for large $N$; this implies that the traffic increase is minimal and certainly acceptable in light of the latency improvements. Concerning future work, it is important to note that the constant-factor differences in latency and traffic for a given sharing level, shown between RDE and STEM in Section 6.3, translate to marginal or negligible differences for a given system size. Since performance improvements of a small constant factor are not important, future work should focus on simplification of the protocols, not on fine tuning for performance.

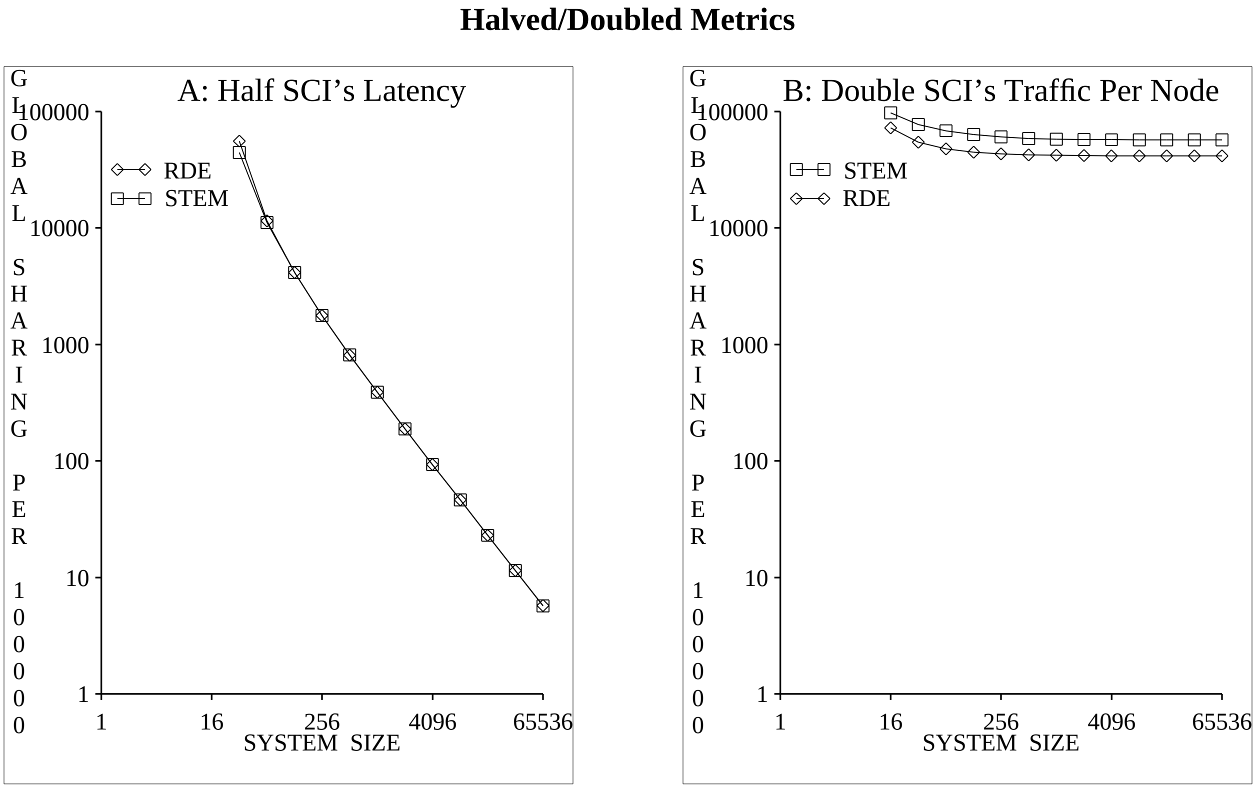



**Fig. 6.5:** These log-log graphs show the amount of global sharing per 100,000 parts such that latency or traffic per node is related to SCI by a factor of two. The keys order the coherence protocols according to the data.

---

## 6.5. Other Comparisons

There are a number of other ways to compare the three coherence protocols. First, consider the deletion latency. With small sharing levels, the average-case latency of a single deletion is about 3 messages for all three. With large sharing levels, the average-case latency of a single deletion is 4 for SCI and RDE and about 7 messages for STEM, as determined by simulation. The average deletion latency for STEM is greater because it keeps the tree intact, but it is constant because most nodes are leaves or close to leaves. If the STEM tree is perfectly balanced, which occurs whenever insertions are sequential, then the latency of one random deletion is bounded above by

$$\frac{2}{2} + \frac{6}{4} + \sum_{i=3}^{\infty} \frac{2i+4}{2^i} = 5.5. \tag{6.7}$$

Parallel deletions scale linearly for SCI and RDE because neighboring nodes must delete sequentially. In contrast, STEM scales logarithmically by allowing leaves to delete in parallel. This scaling is especially important if software selectively invalidates cache lines to help reduce write latency [ChVe88, HLRW92].

Second, consider the required storage per line. Recall that the protocols distribute the directory set throughout the caches so that the required storage is proportional to the sharing level. Of these three, SCI uses the least amount of directory storage with one 16-bit pointer at the directory and two pointers at the caches. STEM uses the next least amount, adding a third pointer and one 5-bit height field at the caches. RDE exchanges STEM's 5-bit height for a 16-bit counter at the caches and adds a 16-bit counter at the directory. The storage requirements make STEM more attractive than RDE.

Third, parts of the directory information become stale over time for RDE and STEM. The temporary pointers in RDE become stale as nodes are removed from the list because the temporary pointers are not updated. After a large number of replacements, the structure resembles a simple list with a few useless temporary pointers. Since the pcounts are also not updated, the structure is not repaired until after the next write. Not only are RDE's stale temporary pointers useless, they also add latency and traffic when they are used (and detected as stale). In STEM, the tree heights become stale over time. However, this mainly affects write latency in STEM and then only if subsequent rotate decisions are based on inflated heights. In a large tree, a large number of nodes must be removed before a STEM height becomes incorrect. In small trees, the effects of node removals may be more pronounced.

Fourth, RDE's combining is more complicated than the combining of SCI and STEM. The combining of SCI and STEM only creates a list, but RDE's combining must use adders in the network and create early (tree) pointers to facilitate efficient distribution of the pcounts. Furthermore, RDE's memory controller must keep track of the pcount, whereas STEM can use SCI's memory controller without modification. This makes STEM more appealing than RDE.

## 6.6. Comparison Summary

In this chapter, we compare the performance of three cache-coherence protocols, SCI, RDE, and STEM, with an emphasis on systems with thousands of caches. Our first performance study shows the efficiency of RDE and STEM for large sharing levels, demonstrating that their read and write latencies scale logarithmically with increasing sharing-set size. In contrast, SCI's read and write latencies scale linearly, causing significant performance degradation of applications with some global sharing. For all three protocols, the traffic per node for both reads and writes asymptotically approaches a constant for large sharing levels. The traffic per node for each of RDE and STEM is greater than SCI's traffic, which is the small price that is paid for logarithmic latency.

An examination of small-scale applications [GuWe92] suggests that some real applications exhibit no global sharing and others exhibit global sharing on the order of 1 percent. Although we agree that software can be rewritten to avoid global sharing, we argue that it is advantageous to build hardware that does not mandate this rewriting, thereby maximizing the reuse of software resources while minimizing programming effort.

In the second performance study, we compare the performance of the three protocols for various levels of global sharing. Regardless of the amount of global sharing, the three protocols are shown to have similar traffic per node as the system size increases. For applications that exhibit no global sharing, the three protocols have the same performance, both for latency and traffic. However, for applications with as little as 0.1 percent global sharing, SCI performs poorly in systems with thousands of nodes. With 65536 nodes, SCI performs poorly with as little as 0.005 percent global sharing. This is because SCI's latency scales linearly for applications with global sharing and the slope of the curve (at a point) increases with the amount of global sharing. In contrast, the latencies of RDE and STEM scale logarithmically such that increasing amounts of global sharing do not affect latency as much as for SCI. In any case, performance of SCI is approximately equal to or worse than the performance of RDE and STEM for all system sizes and future work should focus on simplification of STEM, rather than on the fine-tuning of STEM's performance.

We also compare the three protocols according to a number of other issues. First, all three protocols exhibit constant latency for one cache-line replacement, but STEM is the only one that scales logarithmically for multiple cache-line replacements. Second, SCI requires the least amount of storage per cache line and RDE requires the most, although the total storage space divided by the line size (64 bytes) is moderate for all protocols. Third, some of the structure information in RDE and STEM can become stale over time and hurt performance, more quickly in RDE than in STEM. Fourth, RDE's combining mechanism is more complicated than that of SCI and STEM.

We also discuss STP and DASH and argue that a direct comparison to the other protocols would be unfair. This is because SCI guarantees forward progress (no deadlock and no starvation) and robustness (the ability to recover from arbitrary transmission failures). Trying to extend STP and DASH to meet these constraints is very difficult, since some of the their performance optimizations are incompatible with forward progress and robustness. Trying to add DASH's optimizations to SCI is also difficult, even when removing SCI's constraints of robustness and no starvation. Furthermore, a fair comparison would consider the performance of the different combining mechanisms that are natural. The simulation and/or modeling of the different combining mechanisms would be very slow and it is not clear that there exists a scalable combining mechanism for DASH. Due to these difficulties, DASH and STP are not considered in our performance studies.

The choice of cache-coherence protocol depends on the amount of global sharing that is expected in the applications to be executed. On one hand, if there is no global sharing or if programmers are willing to rewrite software to avoid global sharing, then SCI is the best protocol. DASH is also a valid choice if occasional starvation can be tolerated and if recovery from network errors is not important. On the other hand, if the system will run applications with even 0.1 percent global sharing, perhaps much less for kiloprocessor systems, then STEM is the protocol of choice. RDE is also a valid choice if the following can be tolerated: combining complexity, simultaneous cache-line replacements, and additional cache-line storage. If the workload is unknown, then STEM is the correct choice, handling the eventuality of global sharing.

# Chapter 7

# Conclusions

## 7.1. Motivation

People covet more and more computing power and it is the job of computer architects and engineers to create this power. In 1641, Blaise Pascal (age 18) designed a computing machine capable of performing simple arithmetic calculations. In the 1940s, John von Neumann [BuGv46] proposed his well-known computer model with one processor and memory, where the memory stores both instructions and data[44] and the processor is composed of an arithmetic unit, registers, and an instruction counter. The foundations of computer architecture rest on this model.

The performance of a straightforward implementation of von Neumann's model is limited by sequential execution. One way to increase performance is to execute instructions in parallel. A good model is a Multiple-Instruction Multiple-Data (MIMD) multiprocessor, where multiple processors cooperate to solve the same problem. As discussed in Section 1.1, cooperating processes require a mechanism for sharing data. This mechanism must replicate data for good performance and manage the coherence of replicated data for correctness. However, it is challenging to provide data replication and coherence that is automatic, efficient, and cost-effective for thousands (or millions) of processors. Commercially successful machines of this size, like the Intel Hypercube, have avoided the problem by implementing separate memories per processor and a message-passing mechanism for communication, leaving data replication and coherence to the application programmer. Message-passing multiprocessors are commercially

[44]In contrast, a *Harvard Architecture* has separate memories for data and instructions.

successful and they are the right choice for some applications, such as nearest-neighbor algorithms with course-grain sharing [AGKP90].

However, we believe that some successful machines of the future will support the higher-level mechanism of shared memory, where an efficient view of a single memory is presented to the programmer through automatic replication and coherence-management of data stored in physically separate memories. In shared-memory multiprocessors, smaller and faster memories, called caches, are often used to replicate data that is accessed frequently. The algorithm that uses message-passing to maintain coherence of cached data is called a cache-coherence protocol. Shared memory provides a level of abstraction between the programmer and the message-passing hardware. To be efficient, the underlying message-passing mechanism must have low message delay and high bandwidth and the coherence protocol must limit the number of messages on the critical path, called latency, and the total number of messages, called traffic.

With the exception of combining, all previously proposed coherence protocols suffer from a bottleneck that prevents simultaneous access to recently changed data, as reviewed in Section 1.3. Previously proposed combining mechanisms [GGKM83, PBGH85, Scot92] save state in the network, constrain the return path of responses, and either slow normal accesses to data or use a separate (and expensive) combining network. Without combining, tree saturation [PfNo85] will delay all messages that cross the congested paths, even messages that are not destined for hot spots. There is a need for a cache-coherence protocol that allows simultaneous access to data without the above disadvantages of combining.

## 7.2. Contributions

The Scalable Coherent Interface (SCI) [JLGS90, Gust92a] is an IEEE standard (IEEE Std 1596-1992) [Comm91] for multiprocessors that specifies a topology-independent network and a cache-coherence protocol. The SCI coherence protocol is not efficient when large numbers of processors share frequently changing data because the coherence of cached data is managed with linked lists of cache lines, called sharing lists. Our goal is to investigate ways to efficiently share frequently changing data between thousands of processors and apply these results in the context of SCI. We make three primary contributions.

- In Chapter 2, we demonstrate that arbitrary network topologies can be constructed from SCI rings and we simulate the performance of several candidate topologies for uniform traffic. This demonstration gives insight into the ring-mapping and routing constraints of SCI, leading to a new solution for deadlock avoidance. Without an efficient network, it would be impossible for SCI to realize the performance advantages of the new cache-coherence protocols that we investigate.
- In Chapters 3 and 4, we give the first performance studies of recursive doubling [Comm90], a cache-coherence mechanism for generating and using temporary pointers, shortcuts across SCI's sharing lists. We invent and investigate several protocol variations for request combining and temporary-pointer structures. We give theoretical lower bounds on latency for numerous sets of assumptions and compare these against the protocol variations.
- In Chapters 5 and 6, we give a new cache-coherence protocol, called STEM (permuted acronym for Tree Merging Extensions to SCI) and study its performance. STEM manages cache coherence as trees of cache lines. We investigate several protocol variations and compare their performance against the appropriate lower bounds that we derive. We discuss forward-progress and hot-spot problems of other protocols. We compare the performance of STEM to that of SCI and recursive doubling. We show that STEM is compatible with SCI networks, STEM's traffic increase over SCI is minimal, and STEM has equal or lower latency than SCI for all sharing levels.

An arbitrary topology can be constructed from an interwoven set of SCI rings. SCI guarantees forward progress on a single ring, but forward progress is not explicitly guaranteed on sets of interwoven rings. The normal deadlock-avoidance solution [Tane81] is to partition the queues into classes and restrict space assignment to provide a partial ordering on the use of the classes. Instead, it is possible to restrict the routing for some topologies, like $k$-ary $n$-cubes, such that deadlock avoidance does not mandate partitioning of queues (or virtual channels [DaSe87]). Not only does this result simplify the implementation of SCI topologies, but it provides better utilization of queue space and decreases the probability of tree saturation [PfNo85] that leads to poor performance. Concerning topology candidates and assuming uniform traffic, we show that some form of multistage network topology is best in general, but that the $k$-ary $n$-cube is competitive for small networks with small switch sizes.

The synchronous recursive doubling of Hillis and Steele [HiSt86] and Stone [Ston73] can be modified to work asynchronously while reducing its cache-coherence traffic from $\Theta(N \log(N))$ to $\Theta(N)$, where $N$ is the number of shared copies. This involves generating a good temporary-pointer set; one set augments the sharing list to resemble a binary tree, called the binary structure (Section 3.5.1.2). Since recursive doubling is decoupled from data distribution, the technique of request reserving is used to avoid spin waiting for data, which can be a problem in SCI. Request forwarding can also be used to improve performance, provided that deadlock situations are avoided, but the performance improvement does not justify the additional complexity. In order to prepare for recursive doubling, request combining is necessary to efficiently create preliminary trees through which list positions are distributed. To do this, we augment the combining mechanism of James et al. [JLGS90], which creates a sharing list without congesting the memory controller. Furthermore, by waiting to combine requests, called patient combining, the average preparation latency is reduced by at least 35 percent and the worst-case latency is improved from $\Theta(N)$ to $O(N^{1/2})$.

A study of lower bounds on recursive doubling measures the optimality of a proposed solution for some given constraints. Our studies show that the symmetric structure of Section 3.5.1.1 is not optimal, but that the binary structure of Section 3.5.1.2 is near optimal. A study of lower bounds on recursive doubling also identifies variations of recursive doubling that show promise for further investigation and yields insight into what are the difficult parts of recursive doubling. The number of temporary pointers per cache line is more important than the issues of circular lists, unidirectional information flow, and bidirectional pointers. If recursive doubling is extended, it should first be extended to work with multiple temporary pointers per cache line. The sum of the lower bounds on temporary-pointer creation and data distribution is about the same as the lower bound on these two phases executed in parallel. This implies that no optimal algorithm can hope to significantly overlap these two latencies and, therefore, further exploration of this is futile. Another lower-bounds result is that using the same temporary pointers for data and invalidate distribution necessarily increases the latency of both. This implies that there does not exist one structure that is (near) optimal for each multicast.

STEM is a new cache-coherence protocol for multiprocessors with thousands of processors. Tree merging constructs a binary tree from a linked-list of cache lines by using single AVL rotations [AdLa62]. Trees are probabilistically balanced by the strictly growing property, which is applied to the left side of the tree (originally the list). The original list may be created by using the combining mechanism of James et al. [JLGS90]. Concurrent execution of any associative update [GoLR83], including fetch-and-add and write, can be implemented without changing the network's combining mechanism and without violating sequential consistency [Lamp79]. STEM's directory is mostly ignorant of the coherence protocol, simply responding to requests. This makes STEM's directory no more complicated than any other protocol and allows STEM to use SCI's memory controller.

Concerning variations on STEM, we consider the advantage of asynchronous data forwarding, the cost-performance of the recent-height optimization, and the lower bounds on latency. Forwarding data asynchronously with respect to tree merging is a definite performance advantage and should be implemented. Maintaining the right height of the forward neighbor does not decrease latency enough to justify the storage, so this optimization is not warranted. Concerning lower bounds, the strictly growing property results in average latency that is within 30 percent of the optimal average latency for tree merging. Furthermore, the lower bound is achieved for the common cases of small sharing levels, so there is no reason to search for a better tree-merging configuration. Although the worst-case latency for STEM is linear in the number of cached copies, the probability of the worst case (or any other disastrous deviation from the average) is negligible.

The performance of STEM compares favorably to other cache-coherence protocols. For large sharing levels (hundreds or thousands of copies), DASH and STP suffer from hot spots. STEM's latency is comparable to the recursive-doubling latency and it is equal to or better than SCI's latency. For small sharing levels of at least three, directory protocols, like the Stanford DASH [LLGG90], perform better than STEM because requests can be pipelined at the directory. However, DASH and STP do not guarantee forward progress on SCI networks, making this comparison unfair. In any case, for the common case of pairwise sharing, SCI and STEM perform better than DASH-like protocols because two caches can share data without consulting the directory. STEM is the extension of choice for SCI because STEM has good performance, STEM can combine associative updates without the traditional network complexity, and STEM has equal or better latency than SCI for all sharing levels.

### 7.3. Future Work

At this point, it is time to take a committing step toward construction of a shared-memory multiprocessor with thousands of processors. We believe that this multiprocessor should have a programmable cache controller so that the viability of multiple cache-coherence protocols like STEM can be defended by real programs running on large machines. Since STEM's directory, and directories of other distributed-pointer protocols, are relatively ignorant of the specific cache-coherence protocol, memory and network hardware need not change as complex cache-coherence protocols are invented, tested, and enhanced. Since the combining primitive and the common case of access to private data can be implemented simply and completely in hardware, such a system should be commercially feasible.

In parallel with hardware development, STEM's C specification needs to be debugged. This can begin with uniprocessor simulators and then graduate to small virtual prototypes [HLRW92, RHLL92] or the real hardware. Since the coherence protocol is decoupled from the network and directory hardware, construction of hardware can proceed at full speed without regard to the status of STEM's specification.

The SCI working group (P1596.2) is considering ways to extend STEM to do write update. This work should be continued and completed. Furthermore, when writes are not atomic, it is important to know how different memory models can help the programmer reason with (or without) sequential consistency [Lamp79, DuSB86, AdHi90, GLLG90, Netz91, GAGH92]. What information does the hardware and/or compiler require from the programmer and what debugging tools can help the programmer find data races [AHMN91]?

The same working group should also look for ways to simplify STEM. A detailed comparison to RDE and SCI shows that reducing SCI's latency from linear to logarithmic in the number of participating cache is extremely important. However, the same studies show that a constant-factor performance improvement of about 20 percent for a large sharing-set size, like 64K, is not important. In view of this, effort should be focused on reducing the complexity of STEM. Perhaps a node should always accept a rotate request, instead of rejecting it when it come from a node with a smaller right height. This may increase the resulting tree height, but it would eliminate a decision in the cache controller. Furthermore, complexity in the network should be avoided. Currently, STEM uses the combining mechanism of James et al. [JLGS90], which is compatible with SCI's memory controller. Destroying this perfect fit for sake of a constant-factor performance improvement would be a grave error. Simplification of the cache controller, at the expense of the combining mechanism, would be also subject to ridicule. This is because 1) it is conceivable that the complexity of the cache controller can be handled in a programmable entity, whereas the complexity of the network does not yield to this technique as easily, 2) the effectiveness of combining is proportional to the rate at which combining can take place, which is proportional to the complexity of the combining mechanism, and 3) the complexity of the combining mechanism may affect the flow rate of normal messages through the combining queues.

In this dissertation, we have given a hardware alternative for combining fetch-and-add by using the cache controllers to distribute the results. Is it useful to do this combining in hardware, or is software combining [YeTL87, GoVW89, YeTa90] sufficient for efficient programs? For what sharing levels does hardware combining become useful, if any?

In Appendix A, we give a new synchronization primitive, swap-if-zero, that is both universal [Herl91] and combinable. Because swap-if-zero is universal, unlike fetch-and-add, it can be used to construct a wait-free implementation of any object [Herl91]. Can combinable universal primitives, like swap-if-zero, implement wait-free objects more efficiently than noncombinable universal primitives, like compare-and-swap?

We believe it is time to build shared-memory multiprocessors with thousands of processors. Towards this end, this dissertation demonstrates the feasibility of implementing efficient (parallel) access to rapidly changing data.

# Appendix A

## Swap-If-Zero is Universal and Combinable

It is not straightforward to choose the correct synchronization primitives for implementation in shared-memory MIMD machines. Universal synchronization primitives, like compare-and-swap, are sufficient for implementation of wait-free access to concurrent objects. Combinable synchronization primitives, like fetch-and-add, have implementations that avoid serialization of multiple accesses to the same variable. Unfortunately, compare-and-swap is not known to be combinable and fetch-and-add is not universal. We present a new synchronization primitive, called swap-if-zero, and prove that it is both universal and combinable. Swap-if-zero is a new synchronization primitive that is known to have both properties.

### A.1. Introduction

It is not straightforward to choose the correct synchronization primitives for shared-memory MIMD machines. A computer architect must weigh the function and performance of a proposed primitive against the costs of its possible implementations. For example, can a proposed primitive implement parallel access to a queue such that 1) the halting of one process does not prevent forward progress of other processes and 2) the primitive has a simple hardware implementation?

A *wait-free* implementation of a concurrent object is one that guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes [Herl91]. The use of critical sections, for example, is prohibited because a process in a critical section may be delayed for any number of reasons, including page faults, process swapping, and hardware faults. Fault-tolerant systems require wait-free implementations of concurrent objects.

A synchronization primitive (or a set of synchronization primitives) $X$ is *universal* if there exists a wait-free implementation of any concurrent object while limiting process communication to reads, writes, and primitive (set) $X$. Herlihy [Herl91] proves that memory-to-memory move and swap, augmented queue, compare-and-swap, and fetch-and-cons are each universal. However, it is not clear that any of these has a feasible hardware implementation that does not serialize operations on the same variable. This is a problem for large multiprocessors, where sequential execution of anything substantial dramatically reduces performance.

Combining can prevent the serialization of multiple accesses to the same variable, thereby avoiding tree saturation [PfNo85]. On one hand, it is not known how to combine any of Herlihy's universal primitives [Herl91] because none of these primitives are associative; associativity is a sufficient condition for combinability. On the other hand, fetch-and-add is one combinable primitive that has low-latency implementations [GGKM83, PBGH85, DiKe92] and numerous applications [GoLR83, Ston84], leading Gottlieb et al. [GoLR83] to conjecture that fetch-and-add is universal. However, fetch-and is not universal, as proved by Herlihy [Herl91].

Does there exist a synchronization primitive (or set) that is both universal, for maximum functionality, and combinable, for hot-spot efficiency? According to Herlihy [Herl91], a necessary condition for a universal set $S$ of primitives is that there exists a pair of functions (possibly the same), $f$ and $g \in S$, such that the two functions do not commute, $\exists\, v : f(g(v)) \neq g(f(v))$, and the functions do not overwrite each other, $\exists\, v : f(g(v)) \neq f(v)$ and $\exists\, v : g(f(v)) \neq g(v)$. A sufficient condition for a combinable primitive is that it is associative [GoLR83]. These results suggest that a universal and combinable primitive may exist as one that satisfies these two conditions. Swap-if-zero is one synchronization primitive that satisfies these conditions. Swap-if-zero is an atomic read-modify-write operation on one variable, defined in Figure A.1. Essentially, swap-if-zero is compare-and-swap with a comparison value that is always zero. If the current memory value is zero, then it is replaced by the given operand. In any case, the old value of memory is returned.

Our contribution is the discovery of swap-if-zero and the proof that it is both universal and combinable. Section A.2 proves this result. Section A.3 gives directions for future work.

**Swap-If-Zero Definition**

```
int SwapIfZero(int swap, int *variable)
{
  int temp = *variable;
  if (*variable == 0)
    *variable = swap;
  return(temp);
}
```

**Fig. A.1:** This figure defines the atomic operation of swap-if-zero in C.

---

### A.2. Theory

First, we show that swap-if-zero is universal. From Herlihy [Herl91], a synchronization primitive is universal if it can be used with atomic reads and writes to implement a wait-free consensus protocol for $n$ processes. Informally, a consensus protocol for $n$ processes is a system of $n$ processes that each begin with a unique identifier from some domain. The processes then asynchronously communicate through a shared variable $v$ (initially zero), agree upon one of the identifiers, and then halt. A more formal definition is given by Herlihy [Herl91]. Incidentally, Herlihy shows that an $n$-process consensus protocol can be used to implement wait-free fetch-and-cons and that fetch-and-cons can be used to implement any universal primitive.

Therefore, to show that swap-if-zero is universal, we need only show that swap-if-zero can be used to implement a wait-free consensus protocol for $n$ processes. Let the $n$ processes begin with number from the domain of non-zero integers. Each processes executes one swap-if-zero on $v$, using its unique identifier as the operand. The first operation changes the value to its operand and all of the subsequent operations have no effect. The first operation returns zero and all of the others return the operand of the first operation. If the swap-if-zero for a process returns zero, the process decides on its own operand and halts. Otherwise, the process decides on the return value and halts. Since there are no loops, all processes eventually halt and agree on the operand of the first swap-if-zero. Therefore, swap-if-zero is universal.

Second, we show that swap-if-zero is combinable. Let $v$ be the pointer to the variable and let $p$ and $q$ be the operands of two swap-if-zeros respectively issued by processes $P$ and $Q$.

Then, $SwapIfZero\,(p,\ v)$ followed by $SwapIfZero\,(q,\ v)$ is equivalent to $SwapIfZero\,(r,\ v)$, where

$$r = \begin{cases} p, & \text{if } p \neq 0 \\ q, & \text{if } p = 0 \end{cases} \tag{A.1}$$

and $P$ and $Q$ respectively receive $v$ and $SwapIfZero\,(p,\ v)$. Therefore, swap-if-zero is combinable.

### A.3. Future Work

Swap-if-zero, which directly implements *n*-process consensus, is a new synchronization primitive that is both universal and combinable. However, Herlihy [Herl90, Herl91] observes that it is cumbersome to construct wait-free implementations of objects, given only *n*-process consensus. Are there other primitives that are both universal and combinable?

Herlihy [Herl90] gives a simple methodology for using compare-and-swap to construct wait-free implementations of concurrent objects. Although compare-and-swap is not combinable, it is straightforward to implement in hardware. Unfortunately, Herlihy does not show how to avoid starvation when a process immediately requests a new object operation after the previous one completes. Without the ability to reuse objects, this methodology is not immediately applicable to real parallel programs. What is a more comprehensive methodology for swap-if-zero and/or compare-and-swap that is more simple than direct reduction from Herlihy's proofs [Herl91]? Alternately, how can wait-free work queues be efficiently implemented with swap-if-zero and/or compare-and-swap?

In summary, swap-if-zero is both universal and combinable. The possible problem with swap-if-zero is that it may be a bad primitive for efficient implementation of wait-free objects. It can do the job correctly because it is universal. However, there is currently no good reason to support it in hardware because there is no known way to use it efficiently and make use of its combinability. The fact that swap-if-zero is both universal and combinable is an interesting theoretical result with no immediate application to computer architecture.

# References


[AdLa62] G. M. Adel'son-Vel'skiĭ and E. M. Landis, ''An Algorithm for the Organization of Information,'' *Soviet Mathematics Doklady* **3**, 5 (September 1962), 1259-1262. Russian original in Doklady Akademii NAUK SSSR, TOM 146, Nos. 1-6, 263-266.

[AdHi90] Sarita V. Adve and Mark D. Hill, ''Weak Ordering - A New Definition,'' *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture* **18**, 2 (May 1990), 2-14.

[AHMN91] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer, ''Detecting Data Races on Weak Memory Systems,'' *Proceedings of the Eighteenth Annual International Symposium on Computer Architecture* **19**, 3 (May 1991), 234-243.

[ASHH88] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz, ''An Evaluation of Directory Schemes for Cache Coherence,'' *Proceedings of the Fifteenth Annual International Symposium on Computer Architecture* **16**, 2 (May 1988), 280-289.

[AlGo89] George S. Almasi and Allan Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1989.

[Amda67] G. M. Amdahl, ''Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities,'' *AFIPS Proceedings of the 1967 Spring Joint Computer Conference,* April 1967, 483-485.

[AGKP90] S. B. Anderson, P. W. Gorham, S. R. Kulkarni, T. A. Prince, and A. Wolszczan, ''Discovery of Two Radio Pulsars in the Globular Cluster M15,'' *Nature* **346**, 6279 (July 1990), 42-44.

[ArBa84] James Archibald and Jean-Loup Baer, ''An Economical Solution to the Cache Coherence Problem,'' *Proceedings of the Eleventh Annual International Symposium on Computer Architecture* **12**, 3 (June 1984), 355-362. This paper is not in the 12th ISCA, as is commonly cited.

[ArBa86] James Archibald and Jean-Loup Baer, ''Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model,'' *ACM Transactions on Computer Systems* **4**, 4 (November 1986), 273-298.

[AKLM89] M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. L. Miller, and S-H. Teng, ''Constructing Trees in Parallel,'' *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures (SPAA '89),* June 1989, 421-431.

[BaWa88] Jean-Loup Baer and Wen-Hann Wang, ''On the Inclusion Properties for Multi-level Cache Hierarchies,'' *Proceedings of the Fifteenth Annual International Symposium on Computer Architecture* **16**, 2 (May 1988), 73-80.

[BaSc77] R. Bayer and M. Schkolnick, ''Concurrency of Operations on *B*-Trees,'' *Acta Informatica* **9**, 1 (1977), 1-21.

[Bian89] Ronald Bianchini, ‘‘Ultracomputer Packaging and Prototypes,’’ Ultracomputer Note #152, Ultracomputer Research Laboratory, Courant Institute, NYU, January 1989.

[BrHo90] Eugene D. Brooks III and Joseph E. Hoag, ‘‘A Scalable Coherent Cache System with Incomplete Directory State,’’ *Proceedings of the 1990 International Conference on Parallel Processing (ICPP ’90)*, August 1990, I-553-554.

[Burk92] Henry Burkhardt, III, ‘‘Kendall Square Research Technical Summary,’’ Technical Report, Kendall Square Research, 1992. Contact Susan Parrish (info@ksr.com) for current information.

[BuGv46] Arthur W. Burks, Herman H. Goldstine, and John von Neumann, ‘‘Preliminary Discussion of the Logical Design of an Electronic Computing Instrument,’’ in *Computer Structures: Readings and Examples*, C. G. Bell and A. Newell (eds.), McGraw-Hill, New York, NY, 1971, 92-119. Taken from Report to U.S. Army Ordinance Department, 1946.

[CaKP91] David Callahan, Ken Kennedy, and Allan Porterfield, ‘‘Software Prefetching,’’ *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, April 1991, 40-52.

[CeFe78] Lucien M. Censier and Paul Feautrier, ‘‘A New Solution to Coherence Problems in Multicache Systems,’’ *IEEE Transactions on Computers* **27**, 12 (December 1978), 1112-1118.

[CFKA90] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal, ‘‘Directory-Based Cache Coherence in Large-Scale Multiprocessors,’’ *IEEE Computer* **23**, 6 (June 1990), 49-58.

[ChKA91] David Chaiken, John Kubiatowicz, and Anant Agarwal, ‘‘LimitLESS Directories: A Scalable Cache Coherence Scheme,’’ *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, April 1991, 224-234.

[ChVe88] Hoichi Cheong and Alexander V. Veidenbaum, ‘‘A Cache Coherence Scheme With Fast Selective Invalidations,’’ *Proceedings of the Fifteenth Annual International Symposium on Computer Architecture* **16**, 2 (May 1988), 299-307.

[ChGB91] David R. Cheriton, Hendrik A. Goosen, and Patrick D. Boyle, ‘‘Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture,’’ *IEEE Computer* **24**, 2 (February 1991), 33-46.

[Comm90] P1596 Ballot Review Committee of the IEEE Microprocessor Standards Committee, ‘‘The Extended SCI Cache-Coherence Protocols,’’ P1596/D0.74/Part-III-C, October 1990.

[Comm91] P1596 Ballot Review Committee of the IEEE Microprocessor Standards Committee, ‘‘SCI - Scalable Coherent Interface,’’ P1596/D2.00 18Nov91. Draft for Recirculation to the Balloting Body. This is being edited by the IEEE for publication in early 1993 as IEEE Std 1596-1992. This draft (or the final standard) can be ordered from the IEEE Service Center at 1-800-678-4333 (US) or ++908-562-5420 (fax ++908-981-9667), or contact David B. Gustavson (dbg@slac.stanford.edu) for current status.

[DaSe87] William J. Dally and Charles L. Seitz, ‘‘Deadlock-Free Message Routing in Multiprocessor Interconnection Networks,’’ *IEEE Transactions on Computers* **36**, 5 (May 1987), 547-553.

[Dall90] William J. Dally, ‘‘Performance Analysis of *k*-ary *n*-cube Interconnection Networks,’’ *IEEE Transactions on Computers* **39**, 6 (June 1990), 775-785.

[Deer88] Stephen E. Deering, ‘‘Multicast Routing in Internetworks and Extended LANs,’’ *Proceedings of the 1988 SIGCOMM Symposium*, August 1988, 55-64.

[DePI86] Eliezer Dekel, Shietung Peng, and S. Sitharama Iyengar, ‘‘Optimal Parallel Algorithms for Constructing a Balanced m-way Search Tree,’’ *Proceedings of the 1986 International Conference on Parallel Processing (ICPP ’86)*, August 1986, 1010-1012.

[DiKe92] Susan R. Dickey and Richard Kenner, ‘‘Hardware Combining and Scalability,’’ *Proceedings of the 1992 ACM Symposium on Parallel Algorithms and Architectures (SPAA ’92)*, June 1992, 296-305.

[DuSB86] Michel Dubois, Christoph Scheurich, and Faye Briggs, ‘‘Memory Access Buffering in Multiprocessors,’’ *Proceedings of the Thirteenth Annual International Symposium on Computer Architecture* **14**, 2 (June 1986), 434-442.

[DuSB88] Michel Dubois, Christoph Scheurich, and Fayé A. Briggs, ‘‘Synchronization, Coherence, and Event Ordering in Multiprocessors,’’ *IEEE Computer* **21**, 2 (February 1988), 9-21.

[DuSc90] Michel Dubois and Christoph Scheurich, ‘‘Memory Access Dependencies in Shared-Memory Multiprocessors,’’ *IEEE Transactions on Software Engineering* **16**, 6 (June 1990), 660-673.

[EgKa88] Susan J. Eggers and Randy H. Katz, ‘‘A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation,’’ *Proceedings of the Fifteenth Annual International Symposium on Computer Architecture* **16**, 2 (May 1988), 373-382.

[Elli80a] Carla Schlatter Ellis, ‘‘Concurrent Search and Insertion in 2-3 Trees,’’ *Acta Informatica* **14**, 1 (1980), 63-86.

[Elli80b] Carla Schlatter Ellis, ‘‘Concurrent Search and Insertion in AVL Trees,’’ *IEEE Transactions on Computers* **29**, 9 (September 1980), 811-817.

[Enco86] ‘‘Multimax Technical Summary,’’ Technical Report, Encore Computer Corporation, 1986.

[Feng81] Tse-yun Feng, ‘‘A Survey of Interconnection Networks,’’ *IEEE Computer* **14**, 12 (December 1981), 12-27.

[FoCa84] Ray Ford and Jim Calhoun, ‘‘Concurrency Control Mechanisms and the Serializability of Concurrent Tree Algorithms,’’ *Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, April 1984, 51-60.

[FoWy78] S. Fortune and J. Wyllie, ‘‘Parallelism in Random Access Machines,’’ *Proceedings Tenth Annual ACM Symposium on Theory of Computing*, 1978, 114-118.

[FrWB85] Ariel J. Frank, Larry D. Wittie, and Arthur J. Bernstein, ‘‘Multicast Communication on Network Computers,’’ *IEEE Software* **2**, 3 (May 1985), 49-61.

[GLLG90] Kourosh Gharachorloo, Anoop Gupta, Daniel Lenoski, James Laudon, Phillip Gibbons, and John L. Hennessy, ‘‘Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors,’’ *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture* **18**, 2 (May 1990), 15-26.

[GAGH92] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill, ‘‘Programming for Different Memory Consistency Models,’’ *Journal of Parallel and Distributed Computing* **15**, 4 (August 1992), 399-407.

[GjKM89] Stein Gjessing, Stein Krogdahl, and Ellen Munthe-Kaas, ‘‘Formal Specification and Verification of SCI Cache Coherence,’’ *Norwegian Informatics Conference (NIK ’89)*, Stavanger, Norway, November 1989. Appears Research Report Number 142, Department of Informatics, University of Oslo, Norway, August 1990.

[GjKM90a] Stein Gjessing, Stein Krogdahl, and Ellen Munthe-Kaas, ‘‘Approaching Verification of the SCI Cache Coherence Protocol,’’ Research Report Number 145, Department of Informatics, University of Oslo, Norway, August 1990.

[GjKM90b] Stein Gjessing, Stein Krogdahl, and Ellen Munthe-Kaas, ‘‘A Top Down Approach to the Formal Specification of SCI Cache Coherence,’’ Research Report Number 146, Department of Informatics, University of Oslo, Norway, August 1990.

[Good83] James R. Goodman, ‘‘Using Cache Memory to Reduce Processor-Memory Traffic,’’ *Proceedings of the Tenth Annual International Symposium on Computer Architecture*, June 1983, 124-131.

[GoWo88] James R. Goodman and Philip J. Woest, ‘‘The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor,’’ *Proceedings of the Fifteenth Annual International Symposium on Computer Architecture* **16**, 2 (May 1988), 422-431.

[GoVW89] James R. Goodman, Mary K. Vernon, and Philip J. Woest, ‘‘Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors,’’ *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)* **17**, 2 (April 1989), 64-75.

[Good91] James R. Goodman, ‘‘Cache Consistency and Sequential Consistency,’’ Technical Report #1006, Computer Sciences Department, University of Wisconsin-Madison, February 1991. Appears Technical Report Number 61, IEEE Scalable Coherent Interface (SCI), March 1989.

[GoLR83] Allan Gottlieb, G. D. Lubachevsky, and Larry Rudolph, ‘‘Basic Techiniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors,’’ *ACM Transactions on Programming Languages and Systems* **5**, 2 (April 1983), 164-189.

[GGKM83] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir, ‘‘The NYU Ultracomputer—Designing an MIMD

Shared Memory Parallel Computer,'' *IEEE Transactions on Computers* **32**, 2 (February 1983), 175-189.

[GuWM90] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry, ''Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes,'' *Proceedings of the 1990 International Conference on Parallel Processing (ICPP '90)*, August 1990, I-312-321.

[GHGM91] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber, ''Comparative Evaluation of Latency Reducing and Tolerating Techniques,'' *Proceedings of the Eighteenth Annual International Symposium on Computer Architecture* **19**, 3 (May 1991), 254-263.

[GuWe92] Anoop Gupta and Wolf-Dietrich Weber, ''Cache Invalidation Patterns in Shared-Memory Multiprocessors,'' *IEEE Transactions on Computers* **41**, 7 (July 1992), 794-810.

[Gust88] John L. Gustafson, ''Reevaluating Amdahl's Law,'' *Communications of the Association for Computing Machinery (CACM)* **31**, 5 (May 1988), 532-533.

[Gust91] David B. Gustavson, ''Minutes of the Working Group for the Scalable Coherent Interface,'' ftp access, August 1991. Available at hplsci.hpl.hp.com (15.255.176.57) in directory pub/sci/minutes_word. It is also available as SCI-doc-235, which is part of mailing M25 (along with the approved D2.00 of the SCI specification), from Kinko's copy service, 415-328-3381, Palo Alto, California.

[Gust92a] David B. Gustavson, ''The Scalable Coherent Interface and Related Standards Projects,'' *IEEE Micro* **12**, 2 (February 1992), 10-22.

[Gust92b] David B. Gustavson, *personal correspondence*, November 1992. E-mail: dbg@slac.stanford.edu.

[HeHL88] Sandra M. Hedetniemi, Stephen T. Hedetniemi, and Arthur L. Liestman, ''A Survey of Gossiping and Broadcasting in Communication Networks,'' *Networks* **18**, 4 (1988), 319-349.

[Herl90] Maurice Herlihy, ''A Methodology for Implementing Highly Concurrent Data Structures,'' *ACM SIGPLAN Notices* **25**, 3 (March 1990), 197-206.

[Herl91] Maurice Herlihy, ''Wait-Free Synchronization,'' *ACM Transactions on Programming Languages and Systems* **13**, 1 (January 1991), 124-149.

[Hill90] Mark D. Hill, ''What is Scalability?,'' *ACM SIGARCH Computer Architecture News* **18**, 4 (December 1990), 18-21.

[HLRW92] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood, ''Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors,'' *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, October 1992, 262-273.

[HiSt86] W. Daniel Hillis and Guy L. Steele, Jr., ''Data Parallel Algorithms,'' *Communications of the Association for Computing Machinery (CACM)* **29**, 12 (December 1986), 1170-1183.

[JLGS90] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi, ‘‘Scalable Coherent Interface,’’ *IEEE Computer* **23**, 6 (June 1990), 74-77.

[Jame90] David V. James, *correspondence at SCI meeting*, September 1990. E-mail: dvj@apple.com.

[Jame93] David V. James, *personal correspondence*, January 1993. E-mail: dvj@apple.com.

[John91] Ross E. Johnson, ‘‘Lower Bounds on Latency for Scalable Linked-List Cache Coherence,’’ Technical Report #1029, Computer Sciences Department, University of Wisconsin-Madison, June 1991.

[JoGo91a] Ross E. Johnson and James R. Goodman, ‘‘Interconnect Topologies with Point-To-Point Rings,’’ Technical Report #1058, Computer Sciences Department, University of Wisconsin-Madison, December 1991.

[JoGo91b] Ross E. Johnson and James R. Goodman, ‘‘topology.c,’’ ftp access, Computer Sciences Department, University of Wisconsin-Madison, December 1991. Available at pipe.cs.wisc.edu (128.105.2.11) in directory TRs.

[JoGo92] Ross E. Johnson and James R. Goodman, ‘‘Synthesizing General Topologies from Rings,’’ *Proceedings of the 1992 International Conference on Parallel Processing (ICPP ’92)*, August 1992, (Volume I: Architecture) I-86-95.

[KEWP85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, ‘‘Implementing a Cache Consistency Protocol,’’ *Proceedings of the Twelfth Annual International Symposium on Computer Architecture* **13**, 3 (June 1985), 276-283.

[KeRi88] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, Second Edition 1988.

[KiPr90] D. G. Kirkpatrick and T. Przytycka, ‘‘Parallel Construction of Near Optimal Binary Trees,’’ *Proceedings of the 1990 ACM Symposium on Parallel Algorithms and Architectures (SPAA ’90)*, June 1990, 234-243.

[Knig92] Tom Knight, *personal correspondence*, August 1992. E-mail: tk@ai.mit.edu.

[Knut73] Donald E. Knuth, *The Art of Computer Programming, Volume 3 / Sorting and Searching*, Addison Wesley, Reading, Massachusetts, 1973.

[Kris91] Ernst H. Kristiansen, *preliminary SCI prototype, personal correspondence*, Dolphin Server Technologies, Oslo, Norway, November 1991. E-mail: Ernst.Kristiansen@si.no.

[Krof81] David Kroft, ‘‘Lockup-Free Instruction Fetch/Prefetch Cache Organization,’’ *Proceedings of the Eighth Annual Symposium on Computer Architecture* **9**, 3 (May 1981), 81-87.

[KuPf86] Manoj Kumar and Gregory F. Pfister, ‘‘The Onset of Hot Spot Contention,’’ *Proceedings of the 1986 International Conference on Parallel Processing (ICPP ’86)*, August 1986, 28-34.

[KuLe80] H. T. Kung and Philip L. Lehman, ‘‘Concurrent Manipulation of Binary Search Trees,’’ *ACM Transactions on Database Systems* **5**, 3 (September 1980), 354-382.

[LaMu79] C-Y. Lam and S. Mudnick, ‘‘Properties of Storage Hierarchy Systems with Multiple Page Sizes and Redundant Data,’’ *ACM Transactions on Database Systems* **4**, 3 (September 1979), 345-367.

[Lamp79] Leslie Lamport, ‘‘How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs,’’ *IEEE Transactions on Computers* **28**, 9 (September 1979), 690-691.

[Lawr75] Duncan H. Lawrie, ‘‘Access and Alignment of Data in an Array Processor,’’ *IEEE Transactions on Computers* **24**, 12 (December 1975), 1145-1155.

[LeSB88] Thomas J. LeBlanc, Michael L. Scott, and Christopher M. Brown, ‘‘Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor,’’ *Proceedings ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages and Systems,* July 1988.

[Lebe91] Alvin R. Lebeck, ‘‘Request Combining in Multiprocessors with Arbitrary Interconnection Networks,’’ Master’s Thesis, Computer Sciences Department, University of Wisconsin-Madison, 1991.

[LeSo92] Alvin R. Lebeck and Gurindar S. Sohi, ‘‘Request Combining in Multiprocessors with Arbitrary Interconnection Networks,’’ manuscript in preparation, Computer Sciences Department, University of Wisconsin-Madison, December 1992.

[LeKK86] Gyungho Lee, Clyde P. Kruskal, and David J. Kuck, ‘‘The Effectiveness of Combining in Shared Memory Parallel Computers in the Presence of ‘Hot Spots’,’’ *Proceedings of the 1986 International Conference on Parallel Processing (ICPP ’86)*, August 1986, 35-41.

[LeYa81] Philip L. Lehman and S. Bing Yao, ‘‘Efficient Locking for Concurrent Operations on B-Trees,’’ *ACM Transactions on Database Systems* **6**, 4 (December 1981), 650-670.

[LLGG90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy, ‘‘The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor,’’ *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture* **18**, 2 (May 1990), 148-159.

[LLGW92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam, ‘‘The Stanford Dash Multiprocessor,’’ *IEEE Computer* **25**, 3 (March 1992), 63-79.

[LoTh88] T. Lovett and Shreekant Thakkar, ‘‘The Symmetry Multiprocessor System,’’ *Proceedings of the 1988 International Conference on Parallel Processing (ICPP ’88)*, August 1988, 303-310.

[Manb89] Udi Manber, *Introduction to Algorithms*, Addison Wesley, Reading, Massachusetts, 1989.

[Matl91] Norman Matloff, ‘‘An Argument Against Scalable Cache Coherency,’’ *ACM SIGARCH Computer Architecture News* **19**, 4 (June 1991), 117-123.

[McCr84] E. M. McCreight, ‘‘The Dragon Computer System: An Early Overview,’’ Technical Report, Xerox Corporation, September 1984.

[MoIy85] Abha Moitra and S. Sitharama Iyengar, ‘‘A Maximally Parallel Balancing Algorithm for Obtaining Complete Balanced Binary Trees,’’ *IEEE Transactions on Computers* **34**, 6 (June 1985), 563-565.

[Mosb93] David Mosberger, ‘‘Memory Consistency Models,’’ *ACM SIGOPS Operating Systems Review* **27**, 1 (January 1993), 18-26.

[Netz91] Robert Harry Benson Netzer, *Race Condition Detection for Debugging Shared-Memory Parallel Programs*, PhD Thesis, Computer Sciences Department, University of Wisconsin-Madison, August 1991. Computer Sciences Technical Report #1039.

[NiSt92] Håkan Nilsson and Per Stenström, ‘‘The Scalable Tree Protocol - A Cache Coherence Approach for Large-Scale Multiprocessors,’’ *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing,* December 1992, 498-506.

[NiSt93] Håkan Nilsson and Per Stenström, ‘‘Performance Evaluation of Link-Based Cache Coherence Schemes,’’ *Proceedings of the 26th Hawaii International Conference on System Sciences*, January 1993, 486-495.

[NuAg91] Daniel Nussbaum and Anant Agarwal, ‘‘Scalability of Parallel Machines,’’ *IEEE Computer* **34**, 3 (March 1991), 56-61.

[OKNe90] Brian W. O’Krafka and A. Richard Newton, ‘‘An Empirical Evaluation of Two Memory-Efficient Directory Methods,’’ *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture* **18**, 2 (May 1990), 138-147.

[PaPa84] Mark S. Papamarcos and Janak H. Patel, ‘‘A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories,’’ *Proceedings of the Eleventh Annual International Symposium on Computer Architecture* **12**, 3 (June 1984), 348-354.

[PBGH85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, ‘‘The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture,’’ *Proceedings of the 1985 International Conference on Parallel Processing (ICPP ’85),* August 1985, 764-771.

[PfNo85] G. F. Pfister and V. A. Norton, ‘‘‘Hot-Spot’ Contention and Combining in Multistage Interconnection Networks,’’ *IEEE Transactions on Computers* **34**, 10 (October 1985), 943-948.

[Pugh90] William Pugh, ‘‘Skip Lists: A Probabilistic Alternative to Balanced Trees,’’ *Communications of the Association for Computing Machinery (CACM)* **33**, 6 (June 1990), 668-676.

[PuBr85] P. W. Purdom, Jr. and C. A. Brown, *The Analysis of Algorithms*, CBS Publishing, New York, 1985.

[RHLL92] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood, ‘‘The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers,’’ Technical Report #1122, Computer Sciences Department, University of Wisconsin-Madison, November 1992. To appear in the in Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and

Modeling of Computer Systems, May 1993.

[ReTh86] Randall Rettberg and Robert Thomas, ‘‘Contention Is No Obstacle to Shared-Memory Multiprocessing,’’ *Communications of the Association for Computing Machinery (CACM)* **29**, 12 (December 1986), 1202-1212.

[RuSe84] Larry Rudolph and Zary Segall, ‘‘Dynamic Decentralized Cache Schemes for MIMD Parallel Processors,’’ *Proceedings of the Eleventh Annual International Symposium on Computer Architecture* **12**, 3 (June 1984), 340-347.

[Sagi85] Yehoshua Sagiv, ‘‘Concurrent Operations on B-Trees with Overtaking,’’ *Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, March 1985, 28-37.

[ScDu87] Christoph Scheurich and Michel Dubois, ‘‘Correct Memory Operation of Cache-Based Multiprocessors,’’ *Proceedings of the Fourteenth Annual International Symposium on Computer Architecture*, June 1987, 234-243.

[ScGo91] Steven L. Scott and James R. Goodman, ‘‘Performance of Pipelined K-ary N-cube Networks,’’ Technical Report #1010, Computer Sciences Department, University of Wisconsin-Madison, February 1991. A modified version, ‘‘The Impact of Pipelined Channels on *k*-ary *n*-cube Networks,’’ has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems; contact Steven L. Scott (sls@cray.com) for current status.

[Scot91] Steven L. Scott, ‘‘A Cache Coherence Mechanism for Scalable, Shared-Memory Multiprocessors,’’ *Proceedings of the International Symposium on Shared Memory Multiprocessing*, April 1991, 49-59. Appears Technical Report #1002, Computer Sciences Department, University of Wisconsin-Madison, February 1991.

[ScGV92] Steven L. Scott, James R. Goodman, and Mary K. Vernon, ‘‘Performance of the SCI Ring,’’ *Proceedings of the Nineteenth Annual International Symposium on Computer Architecture* **20**, 2 (May 1992), 403-414.

[Scot92] Steven Lee Scott, *Toward the Design of Large-Scale, Shared-Memory Multiprocessors*, PhD Thesis, Computer Sciences Department, University of Wisconsin-Madison, July 1992. Computer Sciences Technical Report #1100.

[Sedg83] Robert Sedgewick, *Algorithms*, Addison Wesley, Reading, Massachusetts, 1983.

[Sili89] ‘‘Power Series,’’ Technical Report, Silicon Graphics, 1989.

[SiHo91] Richard Simoni and Mark Horowitz, ‘‘Dynamic Pointer Allocation for Scalable Cache Coherence Directories,’’ *Proceedings of the International Symposium on Shared Memory Multiprocessing*, April 1991, 72-81. Appears Technical Report CSL-TR-91-491, Stanford University, Computer Systems Laboratory, August 1991.

[SiWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta, ‘‘SPLASH: Stanford Parallel Applications for Shared-Memory,’’ *ACM SIGARCH Computer Architecture News* **20**, 1 (March 1992), 5-44.

[Smit82] Alan J. Smith, ‘‘Cache Memories,’’ *ACM Computing Surveys* **14**, 3 (September 1982), 473-530.

[SoSG89] Gurindar S. Sohi, James E. Smith, and James R. Goodman, ‘‘Restricted Fetch&Φ Operations for Parallel Processing,’’ *Proceedings of the Third International Conference on Supercomputing*, June 1989, 410-416. Appears Technical Report #922, Computer Sciences Department, University of Wisconsin-Madison, March 1990.

[Stal84] William Stallings, ‘‘Local Networks,’’ *ACM Computing Surveys* **16**, 1 (March 1984), 3-41.

[Ston71] H. S. Stone, ‘‘Parallel Processing with the Perfect Shuffle,’’ *IEEE Transactions on Computers* **20**, 2 (February 1971), 153-161.

[Ston73] Harold S. Stone, ‘‘An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations,’’ *Journal of the Association for Computing Machinery (JACM)* **20**, 1 (January 1973), 27-38.

[Ston84] Harold S. Stone, ‘‘Database Applications of the FETCH-and-ADD Instruction,’’ *IEEE Transactions on Computers* **33**, 7 (July 1984), 604-612.

[SwSm86] Paul Sweazey and Alan Jay Smith, ‘‘A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus,’’ *Proceedings of the Thirteenth Annual International Symposium on Computer Architecture* **14**, 2 (June 1986), 414-423.

[Tane81] Andrew S. Tanenbaum, *Computer Networks*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[Tang76] C. K. Tang, ‘‘Cache System Design in the Tightly Coupled Multiprocessor System,’’ *AFIPS Proceedings of the National Computer Conference*, 1976, 749-753.

[ThSt87] Charles P. Thacker and Lawrence C. Stewart, ‘‘Firefly: a Multiprocessor Workstation,’’ *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)* **15**, 5 (October 1987), 164-172.

[TDLS90] Shreekant Thakkar, Michel Dubois, Anthony T. Laundrie, and Gurindar S. Sohi, ‘‘Scalable Shared-Memory Multiprocessor Architectures,’’ *IEEE Computer* **23**, 6 (June 1990), 71-74.

[ThDe90a] Manu Thapar and Bruce Delagi, ‘‘Stanford Distributed-Directory Protocol,’’ *IEEE Computer* **23**, 6 (June 1990), 78-80.

[ThDe90b] Manu Thapar and Bruce Delagi, ‘‘Cache Coherence for Large Scale Shared Memory Multiprocessors,’’ *Proceedings of the 1990 ACM Symposium on Parallel Algorithms and Architectures (SPAA ’90)*, July 1990, 155-160.

[VeJS89] Mary K. Vernon, Rajeev Jōg, and Gurindar S. Sohi, ‘‘Performance Analysis of Hierarchical Cache-Consistent Multiprocessors,’’ *Performance Evaluation* **9** (1989), 287-302, Elsevier Science Publishers B. V. (North-Holland).

[Wils87] Andrew W. Wilson, Jr., ‘‘Hierarchical Cache / Bus Architecture for Shared Memory Multiprocessors,’’ *Proceedings of the Fourteenth Annual International Symposium on Computer Architecture*, June 1987, 244-252.

[Wint87] Pawel Winter, ‘‘Steiner Problem in Networks: A Survey,’’ *Networks* **17**, 2 (1987), 129-167.

[Witt76] Larry D. Wittie, ‘‘Efficient Message Routing in Mega-Micro-Computer Networks,’’ *Proceedings of the Third Annual Symposium on Computer Architecture* **4**, 4 (January 1976), 136-140.

[YaTB92] Qing Yang, George Thangadurai, and Laxmi N. Bhuyan, ‘‘Design of an Adaptive Cache Coherence Protocol for Large Scale Multiprocessors,’’ *IEEE Transactions on Parallel and Distributed Systems* **3**, 3 (May 1992), 281-293.

[YeTL87] P-C. Yew, N-F. Tzeng, and D. H. Lawrie, ‘‘Distributing Hot-Spot Addressing in Large-Scale Multiprocessors,’’ *IEEE Transactions on Computers* **36**, 4 (April 1987), 388-395.

[YeTa90] Pen-Chung Yew and Peiyi Tang, ‘‘Software Combining Algorithms for Distributing Hot-Spot Addressing,’’ *Journal of Parallel and Distributed Computing*, October 1990.

# Citation Index