Toward the Design of Large-Scale, Shared-Memory Multiprocessors

Steven Lee Scott

Technical Report #1100

July 1992

TOWARD THE DESIGN OF LARGE-SCALE, SHARED-MEMORY MULTIPROCESSORS

By

STEVEN LEE SCOTT

A thesis submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1992

Abstract

The state-of-the-art in multiprocessing today employs thousands of high-performance microprocessors. As system sizes continue to grow, increasing care must be taken to design cost-efficient, balanced (*i.e. scalable*) systems. This thesis addresses the scalability of shared-memory multiprocessors, presenting a practical treatment of scalability, and proceeding to focus on aspects of two critical areas of large-scale system design: interconnection networks and cache coherence mechanisms. In these areas, *pipelined-channel* interconnection networks and *pruning-cache directories* are investigated, respectively.

Pipelined-channel interconnection networks allow multiple bits to be simultaneously in flight on a single wire, decoupling channel throughput from channel latency. The first published performance analysis of the SCI ring, a new IEEE standard employing pipelined channels, is presented. This study serves as a proof-of-concept for pipelined-channel networks, demonstrating their very high performance potential. This analysis is followed by a performance study of large, multi-dimensional, pipelined-channel networks under various physical constraints. Design tradeoffs for multi-dimensional networks have been previously analyzed for *non*-pipelined-channel networks, leading to the conclusion that low-dimensional networks are superior. It is shown that the design tradeoffs are significantly changed when pipelined channels are used. As a result, the optimal dimensionality of a pipelined-channel network is higher than that of a non-pipelined-channel network. In addition, the radix (number of nodes per dimension) of a pipelined-channel network should be kept roughly constant as network size is increased. In non-pipelined networks, the optimal radix increases significantly as network size increases, thereby reducing the traffic capacity of the network.

Pruning-cache directories cache sharing information to allow multicasting of invalidations in hierarchical and cube-based, shared-memory multiprocessors. The performance of pruning-cache directories is investigated through both analysis and simulation, and shown to scale well to very large system sizes, while requiring only a modest amount of storage overhead. Pruning caches are shown to perform better than a similar scheme employing the multi-level inclusion property, while providing better performance in the face of contention than other directory-based coherence protocols. Several issues relating to pruning cache implementation and management are investigated.

Acknowledgements

I've been fortunate to have the help and support of many people while pursuing my PhD, and I now have the pleasurable task of giving them my sincere thanks.

My wife Lori and my two daughters, Courtney and Lauren, have stuck with me through thick and thin. They put up with my absences, they cheered me on when I needed it, and they gave life outside the Computer Sciences department fullness and meaning. This thesis is dedicated most of all to them. Thanks also go to my parents, Dorothy and Peter, for all they have given me over the years, and to my brother Michael, for his sage advice.

I have received much help from members of the Computer Sciences department here at Wisconsin. First and foremost, my advisor Jim Goodman has played a key role in my professional development. His ideas and our many long conversations have laid the foundation for my work and guided me through my years of graduate school. Special thanks also goes to Guri Sohi for all the time he has taken to give help and advice.

Over the years, I have depended on Mark Allmen and Ross Johnson to help me solve problems and discuss ideas. In addition, Steve Dirkse, Phil Woest, Rick Kessler, Sarita Adve, Rich Maclin, Jon Cargille, Jeff Hollingsworth, Mary Vernon, Mark Hill and others not listed here have all given me technical assistance or advice on many occasions. Thanks to all of them.

Finally, I would not have been able to attend graduate school at all if it were not for the generous financial assistance of the Fannie and John Hertz Foundation, the Wisconsin Alumni Research Foundation, and NSF grant #CCR-892766.

Contents

Chapter 1

Introduction

Large-scale multiprocessors offer the possibility of enormous computing power. Furthermore, by riding the crest of microprocessor technology, this power should come at a low cost-performance ratio. Unfortunately, the simple extension of existing system designs to large numbers of processors may not be feasible or may lead to significant per-processor performance degradation.

Intuitively, a design must be *scalable* in order to be efficient with large numbers of processors. The exact meaning of "scalability", however, is far from clear; despite its wide use, the term has no accepted definition. A serviceable first definition for scalable is simply "efficient for very large systems". This implies that the performance of the various system components must be well balanced and bottlenecks must be avoided. System cost and complexity must be manageable as well, avoiding, for example, physical resources that must grow as the square of the system size (number of processors).

This thesis addresses the scalability of *shared-memory* multiprocessors. It is expected that in any large multiprocessor the memory is physically distributed among the nodes of the machine. "Shared memory," therefore, means simply that the address space is shared, presenting a uniform naming mechanism for all of memory. While parts of the thesis are specific to shared-memory machines, other parts are applicable to *distributed-memory* (or *message passing*) machines as well.

The primary distinction between shared- and distributed-memory machines is the abstraction presented to the programmer. A shared-memory machine presents the programmer with the abstraction of a single shared memory space, which is then used for communicating data values and often for synchronization. In a distributed-memory machine, the programmer must specify interprocessor communication explicitly, through message passing or memory transfers. Shared-memory multiprocessors may be further divided into those providing cache coherence in hardware, and those requiring software support (or disallowing caching).

It is widely believed that the shared-memory model is a more natural or convenient abstraction for the programmer. Moreover, the software overhead for communicating through shared

memory is typically much smaller than that for message passing, so that distributed-memory algorithms are generally restricted to coarser grain sizes. Regardless of the technical merits of shared memory, however, the model demands continued focus due to its wide acceptance and success in the marketplace. There is a perception, however, that shared memory machines — and in particular those with hardware-maintained cache coherence — are difficult to scale to very large sizes. This perception is not without some validity, and thus begets a challenge to the designers of shared-memory machines.

There are many problems that must be overcome in order to build efficient, large-scale multiprocessors of any sort. Chief among these is the design of an interconnection structure that provides sufficient throughput for interprocessor communication and memory requests, while delivering acceptable communication latency. For systems guaranteeing cache coherence in hardware, a coherence mechanism must be devised that provides an acceptable balance between cost and performance. Hardware coherence mechanisms in use today are not well suited (for a variety of reasons) for systems built with thousands of processors. The processors themselves must be designed with latency toleration in mind, and with appropriate features to facilitate interrupt handling, memory management, synchronization operations and a host of other requirements related to multiprocessing. Synchronization mechanisms must be developed that are capable of efficiently coordinating large numbers of processors. Renewed attention must be paid to fault tolerance, which becomes increasingly difficult to provide as system size increases. Packaging, cooling, testing and other implementation aspects are very difficult challenges for large systems. Progress is needed on the software front as well. Many operating system issues, such as memory management and process scheduling, become more difficult as system sizes increase. Lastly — and this is by no means a comprehensive list — continued progress in parallel algorithms, programming models, programming tools and compilers is essential to the future of parallel computing in general.

The overall topic of large-scale, shared-memory multiprocessor design is obviously extremely broad. While the end goal of this thesis is the design of efficient and powerful large-scale multiprocessors, the contributions contained here, as the first word of the title suggests, are only one step toward this goal. The thesis deals primarily with the issues of interconnection networks and hardware cache coherence mechanisms. These are two of the most direct challenges to multiprocessor hardware designers, and are critical to large system performance. In both of these areas, a promising design alternative is investigated in depth.

I begin by discussing the concept of scalability. While it is a mistake to value scalability over performance, and while a precise (but useful) definition for scalability may not exist, I believe that the concept of scalability is indeed useful, and can lend valuable insight into the behavior of design alternatives for large-scale implementations. Chapter 2 discusses this topic and presents a working definition for scalability, based upon bandwidth, latency and cost considerations. While not providing a simple predicate (*is* system X scalable?) the definition presented in Chapter 2 provides a useful framework for discussing and evaluating any number of issues related to multiprocessor design. This framework is a valuable contribution in its own right.

Scalability arguments are then applied to various topologies and cache coherence mechanisms. Based upon simplified assumptions (ignoring transmission delay and wiring constraints), the k-ary n-cube is identified as a promising interconnection topology. Consideration of transmission delay points out limitations to the scalability of conventional networks. *Pipelined-channel* interconnection networks are proposed to address this problem. In a pipelined-channel network, the network cycle time is determined by the speed of the switching circuitry, independent of wire lengths. Thus, multiple bits may be simultaneously in flight on a single wire.

Chapters 3 and 4 investigate pipelined-channel interconnection networks in more detail. Chapter 3 presents a performance study of the Scalable Coherent Interface (SCI), a new IEEE standard that uses pipelined channels [IEEE92]. Modeling and simulation are used to analyze the performance of a single SCI ring, the basic building block of all SCI systems. The study serves two primary purposes. It is important in its own right, as the first comprehensive analysis of a new IEEE standard. It provides a useful summary of the ring's operation, a unique analytical model, and performance results that should be of interest to those in the SCI community and those interested in high performance multiprocessor interconnects. In addition, for the purposes of this thesis, it serves as a "proof of concept" of pipelined channels, establishing the high performance of an actual, implementable design.

Chapter 4 extends consideration of pipelined channels to large, multi-dimensional networks. It is shown that not only do pipelined channels provide performance superior to non-pipelined channels for large networks, but that they fundamentally alter the network design tradeoffs. Previous studies of non-pipelined-channel networks [Dall90, Agar91] have shown that low-dimensional networks provide the best performance. By changing the effects of wire length on network performance, pipelined channels argue for higher dimensionality. Moreover, the optimal manner in which networks are grown is changed in such a way as to better preserve the per-processor bandwidth provided by smaller networks and more closely match the intuition

developed in Chapter 2 when wire delay was ignored. These results should have a significant effect on the way in which future large-scale networks are designed.

The focus of Chapters 5 and 6 switches to cache coherence mechanisms. Chapter 5 presents a summary of existing and proposed cache coherence mechanisms and describes a novel approach for maintaining coherence in hierarchical or cube-based multiprocessors. Approximate directory information is held in special-purpose *pruning caches* [Good89, Scot91], and used to limit the propagation of broadcast invalidations to those parts of the hierarchy that need to receive them. This scheme approximates the performance of a full hierarchical directory, but at considerably less cost. Pruning-cache performance is analyzed for cache line invalidations under various distributions of shared lines. Scalability arguments are used to show that the size of pruning caches must grow only as the log of the system size. In addition, because of their hierarchical structure, pruning caches are compatible with hierarchical read combining. A mechanism for read combining in hierarchical or k-ary n-cube networks is described, and the performance analyzed under various sharing assumptions.

Chapter 6 delves deeper into the performance of pruning caches. Simulation is used to investigate performance of pruning-cache systems under a variety of workloads and system sizes. In addition, pruning caches are compared against a similar coherence mechanism based on the *multi-level inclusion* (MLI) property that has been previously proposed for hierarchical systems. Pruning caches are shown to provide superior performance at lower cost. Several pruning cache management issues are also explored in Chapter 6. Various alternatives are simulated, and optimizations to the basic protocol are proposed and simulated.

Finally, Chapter 7 presents concluding remarks and discusses possible avenues for further research.

Chapter 2

Scalability

Scalability is not an end goal, in itself. For a given implementation, cost and performance are the primary concerns. However, as system size increases, a scalable design will eventually provide higher performance than a non-scalable design. For example, a ring interconnect may provide higher performance than a 3-dimensional mesh for small system sizes, due to faster switching times. However, the traffic over a link in the ring (assuming uniform communication) is proportional to the system size, while the traffic over a link in the mesh is proportional to the cube root of the system size. A 3-dimensional mesh, therefore, will perform better than a ring for sufficiently large systems. Even a 3-dimensional mesh will saturate eventually, however. I am interested in designs that can be efficiently implemented with thousands, or tens of thousands, of processors. How a design scales can lend valuable intuition as to its behavior for such large system sizes.

As was mentioned in Chapter 1, this thesis focuses on shared-memory multiprocessors with hardware-maintained cache coherence. The bottlenecks in such systems can be due to software, hardware (*i.e.* the topology) or the communication protocol (*i.e.* the cache coherence mechanism). Any one of these factors may inhibit scalability. However, the three components cannot always be considered in isolation, as their interaction can significantly affect scalability. For example, certain workloads may scale on one machine while not scaling on another. Therefore, we typically cannot make statements about the scalability of a given component or aspect of a system, without making assumptions about the remaining components of the system. This point is addressed with regard to workload assumptions in particular later in the chapter.

Section 1 of this chapter presents some background on the topic of scalability. A working definition of scalability is then presented in Section 2. Section 3 briefly discusses the effect of software on scalability. Finally, sections 4 and 5 apply scalability arguments to network topologies and cache coherence mechanisms, respectively.

1. Background

The issue of scalability has surfaced many times since the introduction of parallel computers. Amdahl [Amda67] originally observed that the efficiency of a parallel computer will

decrease as more processors are used to solve a fixed-size problem. This is due to the (reasonable) assumption of a fixed serial portion of the algorithm. As the number of processors executing the parallel portion of the algorithm increases, the relative time spent performing the serial computation increases.

The relationship can be roughly characterized as follows. Let f be the fraction of inherently serial work in a program. Assume that the remaining fraction $(1-f)$ is perfectly parallelizable and ignore any communication overhead. If T_1 is the running time of the program on a single processor, then T_p, the running time of the program using p processors, is given by

$$T_p = fT_1 + \frac{(1-f)T_1}{p} \tag{2.1}$$

It is clear that regardless of the number of processors used, the absolute maximum speedup of this program, $\frac{T_1}{T_p}$, will be bounded by $\frac{1}{f}$. This eventually dooms attempts to speed up the execution of the program by using more processors. Flatt and Kennedy [Flat89], in fact, showed that for real programs a maximum speedup is obtained using some number of processors, beyond which additional processors can only increase the running time.

Gustafson [Gust88] suggested that instead of increasing the number of processors used to solve a fixed-size problem, we should measure speedup by holding execution time constant, and growing the parallel portion of the problem with the number of processors. This reflects the nature of at least some actual parallel codes [Benn88] and would appear to be a necessary condition in order to maintain high processor efficiency for very large systems. This model fits nicely with the idea that the amount of time people are willing to wait for a computer program to complete has remained roughly constant through the years; rather than solve problems faster, we tend to solve bigger problems in the same amount of time.

Given this primary assumption of scalable software, we need not quit before we have begun, and can continue to define a scalable system. There have been several suggestions as to what scalability means. Patton [Patt85] stated that a scalable design "can be adjusted up or down in size without loss of functionality to scale effects". Goodman, Hill and Woest [Good89] define a scalable *algorithm* as one whose serial portion does not grow with problem size and whose parallel portion contains parallelism at least proportional to the algorithm's complexity. This strict definition requires a scalable algorithm to be solvable in a constant amount of time, regardless of problem size, on an idealized multiprocessor. They define a scalable *system* according to the speed at which it executes a scalable algorithm, allowing limited reductions in speed due to communication latency.

Nussbaum and Agarwal [Nuss91] provide a more general definition of scalability, that assigns a scalability value to any pair of machine and algorithm. Their metric is the asymptotic speedup of a given, fixed-size algorithm on a real machine with unlimited processors relative to the speedup of the same algorithm on an EREW PRAM [Fort78]. Their definition is precise, but rather theoretical in nature, dealing only with asymptotic behavior and ignoring any cost or efficiency considerations.

Johnson [John90] presents a rigorous set of definitions for scalable hardware which is independent of the workload. Using asymptotic behavior, he defines 11 classes of scalability, including *uniformly, architecturally* and *implementationally scalable.*

On the other hand, Hill [Hill90], questions whether scalability can be usefully defined at all. He challenges the technical community to either define the term rigorously, or stop using it altogether. Others [Leno90, Cher89, Hage89] use the term without accompanying definition, relying on the readers' intuitive definitions.

I believe that a rigorous definition of scalability may be of little use, but that we *can* arrive at a useful working definition. Several qualifications need to be offered at the start, however. First, scalability is not a simple binary property; certain designs may be more or less scalable than others. However that does not in practice preclude some designs from being clearly not scalable (in the same manner, we can say that a package is very heavy or that a person is not tall, though weight and height are not binary properties). Second, scalability, in and of itself, says nothing about the cost or performance of a given, fixed-size system. It is useful primarily for providing insight into the behavior of a system as the number of processors grows. Third, even though I am interested in the performance of very large systems, asymptotic behavior of a system may not be of interest. The state of the art is pushing thousands of processors. The behavior of systems with billions of processors is simply not relevant for the foreseeable future.

2. A Working Definition of Scalability

I define scalability in terms of three metrics: cost, latency, and bandwidth. Each of these can be approximated as growing by some order of N, the number of processors in the system. A system can be considered scalable if it scales well with regard to each of these metrics.

Cost: The cost of the system is measured in terms of the required hardware. A full crossbar interconnect, for instance, requires $O(N^2)$ switching elements. An Omega network requires $O(N \log N)$ switching elements. A ring requires only $O(N)$ switching elements. Another effect on the cost metric is the size of memory needed to store directory information. If a directory scheme requires that every line of memory be accompanied by a tag that is proportional to N in

size, then the tag memory must grow as $O(N^2)$ (assuming that the size of main memory grows linearly with N). Some costs should be ignored, as they are practically constant for all but truly asymptotic behavior. For instance, the number of bits used to specify a processor grows as $O(logN)$. But if a single word (32 bits) is used, this effect is not seen until the number of processors exceeds 2^{32}. This sort of behavior can be ignored for scalability considerations. For a system to be considered scalable at all, the cost should be less than $O(N^2)$. By this measure, a full crossbar is not considered scalable. Beyond that, a system may scale better or worse than another regarding cost. For example, a 2-dimensional mesh scales better in cost ($O(N)$) than an Omega network ($O(N \log N)$).

Latency: The latency metric is the *average* latency of a memory request. This is affected by the topology of the system, and can also be affected by the communication protocol and the workload. Ideally, we would like the average latency to remain $O(1)$ as system size increases. Realistically, this is not possible for more than incremental growth of the system. Several interconnection networks (such as the Omega network) provide $O(logN)$ latency upon first approximation. Eventually, however, wire lengths grow, and latency must increase at least as $O(N^{1/3})$, due to propagation of signals in 3-dimensional space. A three dimensional cube topology, which can be implemented with constant length wires, provides latency of $O(N^{1/3})$ for uniform communication.

I argue, however, that asymptotically, the latency of any system must increase as at least $O(N^{1/2})$. Consider an idealized sphere of processors with radius r. The sphere contains $O(r^3)$ processors. The communication distance between two random processors is $O(r)$. The bisection area of the sphere is $O(r^2)$. The traffic across the bisection (assuming uniform communication) is proportional to the number of processors, or $O(r^3)$. Thus the traffic density across the bisection increases as r. This cannot be supported asymptotically. If the traffic density is kept constant, then the sphere may only be packed with $O(r^2)$ processors. The communication latency is then $O(N^{1/2})$. At what size this asymptotic behavior becomes relevant is not clear, and will depend upon the technology being used. However, as an example of this principle applied, the Tera Computer System populates a 3-dimensional mesh of size r^3 with only r^2 processors [Alve90].

Considering this, the tightest absolute requirement that we can impose is that latency cannot grow faster than $O(N^{1/2})$. Even for very large systems, however, we may be able to ignore the asymptotic latency of a design, so certain topologies may scale better than others ($O(N^{1/3})$ versus $O(N^{1/2})$, for instance). Recall, also, that for a given size (even a very large one) a specific system may perform better than another, even if the other has better scaling properties.

Bandwidth: The bandwidth metric is the maximum of the average traffic over each wire resulting from every processor issuing a single memory request. Note that the unit here is not traffic per second, but rather traffic per request, which allows us to consider this metric independently of the latency metric. The traffic is caused by the requests themselves, any associated data, and any traffic generated by the cache coherence mechanism (to retrieve data from another processor or to invalidate lines after a write, for instance).

If the bandwidth metric is greater than $O(1)$, then the load across some link will increase as the system size increases. Unless the original system was designed with bandwidth to spare, the rate at which processors can generate requests will decrease accordingly. For limited scalability, a link may be able to handle $O(logN)$ or $O(N^{1/3})$ traffic, but eventually this will saturate the available bandwidth. Ideally, the bandwidth metric should be $O(1)$.

3. The Effect of Software on Scalability

The three scalability metrics are affected by a combination of hardware, software and the communication protocol.[1] The default assumption regarding software is the *uniform workload* model. The processors' non-local references are uniformly distributed across the memory modules of the system. In addition, the percentage of accesses that are writes to shared variables remains fixed as system size increases. This means that the number of invalidations per request per processor remains roughly constant. If the scalability of a topology is discussed without mention of the software, it is the uniform workload model that is being assumed.

Under a *hot-spot workload*, processors make a higher than average fraction of their requests to a particular memory location or module. This will prevent a system from scaling unless the hardware or communication protocol takes explicit steps to handle it. Concurrent *read* requests are expected to occur frequently. Shared data or instructions read after exiting a barrier are prime candidates. Concurrent write requests are not expected to occur often.

In contrast to the hot-spot workload, a *conspirator workload* can make a multiprocessor scalable that otherwise would not be scalable. For example, if the workload has high geographic locality of communication that matches the physical layout of the system, and the communication protocol allows for localized communication, then the average message will traverse only a

[1] For the purposes of this thesis, "communication protocol" refers to the cache coherence protocol, but other protocols can also have an impact. A routing algorithm, for example, could adversely affect scalability by making unequal use of network links.

fraction of the distance across the system, and an otherwise non-scalable system may scale. Another example of a conspirator workload is one whose fraction of writes to shared variables decreases with system size. This makes the rate of invalidations per request per processor decrease as system size grows larger. This sort of workload occurs, for example, when shared variables are read by all processors between successive writes.

4. Scalable Networks

The framework of Section 2 can now be used to discuss candidate multiprocessor topologies. This is done in Section 4.1, under some simplifying assumptions, which are then addressed in Section 4.2. This section does not attempt to present a complete survey of interconnection networks.

4.1. Preliminary analysis

The primary simplifying assumption of this section is to ignore wire transmission delay. Therefore, latency is related only to the number of "hops" that a message traverses, and not to the length of the wires. The number of wires in the network is assumed only to be constrained by the cost requirement for scalable systems.

A ring interconnect will not scale for uniform workloads because both the rate of traffic across each link and the average communication latency increase as $O(N)$. Extreme conspirator workloads, such as one in which nodes communicate only with nearest neighbors in one dimension, *can* scale on a ring. A single shared bus will not scale, because the traffic on the bus increases as $O(N)$. In addition, though it may not be immediately apparent, the latency of shared bus accesses increases with N (even without the increased queueing delay). This is because the physical length of the bus, as well as its capacitance, increases with N [Wins88].

This suggests an important point: a ring of point-to-point links can be used to simulate a bus. Both have the same scaling properties, but point-to-point links can be clocked at a higher rate (independent of N), leading to higher performance. A broadcast operation is simulated by passing a message completely around the ring. This does not provide global event ordering, however, so modifications are necessary for snooping protocols that depend on this.

One topology that has been widely studied for large numbers of processors [Wils87, Baer88, Wins88, Vern89, Yang92] is the bus hierarchy (see Figure 2.1). This topology will not scale, however, for uniform workloads. Each time a line is written, if a copy exists in any of the other primary subtrees of the hierarchy, an invalidate or update message will have to travel over the root bus. Even if a software coherence mechanism that did not use invalidates

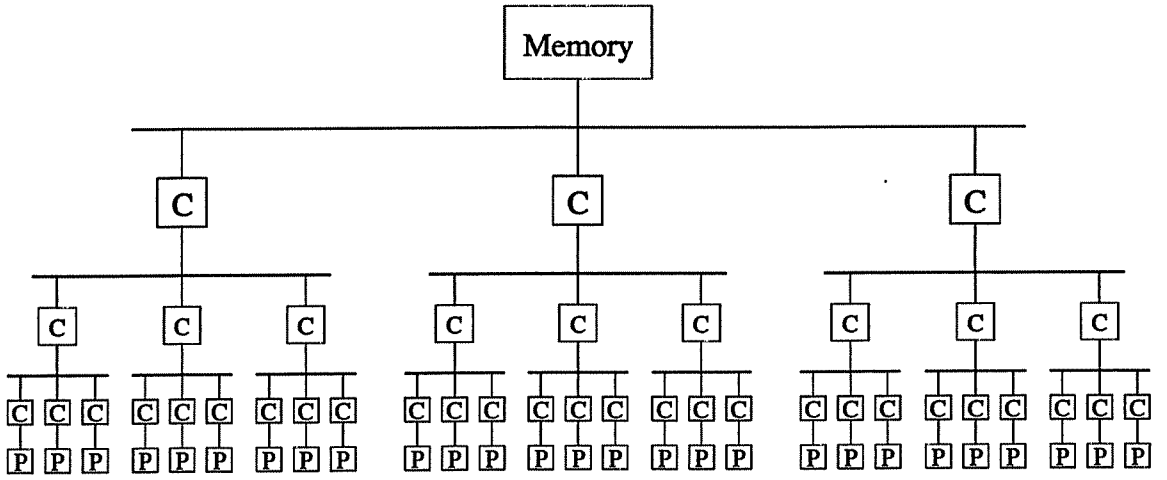


Figure 2.1: A Hierarchical bus-based multiprocessor

were being used, a write to a line that is subsequently read in another subtree requires traffic over the root bus. Thus the traffic over this bus is $O(N)$, and the system does not scale. For a conspirator workload, where the fraction of writes to shared variables decreases with N, the traffic over the root bus can remain $O(1)$. Root traffic can also remain constant with respect to N if *strict* communication locality can be maintained.

Vernon, Jog and Sohi [Vern89], in an analysis of hierarchical bus-based multiprocessors, concluded that the systems scale well only if the average fraction of processors that read a shared line between writes, f_{RI}, remains large as system size increases. This can only be accomplished with a conspirator workload in which the fraction of references that are shared writes decreases with system size. If the fraction of references that are writes to shared variables, f_{WS}, remains constant, then the average number of processors that read a shared line between writes must be $\leq \frac{1-f_{WS}}{f_{WS}}$. Thus, $f_{RI} \leq \frac{1-f_{WS}}{Nf_{WS}}$, which decreases with system size. Their analysis showed that when f_{RI} is small, the system is extremely inefficient for large numbers of processors (due to contention for the root bus).

Another widely studied class of network topologies is multistage interconnection networks, such as the omega network [Lawr75]. The omega network uses $\log_k N$ stages of $k \times k$ crossbar switches to connect N inputs to N outputs. Typically, either the processors and memory modules are at opposite ends of the network (the "dance hall" configuration), or the memory is packaged along with the processors and the network wraps around such that the processor modules are connected to both the network inputs and outputs. Figure 2.2 shows an 8×8 omega network using

2×2 switches and the dance hall configuration.

The cost of an omega network is $O(N\log_k N)$, the communication latency (ignoring wire delay) is $O(log_k N)$, and the bandwidth metric is $O(1)$. The omega network, therefore, scales better than the bus hierarchy. It does not, however, allow locality to be exploited; all memory modules and processors are equi-distant from each other.

Another widely advocated topology is the k-ary n-cube [Sull77]. Rings, 2-dimensional tori, 3-dimensional tori and hypercubes are all sub-classes of this topology. The k-ary n-cube has n dimensions, with k processors in each dimension, for a total of $N = k^n$ processors. For example, a 2-dimensional torus with 64 processors is a 8-ary 2-cube. While "k-ary n-cube" generally refers to a network of point-to-point links, a similar network can be implemented using buses, with k processors per bus and n buses per processor. Goodman and Woest [Good88] refer to such a network as a *multicube*. Figure 2.3 shows a 4-ary 3-cube multiprocessor, implemented with unidirectional links.

The k-ary n-cube has the desirable property that, while it avoids the bottleneck of a single root, it retains a hierarchical structure with respect to any individual root node (see Figure 2.3(b)). Therefore, communications protocols that rely on a hierarchical structure such as that in Figure 2.1 can be implemented on the k-ary n-cube. Pruning-cache directories, which are discussed in Chapters 5 and 6, make use of this hierarchical structure.

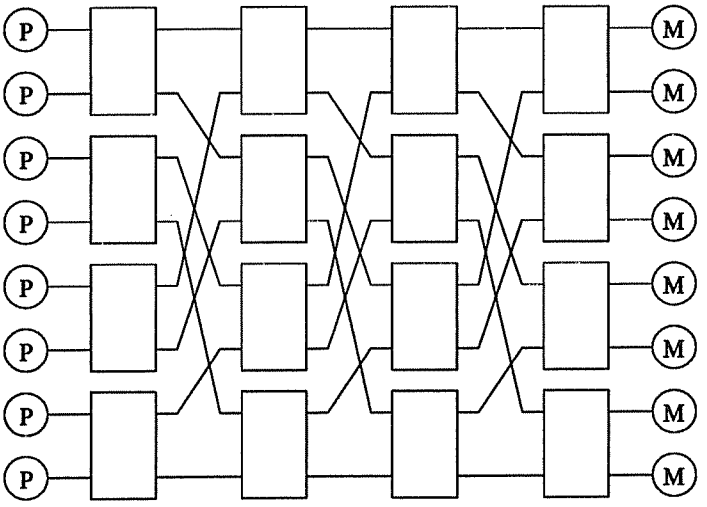
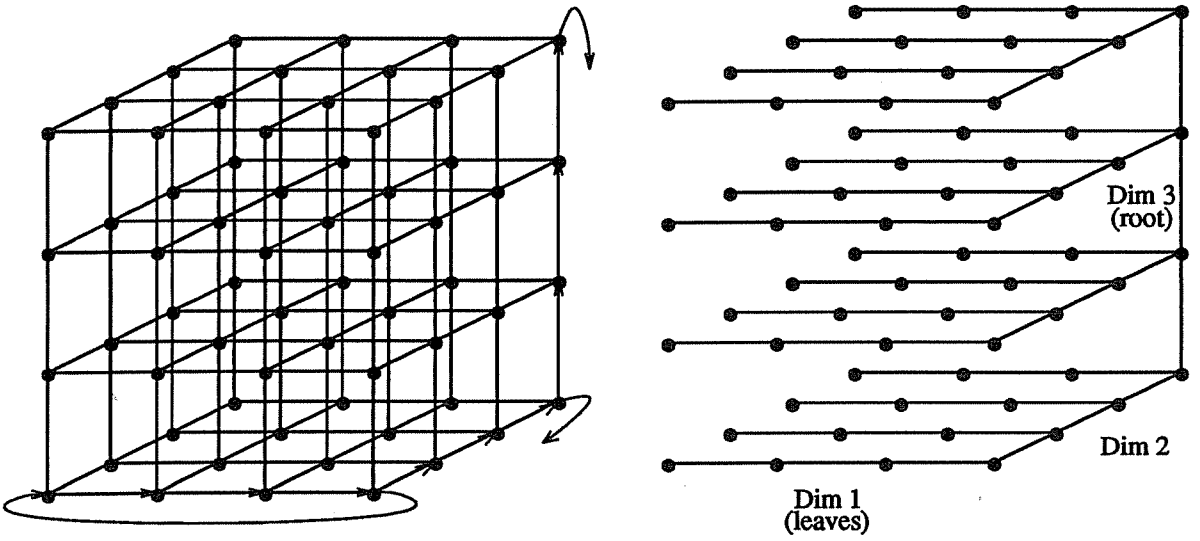


Figure 2.2: An 8×8 omega network



(a) The complete 3-cube (b) One of 16 distinct trees

Figure 2.3: A 4-ary 3-cube multiprocessor (N=64, k=4, n=3)

Each processor is connected to n rings (with an in and out link for each ring), and each ring consists of k processors (if implemented with buses, each processor would connect to n buses, and each bus would connect to k processors). Although not explicitly shown, all rings include end-around connections as illustrated in part (a). Each processor is accompanied by a portion of main memory and one or more levels of cache. Memory is interleaved by cache lines amongst the memory modules. For any given memory module, a tree of rings (or buses) is formed as shown in part (b). This tree is equivalent to a conventional tree hierarchy, given that each parent node (those in the rightmost plane of part (b)) is allowed also to be one of its children. This allows communication protocols based on a hierarchical topology to be implemented.

The k-ary n-cube is scalable, if grown in the proper way. The cost is $O(Nn)$, where $n = \log_k N$. There are N processors and Nn links ($\frac{Nn}{k}$ buses). The average distance between two processors is $O(nk)$ link hops ($O(n)$ bus hops). Thus, assuming uniform traffic, the latency of a communication request is $O(nk)$, and the traffic over a link or bus is $O(k)$. The cost and latency metrics do not preclude scaling, by the definition of Section 2, regardless of how the system is grown. The cost metric favors low dimensional networks, while the latency metric favors high dimensional networks. Bandwidth requirements, however, dictate that the network should be grown by increasing the dimensionality (n). A fixed dimension network (a 2-dimensional torus for instance), does not scale well, due to increasing link traffic. This of course does not exclude limited scalability when n is fixed, especially when n is large.

4.2. Realistic analysis

When wire delay is considered, the scalability of k-ary n-cubes is potentially limited due to physical penalties for high-dimensional networks. When the number of logical dimensions exceeds the number of physical dimensions in which the network is implemented (which, alas, is fundamentally limited to three), then the processors can no longer be connected solely to physically close neighbors, and the network wire lengths must grow [Dall90]. This means that the hop latency is no longer $O(1)$ for high-dimensional networks.

A similar phenomenon occurs for omega networks, where even small- to medium-scale networks have long wires. Although the length of the longest wire is $O(N)$ if the network is laid out as shown in Figure 2.2, a good three dimensional layout can result in maximum wire lengths of $O(N^{1/2})$ [Gott83]. The remainder of this thesis will consider only k-ary n-cube networks, although much of the analysis would be germane to multistage networks as well. This choice is partially because k-ary n-cubes provide a very flexible and general interconnect to study (allowing many configurations, from rings to hypercubes), and partially because they provide an excellent platform for the cache coherence scheme investigated in Chapters 5 and 6.

More important than the actual transmission latency to traverse long wires is the effect that this latency has on the network cycle time. In a conventional, non-pipelined-channel network, the network cycle time must be grown to accommodate the transmission delay across the longest wire in the network. This larger cycle time adds to the switching latency and to the latency of traversing *all* links, regardless of their length. It also effectively decreases the per-processor throughput of the network, by decreasing the rate at which new data can be clocked onto the links. These effects exact a large penalty on high-dimensional networks, which have inherently longer wires than do low-dimensional networks.

This thesis proposes the use of pipelined-channel networks to overcome this problem. As mentioned in Chapter 1, the cycle time in a pipelined-channel network is independent of wire length, allowing multiple bits to be simultaneously in flight on longer wires. Chapter 4 demonstrates that a pipelined-channel network is optimally grown by holding the radix (k) constant and increasing the dimensionality (n). This allows networks to scale well, according to the definition of Section 2.

In addition to wire length, high-dimensional networks can also be penalized by various wiring constraints [Dall90, Agar91]. The *constant node size* constraint, motivated by board- and chip-level pin limitations, holds the number of wires per node constant as the dimensionality of a fixed sized network is varied. If applied to scalability (*i.e.* holding the number of wires per node

constant as the system size is varied), it would cause the link traffic to increase as $O(n)$ and asymptotically place an upper bound on the network dimensionality, causing link traffic to grow as some root of N. This constraint is equivalent to tightening the restriction on system cost, allowing the cost (number of wires in this case) to increase only linearly with system size.

The *constant bisection constraint*, motivated by wiring density limitations, holds the number of wires crossing the bisection of the network constant as the dimensionality is varied. This constraint more severely limits scalability, causing the link traffic to increase as $O(N^{1/3})$. It is essentially equivalent to the argument used in Section 2 regarding asymptotic scalability. A network that is truly bisection constrained is already in the region where bandwidth cannot scale with the number of processors. The only remedy for this situation is to either populate the network with fewer processors, or rely on communication locality (a conspirator workload) to limit bandwidth requirements.

These issues are more thoroughly explored in Chapter 4. It is shown that even for wire-constrained networks, optimal growth of k-ary n-cube networks holds the radix constant. This is a significant departure from non-pipelined-channel networks, in which the optimal radix increases significantly for large networks.

5. Scalable Cache Coherence

The basic function of a cache coherence mechanism is to make sure that after a memory location is modified, subsequent reads to that location return the new (modified) value. Unfortunately, in a multiprocessor, it is difficult to specify the exact time of a write; because of non-deterministic delays to access memory, requests to different locations and from different processors may complete in a different order than that in which they were issued. In addition, as system sizes increase, so does the interval of time between when an event occurs, and when that event can be observed in other parts of the system. The cache coherence mechanism need only make the following two guarantees. First, writes to a given memory location by a given processor, cannot be observed by any processor to occur out of program order. Second, there must exist some global ordering of reads and writes to a given memory location, such that no processor observes any other order.

In addition to cache coherence, a multiprocessor may provide *sequential consistency* [Lamp78], or some weaker form of memory consistency [Dubo86, Adve90, Ghar90], which makes a guarantee about the global ordering of reads and writes to *different* memory locations . Providing some form of memory consistency can significantly impact the cache coherence mechanism. In order to provide ordering of memory operations, the system must be able to

determine when all processors have seen the new value of a write (or at least can no longer see the old value). This thesis does not address the issue of sequential consistency other than by placing this constraint (the ability to determine when coherence actions have completed) upon the cache coherence mechanism.

It is assumed that coherence is maintained over cache lines, and that each line of memory has an associated state (such as *modified, shared* or *private*) in each processor cache where it resides and possibly in main memory as well. When a line is written, all shared copies of the line must be either updated or invalidated, using a *write update* or *write invalidate* protocol respectively. If all or most processors that have a copy of a line when it is written will be re-reading the line, and if multiple writes to the same line by a single processor can be delayed such that only a single update message is sent, then a write update protocol should outperform a write invalidate protocol. However, when one of the above conditions does not hold, a write update protocol may cause excessive update traffic and a write invalidate protocol may perform better.

The choice of whether an update or invalidate protocol should be used has been analyzed by Eggers and Katz for single-bus multis [Egge89]. They concluded that invalidate protocols were preferable for the workloads they analyzed. I argue that *given* that write invalidate performs better than write update for a single-bus multi, that its performance advantages remain the same or increase as system size increases. If all previous readers of a line re-read the line after it is written, then the write update protocol must be performing worse because of multiple update messages, and increasing the system size will not change this. However, if the update protocol results in extraneous updates (updates to cache lines that will not be read again), then its performance relative to the invalidate protocol should become worse as system size increases. This is because, as the system size grows beyond a single bus, the bandwidth used to broadcast an update to all previous readers becomes increasingly larger relative to the bandwidth used to send an update to a single processor. whereas the bandwidth used for an update in a multi is independent of the number of caches being updated. This increases the penalty for extraneous updates. For this reason, and because invalidate protocols appear to be in more widespread favor, this thesis assumes the use of write invalidates.

Scalability arguments can help determine if a particular cache coherence mechanism is well suited for large-scale implementations. I consider two simple coherence mechanisms here, and postpone further analysis until Chapter 5.

5.1. Broadcast invalidate

One simple coherence mechanism is to broadcast an invalidate to all processors whenever a shared line is written. This is the primary approach used for single bus multiprocessors, where a broadcast requires only a single bus transaction. We can immediately see, however, that this approach will not scale for uniform workloads. As the system size increases, each processor will receive invalidations at a rate proportional to N. This will saturate the network or the processors' cache controllers at some point, regardless of topology, and therefore this mechanism is not appropriate for very large systems.

Formally, if the fraction of requests that are writes to shared variables remains fixed (the uniform workload assumption), then $O(N)$ invalidations are generated on each request, each of which must be sent to every processor. From a latency viewpoint, queueing of messages in the network implies that the average latency of a request will grow as $O(N)$, clearly violating the requirement for scalability. From a bandwidth viewpoint, since the number of input lines to any processor can grow by at most $O(logN)$ (the cost requirement for scalability), and the number of incoming invalidation messages per request is $O(N)$, the traffic per line per request grows as at least $O(N/\log N)$, again, clearly violating the requirement for scalability.

It should be noted that for a conspirator workload in which the fraction of writes to shared variables decreases as $O(1/N)$, a broadcast invalidate protocol can scale. In addition, for sufficiently small implementations, a broadcast invalidate protocol may be quite feasible and relatively simple to implement.

5.2. Full-width global directory

Another possible coherence mechanism is to keep track of all shared copies of a line in a global directory. The directory can be distributed along with the memory of the system (as opposed to a "centralized" directory as proposed by Tang[Tang76]). Censier and Feautrier [Cens78] proposed keeping a bit vector of size N for each line, with the corresponding bit set for every processor that has a copy of the line. When the line is invalidated, individual messages are sent to each processor whose bit is set. This does not violate the bandwidth requirement for scalability, assuming that general read traffic scales, because the number of invalidation messages is directly proportional to the number of original read requests that caused the directory bits to be set. However, it clearly violates the cost requirement. The amount of memory storage needed to implement this directory is $O(N^2)$ (assuming that the size of the memory is $O(N)$). In addition, the latency of such an invalidate grows linearly with the number of shared copies, which precludes scalability for workloads with heavy read sharing.

6. Summary

This chapter presented a working definition of scalability and demonstrated its use on several topologies and cache coherence mechanisms. The definition is purposefully informal; the intent being to trade rigor for ease of use. The basic notion is to keep the cost, communication latency and link traffic from growing too fast as system size increases. When used in the wrong way (considering scalability before performance, for instance), the concept of scalability can be abused, but when used correctly, scalability provides useful intuition about the behavior of very large systems.

The k-ary n-cube is a promising topology for future, large-scale multiprocessors. Upon first approximation, k-ary n-cube networks scale very well when the radix is held constant and the dimensionality increased with size. The use of pipelined channels allows the dimensionality to exceed three without causing a corresponding increase in the cycle time of the network, thereby avoiding one of the primary scalability limitations of conventional networks. Wiring constraints, however, can still limit scalability for large systems, as will be explored in Chapter 4.

K-ary n-cubes also provide a substitute for single-tree-based networks (which do not scale for uniform workloads) as a platform for hierarchical communication protocols. The equivalence of buses and rings allows for cube networks to be implemented with either technology, and still employ the same basic protocols. It is likely that "simulated" buses, constructed from point-to-point links, will provide higher bandwidth than their conventional counterparts as logic speeds continue to outpace bus speeds.

Cache coherence mechanisms can greatly affect the scalability of a multiprocessor. A mechanism is needed that scales in space, latency *and* bandwidth. This chapter briefly explored two coherence mechanisms. The simple broadcast invalidate mechanism scales well with regard to cost, but clearly does not scale with regard to bandwidth. At the other end of the spectrum, the full map directory scales with regard to bandwidth, but clearly does not scale with regard to cost, and may not scale for certain workloads with regard to latency. There is a lot of middle ground between these two extremes. Chapter 5 will provide some additional background on proposed cache coherence mechanisms, and present a scalable alternative, pruning-cache directories.

Chapter 3

Analysis of a Pipelined-Channel Ring

In order to recommend pipelined-channel networks, it is not enough to show simply that they are more *scalable* than non-pipelined-channel networks. It is important that they deliver higher *performance* as well. The purpose of this chapter is to demonstrate just that, by analyzing the performance of a small, pipelined-channel network. Once the base performance of a pipelined-channel design is established, we can move on to assess the impact of pipelined channels on the design and scalability of larger networks.

A performance analysis of pipelined-channel networks must be careful to take into consideration any overhead or side effects resulting from the pipelined operation. Care must be taken also to assure that the design being analyzed is realistic and implementable. An ideal method of doing this is to study an existing, "industrial strength" design. A prime candidate for this purpose is the Scalable Coherent Interface (SCI).

1. Background

SCI is a new IEEE standard (1596) that provides very-high-performance, shared-bus-like functionality to a large number of processor nodes [IEEE92, Jame90, Gust92]. Although SCI is an *interface* standard, rather than a *network* standard, the nature of its operation implies pipelined channels. The cycle time is fixed and independent of wire lengths, so that sufficiently long wires will have multiple bits in flight concurrently. Using a packet-based communication protocol over a network of unidirectional links, it provides a shared-memory interface, including cache coherence, to the nodes. The protocol has been developed over a period of approximately four years, has included participation by representatives of dozens of companies and universities, and has assembled appropriate expertise in many different disciplines to solve the plethora of problems associated with a novel design.

There are three major components to the SCI standard: the physical, logical and cache-coherence layers. The logical layer provides the protocol for reliably transmitting packets between nodes. The node interface consists of two unidirectional links, an input and an output, which are used to connect nodes together in the basic topology of a ring. The ring can in theory be arbitrarily large (up to 64K nodes), but performance considerations lead to the expectation that

a ring will be limited to a modest number of processors, numbering at most a few dozen and perhaps as few as two (even *one* is possible). Larger systems can be built by connecting together multiple rings by means of switches (nodes containing more than a single interface). Figure 3.1, for instance, shows a two-dimensional torus constructed from rings. A multistage interconnection network (such as the Omega network) could be implemented by replacing every bi-directional link with a ring of size two, or by more elaborate, higher-performance schemes [John91].

The ring is unusual in that each node provides a bypass buffer capable of temporarily storing a packet arriving from its upstream neighbor while it is transmitting a packet. This buffer allows nodes to transmit concurrently rather than having to wait for a token, but results in long latency if all nodes happen to initiate transmission simultaneously on an idle ring. The basic structure of the SCI ring is similar in nature to the *register insertion ring* [Stal84].

Because of the novel construction of the ring and the attendant clock rates achievable in the design, very high performance is expected, and a peak bandwidth of one gigabyte per second is easy to demonstrate. The nature of the protocol, however, makes both the achievable bandwidth and the observed latency harder to predict. Presented in this chapter is the most detailed study to date attempting to analyze the performance of a single SCI ring. The study deals strictly with the logical layer. The primary criteria for judging the design are its throughput-latency characteristics and its robustness. Performance is analyzed under a variety of workloads and ring sizes. In addition, a mechanism to assure fairness in the ring is investigated to assess its impact on the

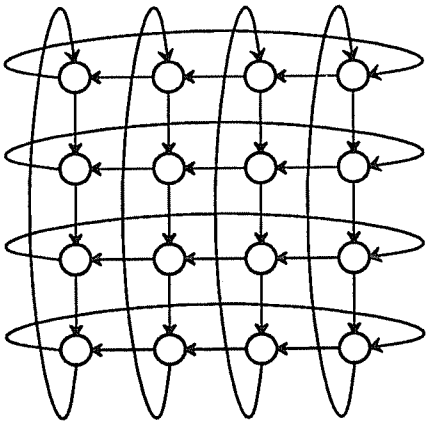


Figure 3.1: A 2-D torus constructed from rings

performance. The SCI protocol does provide priority scheduling, but this aspect of the protocol is not investigated in this study. The ring is studied via analytical modeling and simulation.

The remainder of the chapter is organized as follows. Section 2 describes the protocol of the SCI logical layer. Section 3 describes the analytical model. Section 4 presents and analyzes the results of the study, and Section 5 summarizes the conclusions.

2. The SCI Logical-layer Protocol

The key idea behind the SCI logical layer protocol is the use of unidirectional, point-to-point links that can be clocked at a rate independent of the signal latency between nodes. The basic building block of an SCI system is a *ring* (sometimes called a *ringlet*) of two or more nodes connected by these links. The protocol is designed such that, short of an actual hardware failure, packets are *guaranteed* to be accepted at full speed by each node that they pass through as they traverse the ring. Therefore, a node can output a *symbol* of information on every clock cycle, and there is *no* direct feedback from a node to its upstream neighbor. A packet might not be accepted by its destination, however, due to queue congestion. The protocol uses packet-level acknowledgements to deal with this issue.

2.1. Basic protocol

This section presents a summary of the basic protocol. Details such as ring initialization and error detection/recovery are not covered, nor is all the functionality of the standard presented. Buffer management is somewhat simplified; I assume a single transmit and receive queue per node, whereas the actual system requires dual queues in order to support a higher level protocol. The cache coherence layer of the SCI standard is not considered at all. Much more detail can be found in the standard [IEEE92].

A packet traversing an SCI ring is sent from a *source* node to a *target* node in the form of a *send packet*. The target node then *strips* the send packet, and returns an *echo packet* around the remainder of the ring. This echo packet tells the source node whether or not the send packet was accepted by the target. If the packet was not accepted, then the source must retransmit it.

A send packet consists of a 16 byte header and an optional data component of up to 256 bytes. The header contains command and control information, a 16-bit CRC (Cyclic Redundancy Check) and a 64-bit memory address (16-bit node id and 48-bit intra-node address). I assume a data component size of 64 bytes, which corresponds to the SCI cache line size. Echo packets are 8 bytes long. The link width is 16 bits (the standard defines both a 16 bit copper implementation and a serial, fiber optic implementation)

Figure 3.2 shows a block diagram of an SCI node and ring interface. A node transmits a symbol onto its output link on every SCI cycle. When a node has no packet to transmit, it sends an *idle symbol.* When a source desires to send a packet over the ring, it places the packet in its transmit queue. If the ring buffer is empty and the node is not currently transmitting a packet from the *stripper*, the send packet is immediately output onto the ring. When a source node transmits a packet onto the ring, a copy must either must be saved at the head of the queue (thus blocking further transmissions) or placed into an optional *active buffer*. The copy is either discarded or used for retransmission when the echo packet is received.

Upon arrival at the downstream node, a send packet is parsed and either *stripped* or *passed* along the ring. In the absence of contention, a passing packet may be routed directly from the stripper to the output link. If the ring buffer is not empty or the transmit queue at the node is currently transmitting a packet, the passing packet is routed into the ring buffer. If a passing packet and packet in the transmit queue are ready to transmit on the same cycle, the transmit queue is given priority and the passing packet is routed to the ring buffer.

When the transmit queue is done transmitting a packet, if the ring buffer has accumulated any symbols, output resumes from the ring buffer (which may still be receiving symbols from the stripper). This is known as the *recovery stage*, and lasts until the ring buffer is completely emptied. The node is not allowed to transmit another source packet during the recovery stage. To empty the ring buffer, the node either must see gaps in the stream of incoming packets, or create

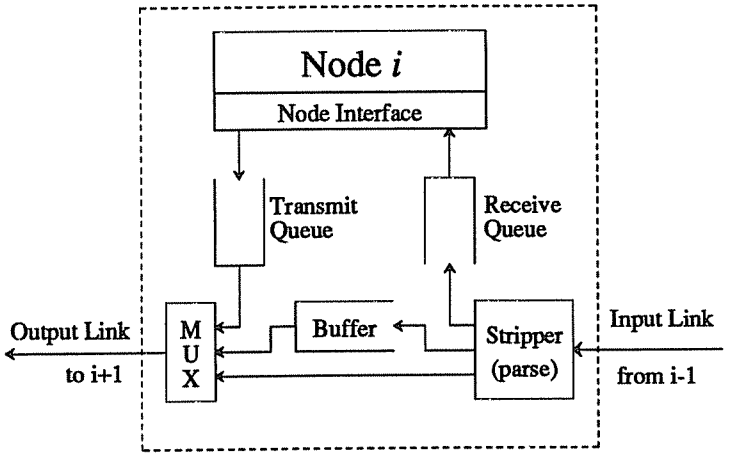


Figure 3.2: An SCI node

gaps in the packet stream by stripping packets for which it is the target. During these gaps, the buffer can be drained while not being simultaneously filled. If the ring buffer is empty after a source transmission completes, then there is no recovery stage.

When a send packet reaches its target, it is stripped and either placed into the receive queue (space permitting) or discarded. The node uses the bandwidth created by stripping the packet to insert idle symbols, transmit symbols from the transmit queue or drain the ring buffer. The last four symbols of the send packet are replaced with an echo packet that continues its way around the ring to the packet's source. At the source, the echo packet is matched with a saved send packet in an active buffer or at the head of the transmit queue, and the appropriate action is taken (discarding or retransmitting the send packet).

One last feature of the protocol that needs mentioning is that packets are always separated by at least one idle symbol. This allows the stripper to periodically delete an idle symbol, if necessary, to adjust for a slowly varying clock period between neighbors (this is known as *elasticity*). It also assures timely distribution of priority and other information carried in the idle symbols. I do not consider elasticity or priorities here, but do require the intervening idle symbols. For the purposes of the basic model, this is equivalent to increasing the length of all packets by one symbol.

2.2. Flow control

The basic protocol described above works fine for uniform traffic rates and routing distributions. However, it allows for nodes to be unfairly starved in the presence of certain non-uniform traffic patterns. Consider a node that partially fills its ring buffer during a transmit queue transmission. If the node then receives a continuous stream of passing packets, then its recovery stage can take arbitrarily long, denying it the chance to transmit another packet. For this reason, the SCI protocol includes a flow control mechanism that uses *go bits* in the idle symbols to enforce an approximate round robin ordering under heavy loads. The flow control mechanism is complicated by a priority mechanism that partitions the ring's bandwidth between high and low priority nodes. While the priority mechanism has certain special uses, such as in real-time systems, it is not likely to be used for general purpose multiprocessors. I assume that all nodes have equal priority, and present the simpler flow control mechanism that results.

Each idle symbol contains a go bit which is either set (making it a *go-idle*) or cleared (making it a *stop-idle*). The stripper passes all idles and passing packets (as well as echos for packets that it strips) to the transmitter stage of the node interface. When it strips a packet, it fills the empty slots with idle symbols. When a node is not transmitting a packet from its transmit queue

and is not in the recovery stage, it simply passes all symbols — send, echo and idle — from the stripper to its output link. Whenever the transmitter emits a go-idle, it continues to emit go-idles until the next packet boundary, possibly converting passing stop-idles into go-idles (this is called go-bit *extension*).

A node may *only* transmit a source packet immediately following a go-idle. During transmission of a packet, a node maintains the inclusive-OR of all go bits it receives from the stripper. If the ring buffer does not fill up at all during transmission, then the node postpends an idle symbol to its packet using the saved go bit it maintained during the transmission, and then continues to either transmit another source packet or output symbols from the stripper.

If the ring buffer *does* fill up at all during transmit queue transmission, then the node enters the recovery stage. All idles sent during the recovery stage, including the idle postpended to the original source transmission, are stop-idles. The node continues, however, to maintain the inclusive-OR of go bits it receives throughout the recovery stage. When the recovery stage ends (the last symbol is drained from the ring buffer), the saved go bit is released in the postpending idle just as it was for the postpending idle of a source transmission when the recovery state was not entered.

The stop idles that are transmitted during a recovery stage inhibit the downstream neighbors from sending new packets and eventually provide enough slack in the incoming packet stream for the node to drain its ring buffer and send a packet. In the absence of contention, all idles on the ring will be go-idles, and a newly arriving send packet can always be sent immediately.

3. Analytical Model

This section presents an analytical performance model of the SCI ring. The model is useful for a variety of reasons: it allows the quick exploration of a large state space, it's precisely defined and can be implemented by other researchers and engineers, and it helps us gain insight into the processes being modeled. Results from the model, as well as from detailed simulations, are presented in section 4.

The model is based upon an approximate, iterative solution of the M/G/1 queue [Klei75] (see Figure 3.3). It does not consider flow control, limited active buffers or target queue overflow. The effect of these factors can be determined from simulation results. The model does consider, and effectively deals with, ring buffer fill-up during transmit queue transmissions, the

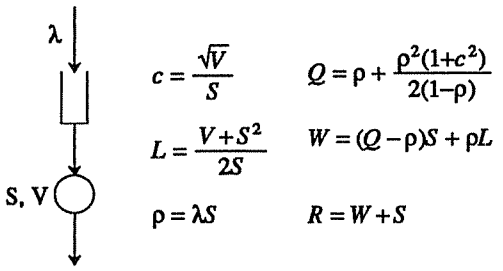


Figure 3.3: Overview of the M/G/1 queue

λ - customer arrival rate (arrivals are Poisson); S - mean service time (service time distribution is completely arbitrary); V - variance of service time; c - coefficient of variation; ρ - utilization; Q - mean number of customers in queue (including any customer in service); L - mean residual life of customer in service; W - mean wait time for a customer before service begins; R - mean response time.

transmission recovery process, the formation and effect of *packet trains*,[2] non-homogeneous packet arrival rates and non-uniform packet destination probabilities, delays due to queueing in ring buffers and variance of transmit queue service times.

Figure 3.4 presents a high-level overview of the model. The bulk of the model concentrates on computing the mean and variance of the transmit queue service time. Service time includes the time to transmit the head of the transmit queue onto the output link *and* drain any accumulated symbols from the ring buffer. After this period, the next packet in the transmit queue will be able to start its service. Given the mean and variance of this service time, the total waiting time in the transmit queue can be calculated using the solution to M/G/1 queue illustrated in Figure 3.3.

A unique feature of the model is that it considers the formation of packet trains on the ring. The extent to which packets are bunched together into trains is characterized by *coupling probabilities*, which are computed separately for each node and link on the ring. The coupling probability on a link denotes the fraction of packets traversing the link that immediately follow the packet in front of them. As will be explained in the following sections, the consideration of packet trains leads to a cyclic dependency in the model equations, which is handled by an iterative solution.

[2] These packet trains arise from collisions between passing packets and source packets as well as from the insertion of trains at source nodes.

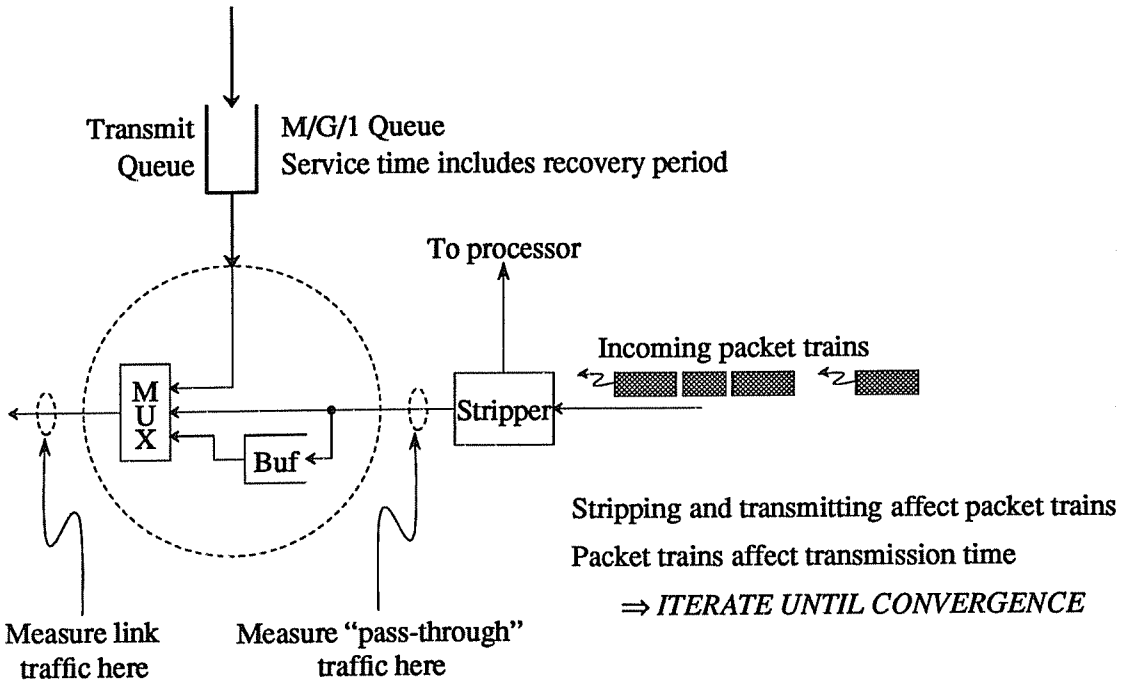


Figure 3.4: Overview of SCI model

3.1. Model inputs

Inputs to the model are the ring size (N), packet arrival rates (λ_i), routing probabilities (z_{ij}), packet lengths (l_{addr}, l_{data}, l_{echo}), packet type ratio (f_{data}, f_{addr}), transmission delay (T_{wire}) and parsing delay (T_{parse}). Note that each node has a distinct arrival rate and a distinct, possibly non-uniform packet destination probability distribution.

3.2. Discussion of model equations

This section provides a brief overview of the model equations. The equations are presented in detail in Appendix A. Equations (A.1) - (A.16) are straightforward. In the first twelve equations quantities such as mean and variance of packets lengths, link utilizations, and various throughputs and ratios are derived directly from the inputs. Note that packet lengths include the idle symbol that is postpended to every packet. The model then ignores these idles and considers only the remaining "free" idles.

Equations (A.13) through (A.16) compute quantities relating to packet trains, which depend upon coupling probabilities. Two key assumptions in the model are that packet trains contain a geometrically distributed number of packets and that the number of idle symbols between packet

trains is geometrically distributed.[3] Thus the mean number of packets in a train is given by the reciprocal of the probability that a packet is not followed directly by another packet (Equation (A.13)), and the probability that an idle symbol is directly followed by a packet, $P_{pkt,i}$, is the reciprocal of the mean space between packet trains (Equation (A.16)).

The transmit queue service time includes the recovery period, which lasts until the ring buffer is empty. Each cycle in which an idle symbol arrives from the stripper allows us to transmit a symbol of information without having to simultaneously place a passing symbol into the ring buffer, thus reducing by one the remaining symbols to send. The service time is over after observing a number of passing idle symbols equal to the length of the packet.

There are two possibilities when transmitting a packet (Equation (A.18)). If a packet arrives when the transmit queue is busy or when the queue is idle and there is currently no passing traffic, then the time required to observe the required number of passing idles is given by a simple binomial distribution (Equation (A.19)). After each idle symbol, another packet train passes through with probability $P_{pkt,i}$. If a packet arrives when the transmit queue is idle but a packet train is passing through the node, then the residual life of the passing packet and the possible interruption of the train must be taken into account as well (Equation (A.20)).

Equations (A.22) - (A.26) compute new estimates of the coupling probabilities. Equation (A.22) considers new couplings that are formed when a send packet is injected at node *i*. Equation (A.23) calculates the mean number of coupled packets that enter the stripper at node *i* for each packet stripped, where stripped packets include echo packets that are consumed and send packets that are converted into echo packets. Equations (A.24) and (A.25) consider couplings from the upstream neighbor that are removed when a packet is stripped at node *i*. Equation (A.26) computes the net effect of stripping packets. Note that these equations assume that coupling probabilities are not correlated with packet length. The relation between service time and coupling probabilities is cyclic. The equations are solved iteratively until the coupling probabilities converge.

After the above convergence, several metrics can be computed. The variance of the packet train length (Equation (A.28)) is computed using the geometric distribution of the number of packets in a packet train. Equations (A.29) - (A.33) compute the variance of the transmit queue service time. This calculation involves an approximation involving the correlation between two

[3] I comment on the accuracy of these and other model assumptions in section 4.8.

components of the service time. Equations (A.34) - (A.37) compute other values relating to the solution of the M/G/1 queue.

The mean backlog seen by a packet passing through node i (Equation (A.38)) is computed by dividing the total backlog created by an injected packet, by the mean number of passing packets per injected packet. Finally, the mean transit time (Equation (A.39)) and mean overall response time (Equation (A.40)) are computed.

Experience implementing this model has shown that convergence is faster for smaller ring sizes. The convergence criteria used in this study is an average change in coupling probabilities of less than 10^{-5}. Approximately 10 iterations were needed for N=4, 30 for N=16 and 110 for N=64. Total time to solve the model for N=64 on a DECstation 3100 is about 1 second. Comparable simulation time (simulating 9.3 million cycles, as was done in this study) is over 4 hours.

4. Results

This section presents results derived from both the analytical model and a detailed, parameter-driven simulator of the SCI ring. The inputs to the model and to the simulator are identical. The ring is modeled as an open system (Poisson arrivals), with the arrival rates, packet lengths, mix of packet types, routing probabilities, ring size, wire transmission delay and packet parsing delay specified as inputs. The simulator has the additional ability to consider flow control and limited buffer space (active buffers and receive queues). Since the ring is modeled as an open system, latency becomes infinite as saturation is reached. An actual system, of course, would have a limit to the number of queued or outstanding requests, and nodes would be stalled at some point rather than continuing to add requests (*i.e.* actual systems would likely be closed rather than open). Section 4.6 illustrates the component of total delay that is due to queueing.

The unit of length in the model and simulator is one link width, and the unit of time is one clock cycle. A 16-bit link with a 2 ns cycle time are used, as per the standard. Using these assumptions, output latencies are presented in *ns* and throughputs in *bytes per ns*. Throughputs are calculated using the entire packet, including address, command and control information. Section 4.7 considers sustained data throughput using a read request/read response model.

Many other parameters have been fixed or limited in order to make the problem space tractable. Since the number of nodes in a ring is expected to be small, ring sizes of 4 and 16 nodes are analyzed. Except where noted, 60% of send packets are address/command only (16 bytes), and 40% include data blocks (80 bytes) (these are referred to as address packets and data packets, respectively). This corresponds to a workload in which most of the traffic consists of paired address and data packets. A fixed minimum delay of 4 cycles per node traversed by a packet is

used: one cycle to gate a symbol onto an output link, one cycle for the symbol to reach its downstream neighbor and two cycles to parse a symbol before routing it to the local node or to the next output link. Message latencies also include one cycle to originally queue the packet, and a delay equal to the packet length to consume the packet as it arrives at the target node. Unlimited active buffers are assumed at each node, but only one or two active buffers are actually needed to approximate this [Scot92].

The simulator implements the protocol described in section 2 on a cycle by cycle basis, explicitly tracking each symbol on the ring. Simulations were run for 9.3 million cycles each, and 90% confidence intervals were computed using the method of batched means. Confidence intervals were generally under or about 1%, except near saturation, where they sometimes increased to a few percent.

4.1. Uniform traffic

Figure 3.5 shows the performance of 4- and 16-node SCI rings with uniform arrival rates and routing probabilities and no flow control. Each graph includes three sets of data, one with all address packets, one with all data packets and one with 40% data packets. Both simulation and model results are shown. The model is very accurate for the 4-node ring. For the 16-node ring, the model is accurate for the all-address-packet workload, but underestimates latency under

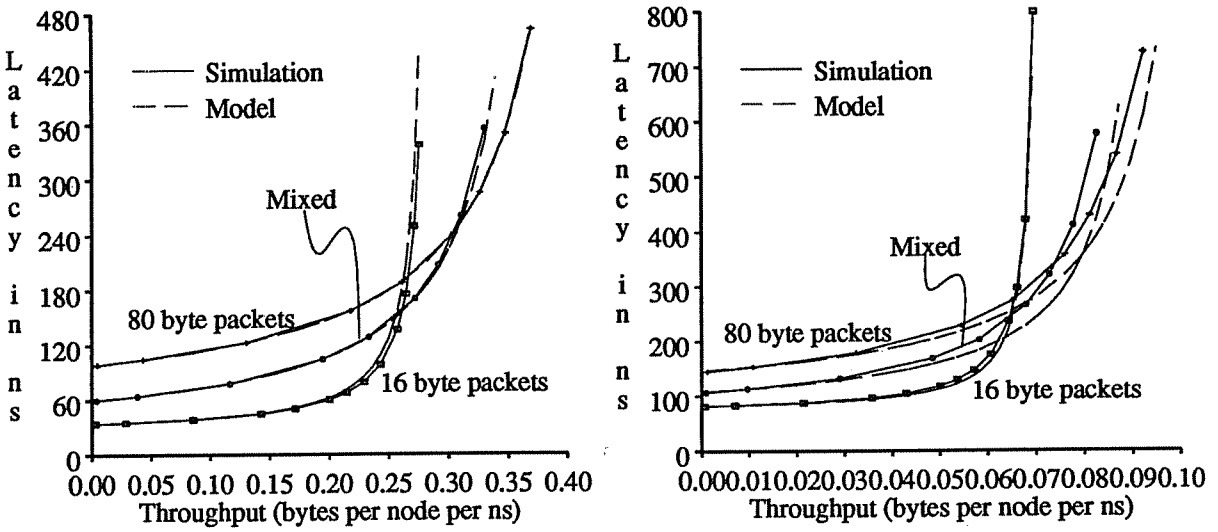
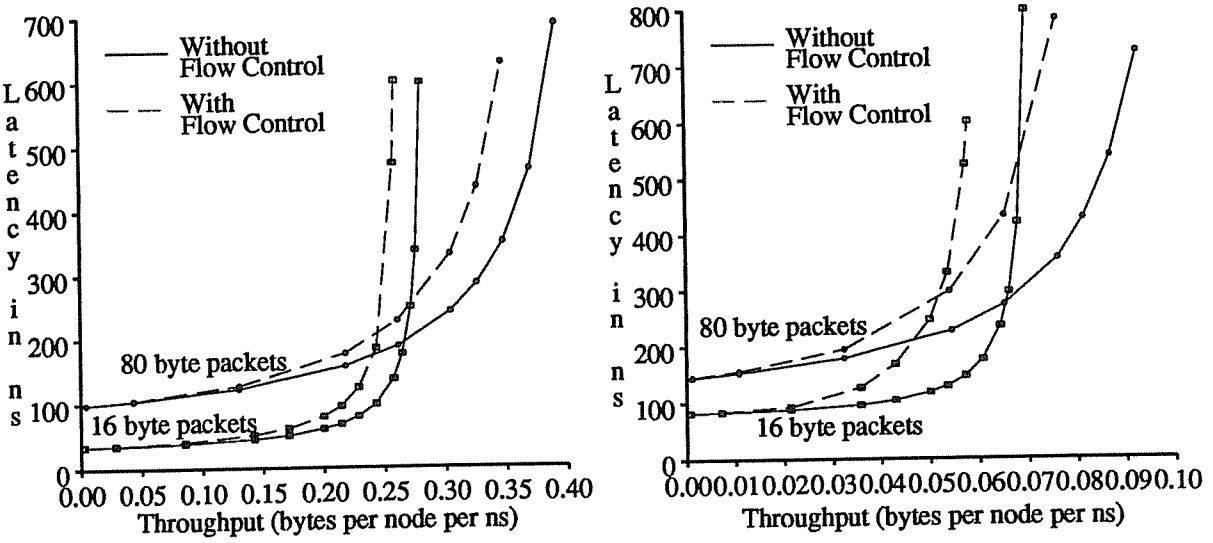


Figure 3.5: Uniform traffic without flow control

moderate to heavy loading for the other workloads. Even for the worst case, however, the model provides a good estimate for the behavior of the ring. The reason for the error is identified and discussed in section 4.9.

Throughput is higher for the workload with larger packet sizes. There are two reasons for this. First, a smaller proportion of the ring bandwidth is used for the idle symbols that must separate each packet. Second, the bandwidth consumed by echo packets becomes smaller relative to the bandwidth used by send packets. Throughput could also be increased by use of packet locality. Unlike a shared bus, a ring requires less bandwidth if the packets are sent a shorter distance (message latency is similarly reduced). For the purposes of this study, equally distributed destinations are assumed.

Figure 3.6 illustrates the effect of flow control on uniform traffic for ring sizes of 4 and 16. Each graph includes two sets of data, one with all address packets, and one with all data packets. Results for the mixed address/data workload fall in between these. We can see that even with uniform traffic loading, flow control significantly reduces the maximum throughput. The reason for this is that there are times when a node cannot transmit a source packet, even though there are available slots in which to do so, because another node has stopped sending go bits in order to clear its ring buffer. The degradation is greater for the 16-node ring than for the 4-node ring.



(a) N = 4 (b) N = 16

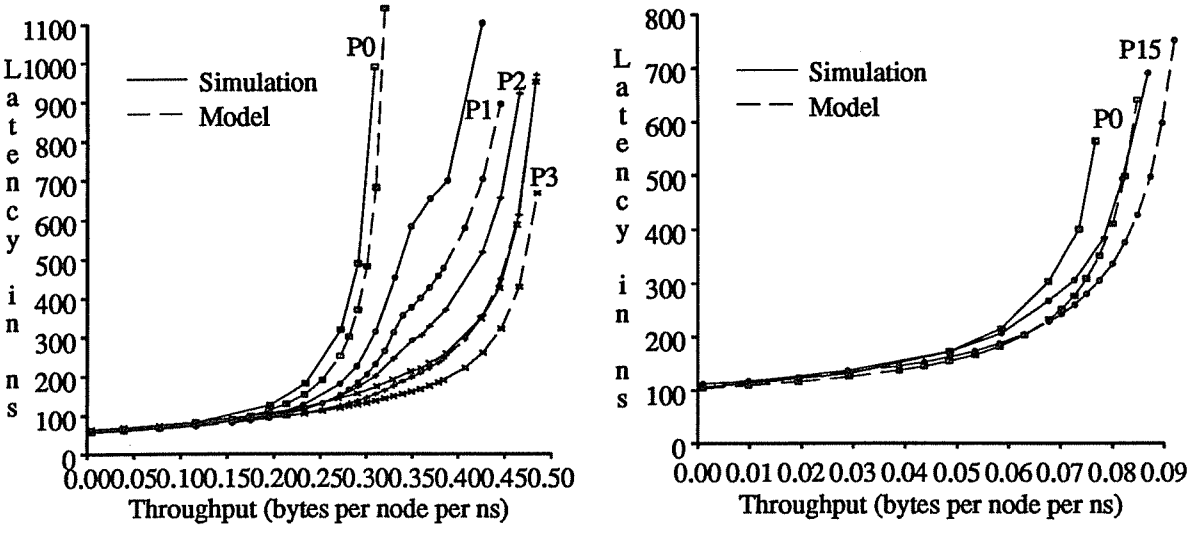
Figure 3.6: Effect of flow control on uniform traffic

Simulations indicate that the throughput degradation from flow control is greatest for ring sizes in the 10 to 20 range, and actually lessens slightly for larger rings [Scot92].

4.2. Node starvation

This section examines the situation in which a node is inhibited from transmitting by reducing the number of breaks it sees in its pass-though traffic. Figure 3.7 presents the performance for 4- and 16-node rings where all nodes are routing uniformly, except that no packets are routed to node 0 (the starved node). Mean message latencies are plotted for individual source nodes (labeled P0, P1, *etc.*). In Figure 3.7(a), we see that P0 saturates before the other nodes. As the throughput per node reaches about 3.2 bytes/node/ns, P0's arrivals can no longer be satisfied and its message latency goes to infinity (recall that this is simulated as an open system). As P1, P2 and P3 increase their throughput beyond this point, the realized throughput of P0 is actually driven back down to 0. This causes the unusual shape in the curves for P1 and P2. P1, P2 and P3 all reach the same saturation bandwidth.

The equations in the model assume that the system is not in saturation, so in order to model the behavior after P0 has saturated, the model detects saturated queues, and automatically throttles back the corresponding arrival rates to keep the transmit queue utilization at exactly one. The model qualitatively predicts correct behavior, including the throttling of P0's throughput and the



(a) N = 4 (b) N = 16

Figure 3.7: Node starvation without flow control

corresponding inflection points in the P1 curve. However, the model underestimates the impact of the non-uniform traffic, and quantitative error is fairly large when the starved node is in saturation.

For a ring size of 16 (Figure 3.7(b)), the disparity between nodes is not as pronounced. The starved node reaches almost as high a bandwidth as the other nodes before it saturates. This is because the non-uniform routing causes smaller differences in link utilizations for the larger ring. The model correctly predicts the spread in performance between the starved node (P0) and the least affected node (P15). The absolute error, however, is fairly significant under heavy loads.

Figure 3.8 demonstrates the effect of flow control on node starvation. Parts (a) and (b) show the message latency for each node as the traffic is varied. In parts (c) and (d), the ring is in saturation (all nodes are trying to send as often as possible), and the realized throughput for each node is shown.

In Figure 3.8(a), we see that the addition of flow control reduces the disparity between the performance of the four nodes, but at an overall reduction in throughput. The throughput of P0 is not driven back down by the other nodes, as it is without flow control. Note, however, that the performance is not fully equalized; P0 achieves a smaller maximum throughput than P1, P1 achieves a smaller maximum throughput than P2, *etc.*

Figure 3.8(c) shows the saturation bandwidths for the 4-node ring with P0 still being starved. Without flow control, P1, P2 and P3 all achieve the same throughput, but P0 is *completely* starved. Because the ring is fully utilized and it is not receiving any packets, it has no opportunities in which to transmit a packet (*i.e*: it enters an infinite recovery stage). The flow control mechanism successfully deals with this problem. With flow control, the total ring throughput is reduced slightly, and the throughput of the non-starved nodes is reduced significantly, but the starved node is no longer kept from transmitting. The flow control mechanism does not achieve full equal partitioning of bandwidth, however. The throughput of a node is limited by how quickly it can empty its ring buffer after a transmission, and the flow control protocol guarantees that even a starved node will make forward progress in the recovery stage by throttling downstream transmissions and thus creating gaps in its incoming packet stream. However, a node whose pass-through traffic is lower overall (due to non-uniform traffic) will still be able to recovery more quickly on average.

Figure 3.8(b) shows the effect of flow control for a ring size of 16 with P0 being starved. The addition of flow control almost completely equalizes the performance of the various nodes (again at an overall reduction in ring capacity). Figure 3.8(d) shows the saturation bandwidths for

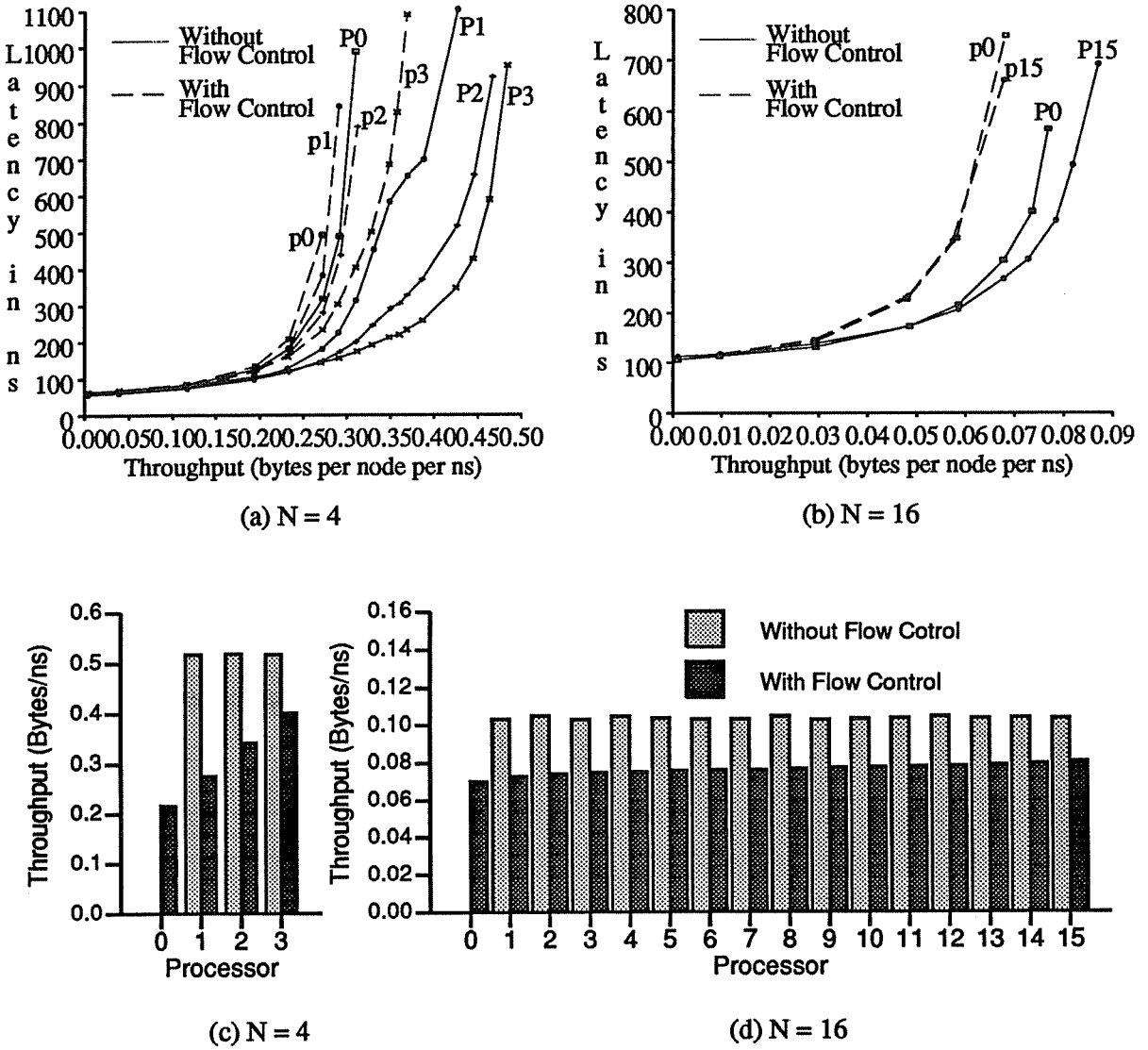


Figure 3.8: Effect of flow control on node starvation

the 16-node ring. Although the impact on P0 was small under light to medium traffic, P0 is completely starved when the ring is fully loaded. Flow control reduces the realized throughput of the non-starved nodes and allows the starved node to transmit. The bandwidth is much more equally divided than it was for the 4-node ring. This is because the differences in link utilizations caused by the non-uniform traffic are smaller for the larger ring.

4.3. Hot sender

This section examines the ring's behavior in the presence of a "hot sender" (a node that attempts to use as much ring bandwidth as possible). Figure 3.9 presents the performance for 4- and 16-node rings where packet destinations are uniformly distributed, but node 0 always wants to transmit a packet. P1, the first downstream node from the hot sender, is severely affected by the extra traffic. The hot node degrades the performance of all other nodes on the ring, affecting the closest nodes more heavily.

The model is very accurate for a ring size of four (Figure 3.9(a)). For a ring size of 16 (Figure 3.9(b)), the model is qualitatively accurate, but slightly underestimates latency for most of the nodes, and significantly overestimates the latency for the immediate downstream neighbor of the hot node (the reason that the model overestimates P1's latency is actually that it underestimates P0's latency (the hot node), allowing P0 to send more than it would in reality).

Figure 3.10 demonstrates the effect of flow control on a hot sender. Parts (a) and (b) show the message latencies for each node as a function of throughput. The addition of flow control equalizes the effect of the hot node on the message latency for the other nodes on the ring. Performance is improved for some nodes and degraded for others, but the hot node's downstream neighbor is no longer severely penalized.

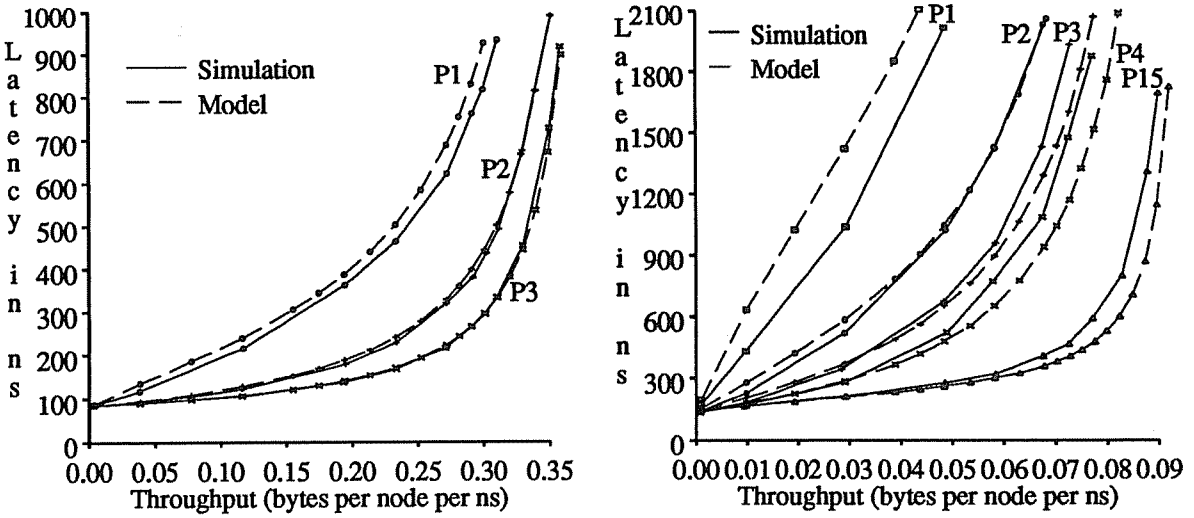


Figure 3.9: Hot sender without flow control

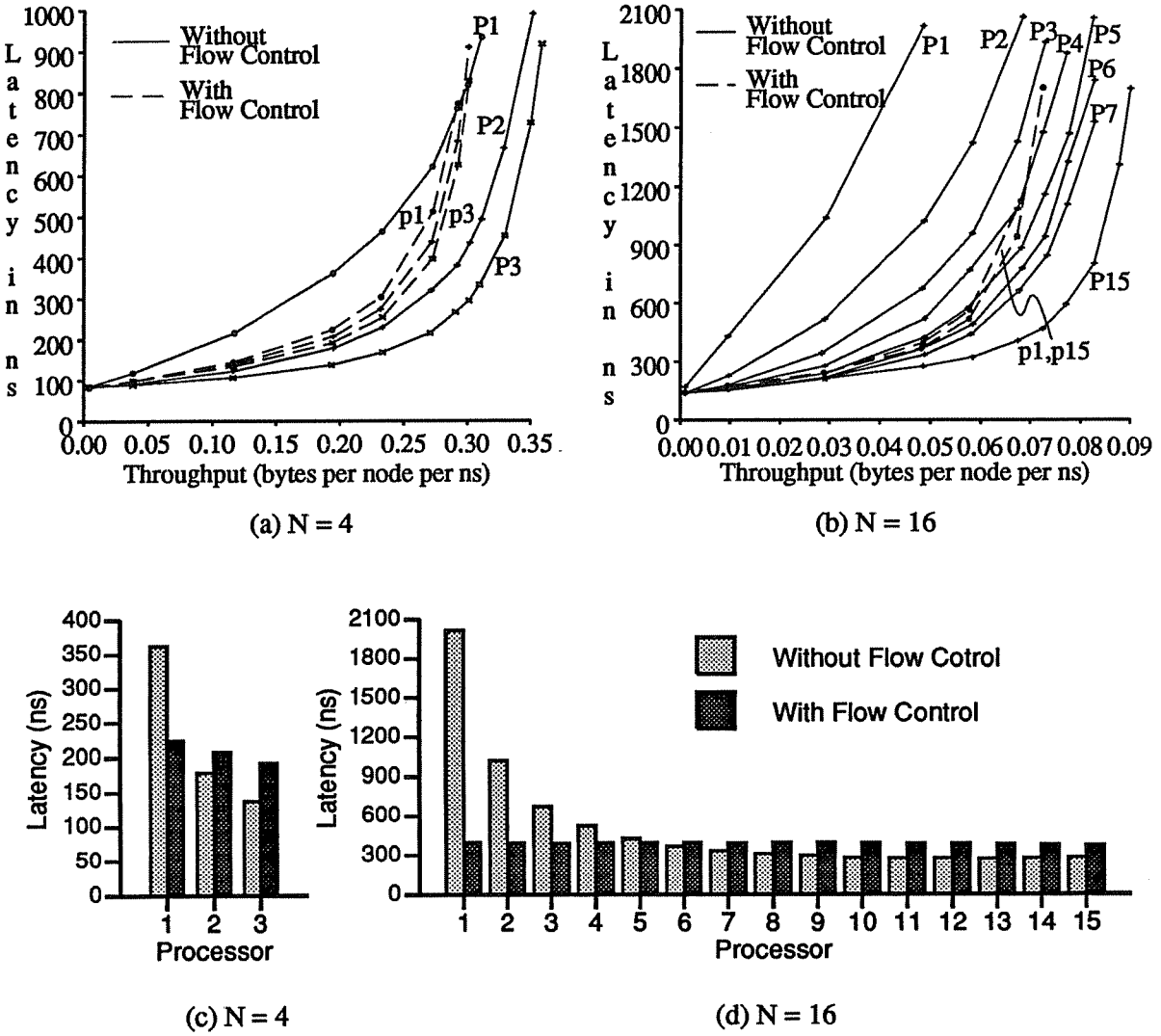


Figure 3.10: Effect of flow control on hot sender

This phenomenon can be clearly seen in parts (c) and (d), which show vertical slices of the throughput-latency curves under moderate throughput from the "cold" nodes (0.194 bytes/ns in Figure 3.10(c), 0.048 bytes/ns in Figure 3.10(d)). Without flow control, the mean message latencies experienced by the cold nodes vary significantly, with the closest downstream nodes being affected the most. With flow control, the hot node affects all other nodes approximately equally. The nearest downstream node, in particular, is no longer subjected to extremely large latencies. The improved ring fairness is achieved at the expense of the hot sender's throughput. Without flow control, it realizes a rate of 0.670 bytes/ns on the 4-node ring. With flow control, it realizes only 0.550 bytes/ns. For the 16-node ring, the hot sender's throughput is reduced from 0.526

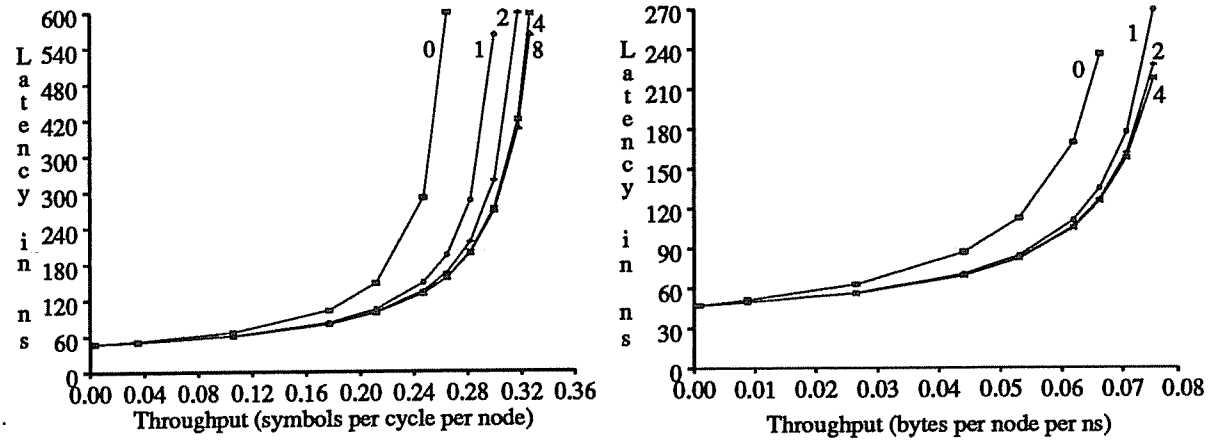
bytes/ns to 0.293 bytes/ns. For certain applications, most notably real-time systems, it may be desirable to allow one node or a set of nodes to consume more than their share of ring bandwidth. SCI provides a priority mechanism to satisfy this requirement.

In addition to hot senders and node starvation, I have examined producer-consumer and other non-uniform workloads. Though not presented here, the results are similar. The flow control mechanism reduces the effects of greedy nodes on the rest of the ring, and provides all nodes with a reasonable approximation to their share of the bandwidth, regardless of the non-uniformities present in the communication pattern.

4.4. Varying the number of active buffers

Active buffers are used to store transmitted packets while awaiting acknowledgements. Figure 3.11 illustrates the effect of varying the number of active buffers in the ring interfaces. With no active buffers, only a single send packet can be outstanding from a node at any time. Each additional active buffer allows another packet to be transmitted before the first packet is acknowledged. Both the 4- and 16-node rings benefit from a single active buffer, and the 4-node ring benefits from a second active buffer. There is very little incremental benefit from additional buffers.

It is somewhat counter-intuitive that the smaller ring benefits more from additional active buffers, because the time between transmitting a packet and receiving the acknowledgement is



(a) N = 4 (b) N = 16

Figure 3.11: Varying the number of active buffers

greater for the larger ring. The smaller ring, however, has a greater per-node packet arrival rate, given the same ring utilization. This increases the need for multiple outstanding requests at one node. Another way to view this is that the overall packet arrival rates for the rings are the same, and thus the smaller ring needs to be able to buffer more active packets per node. Certain types of systems may require a greater number of active buffers, perhaps only at specific nodes on the ring.

4.5. Comparison to a conventional bus

This section compares the SCI ring to a conventional, synchronous bus. Conventional wisdom holds that, while rings may provide higher throughput than shared buses, the buses provide lower latency. The results in this section show that this need not be the case.

The prime advantage of SCI, at the logical layer, is its use of fast, point-to-point links. The unidirectional nature of the communication allows the cycle time to be limited only by the speed of the technology. Standard ECL circuitry available in 1992 allows a 2 ns clock. To compare this against a conventional bus, a simple M/G/1 bus model was used. The model assumes no overhead for arbitration, and single-cycle synchronous transmission in 32-bit chunks (the pin-out for an SCI interface is also 32 bits: a 16-bit input link plus a 16-bit output link). The means service time for the bus model is simply the mean packet length in bus widths, and the service time variance is simply the packet length variance. The response time is then given by the standard M/G/1 solution (see Figure 3.3).

Figure 3.12 compares the throughput-latency characteristics of an SCI ring to a bus as the bus cycle time is varied. Data for the SCI ring are from the simulator with flow control in effect. The workload is 60% address packets (16 bytes) and 40% data packets (80 bytes). Ring/bus sizes of 4 and 16 nodes are used. If a synchronous bus had the same cycle time as the SCI ring, it would clearly provide better performance. This is due not only to the bus' greater width, but to the single cycle broadcast latency. With a bus cycle time of 4ns, latency is still lower when lightly loaded, but the maximum throughput is also lower. This is due to the bus' lack of concurrency.

As the bus cycle time is increased, the latency goes up significantly, and the maximum throughput drops off significantly. Realistic bus cycle times range from 20 to 100 ns. A typical high performance shared bus has a 30 ns cycle time. The Stardent Titan graphics supercomputer uses a 31.25 ns bus [Siew91], for example, and the Silicon Graphics Power Series computers use a 30 ns bus [SGI89]. The ELXSI System 6400 used expensive twisted-pair ECL with differential signaling for their shared backplane, achieving a cycle time of 25 ns [Olso83]. The SCI ring

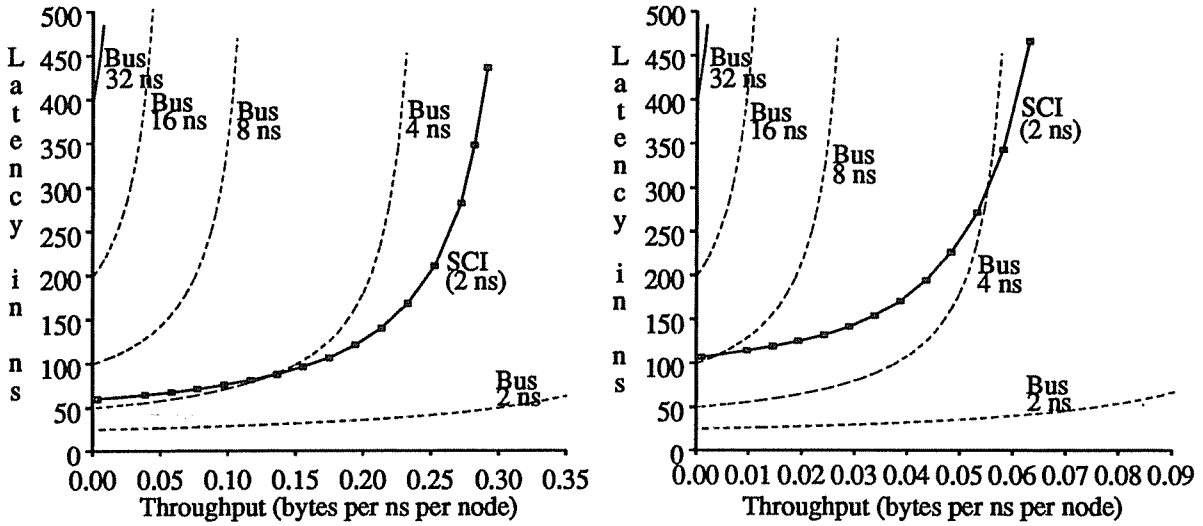


Figure 3.12: SCI ring vs conventional bus

provides far greater bandwidth and lower latency than a bus of comparable width running at 20 ns or slower.

As the number of nodes on a ring increases, the average message latency will increase. As the number of nodes on a bus increases, the average message latency will also increase, due to greater contention for the bus and because the cycle time of the bus will have to be increased to accommodate the greater capacitive loading and longer physical distances. Because of the increased cycle time, the total bandwidth of the bus will decrease as well. The cycle time of an SCI ring is independent of ring size.

4.6. Sustained data throughput using a request/response model

This section considers total sustained data transfer rates on a ring. I assume that the ring traffic consists solely of read request packets and their associated read response packets. Latencies represent an address packet transmission from a processor to a memory, followed by a data packet transmission from the memory to the processor including receipt of the entire data block (memory lookup time is not included). The data block size is 64 bytes, and the throughput includes only the data bytes.

The results are shown in Figure 3.13. Since an address packet is 16 bytes and a data packet includes a 16 byte header along with the 64 bytes of data, exactly two thirds of the send packet symbols contain data. The actual data throughput is thus two thirds of the total throughput. The

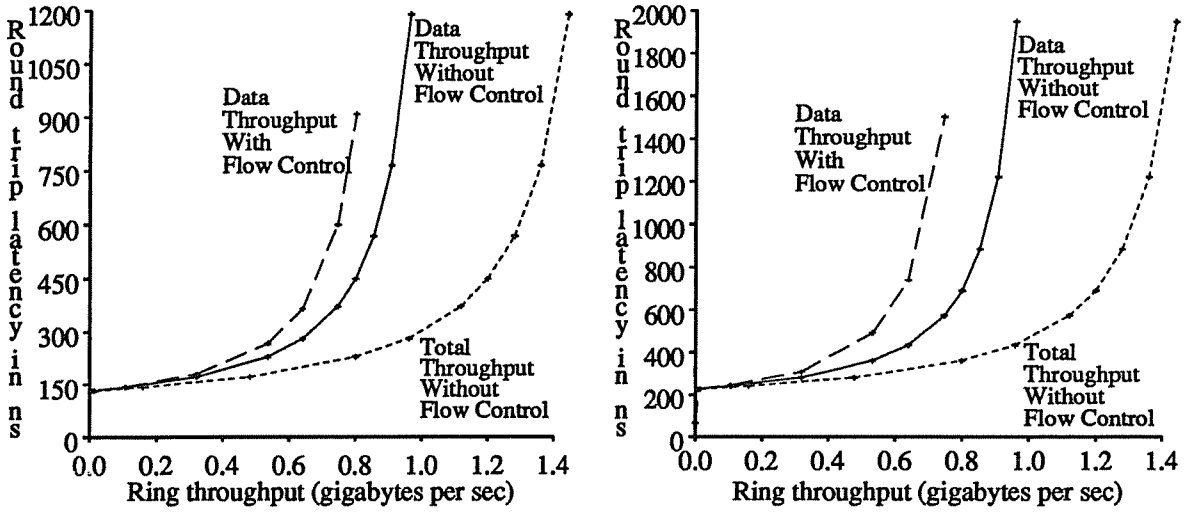


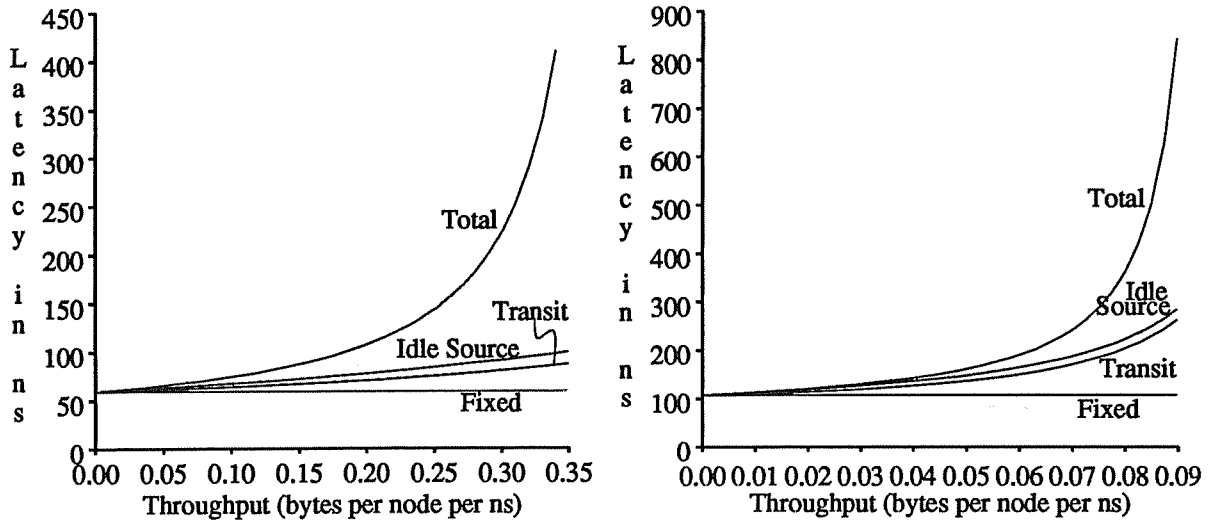
Figure 3.13: Sustained data throughput

throughput shown in Figure 3.13 is the total ring throughput, measured in gigabytes per second.

4.7. Breakdown of message latency

In this section, the mean message latency is broken into several components. Figure 3.14 plots results from the analytical model for ring sizes of 4 and 16. The packet traffic is uniform, with 40% of the packets containing 64-byte data blocks. The latency is broken into 4 components. The *Fixed* curve represents latency due to wire transmission delay and fixed switching overheads. The *Transit* curve represents the time from when a transmit queue begins transmitting until the packet is consumed at the destination. The difference between these two curves is due to delays passing through the ring buffers. The *Idle Source* curve represents the latency seen by a packet arriving at an idle transmit queue (there are no packets in front of it and the node is not in the recovery stage). The difference between the *Transit* curve and this curve represents the time a source packet may have to wait while a packet finishes passing through the node. The *Total* curve represents total, end-to-end latency. The gap between the *Transit* curve and this curve represents the total time a packet is queued at a transmit queue before receiving permission to transmit.

Most of the latency under heavy loads is due to waiting in the transmit queues. In a closed system (where there is a limit on the number of queued packets), the delay due to transmit queueing would level off at some point. Delay due to buffer backlog becomes more significant relative



(a) N = 4 (b) N = 16

Figure 3.14: Breakdown of message latency

to transmit queueing delay as the ring size is increased from 4 to 16. For very large rings, transmission delay becomes dominant except when the ring is very close to saturation.

4.8. Discussion of model error

In this section, several possible sources of error in the analytical model are identified, and the likely significance of each is discussed.

First, the model assumes geometrically distributed inter-packet-train spacing, whereas simulations show that certain lengths of spaces are much more common. For example, the space created by replacing an address packet with an echo packet occurs with high frequency. However, simulation estimates of the coefficient of variation of the inter-packet-train spacing, are very close to 1. Thus, I do not anticipate that this assumption causes significant error.

Second, in computing the variance of the recovery period, the correlation between the residual life of a passing packet train and the remaining portion is unknown. As an approximation, the model assumes a correlation of one, and treats the portion due to residual life of a passing packet train as a constant multiplier to the remaining recovery time (Equation (A.25)). This slightly overestimates the service time variance, but I do not believe this causes significant error.

Finally, the model assumes that the transmit queue utilization and the pass-through ring utilization are independent. That is, that we see the same rate of passing packets regardless of whether the transmit queue is in use. *This is the primary source of error in the model.*

Simulation statistics indicate that the pass-through traffic is lower than average when the transmit queue is idle, and higher than average when the transmit queue is active (during the transmission/recovery stage). This causes the model to underestimate the length of the recovery stage, thus underestimating the overall message latency. The error increases as the mean length of the recovery period increases, which causes the error to grow for larger rings and packet sizes.

Two worthwhile directions for future research are to reduce the error in the current model and to extend the model to account for flow control.

5. Conclusions

This chapter presented a performance study of the SCI ring, including a description of the logical-layer protocol, an efficient, analytical performance model, and extensive simulation results. The performance of the SCI ring was analyzed under uniform and non-uniform workloads and with and without the flow control mechanism. The SCI ring was also compared to a conventional shared bus.

The analytical model developed for the SCI ring did not consider the flow control mechanism, but was found to be accurate for both uniform and non-uniform communication patterns. Where quantitative error was greater, qualitative behavior was still predicted correctly. The primary source of error in the model was identified, and will be the topic of further research, along with extensions to model flow control mechanisms.

The SCI flow control mechanism effectively prevents node starvation, providing all nodes with their approximate fair share of the ring bandwidth. Non-uniform routing still affects the realized node throughputs to some extent, however, with the effect being greater for smaller ring sizes. The flow control mechanism also equalizes the negative impact that a hot node has on the rest of the ring. Without flow control, the downstream neighbors of a hot node see substantially increased message latencies.

The fairness provided by the flow control mechanism comes at the cost of overall ring throughput. Maximum throughput is reduced by up to 30%. The impact is greatest for ring sizes of 8 to 32, and is negligible for a ring size of 2. Possible modifications to the flow control mechanism are being investigated that would gracefully increase maximum ring throughput in return for reduced fairness.

Buffering issues were also briefly analyzed. The performance of an SCI ring can be appreciably improved by the addition of a single active buffer at each ring interface (allowing two outstanding packets from each node). Additional active buffers have very limited added benefit,

although they help smaller rings more than larger rings.

In comparing the SCI ring to a conventional bus, the clock speed was considered along with the number of cycles needed to convey a message. Although the number of cycles is larger for the ring, the faster clock speed gives a significant advantage. A 32-bit bus would have to have a 4 ns clock to be competitive with a 16-bit wide SCI ring with a 2 ns clock (and even then it would have a lower saturation bandwidth). While a 2 ns clock for SCI is realizable in 1992 with standard ECL circuitry, typical high performance multiprocessor buses have cycle times of about 30 ns.

With a 16-bit width and 2 ns cycle time, the SCI ring provides a total peak throughput of over 1 gigabyte per second for uniform traffic. Communication locality can increase this maximum. The flow control protocol decreases maximum throughput, but partitions the ring bandwidth fairly and provides uniform message latency to all nodes. The SCI standard leaves room for future improvements by both increasing the link width and decreasing the cycle time.

The overall conclusion we can draw from this study is that an actual pipelined-channel network, complete with flow control and error-handling mechanisms, can deliver very high performance for small system sizes. Coupled with the superior scaling properties of pipelined-channel networks, as discussed in Chapter 2, the case for using pipelined channels in large-scale networks is compelling. The next chapter explores the trade-offs in multi-dimension, pipelined-channel networks in great detail.

Chapter 4

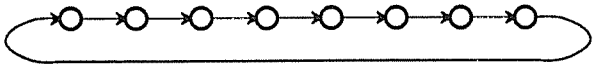
Large-Scale Pipelined-Channel Networks

Arguments presented in Chapter 2 of this thesis suggested the use of the k-ary n-cube topology, based both upon its scalability properties and its suitability as a framework for hierarchical communication protocols. The consideration of wire transmission latency led to the suggestion of pipelined-channel networks to avoid the constricting effect of long wires on network cycle times. Chapter 3 addressed the basic performance of a pipelined-channel design, demonstrating that even for small system sizes, the small cycle time achievable using pipelined channels leads to very high network performance. The task of this chapter, then, is to thoroughly analyze the effects of pipelined channels on large, multi-dimensional networks.

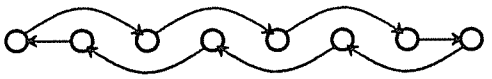
1. Background

A wide variety of interconnection networks have been proposed in the literature (see [Feng81] or [Sieg79] for a summary), each of which can be classified as either *direct* or *indirect*. Indirect networks, such as the omega network [Lawr75], connect processors and memories through multiple intermediate stages of switching elements. Direct networks incorporate the processing elements within the network itself, allowing for direct communication between processors, and therefore allowing communication locality to be exploited [Seit84]. Direct networks are gaining in popularity and have been employed in many recent existing or proposed machines, including the Thinking Machines CM2 [Hill85], Intel iPSC and Paragon, Cosmic Cube [Seit85], MIT Alewife [Agar90], Tera supercomputer [Alve90], CMU-Intel iWarp [Bork90], and Stanford DASH multiprocessor [Leno89].

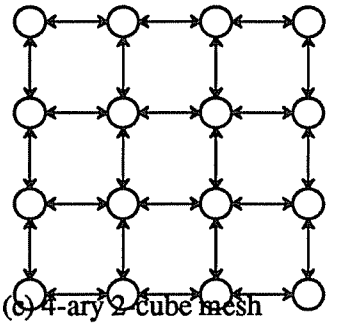
The most commonly used direct networks are variants of the k-ary n-cube. Recall that a k-ary n-cube consists of $N=k^n$ nodes, arranged in n dimensions with k nodes per dimension. Figure 4.1 illustrates several different k-ary n-cubes. Each node is connected via a direct link to its nearest neighbors in each of n dimensions. Links can be bi- or uni-directional, and, if bi-directional, the wrap-around links may be omitted. Examples of k-ary n-cubes include the ring (n=1), 2-dimensional mesh or torus (n=2), 3-dimensional mesh or torus (n=3) and hypercube (k=2). The generality and flexibility of the k-ary n-cube make it an excellent choice for exploring network design tradeoffs.



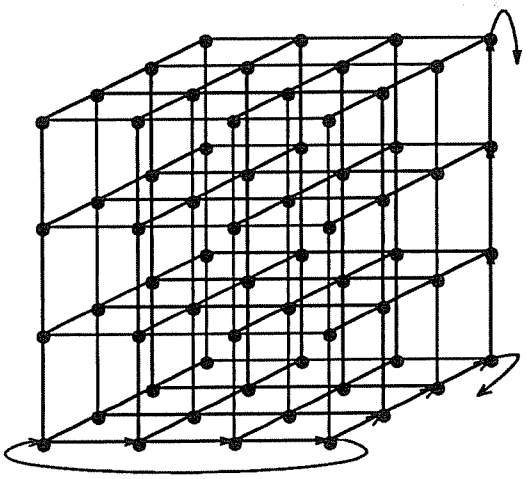
(a) 8-ary 1-cube torus (N=8, k=8, n=1)



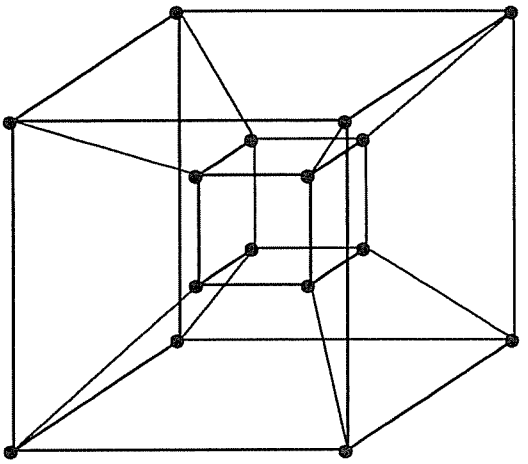
(b) Folded 8-ary 1-cube torus (N=8, k=8, n=1)



(c) 4-ary 2-cube mesh (N=16, k=4, n=2)



(d) 4-ary 3-cube torus (N=64, k=4, n=3)



(e) 2-ary 4-cube (N=16, k=2, n=4)

Figure 4.1: Example k-ary n-cube networks

For a given system size, the primary design choice is the dimensionality of the network. Varying the dimensionality of a network affects the average number of links that messages must traverse and the average rate of traffic across the links. As discussed in Chapter 2, it may have other effects on system design and performance as well. Most previous network studies have made the simplifying assumptions used in Section 4.1 of Chapter 2: that communication latency is measured only in hops (with a single network hop taking constant time) and that link width is independent of dimensionality. Under these assumptions, binary hypercubes appear very attractive, delivering the lowest latency and link traffic of any k-ary n-cube configuration. These assumptions are unrealistic, however.

Dally has investigated network performance while taking wire delay into account and applying the *constant bisection* constraint [Dall90]. This constraint holds the number of wires

crossing the bisection of a network constant as the dimensionality is varied, which causes the link width to decrease as dimensionality is increased. The constraint is motivated by wiring density limitations in VLSI, but may also hold for multi-chip or multi-board implementations. Dally's conclusion, given a linear delay model for wire transmission time, was that the dimensionality of a network should be equal to the number of physical dimensions in which the network is implemented, regardless of network size. The result of this is that the optimal radix of the network increases significantly for large systems.

Agarwal [Agar91] extended Dally's analysis in two important ways. First, he included switching time in the latency equations, which was missing in Dally's study. Second, he considered a weaker wiring constraint: *constant node size*. This constraint is motivated by pin limitations on boards and chips, and holds the number of wires connected to each network node constant as the dimensionality is varied. This also causes the link width to decrease as dimensionality is increased, but more slowly than does the constant bisection constraint. Agarwal concluded that the optimal dimensionality was generally higher than the number of physical dimensions. While he did not specifically address scaling networks, his analysis leads to the conclusion that both n and k should be increased as network size is grown.

Both Dally's and Agarwal's work assumed the use of *non*-pipelined-channel networks, in which the cycle time of the network includes the transmission time across the longest wire in the network. This exacts a heavy penalty on high-dimensional networks because their longer wires give rise to longer cycle times. In a pipelined-channel network, data is clocked onto the wires at a rate determined solely by the switching speed, allowing multiple bits to be simultaneously in flight on sufficiently long network wires. This decouples link throughput from link latency, and fundamentally changes the network design tradeoffs.

The pipelined-channel routing protocol described in Section 2.1 of this chapter also provides some of the same benefits as virtual channels [Dall87, Dall92]. By guaranteeing that buffer congestion occurs only when entering/exiting a dimension, deadlock free routing is provided for uni-directional k-ary n-cubes. In addition, sustained throughput is able to come close to maximum throughput, especially when k is large (see [Scot92] and Section 4 of this chapter). Conventional networks using flit-level flow control typically provide half or less of their maximum throughput, due to coupled resource allocation [Dall90]. Note, however, that the switches described in Section 2.1 require buffers able to contain an entire packet.

Pipelined channels have long been used in wide area networks and local area networks, since the physical delays involved compel their use. There is an abundance of research in the literature regarding transmission protocols, reliability, flow control, routing, performance and

many other issues regarding these networks [Tane88]. Pipelined channels are not widely used in multiprocessor interconnects, however. Limited pipelined channels (allowing a small, fixed number of bits on a wire) are used in the CMU-Intel iWarp [Bork90], various Cray Research machines[Smit92] and the Thinking Machines CM5 [TMC91]. A higher degree of pipelining is achievable using the Caltech Slack chip [Seit92], which is designed to allow tightly-coupled Mosaic channels to communicate efficiently over long cables. The IEEE Scalable Coherent Interface (SCI) [IEEE92], on which I base my model, allows an arbitrary amount of pipelining.

The remainder of this chapter presents a performance study of pipelined-channel k-ary n-cube networks, with particular emphasis on how the design tradeoffs differ from those of non-pipelined-channel networks. Section 2 develops equations for latency in an unloaded network and shows how pipelining argues for higher dimensionality. Section 3 discusses network bandwidth and presents simulation results for pipelined-channel and non-pipelined-channel networks. Section 4 investigates the effects of switching overhead and packet length. Concluding remarks are presented in Section 5.

2. Unloaded Latency

An important metric of network performance is the mean latency of packet transmission in the absence of contention (the unloaded latency). Both switching and wire transmission delay contribute to this latency. The manner in which switching and transmission delays interact is a primary difference between pipelined-channel and non-pipelined-channel networks. This is further explained in the following sections. Section 2.1 describes the network node model and other assumptions. Sections 2.2 and 2.3 derive formulas for the unloaded latency in uni-directional, pipelined- and non-pipelined-channel networks. Sections 2.4 and 2.5 discuss wire length and link width. Section 2.6 discusses bi-directional networks. Finally, Section 2.7 uses the results from earlier sections to calculate the optimal dimensionality of a network under various assumptions and constraints.

The unit of time used in all the analysis is the switch cycle time, *not* including wire transmission delay. For pipelined-channel networks, this *is* the network cycle time, and so reported latencies are in units of cycles. For non-pipelined-channel networks, for which the cycle time must include wire transmission delay, reported latencies are equal to the number of cycles *times* the relative increase in cycle time due to wire transmission delay.

2.1. Model and assumptions

The first assumption made in this work is that we are dealing with uni-directional links. There are two reasons for doing this. First, previous analysis [Dall90, Agar91] has focused on uni-directional networks. Second, pipelined channels are naturally uni-directional. While the results are qualitatively similar for bi-directional networks, some of the details change (see Section 2.6).

The network node model used here is shown in Figure 4.2. To route a packet between two points in the network, the packet is transmitted over n rings, one in each dimension. A dimension is skipped if the packet source and destination have the same ordinate in that dimension. A low-

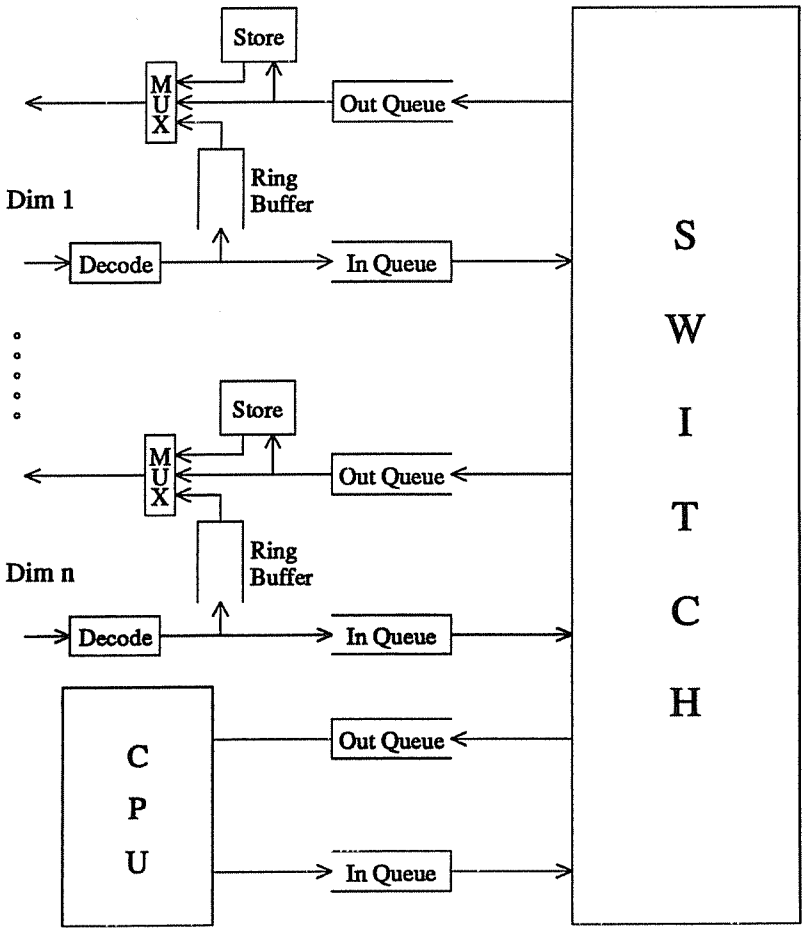


Figure 4.2: A node from a pipelined-channel k-ary n-cube network

level protocol handles the transmission around each ring, independent of the other dimensions and any higher-level, end-to-end protocol. The ring interfaces are based loosely on the IEEE SCI logical layer protocol discussed in Chapter 3 [Scot92, IEEE92]. The links are W bits wide, so a packet of L bits is decomposed into P *flits*, with P given by

$$P = \left\lceil \frac{L}{W} \right\rceil \tag{4.1}$$

Routing is similar to *virtual cut through* [Kerm79]. On being placed in the input queue by the CPU, a packet is switched to the output queue for the appropriate dimension and gated onto the output link (requiring T_{switch} cycles). It then uses T_{wire} cycles to travel to the next network node, where T_{wire} may vary, depending upon the wire length. A copy of the packet is saved locally for possible retransmission. When the head of the packet arrives at a node, T_{decode} cycles are spent decoding the packet. The number of decode cycles is determined by the number of flits needed to form the node address of the packet destination:

$$T_{decode} = \left\lceil \frac{\log_2 N}{W} \right\rceil \tag{4.2}$$

If continuing in the current dimension, the packet is routed through the ring buffer and gated onto the ring output link (requiring T_{pass} cycles). If changing dimensions, the packet is routed through the input queue, switched to the appropriate output queue and gated onto the output link for the new dimension (requiring T_{switch} cycles).

Values of T_{pass}=1 and T_{switch}=2 are assumed. The overhead for changing dimensions is greater than that for continuing in the same dimension, due to the extra switching step involved. To reduce complexity and increase switching speed, the switch could be implemented as a n-element ring. Contention on the ring would be minimal, because most traffic would be routing to the next dimension and would thus need to travel only one hop along the ring (much like a cube connected cycle). In this case, of course, the switching delay would be greater for packets changing to other than the next highest dimension.

Low-level routing in each dimension is modeled after the logical layer of the SCI protocol [IEEE92]. When a packet is removed from a ring (either switched to the processor or to a different dimension), an *echo packet* is routed the remainder of the way around the ring to acknowledge the receipt of the packet on this ring. Packets are stored at the node in which they first enter a ring until the ring acknowledgement is received. It is possible that an input queue will have insufficient space to accept a packet (due to contention), in which case a *negative acknowledgement* is returned around the ring instead. When a negative acknowledgement is

received, the packet is retransmitted.

Ring level acknowledgements are used to avoid the round-trip latency of handshaking between nodes on every flit. An alternative would be to perform handshaking on blocks of data, allowing for data within the blocks to be pipelined, and thus amortizing (or hiding) the handshaking latency over several flits. Ring acknowledgements avoid handshaking latency altogether and retain performance as transmission delays increase. In addition, they can be used to provide fault tolerance by checking a CRC (cyclic redundancy check) at the end of the packet [IEEE92]. Unlike the input queue, the ring buffer is guaranteed to be able to accept a flit on every cycle. An output queue may only initiate a packet transmission on the output link if the ring buffer has enough free space to hold a packet of equal length. In this way the ring buffer cannot fill up before the packet has been completely drained from the output queue. The SCI protocol, on which this is based, is discussed in more detail in Chapter 3, Section 2.

2.2. Latency in a pipelined-channel network

The unloaded latency can be derived by simply accounting for the delays described in Section 2.1. The mean number of hops (assuming uniformly distributed destinations) between nodes in a uni-directional, k-ary n-cube torus is

$$h_{uni}\ mark = n\left\lceil \frac{\sum_{i=0}^{k-1} i}{k} \right\rceil$$

$$= n\left\lceil \frac{k-1}{2} \right\rceil \tag{4.3}$$

Since the switching overhead is greater for changing dimensions than for continuing in the same dimension, we must break this down by dimension. A packet routes in a dimension with probability $\frac{k-1}{k}$, and traverses a mean of $\frac{k}{2}$ hops in a taken dimension. The total mean unloaded latency, in cycles, is

$$Latency_{pipe} = T_{switch} + n\left\lceil \frac{k-1}{k} \right\rceil \left[\frac{k}{2}(T_{wire_{avg}} + T_{decode}) + \left\lceil \frac{k}{2} - 1 \right\rceil T_{pass} + T_{switch} \right] + P - 1 \tag{4.4}$$

Note that the delay due to wire transmission is *added* to decoding and switching delays; it does *not* affect the switch cycle time. This allows the cycle time to be kept small and prevents transmission delay from affecting the queueing delay in the switches and the delay between receiving the head and tail of a packet. $T_{wire_{avg}}$ is the average number of cycles spent traversing a

link. The number of cycles spent traversing a link must be integral, but it may vary for different links, depending upon wire lengths. The values of $T_{wire_{avg}}$ and W are explored in later sections.

2.3. Latency in a non-pipelined-channel network

In a non-pipelined-channel network, the cycle time must include both switch operation and wire transmission. As such, the maximum wire transmission delay has a multiplicative effect on overall latency. It is reasonable to assume for a non-pipelined-channel network, with its larger cycle time, that $T_{pass} = T_{switch}$, and so I use only T_{switch} (and assume a value of 1 cycle), simplifying the expression for latency. The non-pipelined-channel latency is given by

$$Latency_{non-pipe} = \left\lceil 1+t_{wire_{max}} \right\rceil \left[T_{switch} + n \left\lceil \frac{k-1}{2} \right\rceil \left(T_{decode} + T_{switch} \right) + P - 1 \right] \qquad (4.5)$$

Recall that the time unit in Equation (4.5) is the switch cycle time *not* including wire transmission delay, which allows for direct comparison with Equation (4.4). In this case, $t_{wire_{max}}$ is the time spent traversing the longest link in the network (and is not an integer in general). Thus, the first factor, $\left\lceil 1+t_{wire_{max}} \right\rceil$, represents the increase in cycle time of the non-pipelined-channel network due to wire transmission delay. The remaining factor represents the number of cycles used to transmit the packet. Equation (4.5) is similar to Equation (4.4), save that wire delay is taken out of the cycle *count*, and instead is used to increase the cycle *length*.

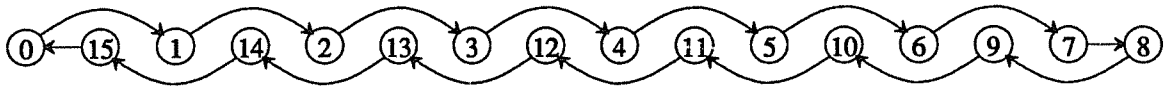
2.4. Wire length

Wire lengths depend upon network dimensionality and layout. I assume that the network is being implemented in three physical dimensions (this differs from [Dall90], where two dimensions are assumed). I also make the simplifying assumption that interprocessor spacing is equal in all dimensions. The long wraparound links in a torus (see Figure 4.1(a)) can be avoided by *folding* the network as shown in Figure 4.1(b). All wire lengths are doubled (over those of a mesh), except for the two end links. Let l_3 be the maximum wire length in a folded, three-dimensional torus. The *mean* wire length is $l_3 \left\lceil \frac{k-1}{k} \right\rceil$, which is smaller due to the two shorter links per ring. This can now be used as a base to compute wire lengths for higher-dimensional networks.

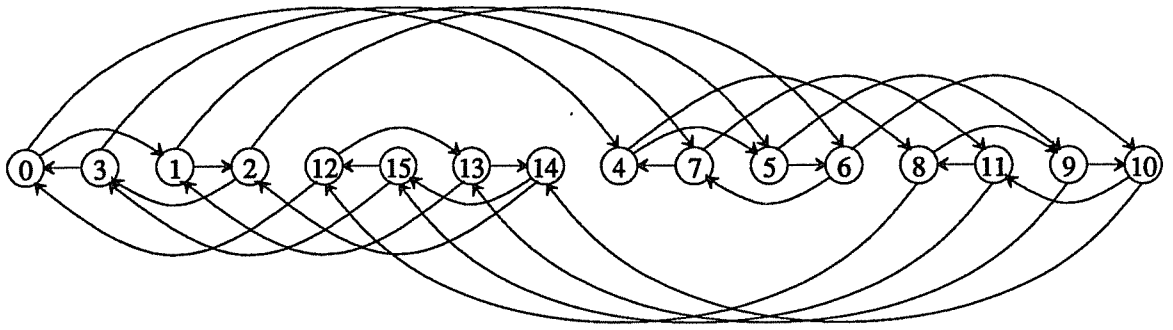
For a network with $n>3$ dimensions, not all links can connect close physical neighbors. Assume that processors are arranged in a three-dimensional cube with $k^{n/3}$ nodes in each dimension. We can embed $n/3$ logical dimensions in each of the three physical dimensions in the

following manner. Address the nodes along a physical dimension as 0 through ($k^{n/3}$–1). A node's address is composed of $n/3$ k-bit digits, which represent its ordinate in each of the $n/3$ logical dimensions. Each node has $n/3$ output links in each physical dimension, connecting it to its downstream neighbors in each of the $n/3$ logical dimensions. The address of the downstream neighbor in dimension i is obtained by adding one, modulo k, to the i^{th} digit of the node address. Figure 4.3 illustrates this embedding, with folding, in a single physical dimension.

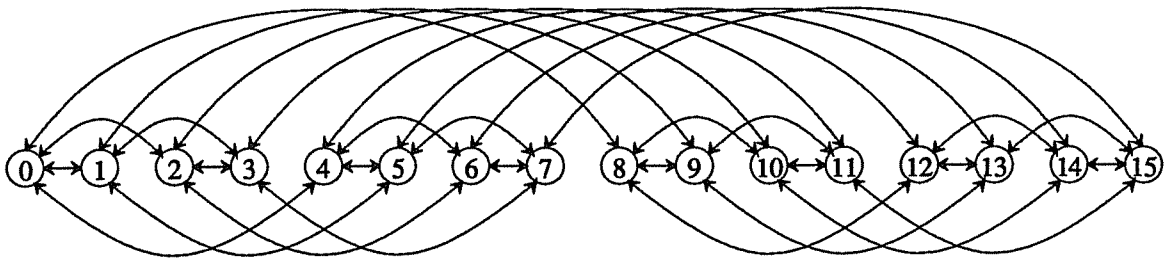
The length of the *longest* wire in such a network grows as $\frac{N^{1/3}}{k} = k^{\frac{n}{3}-1}$. Recall, however, that two of the folded links in each logical dimension are shorter than the others (see Figure



(a) One logical dimension (k=16)



(b) Two logical dimensions (k=4)



(c) Four logical dimensions (k=2)
(each arc represents 2 links in this figure)

Figure 4.3: Embedding multiple logical dimensions in a single physical dimension
The networks with k>2 are folded.

4.3(a)). For this reason, a binary hypercube actually needs no folding, and the longest wire when k=2 (Figure 4.3(c)) is the same length as the longest wire when k=4 (Figure 4.3(b)).

In non-pipelined-channel networks, the maximum wire length is important, as it determines the cycle time of the network. Let S be the ratio of switch cycle time (not including wire transmission) to wire transmission time in a three-dimensional torus. We can now compute $t_{wire_{max}}$, the time to transmit across the longest wire in the network:

$$t_{wire_{max}} = \begin{cases} \frac{1}{k}\left\lceil \frac{N^{1/3}}{S} \right\rceil & \text{if } k>2 \\ \frac{1}{4}\left\lceil \frac{N^{1/3}}{S} \right\rceil & \text{if } k=2 \end{cases} \tag{4.6}$$

Since the maximum wire delay is suffered over *all* links, including the short ones, the total delay due to wire transmission is increased as the dimensionality of a network is increased [Dall90, Agar91].

In a pipelined-channel network, the number of cycles spent traversing a given wire is determined by that wire's length only. Thus $T_{wire_{avg}}$ is a function of the *mean* wire length rather than the maximum wire length. The mean wire length in an n-dimensional network, $n \geq 3$, is given by

$$\begin{aligned} l_n &= l_3\left\lceil \frac{k-1}{k} \right\rceil \left[\left\lceil 1+k+\cdots+k^{\frac{n}{3}-1} \right\rceil \left\lceil \frac{3}{n} \right\rceil \right] \\ &= \left\lceil \frac{l_3(k-1)}{k} \right\rceil \left\lceil \frac{k^{n/3}-1}{k-1} \right\rceil \left\lceil \frac{3}{n} \right\rceil \\ &= \left\lceil \frac{3l_3(N^{1/3}-1)}{nk} \right\rceil \end{aligned} \tag{4.7}$$

and the average number of cycles to traverse a wire is given by

$$T_{wire_{avg}} \approx \left\lceil \frac{3(N^{1/3}-1)}{nkS} \right\rceil \tag{4.8}$$

To obtain an accurate value for $T_{wire_{avg}}$, the wire transmission delay must be rounded up to an integral number of cycles for each link before averaging, taking into account the two shorter links per ring. If the number of logical dimensions is not a multiple of three, then n *div* 3 logical dimensions are embedded in each of the physical dimensions, and n *mod* 3 of the logical

dimensions are embedded across all three physical dimensions. The wire lengths for these left-over dimensions can be short (approximately l_3), and the wire length for all other dimensions is increased by a factor of $k^{\frac{n \bmod 3}{3}}$.

While higher dimensionality leads to greater *maximum* wire length, it does not increase the *total* wire length traversed by a packet. Let d_{uni} be the mean total distance traversed by a packet in a uni-directional torus. If we multiply the mean number of hops (Equation (4.3)) by the mean wire length (Equation (4.7)), we obtain

$$d_{uni} = \left[\frac{k-1}{k}\right]\left[\frac{3}{2} l_3 (N^{1/3}-1)\right] \tag{4.9}$$

which actually *decreases* slightly as the dimensionality is increased. Therefore, wire delay is not an impediment to increasing the dimensionality of pipelined-channel k-ary n-cubes. Note that this result does not hold for bi-directional meshes, as is discussed in Section 2.6.

2.5. Link width

The link width, W, is affected by the dimensionality of the network and the constraint under which it is being designed. The default assumption is the constant link width constraint (which is really no constraint at all). It assumes that the link width, which determines the number of flits into which a packet must be decomposed, is independent of the dimensionality of the network. Since the number of wires attached to a node is equal to $2nW$, this causes the total number of wires per node to increase as the dimensionality of a network is increased.

The constant node size constraint keeps the wires per node fixed as the dimensionality is varied. The link width is then some constant divided by n. The constant bisection constraint keeps the number of wires across the bisection fixed as the dimensionality is varied. The number of wires across the bisection is $B = 2Wk^{n-1}$, so link width is given by

$$W_{const\_bisec} = k\left[\frac{B}{2N}\right] \propto k \tag{4.10}$$

The constant bisection constraint was used by Dally[Dall90] in order to reflect the limited wiring area of a network implemented on a single VLSI substrate. A multiprocessor implemented across multiple boards may also be bisection constrained, or may be node-size constrained due to pin limitations off-chip and off-board. The constant link width constraint may be realistic for sufficiently small systems, depending upon the technology used to implement the network. Intra-node data paths, for example, may dictate the link width and the pin limitations may

not be restrictive. Agarwal [Agar91] considers all three constraints, with the emphasis on the constant node size constraint.

2.6. Bi-directional networks

Pipelined channels are naturally uni-directional, and the analysis thus far has assumed uni-directional networks. This section briefly discusses bi-directional networks. Although routing is simpler in uni-directional networks, there are some disadvantages to their use as well. The first is that the mean number of hops that a packet must traverse is greater for a uni-directional network. The second is that locality is harder to exploit because a node can't communicate directly with its "upstream" neighbor. Both factors are less important when the radix is small.

Bi-directional k-ary n-cubes can be either tori or meshes. The mean number of hops between nodes in a bi-directional, k-ary n-cube torus is

$$h_{bi\_torus} = \begin{cases} n\left\lceil \frac{k}{4} \right\rceil & \text{for } \textit{even } k \\ n\left\lceil \frac{k-1/k}{4} \right\rceil & \text{for } \textit{odd } k \end{cases} \tag{4.11}$$

The mean number of hops between nodes in a bi-directional, k-ary n-cube mesh is

$$h_{mesh} = n\left\lceil \frac{k-1/k}{3} \right\rceil \tag{4.12}$$

Bi-directional tori must be folded to avoid long wrap-around links, but meshes do not, and hence their base wire length is only $\frac{l_3}{2}$.

While pipelined-channels lend themselves to uni-directional networks, they *can* be used in bi-directional networks as well. One way to do this is to replicate all links and run one set in the opposite direction, but this requires halving the link widths. Another is to use packet-level handshaking on individual links, allowing the direction to be switched between packets. It may also be feasible to have two opposite-traveling optical signals sharing the same physical channel.

If pipelined channels are used with bi-directional networks, then it is important to reconsider the relationship between dimensionality and mean total distance traversed by a packet. Recall that for uni-directional tori, the mean total distance decreases slightly as dimensionality is increased (Equation (4.9)). In a bi-directional torus, the mean total distance traversed by a

packet, derived by multiplying h_{bi_torus} (assuming even k) by l_n, is given by

$$d_{bi\_torus} = \left\lceil \frac{3}{4} l_3 (N^{1/3} - 1) \right\rceil \tag{4.13}$$

which is *independent* of dimensionality.

In a mesh, because there is no folding, the mean wire length is given by $\frac{l_3}{2}\left\lceil \frac{N^{1/3}-1}{k-1} \right\rceil \left\lceil \frac{3}{n} \right\rceil$. The mean total distance traversed by a packet is thus given by

$$d_{mesh} = \left\lceil \frac{k+1}{k} \right\rceil \left\lceil \frac{l_3}{2} (N^{1/3} - 1) \right\rceil \tag{4.14}$$

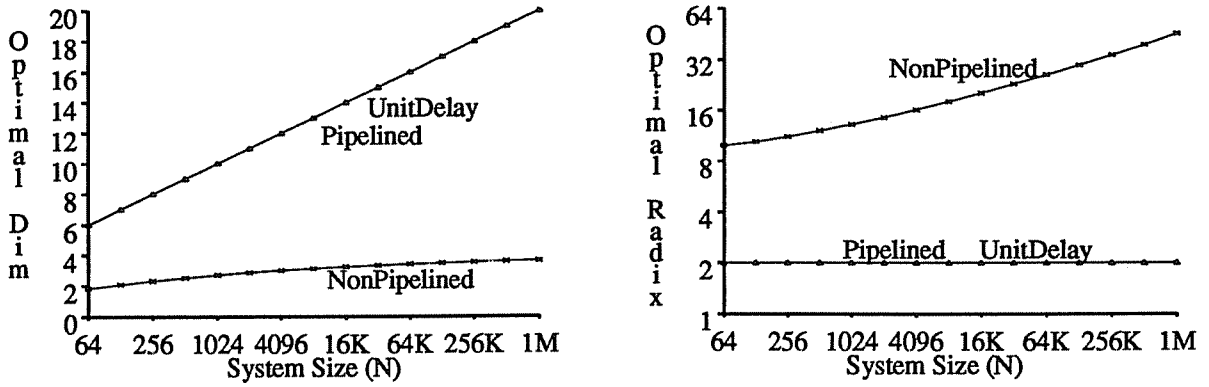
which *increases* as dimensionality is increased. Low dimensional meshes have no backtracking (packets can always traverse the shortest Manhattan distance between nodes) because they do not require folding. The backtracking that is introduced by increasing the dimensionality causes total distance to increase.

2.7. Optimal dimensionality as network size grows

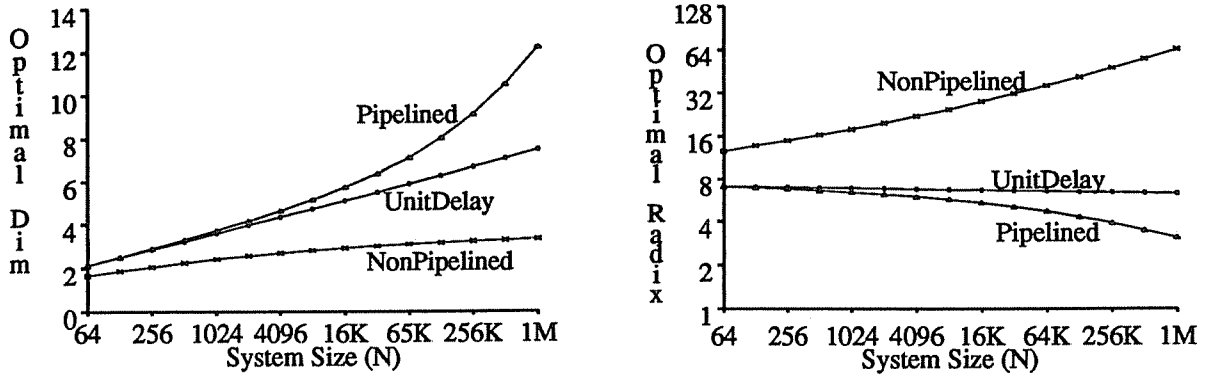
Using Equations (4.4) and (4.5), we can now calculate the dimensionality that gives the lowest unloaded latency for a given system size and constraint. I will consider three network variants. The first, *NonPipelined*, assumes a non-pipelined-channel network, and is characterized by Equation (4.5). The second, *Pipelined*, assumes a pipelined-channel network, and is characterized by Equation (4.4). The third, *UnitDelay*, ignores wire length completely and assumes a unit delay for each network cycle. This has been a common assumption in previous studies, and while not realistic, is useful for purposes of comparison.

Figure 4.4 plots the optimal dimensionality and radix of a network (based on minimizing unloaded latency) as system size grows. Note that the optimal dimensionality and optimal radix are related by the equation $N=k^n$. Parts (a), (b) and (c) assume the constant link width, node size, and bisection constraints, respectively. Each graph presents results for each of the three network variants. The ratio of switch cycle time to base wire delay, S, is 1. The latency used to determine the optimal network configuration is the sum of two packet transmission latencies: one with a 16 byte packet and one with an 80 byte packet. These correspond to the address and data packet sizes in SCI [IEEE92].

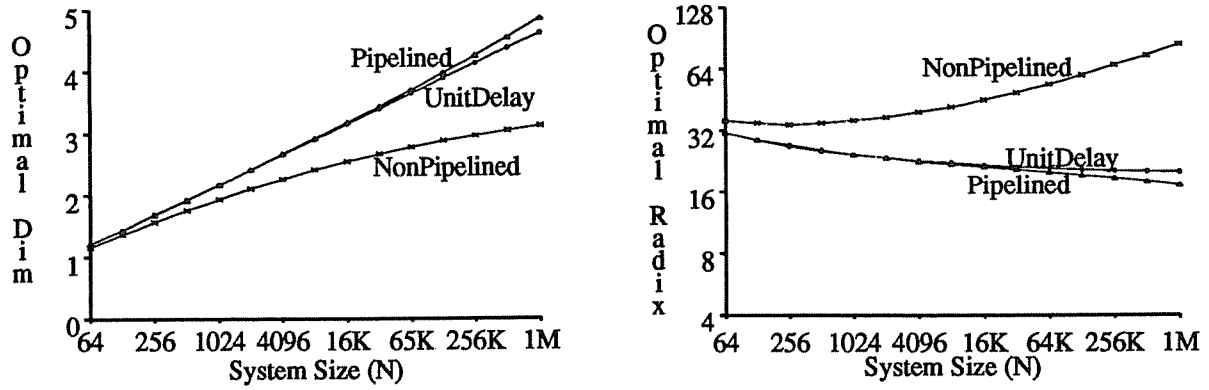
To compute the optimal dimensionality, all discrete quantities are treated as as continuous. Thus, wire and decode delay cycles as well as the dimensionality and radix of the networks can be fractional. When all discrete effects are included, the qualitative results are the same, but the



(a) Constant link width (W=32)



(b) Constant node size ($2nW$=192)



(c) Constant bisection width (W=2 for hypercube)

Figure 4.4: Optimal dimensionality and radix versus network size (uni-directional torus)

results are very sensitive to arbitrary values such as the normalized link width, and the graphs jump erratically, obscuring the general trends. The simulations presented in Section 3 use only discrete values.

Figure 4.4(a) assumes a constant link width of 32 bits. Thus the dimensionality does not affect the decode delay or the number of flits. As the dimensionality of a given sized network is increased, the mean number of hops that a packet traverses decreases, but wire lengths increase. The constant link width assumption may be valid for small networks, but clearly becomes unrealistic for large networks. *UnitDelay* is not affected by wire length increases, so it always performs best with the highest dimensionality (a hypercube) due to the reduction in packet hops. *Pipelined* is affected by the mean total distance traversed by a packet, which decreases as dimensionality increases, so it also performs best with the highest dimensionality. *NonPipelined* is strongly affected by the maximum wire length, and thus has a much lower optimal dimensionality.

Figure 4.4(b) assumes a constant node size of 192 wires (W=32 for n=3). The optimal dimensionality for all three networks is lower under this constraint than under the constant link width constraint. This is due to the effect of decreasing the link width as dimensionality is increased. The radix for *UnitDelay* still remains constant as N increases, but the optimal radix is now 8 instead of 2, and the optimal dimensionality is correspondingly lower. There is now a difference between *UnitDelay* and *Pipelined*, because *Pipelined* does *not* ignore wire lengths, whereas *UnitDelay* does. For large networks, wire delay becomes more significant, and the optimal dimensionality for *Pipelined* becomes greater than that for *UnitDelay* in order to take advantage of the decrease in mean total distance. This has the corresponding effect of decreasing the optimal radix for *Pipelined* as network size increases. As before, the impact of the maximum wire length on the cycle time of the *NonPipelined* network keeps the optimal dimensionality for *NonPipelined* quite low. This has the effect of making the optimal radix grow substantially for large network sizes.

Figure 4.4(c) assumes a constant bisection, normalized to a hypercube with W=2. The tighter wiring constraint keeps the optimal dimensionality for all networks lower than in parts (a) and (b). For small system sizes, the link width is the dominant factor. As system size increases, the number of hops becomes more important, and the optimal radix decreases. For larger system sizes, wire length begins to dominate, and the optimal dimensionality for *NonPipelined* flattens off, causing the optimal radix to increase. *UnitDelay* and *Pipelined* are not affected by the longer wire lengths, and hence have a higher optimal dimensionality than does *NonPipelined.* The wiring constraint, however, keeps the optimal dimensionality low, and there is thus very little difference between *UnitDelay* and *Pipelined.* Note that while the optimal dimensionalities of *UnitDelay* and *Pipelined* are very close, *UnitDelay* does not accurately characterize the latency provided by *Pipelined.*

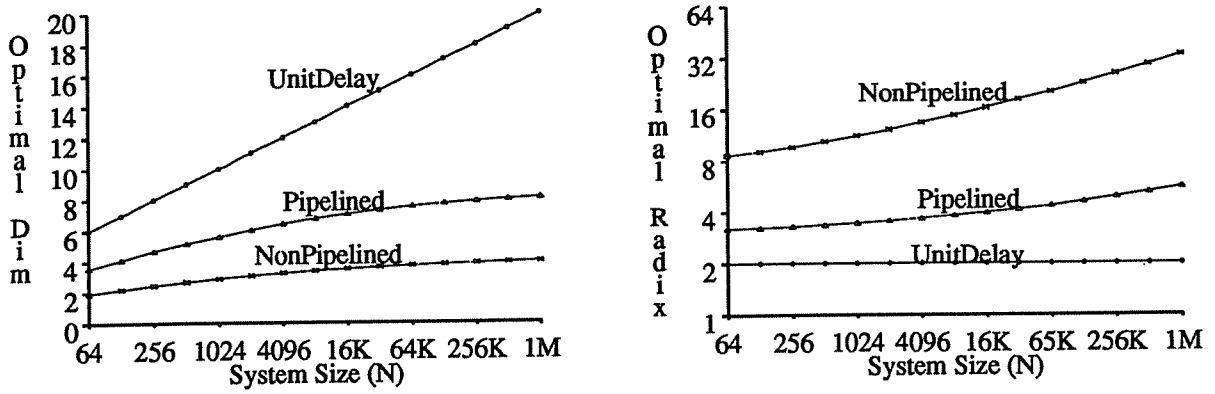
The striking conclusion to be drawn from Figure 4.4 is that pipelining a network significantly impacts the choice of dimensionality. Pipelining argues for *higher* dimensionality. The radix of the pipelined-channel network should be kept roughly constant as system size is increased. When link width is unconstrained, the hypercube provides superior performance for all system sizes. When link width is constrained by either the constant node size or bisection assumptions, the optimal radix increases. The impact of pipelined channels on the network dimensionality is diminished as the wiring constraint is tightened. For bisection-constrained networks the optimal dimensionality differs by an entire dimension only for very large network sizes. The effect will increase, however, as switching times decrease in relation to wire transmission delays.

Since, for a uni-directional torus, the mean total distance traversed by a packet decreases as dimensionality increases, the optimal radix of a pipelined-channel network decreases somewhat for larger systems. As was discussed in Section 2.6, however, this is not the case for bi-directional networks. In a bi-directional torus, the mean total distance traversed by a packet is independent of dimensionality. In a bi-directional mesh, the mean total distance traversed by a packet increases with dimensionality. This changes the relationship between *UnitDelay*, which ignores wire delay, and *Pipelined*, which does not.

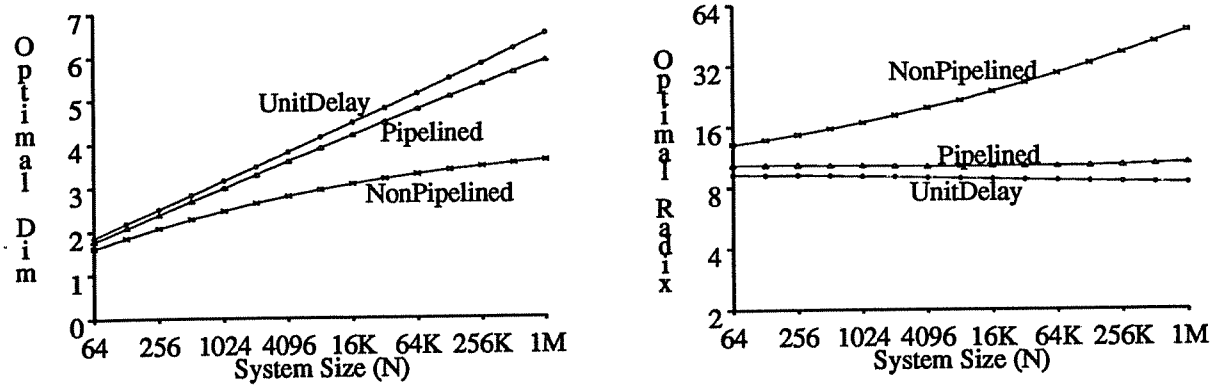
Figure 4.5 plots the optimal dimensionality and radix of a bi-directional mesh as network size grows. As in Figure 4.4, results are given under the constant link width, node size, and bisection constraints. Results are quite similar to those in Figure 4.4, save for the behavior of *Pipelined*. Since the mean total distance traversed by a packet increases with dimensionality for a mesh, the optimal dimensionality for *Pipelined* is lower than that for *UnitDelay*.

Under the constant link width constraint (Figure 4.5(a)), wire delay strongly limits the optimal dimensionality for *Pipelined*, raising the optimal radix to about 4 rather than 2. Under the constant node size and bisection constraints (Figures 4.5(b) and (c)), the wiring constraints limit the dimensionality of both *UnitDelay* and *Pipelined*, which lessens the difference between them.

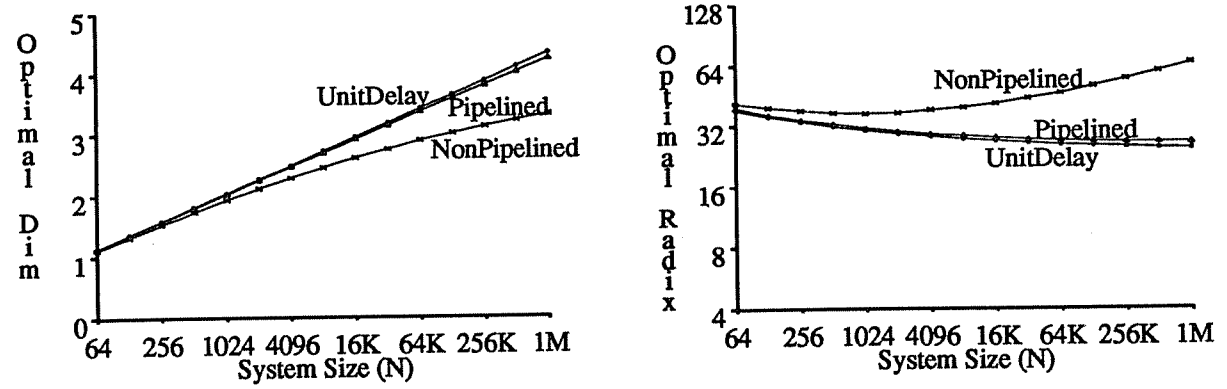
The effect of pipelined channels is qualitatively the same for uni-directional tori and bi-directional meshes: by decoupling throughput from latency, pipelined channels argue for higher dimensionality and for growing networks by keeping the radix roughly constant. The mean total distance traversed by a packet is the primary distinction between the two network types. In uni-directional tori, the mean total distance is smaller for higher dimensionality, so the optimal radix decreases as network size increases. In meshes, backtracking causes the mean total distance to increase with dimensionality, so the optimal radix may increase as network size increases (this



(a) Constant link width (W=32)



(b) Constant node size ($2nW$=192)



(c) Constant bisection width (W=2 for hypercube)

Figure 4.5: Optimal dimensionality and radix versus network size (bi-directional mesh)

effect is muted in Figure 4.5(c) by the short wire lengths caused by the strong wiring constraint).

The analysis thus far has considered only unloaded latency. When bandwidth is considered, the argument for keeping the radix small becomes more compelling. Under uniform traffic, the

rate of traffic across a link is proportional to the radix of the network [Scot91]. Thus, if the network is grown by increasing the radix, the rate of traffic per link will also increase. Although this will be offset by larger link widths when wiring is constrained, the total throughput per processor still argues for lower radix. Section 3 explores bandwidth considerations in more detail.

3. Throughput and Contention

Real networks carry traffic, and traffic causes contention resulting in additional delays. Thus, to obtain a clear picture of a network's performance, latency must be viewed in conjunction with throughput. Latency rises significantly as the rate of traffic approaches the network's maximum capacity. The maximum capacity is affected by the link width and dimensionality of the network. In this section, the maximum throughput of a network is derived, and a simulation study of pipelined-channel and non-pipelined-channel networks is presented. All results are for uni-directional torus networks.

The maximum throughput is determined by the packet length in bits, L, the number of wires traversed by the packet, W_{msg}, and the total number of wires in the network, W_{total}. The maximum throughput, in bits per cycle per processor, is L times the maximum message issue rate:

$$X_{pipe_{max}} = L \left\lfloor \frac{W_{total}}{N W_{msg}} \right\rfloor \tag{4.15}$$

The number of wires in the network is given by

$$W_{total} = N n W \tag{4.16}$$

To derive L and W_{msg}, I assume the use of two packet packet formats: address only (L_{addr} bits) and address plus data (L_{data} bits). A fraction f_{data} of the packets generated are data packets, with the remainder being address only. The ring acknowledgement packets described in Section 2.1 are L_{ack} bits. The mean packet length in bits is

$$L = f_{data} L_{data} + f_{addr} L_{addr} \tag{4.17}$$

When a packet traverses a ring, it travels part way as a data or address packet, and the remainder of the way as an acknowledgement packet. Taking this into account, and the fact that the packet length must be rounded up to the nearest multiple of W, the total number of wires traversed by a packet is given by

$$W_{msg} = W\left\lceil \frac{f_{data}\left\lceil \frac{L_{data}}{W} \right\rceil + f_{addr}\left\lceil \frac{L_{addr}}{W} \right\rceil + \left\lceil \frac{L_{ack}}{W} \right\rceil}{2} \right\rceil n(k-1) \tag{4.18}$$

Substituting Equations (4.16), (4.17) and (4.18) into Equation (4.15) yields

$$X_{pipe_{max}} = \frac{2(f_{data}\,L_{data} + f_{addr}\,L_{addr})}{(k-1)\left(f_{data}\left\lceil \frac{L_{data}}{W} \right\rceil + f_{addr}\left\lceil \frac{L_{addr}}{W} \right\rceil + \left\lceil \frac{L_{ack}}{W} \right\rceil\right)} \tag{4.19}$$

If we assume that packet lengths are an integral number of flits, and ignore differences in packet lengths, then this equation simplifies to

$$X_{pipe_{max}} = \frac{W}{(k-1)} \tag{4.20}$$

Assuming that packet lengths are an integral number of flits, Equation (4.19) shows that the maximum throughput is proportional and approximately equal to $\frac{W}{(k-1)}$.

For non-pipelined-channel networks, the maximum throughput must be scaled down to reflect the longer cycle time needed to include wire transmission. With the time unit set equal to that in Equation (4.20) (one cycle, not including wire transmission time) to allow direct comparison, the maximum throughput in a non-pipelined-channel network is

$$X_{non-pipe_{max}} = \frac{X_{pipe_{max}}}{1+t_{wire_{max}}} \tag{4.21}$$

For non-pipelined-channel networks, not only does the latency increase as wire lengths grow, but the traffic capacity of the network decreases.

Figure 4.6 shows the effect of the "optimal" configurations shown in Figure 4.4 on network throughput. The curves represent the ratio of maximum throughput in a pipelined-channel network to maximum throughput in a non-pipelined-channel network. Equations (4.20) and (4.21) are used, with k and n chosen according to Figure 4.4 and W determined by the corresponding wiring constraint. The pipelined-channel network provides higher throughput under all circumstances, and the relative advantage becomes greater as system size is increased. This is due both to the smaller cycle time resulting from pipelining and to lower contention resulting from smaller network radices (see Figure 4.4).

The difference in throughput is greatest for unconstrained networks (constant link width), for which the difference in optimal dimensionality is greatest. For bisection constrained

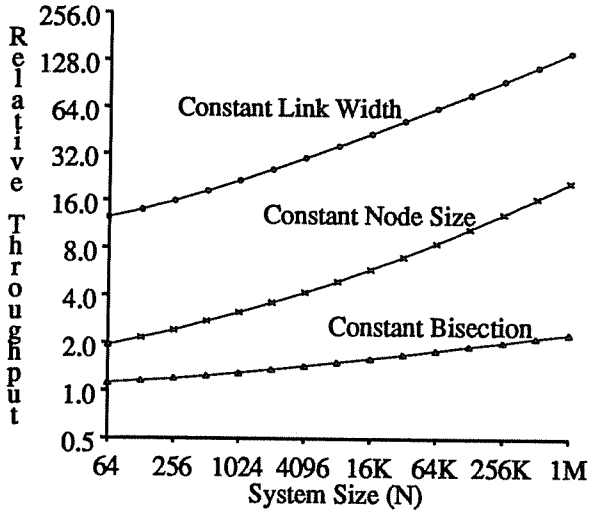


Figure 4.6: Relative maximum throughput (pipelined over non-pipelined) for "optimal" configurations versus network size

networks, the difference in optimal dimensionality is smaller, resulting in a much smaller throughput advantage for pipelined-channel networks. Since integer considerations are being ignored, the wire lengths for networks with $n<3$ are shorter than those for $n=3$, resulting in very little increase in cycle time for the non-pipelined-channel network. If realistic wire lengths were used, the throughput advantage of pipelined-channel networks would be greater. Also, as the ratio of switching to transmission time (S) decreases — as it will do in the future — the throughput advantage will further increase.

3.1. Simulation model

Equations (4.20) and (4.21) give throughput maximums, but do not explain the relationship between throughput and latency. For this, either a queueing model or simulation is necessary. The following section describes a simulator that models the network node shown in Figure 4.2 with great detail and uses an iterative solution technique to efficiently provide performance results for arbitrary network sizes.

The simulation model used in this study is based upon the network described in Section 2.1 and illustrated in Figure 4.2. Parameters to the model include the radix and dimensionality of the network, the link width, the size of the input queues, output queues and ring buffers, the size of the address and data packets, the ratio of address to data packets and the ratio of switch cycle time to base wire delay. Only a single node is simulated. The arrival of packets from the processor and each ring is a Poisson process with rates determined by the system size. Let the packet

arrival rate from the CPU be r, of which a fraction f_{data} are data packets and the remainder address packets. Each packet traverses a mean of $\frac{k-1}{2}$ links in each dimension, and causes an acknowledgment packet to traverse a mean of $\frac{k-1}{2}$ links in each dimension. The base packet arrival rates at each of the n switch input links is thus $(k-1)r$, with $\frac{1}{2}$ the packets being acknowledgements, $\frac{f_{data}}{2}$ of the packets being data packets, and $\frac{1-f_{data}}{2}$ of the packets being address packets. A fraction $\frac{2}{k}$ of all acknowledgement packets received at the switch are destined for this node and can thus be discarded. The remaining acknowledgement packets must be passed along the same dimension. A fraction $\frac{2}{k}$ of all data and address packets are also destined for this node, with the remainder to be passed along the same dimension. The address and data packets destined for this node are either switched to another dimension or routed to the CPU. A packet arriving from dimension i is switched to dimension $i+j$ with probability $\left\lceil \frac{k-1}{k^j} \right\rceil$. The packet is switched to the CPU with probability $\left\lceil \frac{1}{k} \right\rceil^{n-i}$.

If all packets were accepted by the switch, then the simulator could use the above arrival rates and routing probabilities to simulate the entire network. The time to pass through the switch in each dimension, and the time to switch into each dimension would be calculated, and these could be used to construct the total latency through the network for the given CPU request rate. However, some packets may not be accepted, causing negative acknowledgements to be returned, which in turn cause the packets to be retransmitted. This not only increases the latency through the network (because a packet may have to be sent around a ring more than once), but increases the traffic as well.

The relationship between the fraction of packets that are dropped (f), the base traffic (B), and the extra traffic generated as a result of dropped packets (E) is as follows:

$$(B+E)f = E \qquad \Rightarrow \qquad (B+E) = \left\lceil \frac{B}{1-f} \right\rceil \tag{22}$$

The simulator uses an iterative technique to arrive at a solution for the above equations. The initial arrival rates are set according to the system size and the processor request rate as described above. The system is simulated and the fraction of dropped packets is calculated for each dimension for both packet types (address and data). These are used to calculate new arrival rates for

each dimension and packet type, using the rightmost formula in Equation (4.22). The base rates are calculated used the *realized* throughput of CPU requests from the previous iteration. The arrival rates of acknowledgment packets are simply the base rates, but the arrival rates of data and address packets increase (provided the fraction of dropped packets is nonzero). The arrival rates of negative acknowledgments are equal to the sums of the increased arrival rates for address and data packets.

In order to calculate latency, the following quantities are measured for both address and data packets.

λ_i = fraction of packets dropped in dimension i

α_i = T_{pass} for dimension i (includes delays in ring buffers)

β_i = T_{switch} for packets *leaving* dimension i (includes delays in switch queues)

γ_i = mean time between accepting a packet and transmitting the corresponding acknowledgement packet

δ_i = mean time between accepting a negative acknowledgement packet and retransmitting the corresponding packet

The corresponding latency is

$$L = P - 1 + \beta_0 + \sum_{i=1}^{n} \left\lceil \frac{k-1}{k} \right\rceil \left[\frac{k}{2}(T_{wire_{avg}} + T_{decode}) + \left\lceil \frac{k-2}{2} \right\rceil \alpha_i + \beta_i + \left\lceil \frac{\lambda_i}{1-\lambda_i} \right\rceil \left[k(T_{wire_{avg}} + T_{decode}) + (k-2)\alpha_i + \gamma_i + \delta_i \right] \right] \quad (4.23)$$

The throughput, address packet latency and data packet latency are calculated after each iteration, and the iterations continue until the relative change in the results is below 2%. Smaller thresholds are impractical due to the inherent randomness in the simulation output (simulation error becomes dominant within a few iterations). The 90% confidence intervals for the results, derived using the method of batch means, are mostly under 1%, and occasionally as high as 5%.

3.2. Simulation results

Tables 4.1 through 4.3 and Figures 4.7 through 4.9 present system configurations and simulation results for various system sizes, constraints and network types. Parameter values are L_{addr}=128 bits, L_{data}=640 bits, L_{ack}=64 bits, S=1, f_{data}=0.4 and all queue and buffer sizes are 100 bytes.

Table 4.1 (a-c) lists configurations for a 4096-node, pipelined-channel network. The dimensionality of the network is varied under the constant link width, node size and bisection constraints. Each table includes the link width, node size, and bisection for each configuration, and holds one of the three quantities constant according to the appropriate constraint. Decode delay cycles per hop (determined by the link width), delay across the longest wire and the mean wire delay are also shown. Finally, total unloaded network latency and maximum throughput (in bits per cycle per processor) are shown. The values given represent actual system configurations; all discrete quantities are indeed discrete. The mean wire delay is calculated by rounding up wire delays for each link before averaging (note S=1). The unloaded latency is for a round trip message consisting of an address packet followed by a data packet.

Dim	Radix	Link Width	Wires per Node	Wires across Bisection	Decode Delay Cycles	Max Wire Delay	Mean Wire Delay	Roundtrip Unloaded Latency	Maximum Throughput
2	64	32	128	4096	1	1	1.00	407.9	0.852
3	16	32	192	16384	1	1	1.00	166.6	3.578
4	8	32	256	32768	1	2	1.56	132.7	7.668
6	4	32	384	65536	1	4	2.00	107.0	17.892
12	2	32	768	131072	1	4	2.00	86.0	53.677

(a) Constant link width

Dim	Radix	Link Width	Wires per Node	Wires across Bisection	Decode Delay Cycles	Max Wire Delay	Mean Wire Delay	Roundtrip Unloaded Latency	Maximum Throughput
2	64	48	192	6144	1	1	1.00	400.9	1.124
3	16	32	192	16384	1	1	1.00	166.6	3.578
4	8	24	192	24576	1	2	1.56	141.7	5.465
6	4	16	192	32768	1	4	2.00	131.0	8.946
12	2	8	192	32768	2	4	2.00	170.0	13.419

(b) Constant node size

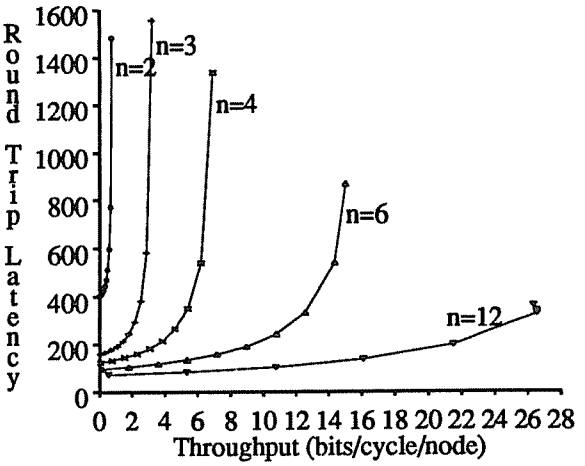
Dim	Radix	Link Width	Wires per Node	Wires across Bisection	Decode Delay Cycles	Max Wire Delay	Mean Wire Delay	Roundtrip Unloaded Latency	Maximum Throughput
2	64	128	512	16384	1	1	1.00	389.9	2.935
3	16	32	192	16384	1	1	1.00	166.6	3.578
4	8	16	128	16384	1	2	1.56	156.7	3.834
6	4	8	96	16384	2	4	2.00	197.0	4.473
12	2	4	96	16384	3	4	2.00	278.0	6.710

(c) Constant bisection

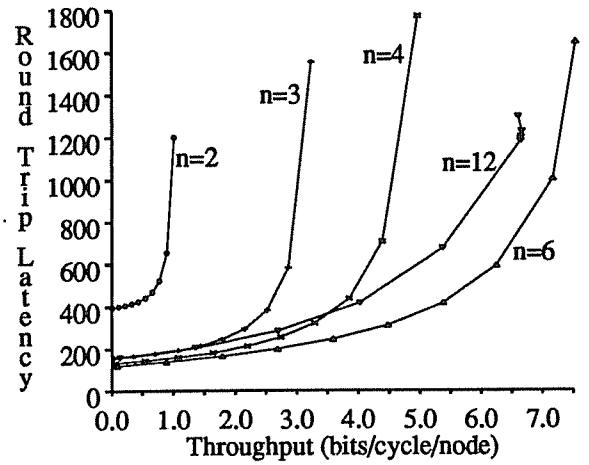
Table 4.1: Actual system configurations (pipelined, N=4096)

Under all constraints, the wire delay increases as the dimensionality is increased. The maximum wire delay increases significantly faster than the mean wire delay. Under the constant node size and bisection constraints the decode delay increases as well. The maximum throughput increases with dimensionality, but at a different rate, depending upon the constraint.

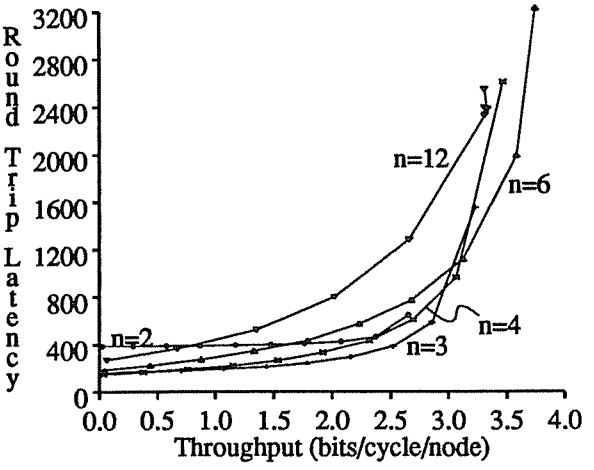
Figure 4.7 presents the simulation results for the networks listed in Table 4.1. The differing throughput capabilities of the networks can be clearly seen. Figure 4.7(a) shows latency versus throughput for networks under the constant link width constraint. The higher-dimensional



(a) Constant link width



(b) Constant node size



(c) Constant bisection width

Figure 4.7: Latency versus throughput (pipelined, N=4096)

networks are clearly superior, providing lower latency for a given throughput and a substantially greater maximum throughput.

In Figure 4.7(b), the constant node size constraint is enforced. The 6-dimensional network is superior in this case. Although there is very little difference between the unloaded latency of the 4- and 6-dimensional networks, the 6-dimensional network performs significantly better under heavy traffic. Similarly, the 3- and 12-dimensional networks have nearly equal unloaded latencies, but the 12-dimensional network performs better as traffic increases. This illustrates why unloaded latency alone is inadequate to characterize the performance of a network. It is interesting to note that the 6-dimensional network achieves higher throughput than the 12-dimensional network even though it has a lower maximum throughput. This is because of its larger link width, which allows a more uniform utilization of its links.

Figure 4.7(c) uses the constant bisection constraint. It is less clear here which network provides the best performance. While the 4-dimensional network has the lowest unloaded latency, the 3-dimensional network has slightly lower latency under heavier loads. The higher-dimensional networks are severely penalized by smaller link widths and greater decode delays.

Table 4.2 presents network configurations for pipelined-channel, 1M-node networks. Several observations can be made in comparing the values to those for 4096-node networks. One is that the difference between maximum and mean wire delay is now quite significant. The *maximum* wire delay is a full 26 cycles for the hypercube, but the *mean* wire delay is only 8.05 cycles. Because pipelined-channel networks do not have to increase their cycle time to accommodate these long transmission times, and because the maximum wire transmission time does not affect the shorter wires, pipelined-channel networks appear especially promising for very large systems. Another interesting observation is that the difference in maximum throughput between the 2- and 20-dimensional networks is *very* large. It is a factor of 1000 under the constant link width constraint. It is clear, however, that the constant link width constraint is not appropriate for very large network sizes; wires per node and bisection simply vary too greatly. For the constant node size constraint, where the total number of wires in the system is independent of dimensionality, the difference in the maximum throughputs is still a factor of 100. And for bisection constrained network, the difference is over a factor of 6.

Figure 4.8 presents simulation results for the networks listed in Table 4.2. The differences in traffic capacity can be clearly seen. Under the constant link width constraint (Figure 4.8(a)), the hypercube is superior. The low-dimensional networks provide much higher latencies and only a small fraction of the capacity. As stated above, however, extremely large networks will

Dim	Radix	Link Width	Wires per Node	Wires across Bisection	Decode Delay Cycles	Max Wire Delay	Mean Wire Delay	Roundtrip Unloaded Latency	Maximum Throughput
2	1024	32	128	65536	1	1	1.00	6168.0	0.052
3	102	32	192	665856	1	1	1.00	940.9	0.531
4	32	32	256	2097152	1	4	3.16	673.1	1.732
5	16	32	320	4194304	1	7	4.38	513.5	3.578
10	4	32	640	16777216	1	26	8.05	342.5	17.892
20	2	32	1280	33554432	1	26	8.05	247.0	53.677

(a) Constant link width

Dim	Radix	Link Width	Wires per Node	Wires across Bisection	Decode Delay Cycles	Max Wire Delay	Mean Wire Delay	Roundtrip Unloaded Latency	Maximum Throughput
2	1024	40	160	81920	1	1	1.00	6164.0	0.060
3	102	26	156	541008	1	1	1.00	946.9	0.412
4	32	20	160	1310720	1	4	3.16	688.1	1.022
5	16	16	160	2097152	2	7	4.38	612.5	1.789
10	4	8	160	4194304	3	26	8.05	474.5	4.473
20	2	4	160	4194304	5	26	8.05	495.0	6.710

(b) Constant node size

Dim	Radix	Link Width	Wires per Node	Wires across Bisection	Decode Delay Cycles	Max Wire Delay	Mean Wire Delay	Roundtrip Unloaded Latency	Maximum Throughput
2	1024	512	2048	1048576	1	1	1.00	6147.0	0.271
3	102	51	306	1061208	1	1	1.00	932.9	0.732
4	32	16	128	1048576	2	4	3.16	821.1	0.866
5	16	8	80	1048576	3	7	4.38	735.5	0.895
10	4	2	40	1048576	10	26	8.05	972.5	1.118
20	2	1	40	1048576	20	26	8.05	1371.0	1.677

(c) Constant bisection

Table 4.2: Actual system configurations (pipelined, N=1048576)

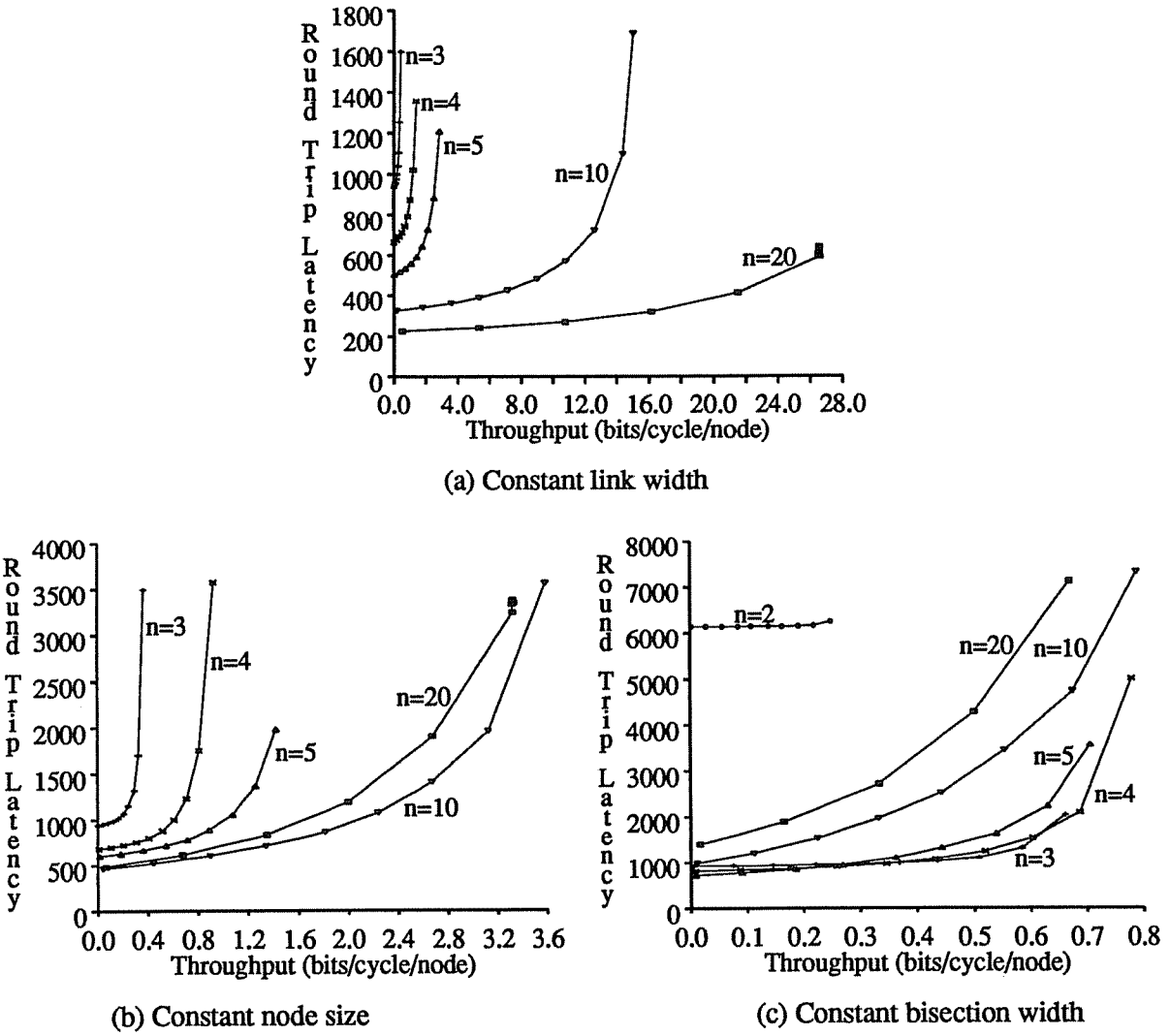


Figure 4.8: Latency versus throughput (pipelined, N=1048576)

not be link width constrained; they will either be node size or bisection constrained. When the node size is constrained (Figure 4.8(b)), the 10-dimensional network provides superior performance, followed by the 20-dimensional network. The 10-dimensional network provides slightly higher throughput even though the 20-dimensional network has a higher maximum throughput. As before, this is due to the larger link width allowing for better link utilization. Under the constant bisection constraint (Figure 4.8(c)), the 4-dimensional network provides superior performance (providing slightly lower latency than the 3-dimensional network under light loads). The 10- and 20-dimensional networks suffer too greatly from the decreased link width and increased decode delays.

In Table 4.3, configurations for a *non*-pipelined-channel 4096-node network are shown. Wire delay is now given as "Cycle Time Increase", the factor by which it increases the cycle time. The unloaded latency and maximum throughput are in terms of the base cycle time (without wire delay), to allow direct comparison with Tables 4.1 and 4.2. The maximum throughput does not increase as fast with dimensionality as it did for the 4096-node pipelined-channel network (see Table 4.1). The unloaded latency also becomes much larger for high dimensions than it did for the pipelined-channel network.

Figure 4.9 presents simulation results for the networks listed in Table 4.3. The results are noticeably different than those in Figure 4.7. Under the constant link width constraint (Figure 4.9(a)), the 12-dimensional network still provides superior performance, but by a smaller margin

Dim	Radix	Link Width	Wires per Node	Wires across Bisection	Cycle Time Increase	Decode Delay Cycles	Roundtrip Unloaded Latency	Maximum Throughput
2	64	32	128	4096	2.00	1	556.0	0.426
3	16	32	192	16384	2.00	1	232.0	1.789
4	8	32	256	32768	3.00	1	246.0	2.556
6	4	32	384	65536	5.00	1	310.0	3.578
12	2	32	768	131072	5.00	1	250.0	10.735

(a) Constant link width

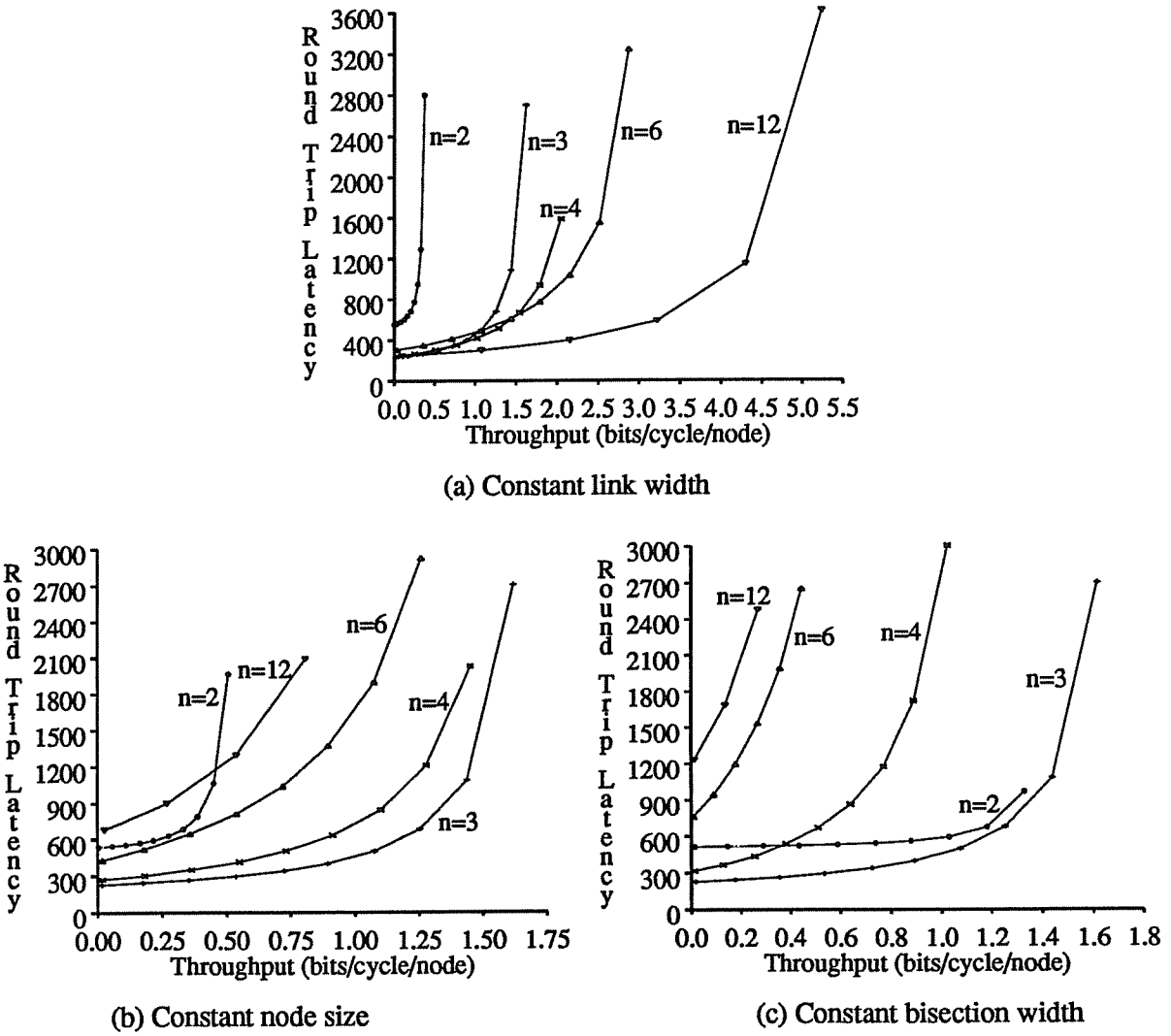
Dim	Radix	Link Width	Wires per Node	Wires across Bisection	Cycle Time Increase	Decode Delay Cycles	Roundtrip Unloaded Latency	Maximum Throughput
2	64	48	192	6144	2.00	1	542.0	0.562
3	16	32	192	16384	2.00	1	232.0	1.789
4	8	24	192	24576	3.00	1	273.0	1.822
6	4	16	192	32768	5.00	1	430.0	1.789
12	2	8	192	32768	5.00	2	670.0	2.684

(b) Constant node size

Dim	Radix	Link Width	Wires per Node	Wires across Bisection	Cycle Time Increase	Decode Delay Cycles	Roundtrip Unloaded Latency	Maximum Throughput
2	64	128	512	16384	2.00	1	520.0	1.467
3	16	32	192	16384	2.00	1	232.0	1.789
4	8	16	128	16384	3.00	1	318.0	1.278
6	4	8	96	16384	5.00	2	760.0	0.895
12	2	4	96	16384	5.00	3	1210.0	1.342

(c) Constant bisection

Table 4.3: Actual system configurations (non-pipelined, N=4096)



(a) Constant link width

(b) Constant node size

(c) Constant bisection width

Figure 4.9: Latency versus throughput (non-pipelined, N=4096)

(note that this would not be the case for a mesh, for which the maximum wire length with 12 dimensions would be twice as long as the maximum wire length with 6 dimensions). The 3- and 4-dimensional networks also provide slightly lower unloaded latency than does the 6-dimensional network. Under the constant node size constraint (Figure 4.9(b)), the 3-dimensional network is now best, with the 4-dimensional network a close second. The 6- and 12-dimensional networks are now very poor choices. Finally, under the constant bisection constraint (Figure 4.9(c)), the 3-dimensional network is clearly superior. The 6- and 12-dimensional networks provide very poor performance due to the greater wire delay and smaller link widths.

Several conclusions can be drawn from the data presented in Section 3. First, it is important to consider bandwidth as well as latency. The unloaded latency does not characterize the performance of a network as traffic increases. Second, pipelined-channel networks favor higher dimensionality than non-pipelined-channel networks do. Pipelining allows the maximum wire length to grow without increasing the cycle time or causing the transmission delay across all links to grow. Finally, pipelining allows for much higher throughput than with non-pipelined-channel networks. This is illustrated by comparing Figures 4.7 and 4.9. The throughputs attainable in the pipelined-channel networks are considerably higher than those in the non-pipelined-channel networks.

4. Other Factors in Network Performance

This section explores two other factors in the performance of pipelined-channel networks. Section 4.1 investigates the effect of varying the ratio of switching to transmission time, S. Section 4.2 investigates the effect of varying the packet length. The results differ somewhat from results obtained previously for non-pipelined-channel networks [Agar91].

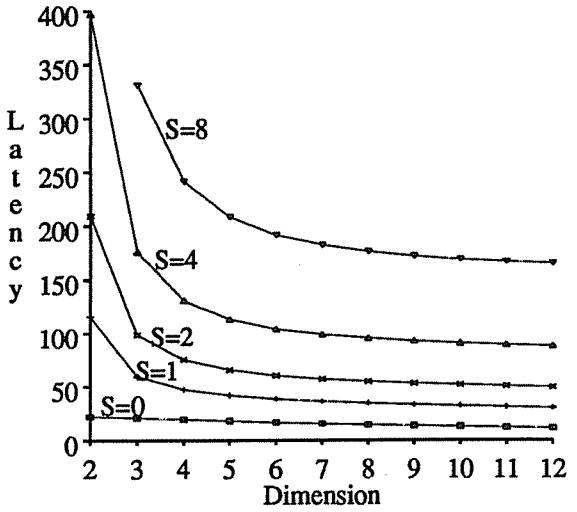
4.1. Effect of switching overhead

Figure 4.10 shows the unloaded latency versus dimensionality for a uni-directional, pipelined-channel, 4096-node network as the ratio of switch cycle time to base wire delay, S, is varied from 0 to 8. Latencies are normalized to the cycle time with $S=1$. As in Figures 4.4 and 4.5, discrete quantities are treated as continuous to better illustrate relationships. Parts (a), (b) and (c) assume the constant link width, constant node size and constant bisection constraints, respectively.

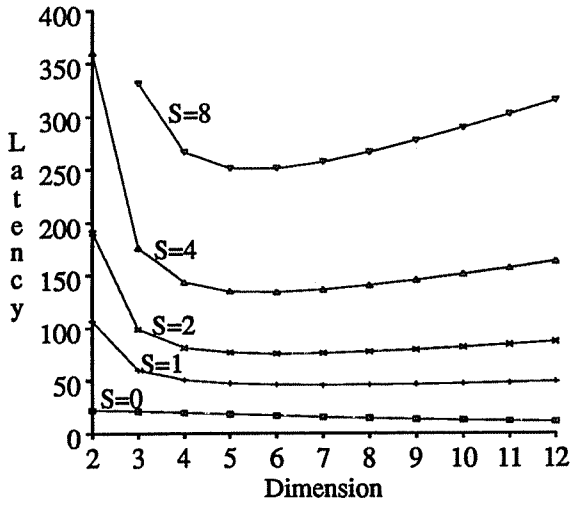
The optimal dimensionality is determined by balancing various components of latency that are affected by the dimensionality. Varying S changes the relative importance of those components that depend upon switching overhead and those that depend upon wire delay.

When switching overhead is ignored ($S=0$), latency depends only upon mean total distance traversed. For uni-directional tori, this assumption favors high dimensional networks and for bi-directional meshes, it favors low dimensional networks. In either case, however, the latency is not very sensitive to dimensionality.

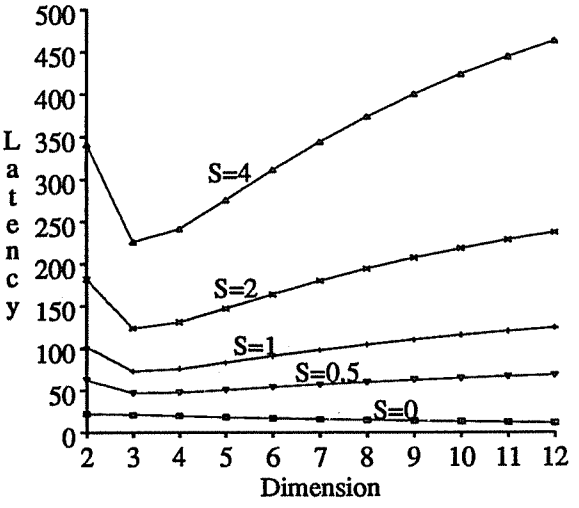
When wire delay is ignored ($S \gg 1$), latency depends only upon the number of switching cycles. For unconstrained networks (Figure 4.10(a)), this assumption favors high dimensional networks. For node-size constrained networks (Figure 4.10(b)), it favors networks with a moderate number of dimensions The optimal dimensionality is derived by balancing narrower



(a) Constant link width



(b) Constant node size



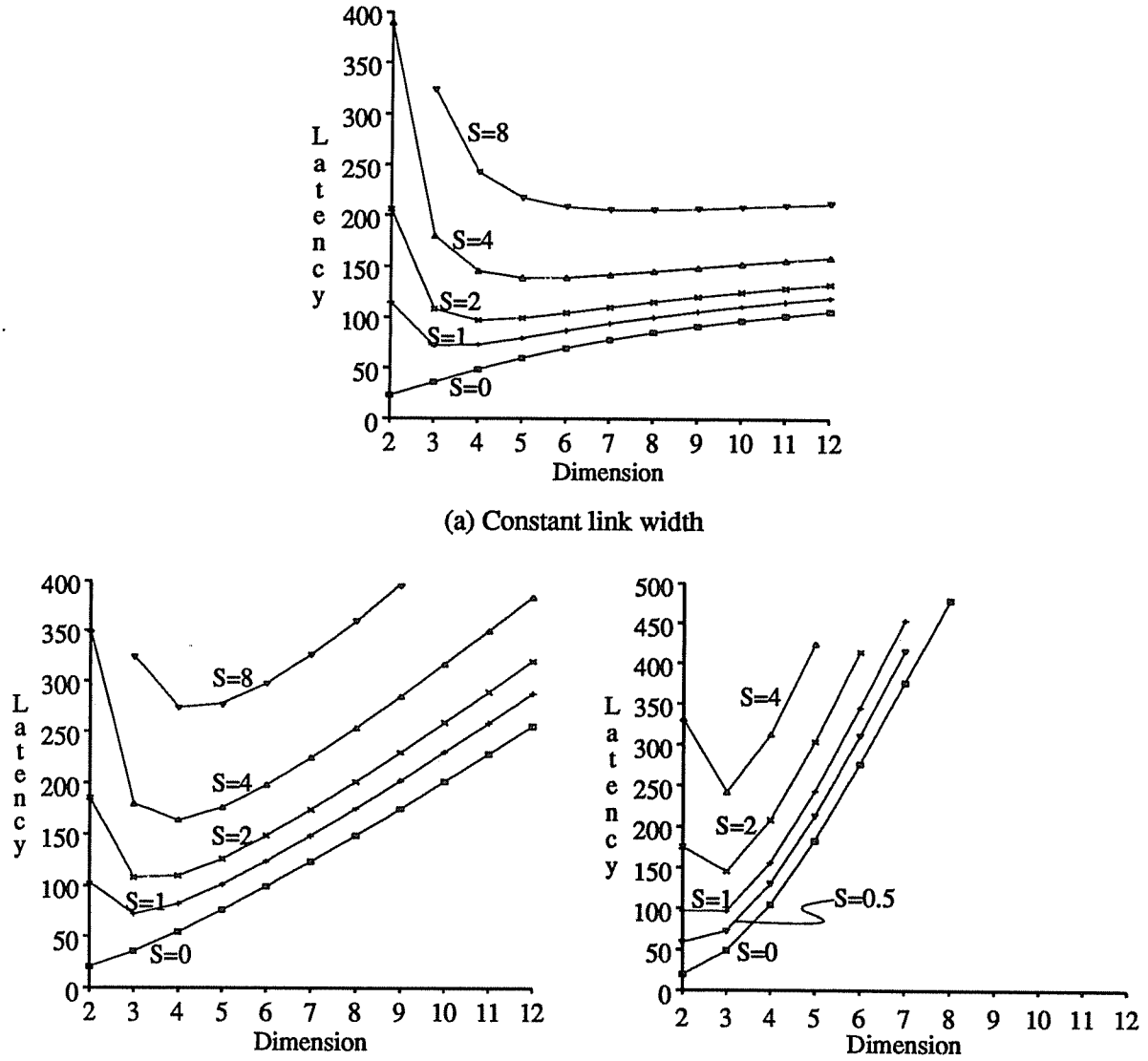
(c) Constant bisection

Figure 4.10: Effect of switch to wire time ratio (pipelined, N=4096)

link widths against decreased hops as the dimensionality is increased. For bisection constrained networks (Figure 4.10(c)), this favors networks with a small number of dimensions, because link widths are very narrow for higher dimensions.

As S is varied, the optimal dimensionality simply shifts between the values for the two extremes. The network latency is more sensitive to dimensionality for larger S.

Figure 4.11 shows the same graphs for *non*-pipelined-channel networks. In non-pipelined-channel networks, when switching overhead is ignored (S=0), low dimensional networks are always superior because of the effect of maximum wire length on the network cycle time. When S is large, higher (how much higher depends upon the wiring constraint) dimensional networks are preferred, due to their smaller number of hops. Therefore, decreasing S will always decrease



(a) Constant link width

(b) Constant node size

(c) Constant bisection

Figure 4.11: Effect of switch to wire time ratio (non-pipelined, N=4096)

the optimal dimensionality by increasing the impact of longer wire lengths [Agar91].

As switching speeds continue to increase, the value of S in real systems will become smaller. This will cause wire transmission delay to become dominant, and the unloaded latency in pipelined-channel networks will become less sensitive to dimensionality. However, *bandwidth* considerations (given uniform traffic) will still favor the use of higher-dimensional networks (see Tables 4.1 and 4.2). At the same time, the optimal dimensionality of non-pipelined-channel networks becomes smaller. Thus, the impact of pipelined channels on dimensionality will become greater. Moreover, decreasing S will enhance the bandwidth advantages of pipelined-channel networks.

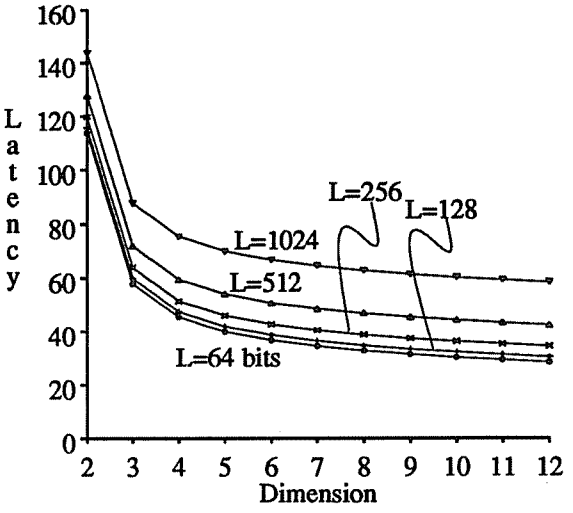
4.2. Effect of packet lengths

Figure 4.12 shows the unloaded latency versus dimensionality for a pipelined-channel, 4096-node network as the packet length, L, is varied from 64 to 1024 bits. Parts (a), (b) and (c) assume the constant link width, constant node size and constant bisection constraints, respectively.

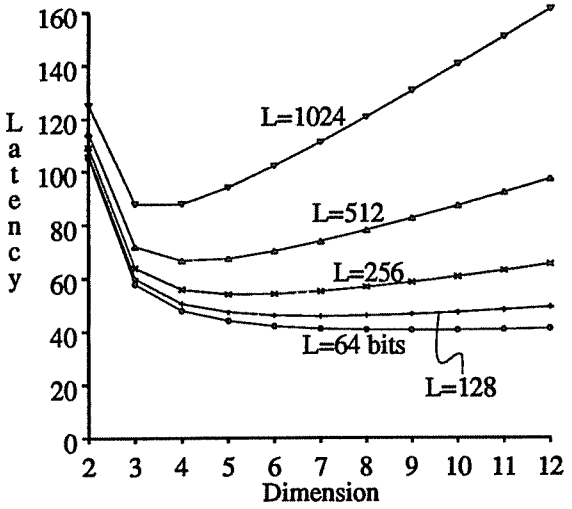
In *non*-pipelined-channel networks, longer packet lengths always decrease the optimal dimensionality [Agar91]. This is because the number of flits, P, becomes dominant over the number of network hops (see Equation (4.5)), making the reduction in number of hops in high-dimensional networks less important relative to the increased wire delay.

A similar phenomenon can be seen in pipelined-channel networks under the constant node size or bisection constraints. Longer packet lengths make the number of flits, P, a greater component of the latency (see Equation (4.4)), and the constraints favor low-dimensional networks, for which the link width is larger. This can be observed in Figures 4.12(b) and (c). Under the constant node size constraint, the optimal dimensionality changes from 3 or 4, when L=1024, to 6 to 12, when L=64. Under the constant bisection constraint, the optimal dimensionality changes from 2, when L=1024, to 4, when L=64.

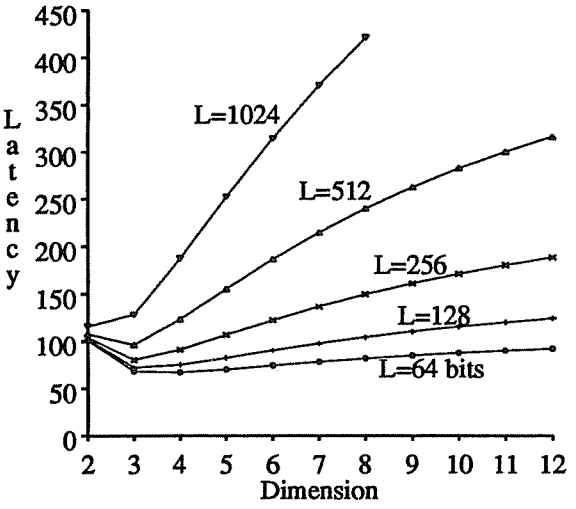
Under the constant link width constraint, however, the packet length does not affect the optimal dimensionality. In this case, P still becomes a greater fraction of the latency for long packets (see Equation (4.4)), but since P is independent of dimensionality and simply added to overall latency, it doesn't change the optimal dimensionality.



(a) Constant link width



(c) Constant node size



(d) Constant bisection

Figure 4.12: Effect of message length (pipelined, N=4096)

5. Conclusions

This chapter has examined the performance impact of pipelined channels on k-ary n-cube networks, and shown that they significantly affect the network design tradeoffs. The key attribute of pipelined-channel networks is that, by allowing multiple bits to be in flight on the same wire, the network cycle time is decoupled from network wire lengths. This removes one of the primary disadvantages of high dimensionality for large networks: increased latency and decreased throughput due to long wires. The result is that higher dimensionality is favored, and that the

networks are more scalable.

The extent to which pipelined channels change the optimal dimensionality of a network depends upon the other disadvantage of high dimensionality: decreased link widths due to wiring constraints. If the wiring is not constrained (the constant link width assumption), then pipelined channels increase the optimal dimensionality dramatically. If the number of wires per node is constrained, then the impact of pipelined channels on optimal dimensionality is smaller, but still very significant. If the network is bisection constrained, then the impact of pipelined channels on optimal dimensionality is relatively small, because the effect of dimensionality on link width keeps the optimal dimensionality low. Even in this case, however, the use of pipelined channels can increase network throughput. As switching speeds continue to increase relative to transmission times, the impact of pipelined channels on network dimensionality will become greater.

The optimal radix of a pipelined-channel network remains roughly constant as system size is increased, regardless of the wiring constraint. When the link width is unconstrained, a hypercube (radix of 2) provides the best performance. Under the constant node size constraint, the optimal radix is between 4 and 10. Under the constant bisection constraint, the optimal radix is in the 16 to 32 range. In uni-directional tori, the mean total distance traversed by a packet decreases slightly with dimension, whereas in bi-directional meshes it increases. These effects cause the optimal radix to decrease or increase, respectively, for large networks. The qualitative impact of pipelined channels, however, holds for both network types.

It is important to consider network throughput as well as latency. Pipelined channels increase the throughput of a network by allowing smaller network cycle times. They further increase effective throughput per processor by allowing higher network dimensionality. Since the traffic per link is proportional to the network radix, high-radix networks have a correspondingly lower traffic capacity. Under the constant node size or bisection constraint this is offset by their larger link widths, but the capacity still decreases as radix increases.

By allowing the network cycle time to be independent of network size, pipelined channels provide a partial solution to the scalability problems of k-ary n-cube networks. When switching delay is dominant, latency grows as the log of the system size (since we grow the network by increasing the dimensionality, not the radix). When wire transmission delay is dominant, latency grows as the cube root of the system size (since the network is embedded in three-dimensional space).

Bandwidth does not scale as easily as latency. Because the radix stays small as system size is increased, the rate or packets traversing each network link is independent of system size (even

though assuming uniform communication). Moreover, the network cycle time remains constant as size is increased. Therefore, if the wiring were not constrained, then the bandwidth of the network would scale perfectly. However, if the network is node size constrained, then the per-processor throughput drops off as the log of the system size. And if the network is bisection constrained, then the per-processor throughput drops off as the cube root of the system size.

There are a few ways in which we can deal with the problem of diminishing throughput. One is to rely on communication locality (a conspirator workload). Another is to simply tolerate the reduced bandwidth for limited scalability. A node size constrained network, for instance, can grow to the square of a given number of processors before the link widths are halved.

Finally, we can throw more hardware at the problem. The constant node size constraint can be ameliorated by using larger packages or by using a cube connected cycle or similar structure to buy us more pins per node. Asymptotically, however, it is clear that networks will be bisection constrained (see the argument in Chapter 2, Section 2); hypercubes, even with bit-serial links, will be impossible to build for sufficiently large systems. To support uniform communication in this case, the network will have to be sparsely populated. If a bisection-constrained network of size $O(r^3)$ is populated with $O(r^2)$ processors, then the communication traffic across the bisection will be proportional to the physical bisection of the network. In this case the cost of the network is proportional to the square root of the number of processors, as is the communication latency.

The theoretical results of this chapter may be difficult to implement in actual systems. It is easier to upgrade a system by increasing the radix than by increasing the dimensionality, as the latter requires changing the structure of a single node. The practical result regarding pipelined channels and dimensionality, then, may be to simply choose a higher dimensionality for the desired range of system sizes than would have been chosen using non-pipelined-channel networks.

Chapter 5

Cache Coherence for Large-Scale Multiprocessors

The most common form of shared-memory multiprocessor today is the *multi* [Bell85], characterized by a shared bus and per-processor *snooping* caches. A shared bus has the natural property that all communications are broadcast, which explains the attractiveness of snooping cache protocols [Good83, Papa84, McCr84, Fran84, Katz85, Enco86, Thac87, Love88]. In these protocols, caches maintain coherence by monitoring all bus traffic and detecting when shared data is modified. As was discussed in Chapter 2, however, a shared bus does not provide scalable bandwidth, and so is unsuitable for multiprocessors with more than a handful of high performance processors. Topologies that provide scalable bandwidth necessarily lack the property that all communications are broadcast, and thus traditional snooping protocols are not feasible.

Directory-based protocols rely instead on coherence information stored in special-purpose directories associated with main memory. While broadcast invalidations can still be used [Arch84], most proposed designs maintain records of cached data, and selectively invalidate cached lines when they are modified. The former approach doesn't scale because of bandwidth constraints, and the latter poses difficulties because of the possibly very large amount of storage necessary to keep track of heavily shared cache lines.

Another opportunity provided by the broadcast nature of a bus occurs when reading shared data. Multiple reads of the same memory line can be combined while waiting to access the bus, allowing for the data to be transmitted over the bus only once. While this ability has found little applicability in single-bus systems, it is inherently appealing in larger systems, where the contention for shared data may be higher. Contention for a particular memory location or module in a large system is known as a *hot spot*, and has been the focus of much research. Hardware mechanisms for combining requests in the network have been proposed [Gott83, Pfis85], as well as mechanisms to improve network performance in the face of contention [Tami88, Scot90, Lang88]. These mechanisms are not necessary under the uniform workload assumption, as the average rate of read requests to a line is independent of system size. However, certain workloads (those using barrier synchronization, for instance) may exhibit significant read contention, in which case it is important that reads not be serialized.

Hardware cache coherence protocols are related to read combining mechanisms for the following reason. If multiple reads for a line are *serviced* concurrently, then the directory must be *updated* concurrently. Thus directories that must be serially updated defeat the purpose of read combining, limiting scalability for workloads with significant read contention. Coherence protocols that serialize invalidation messages to owners of widely shared lines may also limit scalability.[4]

Section 1 of this chapter surveys several proposed coherence mechanisms, with an emphasis on their scalability. Section 2 presents a coherence mechanism, *pruning-cache directories*, that blends the two approaches of broadcasting and directories. Rather than broadcasting invalidations to all processors in a system, invalidations are *multicast* to appropriate subsets of the system using distributed directory information. Section 3 presents a mechanism for hierarchical read combining in a k-ary n-cube, based upon the read combining in multis. The hierarchical nature of pruning-cache directories makes them compatible with this read combining mechanism.

1. Survey of Cache Coherence Mechanisms

Recall that a full-width global directory [Cens78] includes an N-bit vector along with each line of main memory. As discussed in Chapter 2, the full-width directory does not scale in cost, as it requires $O(N^2)$ directory space (assuming a total memory size that grows linearly with the number of processors). There have been many proposals for reduced overhead directories.

1.1. Limited pointer directories

A straight-forward way to maintain a global directory of limited size is to have a fixed number, i, of processor pointers in each directory entry. Either the number of shared copies of a line must be limited to i, or when the number exceeds i, this fact must be recorded and a broadcast invalidate must be issued when the line is modified. Agarwal, Simoni, Hennessy and Horowitz [Agar88] suggested a label of Dir_iB or Dir_iNB to represent the versions of this protocol which do, and do not, use broadcast, respectively. I restrict my attention to Dir_iB, as Dir_iNB does not allow data to be globally shared.

---

[4] Using relaxed memory semantics [Dubo86, Adve90, Ghar90] the latency of most writes (including invalidation of shared copies and collection of the corresponding acknowledgements) can be tolerated. Writes that have not completed at synchronization points, however, can potentially delay program execution.

To keep the hardware costs reasonable, the number of processor pointers that are stored in each directory entry must be small, even for very large systems. Thus, the bandwidth of the system will scale well only if the programs are "well behaved" with respect to sharing. This means that upon invalidation, the number of shared copies of a line must almost always be less than or equal to i, or on the order of N. If some fraction of the invalidations require broadcasts when the number of shared lines is only slightly larger than i, then the bandwidth of the system will not scale, as discussed in Chapter 2. Certain workloads may display the correct behavior for large systems, but it is not clear whether programs will behave correctly in general.

Weber and Gupta [Webe89] analyzed five parallel traces and recorded the distribution of the number of shared copies that needed to be invalidated on a write, for 4, 8 and 16 processor systems. They found that the number of shared lines that needed invalidation on a write was typically low, but that the distribution did spread out as the system size increased. It is difficult to say what the distribution would look like with 100, 1000, or 64K processors.

Agarwal, et al [Agar88], evaluated Dir_1NB and Dir_0B as well as two snooping protocols (Write-Through-With-Invalidate and Dragon) for three 4-way-parallel traces and found Dir_0B to be competitive with the Dragon protocol. Unfortunately, their traces were of limited parallelism and the topology simulated was a single shared bus (where there is little difference between a point-to-point message and a broadcast), so it is difficult to draw conclusions from this work regarding the scalability of the directory protocols.

In more recent work, Chaiken, Fields, Kurihara and Agarwal [Chai90] compared limited directories (without broadcast) to full map directories for several 16- and 64-way parallel codes running on a theoretical multiprocessor with a multi-stage interconnection network. They found that half of their codes performed significantly worse with limited pointer directories than with the full map directory. However, they were able to hand tune two of the codes to eliminate widely read data, bringing the performance close to that of full directories.

One disadvantage of the limited pointer directory scheme is the amount of storage space needed to implement the directory. If the number of pointers per entry can remain fixed or grow very slowly with system size, then the storage overhead will scale well. However, the mean number of shared copies of each line is much less than 1 (it is at most $\frac{C}{M}$, where C and M are the sizes of the cache and main memory, respectively). Therefore, even with only a few pointers per entry, most of the directory space will be unused. Storage overhead is considered in more detail in Chapter 6, Section 6.

1.2. LimitLESS directories

Another way to deal with overflow in a limited pointer directory is to handle it in software. This is done in the *LimitLESS* (Limited directories, Locally Extended through Software Support) cache coherence protocol for the MIT Alewife machine [Chai91]. The LimitLESS protocol maintains a distributed hardware directory with a small number of pointers per cache line. When a read request causes the number of shared copies of a line to exceed the number of hardware pointers provided in the directory entry for that line, an interrupt is issued to the processor local to that directory. The processor then extends the directory entry using local main memory. In this way an arbitrary number of shared copies can be supported.

The most obvious overhead for this scheme is the extra work created for the processor, and the extra time taken to handle directory overflows in software. However, if the probability of overflow can be kept small — and there is reason to think that it could, because most memory lines are not widely shared — then this overhead may be acceptable.

The total directory storage overhead, like that for Dir_iB, will still be much larger than needed to store the necessary sharing information. This is due to the static allocation of hardware pointers to main memory lines. This protocol also does not allow parallel processing of concurrent read requests; all read requests must be processed at the directory entry. This will prevent workloads with any significant read contention from performing well on large systems.

1.3. Coarse vectors

Another way to limit the size of directories is to use *coarse vectors* for the entries [Gupt90]. A coarse vector is similar to a full width directory entry, except that each bit represents a region of two or more processors. When a processor reads a line, the bit for that processor's region is set in the line's directory entry. When a line is invalidated, messages are sent to all processors in the regions whose bits are set. Gupta, et al [Gupt90], suggest that the directories be structured such that an entry can hold one or more processor pointers and then switch over to a coarse vector representation when the pointers overflow. They analyzed the performance of a coarse vector scheme for four programs running on a simulated 32 processor DASH system, and concluded that the coarse vector scheme worked well. However, the directory that they used had regions of only 2 processors each.

The coarse vector scheme is attractive for reducing the directory overhead as system size increases. In addition, since a coarse directory needs only to keep track of regions, it allows some amount of read combining to take place in the interconnect. However, it is clear that the coarse vector scheme just staves off the problem of directory size. As system size increases, either the

size of the directories will increase linearly (as with a full width directory), or the region size will increase, causing performance degradation due to extra invalidation traffic.

1.4. Directory caches

Several researchers have proposed caching directory information to exploit the fact that most lines of memory are not cached by any processor. Weber, Gupta and Mowry [Gupt90] referred to these directory caches as *sparse directories*, and suggested that entries could be either full width, limited pointer or coarse vector, depending on the size of the system and the amount of memory allotted for directory storage. O'Krafka and Newton [O'Kr90] independently proposed the use of directory caches, and suggested that two separate caches be maintained at each node: a small cache of full width entries and a larger cache of limited pointer entries.

Using directory caches provides a large constant-factor reduction in storage overhead. The number of directory entries that are needed is approximately equal to the number of cache lines in the system rather than the number of main memory lines. However, using directory caches does not solve the problem of choosing a size for the directory entries. Any static allocation of pointers to directory entries will cause wasted space by assigning too many pointers per entry and/or performance degradation by assigning too few pointers per entry. The use of multiple caches with different types of entries in each cache can provide a partial solution to this problem. Directory caches do not address the problem of serialization at the directories.

1.5. Sectored directories

O'Krafka and Newton investigated another coherence mechanism called the *sectored directory* [O'Kr90]. This scheme keeps track of the individual *state* of all blocks (called "sub blocks" in the paper), but keeps track of the *sharing* information over sets of n blocks. Each set of n blocks has an associated full width vector that represents the union of the full width sharing vectors for each of the blocks in the set. In this way, the total amount of directory storage is reduced by a factor of n over the Censier and Feautrier scheme.

When a shared block is written, invalidation messages are sent to all processors that have copies of any of the blocks in the set to which to shared block belongs. O'Krafka and Newton found that this scheme led to heavy invalidation traffic and performed worse than a directory cache scheme using a similar amount of storage.

1.6. Dynamic pointer allocation

Simoni and Horowitz [Simo91] proposed a *dynamic pointer allocation* protocol that avoids the problem of statically assigning processor pointers to directory entries. Their scheme maintains a cache of single processor pointers at each node. Each entry in the pointer cache consists of one processor pointer and a link to another entry in the pointer cache. Pointers are allocated out of this pool as necessary to keep track of all shared lines. All unused entries in the pointer cache are chained together in a free list.

Directory entries associated with main memory consist solely of a pointer into the pointer cache. When a memory line is first read, a pointer is taken off the free list in the pointer cache and set to point to the cache that read the line. The main memory directory entry is set to point to the newly allocated pointer cache entry. As additional caches read the line, additional pointer entries are allocated, and a linked list of pointers is built up in the pointer cache. When the line is written, the linked list is traversed, and invalidations are sent to all processors pointed to by the entries. The pointer cache entries are then returned to the free list.

The dynamic pointer allocation protocol adapts to arbitrary distributions of the number of shared copies per line. Each main memory directory entry uses only the necessary number of pointers. Most lines of memory will not be cached anywhere, and thus will not require any processor pointers to be allocated. Each pointer cache needs to have enough entries to keep track of all cached memory lines from the main memory at its node. If the number of cached lines from each main memory module were equally distributed, then the pointer caches would need to have exactly as many entries as there are lines in a cache. However, since memory may be non-uniformly cached, the number of entries per pointer cache should be some constant factor larger than the number of lines per cache. Simoni and Horowitz [Simo91] suggest that directory storage requirements can be further reduced by using a cache for the head links (the pointers into the pointer cache that are associated with each line of main memory).

The dynamic pointer allocation protocol provides the completeness of a full width directory, while requiring only $O(N)$ storage (asymptotically, the storage overhead is $O(N \log N)$, because each pointer cache entry includes a binary number representing a processor). It does not address the problem of serialization at the directories, however. Thus, it is not compatible with read combining mechanisms and would not perform well on large systems for workloads with significant read contention.

1.7. Distributed linked list directories

The cache coherence protocol used in the Scalable Coherent Interface [IEEE92], also uses linked lists to keep track of shared copies of each line, but the lists are physically distributed among the caches of the system. Associated with each cache line in the system are two hardware pointers capable of pointing to other caches. These can be used to build up a distributed, doubly-linked list of caches sharing a particular line of memory. The directory entry associated with a line of main memory points to the head of the associated sharing list if the line is cached. Since the pointers are distributed with their corresponding cache lines, there is never a shortage of pointers, and the storage overhead for pointers scales linearly with the number of cache lines in the system.

The time overhead for building up and breaking down the linked lists in SCI is higher than that in Simoni and Horowitz's scheme, because the lists are physically distributed, requiring multiple messages to be sent over the interconnect when performing list operations. However, the SCI standard is being extended to address the serialization problem, which will allow read requests to be handled concurrently in large systems [Gust92]. Optimizations have been designed that use combining in the network and a third hardware pointer per cache line to construct tree structures rather than flat linked lists. These trees can be used to distribute shared data or invalidations to x processors in $O(\log x)$ sequential messages.

Distributed linked lists are also used in the Stanford Distributed-Directory protocol [Thap90]. This protocol uses singly linked lists rather than doubly linked lists. Although requiring less storage, the Stanford protocol does not provide the same level of robustness as SCI, nor does it allow extensions for concurrent read combining.

1.8. Hierarchical directories

In systems with a hierarchical topology and a multicast capability, a more natural form of concurrent read combining and parallel invalidation can be implemented by partitioning the directory hierarchically. Recall that in a k-ary n-cube, each node defines an embedded tree, with itself at the root (see Figure 2.3(b)). Therefore, while the k-ary n-cube avoids the root bottleneck of a single tree, the coherence protocol with respect to any line of memory can use the tree defined by the home location for the line. For this reason, cube topologies appear to be excellent platforms for hierarchical protocols.

For simplicity however, we can consider a single tree multiprocessor, and the directory entry for a single line. Figure 5.1 depicts such a tree with the processor caches at the bottom. The marked caches represent those that have a copy of the line in question. The interconnect is

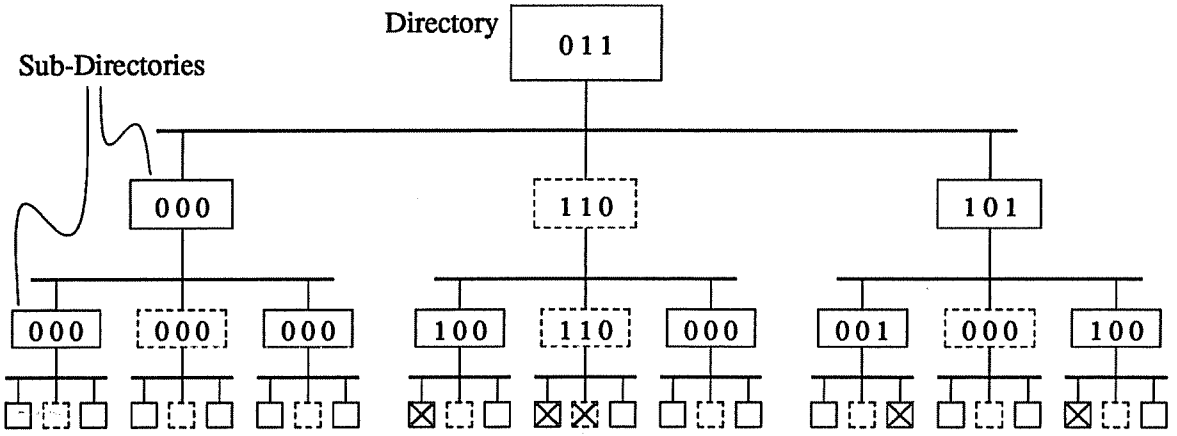


Figure 5.1: Hierarchical directory example

A single tree with a bus interconnect is shown for simplicity. The actual interconnect is assumed to be a cube, implemented with rings for high performance. In a cube-embedded tree (see Figure 2.3(b)), parent nodes are also their own children. Thus the nodes drawn with dotted lines are actually the same nodes as the nodes above them; they are drawn separately simply to aid the conceptual model of a tree. The boxes at the bottom represent data caches, with the cross marks denoting copies of a particular line. Intermediate nodes represent sub-directories with their entries for the given line.

---

depicted as buses, but would likely be implemented with rings for high performance. The directory entry for the line is decomposed hierarchically to fit the tree. The root directory for the line contains a k-bit *pruning vector*. A bit in this vector is set if the corresponding subtree beneath the home location may contain one or more copies of the line. Each of the k nodes along the root ring also contains a k-bit pruning vector for the line, stored in a *sub-directory* at each node. A bit in one of these pruning vectors is set if the corresponding subtree beneath the sub-directory may contain one or more copies of the line. The k^2 nodes at the next level down in the tree also contain k-bit pruning vectors for the line, and so on. Thus, the directory entry for a line consists of a total of $1+k+k^2+\cdots+k^{n-1} = \left\lceil \frac{k^n-1}{k-1} \right\rceil = \left\lceil \frac{N-1}{k-1} \right\rceil$ pruning vectors, stored in as many separate subdirectories.

When a line is invalidated, the invalidate is sent onto the root ring along with the top-level pruning vector, and it is propagated only to those subtrees that may contain a copy of the line. This process is repeated at lower levels by looking up pruning vectors in sub-directories. A total of $n-1$ levels of sub-directory accesses are needed (the lowest level is arguably not needed, as it

only reduces traffic when a copy resides in the parent of a leaf node, but not in any of its children). Hierarchical directories reduce invalidation latency by avoiding serialization of invalidation message issue. They also reduce traffic, as compared to the global (non-hierarchical) directory, because multiple invalidates share part or all of their path through the network.

In addition, read requests can be combined on their way to memory due to the hierarchical structure of the directory. A bit in a directory entry is set when *any* of the processors in the corresponding sub-directory read the line; it is not necessary to know which ones or how many. The read combining works as follows. Assume that the tree is populated with caches at the intermediate nodes, as it would be in a k-ary n-cube-based multiprocessor. As a read request propagates up the tree towards memory, it checks the state of the parent cache at each successive level. If the data is found, it can be returned immediately. If not, a line is allocated for the data in the parent cache and the request continues up toward memory. If it reaches a cache where there is already an outstanding request for the same line, the request may be dropped, and the result decombined when the first request completes (a bit vector denoting the subtrees waiting for the result can be kept in the cache line where combining took place while waiting for the data).

The hierarchical directory, however, increases the complexity of the memory controllers and does not scale in cost. It uses $(N-1)\left\lceil \frac{k}{k-1} \right\rceil$ bits, distributed over several directory structures, for *each* directory entry. Thus, just as the global bit vector scheme, the hierarchical scheme requires $O(N^2)$ storage for the directories.

1.9. Multi-level inclusion

A coherence mechanism that is similar to hierarchical directories in some ways is the *multi-level inclusion* (MLI) property [Lam79, Wils87, Baer88]. The MLI property requires that a cache in a hierarchy contain a superset of all lines residing beneath it in the hierarchy. When the line is invalidated, a parent only propagates the invalidate to its children if it has a copy of the line. This prunes a broadcast invalidate in much the same way as a hierarchical directory. In order to save space, the parent may be required only to have *directory* information about all lines beneath it in the subtree. This can be kept in *inclusion caches*. An inclusion cache simply contains tags for each cache line residing within the subtree rooted at the corresponding node. The logical *inclusion bit* for a line is considered to be set if an entry for that line exists in the inclusion cache, and cleared if there is no entry.

The VMP-MC system[Cher89] enforces multi-level inclusion within a VMP node, but uses directory entries similar to pruning vectors rather than inclusion bits. A pruning vector at a given

node is simply the collection of its childrens' inclusion bits for the same line. The directory entries in VMP-MC are associated directly with memory and cache lines, so MLI is required for the actual data, as opposed to just the tags.

The directory overhead required to enforce MLI does not grow as $O(N^2)$, because inclusion information for a line is not maintained in subtrees that do not contain the line (in Figure 5.1, this would mean that the all-zero entries could be missing). Each of the $O(N)$ lines in the leaf caches requires inclusion information in at most $O(logN)$ parent caches. This overhead is further reduced if data in the leaf caches is shared. Therefore, the total directory space required by MLI grows only as $O(N\ logN)$. Although the directory overhead for MLI scales, it can still be quite large. This is addressed further in the next chapter.

2. Pruning-Cache Directories

A novel way of scaling hierarchical directories is to limit their size and manage them as caches (*pruning caches*) [Good89, Scot91]. We no longer *require* a sub-directory to contain the pruning vector for a given line when it is accessed. If it contains the entry, then we proceed as with the hierarchical directory. If it does not, then we must make the conservative assumption that any subtree may contain a copy of the line (a pruning vector of all ones) and propagate the invalidate to all subtrees. The performance of this mechanism depends upon the hit ratio, h, of the pruning caches. When h=1, the pruning caches act identically to a full hierarchical directory. When h=0, the *top level* pruning vector (stored with memory) acts as a coarse vector [Gupt90]. The invalidation traffic in this case is less than with a broadcast scheme, but still does not scale; invalidation traffic becomes an increasing fraction of total traffic as system size increases.

Figure 5.2 illustrates a pruning-cache-based system. As with Figure 5.1, only a single tree is shown, but the assumption is that this is an embedded tree in a cube network (as in Figure 2.3(b)). The leaf nodes represent processor caches and the internal nodes represent pruning caches. Pruning cache entries for a particular line are shown, and the caches with copies of that line are denoted by cross marks. If the line in question were invalidated, then the invalidate would never reach the pruning caches labeled with an "X", and therefore they do not need an entry for that line. The "Miss" label denotes pruning caches where the invalidation-triggered lookup fails due to a pruning cache miss, causing the invalidation to be broadcast in the subtree beneath the pruning cache. As a result of these pruning cache misses, the underlined processor

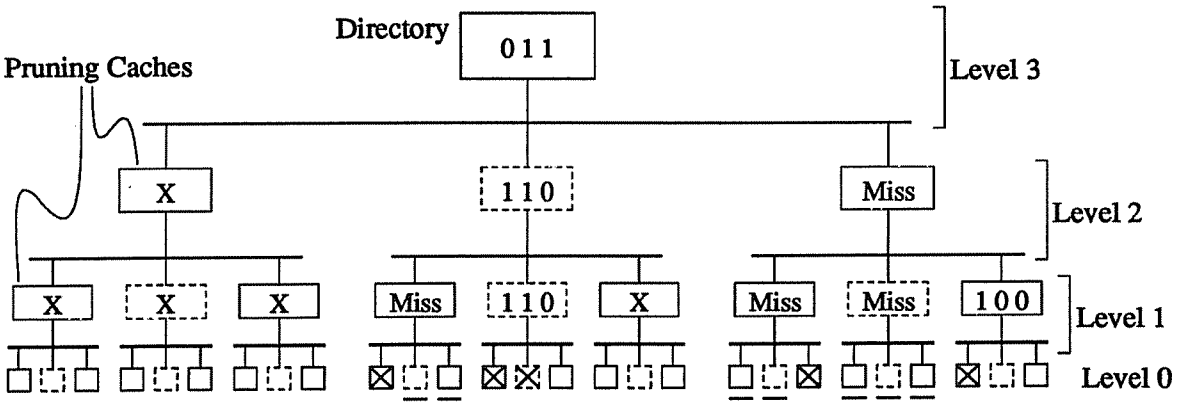


Figure 5.2: A pruning-cache directory example

A single tree with a bus interconnect is shown for simplicity. The actual interconnect is assumed to be a cube (a 3-ary 3-cube in this case), implemented with rings for high performance. In a cube-embedded tree (see Figure 2.3(b)), parent nodes are also their own children. Thus the nodes drawn with dotted lines are actually the same nodes as the nodes above them; they are drawn separately to aid the conceptual model of a tree. The boxes at the bottom represent data caches, with the cross marks denoting copies of a particular line, and the underlines denoting caches that would receive an extraneous invalidation message if the line were invalidated. Intermediate nodes represent pruning caches with their entries for the given line. The X's represent "don't cares" — pruning caches that do not need an entry for the given line.

---

caches receive spurious invalidations.[5]

The key difference between inclusion caches and pruning caches, is that MLI *requires* an inclusion cache to contain entries for all lines beneath it. This means that either an inclusion cache must be built such that all lines that can possibly reside simultaneously in the subtree beneath it can also reside simultaneously in the inclusion cache [Baer88], or when an inclusion cache has to eject an entry, it must invalidate the corresponding line in the subtree beneath it [Wils87]. The first solution is only practical in a single tree system, where the number of caches decreases at each higher level. The second solution, however, is feasible in a cube-based system. Pruning caches are compared to MLI systems via simulation in Chapter 6.

---

[5] The pruning cache hit rate for this example is a very poor 50%. Actual hit rates are expected to be very close to 1.

2.1. Analysis of pruning caches

This section derives expressions for invalidation traffic with and without pruning caches. Pruning caches (partial directories) are compared against two extremes: no directories (requiring broadcasts when a shared line is invalidated) and full hierarchical directories (pruning caches with a hit rate of 1). A similar analysis for bus-based systems can be found elsewhere [Scot91].

In order to analyze the performance of pruning caches, we must first make some assumptions regarding the operation of the system. I will assume two lengths of messages: one for messages containing a line of data, and one for address-only messages. Let these messages require T_{data} and T_{addr} cycles to traverse a link (where T_{data} and T_{addr} are the respective message sizes divided by the link width). Further assume that when a message traverses a ring, the message travels from the source to the target on the ring and an echo packet is returned around the ring from the target to the source. The echo packet is used for fault tolerance and is similar to that used in SCI [IEEE92]. For simplicity, I assume that an echo packet is the same size as an address packet (although in SCI, echo packets are smaller). While the latency to traverse a ring depends upon the distance between the source and target on the ring, the bandwidth used by an address-only message on a ring is always kT_{addr}. A message containing data will use less bandwidth on a ring if the source and target are close, because the message is larger than the echo packet.

Recall that to provide some form of memory consistency, we must know when an invalidation has completed. In a network that can deliver messages out of order, this requires returning acknowledgements from all processors that received the invalidate. In a hierarchical system, acknowledgements from a broadcast or multicast invalidation can be combined at each level: a parent propagates a single acknowledgement up after receiving the acknowledgements from all its children. When a parent places an invalidate onto a leaf ring, it may send up the corresponding acknowledgement as soon as the invalidate has completed its circuit.

With these assumptions in mind, we can now calculate the expected traffic resulting from a broadcast invalidation. The number of rings in a broadcast tree (see Figure 2.3(b)) is $\left\lceil \frac{N-1}{k-1} \right\rceil$, so the traffic from the invalidate packets is $\left\lceil \frac{N-1}{k-1} \right\rceil kT_{addr}$. For all but the root ring in this tree, an acknowledgement packet must be passed up to the next highest ring when the invalidations below the ring have completed. However, for one ring out of k, the acknowledgement packet will have to travel no distance on the next highest ring. Thus the traffic from acknowledgements is $\left(\left\lceil \frac{N-1}{k-1} \right\rceil - 1 \right) k \left\lceil \frac{k-1}{k} \right\rceil T_{addr}$, and the total traffic caused by the invalidation is

$$T_{BC} = \left[\left[\frac{N-1}{k-1}\right] + \left[\frac{N-1}{k-1} - 1\right]\left[\frac{k-1}{k}\right]\right] k\, T_{addr}$$

$$= \left[\left[\frac{N-1}{k-1}\right] + \left[\frac{N}{k}\right] - 1\right] k\, T_{addr} \tag{5.1}$$

Given the hit rate, h, we can calculate the the expected traffic resulting from an invalidation using pruning caches. Assume m shared copies of a line distributed randomly throughout the system, and label the rings as in Figure 2.3(b). A subtree at level i contains the k^i processors that are located at or below the level i ring. The probability that one of the m shared copies resides in a given level i subtree is

$$P_C(i,m) = 1 - \frac{\left[\begin{matrix} k^n - k^i \\ m \end{matrix}\right]}{\left[\begin{matrix} k^n \\ m \end{matrix}\right]} \tag{5.2}$$

If we exclude the k^{i-1} processors in the level i subtree whose path to the root of the tree does not traverse the level i ring, then the probability that one of the m shared copies resides in the level i subtree is equivalent to the probability that an invalidate packet *must* traverse the level i ring on an invalidation, and is given by

$$P'_C(i,m) = 1 - \frac{\left[\begin{matrix} k^n - k^i + k^{i-1} \\ m \end{matrix}\right]}{\left[\begin{matrix} k^n \\ m \end{matrix}\right]} \tag{5.3}$$

The expected traffic from an invalidation (including acknowledgements) using pruning caches with a hit rate of h is now

$$T_{PC} = \left[\sum_{i=1}^{n} k^{n-i} P_{inval}(i,m) k + \sum_{i=1}^{n-1} k^{n-i} P_{inval}(i,m)(k-1)\right] T_{addr} \tag{5.4}$$

where $P_{inval}(i,m)$ is the probability that an invalidate packet traverses a ring at level i.

Calculating $P_{inval}(i,m)$ depends upon our interpretation of h. If we assume that h applies to all pruning-cache references, including those performed by *runaway invalidations* (invalidation that have propagated beyond the necessary subset of processors because of a pruning-cache miss), then we arrive at the following formula:

$$P_{inval}(i,m) = P'_C(i,m) + \sum_{j=i}^{n-2}\Bigg[\Big[P'_C(j+1,m) - P_C(j,m)\Big](1-h)^{j-i+2} + \Big[P_C(j,m) - P'_C(j,m)\Big](1-h)^{j-i+1}\Bigg] + \Big[P_C(n-1,m) - P'_C(n-1,m)\Big](1-h)^{n-i} \quad (5.5)$$

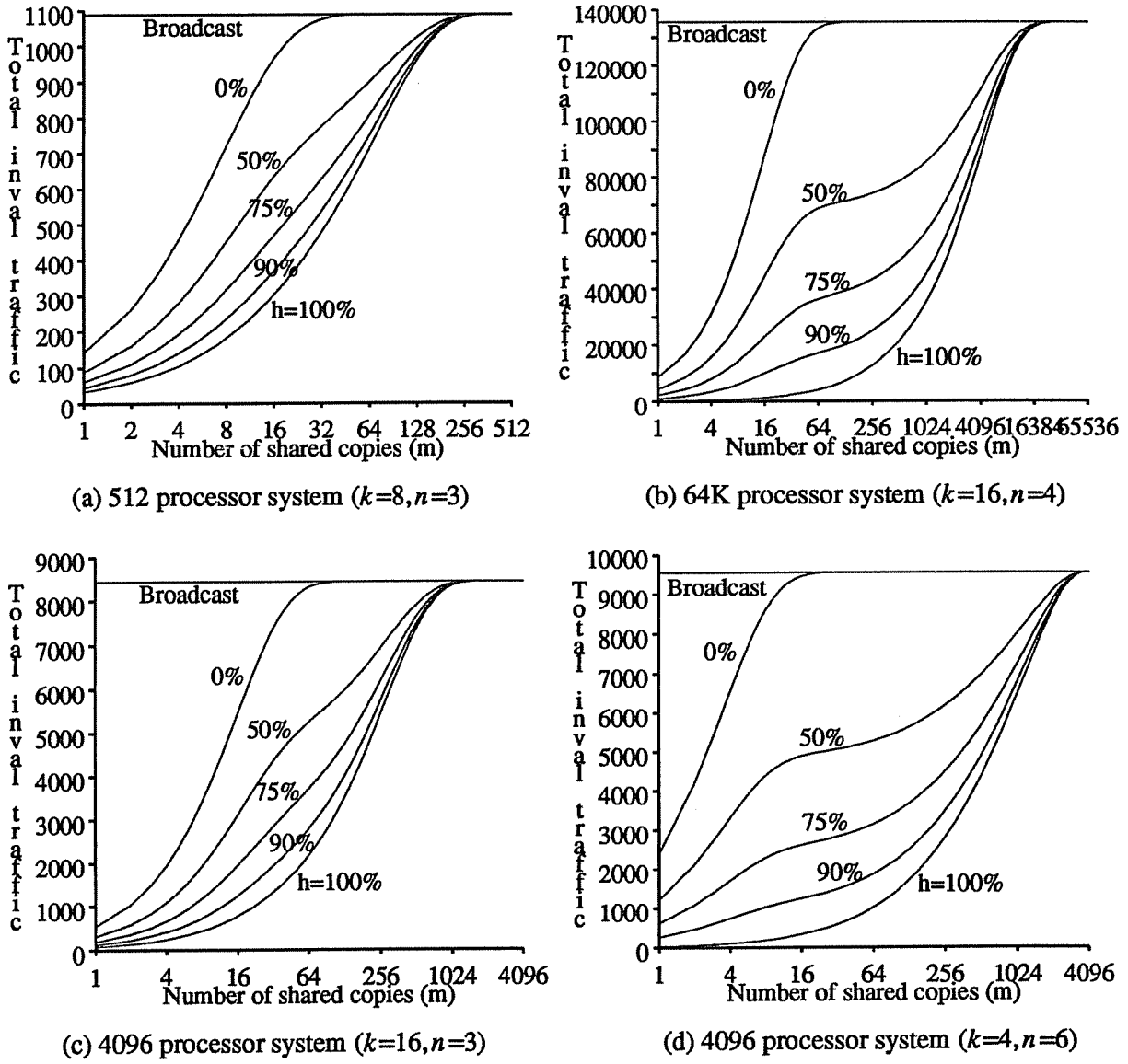
The first term of $P_{inval}(i,m)$ is the probability that an invalidate was *supposed* to traverse the ring at level i, and is the only term that would be nonzero if the hit rate were 1. The remaining terms account for the probability that an invalidate reaches the ring due to pruning-cache misses. The first term inside the summation is due to invalidates that were supposed to traverse the level $j+1$ ring, but were not supposed to reach the level j subtree. The second term inside the summation is due to invalidates that were supposed to reach the level j subtree, but not traverse the level j ring. The last term of $P_{inval}(i,m)$ is equivalent to the second term inside the summation for a value of $j=n-1$. For this value of j, the first term inside the summation does not exist because the top-level directory never misses.

If, on the other hand, we assume that pruning-cache accesses for runaway invalidations always miss, then Equation (5.5) is significantly simplified, and $P_{inval}(i.m)$ is given by

$$P_{inval}(i,m) = \begin{cases} P'_C(i,m) & \text{if } i=n \\ P'_C(i,m) + (1-h)\Big[P_C(n-1,m) - P'_C(i,m)\Big] & \text{if } i<n \end{cases} \quad (5.6)$$

The implication of this assumption is that runaway invalidations are fully broadcast throughout any subtrees into which they are errantly propagated. The choice of formula depends somewhat on the pruning-cache management policy (see Chapter 6, Section 3). Previous published analysis [Scot91] assumed a uniform hit rate for all pruning cache accesses. However, recent simulation results reveal that pruning-cache hit rates are very low for runaway invalidations, so I conservatively assume the use of Equation (5.6) throughout this study.

If $h=1$, then all traffic due to pruning-cache misses goes away, and Equation (5.4) gives us the expected traffic used by a full hierarchical directory. If $h=0$, then the pruning caches are ineffective and traffic is reduced from a full broadcast only by the effect of the pruning vectors in the top level directory. Figure 5.3 plots the expected traffic of an invalidation versus m, for four possible systems (T_{addr} is assumed to be 1). The traffic is shown for a full broadcast (Equation (5.1)) and for pruning-cache systems with various values of h (Equation (5.4)). As indicated in the



(a) 512 processor system (k=8,n=3)

(b) 64K processor system (k=16,n=4)

(c) 4096 processor system (k=16,n=3)

(d) 4096 processor system (k=4,n=6)

Figure 5.3: Pruning-cache performance

figure, pruning caches with a modest hit rate significantly reduce the invalidation traffic.

Statistics gathered from simulations match the curves for h=0% and h=100% very closely. Curves for 0%<h<100% are difficult to confirm because it cannot always be determined whether a particular pruning-cache reference is inside or outside the proper subset of the tree, and thus h cannot be accurately measured. However, simulation results do match approximately. When the pruning-cache hit rate is artificially constrained to a specific value, simulation results match closely. See Chapter 6, Section 4 for these results.

The assumption of a random distribution gives somewhat conservative results, as shown in Figure 5.4. This figure plots the best-case, random, and worst-case traffic caused by a completely pruned invalidation versus the number of shared copies, m, in the system, for a 4096 processor system (k=8, n=4). In the best-case distribution, shared copies are clustered along leaf rings such that higher-dimension rings are traversed as little as possible. This results in traffic of

$$T_{best\ case} = \left[\sum_{i=1}^{n} \left\lceil \frac{m-k^{i-1}}{k^i} \right\rceil + \sum_{i=1}^{n-1} \left(\left\lceil \frac{m}{k^i} \right\rceil - \left\lceil \frac{m}{k^{i+1}} \right\rceil \right) \right] k\, T_{addr} \tag{5.7}$$

In the worst-case distribution, shared copies are spread out among different leaf rings such that higher-dimension rings are traversed as much as possible. This results in traffic of

$$T_{worst\ case} = \left[1 + \sum_{i=1}^{n-1} \left(min(k^i, m) + min((k-1)k^{i-1}, m) \right) \right] k\, T_{addr} \tag{5.8}$$

The random distribution assumed in Figure 5.3 gives traffic estimates very close to the worse case, indicating that there is possibly much to gain by organizing sharing along leaf rings when possible. As an added incentive, sharing along leaf rings would increase the efficiency of pruning caches and read combining in the network. This is validated by simulation results in Chapter 6.

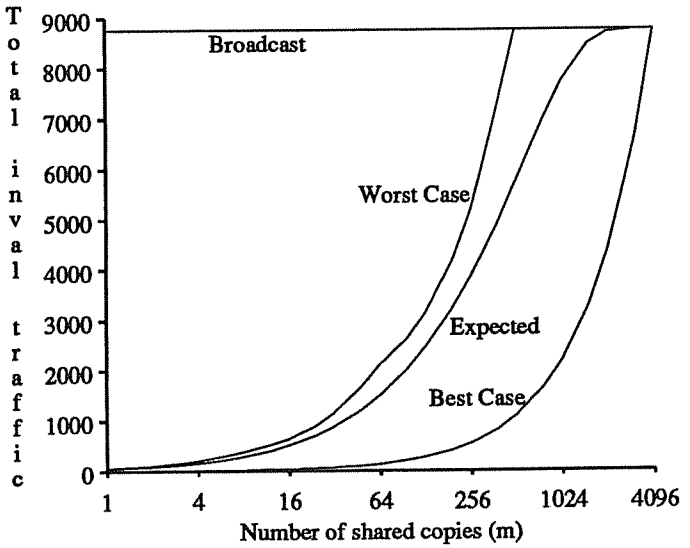
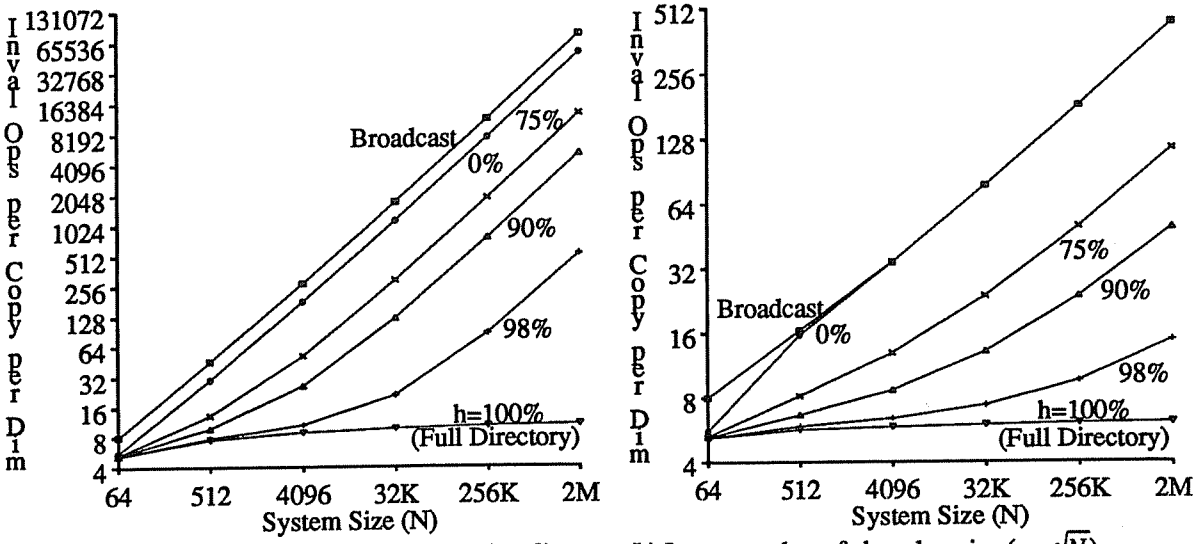


Figure 5.4: Distribution's effect on the traffic of pruned broadcasts
(N=4096,k=8,n=4,h=1)

In order for the system to scale in size without invalidation traffic becoming an increasing fraction of network traffic, the mean invalidation traffic over a link that is caused by each processor making one request (a total of N requests), must be $O(1)$ in system size. *Assertion:* Let I_{norm} be the mean traffic caused by an invalidation, divided by the dimensionality of the system and the mean number of shared copies being invalidated. If I_{norm} is $O(1)$ in the system size, then the mean invalidation traffic over a link that is caused by each processor making one request is also $O(1)$ in the system size. *Proof:* Let T be the mean traffic caused by an invalidation, f be the fraction of processor requests that cause an invalidation and m be the mean number of shared copies of a line when it is invalidated. An average of at least m reads must occur for every write. Thus, f must be less than or equal to $\frac{1}{m}$. The mean invalidation traffic generated by N processor requests is thus less than or equal to $T\left\lceil\frac{N}{m}\right\rceil$. The invalidation traffic is distributed over Nn links. Thus, the mean invalidation traffic per link generated by N processor requests is less than or equal to $\frac{T}{nm} \equiv I_{norm}$. Therefore, if I_{norm} is $O(1)$, then the mean invalidation traffic per link generated by each processor making one request is $O(1)$. □ I will use this normalized invalidation traffic metric, I_{norm}, to illustrate scalability of invalidation operations in Figure 5.5.

For a system that uses a single invalidation message for each shared copy, invalidation of a line requires m messages, each of which requires $O(nk)$ link operations. I_{norm} is $O(k)$ for such a system, and the invalidation traffic scales, provided that the network radix is kept constant. For a system that always performs full broadcast invalidations, however, an invalidation requires $O(N)$ link operations. Invalidation traffic will scale in such a system only if the fraction of processor requests that cause invalidations is $O(n/N)$ or less (*e.g.*: if a large fraction of the processors read each shared line between writes).

Figure 5.5 shows the normalized invalidation traffic (mean traffic of an invalidation divided by the dimensionality of the system and the number of shared copies) for broadcast and pruning-cache-based systems, as system size increases. Part (a) shows the scaling behavior with a fixed, small number of shared copies. Part (b) shows the scaling behavior when the number of shared copies grows as the root of the system size. We see that with a pruning-cache hit rate of 100%, the normalized traffic appears almost constant as system size increases. In fact, this traffic *is* constant asymptotically. When h is less than 100%, however, invalidation traffic is asymptotically unscalable. Thus for sufficiently large systems, broadcasts become too expensive, and MLI will deliver higher performance.



(a) Small number of shared copies (m=8) (b) Large number of shared copies (m=$\sqrt{N}$)

Figure 5.5: Scalability of pruning-cache systems (k=8, n=2....7)

However, for high values of h, invalidation traffic is kept low for system sizes into the thousands. For extremely large systems (hundreds of thousands of processors), even a small pruning-cache miss rate causes a large increase in invalidation traffic. Therefore, pruning caches for very big systems must be large enough to assure very high hit rates. For this reason, the storage overhead of pruning caches is important.

The only entries that need be present in the pruning caches are those for actively shared lines. Entries for private, read-only and passively shared lines may drop out of the pruning caches without affecting performance (in fact, entries for private and read-only lines, if they can be identified, do not have to be placed in pruning caches at all). The following argument shows that the size of a pruning cache needs to grow only as $O(n)$. This results in a total storage requirement of $O(Nn) = O(N \log N)$.

Assume that each processor cache contains a size S set of actively shared data. Because memory is interleaved amongst the memory modules, we can assume that the home memory modules of the shared data are roughly spread out throughout the system. A given pruning cache, therefore, must contain approximately $\frac{S}{k}$ entries for each of the k caches along its level 1 ring (refer to Figure 2.3(b)), $\frac{S}{k^2}$ entries for each of the k^2 caches along its level 2 ring and below, ..., and $\frac{S}{k^{n-1}}$ entries for each of the k^{n-1} caches along its level $n-1$ ring and below. The total

number of entries needed in a pruning cache is thus $(n-1)S$ and pruning caches should retain the same level of performance if they scale in size as $O(n-1)$.

An alternative argument goes as follows. Each cached line requires at most $(n-1)$ pruning cache entries (for its parent's pruning cache, grandparent's pruning cache, *etc.*). The total number of required pruning cache entries is thus $\leq NC(n-1)$, spread out over N pruning caches, which hold equal shares due to symmetry.

2.2. Pruning cache operation

Pruning caches are maintained as read results and invalidates propagate down through their respective trees. When a parent supplies a line to a child in response to a read request (the parent could have either had a copy of the line being requested, or have propagated the request up the tree and just received the result now) it must do two things. First, it looks up the corresponding pruning vector in its own pruning cache and includes it with the line (recall that if the cache misses, an all-one vector is assumed). Second, if the pruning vector was in the cache, the bit corresponding to the child is set.

When the child receives the line, if its bit in the supplied pruning vector is *one*, then it cannot assume anything about the pruning vector for its subtree. If it doesn't already have a copy of the pruning vector in its own pruning cache, then it simply does not create a pruning vector (if it created one, it would have to be all ones, which would be a waste of pruning-cache space). If, when the child receives the line, its bit in the supplied pruning vector is *zero*, then it knows that no cache in its subtree previously had a copy of the line, and it can create an all-zero pruning vector for the line in its pruning cache.[6] If it in turn passes the line down to one of its children, then the appropriate bit in this newly created, all-zero vector would be set.

The top-level pruning vector for a line is kept in the directory with main memory, and thus is always present. On an invalidation, memory clears its pruning vector, as do all pruning caches that the invalidate passes through. A reasonable way to implement acknowledgement combining is to use the pruning vectors used for the invalidation. The pruning vector used to distribute an invalidate (whether retrieved on a pruning cache hit, or assumed all-ones on a pruning cache miss) indicates which children will be returning an acknowledgement. It would be saved when the invalidate is passed down, and as acknowledgements propagated up from the node's children,

---

[6] In fact, it always creates *this* type of all-zero vector (provided it's not a leaf pruning cache), regardless of the policy towards all-zero vectors that will be discussed later.

the corresponding bits would be cleared. When the vector became zero, an acknowledgement would be propagated up to the next level.

The pruning vectors that are saved for acknowledgement combining during an invalidation must be kept somewhere. Either they can be kept in an auxiliary buffer, or they could be pegged down in the pruning caches during the invalidation. The latter choice has the disadvantage that a pruning cache miss for a non-leaf node results in forcing an entry for the line being invalidated into the pruning cache. In either case, the vectors must be retained until the invalidation has completed. Whenever multiple resources must be locked in a distributed environment, deadlock is a concern. This issue is briefly discussed in Chapter 6, Section 5.

There are several important issues related to pruning-cache management that are explored in Chapter 6, Section 3. These include the placement policy, replacement policy and the way in which all-zero vectors are handled. Unless otherwise specified, simulations presented in this thesis use the LRU policy for pruning cache replacements and do not place all-zero pruning vectors into the pruning caches if they require displacing a non-zero entry. The placement policy is discussed in Chapter 6.

3. Read Combining in Cube Networks

Contention for data and synchronization objects is a potential problem that must be addressed when designing very large systems. Synchronization can be handled in a variety of manners, both in hardware and in software. Examples include software combining [Yew87], software queueing [Mell91] and hardware queueing [Good89a]. Although synchronization issues are important, I do not address them in this thesis. I do address the problem of read contention, however, as it is closely related to the cache coherence mechanism. There are several situations in which concurrent read requests to the same line are likely — after a widely shared data object is invalidated, for example, or after a barrier synchronization, as processors read data or code for a new iteration. If these read requests are serialized at the memory, then latencies may become extremely long in large systems. The hierarchical structure of the cube topology and pruning caches, however, allows concurrent read requests to be combined in the network.

In a system with read combining, reads to private data proceed as normal. Reads to shared data access the data caches along their route to memory. Only the parent caches are checked (those at nodes where the path changes dimensions). If the data is found in a parent cache, the data can be returned immediately, otherwise a shadow line is allocated in the parent cache, and the request continues towards memory. If another read request for the same line arrives at the same parent cache before the data arrives, this request can "combine" and be dropped. A bit

vector indicating which subtrees are awaiting the data (called a *pending vector*) is kept in the shadow line and used to distribute the data when it arrives. If more than one subtree is waiting for the data when it arrives, the data is *multicast* on the ring to each of the waiting subtrees. This simply means that the data is sent around the ring along with its pending vector, and each processor whose bit is set retains a copy. When the last processor has taken its copy, the ring echo is sent around the remainder of the ring.

A possible disadvantage of such a combining mechanism is cache pollution caused by parents allocating lines in their cache for their children's reads. Thus, it might be beneficial to keep an auxiliary structure (called a *pending cache*) to store pending vectors. This cache could be quite small, as it would only contain entries for currently outstanding read requests to shared lines. If a pending vector were lost (from either a pending cache or a data cache) before the corresponding data arrived, then the worst-case assumption that all children had requested a copy would have to be made. Thus the data would be broadcast below the processor where the pending vector was lost.

This section derives traffic and latency expressions for concurrent read requests. The latency expressions are optimistic in that they assume no other traffic in the network, but the purpose of the expressions is to evaluate the relative performance with and without combining, so the absolute latencies are less important. Without read combining, the expected traffic resulting from m concurrent read requests is simply m times the expected traffic resulting from one read request, and is given by

$$T_{reg} = m\left[n(k-1)T_{addr} + n\left\lceil\frac{k-1}{2}\right\rceil(T_{addr}+T_{data})\right] \tag{5.9}$$

The expected latency, assuming no other traffic in the network, is dominated by the contention for the memory module or link entering the memory module, whichever is slower. This latency is

$$L_{reg} = E_{min\_links}(m) + T_{addr} + (m-1)max(T_{addr}, L_{mem}) + L_{mem} + n\left\lceil\frac{k-1}{2}\right\rceil + T_{data} \tag{5.10}$$

where L_{mem} is the memory access time and $E_{min\_links}(m)$ is the expected minimum number of links between the memory module and the nearest participating reader (see Appendix B). The traffic caused by the concurrent reads does not present a scalability problem, but the latency may.

With hierarchical read combining, the traffic and latency of m concurrent reads are both significantly reduced when m is large. To calculate the traffic, we must first make some assumptions regarding combining on the root ring. On all lower levels, I assume that the read requests

arrive from below before the data from the first read request arrives from above (*ie:* full combining). Combining can take place on the root, as well, if multiple requests arrive at the memory module before it has retrieved the data for the first request. I will make the conservative assumption, however, that no combining takes place on the root ring. That is, that up to k requests (from the k subtrees at level $n-1$) arrive at the memory and are serviced sequentially. The traffic from the requests is now

$$T_{comb} = \sum_{i=1}^{n}\left[k^{n-i}T_{addr}(k-1)P_C(i-1,m)k\right] +$$

$$\sum_{i=1}^{n-1}\left[k^{n-i}T_{MC}(P'_C(i,m),(k-1)P_C(i-1,m))\right] +$$

$$(k-1)T_{MC}(Pc(i-1,m),Pc(i-1,m)) \tag{5.11}$$

The first term corresponds to the requests propagating up in the tree. The second term corresponds to the data packets being transmitted on the root ring. The last term corresponds to data packets transmitted on lower levels. $T_{MC}(p,c)$ is the expected traffic from a multicast of a data packet on a single ring, where p is the probability that the multicast will have to be transmitted on the ring at all, and c is the expected number of recipients of the multicast that reside in the $(k-1)$ non-local (different than the processor initiating the multicast) processors on the ring. $T_{MC}(p,c)$ includes the traffic from the data packet as well as the ring echo, and is given by

$$T_{MC}(p,c) = pk\left[T_{addr}\left(\frac{1}{1+c/p}\right) + T_{data}\left(1-\frac{1}{1+c/p}\right)\right] \tag{5.12}$$

The latency of m concurrent read requests using read combining is given by

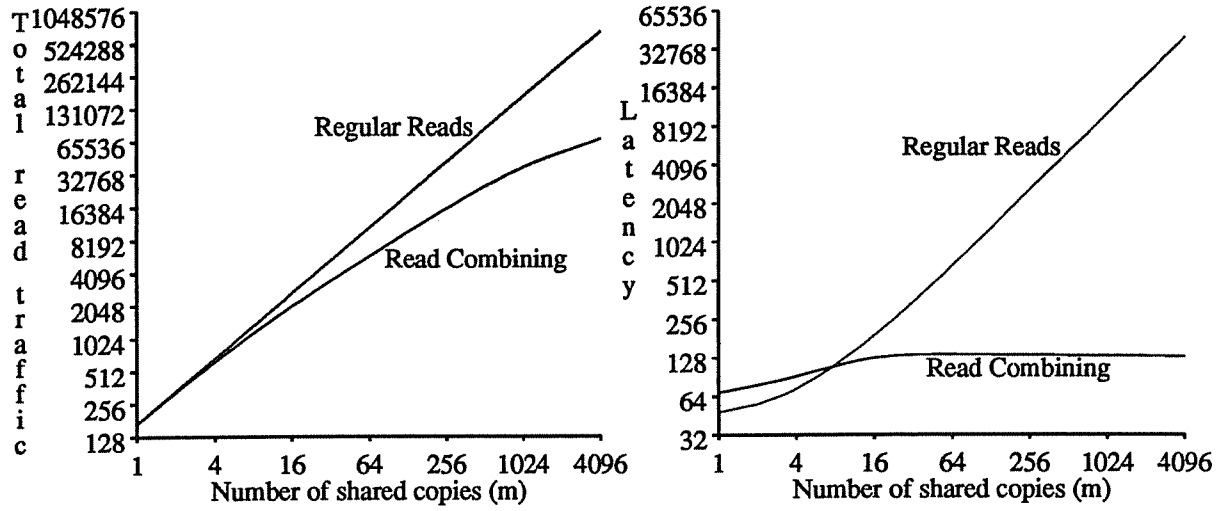
$$L_{comb} = E_{min\_links}(m) + E_{min\_dims}(m)L_{cache} + T_{addr}$$

$$+ max(L_{mem}, T_{addr})(kP_C(n-1,m)-1)$$

$$+ L_{mem} + E_{max\_links}(m) + E_{max\_dims}(m)L_{cache} + T_{data} \tag{5.13}$$

where $E_{min\_links}(m)$, $E_{max\_links}(m)$, $E_{min\_dims}(m)$ and $E_{max\_dims}(m)$ are the expected minimum and maximum links and dimensions between the memory module and the participating readers (see Appendix B).

Figure 5.6 illustrates the potential benefit of hierarchical read combining. I have assumed values of T_{addr}=1, T_{data}=9, L_{cache}=3 and L_{mem}=10. Note that the traffic with combining is always



(a) Total Read Traffic (b) Latency for all Reads to Complete

Figure 5.6: Performance of read combining (N=4096, k=8, n=4)

less than or equal to the traffic without combining, but that the latency with combining is greater than the latency without combining when m is small. This is due to the extra time needed to access caches on the way to and from memory. Thus, robustness in the presence of read contention comes at the price of performance in the presence of no contention. Moreover, most reads will be single reads, and thus will only be delayed by the combining mechanism. Reads to private data, however, can bypass the combining and pruning cache mechanisms altogether.

In an actual system, the relative difference between the two schemes for m=1 would be lessened, because both would suffer queueing delays for the links, which are not represented in Figure 5.6(b). It may also be possible to overlap queueing delays with cache access, or even bypass the cache access entirely when there is no network contention. On the other hand, the addition of read combining does add complexity to the coherence protocol (requiring two additional cache states) that might affect the performance of a specific implementation.

Due to the added complexity of read combining and the increased latency for single reads, the value of the proposed read combining mechanism is unclear. Simulation results presented in the next chapter will attempt to shed light on this issue, but the tradeoffs depend heavily upon workload behavior — a difficult subject to pin down.

Chapter 6

Performance and Management of Pruning-Cache Directories

The previous chapter introduced pruning-cache directories as a cache coherence mechanism for large scale multiprocessors. The performance of a cache line invalidation in a pruning-cache-based system was characterized, and a read combining mechanism compatible with pruning-cache systems was described. This chapter further investigates the performance of pruning caches, and explores several issues relating to their operation.

It was argued in Chapter 5 that total storage for both pruning caches and inclusion caches (for multi-level inclusion (MLI)) scaled with system size as $O(N \log N)$. Furthermore, both mechanisms are hierarchical in nature and thus compatible with concurrent read combining. Therefore, they are both attractive candidates for large-scale systems, and it is important to understand the differences between them. Section 1 of this chapter presents a simulation study that compares pruning-cache directories to MLI. The intent of the study is both to compare the two approaches to cache coherence, and to examine the actual performance of a pruning-cache-based system as the system size is increased.

Section 2 considers different types of sharing behavior, and explores extensions to the basic pruning cache protocol to more efficiently handle certain sharing patterns. The motivation is both to increase the speed and decrease the cost of common operations, and to reduce the pruning cache storage overhead.

In Section 3, several pruning cache management issues are explored, including the replacement policy and treatment of all-zero pruning vectors. Section 4 presents simulation results to validate the analysis presented in Chapter 5. Section 5 discusses the implementation of broadcast operations. Section 6 compares the total storage overhead needed by pruning caches and several other proposed cache coherence mechanisms reviewed in Chapter 5. Finally, a discussion and summary of the results is presented in Section 7.

1. Simulation Study of Pruning-Cache Directories

While the analysis in Chapter 5 provides detailed performance metrics for isolated invalidation and read operations, it cannot provide overall system performance measures without a host

of additional assumptions. Many of these assumptions, such as the pruning-cache hit rate, would be quite difficult to substantiate. In addition, the analysis does not address multi-level inclusion, and the effect of subtree invalidations on the hit rate of subsequent read requests. In order to address these and other issues, a detailed simulation study was performed. The goals were specifically to assess the effectiveness of pruning caches, to compare pruning caches against multi-level inclusion, and to validate the intuitive notion that pruning-cache based systems should scale well.

In all of the simulation results presented in this section, system size is varied from 16 to 256 processors, and the resulting performance is plotted for each of a set of protocols running a given workload. This gives an indication of how the protocols compare to each other, and how they behave as the system size is increased.

1.1. The simulator

The simulator is driven by synthetic traces of second level cache accesses. Inter-reference times are exponentially distributed, and represent the time spent processing and making first level cache hits. First level caches are assumed to contain subsets of the second level caches and to be write back. First level cache accesses and the interaction between the processor, first, and second level cache are not modeled, but contention for the second level caches between the processor and network sides is modeled. Second level caches are 2-way set associative and consist of SRAM tag memory (3 cycle access) and DRAM data memory (10 cycle access), which can be accessed independently. A read hit requires 13 cycles while a write hit requires 3. Memory accesses are 10 cycles. Protocols for the second level cache are briefly described in Section 3.3.

The decision to use synthetic rather than recorded traces was a compromise. Recorded traces have the advantage that they represent at least one possible actual execution of a program. However, unless the traces are from the actual workloads expected to run on the finished machine, there is no reason to give them more credibility than any other traces, synthetic or otherwise. Moreover, the behavior of a program being traced depends heavily upon how it was written, which in turn may depend heavily upon the machine that it was written for. The advantage of synthetic traces is that they are easy to produce, and can be manipulated to mimic many different types of behavior. The resulting traces, however, do not necessarily represent the behavior of any program. Thus, the simulation results presented in this chapter are best used to qualitatively compare the performance of various coherence mechanism, rather than predict the quantitative performance of any one mechanism.

The interconnection network is a k-ary n-cube, with unidirectional, point-to-point links. Link transmission times are assumed to be one cycle. The cache line size is 64 bytes. Address packets are 8 bytes, and data packets 72 bytes. The link widths are 32 bits. The state of all cache lines in all data caches and the contents of all pruning caches and inclusion caches are explicitly maintained. The resulting memory demands of the simulation running on departmental DECstation 3100 workstations dictated a second level cache size of only 128K. This is smaller than it would be in practice, but the workloads are similarly small.

1.2. Workload characterization

The workload is characterized by some number of *segments*, each of which can have independent characteristics. For each segment, the reference probability, size, maximum number of processors sharing data from the segment, dimension along which these sharers are arranged, read/write probabilities, and the spatial standard deviation of the second level cache references are specified. Each processor has a virtual space consisting of its local segments, and the virtual spaces are woven together to form a single global physical address space. This system allows a large degree of flexibility in constructing traces with desired properties.

1.3. Protocols simulated

Four different protocols are simulated. In all of the protocols, each line has a global state (maintained in the directory associated with memory) and each line residing in a cache has a local state. The global state is either shared or modified. If modified, then memory has a pointer to the cache containing the line and memory's copy is invalid. If shared, then memory's copy is valid, and the directory associated with memory contains, in the case of pruning caches, the top level pruning vector, and in the case of MLI, the top level inclusion bit.

All four protocols treat write misses and accesses to private data in the same manner. Misses are routed directly to the home memory module, and the data is returned in modified state. If another cache has a modified (and thus exclusive) copy of the line, memory reroutes the request to that cache, which returns the data, forming a three leg transaction. Memory changes its pointer at the time the request is rerouted. If several write requests occur in succession, then a hardware queue of waiting writers is built up across the caches. Each rerouted write request will be waiting at the appropriate cache for the line to arrive, and will pass the data along after it arrives and is modified. Although concurrent write requests to the same line are generally frowned upon, they can occur both in asynchronous algorithms or in workloads where false sharing (accesses to different words allocated to the same line by multiple processors) is present.

The protocol is significantly complicated by certain race conditions involving write backs, and is not given here in detail. However, it has been extensively tested using random synthetic workloads, a technique that proved extremely useful for detecting bugs in both the simulator and earlier versions of the protocol. Testing via stochastic simulation has proven an effective means of verification [Wood90]. However, before implementing such a complex protocol in hardware, more rigorous verification techniques are recommended [Burc90].

The four protocols differ in how they handle shared read misses, and write misses that find the data globally shared (requiring invalidations). The base protocol (labeled *Direct* in Figures 6.1 through 6.6) routes all read requests directly to memory, performing no read combining. Invalidations are broadcast to all processors, with acknowledgements hierarchically combined as described in Chapter 5, Section 2. The protocol labeled *Comb* uses read combining implemented with pending caches. Parent caches do not allocate space in their caches for children's read requests, but do maintain pending caches so that concurrent read requests can be combined. The *MLI* protocol enforces multi-level inclusion via inclusion caches. Parent caches allocate space in their cache for childrens' shared data read requests, and concurrent read requests are combined using the cache state. MLI is enforced using inclusion caches, however, so shared data may fall out of parents' caches without requiring the data to be invalidated in their children's caches. Invalidations are pruned using the inclusion information, and when entries are lost in the inclusion caches, then the corresponding lines are invalidated in the subtrees beneath the caches. Lastly, the *PC* protocol uses pruning caches to prune invalidations, and performs read combining through the data caches as does *MLI*.

Inclusion and pruning caches are 2-way set associative, using LRU replacement, and are simulated with sizes of 256 and 2048 entries. Given sufficiently large pruning/inclusion caches, both *PC* and *MLI* will perform perfectly (and *identically*), delivering the performance of a full directory. The cache sizes used in the study were chosen to investigate the performance of *PC* and *MLI* when smaller (and thus less expensive) caches are used.

1.4. Workloads simulated

6 different workloads are simulated in these experiments. All but the first workload include a 64K read-only code segment, an 8K private data segment (P_{write}=30%), and an additional data segment that determines the characteristic behavior of the workload.

- The first workload consists of independent processes, each with its own 64K code segment and 128K data segment (P_{write}=30%). It is simulated both with accesses marked as private, to determine the scalability of the system for uniform traffic, and with accesses non-marked

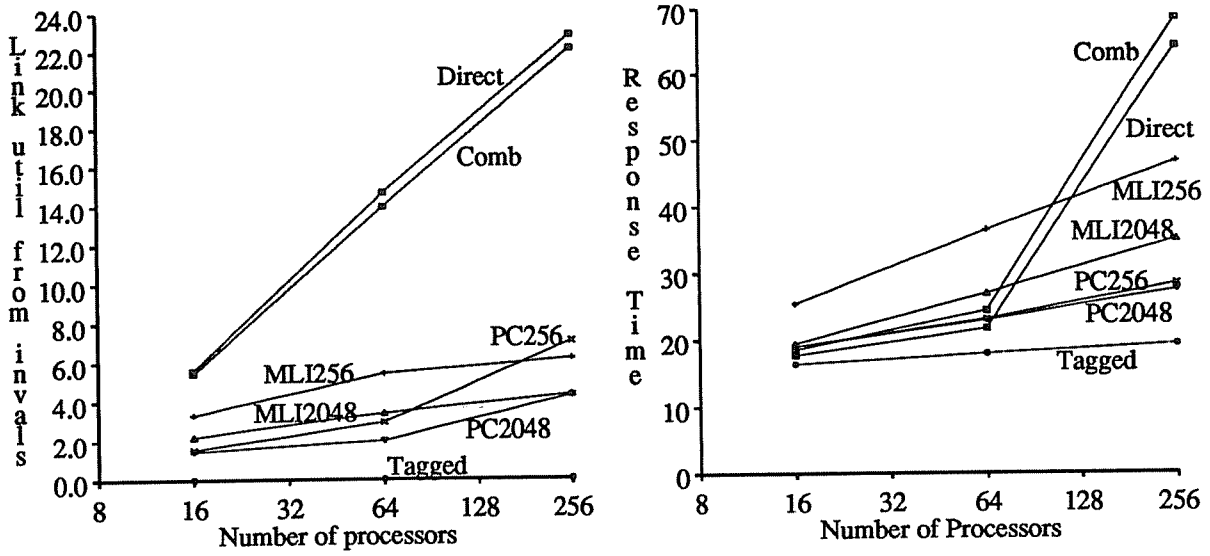
(defaulting to shared), to compare how *MLI* and *PC* handle large data spaces with no active sharing and to observe the performance degradation due to broadcast invalidations.

- The second workload uses a fixed 512K segment shared among all processors. P_{write} is fixed at 30%, so contention for the data increased as system size increased.
- The third workload uses a fixed 512K shared segment also, but scales the write probability down with increased system size (P_{write}=4/N). A workload where most processors read a line between successive writes to the line would exhibit the same sort of behavior.
- The fourth workload uses 4-way sharing with a local segment size of 32K. P_{write} is fixed at 15%. Each line is contained in the virtual space of four different processors, for a total global physical segment size of N(8K). Sharing is arranged along the root dimension, which spreads out sharers among different leaf rings.
- The fifth workload is identical to the fourth, but with sharers arranged along the leaf dimension, which concentrates sharers on the same leaf rings as much as possible. This data partitioning in this workload is designed to exploit locality.
- The sixth workload is identical to the fifth, but with P_{write} set to 5% and a periodic concurrent read request added. The read request occurs approximately once every 1000 data accesses and (somewhat artificially) simulates read contention caused by synchronization.

1.5. Results

Figures 6.1 through 6.6 give results for the 5 workloads. Each figure contains two graphs: one showing invalidation traffic and the other showing response time at the second level cache. The invalidation traffic is the percentage of available link bandwidth used by invalidations (including their acknowledgements). It is obviously desirable to keep this small and as close to constant as possible as system size is increased. The cache response time is affected by both the hit rate and miss penalty (which in turn depends on system size and network traffic). This is expected to rise as system size is increased, but, again obviously, the less rise the better.

Figure 6.1 shows results for independent workloads. For the curve marked *Tagged*, all requests are tagged as private. In this case, no coherence actions are necessary, and all protocols give the same performance. There are no invalidations, and we see in Figure 6.1(b) that the cache response time increases only mildly as system size is increases from 16 processors (two dimensional) to 256 processors (four dimensional). This supports the claim that uniform traffic scales well in a cube network (although the affect of wire length and wiring constraints are not being considered here). When private accesses are not marked as such, however, performance degrades for all protocols.



(a) Invalidate traffic (b) Cache response time

Figure 6.1: Protocol performance and scalability
Independent processes

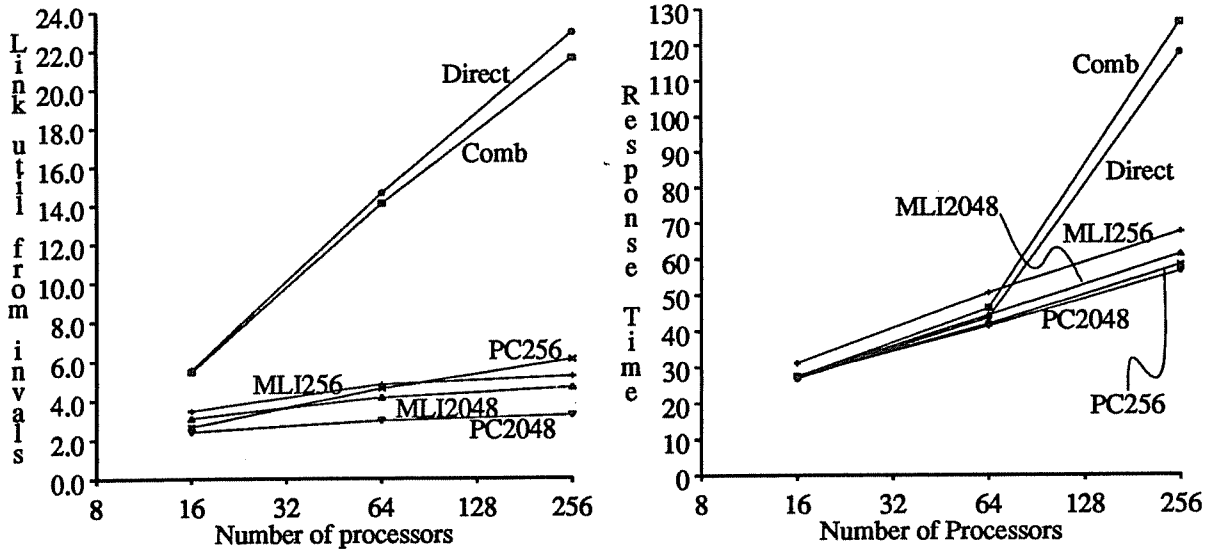
The invalidation traffic soars for *Direct*, supporting the assertion that traffic from broadcast invalidations does not scale. For N=256, the invalidation traffic takes up over 20% of the link utilization. More importantly, the broadcast invalidations have completely saturated the processors' cache tags, which gives rise to the sharp increase in response time.

Since the workloads are completely independent (and fairly large compared to the cache size), the inclusion and the pruning caches are much too small to maintain high hit rates. We see that this affects them in different ways, however. Invalidation traffic is fairly high for *PC*256. This is most likely due to misses for high-level pruning vectors, causing subsequent invalidations to be broadcast to many processors. *MLI*256 keeps the invalidation traffic lower than *PC*256, because it never has to broadcast invalidations, but it results in much higher cache response time due to misses caused by subtree invalidations. With sizes of 2048 entries, the pruning and inclusion caches perform much better, but not perfectly. We still see that the response time for *PC*2048 is significantly lower than the response time for *MLI*2048. This illustrates an important advantage of pruning caches over inclusion caches. When we can no longer keep track of a line, it is better to suffer increased invalidation traffic when the line is written than to prematurely invalidate the line.

If the data caches were big enough to hold all the passively shared data (data marked as shared, but in fact not actively shared) then pruning caches would have an additional advantage over inclusion caches. Entries for these lines would drop out of the pruning caches, and the lines could remain in the data caches below. *MLI*, however, would require entries for these lines to reside in the inclusion caches, and thus would repeatedly invalidate lines as it replaced inclusion cache entries.

Figure 6.2 shows results for a fixed problem size with constant write probability. As system size increases, contention for data increases, but the sharing is very active and the mean number of shared copies on an invalidation remains small. The invalidation traffic for *Direct* and *Comb*, which use broadcast invalidations, increases dramatically with system size. By N=256, this traffic has caused the response time to be twice that of the other schemes (this is due primarily to saturating the processors' cache tags with invalidation messages). Read combining does not take place much in this workload, and we see that *Comb* has a higher cache response time than *Direct*, due to the extra delay in accessing memory.

PC and *MLI* both reduce the invalidation traffic significantly over *Direct* and *Comb*. Unlike the previous workload, however, *MLI* does not have a significantly higher cache response time



(a) Invalidate traffic (b) Cache response time

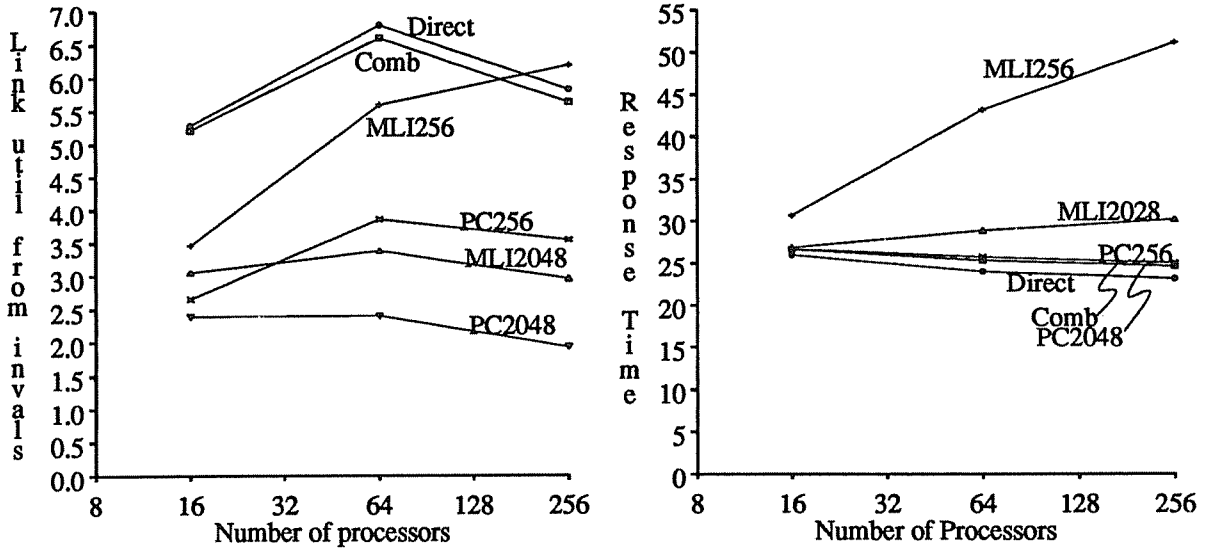
Figure 6.2: Protocol performance and scalability
Fixed problem size, P_W=30%

than *PC*. This is due to the active sharing, which reduces the demands on the inclusion caches and lessens the effect of subtree invalidations.

Figure 6.3 shows results for a fixed problem size with a write probability that decreases with system size. This behavior corresponds to inactive write sharing, and might arise in a workload where processors repeatedly read an entire data set to write a small subset. *Direct* and *Comb* are not significantly penalized for their use of broadcast invalidation, because the frequency at which processors modify shared data decreases with system size. This same characteristic would allow such a workload to run efficiently on a single-tree-based multiprocessor.

The inactive nature of the sharing penalizes *MLI*, however. Pruning-cache entries for inactive lines can be replaced without harm, but inclusion cache entries that are replaced cause invalidations that increase the data cache miss rate and drive up cache response time.

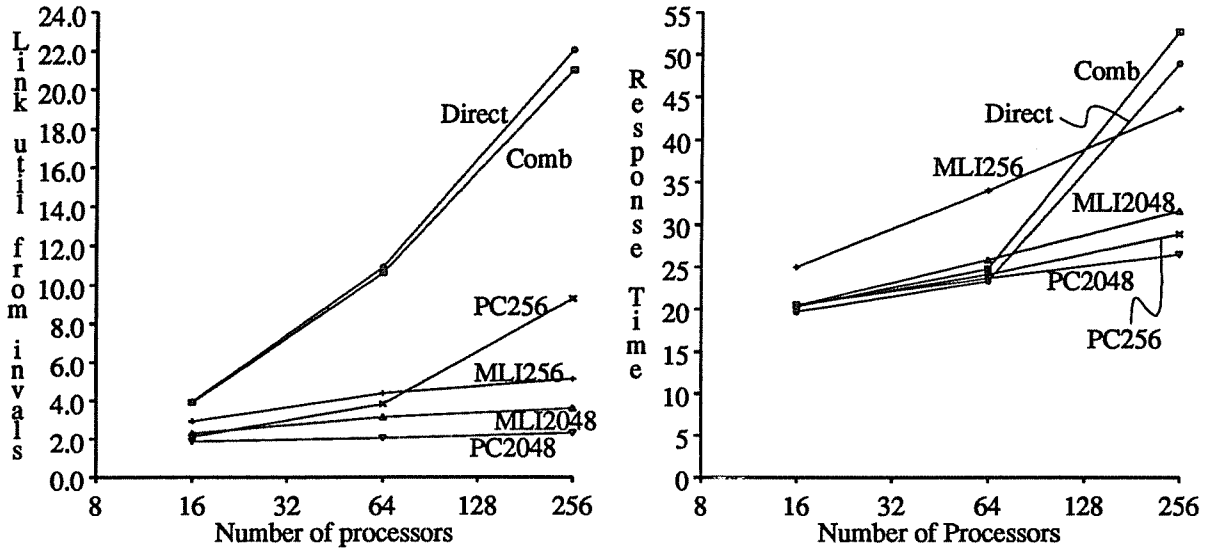
The workload presented in Figure 6.4 scales in size with the number of processors. All sharing is 4-way and distributed along the root dimension (the worst-case distribution for inclusion and pruning caches). Since the local segment sizes and degree of sharing are independent of system size, changes in performance as system size increases should be due solely to scaling



(a) Invalidate traffic (b) Cache response time

Figure 6.3: Protocol performance and scalability

Fixed problem size, $P_{write}=\frac{4}{N}$



(a) Invalidate traffic (b) Cache response time

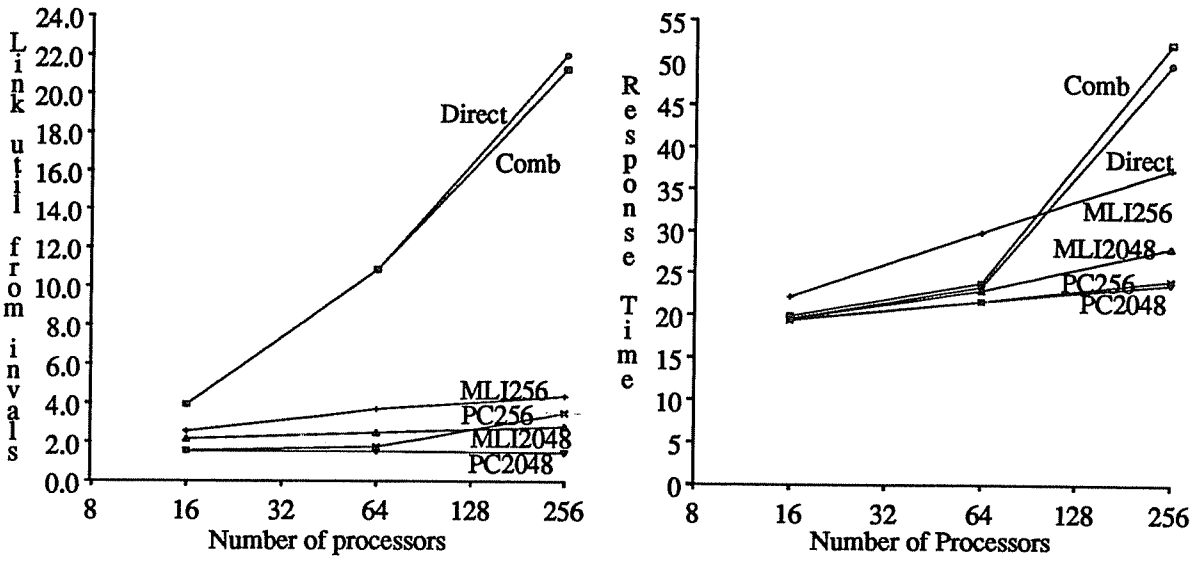
Figure 6.4: Protocol performance and scalability
4-way sharing (root dimension), problem size αN, P_{write}=15%

effects. The large address space and bad distribution result in a poor hit rate for *PC*256, resulting in heavy invalidation traffic for N=256. *PC*2048 maintains a good hit rate for all system sizes, and so invalidation traffic remains low. Although the 256-entry inclusion caches are also too small, it is difficult to tell from the invalidation traffic. It is readily apparent, however, by looking at the cache response times, which are significantly higher for *MLI*256 than for *MLI*2048, *PC*256 and *PC*2048.

Contention caused by broadcast invalidations causes *Direct* and *Comb* to perform very poorly for N=256. Again we see that the combining mechanism did not improve performance, but rather worsened it by increasing the latency to memory for shared read requests.

The workload in Figure 6.5 is identical to that in Figure 6.4, save that the 4-way sharing is now arranged along the leaf dimension (the best-case distribution for inclusion and pruning caches). The performance of *Direct* is unchanged, but the performance of all other schemes improves. The response time of *Comb* lowers slightly, indicating that additional read combining is taking place. *Comb* still performs worse than *Direct*, however; there is not enough read combining in this workload to make up for the increased read latency.

The better distribution has a positive effect on the performance of pruning and inclusion caches as well. Invalidation traffic is lower than with the bad distribution, as is cache response



(a) Invalidate traffic

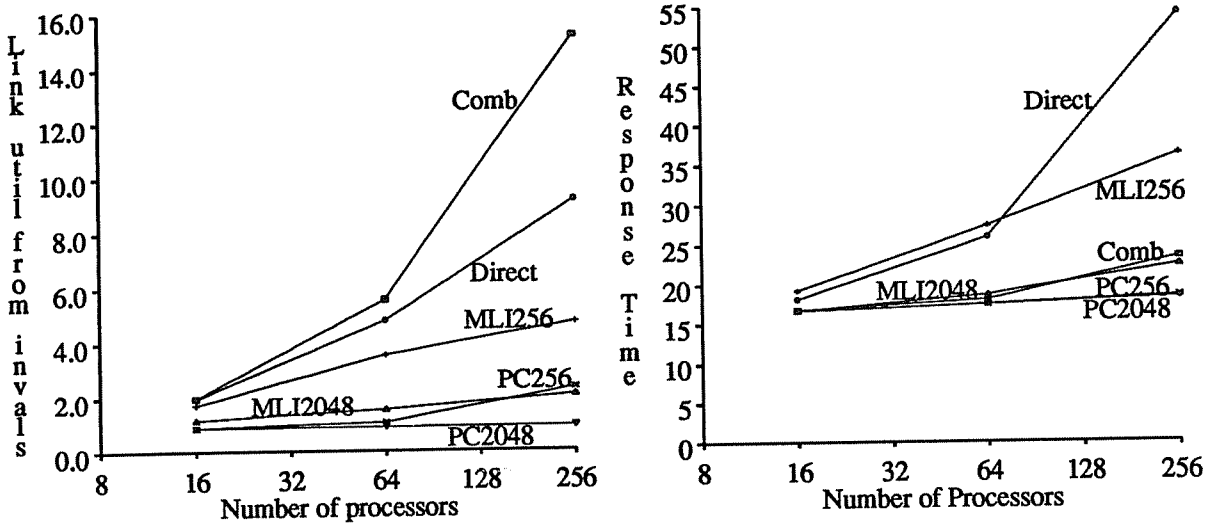
(b) Cache response time

Figure 6.5: Protocol performance and scalability
4-way sharing (leaf dimension), problem size αN, P_{write}=15%

time. The response times of *PC* are still slightly lower than those of *MLI*, even though both the pruning and inclusion caches have very low replacement rates.

The workload in Figure 6.6 is similar to that in Figure 6.5, save that a periodic "barrier-like" operation has been inserted. Approximately once every 1000 data accesses per processor, all processors attempt to read the same line (which is not valid in any of the caches) at approximately the same time. The time between the first read and each other read is exponentially distributed with a mean of 20 cycles. This read operation could represent either a synchronization operation itself, or a concurrent read that was caused by a synchronization event. In addition to this periodic concurrent read, the workload of Figure 6.6 differs from that of Figure 6.5, in that P_{write} is set to 5%, rather than 15%, for the shared data segment. This reduces the degradation from broadcast invalidations so that the effect of read combining can be more easily seen when comparing *Comb* and *Direct.*

For a workload displaying this sort of contention, read combining is clearly beneficial, and the benefits become greater as system size increases. The cache response time for *Direct*, the only protocol that does not use read combining, increases dramatically as system size increases. This is due to contention for the "synchronization" line, *not* increased link or cache tag contention caused by heavy invalidation traffic. Invalidation traffic is responsible for the difference in



(a) Invalidate traffic (b) Cache response time

Figure 6.6: Protocol performance and scalability
"Barrier synchronization", 4-way sharing (leaf dimension), problem size αN, P_{write}=5%

response time between *Comb* and *PC*256/*PC*2048.

How much read contention occurs in actual workloads is an open question. There are several possible causes of read contention. A program using barrier synchronization, may experience contention directly after leaving a barrier. Results produced during iteration i may be widely read at the beginning of iteration i+1. Similarly, certain code fragments may be widely read at the same time, because processors leaving the barrier are well synchronized. In a data parallel programming environment, the states of the various caches may be very similar throughout the program execution, leading to frequent concurrent reads on instruction cache misses. Another cause for read contention is the modification of a widely (and frequently) read variable.

While not answering the question of how much read contention occurs, the results shown in Figure 6.6 demonstrate that, with the combining mechanism described here, read contention will not be a problem if it *does* occur. Read combining mechanisms do add complexity to the coherence protocol, of course, so a detailed workload study would have to be performed to justify their inclusion in an actual system.

2. Enhancements to the Basic Protocol

The basic pruning cache protocol appears to work well when compared to a similar MLI protocol or to a broadcast-based protocol. It is likely, however, that the protocol can be further refined. This section attempts to do so, both by reasoning about the coherence requirements of different types of data and by considering the simulation results presented in the previous section. Section 2.1 briefly discusses several types of cached information, distinguished by their access characteristics, Section 2.2 proposes two modifications to the basic protocol, and Section 2.3 presents simulation results for the modified protocols.

2.1. Types of sharing

Here I identify 9 potential classes of cached information: *private instructions and data, shared instructions, migratory data, passively shared data, globally shared data, data shared by many processors, data shared by few processors, contended instructions and data* and *synchronization objects*. Each class of information has different needs or costs for maintenance of coherence. Below, I discuss the various classes and how the pruning cache protocol does or could support them. These classes are quite similar to those defined by Weber and Gupta [Webe89].

Private instructions and data can be placed exclusively into one cache and kept track of with a single directory pointer. The coherence protocol must be able to retrieve the data and change ownership should the task owning the data migrate to another processor or another task reclaim the space after the first task completes. If references to private instructions and data can be recognized (and a compiler should certainly be able to do so), then they can be treated just like writes to shared data, causing the pruning cache protocol to keep them exclusively in one cache.

Shared instructions may exist in many caches, but are presumably rarely written (only when the physical address space is reclaimed). Thus, although the address space must be invalidated when reclaimed, it probably does not matter how efficiently this is done. There are two choices for managing shared instructions using pruning caches. First, the information regarding their locations could be entered into pruning caches just like any other shared data. If the space was not reclaimed for a long time, then the pruning vectors would likely drop out of the caches and the eventual invalidation of the addresses would consume higher than normal bandwidth (of course if the instructions were globally shared, then a broadcast invalidation would be needed anyway). If the space were reclaimed quickly, then saving the pruning vectors was the right choice. The disadvantage of this choice is that placing the information into the pruning caches pollutes the caches and probably won't be needed.

The other choice is to not place pruning vectors for shared instructions into the pruning caches and then just suffer increased invalidation traffic when the space is reclaimed. The top level vectors could always be placed in their respective directory entries, as this would not cause any pruning cache pollution. This would limit somewhat the amount of invalidation traffic caused later by reclaiming the space. Either of these two choices seems reasonable.

Migratory data has essentially the same needs as private data, but may be harder to identify. If the compiler can identify references to migratory data as such, then they can be fetched for exclusive use, and the pruning cache protocol works well. If they cannot, and the first reference to the data is a read (as it is likely to be), then this read will create a trail of pruning cache entries that a subsequent write will have to use to invalidate the line. This causes extra traffic and delay, even if the invalidation hits in all the pruning caches it accesses, and it also causes extra pruning cache contention. A modification to the basic pruning cache protocol that addresses this problem, PC_{own}, is described and studied in the next section.

Passively shared data is data that is nominally shared, but in fact read/written by only one processor. A good example of this phenomenon can be found in programs that split up a large array into regions for each processor. The data along the border of a processor's region is shared with neighboring processors, but the interior of a processor's region is read and written only by that processor.

If accesses to passively shared data can be recognized by the compiler, then exclusive reads can be performed, as with private data. This is most likely an unrealistic expectation in general, however. Thus, the first read to such a cache line will cause entries to be placed into pruning caches, which will be used to invalidate the line if and when it is later written. After the write, the line can remain in exclusive state in the appropriate data cache. Although this overhead may be acceptable if the line is used many times after the first write, it can be avoided by the PC_{own} protocol described in the next section.

Globally shared data does not present a problem in a pruning-cache-based system, as a write to the data requires a broadcast invalidate, so pruning cache performance is irrelevant. If the data is rarely written, then pruning vectors may drop out of the pruning caches without harm, making way for other entries. If the data is written often, then the latency of the reads and invalidates will be important and the ability to combine reads will be beneficial (this case is discussed below).

Data shared by many processors poses a problem to limited pointer directories, but is easily handled by pruning caches. If the data is actively shared, then the pruning caches should work

well. If not, then pruning vectors can eventually fall out of the pruning caches leaving room for other entries.

Data shared by few processors poses an even larger problem to limited pointer directories if the number of sharers of a line exceeds the number of available pointers. If the number of sharers is less than or equal to the number of available pointers, then the limited pointers will work very well. Pruning caches should work well for this type of data, just as for data shared by many processors. However, this is the case for which the pruning cache hit rate is most important. A low hit rate for frequently modified data that is shared by only a small number of processors will lead to excessive invalidation traffic relative to read traffic.

Contended instructions and data are lines that are that are concurrently read by many processors. Most coherence protocols cause these reads to be serialized. If concurrent reads happen very infrequently, then the latency of the reads and subsequent invalidations is not important. If, however, the data is frequently written and re-read, then the latency of the reads is very important, and the latency of the invalidations is likely important (depending upon the memory consistency model being used and the program characteristics). The combining mechanism possible with pruning caches is very desirable for this class of information.

Synchronization objects present a problem to cache coherence protocols primarily in the form of contention. A barrier notification mechanism, for example, may write a flag on which waiting processors are spinning. The resulting invalidation will cause all waiting processors to re-read the flag from global memory, causing significant read contention. Heavily contended lock variables can cause contention in the same manner. Since pruning-cache directories are hierarchical in nature, they are compatible with read combining and thus well suited for systems with synchronization contention.

Although a pruning-cache-based system may be able to tolerate contention caused by synchronization objects, it would be better to avoid the contention altogether. Mechanisms that support locks using software or hardware queueing can prevent lock contention, and can be used as primitives to support contention free barriers and other synchronization operations as well [Mell91, Good89a]. In addition, the QOLB hardware synchronization primitive [Good89a] can improve the performance of pruning-cache directories by removing interference from migratory data. QOLB automatically migrates lock-protected data from one cache to another, allowing the data to remain in globally modified state. Producer-consumer and pairwise-shared data can likewise be managed directly using QOLB, thus circumventing the cache coherence mechanism.

2.2. Modifications to basic protocol

A primary shortcoming of the pruning cache protocol, identified in the previous section, is that unmarked reads to migratory and passively shared data load the data in shared mode, creating pruning cache entries, when in fact the data is used by only one processor. Another problem with the pruning cache protocol, observable in the simulation results of Section 1, is that read requests are slowed down by intermediate cache lookups on their way to memory. Both of these problems can be addressed by modifications to the basic protocol.

The *direct* modification causes shared read requests to be routed directly to memory, bypassing intermediate caches (or pending caches). Reads routed in this manner will be faster, but are not able to combine. When the read result is returned, it can still cause pruning cache entries to be placed in intermediate pruning caches, allowing for later invalidation of shared lines using the basic pruning cache protocol. I refer to a pruning cache protocol with direct routing as PC_{dir}.

The *ownership* modification causes the first read of a line to be returned in read-only exclusive state, with a single directory pointer keeping track of the "owner". If and when the line is written, a single invalidation can be sent to the owner. If and when a second read request for that line reaches memory, the directory entry converts to a pruning vector, the line is returned to the second processor in shared mode (creating pruning cache entries in intermediate nodes), and a special packet is routed to the first processor that creates the necessary pruning cache entries for its copy of the line. I refer to a pruning cache protocol with the ownership modification as PC_{own}

The advantage of the PC_{own} protocol is that now information is entered into the pruning caches only for truly shared data (shared data which is cached by more than one processor). This significantly cuts down on the amount of information that competes for space in the pruning caches. Invalidation traffic is also reduced, because invalidations to lines in the owned state are sent directly, eliminating the possibility that pruning cache misses will lead to partial broadcast of the invalidations.

The PC_{own} protocol uses direct routing, because the memory must be able to return the read result in the shared exclusive state. Like PC_{dir}, this has the advantage that read requests are faster, but it has the disadvantage that PC_{own} is not compatible with read combining, so concurrent read requests can no longer be satisfied in parallel. The PC_{own} protocol *solves* two problems (slowdown to read requests from having to access caches on the way to memory, and pruning cache pollution and attendant increased invalidation traffic from reading migratory and passively shared data in shared mode) while *creating* a third (the inability to perform read combining). The

solution may be to provide correctness in hardware, but depend on software to some extent for performance. The default read request could be either a direct read or a combining read. Software (meaning the compiler or programmer) would have to indicate when it wanted the other case, and would tag the read requests correspondingly.

A likely policy would be the following. By default, read requests would be shared reads using the PC_{own} protocol. Thus singly-cached lines would be kept track of by pointers in the directory and the pruning caches would contain information only for truly shared lines. Any reads that the compiler knew where private or were to data that was going to be modified would be tagged as private and loaded in the exclusive, read/write state. Any reads that the compiler or programmer had reason to suspect would cause read contention would be marked as combinable, and would use the standard combining pruning cache protocol discussed in Section 1. This is quite similar to the idea of having two networks in the NYU Ultracomputer [Gott83], one for regular traffic and one for combinable traffic, except that here it would be done by specifying the *type* of read requests traversing a *single* network in order to change the way in which the communication protocol handled the messages.

Another possibility would be to have all shared read requests default to direct routing, but check in parent caches for combining if and only if they were queued due to network congestion. In this way, the delay to implement combining would occur only when combining was potentially needed, and would be overlapped with queueing delay that would be suffered anyway.

2.3. Simulation results for modified protocols

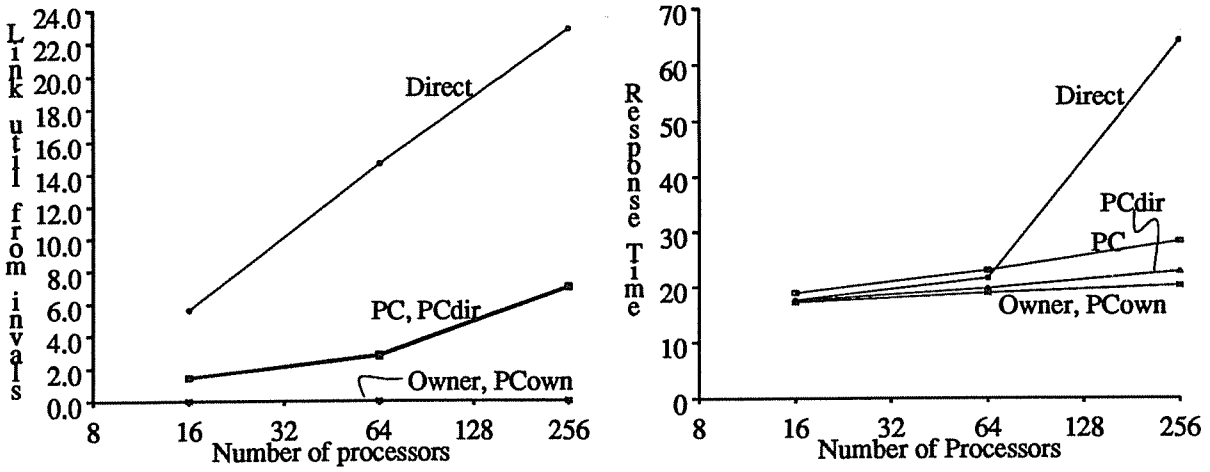
This section presents simulation results for the protocol variants discussed in the Section 2.2. As in Section 1, six workloads are simulated (descriptions appear in Section 1.4), and system size is increased from 16 to 256 processors.

Five different protocols are simulated. All protocols maintain the locations of modified data via a single directory pointer, as described in Section 1. *PC* is the basic pruning cache protocol presented and simulated in Section 1, here with a pruning cache size of 256 entries. Shared read requests are combined using entries in their parents' data caches and invalidations are performed using pruning caches. *Direct* is identical to the *Direct* protocol in Section 1. All read requests are routed directly to memory and invalidations are performed via broadcasts. PC_{dir}, as described in Section 2.2, routes all shared read requests directly to memory, performing no combining, but uses pruning caches (here with 256 entries) to perform invalidations. *Owner* uses a single pointer to keep track of the first reader or a line, but then converts to broadcast invalidation if two or more processors read the line. All read requests are routed directly to memory with no

combining. This is equivalent to the Dir_1BC protocol discussed in Chapter 5, Section 1.1 [Agar88]. PC_{own}, as described in Section 2.2, is equivalent to PC_{dir}, save that the location of the first reader of a shared line is maintained using a single directory pointer. All read requests are routed directly to memory with no combining.

Figures 6.7 through 6.12 give results for the 6 workloads. As in Section 1.5, each figure contains two graphs: one showing invalidation traffic and the other showing response time at the second level cache. The invalidation traffic is the percentage of available link bandwidth used by invalidations (including their acknowledgements).

Figure 6.7 shows results for independent workloads (*without* tagging references as private). Since nothing is shared, the two ownership protocols keep track of all data using single pointers, have almost no invalidation traffic, and have the lowest response time. The pruning caches are too small to hold information on all the lines in the caches, so PC and PC_{dir} have a fair amount of invalidation traffic. PC_{dir} has a lower response time than PC because it routes read requests directly to memory and there is no need for read combining for this workload. *Direct*, just as in Section 1.5, has very high invalidation traffic, and, as a result, very high response time for large system sizes.



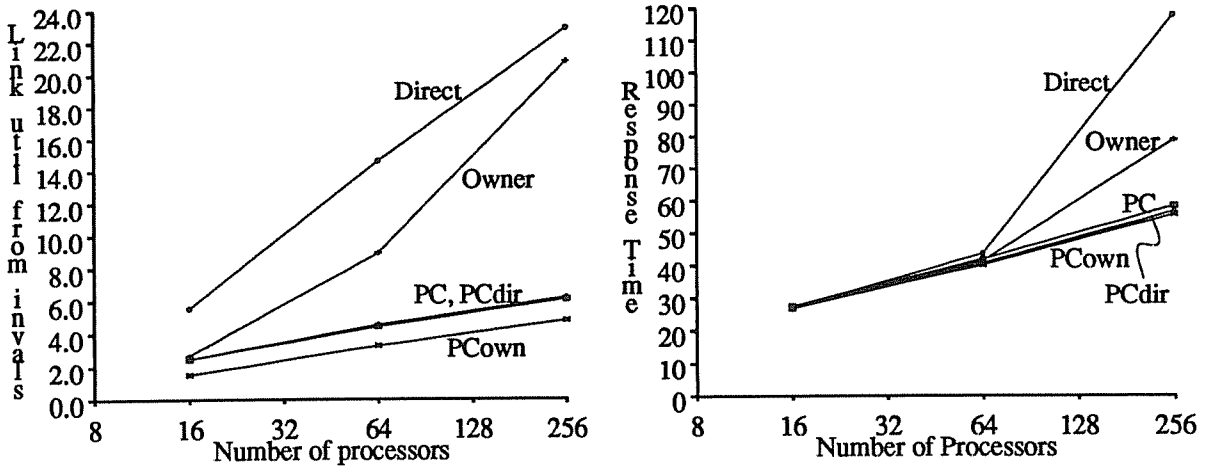
(a) Invalidate traffic (b) Cache response time

Figure 6.7: Performance of modified protocols
Independent processes

Figure 6.8 shows results for a fixed problem size with constant write probability. As system size increases, contention for data increases, but the sharing is very active and the mean number of shared copies on an invalidation remains small. PC_{own} has lower invalidation traffic than PC or PC_{dir}, showing that the ownership modification is helping. *Owner* has lower invalidation traffic than *Direct*, but the invalidation traffic is still unacceptably high. PC_{own} and PC_{dir} both have slightly lower response time than PC due to faster read requests, with PC_{own} marginally lowest because of its lower invalidation traffic.

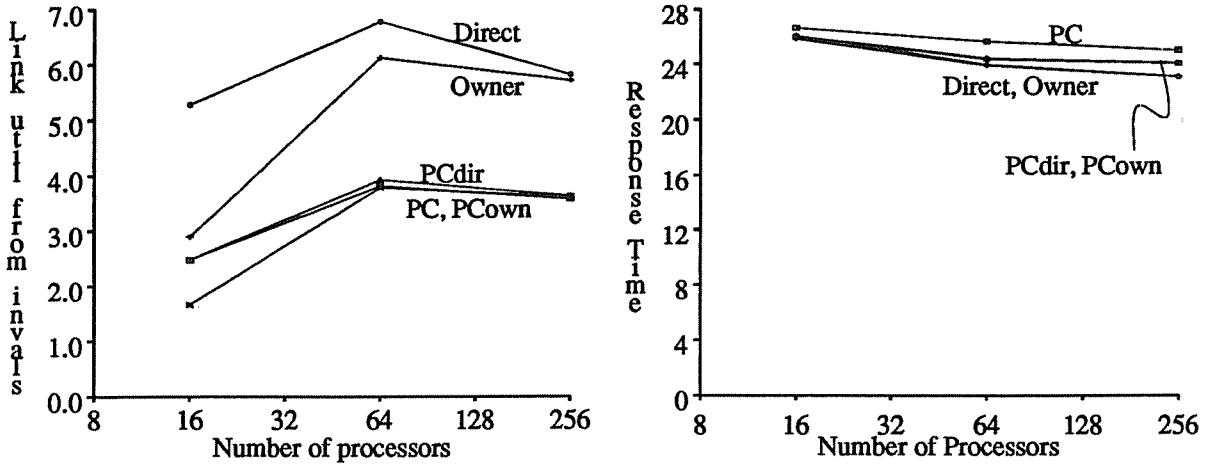
Figure 6.9 shows results for a fixed problem size with a write probability that decreases with system size. This behavior corresponds to inactive write sharing, and might arise in a workload where processors repeatedly read an entire data set to write a small subset. Since writes become very infrequent as system size increases, invalidation traffic remains relatively low for all protocols, and the response times are all quite close. PC has the highest response time, because of the extra overhead for read requests.

The workload presented in Figure 6.10 scales in size with the number of processors. All sharing is 4-way and distributed along the root dimension (the worst-case distribution for inclusion and pruning caches). Since the local segment sizes and degree of sharing are independent of



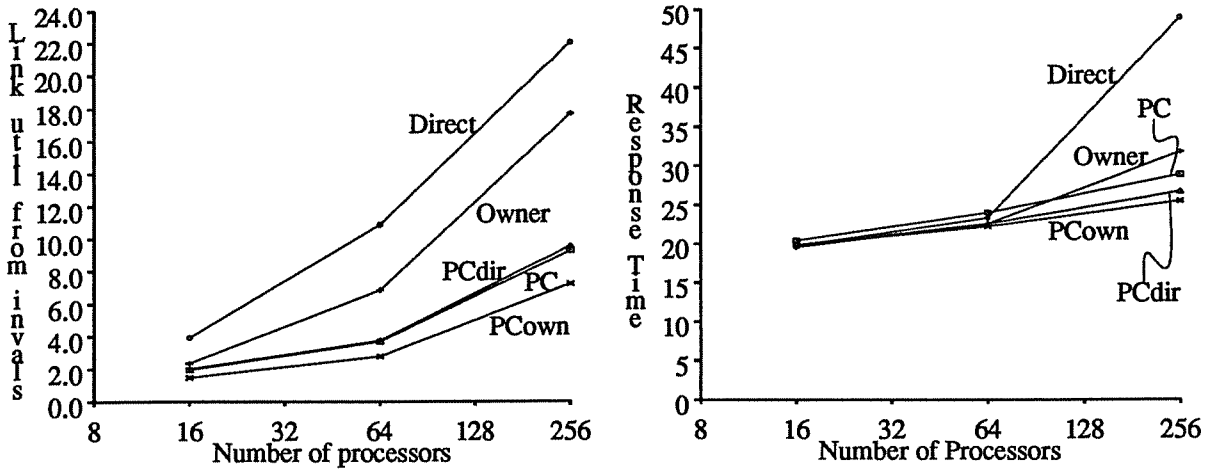
(a) Invalidate traffic (b) Cache response time

Figure 6.8: Performance of modified protocols
Fixed problem size, P_W=30%



(a) Invalidate traffic (b) Cache response time

Figure 6.9: Performance of modified protocols
Fixed problem size, $P_{write}=\frac{4}{N}$



(a) Invalidate traffic (b) Cache response time

Figure 6.10: Performance of modified protocols
4-way sharing (root dimension), problem size αN, P_{write}=15%

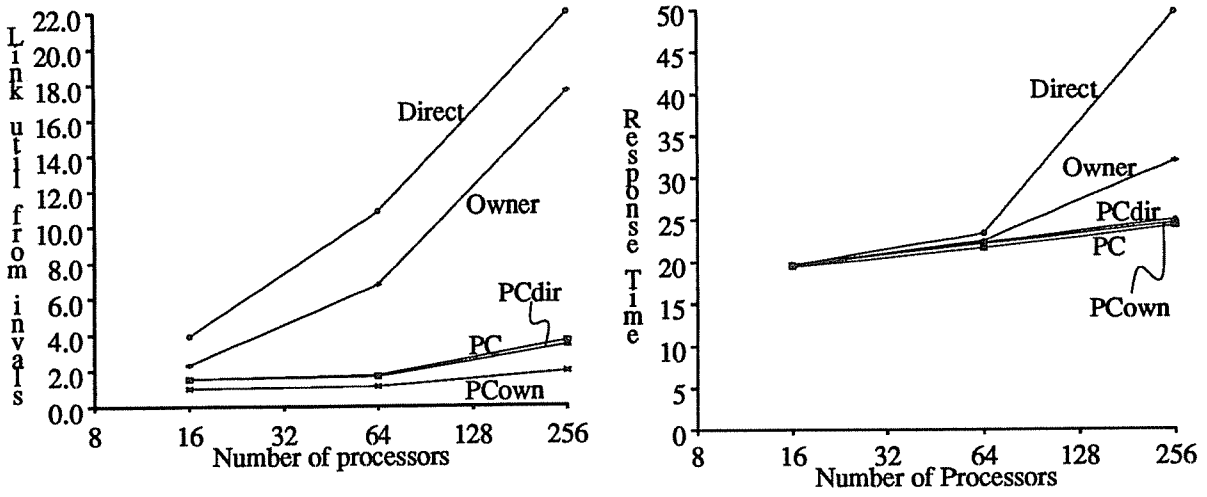
system size, changes in performance as system size increases should be due solely to scaling effects. PC_{own} has the lowest invalidation traffic and response time. PC_{own} and PC_{dir} both have lower response time than PC, due to lower overhead for reads. The response time for *Owner* becomes greater than that of PC for N=256, due to its higher invalidation traffic. *Direct* has high enough invalidation traffic for N=256 to saturate the processor cache tags and lead to very high

response time.

The workload in Figure 6.11 is identical to that in Figure 6.10, save that the 4-way sharing is now arranged along the leaf dimension (the best-case distribution for inclusion and pruning caches). The invalidation traffic for the three pruning cache protocols is lower than in Figure 6.11, indicating that the better partitioning improves pruning cache performance. Since the invalidation traffic was already fairly low, this has very little effect on the response times. The response time of PC is lower than in Figure 6.10, however, due to more frequent read combining.

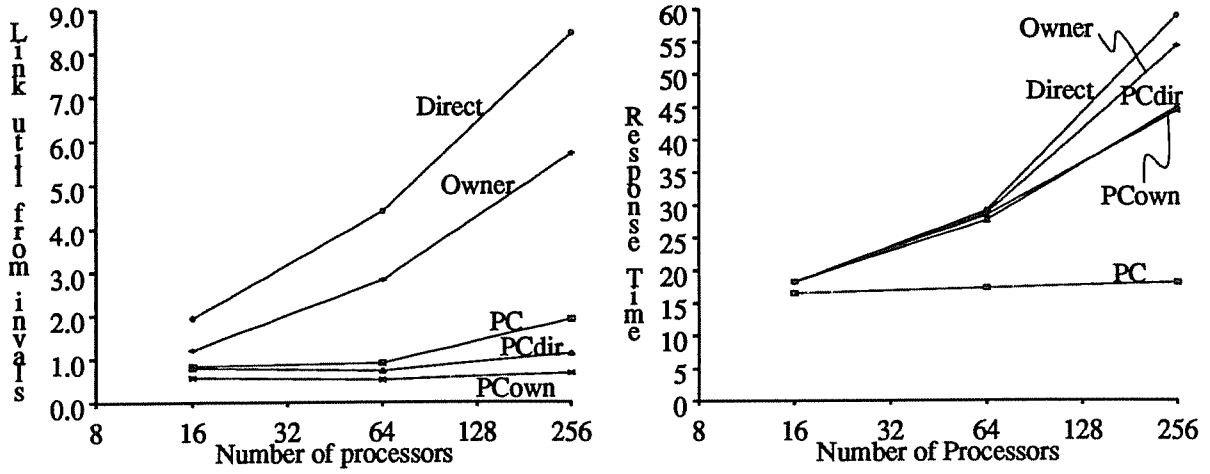
The workload in Figure 6.12 is similar to that in Figure 6.11, save that a periodic "barrier-like" operation has been inserted. Approximately once every 1000 data accesses per processor, all processors attempt to read the same line (which is not valid in any of the caches) at approximately the same time. The workload of Figure 6.12 also differs from that of Figure 6.11, in that P_{write} is set to 5%, rather than 15%, for the shared data segment.

This workload illustrates the disadvantage of the modified pruning cache protocols, PC_{dir} and PC_{own}. By not performing read combining, they are prone to very significant degradation when heavy read contention takes place (it is left to the reader to decide how "heavy" one concurrent read per 1000 data accesses is).



(a) Invalidate traffic (b) Cache response time

Figure 6.11: Performance of modified protocols
4-way sharing (leaf dimension), problem size αN, P_{write}=15%



(a) Invalidate traffic (b) Cache response time

Figure 6.12: Performance of modified protocols
"Barrier synchronization", 4-way sharing (leaf dimension), problem size αN, P_{write}=5%

As mentioned in Section 2.2, the best solution is probably to provide both types of read requests: direct and combinable. The compiler could mark requests according to whether or not they would likely need combining. In this way, most read requests could be routed directly to memory, and contention prone requests could be routed using the read combining protocol.

3. Pruning Cache Management Issues

Pruning caches are fairly complex, and as such give rise to some interesting management issues. Like any cache, placement and replacement policies are important considerations. Pruning caches also present the issue of how to populate and maintain the pruning cache entries (it is not as simple as reads and writes to a data cache). For example, should pruning caches entries be created as read requests traverse up the tree towards memory, or as read results traverse down the tree toward the data caches?[7] A particularly interesting issue is how to handle all-zero pruning vectors, which are qualitatively different than other pruning vectors. Section 3.1 discusses the pruning cache placement policy used in all simulations. Section 3.2 discusses replacement

[7] The answer, in this case, is "on the way down", so that race conditions with invalidations work properly. Otherwise, if an invalidation came through between the time the read request propagated up and the time the read result propagated down, then the pruning vector for the line would be erased, and the read result would be cached with no record in the pruning cache.

policies. Section 3.3 discusses policies for handling all-zero vectors. Section 3.4 presents simulation results for the policies discussed in Sections 3.2 and 3.3.

Design decisions, such as size and associativity, are also important. While size has been studied to some degree in Section 1, quantitatively accurate performance results for various pruning cache sizes can only be obtained by simulating actual workloads of interest. Associativity has largely been ignored; the pruning caches in the simulations have all been 2-way set associative. Associativity is briefly discussed below as it relates to the pruning cache placement policy.

3.1. Placement policy

The pruning cache placement policy determines where in a pruning cache a new pruning vector is placed. Let S be the number of sets in a pruning cache (the number of entries divided by the associativity), B be the cache block size in bytes, and a be the block address of a line for which a pruning vector is being stored in the pruning cache. The block address of a line is simply the byte address with the lower $\log_2 B$ bits truncated.

In a single tree system, the placement function is simple and straight-forward. We would simply use the block address, modulo the number of sets in the pruning cache. In a cube-based system, however, a given pruning cache holds pruning vectors for multiple dimensions. Thus the placement mapping is from a two-dimensional space: (address × dimension). If we ignore the dimension, then a line's pruning cache entries for multiple dimensions will map to the same set, which is obviously not desirable.

The placement policy used in all simulations presented in this thesis is

$$set(a,d) = (a >> (d \log_2 k)) \bmod S \tag{6.1}$$

where k is the radix of the k-ary n-cube, $d \in \{1...n\}$ is the dimension for which the pruning vector applies, and "$x >> y$" means x with the low order y bits truncated. The reasoning behind this choice is as follows. Lines of memory are interleaved among nodes in the system such that the low-order bits of a block address identify the home node of the line. A node address consists of n $\log_2 k$-bit digits. Consider the pruning cache at node i. All level-1 entries in pruning cache i are for lines cached in nodes for which node i is the first-level parent node (see Figure 2.3(b)). Therefore, all the level-1 entries are for lines with the same low-order $\log_2 k$ bits (the low-order bits of i's address). All level-2 entries in pruning cache i are for lines cached in nodes for which node i is the *second*-level parent node. Therefore, all the level-2 entries are for lines with the same low-order $2\log_2 k$ bits (again the low-order bits of i's address). The same reasoning applies to level-3 entries, *etc.*

The placement function in Equation (6.19) simply shifts the line address sufficiently to ignore the low-order bits that are all the same for the given dimension. Thus, physical block addresses in the range 0 to X map into set numbers 0 to $(X>>log_2 k)$ for level-1 PC entries, 0 to $(X>>(2\log_2 k))$ for level-2 PC entries, *etc.* These set numbers are then folded onto the number of actual sets in the pruning cache, as in a conventional cache. A given line's entries for multiple dimensions will not conflict with each other, as the line address is shifted over one digit for each dimension.

The multi-dimensional nature of the mapping creates an interesting problem. If the number of dimensions minus one is greater than the associativity of the pruning caches, then no matter how large the pruning caches are, they can still have conflict misses. This is because the level-1 pruning cache entries will compete with the level-2 pruning cache entries, and so on, up to the level $n-1$ pruning cache entries. If the associativity of the pruning caches is greater then or equal to $n-1$, then a sufficiently large pruning cache is guaranteed to have a hit rate of 1.

3.2. Replacement policy

Given that conflicts will occur within a pruning cache, there must be some replacement policy that determines which entries have priority. All simulations so far have used LRU replacement, which is easy to implement with a set size of 2. Another reasonable choice would be to give priority to pruning vectors closer to the root (*i.e.* entries for higher dimensions). This is because the penalty for a pruning cache miss (broadcasting the invalidation to the subtree beneath the pruning cache) is greater, the nearer the miss is to the root. LRU may approximate this, as vectors for higher dimensions would tend to be accessed more frequently, and we want rarely used vectors to drop out of the cache. A final choice would be to simply replace the entries randomly.

3.3. Policy for handling all-zero vectors

Another important issue is the policy regarding all-zero pruning vectors. When a line is invalidated, all pruning vectors for the line are set to zero. In the ideal case of perfect hit rates, an all-zero vector is almost never needed, as any invalidation for the corresponding line would be pruned by a pruning cache at a higher level or the top level directory.[8] This might suggest that

[8] The exception is when a shared line is cached at the root of some subtree, but nowhere else in the subtree. In this case, the node at the root of the subtree needs an all-zero pruning vector because its parent would correctly propagate an invalidate for the line to it, but it should not propagate the invalidate to any of its children.

all-zero vectors should have the lowest priority in pruning caches, never replacing a nonzero entry. However, in the realistic case where pruning caches do miss, all-zero vectors may provide a firewall to prevent errant propagation of an invalidation that missed in a higher level pruning cache. The issue at hand is whether the positive benefit of the "firewall" outweighs the negative effect on hit rates caused by placing all-zero vectors in the pruning caches.

All-*one* pruning vectors also may warrant special attention. An all-one pruning vector can always be thrown out of a pruning cache with no performance penalty, since pruning cache misses result in the assumption of an all-one vector. The only disadvantage of this policy might be increased complexity to detect and invalidate all-one vectors. I do not expect that the additional pruning cache space provided by invalidating all-one vectors would have a significant effect on performance, but an implementation could choose to do so if convenient.

3.4. Simulation of replacement and all-zero-vector policies

This section presents a simulation study aimed at determining the best replacement and all-zero-vector policy. Three different replacement policies are simulated: *LRU, DIM* and *Random.* In addition, three different all-zero-vector policies are simulated: *AlwaysPlace, SometimesPlace* and *NeverPlace.* The policies differ regarding what they do when an invalidation passes through a node (which provides the knowledge that an all-zero pruning vector now *exists* for each appropriate dimension for the corresponding line at this node). The AlwaysPlace policy places the all-zero vector(s) into the pruning cache, possibly displacing other pruning vectors. The SometimesPlace policy places the all-zero vector(s) into the pruning cache only if they do not cause a non-zero pruning vector to be replaced. Thereafter, the all-zero vectors compete for space in the pruning cache just like any other vectors. The NeverPlace policy does not put the all-zero vectors into the cache at all, assuring that they will not compete for space with non-zero vectors. The replacement and all-zero-vector policies are orthogonal, so the three choices for each policy lead to nine combinations, which are each simulated.

Simulations are run for a 256-node system (k=4, n=4). Pruning caches are all 256 entries. Four different workloads are simulated. As in Section 1, each workload includes a 64K read-only code segment, an 8K private data segment (P_{write}=30%), and an additional data segment that determines the characteristic behavior of the workload.

- The first workload uses a fixed 1MB segment shared among all processors. P_{write} is 30%, causing the data to be written fairly often.
- The second workload also uses a fixed 1MB shared segment, but uses a P_{write} of only 2%, corresponding to data that is mostly read.

- The third workload uses 4-way sharing with a local segment size of 32K. Each line is contained in the virtual space of four different processors, for a total global physical segment size of $N(8K) = 2MB$. P_{write} is 15%. Sharing is arranged along the root dimension, which spreads out sharers among different leaf rings.
- The fourth workload is identical to the third, but with sharers arranged along the leaf dimension, which concentrates sharers on the same leaf rings as much as possible. This data partitioning in this workload is designed to exploit locality.

Figure 6.13 presents the results for the first workload (large shared data segment with lots of active sharing). Part (a) shows the invalidation traffic for each policy and part (b) shows the response time at the second level caches. The differences in invalidation traffic among the various policies result in noticeable, but small differences in response time. Nevertheless, reducing invalidation traffic is always a worthwhile goal, provided that it does not unduly increase cost or complexity.

For each replacement policy, invalidation traffic is lowest when all-zero vectors are never placed into the pruning caches, and highest when all-zero vectors are always placed into the pruning caches. This indicates that the negative effect of pollution caused by all-zero vectors outweighs their positive effect as a firewall against runaway invalidations. Invalidation traffic is also lowest for the DIM replacement policy and highest for the Random replacement policy. This confirms the intuition that pruning cache misses for high dimensions should be avoided, due to their large potential cost in invalidation traffic.

The overall pruning cache hit rate for *primary* references (those for invalidations that are still within the proper subset of processors, as opposed to runaway invalidations) is shown in part (c). Surprisingly, this is *lower* for DIM than for LRU or Random. The reason for this can be seen by examining parts (d), (e) and (f), which show the pruning cache hit rates broken down by dimension. For DIM, the hit rate is higher for the highest dimension and decreases as the dimension decreases, as we would expect. For LRU and Random, this is not the case.

Figure 6.14 presents the results for the second workload (large shared data segment that is mostly read). Figure 6.15 presents the results for the third workload (four-way sharing, with sharers arranged along the root dimension). The results for both of these workloads are quite similar to those for the first workload. The DIM replacement policy and the NeverPlace policy for all-zero vectors result in the lowest invalidation traffic. Differences in invalidation traffic lead to a small, but noticeable, difference in response time. The overall pruning cache hit rate is lowest for DIM, but the skewing of pruning cache hits towards the high-dimensional accesses results in better performance.

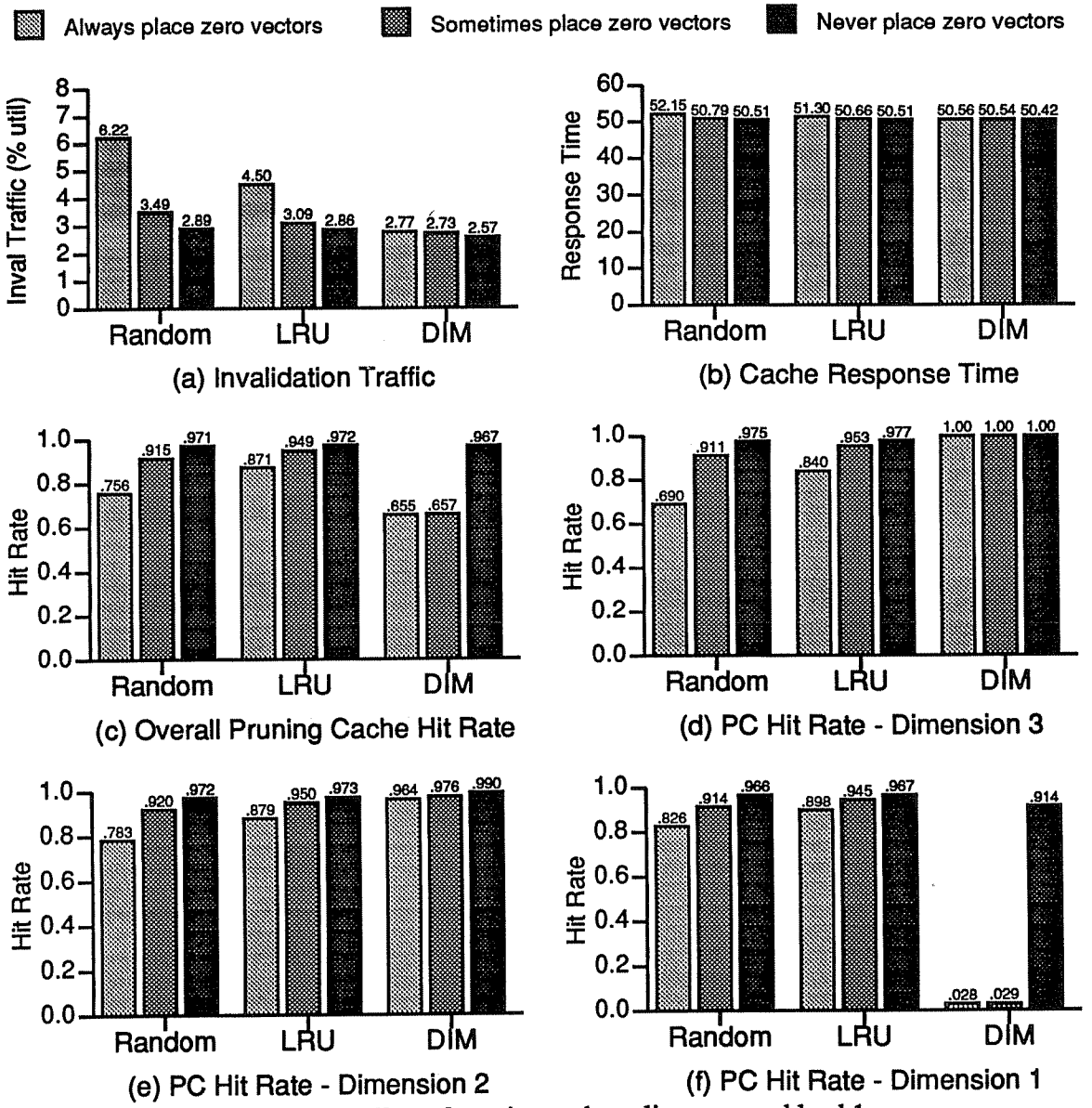


Figure 6.13: Effect of pruning cache policy — workload 1

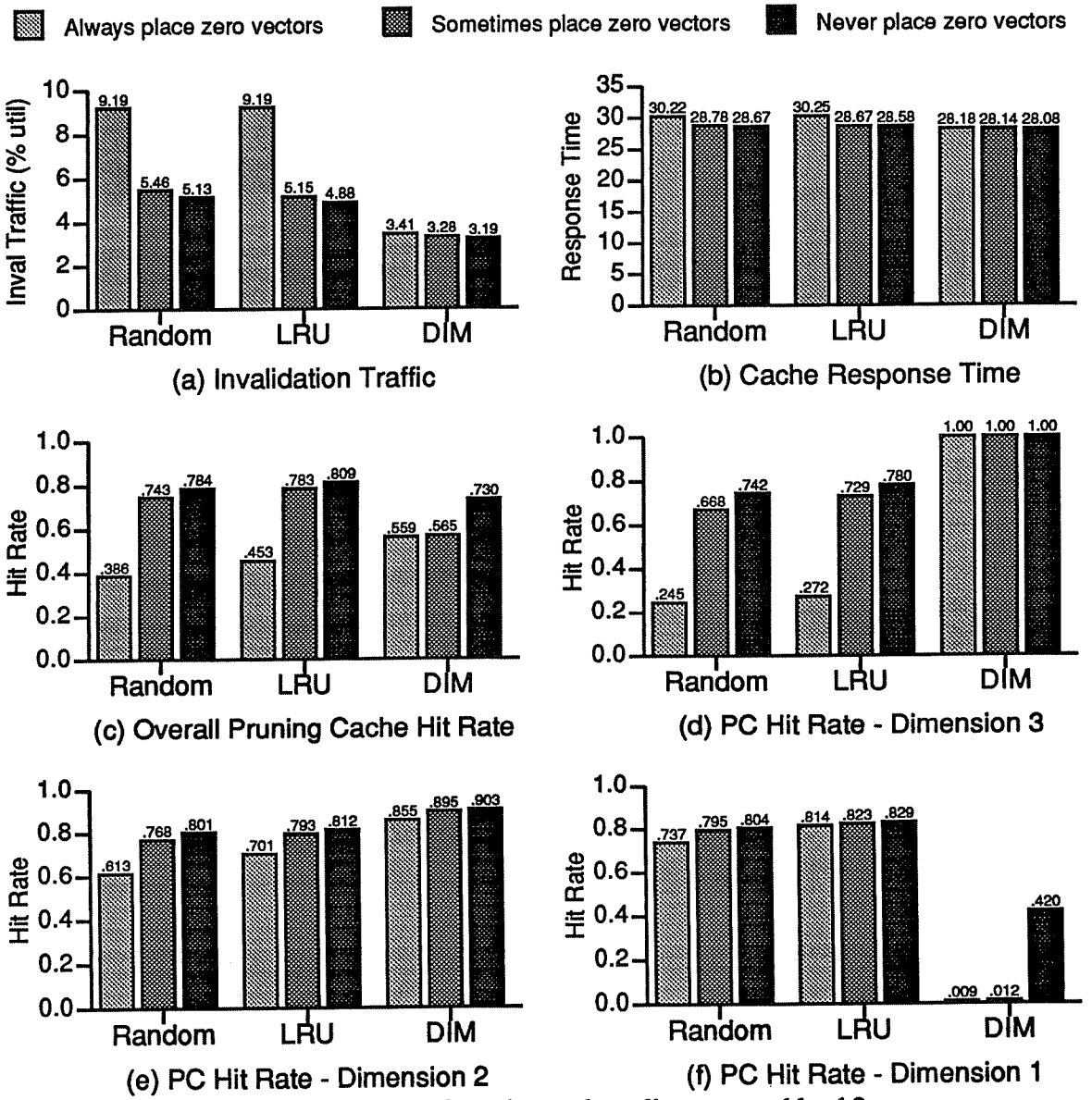


Figure 6.14: Effect of pruning cache policy — workload 2

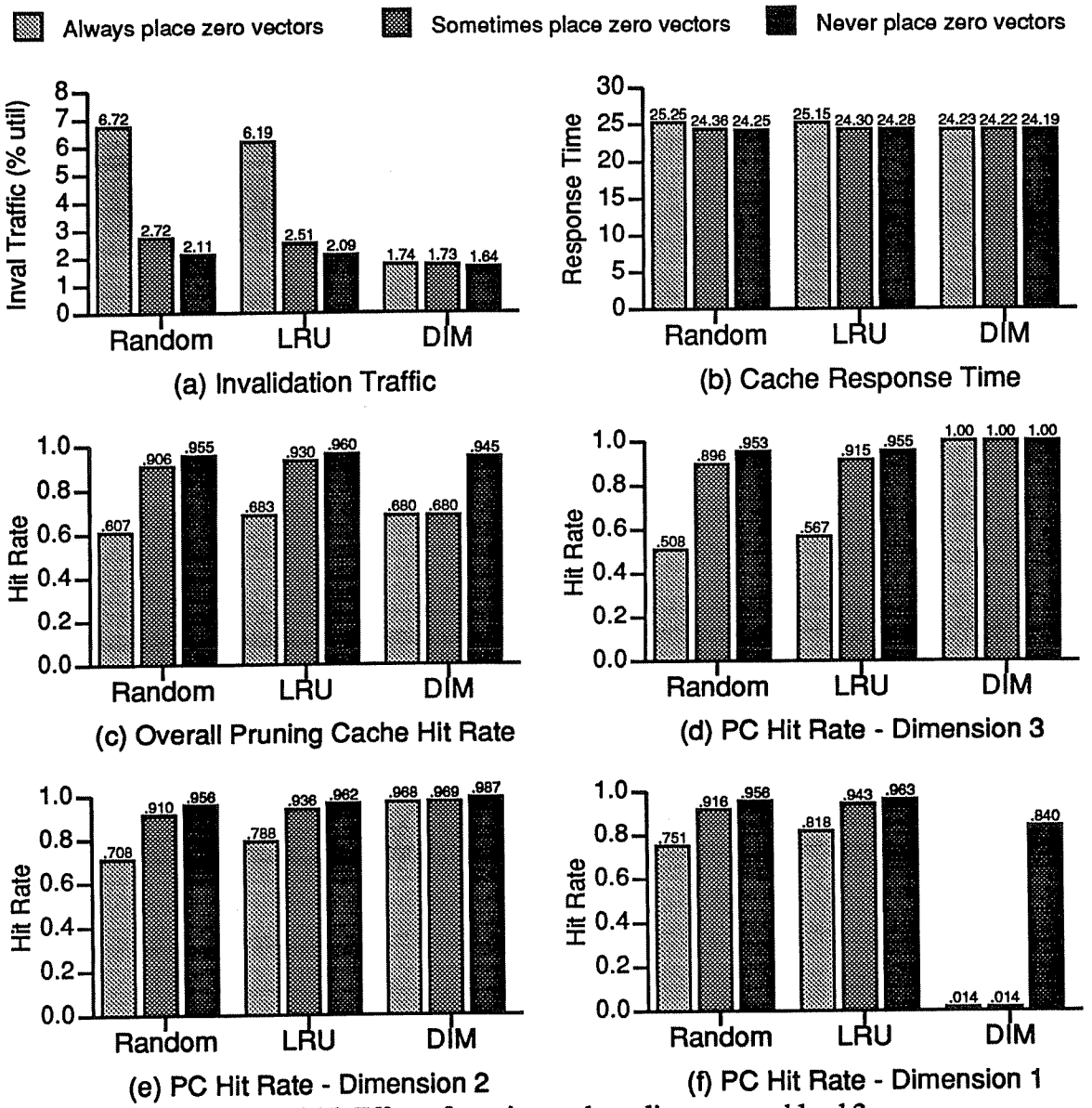


Figure 6.15: Effect of pruning cache policy — workload 3

Figure 6.16 presents the results for the fourth workload (four-way sharing, with sharers arranged along the leaf dimension). The data partitioning in this workload makes the pruning caches perform well enough that changing the policy results in almost no change in performance.

The differences in pruning cache policy resulted in significant differences in invalidation traffic for these workloads, but only small differences in overall system performance (as measured by cache response time). However, the DIM/NeverPlace policy is no more difficult to implement than any of the others, and hence should be used. For larger system sizes, the

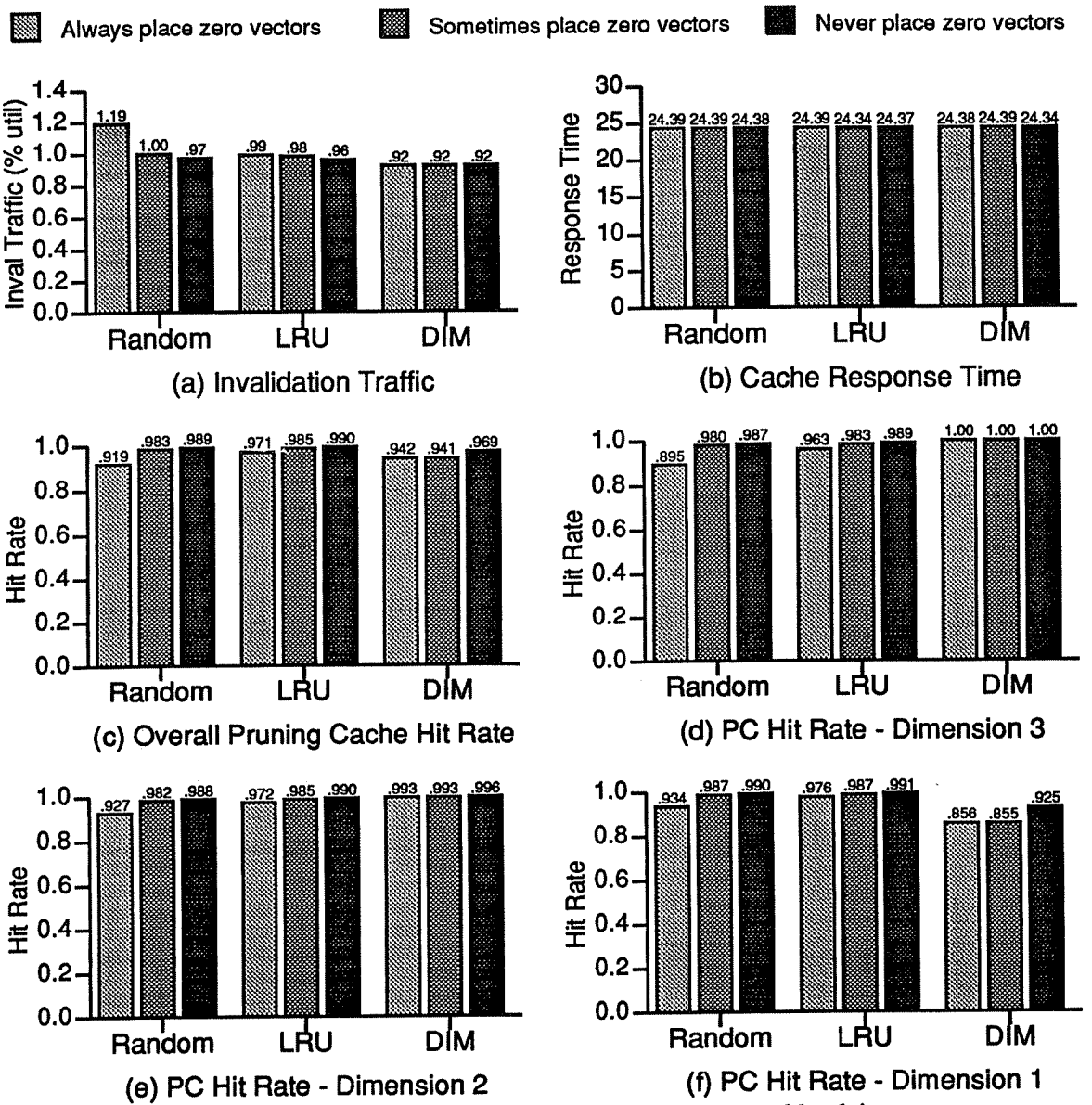


Figure 6.16: Effect of pruning cache policy — workload 4

reduction in invalidation traffic would likely have a larger impact on cache response time (or put another way: in larger systems, invalidation traffic is heavier, so increased invalidation traffic is more likely to cause serious performance degradation).

4. Validation of Analysis

The traffic caused by an invalidation in a pruning-cache-based system was derived in Chapter 5, Section 2.1. This section presents simulation results validating that analysis.

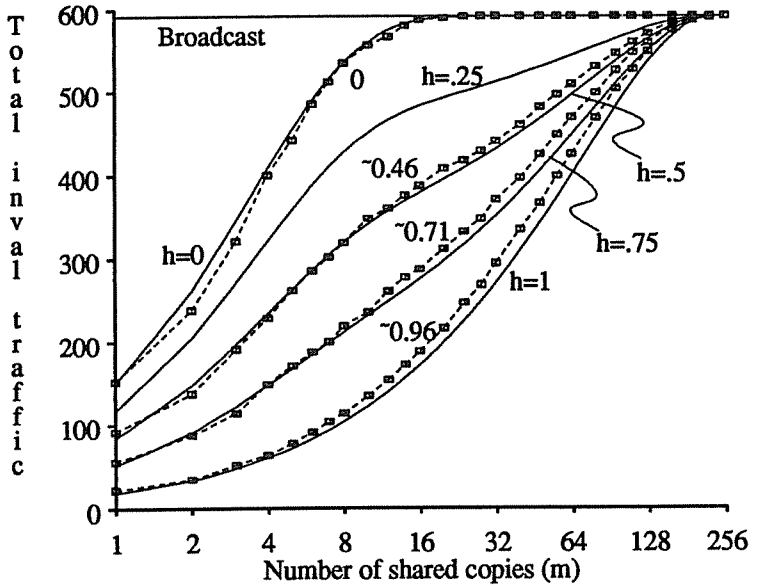
Equation (4.4) computes the traffic for an invalidation, given the hit rate of the pruning caches and the number of shared copies of the line being invalidated. To validate this equation, the simulator was modified to keep statistics on all invalidation operations, determining the pruning cache hit rate, the invalidation traffic and the number of lines being invalidated for each invalidation operation.

There are several difficulties with this procedure. First, it is very difficult to know the exact value of m for a given invalidation operation. Not only is this information not available during normal simulation (requiring, in essence, the simulator to implement a full-width directory on the side), but events involving shared lines do not occur atomically, so at any given time there may be processors in the process of acquiring a shared line. The result is that the value of m that is recorded for a given invalidation operation is only approximate.

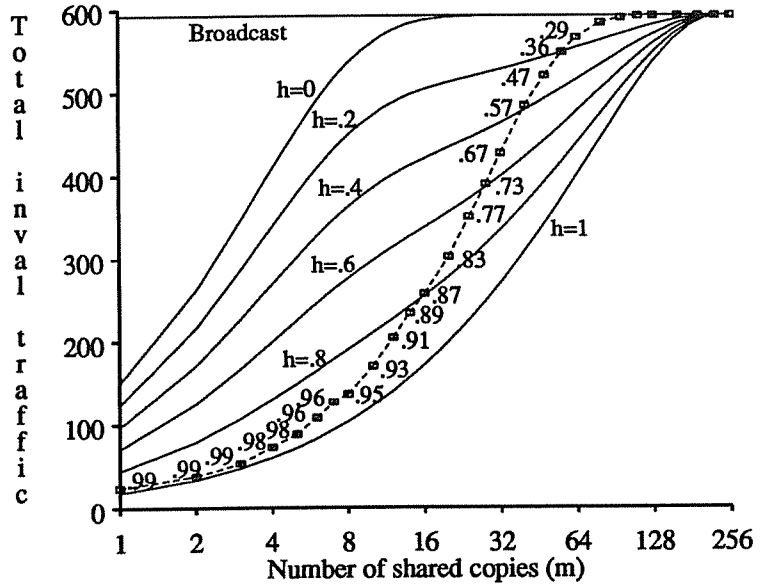
The second major difficulty is that it is not possible to determine if a given pruning cache reference is for a runaway invalidation (in which case it should not be counted in calculating the realized hit rate). It is conservatively assumed that an invalidation message that missed at the pruning cache above is now a runaway invalidation (and this information *can* be maintained as the invalidation propagates down through the system). Pruning cache references due to runaway invalidations are not counted when computing h. In actuality, the invalidation message might have reached its current position in the tree even if it had hit at the pruning cache above, in which case it is not a runaway. This problem causes the measured h to be an approximation, as well as the measured m.

Finally, the assumption of a constant pruning cache hit rate for all references is unrealistic. As we saw in Section 3, the hit rate can vary between dimensions, in which case Equation (4.4) will not predict the correct invalidation traffic. In addition, we expect pruning cache hit rates to vary according to the reference behavior of the data. Pruning cache references for actively shared lines should have higher hit rates than references for inactive lines. This means that during a single simulation, the realized pruning cache hit rate will vary according to what data is being invalidated and will likely vary with m. Despite these difficulties, simulation results are able to confirm the general validity of the invalidation traffic equation. The general conclusion is that the equation is accurate, given the assumptions, but that actual pruning cache hit rates are not constant across an application, varying by dimension and by line.

Figure 6.17 presents some of the validation results. The solid lines represent the invalidation traffic predicted by the equations, and the dashed lines represent simulation results. A 256 processor system (k=4, n=4) was simulated, and single shared 64K data segment was used, resulting in a complete range of m (the number of shared copies of a line upon invalidation)



(a) Constrained Hit Rates



(b) "Natural" Hit Rates

Figure 6.17: Validation of pruning cache invalidation traffic equation

during the simulation.

In part (a), the simulator is artificially constrained to give an approximately uniform hit rate, *h*, for all pruning cache references. A very large pruning cache is used in order to provide a

natural hit rate of close to 1. On a pruning cache reference, the pruning cache is made to artificially miss with probability 1–*h*, and allowed to work normally with probability *h*. Since the natural hit rate is just under one, this results in an overall hit rate of just under *h*. The overall observed pruning cache hit rate is shown for each of the four simulation curves. The hit rate was higher for small values of *m*, and lower for large values of *m*, matching the intuition that the hit rate should be lower for less frequently accessed data. The invalidation traffic measured in the simulator closely matches the traffic predicted by Equation (4.4).

In part (b), a small pruning cache is used (only 28 entries), and the hit rate is not artificially constrained. Results for a single simulation are presented and the hit rates are shown separately for each value of *m*. The small pruning cache size leads to an *overall* hit rate of 0.457. It can be clearly seen, however, that the hit rate varies significantly with *m*. In this case, because larger *m* implies a longer time time between writes (leading to lower temporal locality), the hit rate is lower for larger *m*. Results from simulations with more realistic workloads, consisting of multiple data segments with different access characteristics, display similar results. Pruning cache hit rates vary with *m* according to the various data access patterns, resulting in *h* versus *m* curves with local maximums and minimums.

Results in part (b) appear discouraging, because the measured invalidation traffic for medium and large values of *m* is somewhat higher than predicted by Equation (4.4) for the measured values of *h*. This is because *h* varies by dimension as well as by *m*. With LRU replacement and the small pruning cache size, hit rates are higher for *lower* dimension and lower for *higher* dimensions. Thus a disproportionate number of misses happen at the highest dimension, leading to greater invalidation traffic. When the same experiment is run with the DIM replacement, the opposite occurs: invalidation traffic is somewhat lower than predicted by Equation (4.4). Even with this source of error, however, the difference between predicted and measured invalidation traffic is at most about 10%. In practice, pruning caches should be large enough that hit rates are high for *all* dimensions, leading to performance much like the bottom curve in Figure 6.17(a).

5. Implementing Broadcast

There are two instances in which the pruning cache protocol employs broadcast:[9] when a shared read request is decombined on a ring, and when an invalidation is sent to more than a

[9] To be precise, *multicasts* are used, of which full broadcasts are a sub-case in which all nodes receive the multicast. I use "broadcast" here in this looser sense.

single processor. The primary difficulty with broadcasts is that, like other operations, we must know when they have completed, and this becomes harder when more than two nodes are involved. This section briefly discusses some of the difficulties presented by broadcast operations and suggests possible solutions.

The broadcasts used in read decombining are easier to deal with because the recipients can respond as soon as the broadcast message is received. In addition, the acknowledgements are local to a single ring (they simply serve the purpose of letting the sender on the ring discard the packet); nothing must be returned up to a higher level. The acknowledgements are required for fault tolerance and for queue overflow handling.

In the absence of hardware errors, ring broadcast could be implemented using a simple bit mask. The mask would be set by the source to indicate the set of nodes on the ring for which the packet is intended. With conventional blocking networks, the packet would simply make its way around the ring, blocking when necessary to allow queues to drain, and allowing each marked node to retain a copy. Normal deadlock prevention techniques, such as virtual channels [Dall87], must be used in this case.

Pipelined-channel rings as discussed in this thesis could also implement broadcast. Nodes would clear their respective bits in the bit mask as the packet traversed the ring. Any node that could not accept the packet due to queue overflow would not clear its bit. When the packet arrived back at the source, if at least one node did not retain a copy, then the packet would be retransmitted with the modified bit mask. The SCI protocol implements broadcast as one of its non-coherent transactions [IEEE92]. Rather than use a full bitmask, a packet that cannot be accepted at a node is immediately converted to a negative echo, disallowing further downstream nodes from receiving the packet on the first pass. Upon retransmission, the packet is sent to the node that could not accept the packet for resumed broadcasting.

A potential problem arises when broadcasting on a pipelined-channel ring if a hardware error occurs (detected by the use of a CRC and hardware timers). Since the modified bit-mask is now lost, there is no record of which nodes successfully received the packet, and thus which nodes must receive the retransmission. This problem exists for point-to-point transmission as well, however, it is not unique to broadcasts. SCI simply traps to software to handle this presumably rare event [IEEE92]. The same can be done when a broadcast transmission fails, although it may increase the complexity of the software recovery.

Broadcasts that are used for invalidations pose a more difficult problem, because the recipients on a ring cannot, in general, respond directly after receiving the invalidation. If they are

not leaves in the broadcast tree, then they must, in turn, broadcast the invalidation beneath them, wait for acknowledgements, and *then* acknowledge their parents. This requires that state be saved over an extended period, during which time other operations must be processed. Care must be taken not to allow deadlock to arise.

As mentioned in Chapter 5, Section 2.2, the pruning vector used when transmitting an invalidation onto a ring is also the information needed to combine the acknowledgements, and can be saved either in an auxiliary buffer or in the pruning cache itself. The latter choice requires that all-zero vectors always be placed in the pruning caches, which we saw in Section 3 was a bad idea. Therefore, I suggest the use of a separate *outstanding invalidation buffer* (OIB) at each node. In either case, however, it is important that deadlock be avoided.

One possible way to prevent deadlock would be to structure the OIB according to dimension, providing separate space for level 2 entries, level 2-3 entries, level 2-4 entries, *etc.* This removes cyclic dependencies that might otherwise arise for OIB space. Invalidations on level 1 rings do not require OIB entries (the invalidation is implicitly acknowledged when it successfully completes its circuit around the ring). OIB entries for level 2 rings must wait for receipt of all level 1 acknowledgements, and therefore do not depend upon allocation of OIB space at any node. OIB entries for level 3 rings must wait for receipt of all level 2 acknowledgements, and therefore depend only upon allocating level 2 OIB space, *etc.* Note that this is only an issue in in systems with 4 or more dimensions, as acknowledgement collection is not needed on the level 1 rings, and the top level pruning vectors are kept in the main memory directories and thus are always available for collecting invalidations.

The size of an OIB can be quite small, as evidenced by the following back-of-the-envelope analysis. Assuming a pruning cache hit rate of close to 1, we need only about $(n-2)$ OIB entries for every copy of a line being invalidated. If we assume a fraction f_W of references are shared writes, then when all processors perform one reference, there will be approximately Nf_W shared writes, each invalidating a maximum of $\left\lceil \frac{1-f_W}{f_W} \right\rceil$ copies on average. This results in a total of approximately $(Nf_W)\left\lceil \frac{1-f_W}{f_W} \right\rceil (n-2)$ total OIB entries or $(1-f_W)(n-2)$ OIB entries per node. Several times this number could easily be implemented, causing OIB overflow to be rare. OIB overflow could thus be handled by a software trap.

OIB overflow could alternately be handled in hardware, by simply refusing invalidation requests when the OIB was full. This would essentially block the parent of the refusing node from transmitting anything else in the corresponding dimension. However, due to the structuring

of OIB space, the events that must occur to free up the needed OIB space for the parent's invalidation request involve receiving acknowledgement packets from children in lower dimensions, so the temporary blockage should not lead to deadlock.

6. Comparison of Coherence Mechanisms' Storage Overhead

It has already been argued (in Chapter 5, Section 2.1) that pruning caches scale in size as $O(n)$. However, the absolute amount of storage required to implement pruning caches is also important. This section explores the required storage for both pruning caches and several other directory schemes. Section 7.1 derives formulas for the required storage of each scheme, given the distribution of the cached data. Section 7.2 presents results for each scheme for several different sharing distributions as the system size is varied. Section 7.3 briefly discusses storage overhead in light of the results.

6.1. Analysis

The formulas derived in this section compute the total storage, in bits, necessary to keep track of all cached data in the system. The following values are used in the analysis: N is the number of processors, P is the size of a processor pointer ($\lceil \log_2 N \rceil$), T is the size of a tag in a pruning or directory cache, M is the memory per processor (in lines), C is the cache space per processor (in lines), n and k are the network dimensionality and radix (relevant only to pruning cache systems), and $f(m)$ is the fraction of lines in a cache that are shared by m processors.

Note that the storage overheads computed here represent the total directory storage *required* to keep track of all cached lines, *given* a particular sharing distribution ($f()$). The total amount of storage actually *built* into the system must be sufficiently large to keep track of the cached data under all expected distributions, and also allow for an uneven usage of memory lines from different memory modules.

For each coherence scheme, both the memory storage and the cache storage are computed. The memory storage corresponds to the directory information associated directly with each main memory line, and is assumed to be built with the same technology as main memory. The cache storage corresponds to cached directory information, and is assumed to built with faster/more expensive memory, which is why it is broken out separately.

The storage overhead for a full width directory consists simply of the pointers associated with each line of main memory:

$$memory\,(\text{full width}) = N^2M \tag{6.2}$$

$$cache\,(\text{full width}) = 0 \tag{6.3}$$

Similarly, the storage overhead for limited pointer directories [Agar88] consists of the pointers associated with each line of main memory:

$$memory\,(Dir_iB) = NMiP \tag{6.4}$$

$$cache\,(Dir_iB) = 0 \tag{6.5}$$

Dir_iB reduces the amount of information maintained by the coherence protocol by not fully keeping track of shared data. A more fair comparison would be to consider $Dir_ivector$, which uses i directory pointers, but falls back on a cache of full-width vectors to keep track of lines with more than i sharers. The memory overhead is the same as for Dir_iB, and the cache overhead is now

$$cache\,(Dir_ivector) = \sum_{m=i+1}^{N} Cf\,(m)\,(N+T) \tag{6.6}$$

The directory cache scheme [O'Kr90, Gupt90] associates no directory information with main memory, but use two kinds of caches, one with i processor pointers per entry and one with full width vectors. The storage overhead is thus

$$memory\,(\text{directory cache}_i) = 0 \tag{6.7}$$

$$cache\,(\text{directory cache}_i) = \sum_{m=1}^{i} Cf\,(m)\,(iP+T) + \sum_{m=i+1}^{N} Cf\,(m)\,(N+T) \tag{6.8}$$

The dynamic pointer allocation scheme [Simo91] uses a single pointer for every cached line, independent of the sharing distribution. Associated with each line of main memory is a pointer into the local pointer cache (assumed to have C entries). Each entry in the pointer cache contains a processor pointer and a pointer back into the local memory. The total storage overhead is thus given by

$$memory\,(\text{dynamic}) = NM\,(log_2C) \tag{6.9}$$

$$cache\,(\text{dynamic}) = NC\,(P + \log_2M) \tag{6.10}$$

The pruning cache scheme requires a single processor pointer/pruning vector with each line of memory, and pruning caches at each node. The memory overhead is given by

$$memory(PC) = NMP \tag{6.11}$$

Assuming random distribution of shared copies, as in the analysis of Chapter 5, Section 2.1, the number of pruning cache entries needed to keep track of m cached copies of a line is

$$PC\_entries(m) = \sum_{i=1}^{n-1} k^{n-i} \left[1 - \frac{\binom{k^n - k^i}{m}}{\binom{k^n}{m}} \right] \tag{6.12}$$

The required cache overhead for a pruning cache system is thus

$$cache(PC) = \sum_{m=1}^{N} C f(m) PC\_entries(m) (T+k) \tag{6.13}$$

If the ownership protocol is used, then only truly shared lines require pruning cache entries, and the required cache overhead is

$$cache(PC_{own}) = \sum_{m=2}^{N} C f(m) PC\_entries(m) (T+k) \tag{6.14}$$

6.2. Results

Several assumptions must be made in order to present actual storage overheads. I assume the use of 64 byte lines, 256 kilobyte caches, and 32 megabytes of memory per processor. All cache lines are assumed to be occupied. The tag size used for pruning caches and directory caches is 23 bits. This assumes a 40 bit physical address and 2-way set associative pruning/directory caches with the same number of entries as the data caches (4096). Storage overhead is given as the percentage increase in memory, for both main memory and cache.

I assume that some fraction of lines in a processor's cache are private, some fraction of lines are globally shared, and the rest are shared to some intermediate extent. The number of processors that share each of these lines is specified by a geometric distribution (truncated at a maximum of N processors). The parameter of the geometric distribution, α, is approximately (because the distribution is truncated) the mean number of processors sharing each line. An α of 1 implies that all the shared lines are in fact private. An α of ∞ implies that a shared line is equally likely to be shared by any number of processors. Note that the more widely lines are shared, the fewer unique lines of memory are cached; if all data was globally shared, a total of C lines would be cached; if all data was private, a total of NC lines would be cached.

Results are given for three different sharing distributions. In the "light" sharing distribution, 80% of the lines are private, 18% are shared with $\alpha = 1.5$, and 2% are global. In the "medium" sharing distribution, 50% of the lines are private, 48% are shared with $\alpha = 2$, and 2% are global. In the "heavy" sharing distribution, 50% of the lines are private, 40% are shared with $\alpha = N^{1/2}$, and 10% are global.

Figure 6.18 shows the storage overhead for full width directories. Although acceptably low for a system size of 64, the overhead grows unacceptably as system size increases, reaching 100% of memory at only 512 processors.

Figure 6.19 shows the storage overhead for *Dir$_i$vector* (limited pointer directories with vector cache backup). In part (a), one pointer is allocated per directory entry. Memory overhead is quite low, and cache overhead is acceptably low for system sizes up to 1024. For a 4096-node system, cache overhead becomes excessive (75% for the medium sharing case). Overhead for the 32K-node system is extremely high. In part (b), three pointers are allocated per directory entry. This makes the main memory overhead much larger, but perhaps still acceptable. The cache overhead does not become excessive until 32K nodes. In part (c), eight pointers are allocated per directory entry. The main memory overhead is now quite large, and is almost entirely unused (most memory lines are not cached at all, much less by eight processors), which is not acceptable. The overhead for the vector cache still becomes quite large under the heavy sharing distribution for 32K processors. While, the storage overhead for limited pointer directories with a few pointers may be acceptable for system sizes up to a few thousand, it is clear that beyond that size the storage requirements become too great.

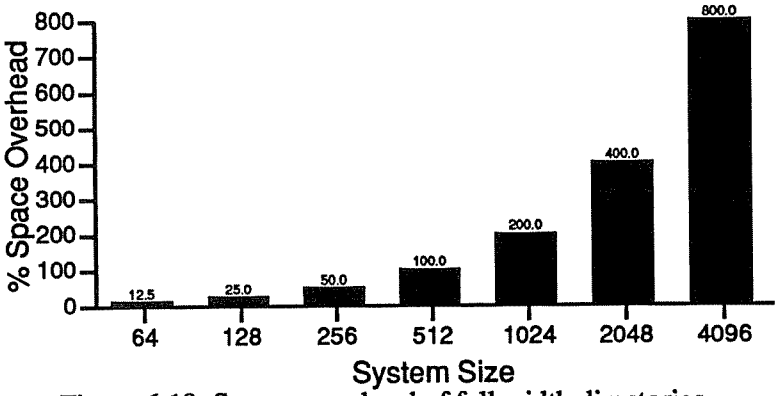
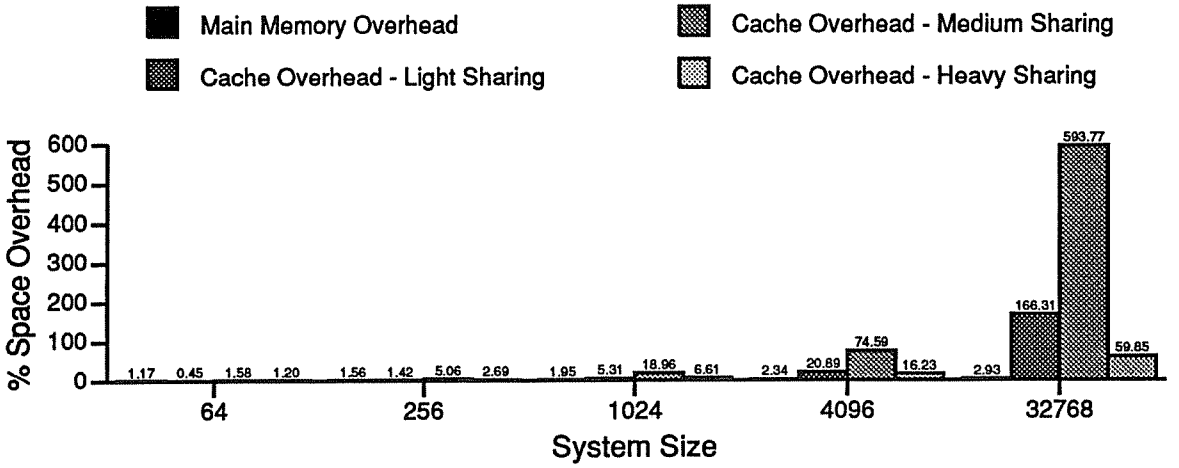
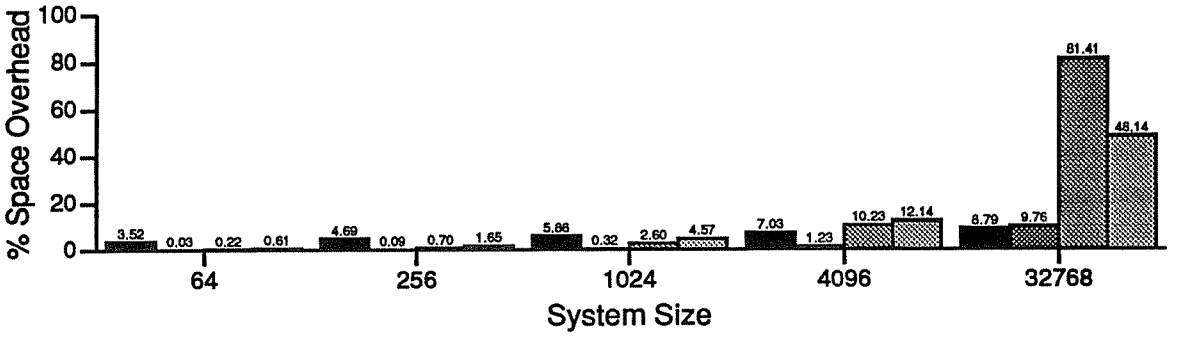


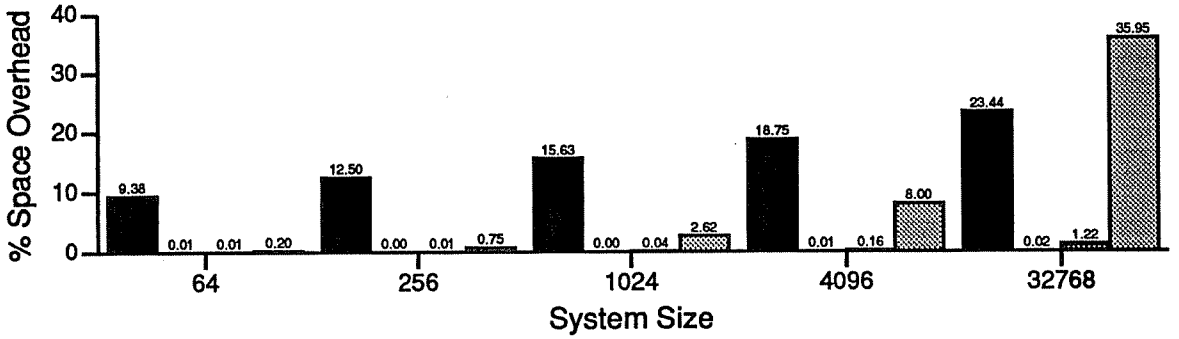
Figure 6.18: Storage overhead of full width directories



(a) One pointer per directory entry



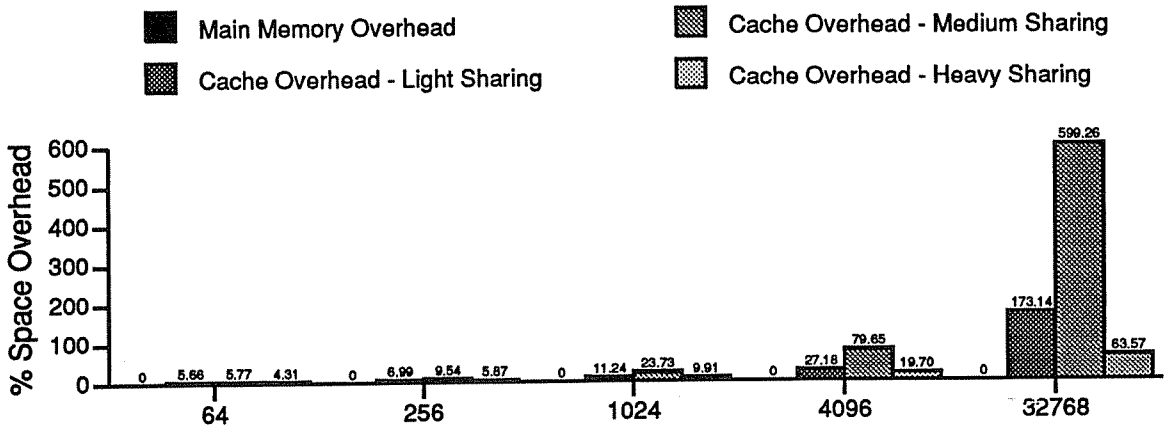
(b) Three pointers per directory entry



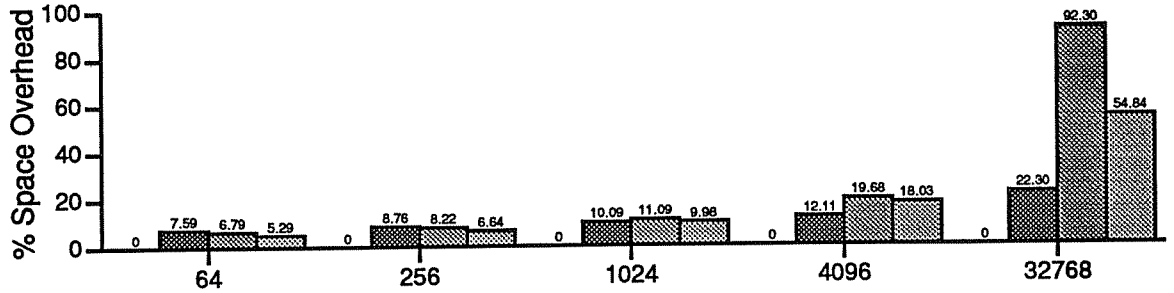
(c) Eight pointers per directory entry

Figure 6.19: Storage overhead of limited pointer directories, *Dir$_i$vector*

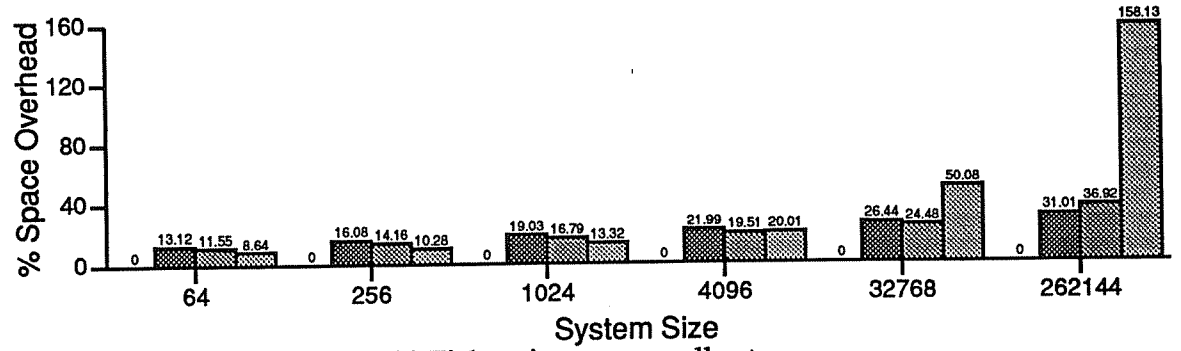
Figure 6.20 shows the results for directory caches. No main memory overhead exists for this scheme because all directory information is cached. Parts (a), (b) and (c) assume that the small directory entries contain 1, 3 and 8 pointers, respectively. With 1 or 3 pointers in the small directory entries, the cache of full vectors becomes excessively large for system sizes in the 4096



(a) One pointer per small entry



(b) Three pointers per small entry



(c) Eight pointers per small entry

Figure 6.20: Storage overhead of directory caches

to 32K processors range. Unlike with limited pointer directories, however, it may be reasonable to use a larger number of pointers for the small directory entries when using directory caches, because the entries are not duplicated for every line of main memory. The growth in cache

overhead is much more gradual in part (c) where 8 pointers per entry are assumed. The overhead is still fairly significant, however, exceeding 20% for a 4096-node system.

Figure 6.21 shows the results for dynamic allocation. The memory and cache overheads for this scheme are smaller than those of the previous two schemes, and the scaling behavior is also much better. The overhead remains quite small for extremely large systems.

The required storage does not depend upon the extent of sharing, but it *does* increase if lines from various main memory modules are cached non-uniformly. This potential non-uniform usage of memory is not captured in the analysis of this section, and is the reason that actual implementations must include more storage than is shown as necessary here. If non-uniform usage results in a shortage of processors pointers at a given node, then the protocol invalidates existing cached lines in order to free up pointers. Available processor pointers in the pointer cache are maintained using a free list, however, so the cache can be completely utilized before having to invalidate cached lines.

Figure 6.22 shows the results for pruning-cache directories. Part (a) assumes the basic pruning cache protocol, and part (b) assumes the PC_{own} protocol, which only creates pruning cache entries for lines shared by two or more processors. The storage overhead for pruning caches is similar to that for dynamic pointer allocation, a bit higher for the basic protocol and slightly lower for the ownership protocol. The overhead for pruning caches grows slightly more quickly with system size than that for dynamic pointer allocation, but slowly enough that the overhead is still fairly low for extremely large systems.

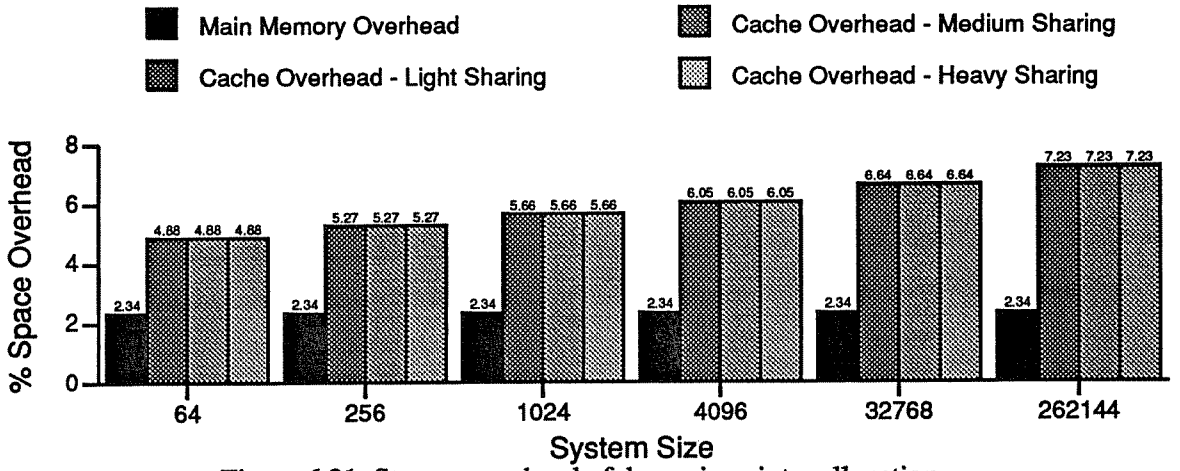


Figure 6.21: Storage overhead of dynamic pointer allocation

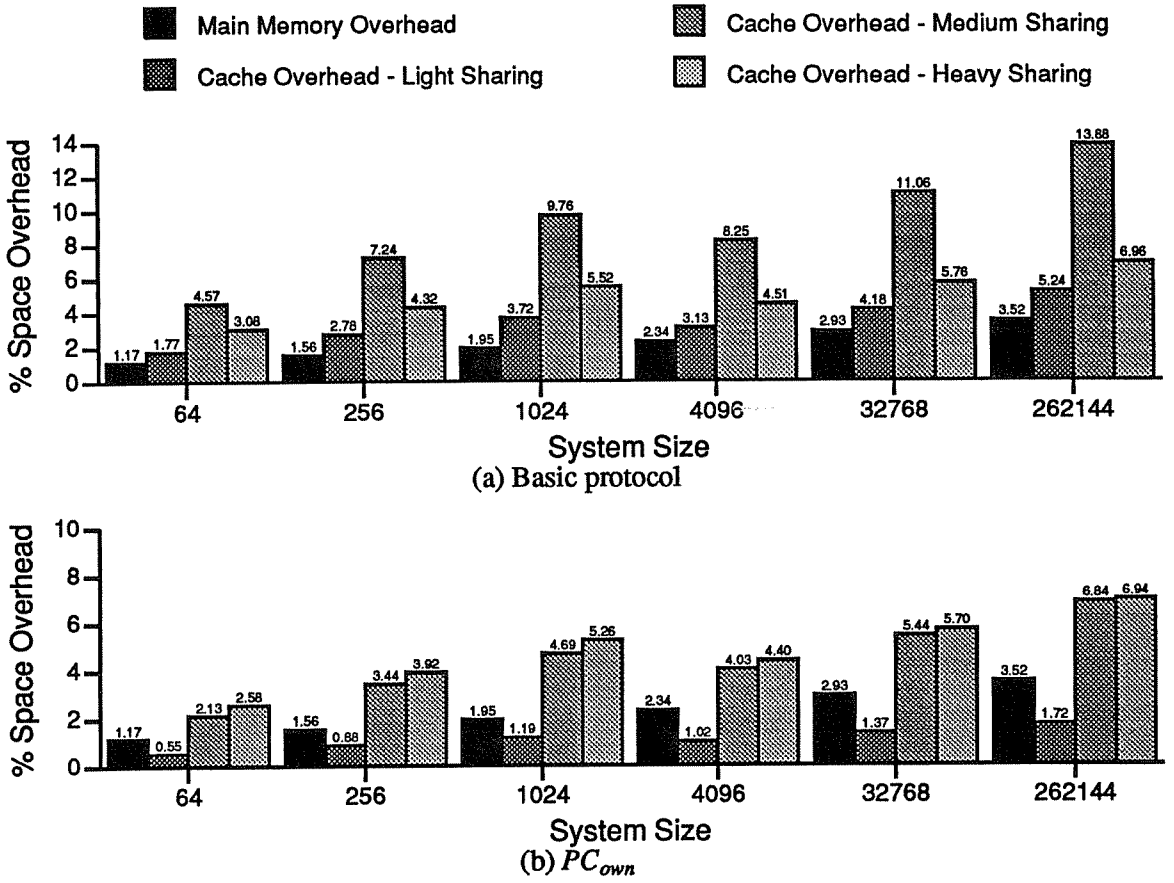


Figure 6.22: Storage overhead of pruning-cache directories

One advantage of pruning caches with regard to storage overhead is graceful degradation in the face of a storage shortfall. Especially if the replacement is prioritized by dimension, running out of pruning cache entries causes only a mild increase in invalidation traffic (because entries for the leaf rings can be thrown out first). This means that pruning caches can be built relatively near the average capacity rather than having to build them much larger to make sure they rarely run out of space.

7. Summary

This chapter has presented a thorough exploration of pruning cache performance and management. Simulation was first used to analyze the performance of pruning caches in the context of an operational system and to compare pruning caches to the similar multi-level inclusion (MLI) scheme. Several modifications to the basic pruning cache protocol were then simulated, as well as various management policies relating to pruning cache replacement and handling of all-

zero pruning vectors. Several other topics were also covered in this Chapter. Simulation results confirming the analysis presented in Chapter 5 were presented. Difficulties in the implementation of broadcast were discussed, and solutions were presented. Finally, the scalability of pruning caches with regard to storage overhead was quantified and compared to several other coherence mechanisms.

Simulation results indicate that pruning-cache-based systems scale well and perform better than MLI-based systems using comparably sized inclusion caches. Pruning caches store information about where data does *not* reside, so the loss of a pruning-cache entry simply results in extra broadcast traffic when (and if) the corresponding data is invalidated. In contrast, inclusion caches guaranteeing MLI store information about where data *does* reside. When an inclusion cache entry is lost, the corresponding data must be invalidated in the subtree beneath the cache. I have compared pruning caches against inclusion caches and found that pruning caches are more cost effective: they can provide higher performance while storing less information.

Pruning caches perform more effectively than inclusion caches because the penalty for increased invalidation traffic is less than the penalty for prematurely invalidating data. However, as was indicated by Figure 5.5, the penalty for pruning cache misses increases with system size. Although it should be relatively easy to keep pruning cache hits very high (>98%) as system size increases, it is likely that when system size reaches the order of one million processors, the penalty for pruning cache misses will exceed the penalty for subtree invalidations. Therefore, the effective scalability of pruning-cache-based systems may be limited to tens or hundreds of thousands of processors. It is likely that many other factors being completely ignored here (power, cooling, fault tolerance, *etc.*) will prove limiting before this size restriction is put to the test, however.

Pruning caches allow data that is passively shared (declared shared, but actually used by only one processor) to remain in data caches even though the directory information may be purged. Simulation results show, however, that performance is improved when the protocol is extended to provide the notion of ownership (PC_{own}). This prevents information regarding passively shared data from entering pruning caches in the first place. It provides the same function for migratory data.

Read-combining can be incorporated into a pruning-cache-based system by extending the coherence protocols. The read combining mechanism successfully eliminates degradation caused by concurrent read contention, but may increase read latency in the absence of contention. In addition, it is incompatible with the PC_{own} protocol, which routes read requests directly to memory. While contention is likely to be introduced by certain types of synchronization, detailed

studies of large parallel applications are needed to determine the extent of actual read contention.

I suggest that read requests in a pruning-cache-based system *default* to direct routing and the PC_{own} protocol, resulting in no combining opportunities but providing lower latency. Accesses to synchronization variables or shared data/instructions that the programmer/compiler thought might suffer from contention could be marked as combinable. These requests would use the combining mechanism, resulting in higher latency, but avoiding possible serialization from read contention. This solution does not require a separate combining network, as was proposed for the RP3 [Pfis85]; it is simply built into the coherence protocol.

With the use of pruning caches, performance is somewhat improved by partitioning work such that sharing takes place along lower dimensions. This is due primarily to better pruning cache performance, but also to increased read combining for workloads which exhibit this behavior. A related topic is that of distributing data such that processors use data whose home memory location is local or nearby. If sufficiently large caches are used, then data automatically migrates to the correct locations, and only shared data results in network traffic. However, since some conflict and capacity misses are likely to occur, it may make sense to allocate some pages of memory *locally* to a node rather than spread across nodes, as has been suggested in this thesis (a similar capability was included in the RP3[Pfis85]). This could be used to store local/private data. Since data sharing involves accesses to the directory information for the data, is may also make sense to allocate shared data in contiguous memory at a node and partition problems to increase locality. However, this approach may increase the probability of hot spots at a particular node's memory. This is an interesting topic of consideration, but is not treated in this thesis.

Simulation results indicate that the DIM replacement policy provides superior pruning cache performance over random and LRU replacement. Results also show that all-zero vectors should not be placed into pruning caches when a line is invalidated. Their pollutive effect outweighs any "firewall" effect that they provide.

The final topic discussed in this chapter was the storage overhead of pruning caches and several other coherence mechanisms. Both the pruning cache and dynamic pointer allocation protocols have a low storage overhead and scale very well with system size. Although not analyzed, the overhead for SCI is also expected to scale well (up to their maximum size of 64K processors), as the storage requirement is directly proportional to the cache and main memory. Limited pointer directories (either cached or attached to main memory) with caches of full width vectors do not scale as well. They may be acceptable for implementations with up to a thousand or so processors.

Chapter 7

Conclusion

I firmly believe that the day will come when (1) multiprocessing is truly mainstream (easy to use, accessible, accounting for a large fraction of total computing) and (2) the most powerful supercomputers (by far) will be comprised of thousands of high performance microprocessors. We are a long way from that day. I also believe that, despite the current dominance of message passing machines, the most common multiprocessor of the future will support the shared memory paradigm. This thesis attempts to take us one step closer to the realization of these beliefs.

Envision connecting ten thousand state-of-the-art microprocessors together. Give each processor its own local memory, and provide a uniform physical address space. Call this the world's most powerful computer. Certainly we can do this *today*. Why would this not work, or would it?

One of the primary challenges would be programming the machine. Writing parallel applications or parallelizing existing applications is *hard* to do in general. Finding the parallelism in a problem, partitioning the data, scheduling the work and managing synchronization well are beyond the ability of current compilers. These problems, although extremely important, are also beyond the scope of this thesis.

Let us assume that the language, compiler and operating system designers solve the above problems (or that some dedicated applications programmers manage the details themselves). Further assume that a program running on the machine actually *uses* the shared address space to effect a non-trivial amount of communication. How does our hypothetical system perform now?

The primary problems that we now have to contend with are throughput in the interconnect and latency of inter-processor communication. Processor caches are almost certainly needed to alleviate the above problems, and thus cache coherence is an issue as well. Current cache coherence mechanisms are simply not feasible (due to storage requirements or performance) for systems of this size.

The throughput problem is exacerbated not only by the large number of processors communicating, but by the greater distance the communication must travel, assuming that the communication is not entirely local. There is really no way around this problem; the interconnection

network must provide the needed throughput.

Communication latency is inherently larger in this system than in current multiprocessors, again assuming that the communication is not entirely local. We should strive both to tolerate latency and to reduce it as much as possible. Reducing communication latency, like providing sufficient interprocessor throughout, depends upon the interconnection network.

Tolerating latency is achieved by overlapping communication with computation. It takes the form of either fetching data before it is needed or switching contexts when needed data is not yet present (or both). It is my belief that the best approach is to allow processors and caches to have many outstanding requests and to provide a rich set of synchronization primitives to coordinate sharing. This is a complex and interesting topic in its own right, but is beyond the scope of this thesis.

The goal of this thesis is to shed light on the issue of multiprocessor scalability, and specifically to address the problems associated with interconnection network performance and cache coherence for very-large-scale multiprocessors. This thesis makes four primary contributions.

- A working definition of *scalability* is provided. The definition is purposefully non-rigorous, but provides useful insight into the behavior of large-scale systems. Using scalability arguments, the case is made for pipelined-channel k-ary n-cube interconnection networks.
- The first performance study of the Scalable Coherent Interface ring is presented. This study is valuable in its own right by helping us better understand a new high-performance interconnect standard, and also serves as a proof of concept for pipelined-channel networks.
- Pipelined-channels are shown to have a large impact on the design of large-scale interconnection networks. They provide higher and more scalable performance than do non-pipelined-channel networks, and significantly change the network design constraints.
- A new cache coherence mechanism, pruning cache directories, is investigated for use in large-scale systems. Through analysis and simulation, the mechanism is shown to be both effective and efficient, providing scalable performance for a broad range of workloads.

The definition of scalability revolves around three metrics: cost, latency and bandwidth. The basic idea is to keep the cost, average communication latency, and traffic through any point from growing too quickly as system size is increased ("system balance" is another way of viewing this task). Using these metrics, it is easy to reason about the behavior of systems as size is increased.

K-ary *n*-cube networks have excellent scaling properties when (1) the radix is held constant, increasing the dimensionality only, and (2) physical constraints such as wire delay, pin limitations and wiring density are ignored. When physical constraints are not ignored, however, scalability is significantly limited, and the optimal radix of the network increases with network size. Pipelined channels allow wire delay to be essentially ignored, thus increasing scalability, but do not remove constraints due to wiring density or pin limitations. Ultimately, all networks are wiring-density constrained for sufficiently large sizes, which, assuming some amount of global communication, will cause communication latency to increase as $O(N^{1/2})$.

Scalability should not be emphasized over performance. Pipelined-channel networks must be feasible to implement and provide good performance in order to recommend them for actual systems. The physical and logical layers of the Scalable Coherent Interface serve as a proof-of-concept for pipelined-channel networks. The design includes all the details needed in a real system and is being implemented today. By demonstrating that the SCI ring provides very high performance, the validity of pipelined channels is established. Not only does a single SCI provide throughput on the order of a gigabyte per second, but the message transmission latency on a ring is shown to be lower than that on a typical high-performance shared bus.

The use of pipelined channels significantly affects the design tradeoffs for *k*-ary *n*-cube networks. By decoupling the throughput of a link from the transmission latency across the link, pipelined channels remove one of the two primary disadvantages of high dimensional networks. Cycle time is no longer affected by the transmission time across the long wires inherent to high-dimensional networks. When wiring is not constrained, binary hypercubes now provide the best performance. When wiring *is* constrained, as is realistic for actual systems, the optimal network dimensionality is somewhere between the high dimensionality of a hypercube and the low dimensionality that is optimal for non-pipelined-channel networks.

Hardware-maintained cache coherence is challenging for large multiprocessors. The difficulty lies in maintaining enough information to prevent frequent broadcasts, which do not scale due to traffic, without requiring excessive storage for coherence information. I have investigated pruning-cache directories as a method to exploit the benefits of broadcasting while attempting to limit communication to the subset necessary to notify all interested parties of an

invalidation.

Pruning caches, particularly with the ownership modification (PC_{own}), provide robust performance over a broad spectrum of workloads. While utilizing a simple directory scheme, they can nevertheless exploit the limited broadcast inherent in bus- or ring-based topologies, much as snooping cache protocols do. Not only does this provide efficient coherence maintenance, but it allows read combining to be performed by the protocol, allowing workloads with concurrent read contention to scale to large system sizes. The storage requirements of pruning-cache directories are modest when compared to other coherence mechanisms, and scale well with system size. I conclude that pruning-cache directories are an excellent alternative for providing cache coherence in systems with many thousands of processors.

The work presented in this thesis suggests several interesting topics for future research. One particularly utilitarian topic would be to extend the analytical model of the SCI ring to handle the flow control protocol. This would be of tremendous use to future SCI designers, by allowing quick analysis of SCI designs that involve asymmetric configurations or traffic patterns. In addition to the utility of the result, this exercise could lead to some interesting modeling techniques, much as the current model did regarding packet train formation. Extending the model to multi-ring topologies is another possibility.

The analysis of pipelined channels could also be extended in some interesting ways. Two examples would be to consider their effect on topologies other than k-ary n-cubes, and to consider the effect of auto-correlated (bursty) or localized traffic. Case studies using specific technologies would also be useful.

The investigation of pruning-cache directories has raised some interesting questions. Perhaps the most interesting is the whether it is worth the trouble to provide read combining capabilities. The ability to combine concurrent read requests is the primary advantage of pruning cache directories over the dynamic pointer allocation scheme or the basic SCI protocol. It would be nice to have a better understanding of how useful that ability is. Another interesting question regarding read combining is whether the compiler can reasonably distinguish between normal references and references that might profit from combining.

Finally, it may prove valuable to question the basic premises upon which this thesis was built. Much of the thesis has focused on mechanisms for providing cache coherence in hardware. Many software coherence schemes have been proposed, as well. There is, perhaps, a fertile middle ground that involves cooperation between the compiler, operating system and hardware. And

there is always the question of whether we should employ the shared memory paradigm at all. How best to support parallel programming in the future remains very much an open question.

Appendix A
SCI Performance Model Equations

All lengths are in symbols. All times are in cycles.

MODEL INPUTS:

N	Number of nodes on the ring
$z_{i,j}$	Fraction of node i's packets routed to node j
λ_i	Packet arrival rate at node i's transmit queue
f_{data}	Fraction of transmitted packets that are data packets
f_{addr}	Fraction of transmitted packets that are addr packets
l_{data}	Length of a data packet (including postpended idle)
l_{addr}	Length of an address packet (including postpended idle)
l_{echo}	Length of an echo packet (including postpended idle)
T_{wire}	Number of cycles to traverse a wire
T_{parse}	Number of cycles to parse an incoming packet

PRELIMINARY CALCULATIONS:
These values are calculated directly from the model inputs. They do not change as the model iterates.

l_{send} Mean length of a send packet

$$l_{send} = f_{data}\, l_{data} + f_{addr}\, l_{addr} \tag{A.1}$$

X_i Mean throughput at node i (this includes packet header information)

$$X_i = \lambda_i\,(l_{send} - 1) \tag{A.2}$$

λ_{ring} Total packet arrival rate

$$\lambda_{ring} = \sum_{i=0}^{N-1} \lambda_i \tag{A.3}$$

$r_{echo,i}$ Rate of echo packets passing through node i (this includes echo packets that are created by node i, but not echo packets that are consumed by node i)

$$r_{echo,i} = \sum_{j \neq i} \lambda_j \sum_{\substack{k=j+1 \\ (mod\, N)}}^{i} z_{jk} \tag{A.4}$$

$r_{data,i}$ Rate of data packets passing through node i (this does not include data packets

transmitted by node i or stripped by node i)

$$r_{data,i} = f_{data} \sum_{j \neq i} \lambda_j \sum_{\substack{k=i+1 \\ (mod\ N)}}^{j-1} z_{jk} \tag{A.5}$$

$r_{addr,i}$ Rate of address packets passing through node i (this does not include address packets transmitted by node i or stripped by node i)

$$r_{addr,i} = f_{addr} \sum_{j \neq i} \lambda_j \sum_{\substack{k=i+1 \\ (mod\ N)}}^{j-1} z_{jk} \tag{A.6}$$

$r_{pass,i}$ Total rate of packets passing through node i

$$r_{pass,i} = r_{echo,i} + r_{data,i} + r_{addr,i} = \sum_{j \neq i} \lambda_j \tag{A.7}$$

$r_{rcv,i}$ Rate of packets routed to node i

$$r_{rcv,i} = \sum_{j \neq i} \lambda_j z_{ji} \tag{A.8}$$

$n_{pass,i}$ Mean number of packets passing through node i per packet transmitted by node i

$$n_{pass,i} = \frac{r_{pass,i}}{\lambda_i} \tag{A.9}$$

$U_{pass,i}$ Utilization of node i's output link by packets passing through node i

$$U_{pass,i} = r_{data,i} l_{data} + r_{addr,i} l_{addr} + r_{echo,i} l_{echo} \tag{A.10}$$

$l_{pkt,i}$ Mean length of a packet passing through node i

$$l_{pkt,i} = \frac{U_{pass,i}}{r_{pass,i}} \tag{A.11}$$

$L_{pkt,i}$ Mean residual life of packet passing through node i (for the purposes of this model, a packet is defined to be passing through a node if a symbol from the packet was output on the previous cycle; thus, the residual life of a packet of length l can take on only the discrete values $\{0,1,2,...(l-1)\}$)

$$L_{pkt,i} = \left[\frac{r_{data,i} l_{data}^2 + r_{addr,i} l_{addr}^2 + r_{echo,i} l_{echo}^2}{2U_{pass,i}} \right] - \frac{1}{2} \quad \text{(A.12)}$$

CALCULATIONS INSIDE ITERATION:
These values are directly or indirectly dependent upon the coupling probabilities. They are recomputed on every iteration until the coupling probabilities converge.

$n_{train,i}$ Mean number of packets in a packet train passing through node i (this uses the assumption that the number of packets in a train is given by a geometric distribution, parameterized by the coupling probability)

$$n_{train,i} = \frac{1}{1 - C_{pass,i}} \quad \text{(A.13)}$$

$l_{train,i}$ Mean length of packet train passing through node i

$$l_{train,i} = l_{pkt,i} n_{train,i} \quad \text{(A.14)}$$

$l_{idle,i}$ Mean space between packet trains passing through node i

$$l_{idle,i} = \left[\frac{1 - U_{pass,i}}{U_{pass,i}} \right] l_{train,i} \quad \text{(A.15)}$$

$P_{pkt,i}$ Probability that an idle symbol passing through node i is directly followed by a packet (this uses the assumption that the spacing between packet trains is geometrically distributed)

$$P_{pkt,i} = \frac{1}{l_{idle,i}} \quad \text{(A.16)}$$

Calculating the transmit queue service time: The service time for a packet of length l is simply the time taken to observe l idle symbols passing through the node. During a cycle in which an idle symbol arrives from the stripper, we reduce by one the number of symbols yet to transmit. During a cycle in which a packet symbol arrives from the stripper, we transmit one symbol, but also place one symbol into the ring buffer, prolonging the recovery period by one cycle and resulting in no reduction of the number of symbols yet to transmit. In the absence of any passing traffic, we see l passing idles in a row, and the service time is just l. Otherwise, we use the assumption that spacing between passing packet trains is geometrically distributed with parameter $P_{pkt,i}$. Every passing idle symbol is followed either by a packet train (which prolongs the recovery period by an average of $l_{train,i}$ cycles, and is followed by an idle symbol) with probability $P_{pkt,i}$, or by another idle symbol with probability $1-P_{pkt,i}$.

$P_{interrupt,i}$ Probability that a packet arriving to the transmit queue at node i finds the queue idle (no other packets are in the transmit queue and the ring buffer is empty), but a packet train passing through the node (a packet symbol was output on the previous cycle)

$$P_{interrupt,i} = (1-\rho_i)U_{pass,i} \tag{A.17}$$

$S_{reg,i}$ Mean transmit queue service time at node i when *not* interrupting a passing packet train (in this case, we require l_{send} passing idles to transmit the packet and complete the recovery period, and each of these idles is preceded by a passing packet train with probability $P_{pkt,i}$, prolonging the recovery period by an average of $l_{train,i}$ cycles)

$$S_{reg,i} = l_{send}(1+P_{pkt,i}l_{train,i}) \tag{A.18}$$

$S_{interrupt,i}$ Mean transmit queue service time at node i when interrupting a passing packet train (similar to $S_{reg,i}$, but in this case, we must first wait for the residual life of the passing packet, then on the first cycle of transmission either the passing train continues [with probability $C_{pass,i}$] or it does not [in which case we see one passing idle], then we need another $(l_{send}-1)$ passing idles, each of which is preceded by a passing packet train with probability $P_{pkt,i}$

$$S_{interrupt,i} = L_{pkt,i} + C_{pass,i}l_{train,i} + l_{send} + (l_{send}-1)P_{pkt,i}l_{train,i} \tag{A.19}$$

S_i Mean transmit queue service time at node i

$$S_i = P_{interrupt,i}S_{interrupt,i} + (1-P_{interrupt,i})S_{reg,i} \tag{A.20}$$

ρ_i Utilization of transmit queue at node i

$$\rho_i = \lambda_i S_i \tag{A.21}$$

Calculating new coupling probabilities: Coupling probabilities are affected by transmitting packets and stripping packets. Equation (A.22) calculates the new coupling probability on output link i by using the coupling probability of the pass-through traffic at node i and considering the effect of transmitting a single packet from the transmit queue. Equations (A.23) through (A.26) calculate the new coupling probability for the pass-through traffic at node i by using the coupling probability of the incoming link $(i-1)$ and considering the effect of stripping a single packet. Note that the rate of traffic across any link is λ_{ring}, the rate of echo packets consumed at node i is the same as the rate of send packets transmitted by node i (λ_i), and the rate of send packets that are stripped and converted to echo packets by node i is $r_{rcv,i}$.

$C_{link,i}$ Probability that packet on node i's output link is coupled to the packet in front of it (the three terms in the numerator denote the number of packets that were already coupled, the probability that the packet being transmitted is itself coupled to the packet in

front of it, and the number of passing packet trains that are caused to couple because they arrived during the transmission/recovery period and were routed into the ring buffer)

$$C_{link,i} = \frac{\left[n_{pass,i}C_{pass,i} + [\rho_i+(1-\rho_i)U_{pass,i}] + P_{pkt,i}l_{send}\right]}{(n_{pass}+1)} \tag{A.22}$$

$F_{in,i}$ Mean number of coupled packets entering stripper i per stripped packet (where a stripped packet is either an echo that is consumed or a send packet that is removed and replaced with an echo)

$$F_{in,i} = C_{link,i-1}\left[\frac{\lambda_{ring}}{\lambda_i + r_{rcv,i}}\right] \tag{A.23}$$

$P_{uncouple}$ Probability that stripping a packet at node i causes the following packet to become uncoupled, given there is a following packet (the first factor accounts for the fact that only consumed echo packets can decouple a following packet because stripped send packets are replaced with echo packets in their end positions; the second factor prevents "double counting" packets — we do not want to consider what happens to the packet following a stripped packet if it itself is stripped)

$$P_{uncouple,i} = \left[\frac{\lambda_i}{\lambda_i + r_{rcv,i}}\right]\left[\frac{\lambda_{ring} - \lambda_i - r_{rcv,i}}{\lambda_{ring}}\right] \tag{A.24}$$

F_{out} Mean number of coupled packets leaving stripper i per packet stripped at node i (the four terms in this equation correspond to the events of stripping a single-packet train, stripping the tail of a train, stripping a packet from the middle of a train, and stripping the head of a train; each term consists of the probability of that event, multiplied by the resulting number of coupled packets after the occurrence of that event)

$$\begin{aligned} F_{out,i} = {} & (1-C_{link,i-1})^2 F_{in,i} + C_{link,i-1}(1-C_{link,i-1})(F_{in,i}-1) \\ & + C_{link,i-1}^2 (F_{in,i}-1-P_{uncouple,i}) \\ & + (1-C_{link,i-1})C_{link,i-1}(F_{in,i}-P_{uncouple,i}) \end{aligned} \tag{A.25}$$

$C_{pass,i}$ Probability that a packet passing through node i (having just left the stripper) is coupled to the packet in front of it (the numerator is the rate of coupled packets leaving the stripper and the denominator is the total rate of packets leaving the stripper

$$C_{pass,i} = \frac{F_{out,i}(\lambda_i + r_{rcv,i})}{r_{pass,i}} \tag{A.26}$$

CALCULATING FINAL MODEL OUTPUTS:

$V_{pkt,i}$ Variance of packet length for packets passing through node i (this is derived directly from the definition of variance: $V(X) = E[(X - E[X])^2]$)

$$V_{pkt,i} = \left[\frac{r_{data,i}(l_{data} - l_{pkt,i})^2 + r_{addr,i}(l_{addr} - l_{pkt,i})^2 + r_{echo,i}(l_{echo} - l_{pkt,i})^2}{r_{pass,i}} \right] \tag{A.27}$$

$V_{train,i}$ Variance of length of packet trains passing through node i (this derivation uses the fact that a packet train is comprised of a geometrically distributed number of packets, for which we know the mean length and variance of length)

$$\begin{aligned}
V_{train,i} &= E[l_{train,i}^2] - E[l_{train,i}]^2 \\
&= \sum_{i=1}^{\infty} (1 - C_{pass,i}) C_{pass,i}^{i-1} (iV_{pkt,i} + [il_{pkt,i}]^2) - E[l_{train,i}]^2 \\
&= (1 - C_{pass,i}) V_{pkt,i} \sum_{i=1}^{\infty} iC_{pass,i}^{i-1} + (1 - C_{pass,i}) l_{pkt,i}^2 \sum_{i=1}^{\infty} i^2 C_{pass,i}^{i-1} \\
&\qquad - \left[\frac{l_{pkt,i}}{(1 - C_{pass,i})} \right]^2 \\
&= \frac{(1 - C_{pass,i}) V_{pkt,i}}{(1 - C_{pass,i})^2} + (1 - C_{pass,i}) l_{pkt,i}^2 \left[\frac{(1 + C_{pass,i})}{(1 - C_{pass,i})^3} \right] - \left[\frac{l_{pkt,i}}{(1 - C_{pass,i})} \right]^2 \\
&= \frac{V_{pkt,i}}{(1 - C_{pass,i})} + \frac{l_{pkt,i}^2 C_{pass,i}}{(1 - C_{pass,i})^2}
\end{aligned} \tag{A.28}$$

Calculating service time variance: The service time variance calculation is approximate and rather awkward. The *variable* part of the service time can be broken into two components, with means:

$$\mu_{1,i} = (1 - \rho_i) U_{pass,i} [L_{pkt,i} + (C_{pass,i} - P_{pkt,i}) l_{train,i}] \tag{A.29}$$

$$\mu_{2,i} = l_{send} P_{pkt,i} l_{train,i} \tag{A.30}$$

The second component corresponds to the sum of a binomially distributed (l_{send} trials with probability $P_{pkt,i}$) number of packet train lengths, each with mean $l_{train,i}$ and variance $V_{train,i}$. We can compute the variance of this "binomial" component exactly, as follows. Let $type \in \{addr, data\}$ and $\phi_{type,i}$ be the random variable representing the binomial component of service time for

transmission of a type *type* packet.

$\nu_{type,i}$ Variance of "binomial" component of service time for transmission of a type *type* packet

$$\nu_{type,i} = E[\phi_{type,i}{}^2] - E[\phi_{type,i}]^2$$

$$= \left[\sum_{j=1}^{l_{type}} \binom{l_{type}}{j} P_{pkt,i}{}^{j}(1-P_{pkt,i})^{l_{type}-j}(jV_{train,i} + [jl_{train,i}]^2)\right]$$

$$- (l_{train,i}P_{pkt,i}l_{type})^2 \qquad \text{(A.31)}$$

To calculate the total service time variance, it is assumed that the service time components corresponding to $\mu_{1,i}$ and $\mu_{2,i}$ have a correlation of 1 (this is certainly not true, but the correlation *is* expected to be high). We can then use the fact that $V[cX] = V[X]c^2$ when c is constant.

$V_{type,i}$ Variance of service time for a packet of type *type* transmitted at node i

$$V_{type,i} = \nu_{type,i}\left(\frac{\mu_{1,i}+\mu_{2,i}}{\mu_{2,i}}\right)^2 \qquad \text{(A.32)}$$

V_i Overall variance of service time for an packet transmitted at node i

$$V_i = E[S_i{}^2] - E[S_i]^2$$

$$= f_{data}(V_{data,i} + S_{data,i}{}^2) + f_{addr}(V_{addr,i} + S_{addr,i}{}^2) - S_i{}^2 \qquad \text{(A.33)}$$

c_i Coefficient of variation of S_i

$$c_i = \frac{\sqrt{V_i}}{S_i} \qquad \text{(A.34)}$$

Q_i Mean transmit queue length at node i (from the solution of the M/G/1 queue)

$$Q_i = \rho_i + \frac{\rho_i{}^2(1+c_i{}^2)}{2(1-\rho_i)} \qquad \text{(A.35)}$$

L_i Mean residual life of transmit queue service time at node i (from the solution of the M/G/1 queue)

$$L_i = \frac{V_i + S_i^2}{2S_i} \tag{A.36}$$

W_i Mean wait time in transmit queue at node i (from the solution of the M/G/1 queue)

$$W_i = (Q_i - \rho_i)S_i + \rho_i L_i \tag{A.37}$$

B_i Mean buffer backlog seen by a packet passing through node i (buffer backlog occurs only as a result of packet transmissions; the three terms inside the square brackets compute the total mean packet-cycles of delay caused by a single packet transmission, and this is divided by the mean number of packets that pass through the node per transmitted packet in order to obtain the mean delay that a passing packet sees)

$$B_i = \left[(1-\rho_i)U_{pass,i}(C_{pass,i} - P_{pkti})l_{send}n_{train,i} + f_{data}P_{pkt,i}l_{data}\left[\frac{l_{data}+1}{2}\right]n_{train,i} + f_{addr}P_{pkt,i}l_{addr}\left[\frac{l_{addr}+1}{2}\right]n_{train,i}\right]\frac{1}{n_{pass,i}} \tag{A.38}$$

T_i Mean transit time for a packet transmitted from node i (latency once transmission actually begins)

$$T_i = 1 + T_{wire} + t_{parse} + l_{send} + \sum_{j \neq i} z_{ij} \sum_{\substack{k=i+1 \\ (mod N)}}^{j-1} (1 + T_{wire} + t_{parse} + B_k) \tag{A.39}$$

R_i Mean response time for a packet transmitted from node i

$$R_i = W_i + P_{interrupt,i}L_{pkt,i} + T_i \tag{A.40}$$

Appendix B
Miscellaneous Read Combining Equations

These equations compute quantities relating to read combining in a k-ary n-cube. A total of m processors, randomly distributed throughout the system, are assumed to be reading a shared line of memory. The equations included in this appendix are simply support equations for the analysis of Chapter 4, Section 3.

$P_{all\_in}(m,x)$ Probability that all m readers of a line reside within a subset of x processors

$$P_{all\_in}(m,x) = \begin{cases} \prod_{i=0}^{m-1} \left(\frac{x-m}{k^n-m} \right) & \text{if } m \le x \\ 0 & \textit{otherwise} \end{cases} \tag{B.1}$$

$P_{none\_in}(m,x)$ Probability that none of m readers of a line reside within a subset of x processors

$$P_{none\_in}(m,x) = P_{all\ in}(m,k^n-x) \tag{B.2}$$

$N_d(x)$ Number of nodes within x dimensions of a node

$$N_d(x) = \sum_{i=0}^{x} b(n,i)(k-1)^i \tag{B.3}$$

$N_l(x)$ Number of nodes within x links of a node

$$N_l(x) = \sum_{d_1=0}^{k-1} \sum_{d_2=0}^{k-1} \cdots \sum_{d_n=0}^{k-1} \delta\left(\sum_{j=1}^{n} d_j \le x\right) \tag{B.4}$$

$E_{min\_dims}(m)$ Expected minimum number of dimensions traversed by one of the m readers of a line on its way to memory

$$E_{min\_dims}(m) = \sum_{i=1}^{n} i \left[P_{none\_in}(m, N_d(i-1)) - P_{none\_in}(m, N_d(i)) \right] \quad \text{(B.5)}$$

$E_{max\_dims}(m)$ Expected maximum number of dimensions traversed by one of the m readers of a line on its way to memory

$$E_{max\_dims}(m) = \sum_{i=1}^{n} i \left[P_{all\_in}(m, N_d(i)) - P_{all\_in}(m, N_d(i-1)) \right] \quad \text{(B.6)}$$

$E_{min\_links}(m)$ Expected minimum number of links traversed by one of the m readers of a line on its way to memory

$$E_{min\_links}(m) = \sum_{i=1}^{n(k-1)} i \left[P_{none\_in}(m, N_l(i-1)) - P_{none\_in}(m, N_l(i)) \right] \quad \text{(B.7)}$$

$E_{max\_links}(m)$ Expected maximum number of links traversed by one of the m readers of a line on its way to memory

$$E_{max\_links}(m) = \sum_{i=1}^{n(k-1)} i \left[P_{all\_in}(m, N_l(i)) - P_{all\_in}(m, N_l(i-1)) \right] \quad \text{(B.8)}$$

References

[Adve90] Adve, S. V. and M. D. Hill, Weak Ordering - A New Definition, *Proc. 17th Annual International Symposium on Computer Architecture*, May 1990, 2-14.

[Agar88] Agarwal, A., R. Simoni, J. Hennessy, and M. Horowitz, An Evaluation of Directory Schemes for Cache Coherence, *Proc. 15th Annual International Symposium on Computer Architecture*, June 1988, 280-289.

[Agar90] Agarwal, A., B.-H. Lim, D. Kranz, and J. Kubiatowicz, APRIL: A Processor Architecture for Multiprocessing, *Proc. 17th Annual International Symposium on Computer Archtecture*, May 1990, 104-114.

[Agar91] Agarwal, A., Limits on Interconnection Network Performance, *IEEE Transactions on Parallel and Distributed Systems*, October 1991, 398-412.

[Alve90] Alverson, R., D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, The Tera Computer System, *Proc. 1990 International Conference on Supercomputing*, June 1990.

[Amda67] Amdahl, G. M., Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, *Proc. AFIPS Conference*, April 1967, 483-485.

[Arch84] Archibald, J. and J. -L. Baer, An Economical Solution to the Cache Coherence Problem, *Proc. 11th Annual International Symposium on Computer Architecture*, June 1984, 355-362.

[Baer88] Baer, J.-L. and W.-H. Wang, On the Inclusion Properties for Multi-Level Cache Hierarchies, *Proc. 15th Annual International Symposium on Computer Architecture*, June 1988, 73-80.

[Bell85] Bell, C. G., Multis: a New Class of Multiprocessor Computers, *Science* 228, April 1985, 462-467.

[Benn88] Benner, R. E., J. L. Gustafson, and R. E. Montry, Development and analysis of scientific application programs on a 1024-processor hypercube, SAND 88-0317, Sandia National Laboratories, Albuquerque, NM, February 1988.

[Bork90] Borkar, S., R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J.

Webb, Supporting Systolic and Memory Communication in iWarp, *Proc. 17th Annual International Symposium on Computer Architecture*, May 1990, 70-81.

[Burc90] Burch, J. R., E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang, Symbolic Model Checking: 10^20 States and Beyond, *Proc. Fifth Anual Symposium on Login in Computer Science*, June 1990.

[Cens78] Censier, L. M. and P. Feautrier, A New Solution to Coherence Problems in Multicache Systems, *IEEE Transactions on Computers* **C-27**(12), December 1978, 1112-1118.

[Chai90] Chaiken, D., C. Fields, K. Kurihara, and A. Agarwal, Directory-Based Cache Coherence in Large-Scale Multiprocessors, *IEEE Computer* **23**(6), June 1990, 49-58.

[Chai91] Chaiken, D., J. Kubiatowicz, and A. Agarwal, LimitLESS Directories: A Scalable Cache Coherence Scheme, *Proc. ASPLOS IV*, April 1991, 224-234.

[Cher89] Cheriton, D. R., H. A. Goosen, and P. D. Boyle, Multi-Level Shared Caching Techniques for Scalability in VMP-MC, *Proc. 16th Annual International Symposium on Computer Architecture*, May 1989, 16-24.

[Dall87] Dally, W. J., Deadlock Free Message Routing in Multiprocessor Interconnection Networks, *IEEE Transactions on Computers* **C-36**(5), May 1987, 547-553.

[Dall90] Dally, W. J., Performance Analysis of k-ary n-cube Interconnection Networks, *IEEE Transactions on Computers* **39**(6), June 1990, 775-785.

[Dall92] Dally, W. J., Virtual-Channel Flow Control, *IEEE Transactions on Parallel and Distributed Systems* **3**(2), March 1992, 194-205.

[Dubo86] Dubois, M., C. Scheurich, and F. Briggs, Memory Access Buffering in Multiprocessors, *Proc. 13th Annual International Symposium on Computer Architecture*, June 1986, 434-442.

[Egge89] Eggers, S. J. and R. Katz, The Effect of Sharing on the Cache and Bus Performance of Parallel Programs, *Proc. ASPLOS III*, April 1989, 257-270.

[Enco86] Encore,, *Multimax Technical Summary,* Encore Computer Corporation, 1986.

[Feng81] Feng, T., A Survey of Interconnection Networks, *Computer*, December, 1981, 12-27.

[Flat89] Flatt, H. P. and K. Kennedy, Performance of Parallel Processors, *Parallel Computation* **31**, 1989, 1-20.

[Fort78] Fortune, S. and J. Wyllie, Parallelism in Random Access Machines, *Proc. Tenth ACM Symposium on Theory of Computing*, 1978, 114-118.

[Fran84] Frank, S. J., Tightly Coupled Multiprocessor System Speeds Up Memory Access Times, *Electronics* **5**(1), January 1984, 164-169.

[Ghar90] Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, *Proc. 17th Annual International Symposium on Computer Architecture*, May 1990, 15-26.

[Good83] Goodman, J. R., Using Cache Memory to Reduce Processor-Memory Traffic, *Proc. 10th Annual International Symposium on Computer Architecture*, June 1983, 124-131.

[Good88] Goodman, J. R. and P. J. Woest, The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor, *Proc. 15th Annual International Symposium on Computer Architecture*, June 1988, 422-431.

[Good89] Goodman, J. R., M. D. Hill, and P. J. Woest, Scalability and Its Application to Multicube, Computer Sciences Technical Report #835, University of Wisconsin-Madison, Madison, WI 53706, March 1989.

[Good89a] Goodman, J. R., M. K. Vernon, and P. J. Woest, Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors, *Proc. ASPLOS III*, April 1989, 64-75.

[Gott83] Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, The NYU Ultracomputer -- Designing a MIMD, Shared Memory Parallel Machine, *IEEE Transactions on Computers* **C-32**(2), February 1983, 175-189.

[Gupt90] Gupta, A., W. Weber, and T. Mowry, Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes, *Proc. 1990 International Conference on Parallel Processing*, August 1990, I312-I321.

[Gust88] Gustafson, J. L., Reevaluating Amdahl's Law, *Communications of the ACM* **31**(5), May 1988, 532-533.

[Gust92] Gustavson, D. B., The Scalable Coherent Interface and Related Standards Projects, *IEEE Micro* **12**(1), February 1992, 10-22.

[Hage89] Hagersten, E. and S. Haridi, The Cache Coherence Protocol of the Data Diffusion Machine, SICS Research Report R-89004, Swedish Institute of Computer Science, Kista, Sweden, May 1989.

[Hill90] Hill, M. D., What is Scalability?, *Computer Architecture News*, December 1990, 18-21.

[Hill85] Hillis, W. D., *The Connection Machine,* MIT Press, Cambridge, MA, 1985.

[IEEE92] IEEE,, *IEEE Std 1596-1992 (Scalable Coherent Interface).* 1992.

[Jame90] James, D. V., A. T. Laundrie, G. S. Sohi, and S. Gjessing, SCI (Scalable Coherent Interface) Cache Coherence, *IEEE Computer*, July 1990.

[John90] Johnson, R. E., Scalable Read-Sharing in SCI, Ph. D. Preliminary Exam, Department of Computer Sciences, University of Wisconsin-Madison, Madison, WI, August 1990.

[John91] Johnson, Ross E. and James R. Goodman, Interconnect Topologies with Point-To-Point Rings, Computer Sciences Technical Report #1058, University of Wisconsin-Madison, December 1991. Condensed version to appear in ICPP, August 1992.

[Katz85] Katz, R. H., S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, Implementing a Cache Consistency Protocol, *Proc. 12th Annual International Symposium on Computer Architecture*, June 1985, 276-283.

[Kerm79] Kermani, P. and L. Kleinrock, Virtual Cut-Through: A New Computer Communication Switching Technique, *Computer Networks* 3, 1979, 267-286.

[Klei75] Kleinrock, L., *Queueing Systems, Volume I,* John Wiley and Sons, New York, 1975.

[Lam79] Lam, C.-Y. and S. E. Madnick, Properties of Storage Hierarchy Systems with Multiple Page Sizes and Redundant Data, *ACM Transactions on Database Systems* 4(3), September 1979, 345-367.

[Lamp78] Lamport, L., Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM* 21(7), July 1978, 558-565.

[Lang88] Lang, T. and L. Kurisaki, Nonuniform Traffic Spots (NUTS) in Multistage Interconnection Networks, *Proc. 1988 International Conference on Parallel Processing*, August 1988.

[Lawr75] Lawrie, D. H., Access and Alignment of Data in an Array Processor, *IEEE Transactions on Computers* C-24(12), December 1975, 1145-1155.

[Leno89] Lenoski, D., J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, Design of the Stanford DASH Multiprocessor, CSL Technical Report 89-403, Stanford University, December 1989.

[Leno90] Lenoski, D., J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor, *Proc. 17th Annual International Symposium on Computer Architecture*, May 1990, 148-158.

[Love88] Lovett, T. and S. Thakkar, The Symetry Multiprocessor System, *Proc. International Conference on Supercomputing*, 1988, 303-310.

[McCr84] McCreight, E. M., The Dragon Computer System: An Early Overview, Technical Report, Xerox Corp., September 1984.

[Mell91] Mellor-Crummey, J. M. and M. L. Scott, Synchronization Without Contention, *Proc. ASPLOS IV*, April 1991, 269-278.

[Nuss91] Nussbaum, D. and A. Agarwal, Scalability of Parallel Machines, *Communications of the ACM* 34(3), March 1991, 56-61.

[O'Kr90] O'Krafka, B. W. and A. R. Newton, An Empirical Evaluation of Two Memory-Efficient Directory Methods, *Proc. 17th Annual International Symposium on Computer Architecture*, May 1990, 138-147.

[Olso83] Olson, R. A., B. Kumar, and L. E. Shar, Messages and Multiprocessing in the ELXSI System 6400, *COMPCON83*, February 1983, 21-24.

[Papa84] Papamarcos, M. S. and J. H. Patel, A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories, *Proc. 11th Annual International Symposium on Computer Architecture*, June 1984, 348-354.

[Patt85] Patton, P. C., Multiprocessors: architecture and applications, *IEEE Computer* 18(6), June 1985, 29-40.

[Pfis85] Pfister, G. F. and et al, The IBM Research Parallel Processor Prototype (RP3): introduction and architecture, *Proc. 1985 International Conference on Parallel Processing*, August 1985, 764-771.

[Scot90] Scott, S. L. and G. S. Sohi, The Use of Feedback in Multiprocessors and Its Application to Tree Saturation Control, *IEEE Transactions on Parallel and Distributed Systems* 1(4), October 1990, 385-398.

[Scot91] Scott, S. L., A Cache Coherence Mechanism for Scalable, Shared Memory Multiprocessors, *Proc. 1991 International Symposium on Shared Memory Multiprocessing*, April 1991, 49-59.

[Scot92] Scott, S. L., J. R. Goodman, and M. K. Vernon, Performance of the SCI Ring, *Proc. the 19th Annual International Symposium on Computer Archtecture*, May 1992, 403-414.

[Seit84] Seitz, C. L., Concurrent VLSI Architectures, *IEEE Transactions on Computers* **33**(12), December 1984, 775-785.

[Seit85] Seitz, C. L., The Cosmic Cube, *Communications of the ACM* **28**(1), January 1985, 22-33.

[Seit92] Seitz, C. L., *Personal correspondence.* April 1992.

[SGI89] SGI,, Power Series, Silicon Graphics Technical Report, 1989.

[Sieg79] Siegel, H. J., Interconnection Networks for SIMD Machines, *Computer* **12**(6), June, 1979, 57-65.

[Siew91] Siewiorek, D. P. and P. J. Koopman, Jr., *The Architecture of Supercomputers: Titan, A Case Study,* Academic Press, San Diego, CA, 1991.

[Simo91] Simoni, R. and M. Horowitz, Dynamic Pointer Allocation for Scalable Cache Coherence Directories, *Proc. 1991 International Symposium on Shared Memory Multiprocessing*, April 1991, 72-81.

[Smit92] Smith, J. E., *Personal correspondence.* April 1992.

[Stal84] Stallings, W., Local Networks, *Computing Surveys* **16**(1), March 1984, 3-41.

[Sull77] Sullivan, H. and T. R. Bashkow, A Large Scale, Homogeneous, Fully Distributed Parallel Machine, *Proc. 4th Annual International Symposium on Computer Architecture*, March 1977, 105-117.

[Tami88] Tamir, Y. and G. L. Frazier, High-Performance Multi-Queue Buffers for VLSI Communication Switches, *Proc. 15th Annual International Symposium on Computer Architecture*, June 1988, 343-354.

[Tane88] Tanenbaum, A. S., *Computer Networks,* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[Tang76] Tang, C. K., Cache System Design in the Tightly Coupled Multiprocessor System, *Proc. AFIPS*, 1976, 749-753.

[Thac87] Thacker, C. P. and L. C. Stewart, Firefly: a Multiprocessor Workstation, *Proc. ASPLOS II*, October 1987, 164-172.

[Thap90] Thapar, M. and B. Delagi, Stanford Distributed-Directory Protocol, *IEEE Computer* **23**(6), June 1990, 78-81.

[TMC91] TMC,, *CM5 Reference Manual,* Thinking Machines, Inc., Cambridge, MA, 1991.

[Vern89] Vernon, M. K., R. Jog, and G. S. Sohi, Performance Analysis of Hierarchical Cache-Consistent Multiprocessors, *Performance Evaluation* **9**, 1989, 287-302.

[Webe89] Weber, W. and A. Gupta, Analysis of Cache Invalidation Patterns in Multiprocessors, *Proc. ASPLOS III*, April 1989, 243-256.

[Wils87] Wilson, A. W., Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors, *Proc. 14th Annual International Symposium on Computer Architecture*, June 1987, 244-252.

[Wins88] Winsor, D. C. and T. N. Mudge, Analysis of Bus Hierarchies for Multiprocessors, *Proc. 15th Annual International Symposium on Computer Architecture*, June 1988, 100-107.

[Wood90] Wood, D. A., G. Gibson, and R. H. Katz, Verifying a Multiprocessor Cache Controller Using Random Case Generation, *IEEE Design and Test of Computers*, 1990.

[Yang92] Yang, Q., G. Thangadurai, and L. Bhuyan, Design of an Adaptive Cache Coherence Protocol for Large Scale Multiprocessors, *IEEE Transactions on Computers* **3**(3), May 1992, 281-293.

[Yew87] Yew, P.-C., N.-F. Tzeng, and D. H. Lawrie, Distributing Hot-Spot Addressing in Large Scale Multiprocessors, *IEEE Transactions on Computers* **C-36**(4), April 1987, 388-395.