A Communication Architecture for Multiprocessor Networks

A. Nowatzyk

April 1989

TR-89-181

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in Computer Science at
Carnegie Mellon University

This research was sponsored in part by a Grant of the German Academic Exchange Service (DAAD) and by an IBM Fellowship.

Abstract

The system described in this thesis explores the territory between the two classical multiprocessor families: shared memory and message-passing machines. Like shared memory systems, the proposed architecture presents the user a logically uniform address space shared by all processors. This programming model is supported directly by dedicated communication hardware that is translating memory references into messages that are exchanged over a network of point to point channels. The key parts of this work are the communication system and its integration with contemporary processor and memory components to form a homogeneous, general-purpose multiprocessor.

The communication system is based on an adaptive routing heuristic that is independent of the actual network topology. High priority was given to optimal use of the physical bandwidth even under heavy or saturated load conditions. The communication system can be extended in small, incremental upgrades and supports medium haul channels that can link two or more multiprocessor clusters together in a transparent fashion.

Integration of the communication system is based on the shared memory model. This avoids the overhead of explicitly sending and receiving messages but introduces the problem of maintaining a consistent state. Memory coherence is achieved through the notion of time. A system wide clock of sufficient precision to sequentialize concurrent access is maintained in hardware. As a measure to avoid unnecessary synchronizations, the memory model is relaxed to allow transient inconsistencies. Application code can explicitly resort to strongly coherent memory access at the expense of higher latency.

The primary tool for assessing the performance of the proposed architecture was a simulator that can execute application programs for the target system. Nonintrusive instrumentation was provided down to individual clock cycles. A trace-based visualization tool aided both the debugging of the architecture and the application code benchmarks.

Acknowledgments

I thank my thesis committee, Bob Sproull, Roberto Bisiani, Allan Fisher and Doug Jensen for the care with which they read my thesis and for their thoughtful comments and suggestions. Bob Sproull earned my deepest respect for his profound knowledge. He always knew how to ask the toughest questions. Roberto Bisiani was a very helpful and supportive advisor who was always willing to discuss new idea. Allan Fisher's careful review of my thesis led to many significant improvements. I value Doug Jensen as a true professional whose insights and comments draw on many years of practical experience.

I further thank Frieder Schwenkel who planted the mental seeds for this research. My work has benefitted from many discussions with numerous faculty, researchers, visitors, staff and students here at CMU and elsewhere. In particular, I would like to express my gratitude to Thomas Anantharaman, Lawrence Butcher, Michael Browne, Andy Gruss, Feng-Hsiung Hsu, Duane Northcutt and John Zsarnay.

Finally, the CMU CS department and the inhabitants of the engineering lab provided an environment that made this research both possible and enjoyable.

Chapter 1
Introduction

The quest for ever-increasing performance proceeds in four major directions: implementation technology, processor architecture, software support and system organization. Advances in technology made computers possible in the beginning and it is still the main driving force. Circuit sizes are shrinking and integration densities are rising at an almost constant pace. In addition to these traditional engineering feats, the progress in interconnection, packaging and cooling technology is equally impressive. New devices such as the quantum-well resonant tunneling bipolar transistor [145] indicate that circuit speed will continue to improve. While superconducting circuits still have not emerged from the laboratory, recent advances in superconducting materials could result in faster interconnection technology [80].

Processor architecture has evolved from an art into a solid engineering discipline. Differences among contemporary CPUs have more to do with patent rights and desire for backward compatibility than with diverging perceptions of the design tradeoffs. Processor architecture is a mature field where different designs using the same technology usually result in comparable performance. Since the ground rules are well understood, recent designs pay more attention to the interaction with other areas, such as compiler technology.

Software development is a more recent and less mature area. One of the more critical parts is the compiler technology. At the low end, optimizers and code generators are relatively well understood. The efforts to match these systems with processor architectures have been highly successful. However as the level of abstraction increases, consensus and established engineering principles are rapidly replaced by intuition and experimentation.

Finally, the least mature area and the subject of this thesis is the system organization, which includes parallel processing, distributed computing and concurrent programming. At the circuit level, parallelism has been used to replace slow, bit serial machines with word parallel organization. Still part of the processor architecture, pipelining distributes work among a collection of dissimilar, specialized units [2]. This approach eventually produced vector processors with very long pipelines to support regular and repetitive operation [25, 123, 147]. More recently, very long instruction word machines (VLIW's) are trying to overcome the rigidity and limited scope of vector processors by creating a pool of less specialized functional units that are interconnected by a set of data paths that can be reconfigured on a cycle by cycle basis [29, 44]. The VLIW draws heavily on specialized compiler technology.

Orthogonal to pipelining, multiple identical processing elements are also used to perform the same or similar sequence of operations on different data. The observation that certain algorithms apply one procedure to each element of a large array of data structures resulted in array processors with a single control thread operating on multiple data streams [11]. More extreme examples are systems with a large pool of extremely simple processing elements [53, 12, 30]. All array processors thrive on a high degree of regularity in their applications, hence are special purpose machines.

Two factors favor vector and array multiprocessors: no need for multiple controllers and simplicity of synchronization. Multiple controllers imply more memories, sequencers, instruction decoders, etc. Decreasing hardware cost and higher integration levels tend to minimize this factor. The flexibility gained with multiple control threads will eventually outweigh the extra hardware and synchronization cost. Hence there is a trend towards parallel systems composed of independent processors, also known as multiple instruction multiple data steam (MIMD) machines.

This thesis is focused on the structure of the communication architecture of MIMD machines and its implication on the application software. In this context, the actual processor architecture is assumed to be a black box not unlike current high-end CPUs. However, there are a number of relative simple changes to a CPU architecture that could ease multiprocessor integration.

Throughout this thesis, the problem of connecting the nodes of a tightly coupled MIMD machine is structured into two distinct layers: the message system and the communication architecture. The message system encompasses the first four layers of the OSI model, from the physical link layer through the transport layer. The communication architecture deals with the remaining three layers. However, the OSI model is inadequate for describing MIMD communication structures because its intended scope deals with loosely coupled, distributed and independent nodes. The underlying assumption in this thesis is that the entire machine is controlled by *one* potentially distributed operating system. The system may run one multithreaded task that uses all resources, a composite workload, or any combination thereof. However the single operating system assumption implies that all resources are controlled by one entity.

The message system takes care of the physical transfer of data between nodes. This includes the implementation of communication channels, transmission protocols, network topology, routing, buffering, sequencing, flow control, etc.

The communication architecture defines the interface between programs running in each node and the data transmission service provided by the message system. This includes the programming model, synchronization facilities, control of resources, etc.

The contributions of this thesis include:

- A systematic review of a representative collection of network topologies. The evaluation includes bandwidth and latency tradeoffs.
- A method to reconcile the conflict between bandwidth and latency optimization.
- A precise clock distribution method that is extendible to distributed systems.
- A high performance message system that employs a new adaptive routing method. This message system is specifically optimized for high, fine-grained load. Such loads are typical of virtual shared memory systems.
- A system for distributing the message system over a local area in an integrated and transparent fashion.
- A simple and intuitive programming model. A notion of weak coherency that is based on real time is used. Weak coherency allows cost effective implementations of virtual shared memory on a private memory machines.
- An implementation outline and performance analysis of the programming model on top of the new message system.

This thesis emphasizes feasibility and cost effectiveness. Therefore it does not compete with *money-is-no-issue* supercomputers and is not concerned with billions of processors. Hence theoretical considerations of asymptotic limits and proofs that drop constant factors are intentionally missing.

Multiprocessors with conventional control structure are not the only approach to high performance [8]. However radically different architectures such as data flow machines, cellular automata, associative processors and others are beyond the scope of this thesis.

1.1. Background

There are basically two broad families of MIMD machines: shared and private memory systems. As the name implies, shared memory machines allow some or all of the memory to be accessed by all processors. Initially, memory sharing was used in high-end commercial mainframes to offload the mundane I/O tasks from the expensive CPU. Examples include the I/O processors of the CDC-6600 series and the channels of IBM's 360 architecture. Later CPU models added *"true"* multiprocessing by having more than one CPU, for example IBM's 370/155MP.

While being true multiprocessors, the design of these machines was motivated by reliability considerations. One failing CPU could be replaced by the other, sharing the same I/O devices and the same expensive memory. Concurrent operation of multiple CPU's on the same task was generally not intended and frequently not supported by the operating system. These machines are frequently omitted from multiprocessor surveys. *True* shared memory multiprocessors became popular once the cost of mini-computers allowed experimentation.

Private memory multiprocessors or ensemble machines are collections of single processor machines. Each node has its own CPU, memory, and peripherals. Communication is accomplished by channels that can connect two or more nodes and that appear as peripheral I/O devices to each node. Private memory multiprocessors differ from networks of distributed computers only by the speed and bandwidth of their communication channels. Part of the popularity of private memory computers is the ease of their construction: adding a peripheral to a selfcontained machine is much easier than connecting the processor to memory data paths.

1.1.1. Shared Memory Systems

Typical memory access rates are normally below the potential bandwidth demand of a processor implemented in the same technology. Therefore systems with multiple processors connected to a common memory face a bandwidth problem. This problem is twofold: the raw memory bandwidth has to match the processor demands and the interconnection structure must be able to deal with the total memory traffic.

The memory bandwidth can be increased by providing multiple, independent memory units or banks and crossbar switches can carry the traffic of a modest number of processors. This approach was pioneered in C.mmp [155]. Subsequent systems of this architecture replaced the crossbar switch with cheaper, multistage interconnection networks. The most common of these is the omega network [153], which is used in BBN's Butterfly [126] or IBM's RP^3 [112].

Providing multiple memories for multiple processors only creates a static bandwidth balance. Contention caused by several processors accessing the same memory bank can severely degrade performance. In the case of multistage interconnection networks, the memory contention problem is amplified by a backlog in the network that also affects traffic to unrelated memory banks. This tree saturation or hot-spot problem [113, 110] can degrade network performance for extended periods after transient contention. Omega networks have a constant number of stages between the processor and the memory modules. Alternative variable depth topologies such as the binary hypercube, that colocate processors and memories, have been shown to be less sensitive to pathological access patterns [21].

Hotspot induced network degradations can be mitigated by switches that can sense contention and quickly abort connection requests [146, 40] or by adding redundancy to the network [148, 61]. However, this does not solve the memory contention problem which is prone to arise from synchronization operations. Combining networks and the *fetch-and-add* primitive were proposed to solve this problem [121]. However, combining networks is a hard technique to implement.

Machines that rely on a single bus avoid problems related to the switch entirely [50, 114]. Naturally, a single bus carrying the traffic of multiple processors is prone to become a bottleneck. Therefore all bus based shared memory multiprocessors employ caches in each processor to reduce the main memory bandwidth demand. Since the bus traffic is visible by all processors, cache controllers can exploit this broadcast property to maintain coherence [69, 70, 5, 108].

Synchronization operations benefit from the simplicity of a single bus. The *test-and-test-and-set* instruction uses the snooping cache coherency mechanism to avoid the memory traffic associated with spin-locks [122]. However, in situations with many waiting processors, performance degrades as lock releases cause substantial cache reload traffic [3].

1.1.2. Private Memory Systems

Private memory systems became popular with Caltech's Cosmic Cube [127], which influenced numerous commercial and academic machines. The direct descendants of the cosmic cube include Intel's iPSC-I, Ametek's S-14 and JPL's Mark-II. The latter two machines added a dedicated communication processor to reduce communication overhead. Common to this class of machines is a conventional microprocessor board (8*86 - based) augmented by a number of serial, node-to-node communication channels. The channels act as simple peripheral devices that are attached to the processor bus and operate out of the local memory.

The Ncube/10 system uses essentially the same organization but integrates CPU, memory control, and the I/O channels in a single chip [89]. Because of this dense, custom VLSI approach, 64 processors are packaged on a single board. More recent systems include JPL's Mark-IIIa [22], which features more powerful processors and faster links, and Ametek's 2010 [129], which uses a grid topology instead of a hypercube. Floating Point System's T-series [56] added vector processing to each node of a hypercube machine. The commercial failure of the FPS-T is partially attributed to a radically different software environment (OCCAM) and the subsequent lack of applications and libraries.

INMOS's Transputer (US Patent 4,704,676 November 3, 1987) is a family of processors with integrated communication channels [151, 60, 66, 144]. Like the Ncube-nodes, each processor requires very few external components. However, unlike the Ncube processor, the transputer is a readily available component that has found wide acceptance. Initially conceived as an embedded controller dedicated to specific applications, the transputer does not use a particular topology. Instead, the four I/O channels are intended to be used in any way best suited for a particular application. As subsequent generations have become more powerful, general purpose systems have emerged that have added programmable switches to configure the system dynamically.

All systems in this class offer explicit communication primitives that more or less follow Hoare's model of communicating sequential processes [62]. The most ardent followers are transputer systems that use OCCAM, a CSP language with emphasis on correctness. However, the more common approach is to add a library of *send* and *receive* primitives to a conventional language such as C or Fortran. Differences exist as to the degree of abstraction from the actual communication network. The trend is toward generic send and receive operations that can exchange data among any nodes in the system. Lower level primitives expose the communication network to the user. For example, Transputer systems require the application code to be aware of the network topology. Even machines capable of automatic routing, such as JPL's Mark-III, allow routing controlled by the application program (under the *Crystalline* operating system [101]).

The rationale for user controlled routing is a perceived performance advantage at the expense of programming convenience. In a sense, this tradeoff is similar to the pro's and con's of virtual memory. It will be argued that direct routing control is unnecessary because the hardware implementation achieves almost optimal performance.

Automatic, generalized routing support does not abandon the ability to fine-tune an application by carefully matching process and data allocation to the network topology. Just as the paging behavior of programs can be optimized by proper data allocation, the communication demand of a parallel program can be reduced by a good decomposition. However direct routing control - roughly equivalent to explicit memory overlays - is not needed as long as the network is well characterized. For example, the cost for nearest neighbor communication ought to be smaller than the cost of sending packets to more distant nodes.

1.1.3. Virtual Shared Memory Systems

The current dominance of private memory systems is partially due to their implementation simplicity. Explicit communication also requires less communication bandwidth, as only the relevant data is moved [92]. Because nodes of private memory machines can be physically small, cost effective, and powerful, systems composed of such nodes are highly scalable and can be customized for particular applications. Furthermore, private memory machines are quite similar to distributed systems and hold the potential to be extendible over a local area in a transparent fashion.

However, it is widely recognized that explicit communication requires more programming effort [68]. Dr. Heath's "*Shared Memory is great while it lasts*" at the second conference on Hypercube Multiprocessors met with little dissent. Hence it is common that systems are being proposed and built that try to combine the virtues of both worlds. An overview of such hybrid systems will be given in Chapter 4.

1.2. The Structure of This Thesis

This thesis is structured into three broad chapters as outlined in Figure 1-1. Chapter 2 selectively reviews the foundations of communication networks. It is selective in the sense that only areas applicable to high bandwidth, short-haul networks are covered. Generalizations and trends that can guide a physical implemention or that can prune part of the vast design space are emphasized. Chapter 2 also covers the previous and related research on message-passing machines.

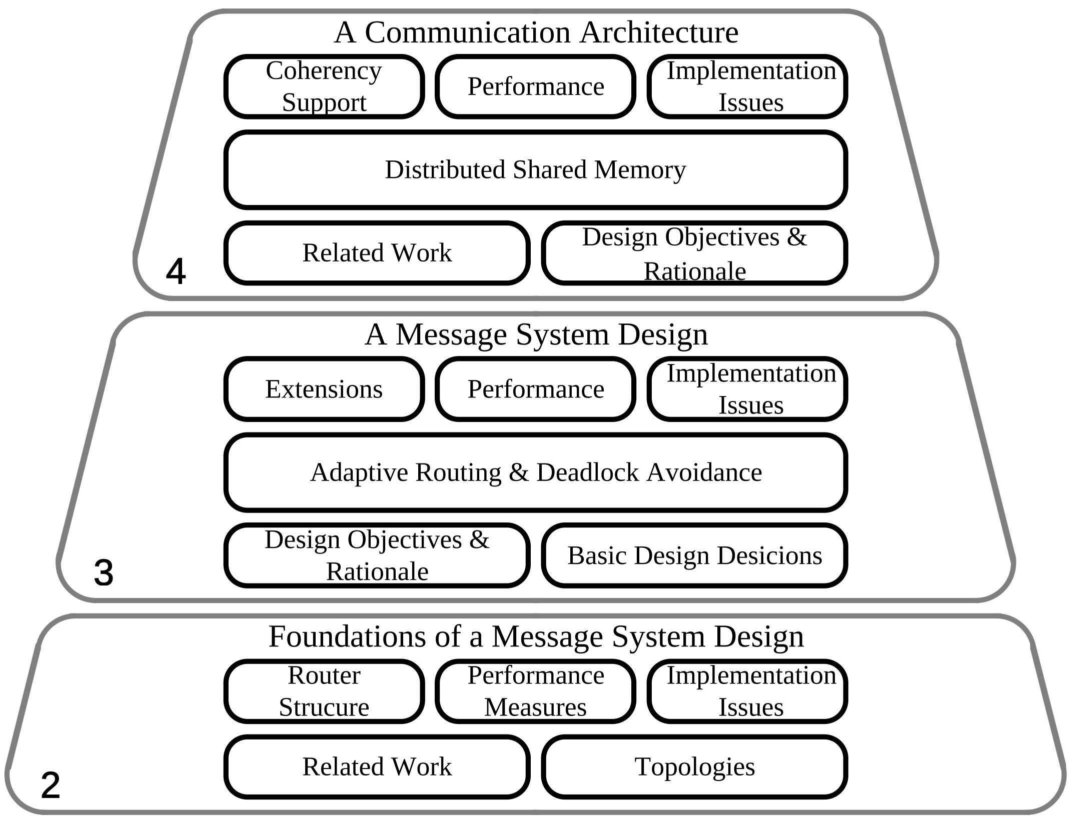


Figure 1-1: Thesis Structure

Chapter 3 describes the design and rationale of a new message-passing system. This design is based on the foundations presented in the previous chapter. The emphasis has shifted from a broad overview to a tightly focused design. The key objectives include high utilization of the physical resources under high load conditions, topology independence, easy integration with the shared memory programming model and compatibility with current implementation technologies (such as VLSI). Some of the design rationales and objectives originate from the communication system that is built on top of this message-passing architecture and is described in Chapter 4. There are some interdependencies between these two designs. Acknowledging the general bottom-up philosophy underlying this thesis, the lower-level message-passing system is described first.

While the message system is intended to be used as a building block for a specific communication architecture, it is evaluated in its own right. This evaluation uses a set of traffic patterns and load conditions that were deliberately selected to model the most adverse and demanding operating environment. It is this area of highly dynamic and irregular traffic in which effective routing strategies are important.

Chapter 4 presents the design of the entire communication architecture, integrated with high-end processors into a multiprocessor system. This chapter is subdivided into a discussion of the proposed programming model, its operational characteristics, the intended scope of the system and an evaluation of the expected performance. Implementation details are covered to the extent necessary to defend the proposed design decisions.

The description of important tools used in this thesis is deferred to the appendix, as is the discussion of some interesting engineering problems that arose during work on this thesis.

1.3. Methodology

This thesis is based primarily on simulations. Three generations of simulators, some of which gained external use[1], were written and used during the course of this research. In all cases, attention was paid to insure that the simulations were based on realistic assumptions about the underlying hardware. To aid this realism, several implementation studies were carried out.

Further strengthening the bonds to reality were experiments performed on the *MARK-III* hypercube system at the Jet Propulsion Laboratory in Pasadena CA, on a *Ncube* multiprocessor at the National Bureau of Standards, on a locally available Intel iPSC/I, and on an Encore Multimax multiprocessor system.

Supplementing the simulation work are analytical studies. These were greatly aided by

---

[1]*CBS* was the 2nd simulator written during this thesis and was used by Prof. A. Gupta at Stanford for some of his research and for a parallel processing class.

symbolic manipulation tools such as *MACSYMA*[2] and by *GreatSPN*[3], a tool to specify and analyze computer systems by means of generalized petri nets [91, 97].

---

[2] *MACSYMA* is a symbolic manipulation program developed at the MIT Laboratory for Computer Science.

[3] *GreatSPN* was developed by Giovanni Chiola, Dipartimento di Informatica, Universia' degli Studi di Torino, corso Svizzera 185, 10149 Torino, Italy.

Chapter 2
Foundations of a Message System Design

This chapter maps the design space for message systems of network-based multiprocessors. Related research results and experience with commercial and academic ensemble machines are reviewed to provide reference points. Six selected topology families are described and are subsequently used to show various tradeoffs available to the message system designer. While not all possible network topologies are covered, a very wide range of networks with widely different characteristics are shown.

The structure of the routers in each network node interact with the network topology selection. Two basic design approaches - crossbars and register files - are analyzed. It will be shown that register file based routers offer superior performance.

The primary performance measures for message systems are their bandwidth and latency. It will be shown that bandwidth and latency are generally conflicting goals. However, optimization for both bandwidth and latency is possible if networks are operated with fine grained, independently routable data.

2.1. Related Work

Fujimoto designed a VLSI-based communication building block for multiprocessor networks [49]. Queuing theoretical models of the architecture demonstrated the superiority of virtual-cut through routing [71]. In the analysis of the design options, he discovered that low fan-out networks lead to reduced latencies if the total I/O bandwidth per node is held constant. The constant bandwidth constraint originates from the packaging and interconnection technology of VLSI components. The delay caused by the actual transmission of data across the network dominates the transit delays for each intermediate node; hence it is beneficial to increase I/O bandwidth by reducing the number of I/O channels per node at the expense of an increase in network diameter.

Fujimoto proposed a table-driven, topology-independent router. This allows the use of low fan-out networks with low diameter. Optimizing networks by this measure is subject to ongoing research in graph theory [37, 65, 94]. The list of the largest known graphs is maintained by J.C. Bermond of Universite de Paris-Sud. A recent *top ten* list is given in Table 2-1.

Diameter	Fan-Out = 3	Fan-Out = 4	Fan-Out = 5
2	10	15	24
3	20	40	70
4	38	95	182
5	70	364	532
6	128	734	2742
7	184	1081	4368
8	320	2943	11200
9	540	7439	33600
10	938	15657	123120

Table 2-1: Maximum Network Sizes for a given Fan-Out and Diameter

Stevens [139, 138], and later Dally [36], also recognized the advantage of low fan-out networks, but opted to use simpler but less efficient networks. Further reducing the router complexity is wormhole routing [128], which blocks the transmission path instead of resorting to a buffer in case the outbound resources of a transient node are busy. The omission of buffers causes severe network degradation under high load conditions.

Common to these designs is the assumption that all data of one message traverse the same path. It turns out that this is the main reason that optimizing for bandwidth conflicts with optimizing for latency.

2.1.1. Packet Switching vs. Circuit Switching

Packet-switched networks require nodes to completely receive a packet before further transmission. Hence the transmission times of each channel along the path of a packet accumulate. The complete reception requirement prevents parallel transmissions of data from one packet over multiple point-to-point channels. Circuit switched networks avoid this cumulative delay by establishing a transmission path between source and destination first and then transmitting the data across the entire path with minimal delay in each transient node.

Virtual cut-through tries to combine the virtues of both approaches: it uses circuit switching whenever possible, but it doesn't block the communication channels by resorting to packet switching instead.

Circuit switching is used in the iPSC/2 [105] and the Hyperswitch [28]. The *Direct-Connect* communication technology of the iPSC/2 sends a probe along a deterministic path (E^3-routing) and transmits data once the path is completed. The probe of the Hyperswitch is more elaborate and is able to search for a path through the network.

Circuit-switched networks must amortize the cost for establishing the data path over the entire message. The 48bit probe packet needs about 900nsec to traverse a node. Therefore if there are only a few bytes of data to transmit, circuit switching becomes unattractive.

Besides being inefficient for short messages, the blocking of communication resources becomes a problem when the network load is high. In this case the probability for blocking is high and a substantial amount of network bandwidth is wasted in unsuccessful connection attempts.

2.2. Network Topologies

Research on communication network topologies dates back to the invention of the telephone and has resulted in a vast body of theoretical and practical results [154]. With the advent of computer networks and multiprocessor systems, research interest was renewed, and has led to numerous competing topologies with no universal winner.

The selection of an optimal topology is an engineering compromise that has to balance several conflicting requirements. For example, topologies that are easy to build tend to have lower bandwidth. Some factors that influence the topology choice are system size, scalability, bandwidth, implementation technology, spatial distribution, fault tolerance, and intended applications:

- For systems with up to a few hundred processors, wiring and connectors are minor problems. However, topologies for larger systems are constrained by the cost of electrical and mechanical interconnection.
- Scalable systems have fewer variables that can be optimized because the system size is no longer fixed at design time. Increasing system size should be possible at a constant - or even decreasing - cost to performance ratio, which is quite difficult for a large range of scale. Other factors include a low upgrade complexity and an incremental extension capability.
- Different topologies vary in their efficiency in using the basic channel speed, which in turn depends on the implementation technology. Systems with identical channel speeds but different topologies would provide different effective bandwidths that are dependent on the message traffic pattern. A uniform message distribution is usually used to compare the effective bandwidth of different topologies.
- The feasibility of a particular topology can depend on the implementation technology: small, densely packed processing nodes on PC-boards or ceramic carriers impose certain limitations on the number of channels (wires) and the interconnect pattern (physical routing). Packaging technology imposes pinout

limitations. Media with expensive interfaces (optical fibers, high speed transmission lines, wide busses) will favor topologies with fewer channels.

- If processing nodes or clusters of nodes are physically distributed, the number and routing of the channels between them become important. For example, it is impractical to have multiple wires emanating from workstation-sized processing clusters and to route each of these to arbitrary places. The spatial distribution will impose constraints on the topology.
- Fault tolerance demands redundancy from the topology and it requires practical methods to deal with faults. For example, routing around a failed channel should not require the execution of complex algorithms.
- If the class of intended applications can be characterized by a dominant message traffic pattern, the topology can be structured to support such a pattern efficiently. Static decompositions with regular dataflow or a high degree of spatial locality are examples of such applications.

Proponents of a particular multiprocessor technology tend to place different weights on these factors and arrive at different conclusions. Enthusiastic arguments over the merits of competing topologies are frequently based on such differences in the fundamental assumptions.

This thesis claims that communication network topology is an overrated issue lacking the potential for an ultimate solution. Consequently, the proposed architecture is virtually topology independent[4]. This approach allows the comparison of different topologies implemented in the same technology with the same underlying assumptions. These results provide guidelines for selecting a particular network configuration given sufficient information on the design parameters.

2.2.1. Topology and other Definitions

The topology of the interconnection network is a strongly connected, directed graph $G:(V, E \subseteq (V \times V))$ with $N=|V|$ nodes. The vertices of the graph can represent two different types of nodes: *terminal* and *intermediate* nodes. A terminal node represents a processing element that can produce and/or consume messages. Intermediate nodes can neither source or drain messages: they are plain routers or switches. Aside from this difference, all nodes are capable of routing messages, i.e. the router or switch is an integral part of each node. Naturally, a node with only one attached channel tends to have rather simple routing capabilities.

Each edge in the graph represents a channel that can transmit a packet. In the implementation described in later sections, channels can be of two types that differ in their

[4]Usable topologies must conform to a few fundamental restrictions, such as strong connectivity.

bandwidth and latency. However, as far as this discussion on topologies is concerned, all channels are created equal.

The actual implementation of a communication channel may support data transfer in both directions. Such bidirectional channels are represented as two edges with opposing directions that connect two nodes. Half-duplex channels cause some complications because the two channels are no longer independent: only one channel can carry information at a time. As a first order approximation, two channels with one-half capacity are used.

2.2.2. Topologies under Consideration

The following list of topologies is meant to be representative but not exhaustive:

k-ary Hypercubes are characterized by the *arity* k and the dimension d. Node V_i is connected to node V_j **iff** [5]:

$$\exists x \in [0, d-1] : (digit_k(x, i) + 1) \bmod k = digit_k(x, j)$$

The most popular members of this family of topologies are rings ($d=1$), toroids ($d=2$) and binary hypercubes ($k=2$).

k-Shuffles are characterized by the base k and the dimension d. Node V_i is connected to node V_j **iff**:

$$\exists x \in [0, d-1] : ((i \cdot k) \bmod k^d) + x = j$$

Cube Connected Cycles are characterized by the *arity* k and the dimension d. Node V_i is connected to node V_j **iff**:

$$\exists x \in [0, d-1] : (digit_k(x, i/d) + 1) \bmod k = digit_k(x, j/d) \wedge i \bmod d = j \bmod d$$

or

$$(i+1) \bmod d = j \bmod d$$

or

$$(i-1) \bmod d = j \bmod d$$

Cube connected cycles are essentially k-ary hypercubes with each node replaced by a ring of d nodes, such that each node is connected to a constant number of channels.

Fat Trees are binary trees of depth d with 2^d leaf nodes and $2^{d-1}-2$ intermediate nodes. They differ from a conventional tree by increasing the channel bandwidth as a function of the distance to the leaf nodes. For a given traffic pattern, this function can be chosen such that there are no bandwidth bottlenecks in the networks.

Stars connect N nodes to a central routing node. In the case of larger networks, the central router can become a collection of routing nodes that form a different topology.

Random Graphs are characterized by the number of nodes N and the number of channels

[5] $digit_k(x, i)$ denotes the x^{th} -most significant digit of i expressed as an k-ary number.

d originating from each node. To form a viable network, the graph must be strongly connected. In order to ease the analysis, the random graphs considered in this thesis have an embedded ring of *N* nodes and each node of the graph has exactly *d* inputs and *d* outputs.

These families of topologies cover the networks of most current implemented and proposed architectures for ensemble machines. Notable exceptions are hexagonal grids [138, 139] and trees with embedded rings [132, 39].

2.2.2.1. k-ary Hypercubes

Binary hypercubes became popular with Caltech's Cosmic Cube [127] and were subsequently used in numerous research and commercial systems [89, 57, 129, 7, 22].

Some reasons for choosing the binary hypercube topology are:

- Routing is simple. The exclusive or-ed binary representations of source and destination address yield a bitvector in which an one corresponds to every dimension that the packet has to traverse. By ordering this sequence of transmissions, a precise route of minimal length is established ($\mathbf{E}^3$ - routing [142]). This property greatly simplifies deadlock avoidance.
- Binary hypercubes are scalable by doubling the number of processors. The network has a low, logarithmic diameter.
- Many application-specific topologies - for example rectangular grids, binary trees, etc. - can be embedded in a binary hypercube. This is important for some systems that lack hardware-assisted routing facilities or applications that require minimal communication delays. In both cases, the user has to deal with the topology explicitly.

Some of the disadvantages of binary hypercubes include the problems of implementing large systems because the connection pattern does not match conventional implementation methods, such as printed circuit boards which favor planar graphs. The number of channels emanating from each node often limits the bandwidth. For example, integrated circuits support only a modest number of off-chip connections. This led to bitserial channel implementations [89, 57, 144] at the expense of increased latency. It has been argued that modest-sized systems with lower dimensional networks can lead to lower overall latencies [34, 129, 31, 88, 130].

2.2.2.2. k-Shuffles

The class of shuffle-exchange networks has received less attention as an interconnection topology for ensemble machines. This is partially due to the more complex routing. Application-specific connection patterns are harder to embed and the larger system configurations are not naturally composed out of smaller ones.

Figure 2-2 depicts a 2D 4-shuffle. This network has a diameter of two while a comparable binary hypercube has a diameter of four. The advantage of k-Shuffles in general is their low network diameter.

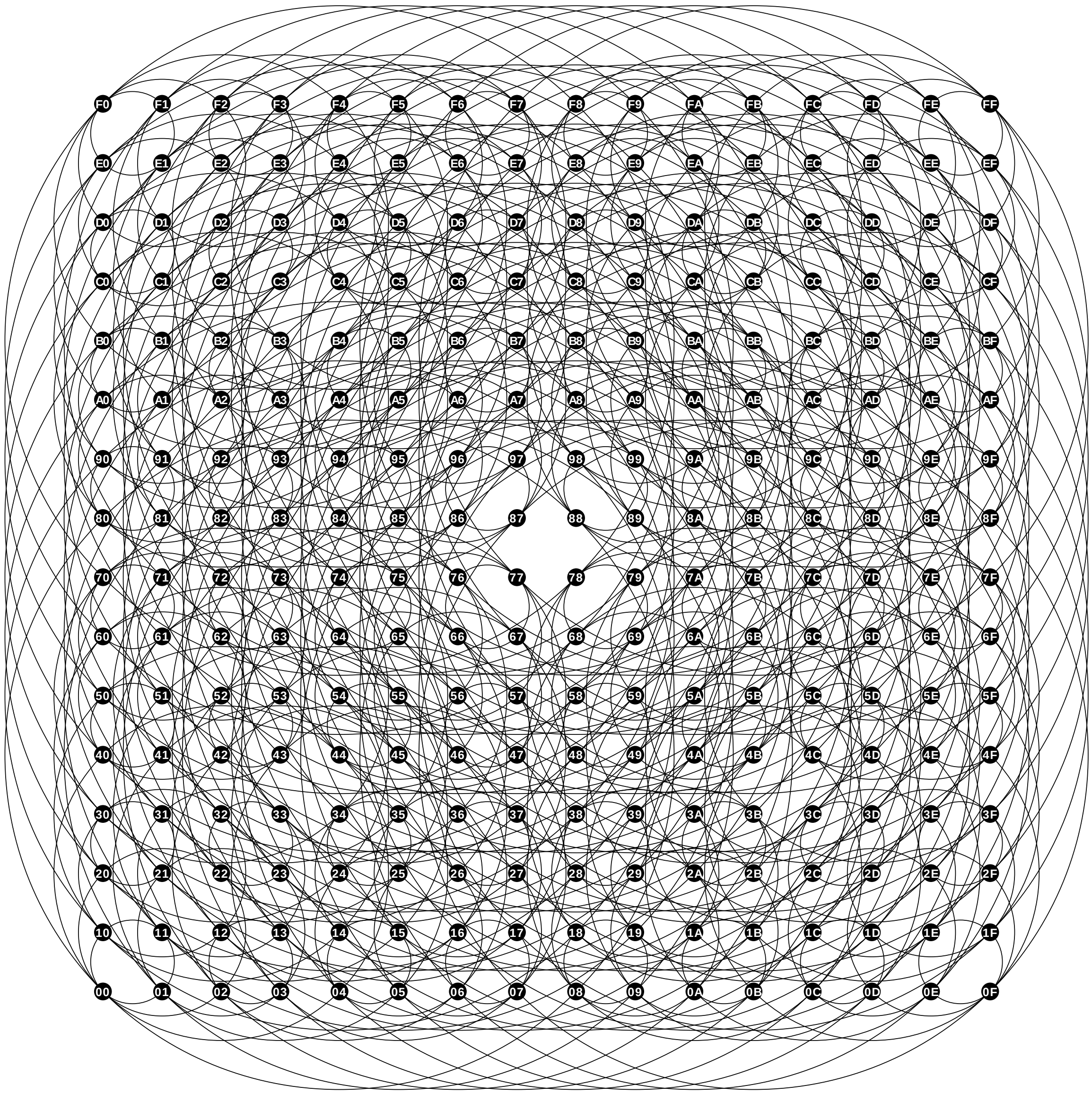


Figure 2-1: 256 node binary hypercube

2.2.2.3. Cube Connected Cycles

The cube connected cycle topology has a constant number of connections per node. The structure of one node does not have to be changed as the system scales up. Because only three ports are required, each port may have a higher bandwidth.

There are two classes of channels in a cube connected cycle that will carry different traffic loads. Intra-cycle channels connect the nodes of one cycle while inter-cycle channels connect nodes of different cycles. As system size increases, the intra-cycle channels will have to carry more transient traffic. The ratio of intra-cycle to inter-cycle channel utilization increases

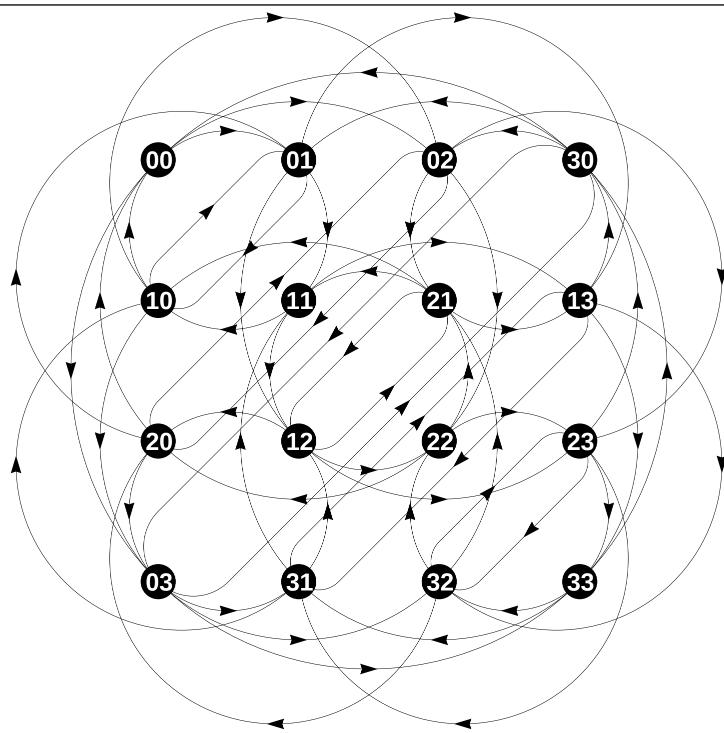


Figure 2-2: 2-Dimensional 4-Shuffle Topology

with total system size. A balanced system may allocate more bandwidth to the intra-cycle channels in order to offset this effect. The cube connected cycles considered here use two channels within a cycle to form bidirectional links. If the arity of the cube is higher than two, the inter-cycle channels become unidirectional and form cycles across the dimension of the cube. So two of the three ports for a node are dedicated to intra-cycle traffic and one port carries inter-cycle traffic.

2.2.2.4. Fat Trees

The fat-tree topology tries to preserve the clean layout of a binary tree without the bandwidth bottleneck of the root channel. As indicated by the line width in Figure 2-4, the channel bandwidth increases with the distance to the leaf nodes. Processing nodes are attached only at the leaves of the tree. The intermediate nodes are just routing elements.

The proponents of fat trees point out that this topology can simulate any other topology at a polylogarithmic increase in cost [84]. This addresses the problem of embedding application specific communication patterns into the tree. However, given efficient hardware support for routing, this layer may not be exposed to the user directly.

The *clean* layout of trees is significant for the construction of very large systems that are constrained by the wiring.

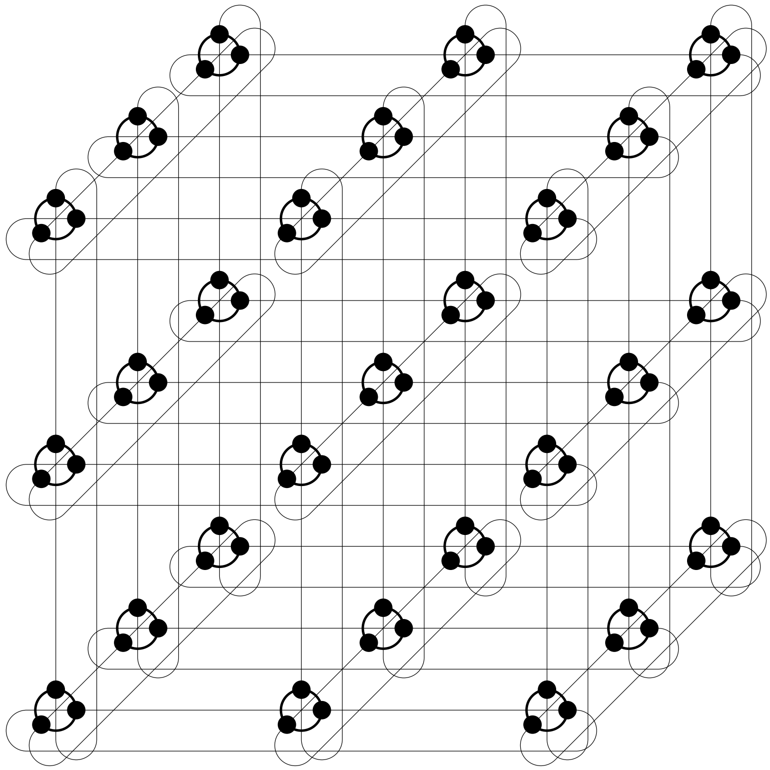


Figure 2-3: 3-Dimensional 3-ary Cube Connected Cycles Topology

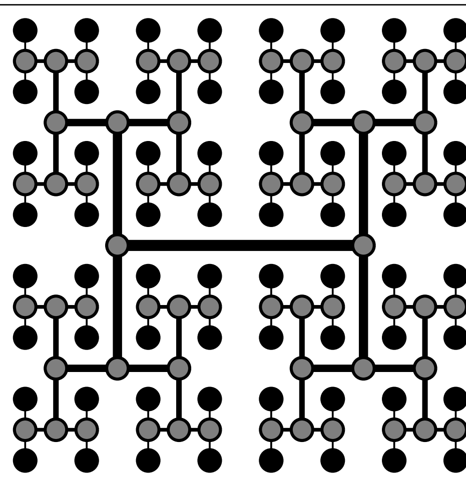


Figure 2-4: A 64 Node *Fat-Tree* Topology

Since channel bandwidth does not come in arbitrary increments, the increased bandwidth is actually achieved by running several channels in parallel. All channels have the same bandwidth. The total bandwidth of a bundle of channels is selected such that the system is balanced under uniform traffic[6]. The bandwidth of leaf node channels is set to 1. The bandwidth of the other channels is:

$$B(n) = 2^n \frac{N-2^n}{N-1} \tag{2.1}$$

where n is the distance from the leaves and N is the total number of nodes. $\lceil B(n) \rceil$ channels are used to implement this topology in this thesis. It should be noted that the channel allocation depends on the total number of nodes. This is undesirable because it destroys the simple recursive composition of larger systems out of small ones: the configuration of a sub-tree depends on the system size. On the other hand, setting the 2nd factor of Equation (2.1) to 1 is quite costly.

Large routing nodes must be composed out of several routing elements because it is unreasonable to assume unbounded fan-in/out on the routing elements. Since there is little freedom in routing decisions, the distribution of channels to routers has little impact on performance. A slight degradation due to load imbalance of the different routing elements of a node is minimized by redistributing the outbound channel of one routing element across the routing elements of the next layer. The degradation decreases rapidly with the feasible fan-out of a routing element. A maximal fan-out of 16 was used in the simulations. In case of analytical approximations of the system, an unbounded (optimistic) fan was assumed.

2.2.2.5. Stars

A centralized router with nodes directly attached via dedicated channels is attractive if the channels are scarce resources. This situation may exist if the channels are expensive or if connections between nodes are difficult to implement. In case of larger systems, the central node may consist of several routing elements that use some other topology for inter-router communication. Because pure star configurations are not scalable, only small systems of this type were considered.

2.2.2.6. Random Graphs

A random network topology is constructed by starting with a ring of N processors. This uses one channel of each node and insures that the resulting graph is strongly connected. It should be noted that there are strongly connected graphs that do not have an embedded complete ring. However, a reasonably broad class of topologies are covered; and in return for this simplification, both construction and analysis are greatly eased.

For a given number of channels d, additional connections are added randomly until each node has d incoming and d outgoing connections. Simulation results are based on an actual

---

[6] All destinations for a given source node are equally likely.

network that was constructed with a pseudo random number generator. For some analytical results, the average over all possible graphs is used.

2.3. Router Implementations

Each node of a multiprocessor interconnection network has a number of input and output ports that can be connected to ports on other nodes to form the actual communication channels. Bidirectional channels are formed by combining an input port with an output port. The distinction between input and output, and the implied directionality is useful because all actual implementations are sensitive to the data flow[7].

All input ports are created equal and packets received on any input port can be routed to any output port. Routers can have a finite amount of buffer space for the temporary storage of transient packets. Network nodes with processing elements may provide a datapath into the router that differs in function and bandwidth from the external ports.

This concept of a routing element is influenced by current implementation technologies. At the packaging and interconnect level, the number of I/O signal paths is quite limited. Since these wires are a scarce resource, optimal use of these signal paths is mandatory. A high performance routing element requires a number of tables, buffers and data paths plus a substantial amount of glue logic. This demand appears to be well within current integration levels for VLSI circuits. However, I/O limitations and integration level suggest that only one routing element will fit on one chip conveniently. This adds to the fact that high end processing elements[8] require several chips (processor, cache, memory and some controller / glue logic). Furthermore, the modularization into routing elements as a functional building block with well-defined interfaces helps the overall system design process.

On the down side of this abstraction, more unconventional approaches such as runtime-configurable gate arrays (*Logic Cell Arrays*, [156]), and cellular automata are not considered.

There are two basic structures for a routing element in the sense outlined above: crossbars and multiported register files.

A crossbar is a stateless combinational circuit that can realize any input to output port permutation plus any multicast configuration (one input port is connected to more than one output port). Usually, each output port is a multiplexor that is connected to each input port

[7]True bidirectional connections are electrically challenging and require extra time to switch directions.

[8]The current experience with multiprocessor systems implies that a machine with few high performance processors is more efficient and easier to program than a system with many low performance processors of comparable cost.

and individually controlled. While the crossbar is stateless, it does have a signal propagation delay that can be significant for large switches. Therefore it is reasonable to assume a pipeline register on either the input or output ports of a crossbar switch.

A multiported register file is capable of concurrent read and write operations to different locations. Input / output ports are associated with the write / read ports to the register file. Right down to the gate-level, multiported memory is quite similar to a set of pipeline registers sandwiched between two crossbars. The primary difference is that the pipeline registers are no longer associated with a particular port. If the time constraints permit, implementations can be simplified by time-domain multiplexing. Other reasons that favor a regular register structure will be discussed in subsequent sections.

The merits of each approach - pipelined crossbar or multiported register file - depend heavily on how the system will be used. First-order characterizations of the router environment are:

- Packet vs. circuit switching
- Asynchronous vs. synchronous operation
- Fixed length vs. variable length packet support

The area of routing strategies has less impact on the basic structure of the router and is discussed in section 3.5.

The distinction of circuit switching vs. packet switching has lost precision with the introduction of methods that try to combine the advantages of both worlds [71]. For example, *wormhole routing* [36, 128] typically subdivides a packet into smaller flow control units (*flits*) that are treated in store-and-forward fashion, while all resources (channels, buffers, etc.) along the transmission path are bound to one packet for the duration of the complete data transmission. Hence wormhole routing has some circuit switching characteristics (resource allocation constraints). If true circuit switching is the mode of operation, a register file is not a viable implementation.

The definition of synchronicity depends on which system level is being considered. At this point, the very basic implementation options are under investigation. In a synchronous implementation, signals of the communication channels are derived from a central clock, while in an asynchronous system each processing element may use its own, independent clock. So the question is whether knowing that packets can arrive only at well-defined points in time alters the expected performance.

Most current systems deal with variable-length packets. For various implementation reasons (limited buffer space, fixed number of bits in the length field, limited span of error detection and recovery codes, etc.) the maximum packet size is usually bounded. The permissible message lengths at the transport layer may exceed the range of packet lengths supported at the data link layer. Therefore, it is usually necessary to have an agent in the system to break messages into smaller units and to reassemble them on the receiving site, regardless of whether the data link layer can handle variable length packets.

2.3.1. Router Performance: The Model

In order to assess the fundamental tradeoffs in the design of a routing element, generalized stochastic petri nets (GSPN's, Appendix B) are used [96, 26]. This approach is based on several simplifying assumptions:

Uniform address distribution:
: Packets entering the routing element are addressed to any of the output ports with equal probability. It is possible to deviate from this assumption by assuming a different, static distribution to explore the behavior under asymmetric load conditions. For example, the presence of a *hot spot* in the network may cause output #0 to be a more likely destination.

Steady state equilibrium:
: Results are averaged over long sequences of operations under constant traffic conditions.

Preservation of packets:
: The router does not generate or consume packets.

None of these assumptions is valid in an actual system nor is an analysis of a routing element in isolation likely to yield quantitative data on the entire network performance. However, using these assumptions is a common and useful approach to a preliminary exploration of the design space. Routing elements that function poorly in isolation are not likely to be good building blocks for a high performance communication network.

The GSPN of a plain 2 by 2 crossbar is given in Figure 2-5. Tokens in the input places (*Input1* and *Input2*) represent incoming packets. The immediate transitions *i1* and *i2* are enabled by the presence of a token in *I1idle* and *I2idle* if that input port is ready to accept a packet. Once a packet is accepted, the immediate transitions *I<src>O<dst>sel* decide which output port is addressed. Asymmetrical address distributions are handled by different firing rates that are proportional to the transition probabilities. In this model, a packet is addressed to a particular output port. Actual implementations may deviate from this restriction by allowing a packet to take alternative routes, potentially depending on local traffic conditions. These adaptive routing methods will be discussed later. The restrictive routing represented in Figure 2-5 is a fair description of the E^3 routing method. Adaptive routing will differ from this, but it should be noted that the majority of packets in an adaptive system under uniform load conditions will have few alternatives (see Figure 3-18).

Idle output ports are represented by a token in *O1idle* and *O2idle*. An output port can pass one packet at a time. The timed transitions *Xmt1* and *Xmt2* model the time necessary to transmit a packet. Wide, solid bars represent deterministic transitions that fire a fixed

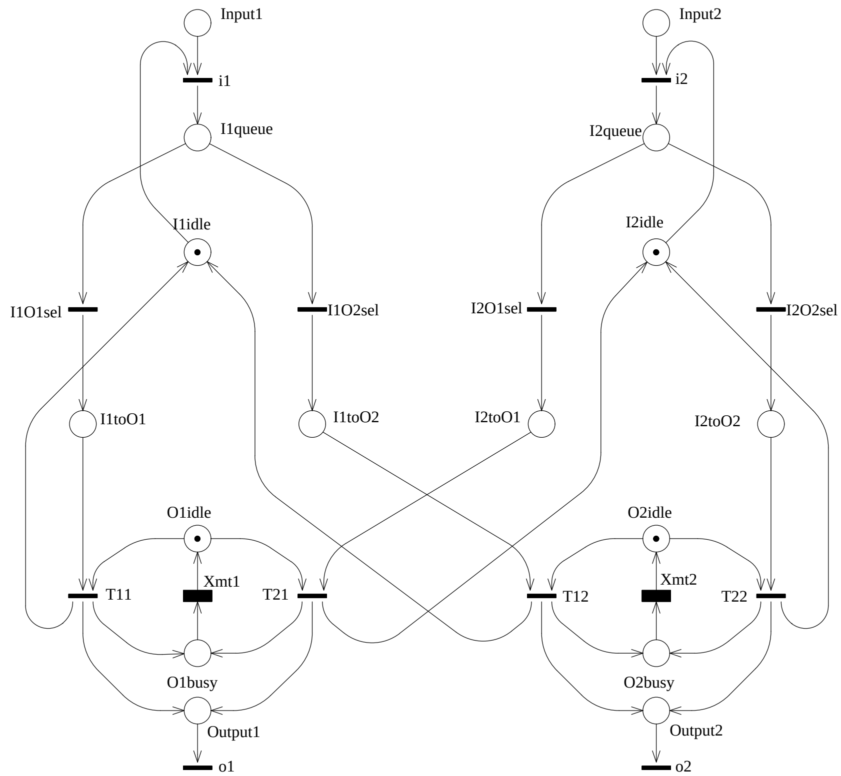


Figure 2-5: GSPN for a 2 by 2 crossbar

amount of time after the transition has been enabled. The output transitions *o1* and *o2* pass packets (tokens) back to the surrounding environment.

The net for a register file based router (Figure 2-6) is simpler because packets *"forget"* where they came from once they are stored in the file. This property allows the inputs to be collapsed into a single port with appropriately increased bandwidth. The place *Iidle* has a token for each unused register. Once a packet is admitted by transition *i*, it is assigned to a particular output. The transition rates of *t1-4* can be adjusted to model non-uniform address distributions. They are equal in the case of the uniform distribution assumption. As in the crossbar case, transitions *Xmt1-4* control the time to send out a packet.

The environments used to exercise the router are outlined in Figure 2-7. Place *Input1* is connected to an infinite packet source that will saturate that input. The transition *Isource* is enabled whenever there is no token in *Input1*. This provides an asynchronous saturation load. The inhibition arc prevents flooding.

Place *Input2* is connected to a negative exponentially timed transition with a certain rate. Due to the timed nature of the transition, no flow control is needed as long as the packet insertion rate does not exceed the capacity of the router.

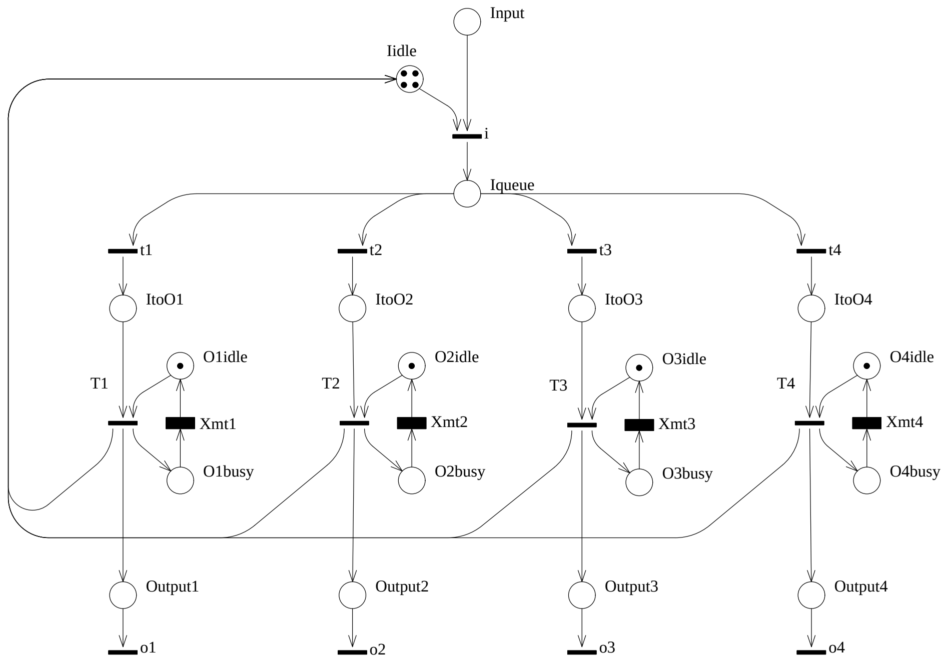


Figure 2-6: GSPN for a 4 by 4 register file

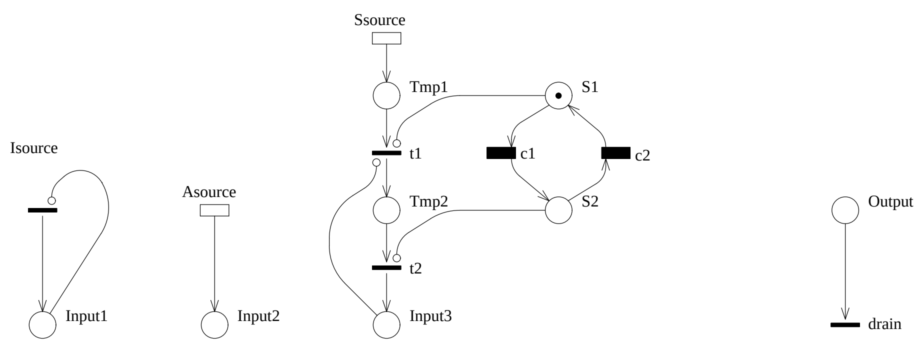


Figure 2-7: Environment GSPNs

Place *Input3* is connected to a synchronized load generator. The actual packet source can be of any type. The transitions *t1* and *t2* act according to the revolving door principle: they are never enabled at the same time. Places *S1* and *S2* form a clock generator that is shared among all load generators. The delay of transition *c1* is very short so that the cycle time is almost entirely determined by transition *c2*. The sum of both delays is equal to the inverse of the output port transition rates.

Output ports are simply connected to *drain* transitions that remove tokens from the network. Maintaining a bounded number of tokens keeps the analysis tools happy.

2.3.2. Router Performance: Saturation Throughput

One measure of the performance of the routing element is its saturation throughput. Given an infinite supply of packets connected to all inputs, the saturation throughput is defined by the utilization of the output ports. If the output ports are always busy, 100% utilization of the outbound bandwidth is achieved. Table 2-2 lists the steady state fraction of output port busy time under the uniform address distribution assumption[9]. The data for variable length packets assume a negative exponential distribution.

Normalized Saturation Throughput							
Operation Mode	Packet Size	Crossbar 2x2	Crossbar 4x4	Crossbar 8x8	Register File 2x2	Register File 4x4	Register File 8x8
Synchronous	Fixed	0.833	0.775	0.752	0.833	0.775	0.752
Synchronous	Variable	0.738	0.657	0.630	0.738	0.657	0.630
Asynchronous	Fixed	0.833	0.775	0.752	0.833	0.775	0.752
Asynchronous	Variable	0.750	0.669	0.640	0.750	0.669	0.640

Table 2-2: Basic Router Performance

Due to the traffic saturation, the input structure has no impact on the result. The lack of a difference between the crossbar and register file figures is merely testing the analysis tools. There is also no difference between asynchronous and synchronous operations on fixed length packets, which is due to the output ports synchronizing each other in this model. Self-synchronization depends on the saturation condition: if the load is reduced slightly, the (short) idle time between packet arrivals will disturb the lock-step operation. In order to prevent a synchronous start for the asynchronous model, the enabling tokens are placed into the output-idle places via a non-deterministically timed transition. This has no effect on the steady state condition as long as the saturation persists.

It is surprising that synchronizing the inputs reduces throughput by less than 2% in the variable packet size case. Exponentially distributed packet sizes are probably on the extreme end of *variableness*. A bimodal distribution - typical for Ethernet traffic - is probably a more realistic assumption. The unrepresentative traffic on the 8th floor subnet of the CS departmental Ethernet on February 15, 1989 suggested a 25 to 1 ratio in packet sizes and a 65% fraction of small packets. Figure 2-8 gives the saturation throughput for an 8 by 8 router operating asynchronously on uniformly addressed packets.

This evidence suggests that systems designed to work efficiently under high loads perform

[9] Most of these results were obtained by Monte Carlo simulation. The smaller nets were solved directly with *GreatSPN* [26]. The simulation errors at the 99% confidence level are 0.1% or better.

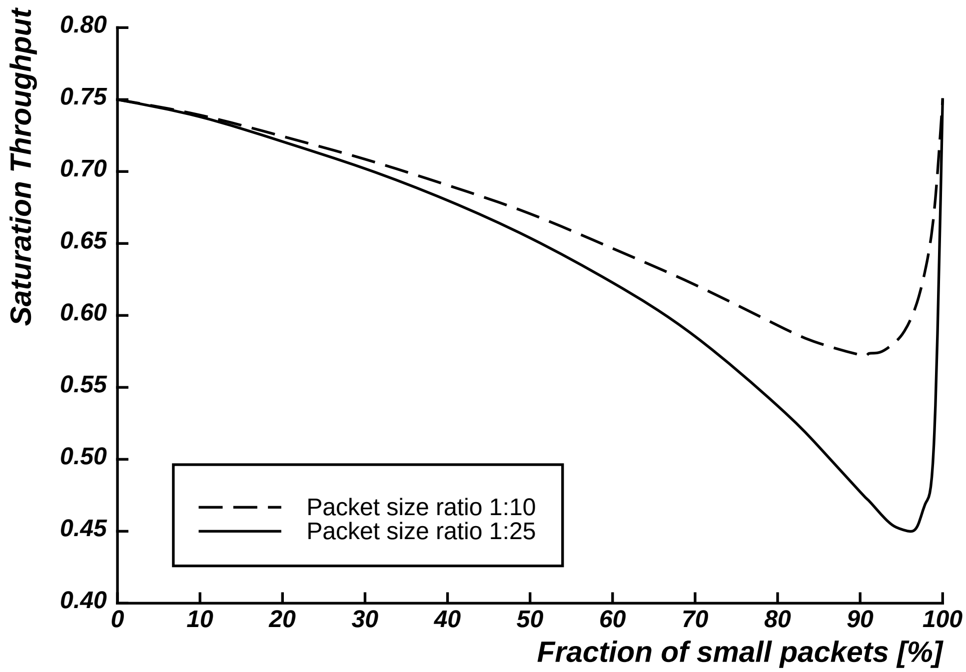


Figure 2-8: Saturation throughput for bimodal distributed packet sizes

better if they operate on a fixed packet size. It should be noted that the worst loss in bandwidth occurs when a few large packets interfere with a series of predominantly short packets. It turns out that this is precisely the type of traffic that a message-based, shared-memory multiprocessor will have: a large number of small read/write packets and a small number of large page transfers.

A register file based router can have more registers than input or output ports. The net in Figure 2-6 can model such a design by increasing the number of tokens in the *Iidle* place. Similar changes to a crossbar are somewhat more difficult because each storage element is dedicated to either an input or output port. Therefore the extra resources added to a register file design can be expected to be better utilized because they are shared by all channels. It will be shown later that the register structure is convenient for implementing a number of adaptive routing functions.

Figure 2-9 is based on an 8 by 8 register file router that operates asynchronously. A modest amount of bandwidth increase was observed from adding extra buffer space. It is interesting to note that variable-length operation benefits more from this extra storage than fixed-length operation.

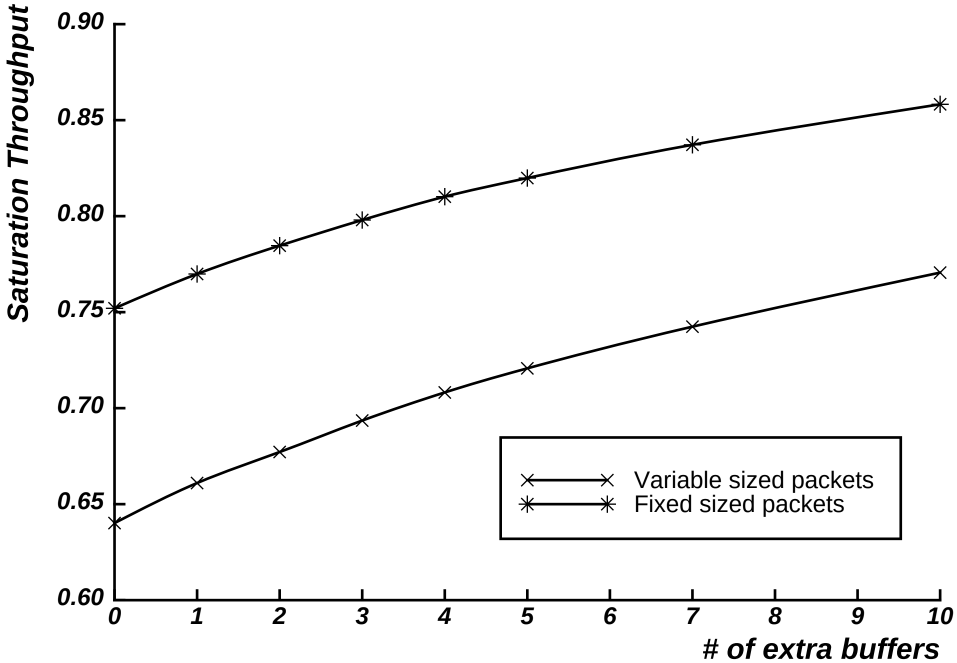


Figure 2-9: Saturation throughput with extra buffer space

2.3.3. Router Performance: Latency

Another indication of potential router performance is the amount of delay that each packet experiences before it passes through the router. Again, given the stochastic Petri net models, the latency can be computed based on uniform source and destination address distributions. The delays between packet insertions follow a negative exponential distribution. The uniform source distribution assumption means that each input channel carries the same load. This does not matter for the register file based router, which essentially sees only the combined rate. It does, however, change the performance of the crossbar router considerably. Uniform input distribution is actually a best case assumption for the crossbar: any imbalance degrades performance even further.

The latency of a packet is defined here as the extra transfer time that occurs if a router is inserted into the transmission path. A plain wire is said to have latency 0. The actual transmission time is unavoidable and is not part of the latency contribution of a router. Latencies are expressed in terms of the average packet transmission time, so a latency of 1 indicates that the average transmission time doubled.

For each configuration, the packet input rate is slowly increased until the latency exceeds 10, an arbitrarily chosen pain threshold. The input rate, or offered load, is normalized to the aggregate bandwidth.

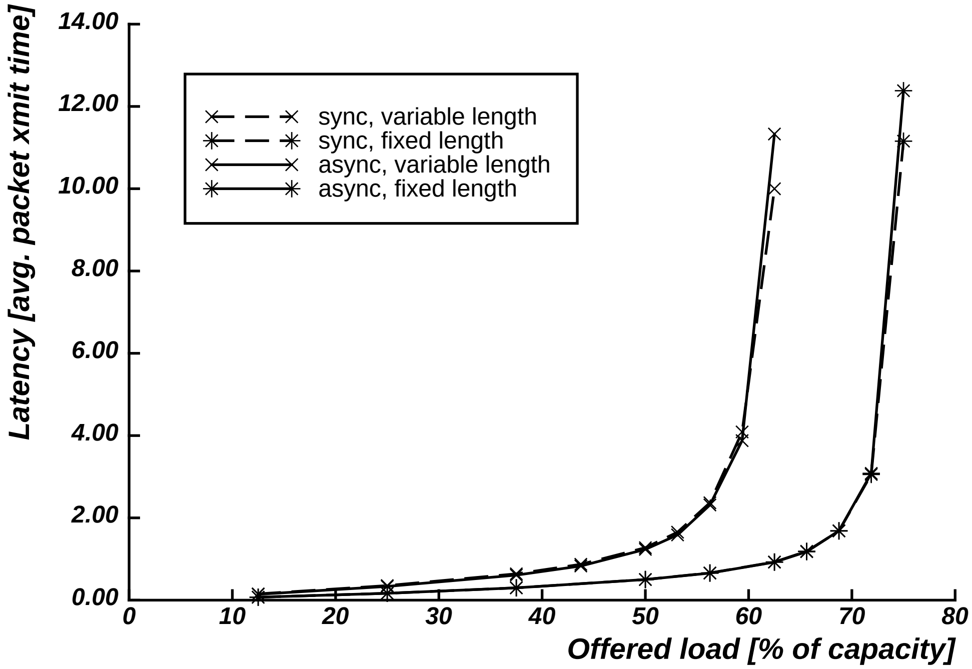


Figure 2-10: Latency vs. Load for register file based router

Figure 2-10 shows the latency for an 8 by 8 register file based router. It is interesting to note that synchronized routers fare a few percent worse at low loads while they outperform asynchronous operation at higher loads by a few percent. This observation holds regardless of router structure and packet sizes. However, this effect is too small to have any significance for actual implementations.

Of more importance is the superiority of fixed-length packet operation over variable sizes. Packet sizes were assumed to follow a negative exponential distribution with a mean equal to the fixed-length case. The latencies of an 8 by 8 crossbar under equivalent load conditions are given in Figure 2-11. While the asymptotic saturation bandwidth is the same as for the register file, the latency increase with load is much worse. A fixed-length, register file based router running at 70% load has a lower mean latency than a crossbar running at 20%.

2.3.4. Router Performance: Summary

The simplistic analysis of a single routing element in an artificial environment resulted in a number of qualitative observations that were later confirmed by detailed simulations of complete networks. The main results are:

- Register-file routers offer lower latencies than crossbar routers. They have comparable saturation throughput if the load on the crossbar is equally distributed across all inputs.

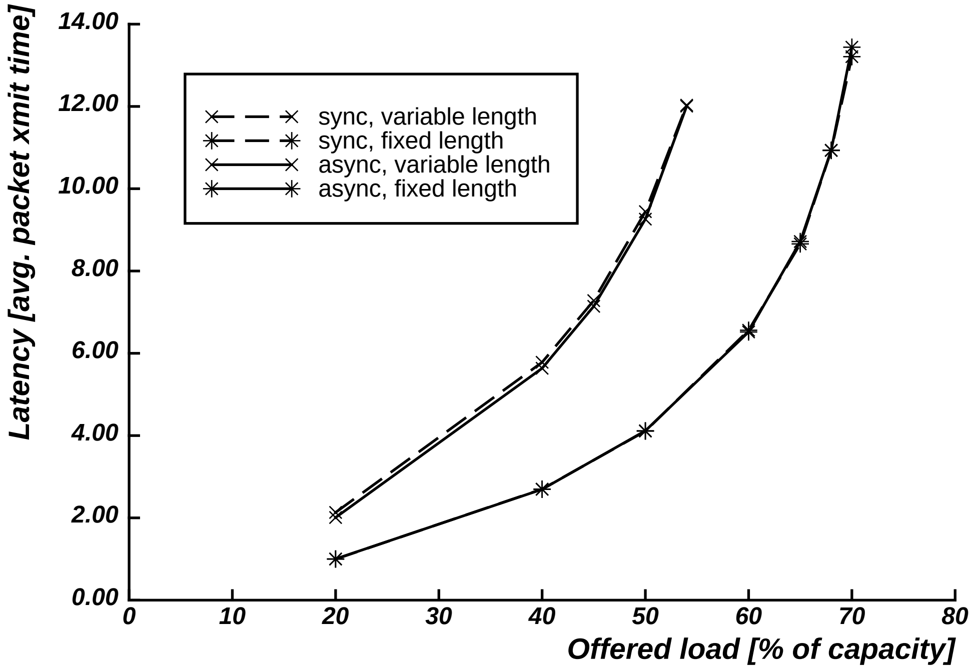


Figure 2-11: Latency vs. Load for crossbar based router

- Operating on fixed-sized packets considerably reduces latency and increases saturation throughput.
- There is no significant difference in either throughput or latency between synchronous and asynchronous operation.

It should be noted that these results do not assume any particular routing strategy or network topology.

2.4. Performance Measures

The performance of a communication network can be characterized along three broad categories: bandwidth, latency and feasibility. This section defines these measures and gives some analytical results that are derived from the topologies and a number of simplifying assumptions. Chapter 3 will address these issues in more concrete terms.

2.4.1. Bandwidth

Bandwidth is a measure of how much information can be moved through a channel within a given amount of time. This is a very simple measure if a single point to point connection is involved. However, even in this case, bandwidth could be defined in different ways depending on how the various protocol and encoding overheads are accounted for.

Definition 1: The *physical bandwidth* B_{phys} is the maximum transmission rate for a given channel and a given bound on the error rate. For example, electrical connections are characterized by the number of signal paths, their electrical fidelity, and the noise level. To fully utilize this potential, the transmitting and receiving circuits must be quite complex: a state of the art modem may squeeze 19200 Kbit/sec over a phone channel within a mere 3.5 Khz spectrum and 20 db signal to noise ratio in the presence of significant phase and amplitude aberrations. Circuits to drive wires within one cabinet are much simpler, faster but less efficient. Bit rates on short wire lengths rarely approach the upper transmission frequency.

Issues relating to the physical bandwidth will be addressed later. At this point a higher level measure of bandwidth is required:

Definition 2: The *usable bandwidth* B_{use} is the potential throughput presented to the network layer of the OSI-model for one channel. This is a fraction of the physical bandwidth due to modulation, encoding, synchronization and flowcontrol overheads.

The bandwidth of a system with more than two nodes and a multitude of channels is a complex function of the topology, the mode of operation and the traffic pattern. In order to characterize the impact of the topology on the throughput of the communication network, simplifying assumptions about the operation mode and the traffic pattern are made:

Definition 3: The *topological bandwidth* B_{top} is the asymptotic limit for saturated operation with packet sizes approaching 0, assuming uniform address distribution and perfect routers in each node. This is essentially a flow analysis in a directed graph. Each node injects B_{top} bytes per second into the network such that each of the N nodes will receive B_{top}/N bytes per second. Each node will send and receive the same amount of data. Data flows only along the shortest path between the sending and receiving node. If there are multiple shortest paths, traffic is distributed equally. The topological bandwidth is defined as the maximum B_{top} such that the superimposed traffic by all nodes will not exceed the usable bandwidth for any given channel in the network.

Topological bandwidth can be normalized by setting the usable bandwidth for each channel to 1. Physical implementations are frequently limited by the total usable bandwidth per node[10]. It is therefore interesting to normalize the usable bandwidth for each channel to $1/d$, where d is the number of channels emanating for one node.

2.4.1.1. Topological Bandwidth for k-ary Hypercubes

The analysis of k-ary hypercubes is aided by their symmetry. Averaging the load of all nodes will give the same value for any channel in the system. Therefore it is possible to compute the total usable bandwidth of the system and distribute it uniformly across all nodes. For each node, the amount of bandwidth needed to send packets of size 1 to each node in the system is derived from the topology. Dividing these two numbers yields the topological bandwidth per node.

---

[10]These limits arise from packaging and routing constraints and will be discussed in section 2.4.3.

Let k be the arity and d be the number of dimensions of a cube with $N = k^d$ nodes. To send one packet through the network, it has to traverse a certain number of channels, each time consuming one unit of bandwidth. This distance distribution depends on the assumption made about the traffic pattern. Based on the uniform traffic assumption, k-ary hypercubes have a multinomial distance distribution:

$$H(k,d) = \frac{1}{k^d} \sum_{\substack{n_1, \ldots, n_k \\ n_1 + \cdots + n_k = d}} \left(\frac{d!}{n_1! \cdots n_k!} \sum_{i=0}^{d-1} i\, n_i \right) = \frac{k-1}{2} d \tag{2.2}$$

$H(k,d)$ is the average number of hops (i.e. number of channels traversed) for one packet. For example if $k=2$ and $d=2$, there are 4 possible destinations: one node is 0-hops away (i.e. origin), two nodes are 1 hop away, and one node is 2 hops away. On average, one channel is busy for one time unit to transmit an unity packet, so $H(2,2)=1$.

If all channels in the system have unity usable bandwidth, the topological bandwidth is simply $d/H(k,d)$ because there are d channels per node. This value is plotted for various configurations in Figure 2-12. Not surprisingly, networks with a large number of channels ($=d$) have higher bandwidth. A more interesting observation is that hypercubes of any arity provide constant bandwidth per node regardless of network size.

For systems that have limits on the total I/O bandwidth per node, the topological bandwidth computation should assume channels of $1/d$ capacity. Examples for such a limitation are nodes that are bounded by the number of I/O wires: the message system could be packaged in a case with 32 pins available for channels to other nodes. This illustrative device may be able to transmit one bit per I/O pin every 50 ns. As such it could support 32 independent channels running at 20 Mbits/sec each or it could be configured for 4 byte-wide channels running at 20 Mbyte/sec each. In any event, the total I/O bandwidth per node would be a constant.

Figure 2-13 is normalized to a constant total I/O bandwidth. Naturally, network size independent bandwidth is no longer attainable: larger systems allow less I/O traffic per node. The performance discrepancy between constant arity and constant dimensionality networks is somewhat reduced, but there is still a significant advantage for higher dimensional networks.

2.4.1.2. Topological Bandwidth for k-Shuffles

The analysis of k-Shuffles is more difficult because uniform traffic is no longer evenly distributed across all channels. Furthermore, k-Shuffles are relatively asymmetric, which prevents simple recurrence relations. The net result is that there is no simple expression for the topological bandwidth of k-Shuffles. Instead, the results given here were obtained by

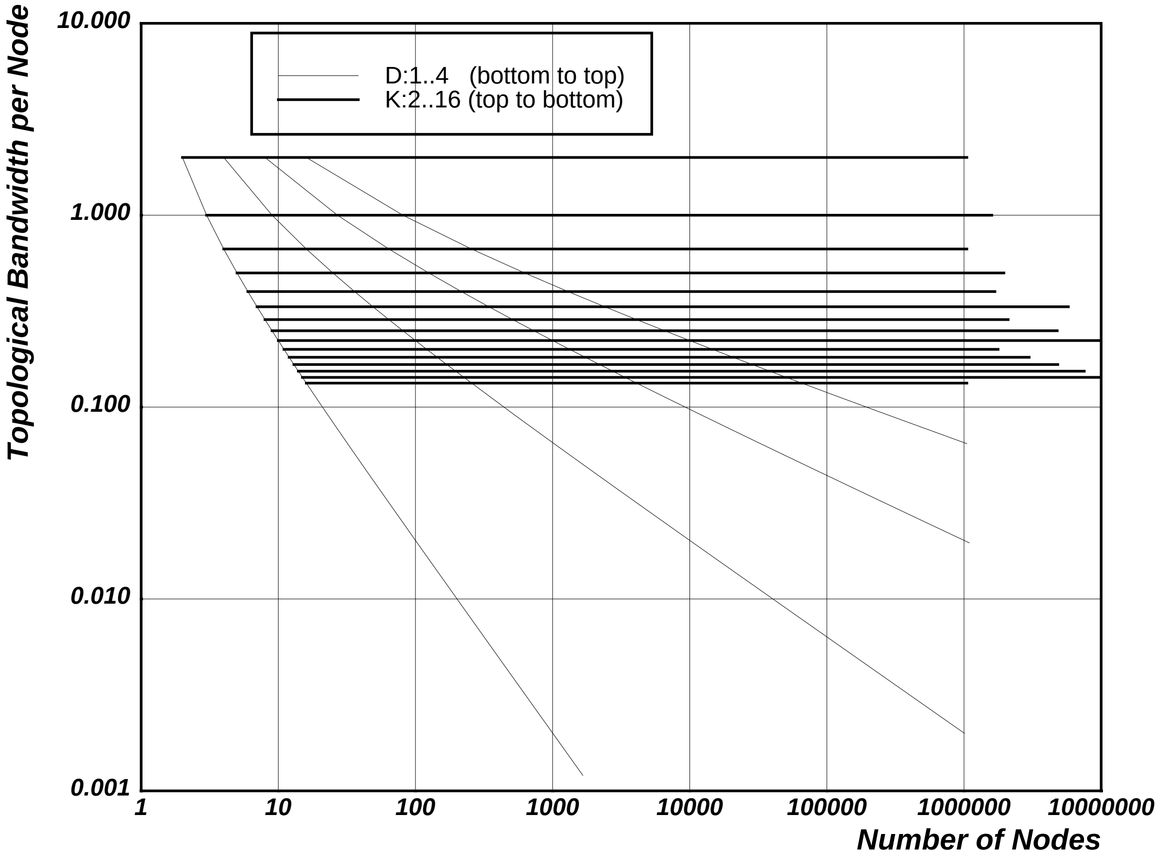


Figure 2-12: Topological Bandwidth for Hypercubes

constructing the actual network[11].

The uneven channel load distribution under uniform traffic wastes some bandwidth. For example, each diagonal node has a channel connecting one output to one of its own inputs. This channel does not contribute to the overall performance. A typical load distribution is shown in Figure 2-14. The solid line in Figure 2-14 is the result of convolving the exact histogram with a gaussian window.

Figure 2-15 is based on the unity channel capacity assumption. Given that the number of channels per node is k, it is hardly surprising that high-k networks fare better. The extreme is the $d = 1$ case, which is simply the fully connected graph.

Limiting the total I/O per node results in the data shown in Figure 2-16. As in the case of hypercube networks, high fan-out node networks offer higher bandwidth.

[11]A data structure corresponding to the desired configuration that has a counter corresponding to each channel is constructed. Each node is "*transmitting*" one message to all other nodes and the counter corresponding to each traversed channel is incremented. This approach is aided by the fact that k-Shuffles have exactly one shortest path for each pair of nodes.

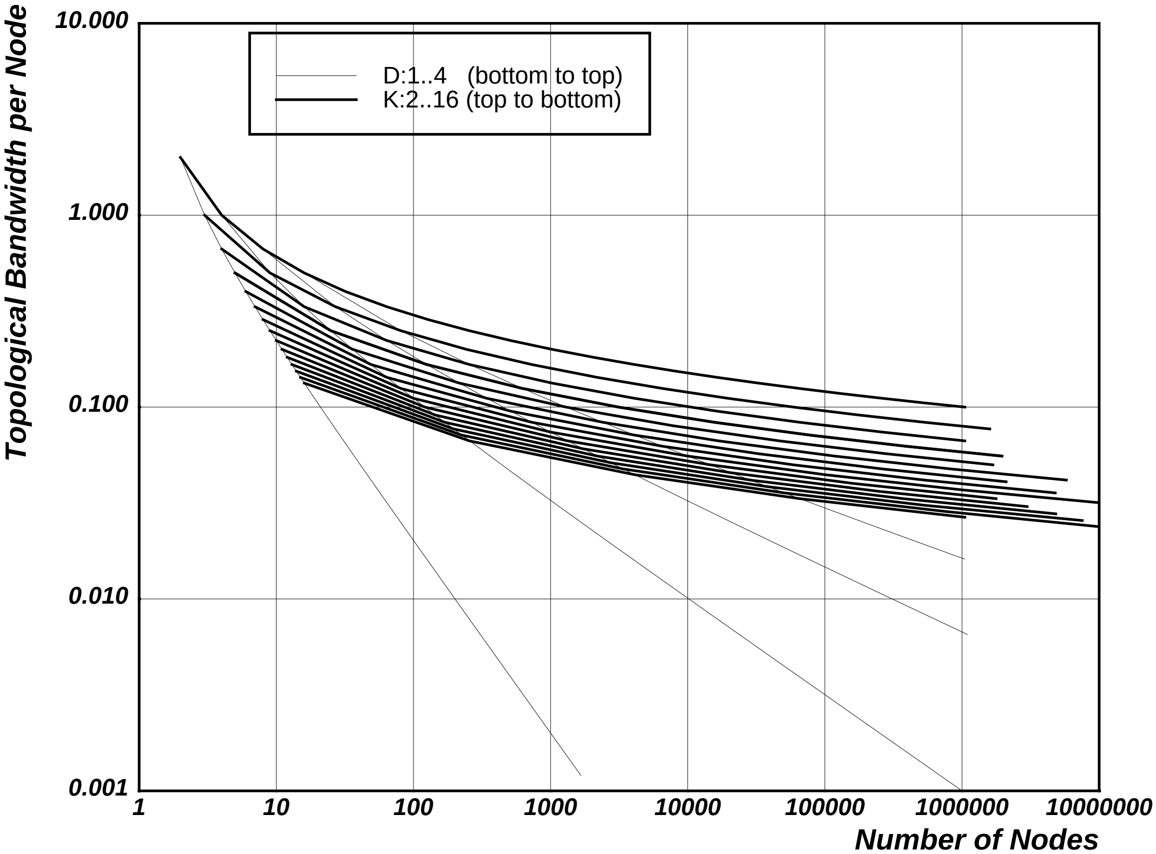


Figure 2-13: Normalized Topological Bandwidth for Hypercubes

2.4.1.3. Topological Bandwidth for Cube-Connected Cycles

The analysis of cube-connected cycles is similar to the one for k-ary hypercubes. The main difference is due to the composite nature of this topology: there are two types of channels, intra- and inter-cycle. Uniform traffic loads each channel type equally, but the load of inter-cycle channels differs from the load of intra-cycle channels.

The inter-cycle traffic generated by one node can be derived from Equation (2.2): $h(k, d) = k^d H(k, d)$ is the number of inter-cycle channels traversed by the packets of one node sending one packet to each cycle in the network. There are $N_c = k^d$ cycles in the network with d nodes each, for a total of $N = dN_c$ nodes. Therefore, a total of $N_c d^2 h(k, d)$ inter-cycle transmissions will take place if all nodes send one packet to all other nodes. Given the total number of inter-cycle channels ($= dN_c$) and the probability for a particular destination ($= 1/N$), the inter-cycle bandwidth is:

$$B_{top,\,inter}(k, d) = \frac{N_c d}{\frac{1}{N} d^2 h(k, d) N_c} = \frac{1}{H(k, d)} \qquad (2.3)$$

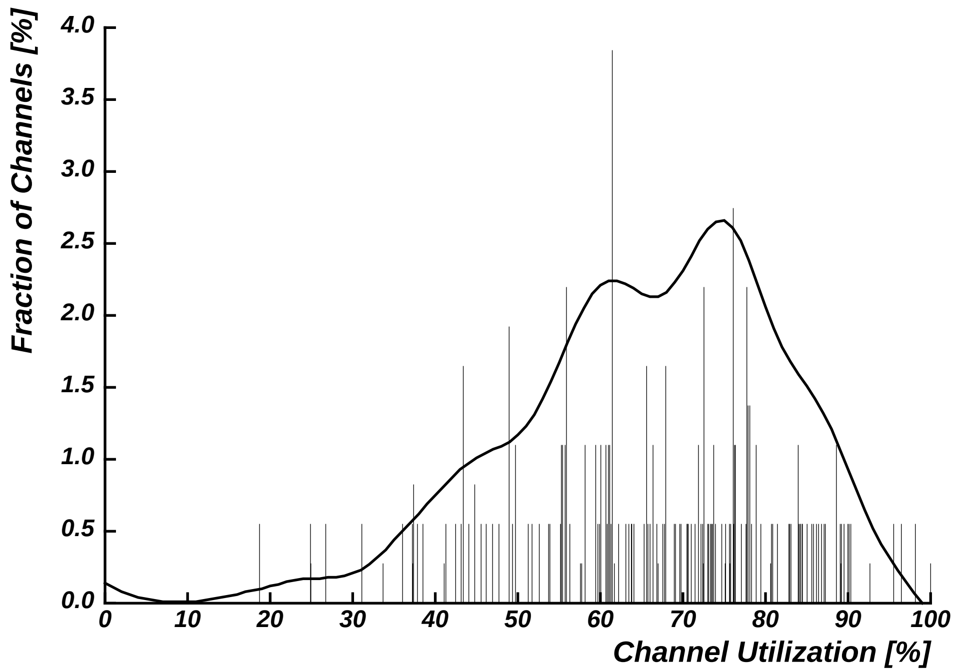


Figure 2-14: Channel Utilization for a 6-Dimensional 3-Shuffle

There are two components to the intra-cycle traffic: transient and local packets. Given optimal routing [117], one intra-cycle channel must be traversed for each transient packet that is changing dimensions in the cube structure. Again considering one node sending one packet to each other cycle, the number of intra-cycle transmissions due to transient traffic is:

$$t(k,d) = \sum_{i=2}^{d-1} (i-1)\binom{d}{i}(k-1)^i \tag{2.4}$$

Local traffic stays within one cycle. The number of intra-cycle transmissions for one node sending one packet to all other nodes within the same cycle is:

$$l(d) = \begin{cases} (d^2-1)/4 & \text{if } d \text{ is odd} \\ d^2/4 & \text{otherwise} \end{cases} \tag{2.5}$$

Inter-cycle traffic causes the equivalent of one local transmission. It depends on the routing algorithm whether these transmissions occur at the origin cycle, the destination cycle, or any intermediate cycles. Due to the network symmetry, any consistent policy will have the effect of distributing this traffic evenly. Given that there are $N_c 2d$ intra-cycle channels, the intra-cycle bandwidth is:

$$B_{top,\,intra}(k,d) = \frac{N_c 2d}{\frac{1}{N}(N_c d^2 t(k,d) + N^2 l(d))} = \frac{2k^d}{t(k,d) + k^d l(d)} \tag{2.6}$$

Figure 2-17 plots $min(B_{top,\,intra}, B_{top,\,inter})$ for various values of k and d. Low-arity networks $(k = 2, 3)$ become bounded by the intra-cycle bandwidth as the number of dimensions

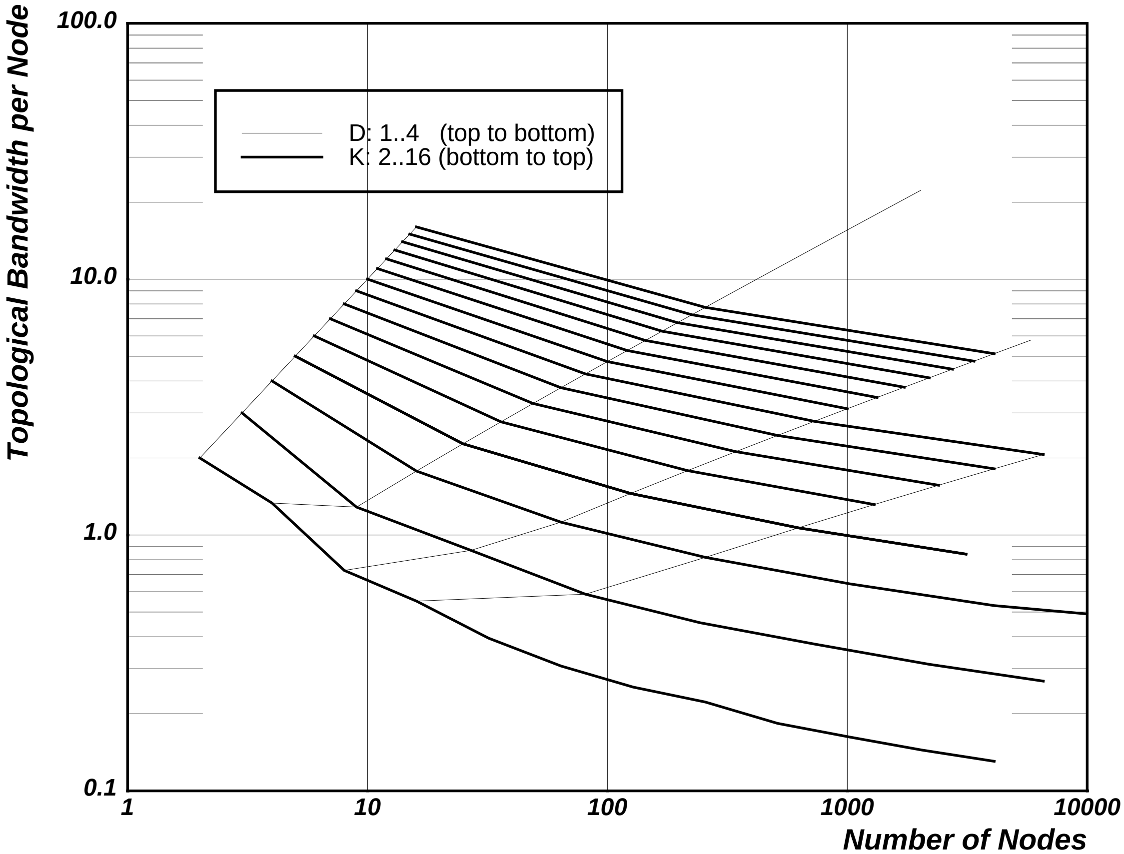


Figure 2-15: Topological Bandwidth for k-Shuffles

increases. This effect is shown by the dotted lines. High-arity networks are limited only by the inter-cycle bandwidth. It should be noted that cube-connected cycles have a constant number of channels per node (=3), so normalizing the total I/O bandwidth per node to 1 is simply scaling the bandwidth given in Figure 2-17 by $1/3^{rd}$.

2.4.1.4. Topological Bandwidth for Fat Trees

Fat trees have unity topological bandwidth by virtue of their construction because the degree of *fatness* in Equation (2.1) was set to achieve balanced channel utilization for uniform traffic.

Intermediate routing nodes require a higher, but bounded fan-out (larger networks need multiple intermediate routing nodes at one tree node). Assuming that intermediate nodes are simpler, they might be subject to less stringent I/O limitations as are the leaf nodes (i.e. location of processor and memory). Therefore, intermediate nodes may support unity usable bandwidth on all channels. Arguably, this qualifies fat tree as a constant fan-out topology.

However, channels and intermediate routing nodes do contribute to the overall system cost

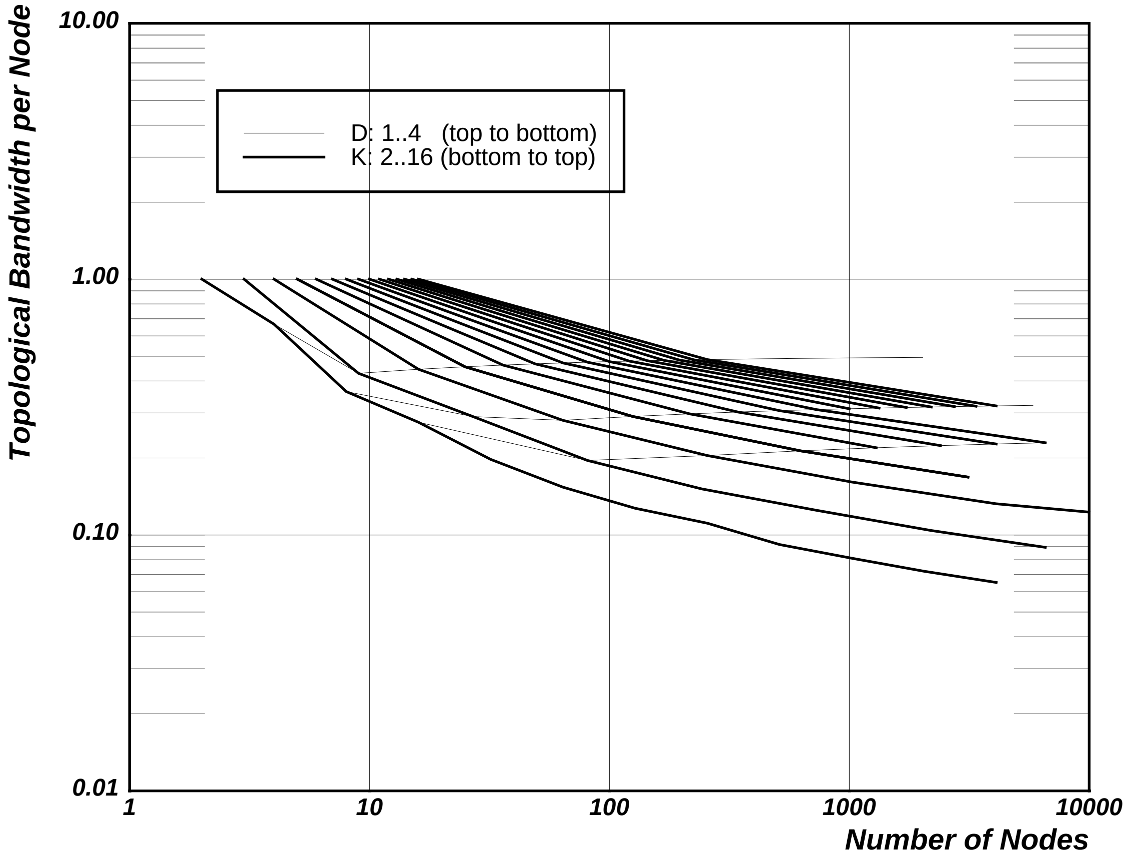


Figure 2-16: Normalized Topological Bandwidth for k-Shuffles

and complexity. In order to get a rough comparison with the other network topologies, channel bandwidth was normalized such that the total number of channels in the system divided by the number of leaf nodes is of unity bandwidth. This data is shown in Figure 2-18.

2.4.1.5. Topological Bandwidth for Stars

Centralized routers are probably not bounded by I/O constraints because there are only a few of them in a system. Therefore it could be argued that the router node can be designed large enough to accommodate unity usable bandwidth I/O channels. This results in a system with a *topological bandwidth* of $N/(N-1)$. This factor accounts for traffic that originates and ends on the same node.

However, the cost and complexity of a centralized router grows quadratically with the number of supported channels. The efficiency of a crossbar decreases with the number of channels as shown in the previous section. Further losses are due to the need of a centralized controller that sets up the crossbar. Most of the adaptive routing heuristics discussed in the next chapter don't scale nicely ($O(N^3)$) in the number of channels and are viable only for relatively small switches.

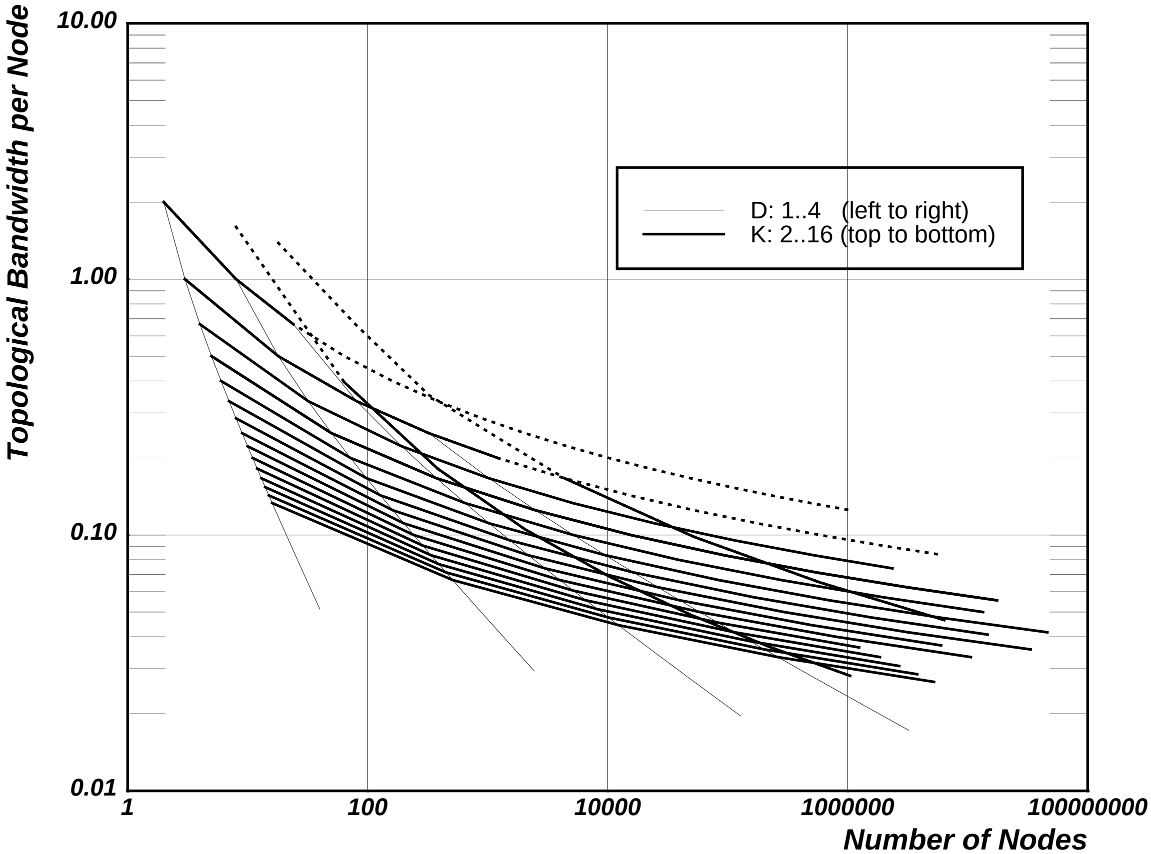


Figure 2-17: Topological Bandwidth for Cube-Connected Cycles

Unlike the case of fat tree, there does not appear to be a reasonable way to account for these costs and losses; hence the concept of a topological bandwidth is not really applicable in this case because the centralized router does not scale arbitrarily.

2.4.1.6. Topological Bandwidth for Random Graphs

Random graphs are analyzed to gain a reference point: the topological bandwidth of a network configuration can be compared to the expected performance of an arbitrary network. Similarly, other network figures of merit can be compared.

Random networks within the scope of this paragraph are networks that don't really exist. Instead, these networks are *abstract networks* with properties equal to the average over a large number of real, randomly connected networks.

A random network is characterized by the number of nodes ($=N$) and by the number of channels ($=d$) that originate from each node. It is assumed that the network is strongly connected so that each node is able to send a packet to each other node in the network. In a

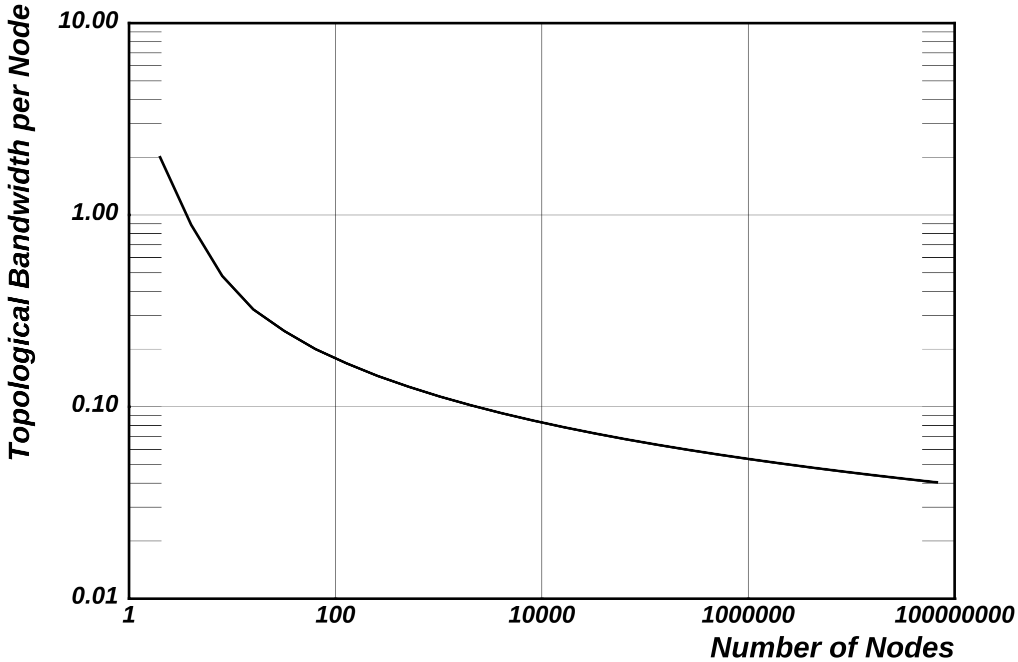


Figure 2-18: Normalized Topological Bandwidth for Fat Tree Networks

real network, nodes could be uniquely numbered and each node would be connected to a certain, known set of nodes so that each channel in the system has a defined origin and destination. The abstract random networks differ in that the destinations of the outbound channels are *not* known. The probability that a channel is connected to any particular node in the network is $1/N$. Instead of having a fixed destination, channels have a probabilistic destination distribution.

To analyze the performance of this "*network*", a node is selected as the start of a minimal broadcast tree. Packets are sent from the designated root node to all other nodes in the network. Each packet uses the minimal number of channel traversals to get to its destination. To construct this broadcast tree, consider Figure 2-19.

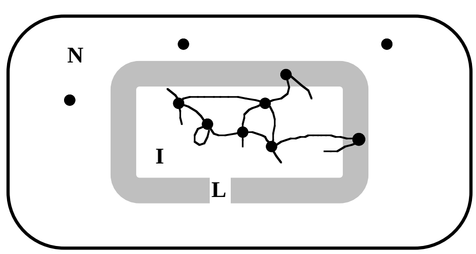


Figure 2-19: Random Network Construction

The broadcast tree is constructed, breadth first, in a number of extension steps. At step 0, the tree has only one node - the root node - which can send packets to itself in 0 channel traversals. At step 1, all nodes are added that are one hop away from the root node. This tree is iteratively expanded until it covers the entire set of nodes in the network ($=\mathbf{N}$, $|\mathbf{N}|=N$). The root node is located somewhere in the interior set of nodes $\mathbf{I}$. Each node in the set $\mathbf{E}=\mathbf{I}\cup\mathbf{L}$ has received its packet from the root node. Nodes in the leaf node set ($=\mathbf{L}$) received their packet from the root node in the last expansion cycle. Any node $\notin \mathbf{E}$ that will receive a packet after the next expansion cycle must be connected to a node $\in \mathbf{L}$. There are $D\cdot|\mathbf{L}|$ channels originating from the nodes in $\mathbf{L}$, each of which is connected to any node in $\mathbf{N}$ with equal probability. The probability P_e of expanding $\mathbf{E}$ by exactly i nodes depends on the total number of nodes N, the number of nodes already in $\mathbf{E}$, and the number of potential channels out of $\mathbf{E}$; $m=D\cdot|\mathbf{L}|$ is:

$$P_e(N,|\mathbf{E}|,m,i) = \frac{1}{N^m}\binom{N-|\mathbf{E}|}{i} f(|\mathbf{E}|,m,i) \tag{2.7}$$

where $f(\cdots)$ is defined recursively by:

$$f(|\mathbf{E}|,m,0) = |\mathbf{E}|^m$$

$$f(|\mathbf{E}|,m,i>0) = (|\mathbf{E}|+i)^m - \sum_{j=0}^{i-1}\binom{i}{j} f(|\mathbf{E}|,m,j)$$

or in closed form:

$$f(|\mathbf{E}|,m,i) = \sum_{j=0}^{i}\binom{i}{j}(-1)^{(i-j)}(|\mathbf{E}|+j)^m \tag{2.8}$$

The state of the broadcast tree after the n^{th} expansion step can be described as a two dimensional probability distribution: $S_n(i,l)$ is defined as the probability that $|\mathbf{I}|=i$ and $|\mathbf{L}|=l$ after n expansion steps. Obviously:

$$S_0(i,l) = \begin{cases} 1 & \text{if } i=0 \wedge l=1 \\ 0 & \text{otherwise} \end{cases}$$

Applying Equation (2.7) yields the probability distribution after a purely random expansion step, such that destinations of all channels leaving $\mathbf{E}$ are really uniformly distributed:

$$S'_{n+1}(i,l) = \sum_{e=j+k} S_n(j,k)\,P_e(N,e,kD,l) \tag{2.9}$$

If $S'_n(i,l)$ were used for the next iteration, the network would fall apart because there is no assurance that all nodes can be reached from the root node. In particular, some $S'_n(i<N,0)$ are greater than 0, so there is a finite probability that an incomplete broadcast tree has no leaf nodes with outbound channels to the rest of the network. In order to enforce connectivity, $S_n(i,l)$ is derived from $S'_n(i,l)$ by setting $S_n(i<N,0)=0$ with a renormalization so that $\sum_i S_n(i,l)=\sum_i S'_n(i,l)$:

$$S_n(i<N,0) = 0 \tag{2.10}$$

$$S_n(i<N,l>0) = \frac{S'_n(i,l)}{\sum_{k=0}^{N} S'_n(i,k)}$$

$$S_n(N, l) = S'_n(N, l)$$

The exception for $i = N$ to the normalization step in Equation (2.10) is due to the fact that the tree depth is limited to $N-1$. For practical purposes, it is also of interest that $S_n(i, l) = 0$ for $i + l > N$, so that it can be stored as a triangular matrix.

It should be noted that no assumption is made on the number of channels that lead into a node. In this sense, the networks considered here are a superset of the definition given in section 2.2.2.

At this point, a random network is described by a series of probability distributions $S_n(i, l)$, $n = 0, \ldots, N-1$ characterizing the evolution of one broadcast tree. Since nodes have no identity, this tree is representative for all N trees. Hence the symmetry argument can be used to compute the expected topological bandwidth. The expected number of leaf nodes $N_{lv}(n)$ at depth n is:

$$N_{lv}(n) = \sum_{i,l} l S_n(i, l) \tag{2.11}$$

This leads to the unnormalized topological bandwidth:

$$B_{top}(N, D) = \frac{ND}{\sum_{n=0}^{N-1} n N_{lv}(n)} \tag{2.12}$$

Figure 2-20 plots $B_{top}(N, D)$ for various network sizes and fan-out values.

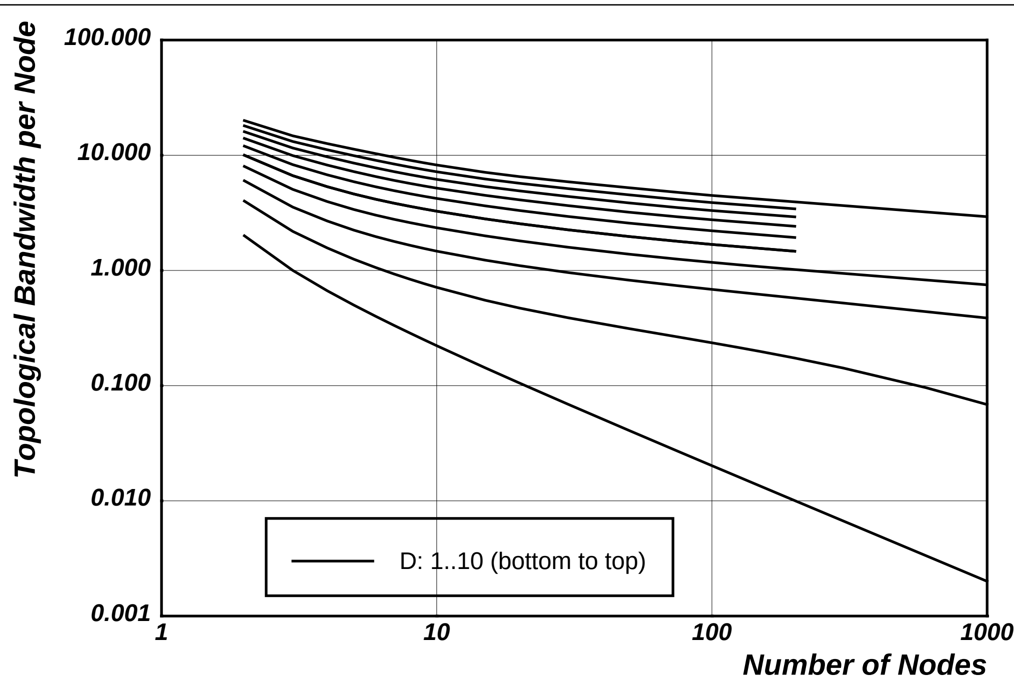


Figure 2-20: Topological Bandwidth for Random Networks

For I/O constrained networks the topological bandwidth is rescaled for $1/D$ capacity channels in Figure 2-21.

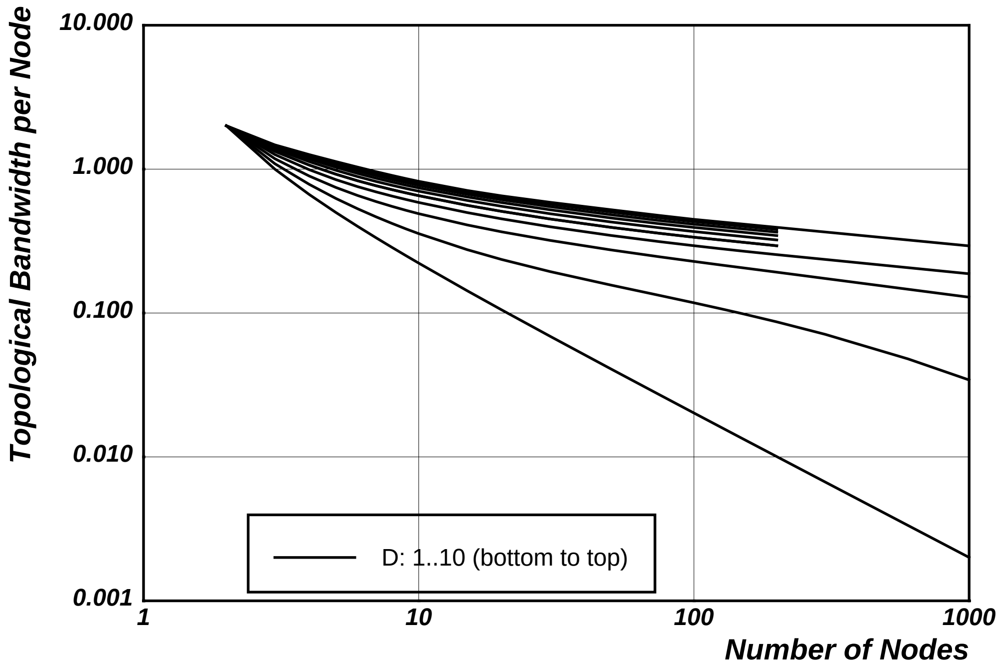


Figure 2-21: Normalized Topological Bandwidth for Random Networks

2.4.1.7. Summary

A common trend among all network families is that higher fan-out nets achieve higher bandwidth. However, bandwidth is not the only factor in network performance and other considerations favor lower fan-out networks, as shown in later sections.

Normalized Topological Bandwidth		
Network Family	100 Nodes	1000 Nodes
k-Shuffles	0.40	0.31
Random Networks	0.39	0.29
k-ary Hypercubes	0.30	0.20
Fat Trees	0.18	0.11
Cube-Connected Cycles	0.20	0.044

Table 2-3: Network Bandwidth Summary

Table 2-3 gives a rough overview. The normalized topological bandwidth is extrapolated to 100 and 1000 nodes. The 100 node network is allowed to have 7 outbound channels per node while the 1000 node nets are limited to a fan-out of 10. In each case the best topology for each network family was selected. It is surprising to see how well random networks score. k-Shuffles, despite their uneven channel utilization have a considerable lead over the other types. Finally, cube-connected cycles suffer from their low fan-out by this measure.

2.4.2. Latency

Many multiprocessor applications critically depend on communication speed (i.e. low latency). The sustainable communication volume (i.e. bandwidth) becomes less important in these cases because the total performance will be bound by the time needed to deliver a message. Long delays simply reduce the traffic volume, thus wasting bandwidth.

The total latency is of interest to the application level:

> **Definition 4:** The *total latency* T_{tot} is the time from the start of a communication operation in the originating node to the end of the communication operation in the destination node.

T_{tot} is typically the time from the start of the execution of a *send*-function until the successful return of the corresponding *receive*-instruction, which was issued by the receiving node at the most opportune moment. This common definition generally omits the time needed to prepare a message (for example to set up the destination address, message type and length, etc.) and the time needed to extract the actual information from the received message. Instead, definition 4 includes these hidden costs because they will be the subject of Chapter 4.

The total latency has two distinct components:

> **Definition 5:** The *network latency* T_{net} is the time required by the network to transport a message between nodes. This includes all delays caused by routing through intermediate nodes and by waiting for communication resources.

> **Definition 6:** The *communication system latency* T_{cs} is the time required by the communication system that interfaces the network to the local processing node. This includes delays caused by breaking messages into packets, by queuing received traffic for processor attention, etc.

This section discusses the nature of the network latency. T_{net} depends on factors such as routing policies, modes of operations, and message lengths that strongly interact and tend to dominate the contribution of the network topology. The network latency can be approximated by:

$$T_{net}(l, n_s, n_d) = g(l) + f(l)\,T_{sf}(n_s, n_d) \tag{2.13}$$

$g(l)$ and $f(l)$ are affine functions of the message length l while $T_{sf}(n_s, n_d)$ is the latency for a message of unit length sent from node n_s to node n_s in store-and-forward fashion. All systems, including those that use circuit switching, wormhole routing, or virtual cut-through, have a latency contribution proportional to $T_{sf}(n_s, n_d)$. For circuit-switched networks, this latency component is due to the need for the header to form the transmission path. The header (or at least the destination address field) must be received, verified (parity checked), and interpreted by each router along the transmission path. Hence the header will exhibit a latency comparable to a packet of appropriate size moved in store-and-forward fashion. The message length dependent part of the network latency is expressed in $g(l)$ (circuit switching, ...) and/or in $f(l)$ (store-and-forward, ...).

By using Equation (2.13) as an approach to analyze the network latency, the topology-dependent contribution is encapsulated in the store-and-forward latency T_{sf}. T_{sf} can be computed independently for various topologies and traffic assumptions.

2.4.2.1. Worst Case Store-And-Forward Latencies

The worst case store-and-forward latencies are simply the network diameters. It is assumed that an one length unit message can be transmitted over a channel in one time unit. This assumption neglects the bounded I/O capacity constraint: networks with d

Network Family	Parameter	Nodes	max Latency
k-ary Hypercubes	$k =$ arity, $d =$ dimensions	k^d	$(k-1)d$
k-Shuffles	$k =$ fan-out, $d =$ dimensions	k^d	d
Cube-Connected Cycles	$k =$ arity, $d =$ dimensions	$d\,k^d$	$max(0, dk-1+\lfloor d/2 \rfloor)$
Fat Trees	$d =$ depth	2^d	$max(0, 2d-1)$
Stars	$n =$ # of nodes	n	2
Random Networks	$n =$ # of nodes, $d =$ fan-out	n	not defined

Table 2-4: Worst Case Latencies

channels per node have channels of capacity $1/d$ and require d times as much to transmit the same amount of data. Consequently, the I/O boundedness leads to the same normalization that was used to compute the topological bandwidth. The normalized worst case latencies are the latencies given by Table 2-4, multiplied by the number of channels per node.

Latencies are analytically tractable only if an idle network is assumed. Any network traffic will cause some contention for network resources that results in higher latencies. Since the objective of this section is to outline general design principles, this assumption is justifiable. Later sections will show simulation results based on realistic assumptions.

2.4.2.2. Average Store-And-Forward Latencies

Average network latencies are based on the uniform traffic assumption. For k-ary hypercubes, this latency is given by Equation (2.2). Without normalization - as plotted in Figure 2-22 - the latency favors low-ary networks. Naturally, this is overly optimistic. When the total channel capacity per node is held constant, the low-ary cube no longer scores best. This is a typical case in minimizing latency conflicts wwhile maximizing bandwidth.

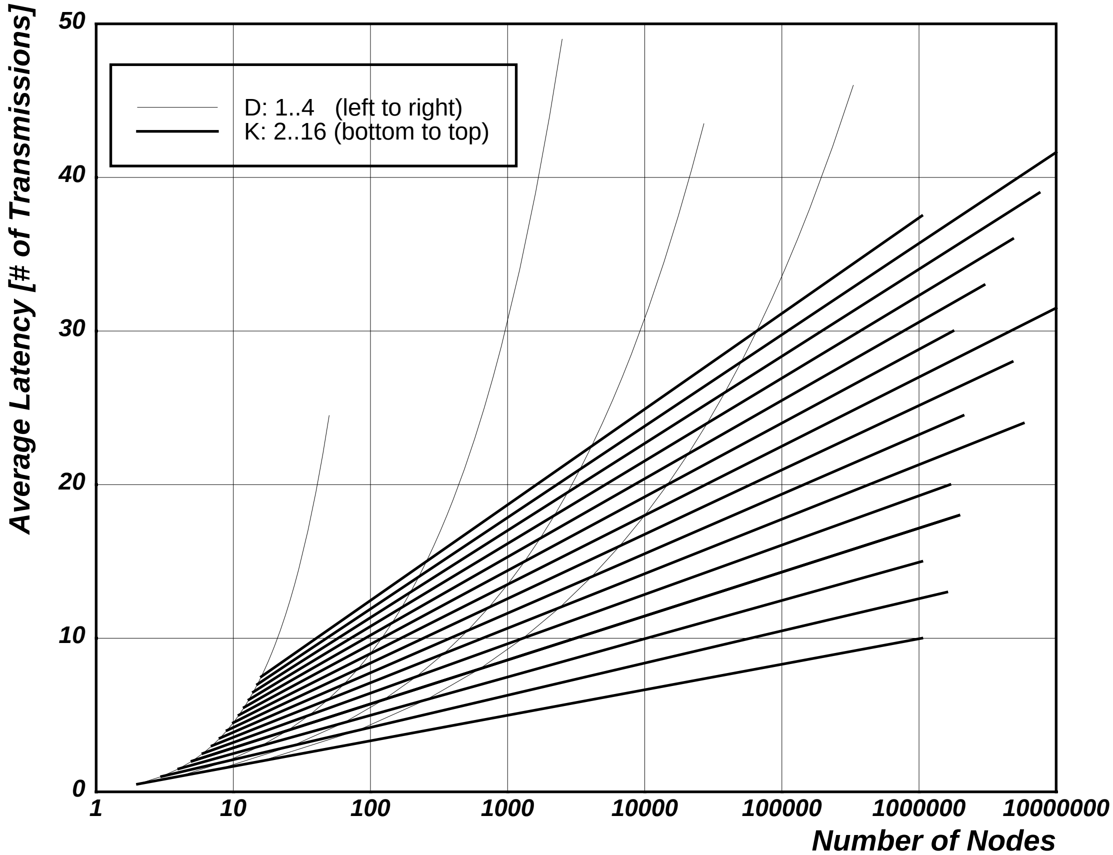


Figure 2-22: Average Latency for k-ary Hypercubes

Figure 2-23 shows that binary hypercubes have about 33% higher latencies than 4-ary cubes. Using Equation (2.2), the normalized network latency is:

$$T_{sf}(k,d) = \frac{k-1}{2}d^2 = \frac{k-1}{2\ln^2 k}\ln^2 N \qquad (2.14)$$

When expressed in terms of network size and arity, Equation (2.14) has a shallow minimum for $k=5$. While the average latency changes only a few percent for $k=4 \cdots 7$, 5-ary hypercubes have about 35% lower average latency than binary hypercubes for the same network size.

So far the discussion has been based on the implicit assumption that the underlying unity length message is one atomic entity. While this is true if T_{sp} is used for one packet or for the header of circuit switched systems, it is not true if the underlying system breaks up messages into independent packets. This approach can be used to avoid the conflict between bandwidth and latency optimization. If a message can be routed across several channels in parallel, latency can be reduced even for low-ary networks.

Figure 2-24 shows the normalized latency of non-atomic unity-messages that may employ several paths through the network in parallel. Messages sent to a distant node have more

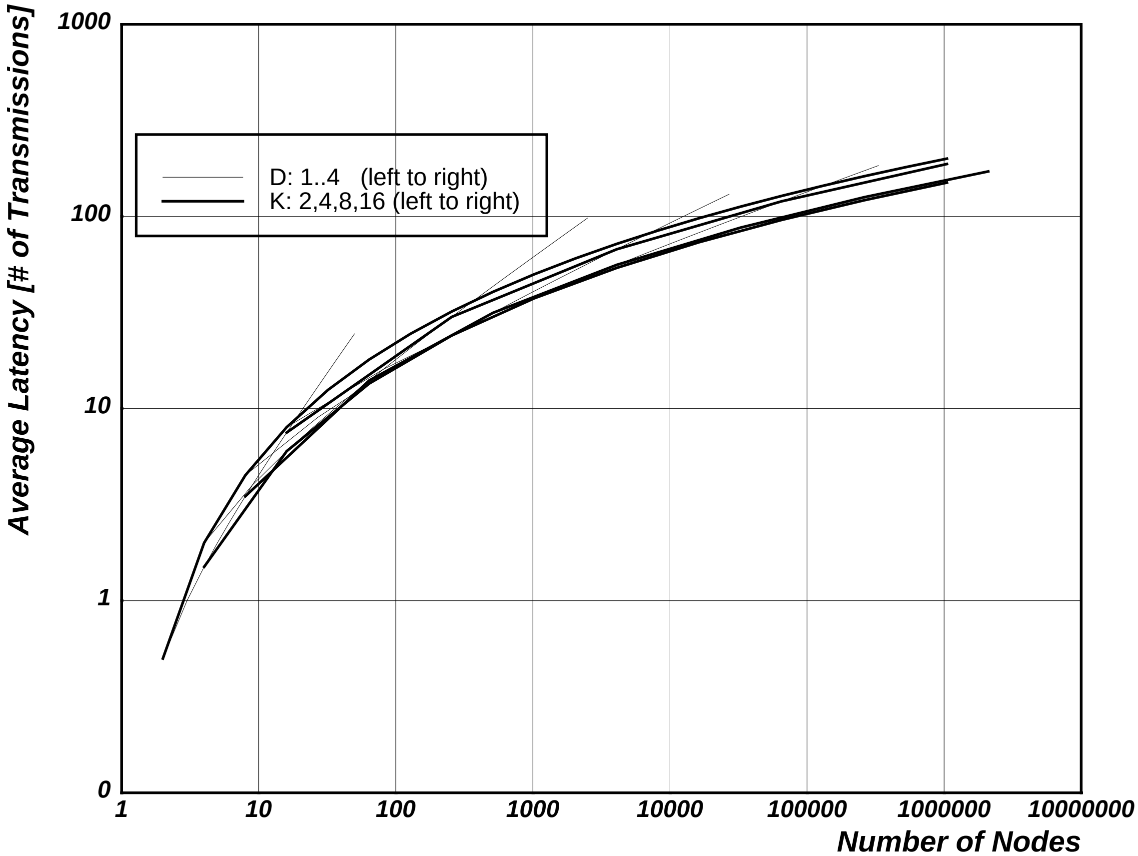


Figure 2-23: Normalized Average Latency for k-ary Hypercubes

parallel channels available than messages sent to adjacent nodes. For k-ary hypercubes the number of independent paths is equal to the number of dimensions in which source and destination nodes differ. Messages to more distant nodes can use more independent routes. This has an equalizing side-effect: the variance of the latency distribution is reduced.

k-Shuffles exhibit a more severe conflict between latency and bandwidth. While the un-normalized latencies (Figure 2-25) are in line with the bandwidth optimization, normalization changes the situation dramatically. Low fan-out k-Shuffles show significant advantages. When the network size increases beyond 1000 nodes, 3-shuffles are slightly faster than 2-shuffles.

Since the shortest path in a k-shuffle is unambiguous, there are no alternate paths that could be used to reduce latency by parallel transmissions.

Cube-Connected Cycle networks have properties quite similar to k-ary hypercubes. Since they have a constant fan-out topology, no normalization is necessary. As the number of dimensions increases, the cycle length becomes more significant. This results in lower

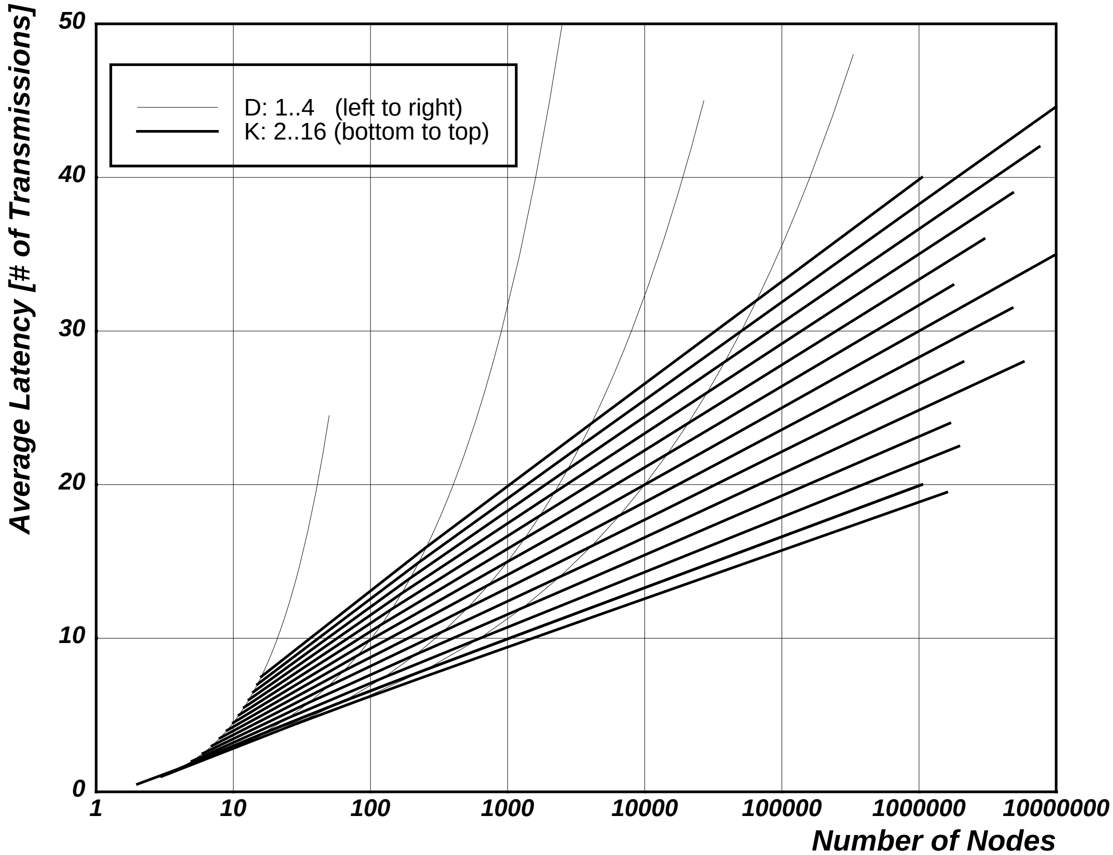


Figure 2-24: Normalized Average Latency for k-ary Hypercubes (with parallel channel utilization)

latencies for higher-ary networks. Beginning at about 1000 nodes, 3-ary Cube-Connected Cycle networks have lowest latencies.

Fat Trees exhibit logarithmic latencies. However, this is not comparable to the other networks because there is no good way to account for the extra cost of the internal routing nodes.

Random networks - as a class - do not have a defined worst case latency. A random network with n nodes may require up to $n-1$ transmission for a worst case packet. However, the probability that a particular implementation of a random network has such a worst case is quite small for higher fan-out networks. If the fan-out is one, the probability for that worst case is one. Therefore, the worst case network latency for random networks is a probability function that depends on the number of nodes and the fan-out. Figure 2-29 plots the worst case latencies for 100-node random networks with varying fan-out. The fan-out ranges from 2 (rightmost graph) to 10 (leftmost graph).

Average latencies are defined as the average over all possible random networks. Without

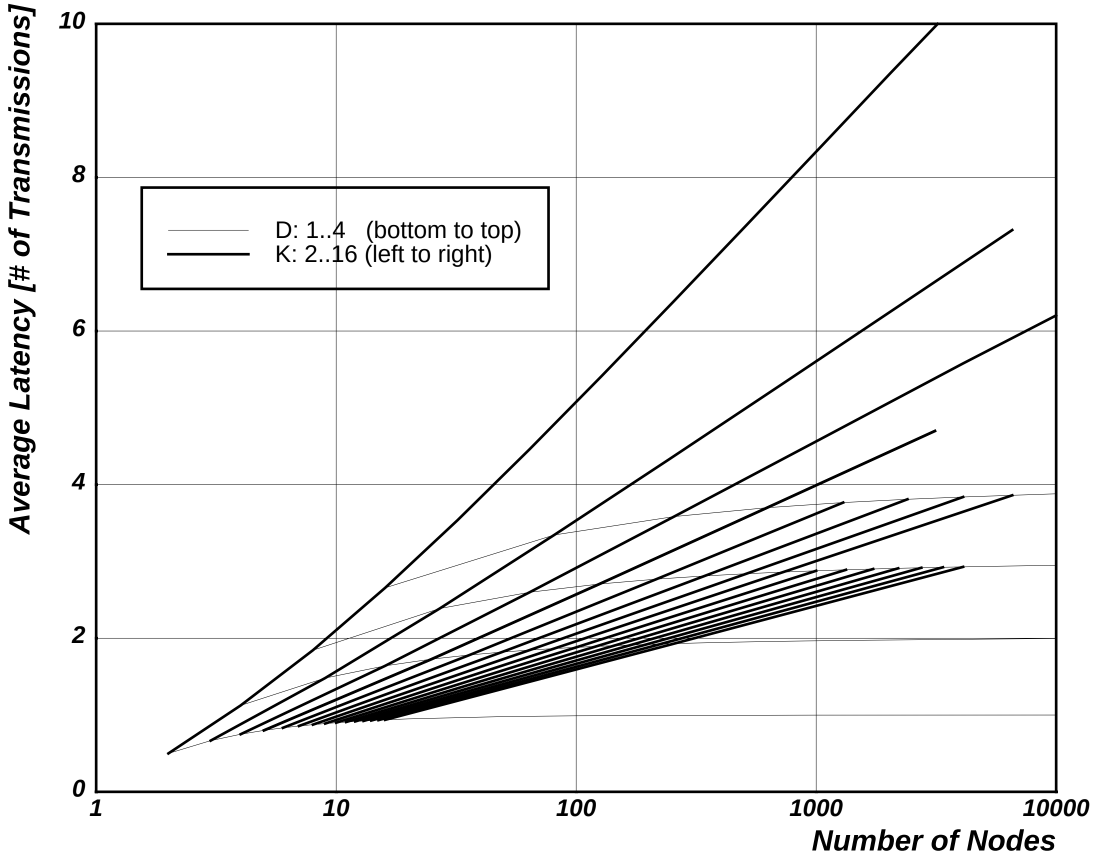


Figure 2-25: Average Latencies for k-Shuffles

normalization, the relation is similar to the inverse bandwidth data. However, once normalized, the optimal fan-out to minimize latency becomes a function of the network size (Figure 2-31). Up to about 20 nodes, rings perform better than more complex networks. From 20 to about 200 nodes, networks with 3 channels are best. Larger nets should use a fan-out of 4 or more. The optimal fan-out is a very slowly increasing function of the network size.

2.4.2.3. Summary for Store-And-Forward Latencies

Lower fan-out network topologies generally have smaller latencies unless parallel transmissions are feasible. As the network size increases, the optimal fan-out increases slowly. This implies that the optimal network selection will have to balance between two conflicting design objectives, bandwidth and latency.

Table 2-5 lists the normalized average latencies extrapolated to a 100 and 1000 node network. In each case the network parameters were selected to achieve the lowest latencies within a given network family.

[12]The cost for internal routing nodes and channels is not included.

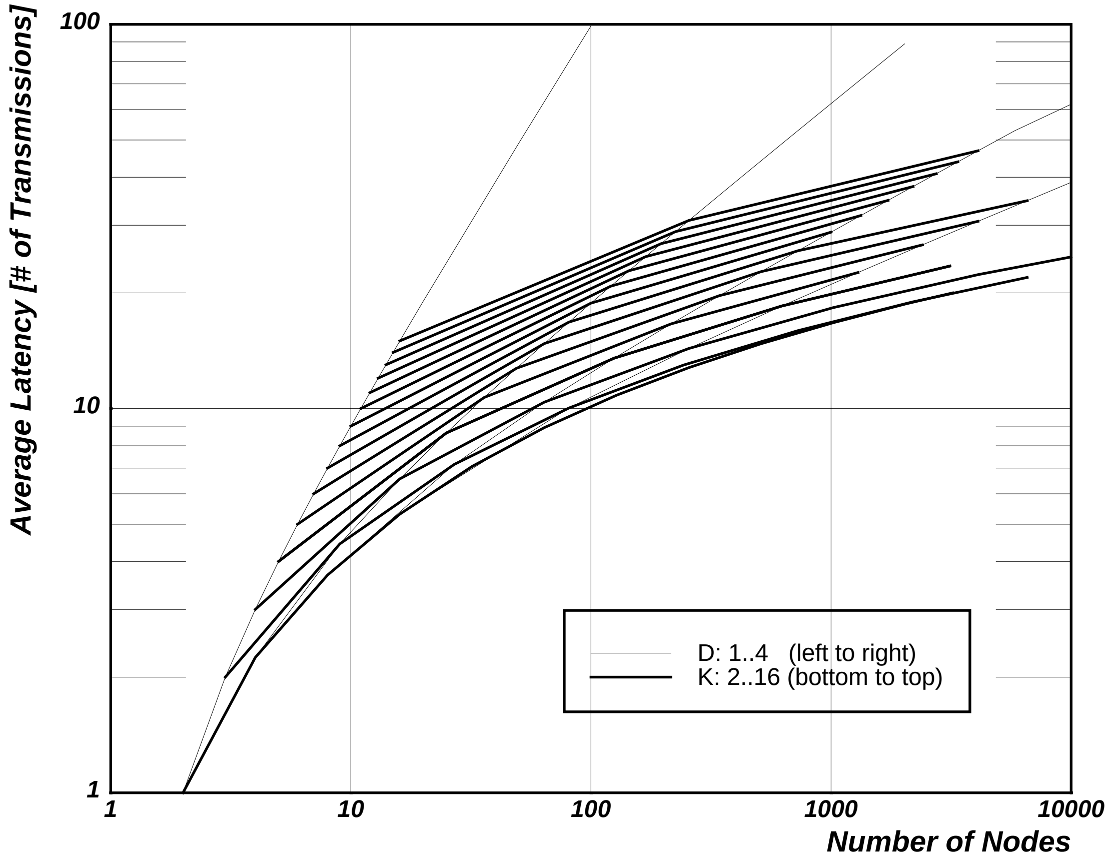


Figure 2-26: Normalized Average Latencies for k-Shuffles

2.4.3. Feasibility

Various measures of feasibility have been proposed to estimate the cost and complexity of implementing a particular network. Among these are:

- The number of input and output signals per node.
- The bisection width, defined as the number of wires crossing an imaginary boarder dividing the network in two parts of equal size.
- The wire length of the communication channels.
- The existence of a planar embedding of the network topology.

Each measure tries to estimate the implementation cost of a network as some function of the network size and structure. Unfortunately, the answer varies greatly depending on the available technology and the desired system speed.

The number of I/O wires per node is limited because of contemporary packaging technology. The integration density is steadily increasing, but the interconnection technology is not keeping up with that pace. Connecting several integrated circuits may use multichip ceramic

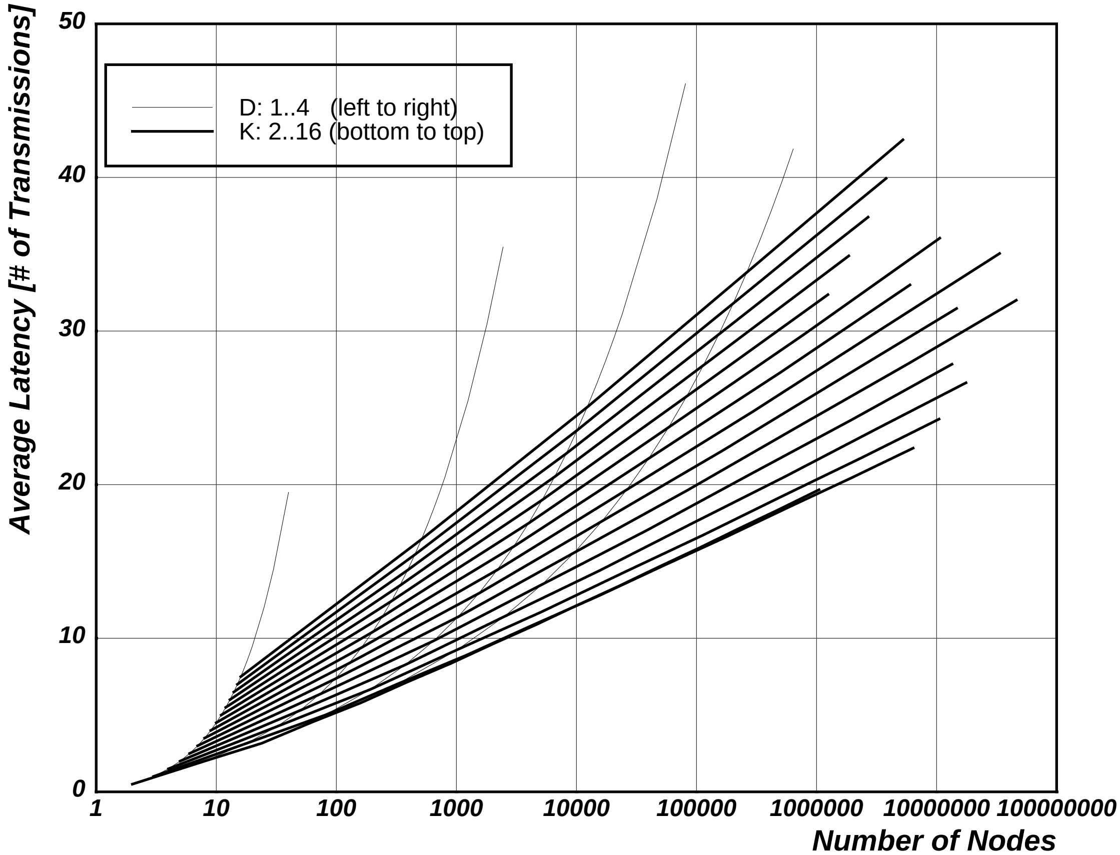


Figure 2-27: Average Latency for Cube-Connected Cycles

carriers, conventional packaging plus PC-boards or other high density packaging technologies. In each case, rather large - compared to the transistors and wires on chip - bonding pads are necessary to ensure mechanical alignment, reliable contact, and good heat transfer. These large features translate to considerable area demands and represent a significant electrical load, which requires much larger drivers than on-chip wires. The net result is that the number of I/O signals is typically limited to about 200 signals per chip. This I/O constraint was accounted for by normalizing the bandwidth and latency by the number of I/O pins.

The bisection width gives a rough idea of the wirability of a system. The reasoning is that networks with high bisection widths require a lot of wires between the two imagined halves. Hence they are difficult and costly to implement. On the low-end of the scale are high-arity, low dimensional hypercubes. For an even-ary hypercube of N nodes, the bisection width is simply N/k. Hence if k is large, the bisection width is small. This measure favors rings or toroids. Fat Trees have a significantly higher bisection width of $\frac{N^2}{4(N-1)}$ but are just as easy to wire due to their recursive nature.

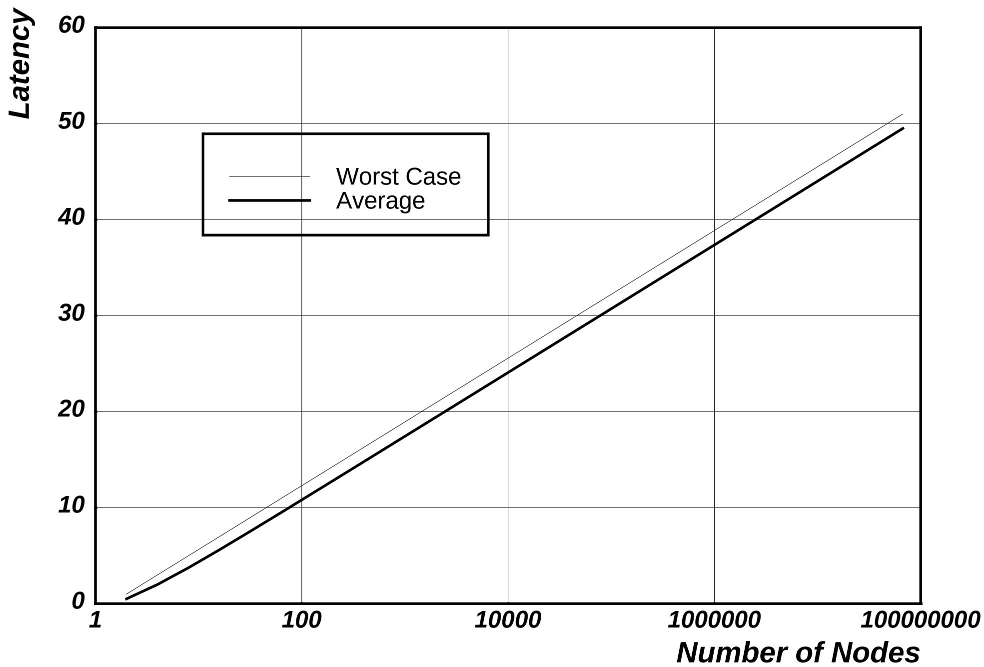


Figure 2-28: Latency for Fat Tree Networks

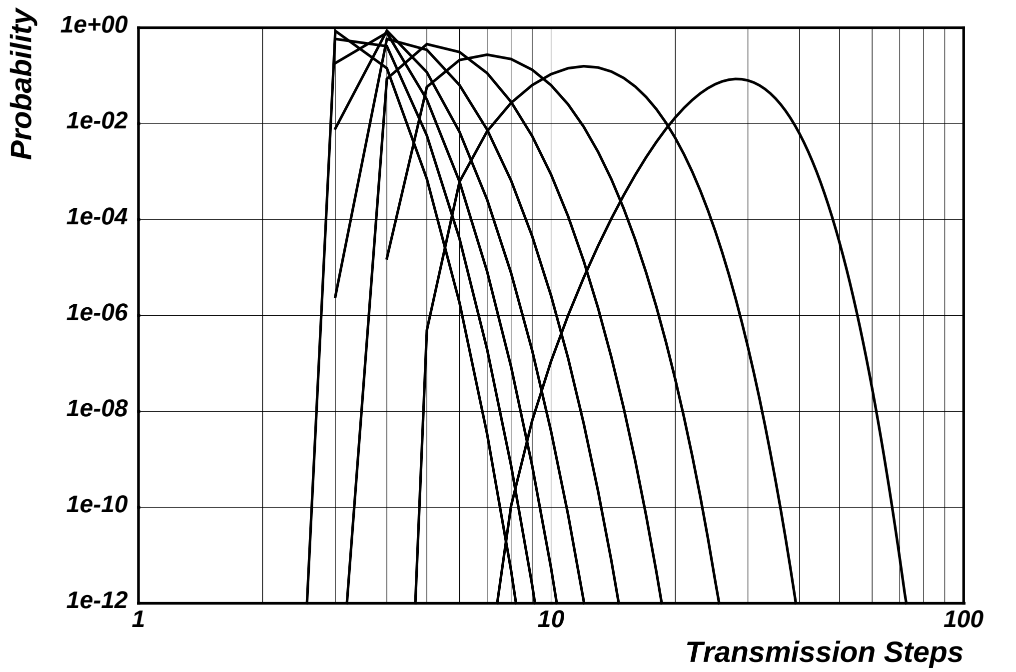


Figure 2-29: Worst Case Latency Distributions for 100 Node Random Networks

The physical wire length dictates the minimum delay due to the finite propagation delay.

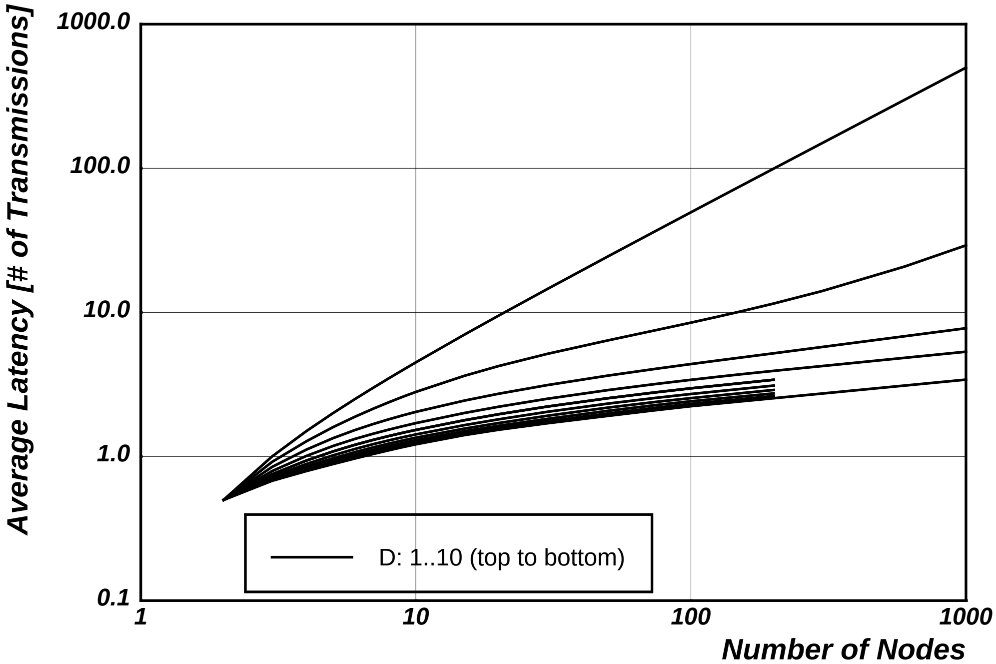


Figure 2-30: Average Latency for Random Networks

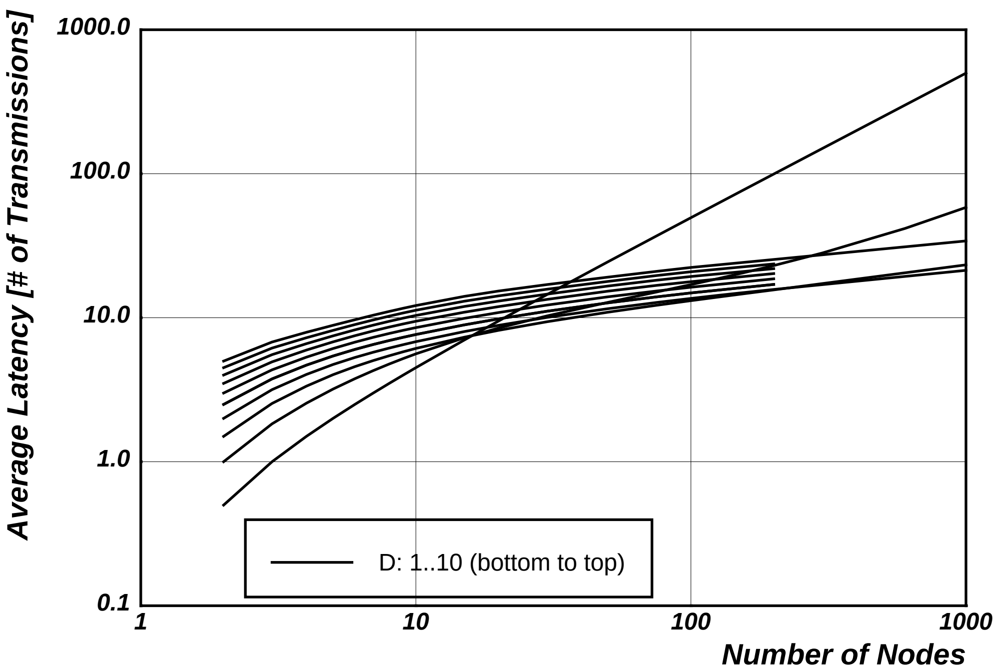


Figure 2-31: Normalized Average Latency for Random Networks

One meter of transmission line requires about 5nsec. All high performance systems will run their channels as transmission lines; hence the delay is proportional to the wire length. This

Normalized Average Latencies		
Network Family	100 Nodes	1000 Nodes
k-Shuffles	10.1	16.6
Fat Trees[12]	10.7	17.4
Random Networks	13.1	21.3
Cube-Connected Cycles	15.4	25.6
k-ary Hypercubes	16.1	36.4

Table 2-5: Network Latency Summary

establishes a relation between the system size, the size of each node, and minimal, wire induced delays. While a theoretical analysis of this relation is quite challenging, it is also of little practical importance. Good or bad engineering and the implementation technology can amount to huge changes in the constant factors. For example, the TF-1 [38] project demonstrated the feasibility (not necessarily affordability) of a high speed 32K node interconnection network with wire length of several meters. Part of this design uses the fact that transmission lines are storage elements and that they can be integrated into the system as just another pipeline stage.

The existence of a planar embedding of the network topology is helpful but not necessary. For example, the Cray-3 interconnection technology uses all 3 dimensions and achieves almost the same wire density in the third dimension [33]. Multilevel PC-boards, ceramic sub-assemblies, cables, and other PC-board interconnection technology are not constrained to planar topologies.

The bottom line is that the feasibility of a network is a complex question that will be decided by the actual implementation technology. Some networks, such as k-Shuffles will be harder-to-implement. However, even hard to implement networks like binary hypercubes were successfully realized in commercial machines [89, 30].

2.5. Implementation Issues

This section on implementation issues is primarily concerned with some of the low level considerations, such as the electrical and mechanical problems of communication networks. Only rather general concepts are reviewed because the concrete design presented in the next chapter will address the details.

2.5.1. Synchronous vs. Asynchronous Operation

One of the most fundamental design decisions is the selection of a timing discipline. Established disciplines include:

- Synchronously clocked circuits.
- Asynchronously clocked circuits.
- Self-timed circuits.

Synchronous designs are probably the most common. Most contemporary designs are synchronous, as are all IBM mainframes, Cray high performance computers, etc. Signals are aligned to a global system clock. The appeal of this method is its simplicity: the entire system state is defined at certain periodic points in time and the next system state is merely a boolean function of the current system state. The system operation is a sequence of frozen states, which provides a simple model for simulators, logic synthesis tools, and other CAD tools. The actual timing is quite decoupled from the functionality and timing verifiers can be used to ensure that the logic implementation meets the deadline defined by successive clock transitions. The downside of this approach is that all functions are constrained by the same clock period, which must be wide enough to accommodate the slowest component. Furthermore, the cycle must be conservative enough to include worst case device delays under all possible conditions. Hence margins must be included for temperature and supply voltage variations, device aging, etc. In large systems, the precise distribution of the clock signal itself requires attention.

Asynchronously-clocked systems partition the logic into regions that operate on a local clock. The local clock avoids the clock distribution problem and can accommodate different logic families that run at different speeds. Unfortunately, signals that cross regional boundaries require synchronization, a provably intractable problem [24]. Synchronizers can be built with very low failure probabilities so that reasonably reliable operation of the entire system is possible. Nevertheless, synchronization requires time.

Driven by the desire to run all parts of a system at the highest possible speed and to avoid synchronization problems altogether, self timed logic disciplines were developed. True self-timed logic adds a protocol to all logic signals that tells subsequent circuitry when a signal is valid. This leads to very elegant composition rules that can avoid all timing-related errors. Furthermore, no clock distribution is necessary because there is no clock. On the down side, no global system state is defined and debugging a system becomes more difficult. Also, self-timed circuitry is more complex so that part of the potential speed advantage is lost by higher complexity which leads to less circuit density.

2.5.2. Directional vs. Bidirectional Channels

Directional channels are easier to implement, especially with high speed outputs driving transmission lines. In this case, it is sufficient to terminate the wires at one end, usually at the receiving site. The complication introduced by bidirectional operations include:

- The need for output drivers and input receivers at both ends.
- The requirement for termination at both ends and therefore higher power consumption. Alternatively, special circuitry can be used that acts as a termination according to the prevalent data-flow (for example switchable or active termination).
- The need for coordination of the output drivers. Output driver conflicts can be destructive if they persist for extended periods. Transient output conflict can produce large amounts of electrical interference due the the associated current spikes.
- The requirement for buffer zones between operations in opposing directions. Especially in cases with a significant transmission delay, several bits may be in transit on the same wire. This is similar to a pipeline that needs to be drained before is used in the reverse direction. Frequent direction changes will cause a net loss of bandwidth in these cases.

The factors become more severe as the speed increases. The fastest logic families (like 100K ECL) offer few bidirectional devices and bidirectional operation is generally avoided.

There are significant advantages to bidirectional operation in communication networks:

- The network diameter can be reduced in some topologies.
- The network resources can adapt better to changing traffic patterns.
- The wire and I/O pin utilization may increase.

The network diameter of a directional ring with k nodes is $k-1$. If the same ring uses bidirectional channels, the network diameter is reduced to $\lfloor k/2 \rfloor$. Likewise, the average number of channels needed to communicate between two random nodes will decrease from $\frac{k-1}{2}$ to $\frac{k}{4}$ if k is even, or to $\frac{k^2-1}{4k}$ if k is odd. This diameter reduction applies directly to k-ary hypercubes and cube-connected cycles. The connections along each dimension form rings of length k in both cases. Thus high-ary networks of these types see a diameter reduction of 50% that translates directly into lower latencies and higher topological bandwidth. Similar improvements occur for high-dimensional k-Shuffles and low fan-out random networks. Essentially all high-diameter network topologies benefit significantly from bidirectional operation.

Low-diameter networks (like binary hypercubes) or networks with implied bidirectional nature (like fat trees and stars) do not see any diameter reduction[13].

[13] A network that would lose connectivity if channels were directional requires bidirectional links. If no bidirectional channels are feasible, the links must use two channels in opposing directions.

If the presence of traffic controls the direction of transmissions, a simple adaptive resource allocation occurs. Directions should alternate fairly while traffic is pending for both directions. If there is no traffic in one direction, the entire channel capacity could be devoted to the other direction. Unlike other adaptive routing mechanisms, the traffic sequence is not altered.

Some topologies require bidirectional links. For example, the binary hypercube has two node rings along each dimension. These are simply two channels connecting the same nodes in opposing direction. A bidirectional channel can replace both, reducing the number of I/O connections and wires by two. Naturally, channel capacity decreases likewise.

Chapter 3
A Message System Design

The message system design described in this chapter is intended for network-based multiprocessors that support a form of shared memory. This design goal dictates a number of fundamental design decisions. Anticipating a load of predominantly short messages, a store-and-forward approach with fixed-length, minimal sized packets is chosen. As shown in the previous chapter, this approach removes the conflict between bandwidth and latency optimization and favors the use of low diameter networks.

Another basic design decision is the use of strictly synchronous operation. It will be shown that the required clock distribution is well within reach of existing technology. A novel way of synchronizing the system clocks of multiple, distributed subsystems will be developed.

In addition to the fast, short haul communication channels commonly used in message-passing multiprocessors, a second type of channels that can integrate multiple, distributed processor clusters in a transparent fashion is described.

The core of the message system is an adaptive, topology-independent router. A step by step evolution of the routing heuristic is presented. A pragmatic approach to the deadlock problem is derived from the resource ordering method.

An outline for a VLSI implementation of the proposed message system demonstrates that the size and complexity are well within reach of current technology.

Finally, simulation results are presented. Of particular interest are results for high traffic loads. The message system was specifically designed for good performance near saturation.

3.1. Design Objectives and Rationale

Most message system designs strive to maximize bandwidth and minimize latency. The message system described in this chapter is no exception: high bandwidth and low latency are the primary design goals. However, these two conflicting goals are rather fuzzy guides and a number of secondary design objectives are needed:

Effective resource utilization:
The cost of the message system in terms of communication channels (IC pins, PC-board area, connectors, wires, etc.) and circuit complexity (gates, memory, chip area, etc.) is assumed to be externally constrained and the objective is to design an optimal system. This is unlike designing a system to meet given performance specifications.

Graceful saturation:
Communication is viewed as a system resource, no different than CPU speed, memory capacity, and I/O capabilities. In particular, communication is as likely to become a bottleneck as are the other system resources. Programs can be CPU bound, memory bound, I/O bound, or communication limited. Any active program is constrained in some of these ways. The implication is that saturating the message system is not an exceptional event. Some parallel programs will be bound by their communication requirement and the message system should perform well under such high load conditions. High loads should not cause any significant loss in bandwidth, just as fully-using the CPU should not slow it down. Naturally, it is desirable that the various resource bounds of a balanced system are roughly equal for typical programs. This philosophy differs from those that view the CPU as the only legitimate bound to performance.

Topology independence:
Given the lack of a universally optimal topology, the message system should be flexible enough to cope with different topologies. Topologies may be chosen to suit particular applications or to meet physical implementation constraints. Incremental upgrades or extensions are likely to result in odd topologies as are component failures.

Scalability: System configurations are likely to change over time. While it is hardly possible to scale arbitrarily *and* maintain efficiency at any size, a certain range of system sizes should be anticipated. System sizes of about 10 to 1000 nodes are the target here.

Local area spanning:
The message system is extensible beyond one cabinet and may be physically distributed over a machine room or a building. As a consequence, the design incorporates two types of channels with different capabilities: high speed, short distance channels to connect nodes within one cluster and medium speed, medium distance channels to connect multiple clusters. The message system integrates both channel types into a logically homogeneous network. Naturally, the difference in performance can't be hidden.

Symmetry: The lack of a centralized control is considered important. This might be a matter of intuition, aesthetics or religion.

Besides these design objectives, a model of the intended application is needed. This is summarized in a number of assumptions on how the message system is going to be used:

Short Messages: Typical traffic is assumed to be composed of predominantly small messages, where *'small'* is defined as being one word, one address, or one cache line.

Short lived traffic patterns:
Traffic patterns are likely to be highly dynamic, irregular, and volatile.

Both assumptions tend to increase the system complexity or tend to reduce performance: Longer messages enjoy a higher data to overhead ratio and static traffic patterns allow compile-time optimizations that can reduce network contention. Therefore, if these assumptions do not hold, no loss in performance is expected. In that case, the design would appear unnecessarily complex and/or expensive.

3.2. Basic Design Decisions

Based on the general design tradeoffs outlined in Chapter 2 and on the design objectives given above, three fundamental design decisions are made:

- Synchronous operation. All nodes within one cluster are driven from one clock source. The clock sources of different clusters are synchronized.
- Packet switching. Messages are broken into a number of packets that traverse the network independently.
- Minimal, fixed-length packets. All packets are of the same size and this size is chosen to be that of the smallest message.

These decisions are the basis of the design described in this chapter.

3.2.1. Synchronous Operation

Most digital systems currently in use operate synchronously from a clock oscillator. In particular, all commercial high end CPU's operate in this mode[14]. Since the message system is ultimately connected to the CPU, driving both with the same clock source can reduce synchronization delays. Using synchronous circuitry in the message system is simply adapting to the common design style. Using the same clock for all nodes within one cluster is somewhat less common. Descendants of Caltech's Cosmic Cube tend to favor independent clocks and asynchronous channel protocols. The Ncube systems are a notable exception to this trend. The feasibility of a centralized clock for each cluster will be shown in section 3.3.

Synchronous channel protocols are simpler to design and - given that the message system circuitry is synchronous - faster. Asynchronous protocols can tolerate a wide range of channel delays and do not require a carefully designed clock distribution, instead they need carefully designed synchronizers.

---

[14]Some processors, such as Motorola's 68020, support an asynchronous bus protocol. However, even these processors are internally synchronous and run faster if the external interface is driven by the same clock, thus avoiding synchronization delays.

Beside being convenient for the circuit designer and boosting bandwidth by a small amount, this design decision will be justified by the communication architecture described in Chapter 4.

3.2.2. Packet Switching

There are only two fundamental options: packet switching and circuit switching. Hybrid systems capable of both modes (for example the JPL Hyperswitch) require extra complexity, some of which will be unused at any given time because the modes are mutually exclusive. These modes of operation have significantly different characteristics:

Message Length: Naturally, any message system favors long messages as it improves the data to overhead ratio. Given that, circuit switching depends far more on long messages because the overhead is higher: establishing a complete path from the source to the destination requires the commitment of more resources. Establishing a path is essentially equivalent to sending a 0-length packet (header only) through the network *and* reserving all intermediate channels. A packet needs only one channel at a time. Once a path is established, data transfer may proceed at the full channel speed. Thus the cost is amortized over the entire message. The cost for packet switching is smaller but has to be paid for each hop.

Latency: The latency of pure packet switching grows linearly with the number of hops and the message size[15]. Circuit switching can eliminate the retransmission delay so that latency becomes a linear function of the message length. However, this is accurate only for very long messages where the overhead to establish the connection is negligible.

Load sensitivity: Establishing a circuit requires multiple channels to be committed, which is intrinsically harder than finding a single free channel if the probability for a channel to be busy is high. Thus the time needed to establish a circuit will increase with the network load. Furthermore, establishing a circuit uses resources (locking partial connections). As the time to establish a connection increases, so does the wasted bandwidth. This positive feedback is highly undesirable. Packets don't require any channel capacity while waiting to proceed.

Routing: Closely related to the load sensitivity is the potential for optimal routing decisions. Packet switching requires only one resource at a time which results in a less constrained search space for routing decisions.

Other distinctions between packet and circuit switching were discussed in section 2.1.1. Given that the expected traffic favors short messages and that high network loads are expected, packet switching appears superior.

---

[15]This excludes virtual cut through routing.

3.2.3. Minimal Fixed-Length Packets

It was shown in the previous chapter that there is a significant performance advantage for all packets being of the same size. Naturally, the unused part of a packet is wasted, hence packets of minimal and fixed size are used. It turns out that this approach has several other advantages:

- Buffer space is predictable and small. This allows packets to be stored in *register files*, a structure that is quite compatible with VLSI implementations. The predictable size simplifies much of the data path design.
- There is no need to encode the packet length in the header.
- Routing decisions have to be made in a periodic time frame. This allows for simple, finite state machines that run synchronized to the transmission circuitry. Sending the header of a packet first allows for pipelining such that the routing decisions are completed when the next transmission cycle starts. As a result, packets are sent back to back and no channel bandwidth is wasted.
- The packets of longer messages can use different routes through the network. This increases the effective bandwidth between two nodes in the presence of multiple channels.
- At the lowest level, network traffic is a completely homogeneous *"fluid"*, and many network properties resemble properties of a system of pipes. While this property doesn't necessarily translate to a measurable performance gain, it is helpful to understand several characteristics of this approach.

Naturally, there is a significant disadvantage: the packet header to data ratio is high. This implies that a sizable (but constant) fraction of the network bandwidth is wasted. Given the stated objectives and assumptions, the benefits outweigh the cost.

3.3. Clock Distribution

The distribution of a global clock is perhaps an overrated problem as there are numerous designs and systems in existence [22] that operate on independent clock sources that easily could have used a common clock, thus avoiding all problems associated with synchronizing independent signals. Large systems that operate on a single clock include [106, 13, 112, 123]. A single clock source eliminates the need for synchronizers in many places, most notably in the communication channels. This avoids delays and reduces the potential for intermittent failures[16]. Another advantage (with current technology) of synchronous operation in the abundance of logic synthesis tools tailored for various forms of finite state machines, that the the main building blocks of most control structures.

The most important parameter for a global clock distribution is the skew between any two

[16]It has been shown [24] that perfect synchronizers do not exist. Careful design can reduce the failure rate to acceptable levels with mean times between synchronization failures in excess of the expect equipment lifetime.

consumers of the clock signal. Skew is introduced either by variations of the signal path delay or by noise added to the signal from external sources such as cross talk from other signals or thermal noise from active components. The main skew sources are the logic gates that are commonly used to amplify clock signals. Gate delays are temperature sensitive, change over time due to aging, and vary across different chips (even if they are of the same type). In general, the delay of a digital gate is a poorly controlled quantity: the bounds given in the specification of the manufacturer are frequently so wide that the worst case skew of a few gates exceeds the period of a fast clock. Compared to gates, properly designed, impedance-matched transmission lines are extremely stable.

The key to the design of a large scale, fast clock distribution is to treat the signal as an analog, RF source. Figure 3-1 outlines this approach. The continuously running clock from the crystal oscillator is amplified with a broadband power amplifier. A bandwidth of 3 times the clock frequency is sufficient to reproduce a square wave for digital applications. Broadband RF amplifiers are available with 1 - 2 Ghz bandwidth as building blocks packaged like common digital components. Alternatively, MMICs[17] are superior clock distribution amplifiers when compared to conventional logic buffers.

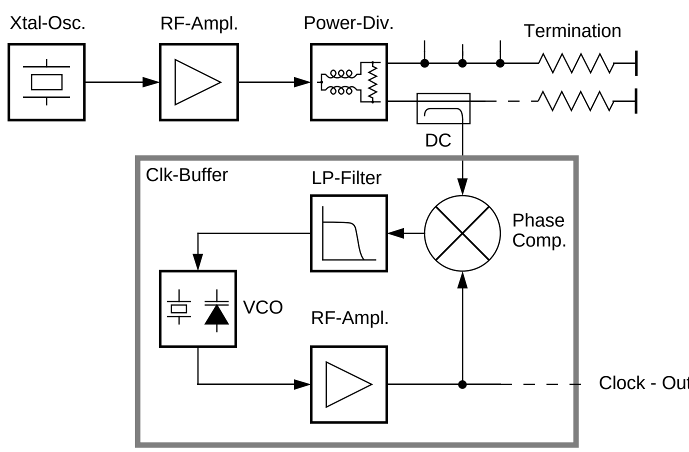


Figure 3-1: Precision Clock Distribution

Phase errors introduced by the RF amplifier itself are small because it is operating at a fixed frequency. The errors don't matter because the clock isn't split yet. Instead of using digital gates to divide the clock signal, passive power splitters are used. Power splitters are

[17]MMICs (Monolithic Microwave Integrated Circuits) are broadband gain elements with I/O characteristics matched to common transmission line impedances, for example 50 ohm. MMICs are available with operating frequencies of several GHz from Avantek, Hewlett-Packard, Mini-Circuits, etc.

available with 2 to 48 outputs and don't introduce any significant[18] phase error due to their passive nature. The disadvantage is their loss in signal amplitude, hence the need for the broadband amplifier upfront. Each branch after the power splitter can support about 10 - 20 clock consumers directly. For larger systems, a low skew clock buffer can be built with a phase-locked loop: a directional coupler (DC) taps a small fraction of the signal from a primary clock branch and feeds it into the reference port of a RF-phase detector. The directional coupler is easily realized with a resistor and a stripline pattern which becomes part of the printed circuit board. The lo-port of the phase comparator is driven by the amplified output of a voltage controlled oscillator (VCO) that also supplies the desired clock signal. The phase detector output passes through a lowpass filter and controls the VCO. Since the operating frequency doesn't change, the lowpass filter can have a very low cutoff frequency so that - besides amplification - incoherent noise is removed from the clock. The phase error introduced by this clock amplifier can be kept well under 5 degrees or about 0.1 ns for a 100 MHz clock with a simple implementation. The phase error is largely independent of the output power, so this clock buffer can drive another RF-power divider.

The bottom line is that the implementation of a large (exceeding 1000 processors), synchronous system with a single, global clock and tight bounds on the skew between any pair of processors is feasible with a conservative design using mature[19], well understood RF-technology.

3.3.1. Synchronizing Distributed Clocks

The communication architecture envisioned in this thesis supports multiple clusters of nodes. Each cluster consists of a number of nodes packaged in one cabinet. The physical proximity of nodes within one cluster allows synchronous operation driven by one central clock. Clusters are connected by medium haul channels comparable to local area networks. While it is possible to distribute a low skew clock over large distances[20], it becomes quite complex. Furthermore, multiple and/or RF-grade cables at LAN distances are quite expensive and inconvenient.

For reasons that will become obvious in Chapter 4, the clocks of all clusters need to be synchronized. However, no tight bounds on the clock skew will be required, so that the clocks of two different clusters may vary their phase as long as their average phase relation is

---

[18]The absolute phase difference of two output ports may differ by as much as 4 degrees, or 0.1 ns at 100 MHz, but this - rather small - error doesn't change much over time, hence could be compensated for by adjusting the interconnection cable length.

[19]Transmission lines, power splitters, RF-amplifiers and PLL circuits have been in common use for decades.

[20]The VLA radio telescope in Socorro, New Mexico maintains coherency over 27Km at microwave frequencies.

constant and transient phase differences do not exceed certain limits[21].

There are no clock wires connecting clusters, instead clocks are recovered from the data transmissions. These transmissions are synchronous and continuous so that a reconstructed clock is always available. Periods with no data to be transmitted are filled with idle packets that exchange low priority status information. Clusters may be connected in arbitrary networks. For each cluster, the recovered clocks of the incoming channels are compared to the local clock as outlined in Figure 3-2.

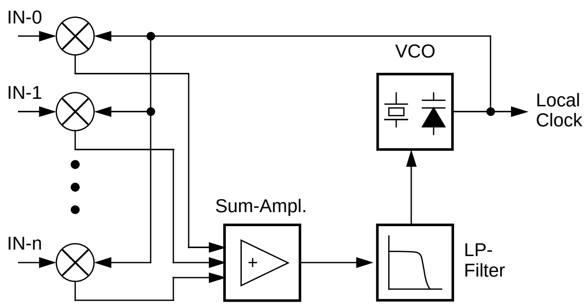


Figure 3-2: Distributed Clock Generator

Each recovered clock (*IN-0,..., IN-n*) is fed into a frequency/phase comparator. Frequency/phase comparators differ from plain phase comparators in their ability to compare the frequency of uncorrelated signals. Plain phase comparators, as used in Figure 3-1, are simpler and more precise in the locked state[22]. However, they produce random output signals if the input signals are uncorrelated. A normal PLL circuit will eventually acquire lock. The *"PLL"* circuit in Figure 3-2 tries to achieve phase lock among a distributed set of oscillators and probably would not work with plain phase detectors.

A typical frequency/phase comparator is outlined in Figure 3-3. This circuit is easily integrated with current CMOS technology. The output acts as a charge pump and is meant to be connected to a high capacitance node. The two output transistors are very small and are acting as current sources that are briefly turned on to add or remove charge from the output node, which becomes part of the subsequent low pass filter. This has the advantage that the outputs of several comparators can be tied together to form a cumulative integrator. In the locked state, the net output current becomes 0. Multiple units can be integrated on one chip such that both output transistors are disabled if the corresponding channel is unused or loses synchronization. A detailed discussion of this circuit can be found in [119].

---

[21] The absolute phase error may approach 1/2 of the packet transmission time or about 32 clock cycles. This limit is about two orders of magnitude larger than the performance of the described synchronization scheme.

[22] A simple diode ring mixer or an exclusive-or gate are satisfactory phase detectors.

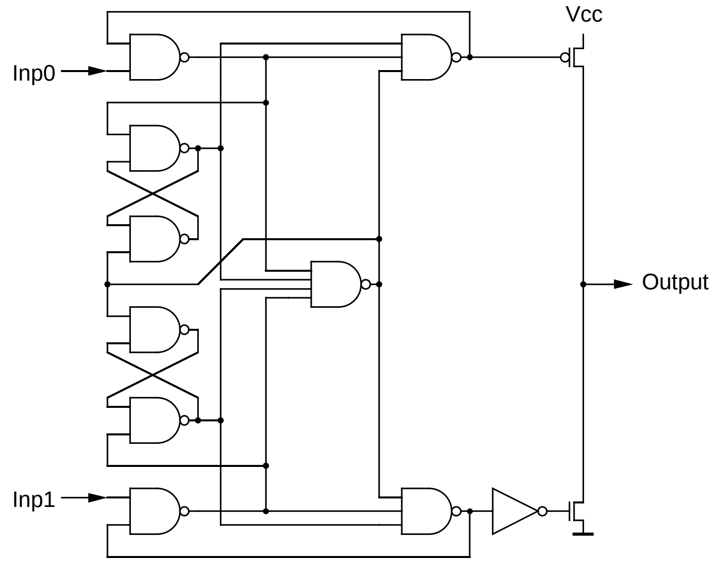


Figure 3-3: Frequency/Phase Comparator

It is important to note that the VCO of Figure 3-2 is crystal based. The operating frequency of a crystal oscillator is very well defined. Tolerances of 10^{-6} are not uncommon. The assumption that the open loop center frequencies over the operating temperature range of all clusters are within $\pm 10^{-4}$ of the design specification is quite conservative. Crystal oscillators can be electrically tuned within a narrow range of about $\pm 10^{-3} \cdot f_0$. Such a voltage controlled (crystal-) oscillator is a good local clock source even if the control loop is disabled or in an arbitrary state.

To demonstrate proper synchronization of circuit 3-2 a system of n nodes is analyzed. The absolute phase $\phi_i(t)$, $i = 1 \ldots n$ of each oscillator depends on the time t. If the system was turned on at $t = 0$, the phases are:

$$\phi_i(t) = \phi_i(0) + \int_0^t (\omega_i + K_o K_d \sum_{i \neq j} a_{ij} (\phi_j(x) + d_{ij} - \phi_i(x))) \, dx \qquad (3.1)$$

ω_i is the frequency [rad/sec] of the i^{th} oscillator if the control voltage is set to 0. K_o is the conversion factor for the VCO in rad per volt, and K_d is the sensitivity of the phase detector in volt per rad. At this point, a linear phase detector is assumed. It can be shown that limiting the output of the phase detector will still result in a proper synchronization. This limiting is essentially the characteristic of a frequency/phase detector. A plain phase detector would produce ambiguous and non-monotonic outputs that invalidate this analysis. The product $K = K_o K_d$ is the dimensionless loop gain.

The network topology is specified by the coefficients a_{ij} which is the number of channels from node j to node i. Assuming $a_{ij} = a_{ji}$ simplifies the analysis but is not strictly necessary.

Each channel has a certain delay d_{ij} which is expressed in terms of a phase shift of a signal with the steady state operating frequency ω.

Differentiation of Equation (3.1) yields a non-homogeneous linear equation system:

$$\frac{\partial}{\partial t}\vec{\phi}(t) = \mathbf{A}\vec{\phi}(t) + \vec{f} \tag{3.2}$$

with:

$$\begin{aligned} &\vec{\phi} = [\phi_1 \cdots \phi_n]^T \\ &\mathbf{A} = [a_{ij}]_{i,j=1}^{n} \quad ; \quad a_{ii} = -\sum_{i \neq j} a_{ij} \\ &\vec{f} = [f_1 \cdots f_n]^T \quad ; \quad f_i = \omega_i + K \sum_{i \neq j} a_{ij} d_{ij} \end{aligned} \tag{3.3}$$

The structure of **A** is critical to the remaining steps. **A** represents a strongly connected graph (i.e., the inter-cluster network) and is therefore irreducible [81]. Because of the assumption that a channel from node i to node j implies the existence of a channel in reverse direction, $\mathbf{A} = \mathbf{A}^T$. Furthermore since **A** is real, it is also hermitian. Therefore **A** has only real eigenvalues. The diagonal elements of **A** are negative semi-dominant, hence **A** is negative semi-definite: all eigenvalues $\lambda_1 \cdots \lambda_n$ of **A** are ≤ 0. Equation (3.3) also implies that one eigenvalue is 0 and that the corresponding eigenvector is $\vec{1}$ because $\mathbf{A}\vec{1} = 0$.

Let $\mathbf{P} = [\vec{x}_1 \cdots \vec{x}_n]$ be the matrix of right eigenvectors of $\mathbf{A}$, $\mathbf{D} = diag[\lambda_1 \cdots \lambda_n]$ be a diagonal matrix composed of the corresponding eigenvalues, and $\mathbf{Q} = [\vec{y}_1 \cdots \vec{y}_n]$ be the matrix of left eigenvectors. Therefore $\mathbf{A} = \mathbf{PDQ}^T$. Rewriting Equation (3.1) yields:

$$\frac{\partial}{\partial t}\mathbf{Q}^T\vec{\phi}(t) = \mathbf{DQ}^T\vec{\phi}(t) + \mathbf{Q}^T\vec{f} \tag{3.4}$$

This decouples the equation system into n independent differential equations. Introducing $z_i(t) = \vec{y}_i^T \vec{x}_i(t)$ and $g_i = \vec{y}_i^T \vec{f}$ leads to:

$$\frac{\partial}{\partial t} z_i(t) = \lambda_i z_i(t) + g_i \tag{3.5}$$

if $\lambda_i = 0$:

$$z_i(t) = t g_i + c_i = t \frac{1}{n} \sum_{l=1}^{n} f_l + c_i \tag{3.6}$$

otherwise:

$$z_i(t) = c_i e^{\lambda_i t} - \frac{g_i}{\lambda_i} \tag{3.7}$$

Backsubstitution with $\vec{\phi}(t) = \mathbf{P}\vec{z}(t)$ gives the phase functions for each oscillator. The integration constants c_i, $i = 1 \cdots n$ could be determined by the state of the system at $t = 0$. However, the precise form of the phase function is not required to demonstrate that the system synchronizes properly, because the phase functions of any system are mere linear combinations of functions described by either Equation (3.6) or (3.7).

Equation (3.6) yields the steady state operating frequency ω, which is the coefficient of t:

$$\omega = \frac{1}{n}\sum_{i=1}^{n}(\omega_i + K\sum_{j \neq i} a_{ij} d_{ij}) \tag{3.8}$$

As expected, this is mainly the average of the open loop frequencies of the individual oscillators. The delays of each channel contribute to a net increase of the operating frequency. This sets an upper limit of the loop gain because of the narrow tuning range. It turns out that this limit is largely irrelevant because the loop gain is subject to the normal stability consideration that governs the selection of loop gain, natural loop frequency, lowpass filter cut-off frequency, etc. which constrains K even further. A low gain, low natural loop frequency is desirable to make the system less sensitive to transient errors. This reduces the capture range (a non-issue here) and increases the time required for synchronization.

Equation (3.7) describes the turn-on transient. This is essentially an exponentially decaying function of time because $\lambda_i < 0$. The non-zero eigenvalues are proportional to K, so that higher loop gain reduces the duration of the synchronization time. It is interesting that the convergence of the system is an exponential function in time.

Figure 3-4: Clock Synchronization for 32 Nodes

Figure 3-4 shows the phase functions for a randomly connected ($D = 2$) 32-node configuration. K was set to $8 \cdot 10^{-6}$ and the plot spans about 200,000 cycles. Initial phases and all channel delays were randomly drawn from $[-\pi, \pi)$. The open loop frequencies were randomly drawn from $[1 - 10^{-4}, 1 + 10^{-4}]$. The phase detectors were limited to $\pm\pi$ and the lowpass filter had a 3db point of 0.0002 [rad/sec]. It should be noted that the low-pass filter has plenty of bandwidth and it does not significantly alter the analysis presented above. The low pass filter was included in this numerical solution to the phase equations to show an example of a complete system. Phases were plotted over a range of $\pm 180^o$ with respect to the steady state phase. Halfway through the run, an artificial upset of the phase for node 16 by 180 degrees was introduced to simulate a transient error in one node. Very little of this temporary error is propagated to the attached nodes (17 and 24).

3.4. Communication Channel Characteristics

The message system will consist of only two principal components: the routing element and the communication channels. The routing element is assumed to fit into one integrated circuit and will be discussed in depth later.

Communication channels consist of the media (wires, connectors, etc.) and the interface circuitry. The characteristics (bandwidth, delay, error rate, etc.) of a channel depend on their physical distance. Generally, with increasing distance, it becomes more difficult to transmit data.

Designs for message-passing multiprocessors usually consider only one type of channels. Because current machines are generally confined to one cabinet, these channels are relatively short and can use typical mainframe interconnect technology such as backplanes, ribbon cables, and printed circuit boards. Within the scope of this thesis, such a collection of processing nodes is referred to as one cluster.

Traditionally, multiple clusters are treated as completely separate systems that could be linked through a local area network. Breaking with tradition, this thesis integrates the linkage into the message system design. The goal is to link two or more clusters in such a way that the combined system logically behaves as one homogeneous machine. Hence part of the functionality of a typical local area network becomes part of the message system.

The interconnection of multiple clusters requires far more bandwidth than conventional local area networks (LAN) can provide. Furthermore, the typical protocols used in contemporary LAN's are too cumbersome to provide low latency communication. Therefore a new interconnect structure is developed.

Since the physical and electrical constraints for inter- and intra-cluster channels differ substantially, two distinct types of channels will be assumed. The characteristics of each type will be discussed in the subsequent sections 3.4.1 and 3.4.2.

These discussions are each based on fairly conservative implementation studies that will be used to provide realistic performance estimates for subsequent simulations. The details of these implementations are not essential to the concept of the proposed architecture: faster circuits and denser interconnect technology will eventually lead to faster intra-cluster channels and new media may speed up the inter-cluster channels. However, the latency distinction between these two types will remain, due to the finite speed of light.

3.4.1. Intra-Cluster Channels

Within one cabinet, the channel media is simply a bundle of wires. As in any high speed digital system, a *wire* must be treated as a transmission line. There is little point in building a multiprocessor system with low performance interconnection technology, such as wire-wrapping. A detailed and practical treatment of transmission lines is given in [93]. The key points of transmission line designs are:

Delay: The delay depends on the type of the transmission line (coax cable, twisted pair wire, (micro-)strip lines), the geometry, and the dielectric constant of the insulating material. In any event, delays are quite significant (order of 5ns per meter) and the signal paths of a channel should be of equal delay.

Termination: Transmission lines must be terminated to avoid reflections. Reflections degrade the signal fidelity and can cause incorrect signal levels. Since signal aberrations due to reflections persist for multiples of the signal delay, they can severely limit the data rate.

Secondary Considerations: The attenuation, bandwidth limit, and dispersion of transmission lines become a factor for extremely high performance systems, for example a GaAs-based Cray-3. At the more modest speeds of $1 \cdots 1.5\,\mu m$ custom CMOS, these factors are not critical.

Given the physical proximity of nodes, channels can use two common signals: the system clock and a reset wire that is asserted at boot time. Signal delays are a fraction of a bit-time for a conservative design and the direction of a channel can be reversed. Table 3-1 lists the approximate characteristics of the expected interconnection media.

Intra-Cluster Media	
# of signal paths	4 – 8
signal delay	$\leq 10\,ns$
symbol time	$40\,ns$
physical bandwidth	$12.5 - 25\,Mbyte/sec$
common clock	yes
common reset	yes
bidirectional	yes

Table 3-1: Intra-Cluster Channel Characteristics

3.4.1.1. Transceiver Implementation

Transmission line termination frequently uses discrete resistors or resistor packs. These extra components take up PC board space and add to the overall system complexity. Furthermore, parallel termination (typical for ECL systems) increases power consumption.

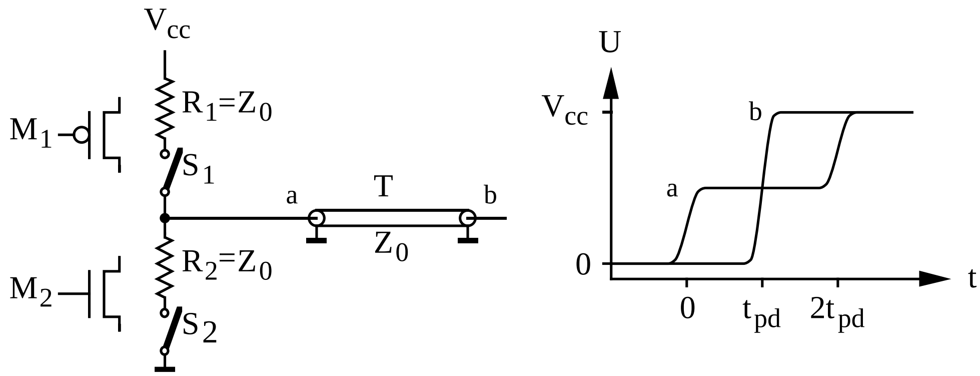


Figure 3-5: Self-Terminating CMOS Output Driver

Knight and Krymm [78] developed a method to incorporate the termination into the output buffer of a CMOS IC. The basic idea is reproduced in Figure 3-5. The transmission line T is series terminated at the source. Instead of using one real resistor connecting a low impedance buffer to the transmission line, the termination resistor becomes part of the final output stage. For example, if the output sends a logic *high* signal, S_1 is closed and S_2 is open. The transmission line is connected to V_{cc} via R_1. If the value of R_1 matches the transmission line impedance Z_0, reflections from the unterminated destination end of T will be absorbed at the source. The far end of the transmission line B will see a clean swing from 0 to V_{cc}, while the near end will appear as a real resistor for $2t_d$. R_1 and R_2 are actually implemented by partially turning on the output transistors M_1 and M_2. The driver can be made adaptive by sensing the half swing plateau during transitions. For example, if the plateau for a *high* to *low* transition is higher/lower than $V_{cc}/2$, the gate potential for M_2 is increased/decreased.

The beauty of this circuitry lies in the lack of external components and the tolerance for variations of the transmission line impedance. Furthermore, current is used only during transitions and the circuit is capable of bidirectional operation. Due to the series termination, the device can cope with an output conflict (both ends of a bidirectional signal path are accidentally driven with different logic values). Output conflicts do not result in current spikes, so the switch-over timing for a signal direction change becomes uncritical.

3.4.1.2. Channel Protocol

Besides the signal paths, a certain amount of control logic is necessary to build a usable channel. This control logic has to execute a protocol to coordinate the sending and receiving of data. The protocol functions are:

- Establish synchronization between the controllers on each end of the channel.
- Decide on a data flow direction for each transmission cycle.
- Determine if a transfer was successful.
- Recover from transmission errors and/or loss of synchronization.

These functions require a certain amount of information being exchanged between the controllers. Many channel implementations use a number of extra wires for this purpose, for example *data-ready* and *acknowledge* signals. This *out-of-band* signalling has the advantage that control signals cannot be confused with data signals and that the protocol is relatively simple. The disadvantage is the requirement of extra wires. Wires are a scarce resource and the control signals do not require that much extra bandwidth. *In-band* signaling does not use any extra wires: control signals are carried over the normal data path. A time-multiplexing scheme determines when data is sent and when control signals are exchanged.

The first task of the protocol is to establish synchronization. Since both sides of a channel are within one cluster and *see* the same clock, synchronization is merely a matter of resetting each controller to an appropriate initial state.

Actually, the *common clock* assumption requires a bit more consideration. There are a number of different clocks:

The *system clock*: This is the fastest periodic signal in a node. It essentially controls the execution of all state machines and all other signals are derived from this clock.

The *transmit clock*: Data is moved over a channel at this rate. The transmit clock period defines how long a bit is on an I/O wire. The transmit clock may be equal to the system clock, but in general it will be an integer fraction of the system clock. A high end CPU may use a system clock of 40 MHz; however transmitting data at this rate (25 ns) is quite demanding.

The *transmission cycle clock*: Since there are fewer wires per channel than there are bits in a packet, several transmit cycles are required to send one packet. The transmission cycle clock is simply a signal that is asserted once per packet transfer. Since packets are of fixed size, this clock is an integer fraction of the transmit clock.

The common clock assumption states that all of these clocks are in lock for all nodes within one cluster. There are several ways to achieve this synchronization without inflating the number of clocks. One way is to distribute only the slowest clock. All other clocks can be synthesized on-chip with a PLL circuit. This approach is used by the Transputer: a common 5 MHz clock is distributed and all members of that processor family derive their actual clocks from that signal. Alternatively, the system clock can be distributed and other clocks are generated with simple counters / dividers. This requires some means of keeping the counters in each node synchronized, for example by overloading the system-wide reset signal with a second function: a continuously asserted reset signal causes the initialization. After completing the initialization, the running system will periodically assert the reset to synchronize the slower clocks[23].

System integration is simplified if the channel control circuitry is completely symmetrical. A channel interface emerges from the message system chip as a collection of pins that may be connected to any other channel. In particular, no distinction is made between transmitters and receivers. However, any deterministic protocol can't be completely symmetric.

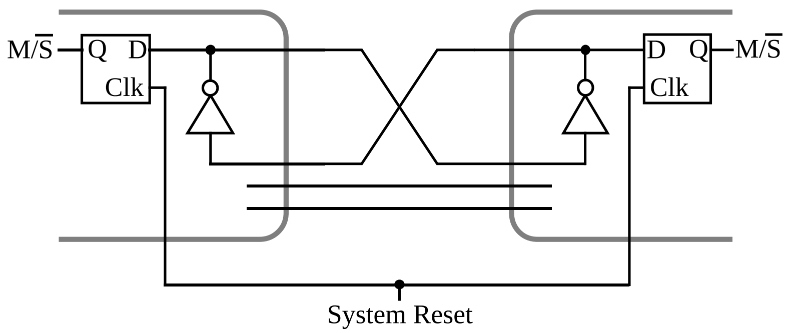


Figure 3-6: Channel Initialization Procedure

Figure 3-6 describes a circuit to initialize a channel. While the *system-reset* signal is asserted, each channel interface will turn off the output drivers for the I/O pins $1 \cdots n$. I/O pin 0 is driven with the inverse of I/O pin 1 and each channel connection transposes these 2 signals. Thus a distributed R/S flip-flop is formed. Given a long enough reset period, this flip-flop will settle in a defined state. At the end of the reset period, this state is saved in each interface. This bit designates one controller as the *master* and the other as the *slave* for each individual channel. This distinction will be used to start the protocol and to recover from loss of synchronization.

After the reset period, the master will start transmitting a packet. There are three parts to a transmission, sent sequentially over a channel: the actual packet, a postamble, and an acknowledge. The structure of the packet is (almost) immaterial to the channel operation. No interpretation of the packet takes place, except for the generation and absorption of idle packets. All other packets are simply treated as sets of bits.

[23]Optimistic designs may simply rely on all clock dividers to stay synchronized, but that would result in a rather fragile system due to the cumulative failure characteristic.

The postamble is sent after the packet and in the same direction. It contains 2 bits: a virtual channel flag and a parity bit[24]. The virtual channel flag bit will be described in section 3.4.1.3.

A parity bit is included so that the channel is able to detect single bit errors. The parity bit is set such that the number of bits in the packet plus the parity bit is even. By convention, a packet consisting entirely of 0's is not allowed[25]. Furthermore, the packet length in bits must be even. This convention ensures that any correct transmission will cause at least one transition of the channel wire. This addresses the problem of detecting failures that leave the channel wires undriven. Given that the receiver circuits have a high impedance, wires will maintain their last level for more than a packet transmission time unless they are driven. Requiring at least one signal transition per packet transmission will cause any steady signal to be recognized as an error. Such a situation may arise if a channel is unused or if controllers lose synchronization and try to receive at the same time.

The acknowledge symbol sends three bits in the reverse direction: the *synchronization check* bit, the *skip* bit, and the *blocked* bit.

The synchronization check bit is the complement of the parity bit. This inverted echo guarantees that failure to respond will be detected. Transmission errors (parity) and synchronization loss are combined in one error condition that are detectable at both ends on the channel.

The skip bit is used to control the flow of data. By default, data flow is reversed on each transmission cycle. However, if one side has no packets to send, it skips its transmission slot. The skip bit is asserted if the receiving node has no packets to send and will continue to receive packets. The transmitting node is expected to send a packet in the next transmission cycle. If both nodes lack traffic, an idle-packet is sent.

The block bit is used as a flow-control mechanism. A transmission may have been successful, but the receiving node lacks space to store the packet. Unlike a transmission error, the router may retry sending the packet.

Figure 3-7 summarizes the protocol. After system reset or in the event of an error, the *master* end of a channel will start sending a packet and the *slave* will start receiving. A change of roles takes place if the receiving node does not assert the *skip* bit as part of the

---

[24]The communication architecture (section 4.4.3) will add two bits to the postamble and to the acknowledge symbol.

[25]Part of the packet is a type field and the 0-type is reserved for the idle packet, which in turn has a 'one' elsewhere.

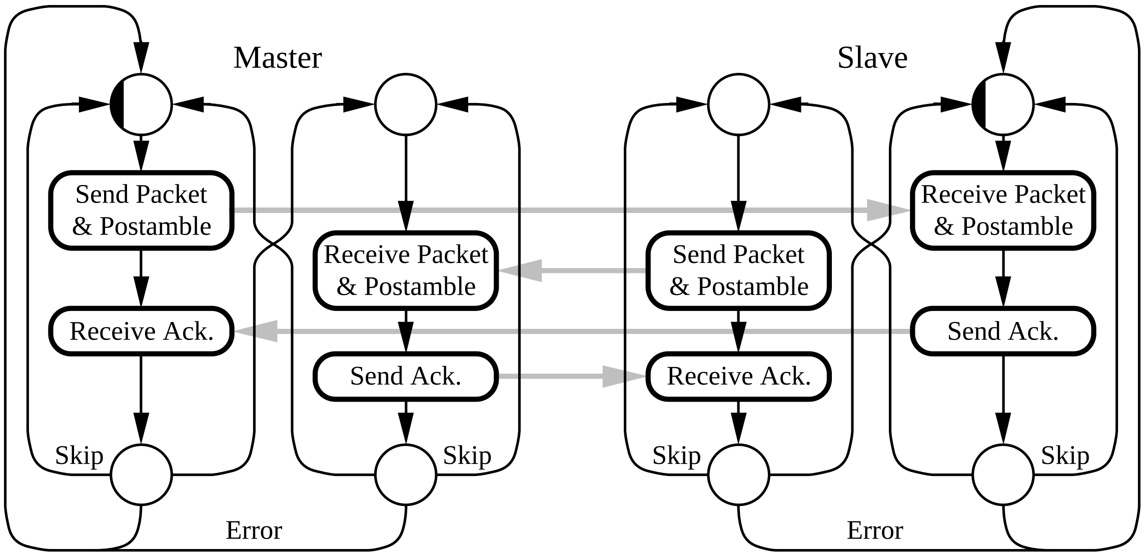


Figure 3-7: Channel Protocol

acknowledge. Synchronization and/or transmission errors will cause both nodes to return to their initial state. Proper operation requires that both nodes are able to detect the error. Three cases are possible:

- A loss of synchronization causes both nodes to send during the same transmission cycle. Since no node was receiving, no acknowledge will be sent and the lack of a transition of the synchronization check bit will be detected at both ends[26].
- A loss of synchronization causes both nodes to receive during the same transmission cycle. Since both nodes are receiving, no transition will occur and both nodes will find an invalid packet.
- A bit error occurs during a properly synchronized transmission. The receiving node does not acknowledge the transfer and the sending node will detect the lack of a transition of the synchronization check bit.

In Chapter 4, this error-signalling facility will be (mis-)used to reject packets in order to constrain the transmission path in certain cases. Hence, only the receiving node should log error events because it is able to distinguish between a real hardware problem and a rejection.

[26]Because of the series termination, wires driven at both end with conflicting signals will settle at $V_{cc}/2$. Given enough hysteresis of the receiver, this condition could be recognized as a lack of transitions. More realistically, the actual implementation may accept a 50% chance for immediate recovery. Depending on how much weight is given to fault tolerance, more channel bandwidth can be allocated for redundancy. The protocol presented here uses the bare minimum.

3.4.1.3. Virtual Channels

One approach to avoid deadlocks is to multiplex two (or more) virtual channels over one physical channel. Bandwidth is split fairly across all virtual channels and virtual channels cannot block each other. The originating node decides which virtual channel gets to send its packet. Any virtual channel without a pending packet is ignored. The receiving node needs to know which virtual channel sent the packet because each virtual channel has its own receive buffer. Disjoint storage space is required to ensure lack of interference.

For the proposed message system, two virtual channels are supported. Therefore only one extra bit is required in the postamble.

3.4.2. Inter-Cluster Channels

Channels meant to connect distributed clusters face a significantly different environment. The maximal physical distance for inter-cluster channels is (arbitrarily) set to 200m. This is roughly similar to the dimensions of high speed local area networks. The typical media includes twisted pair wires, coaxial cables, and optical fibers. This list is roughly ordered by cost. Twisted pair wires are relatively cheap, frequently bundled, and offered in a wide variety of types. Coaxial cables are generally somewhat more expensive and offer higher bandwidth. Optical fibers are considerably more expensive due to the need for conversion between electrical and optical signals. The characteristic advantage of optical fibers - high bandwidth over long distances - is quite small at 200m.

Currently - that is in 1989 - a bundle of twisted pair cables offers the best bandwidth to cost ratio. This may change in favor of optical fibers at some point. However, the optimal bandwidth per cost ratio will likely remain in favor of multiple medium-performance channels as opposed to a few high-performance channels. It is easier, more flexible and cost effective to design an inter-cluster network out of multiple medium performance channels with a low incremental cost and near linear scalability. Another factor favoring a network of many cheap channels arises from the observation that higher fan-out networks offer higher topological bandwidth and better fault tolerance.

Based on these considerations, the intended inter-cluster media are bundles of twisted pair wires. Because of the potential distance between clusters, a number of design parameters differ from those of intra-cluster channels: the wire-delay becomes more critical (exceeding the time to transmit one bit), the number of wires faces practical constraints, the electrical characteristics are more unfavorable (attenuation, dispersion, noise, etc.) and the cost of the wire is no longer negligible. Table 3-2 summaries the features of the inter-cluster media.

Inter-Cluster Media	
# of signal paths	1
signal delay	$\leq 1000\,ns$
symbol time	$50\,ns$
physical bandwidth	$2.5\,Mbyte/sec$
common clock	no
common reset	no
bidirectional	no

Table 3-2: Inter-Cluster Channel Characteristics

3.4.2.1. Twisted Pair Wire Interface

Wires that leave the cabinet face the *real* world, a much harsher environment. Lacking the shielding of a metal box (Faraday cage), external wires are subjected to a wide range of electronic interference from other equipment, various rf-sources, and electrostatic discharges. Furthermore, there are strict limits on the amount of rf-signal that an external wire may emit.

Ideal twisted pair wire does not emit or receive electromagnetic waves if it is driven symmetrically. Real wires are quite close to the ideal and a shield (common to a bundle) can take care of the residual near-field emission. The easiest way to ensure symmetry is the use of a broad-band transformer[27] [28]. This approach has the additional advantage of electrically isolating the external wire from the cluster. Thus immunity from electrostatic discharges and improved common-mode rejection is achieved.

Figure 3-8 outlines the CMOS interface circuitry to drive twisted pair wires (TPW). $T1$ and $T2$ are 1:1 broad-band transformers. $T1$ is driven by an H-bridge configuration. This is simpler than using a center-tapped primary for $T1$ and avoids the problem of pin potentials exceeding the rails. Alternative approaches would require an additional power supply voltage, which is also undesirable. Another advantage of the H-bridge is an increase of drive capability: the primary of $T1$ could be driven with $V_{ss} = 2\,V_{cc}$. Drive capability will become important due to expected attenuation of the TPW.

---

[27]A broad-band transformer uses a small ferrite toroid with two sets of windings. The usable bandwidth ranges from 10Khz to about 400 MHz for small, low cost devices. Several transformers can be packaged in one case similar to typical IC's. Electrical isolation up to about 1000V is common.

[28]Symmetrical twisted pair wire drivers are quite common for ECL and high-speed interface circuitry. In a practical test, it was found that these devices have a significant asymmetric component such that the addition of a transformer caused a noticeable increase in signal to noise ratio.

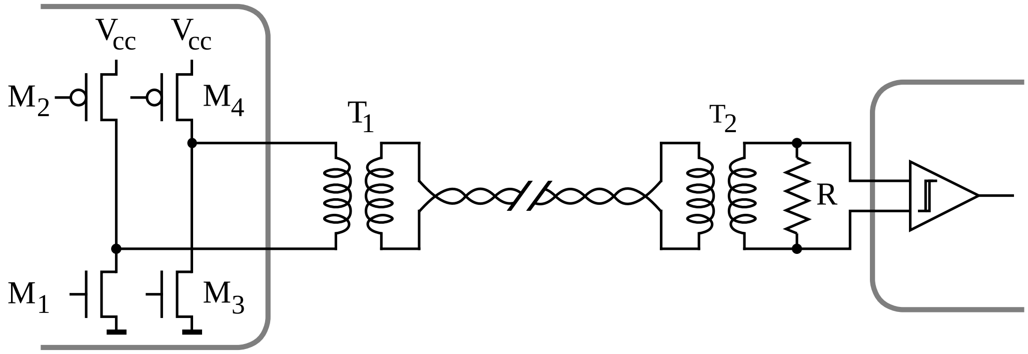


Figure 3-8: Twisted Pair Wire Interface Circuits

The transistors $M1/M3$ are driven into saturation. They will become a low impedance connection to ground when they are switched on. $M2/M4$ are switched on only partially such that series termination of the TPW is achieved. This uses the same method as in circuit 3-5. The measured impedance of the TPW used in a test set-up was 91 ohms. Typical TPW's have impedances of about 80 to 130 ohms, well within the drive capabilities of CMOS output stages. It turns out that a considerable improvement of the signal to noise ratio for long TPWs is possible through pre-emphasis. A boost of the high end of the used spectrum is achieved by increasing the gate potential for $M2/M4$ for one bit period after each transition.

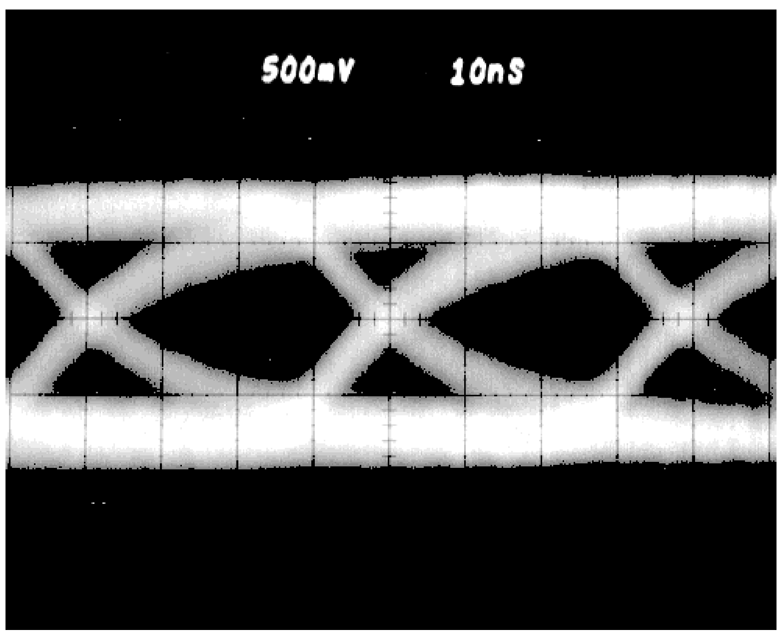


Figure 3-9: Inter-Cluster Channel Eye-Pattern for Random Data (without Pre-Emphasis)

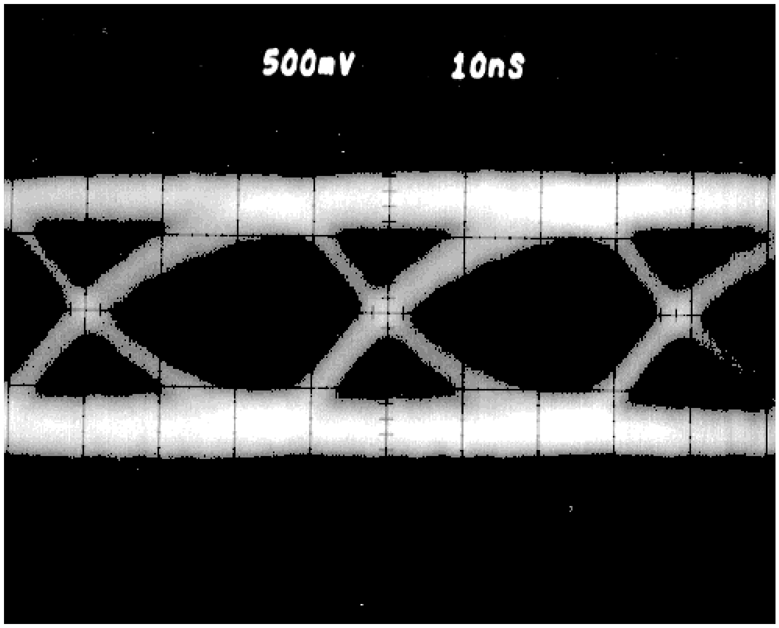


Figure 3-10: Inter-Cluster Channel Eye-Pattern for Random Data (with Pre-Emphasis)

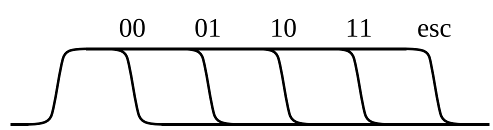
Figure 3-9 shows the eye-pattern of a run length limited code over a 100m TPW. The characteristics of this cable are given in appendix D. A 25 MHz bit clock is used with a discrete implementation of circuit 3-8. A 64 bit shift-register with linear feedback is used to generate a stream of random data that is encoded in a variable pulse width code. The pre-emphasis used for the right scope picture decreases the driver impedance to 45 ohms for the duration of 40ns after each transition. This simple method of enhancing the high frequency response is both highly effective and easy to implement.

The receiver part of Figure 3-8 terminates the TPW with its characteristic impedance and uses a comparator to recover a digital signal. A permanent termination is possible because the TPW is directional. The switch-over time due to propagation delays is much too high for efficient bidirectional operation. A discrete termination is simpler than active termination and more robust. Short connections will cause a noticeable load on the termination resistor. The comparator needs about 10mV of hysteresis, or about 10% of the minimal expected signal amplitude. This hysteresis helps to improves noise immunity.

Transformer coupling is not able to transmit DC-levels. Consequently, the transmission code may not send a constant signal. Long sequences of ‘0’ or ‘1’ also cause problems in the data reconstruction. The receiver has to recover some time reference to decode the signal. Encoding methods for systems with bandpass characteristics are in widespread use. For example, all magnetic recording media face the same problem. In general, the AC coupling and data recovery problem is solved with run length limited (RLL) codes that specify a ceiling on the time between two transitions.

The RLL code is proposed for use in the FDDI system and is specified in the ANSI standard X3T9.5: 4 bits of data are encoded in a 5 bit pattern. The resulting bit pattern is subsequently NRZI encoded such that each ‘1’ in the pattern causes a transition of the transmitted signal. This code guarantees at least 2 transitions per symbol and a maximum transition separation of 3 bit times. This code causes a 20% loss of raw bandwidth that is used to provide a time reference. There are denser RLL codes which can place transitions on fractional bit-clock periods at the expense of tighter group delay tolerances. The 4B/5B code is not perfectly DC-free because pathological data patterns may cause a 2:3 duty cycle over arbitrary long periods. Such relatively small DC components pose no significant problems to a transformer coupled system. However, the signal to noise ratio is reduced because the 0-threshold of the comparator will shift. Due to finite slew rate of a bandwidth-limited signal, the duty cycle of the received signal will change. For example, a unity signal with a duty cycle of $a:b$ will cause the threshold to shift to $\frac{a}{a+b}$. The speed of this change depends on the lower cutoff frequency. In general, the threshold will change slowly, giving rise to some intersymbol interference. A 2:3 duty cycle will change the threshold by 10% and reduce the noise margin by at least this much. Furthermore, if the rise time is r (expressed in bit-times), the a-cycle time will change by $r\frac{b-a}{b+a}$. Again for a 2:3 signal with a *natural* rise time, the a-period is changed by 10% of one bit-time[29].

The 4B/5B code and related codes require the reconstruction of the bit-clock at the receiving site to decode the data. This is typically done with a precision PLL circuit. The code provides enough transitions for a PLL to stay locked. An alternative to the 4B/5B code uses variable pulse-width modulation. Two bits of information are encoded in the time between two consecutive transitions:



In-band control signals are encoded through an escape sequence that uses a longer pulse (esc). In the simplest case, the time between transitions is an integer multiple of the bit-clock period. This achieves the same average density as the 4B/5B code. Higher densities are possible if the pulse-width uses fractional bit-clock periods. Provided that the minimal pulse-width does not change, the nominal bandwidth of the code remains constant. This allows

[29]A signal changing at the maximum transition rate will appear as a sine wave of the upper pass band frequency. Normalized to an amplitude of 1, this signal is used to define the natural rise time. In this case, the received signal is approximated by a trapezoidal waveform.

tradeoffs between code density and signal to noise ratio. Short channels with less signal degradation may use denser codes. Other advantages are:

- Instant synchronization. The escape sequence can be recognized even if synchronization was lost. Furthermore, no preamble is necessary to ensure that the receiver PLL has locked.
- Receiver simplicity. The decoder is based on a precision delay element and finite state machine. While the total transistor count is comparable to a 4B/5B decoder, no dedicated PLL is need. PLLs frequently require external tuning and/or a large capacitor for the lowpass filter section. Given the desire to integrate multiple receivers on one chip, extra pins or large chip areas devoted to the filter capacitor are undesirable.
- True DC freedom. A transition stuffing mechanism (described below) ensures perfect DC balance to enhance the noise margin and minimizes intersymbol interference.
- Data compression. The expected traffic (data and control) will have a higher fraction of 0's than 1's. Due to the variable-length nature of the code, 0's take less time to transmit than 1's.
- Selftest. The fill transitions provide synchronization tests. The receiver *knows* when a fill transition is expected and fill transitions are always at the beginning of a minimal length pulse. Thus receiving a pulse of a different size indicates that synchronization was lost.

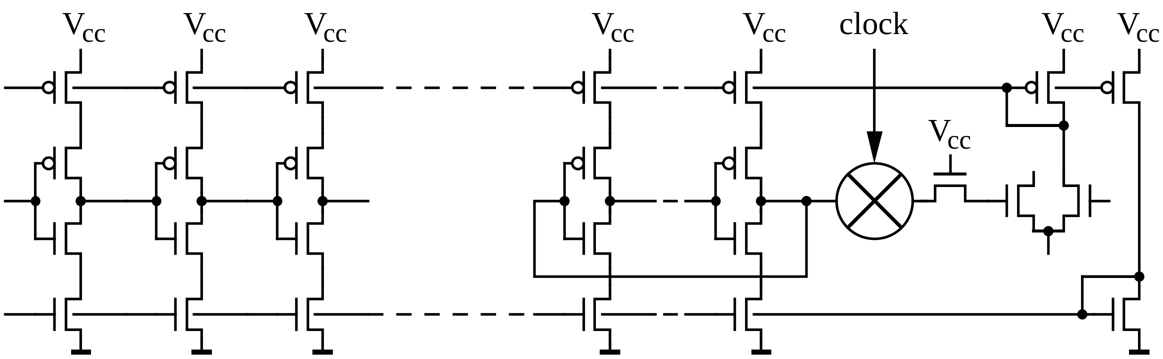


Figure 3-11: PLL Stabilized Delay Line

The timing element of the decoder is a tapped delay line (Figure 3-11). This delay line is stabilized with a PLL circuit to minimize the impact of temperature, supply voltage, and process variations on the delay. It should be noted that only one stabilizer is need for all delay lines on one chip. Furthermore, it operates on the system clock frequency so that the filter section is largely non-critical because the operation frequency is high and a stable, periodic signal is available. This allows for a much simpler low pass filter than the one needed in a data clock recovery circuit.

Figure 3-12 outlines the decoder circuit. The received signal enters a delay line of 5 bit periods. Eight gates are provided to determine the 5 possible pulse widths (4 for each

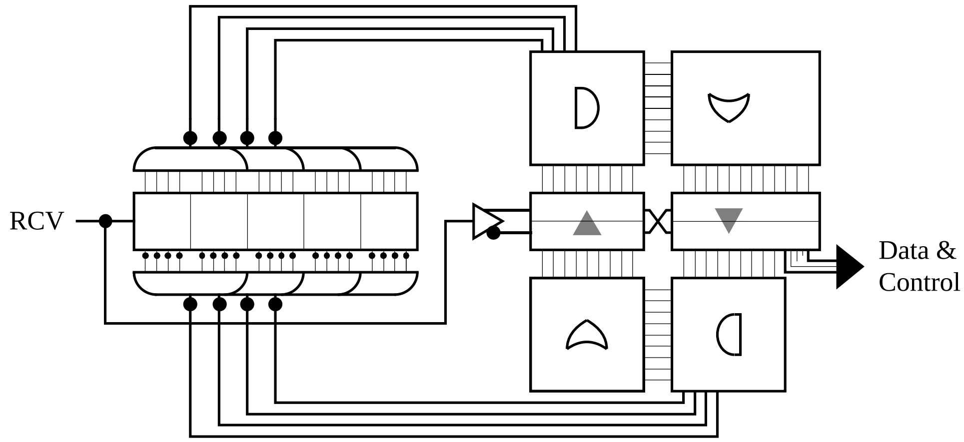


Figure 3-12: Decoder Circuit

polarity). Multiple tapping per bit may be used to build a *digital low pass filter*. Using the fact that transitions move from left to right through this circuit, only two transistors per input for each group of gates are needed.

A conventional, PLA based finite state machine implementation is replicated twice such that one half is used for 0 to 1 transitions while the other half takes care of 1 to 0 transitions. Data of two pulses are combined to a 4 bit symbol that becomes available along with an appropriate clock.

During normal operation, this circuit is free of metastability problems. Marginal signals and/or interference may cause pulses of unexpected width which in turn may cause the width gates to change near a clock transition. To minimize the impact of this potential failure mode, the outputs of the width gates are fed through metastability hardened registers as described in [32].

Figure 3-13 shows a simplified version of the finite state machine (FSM) for the decoder. This FSM is expressed as a C-code description of the transfer function that specifies a *NEW* state in terms of the *OLD* or current state and the inputs to the FSM. Tools are available to directly translate this description into a PLA [102]. This description is also parameterized by a DC-threshold value (*DCT*) that controls how much the duty cycle may temporarily deviate from 1:1 before a fill transition is inserted. There is a tradeoff between this threshold, the circuit complexity, and the amount of bandwidth lost to fill transitions.

Table 3-3 shows these tradeoffs for various DC-thresholds. The number of states is largely due to the counter structures which are folded into one product machine by *c2fsm*. After minimization, the resulting PLA's are quite small (10-12 state bits, about 20 product terms). The last row of Table 3-3 refers to an implementation that does not guarantee a 1:1 duty cycle.

```
#define DCT     5              /* DC threshold                  */
#define RESET   3              /* Channel reset                 */
%input  dly_tap(dt4, dt3, dt2, dt1);
%output data(d1, d0) = 0,
        cntl_valid   = deasserted,
        data_valid   = deasserted,
        error        = deasserted;
%internal
        rcv          = deasserted;
        esc          = deasserted;
        int dc       = 0;
%%{
    int        i;
    deassert(data_valid);      /* assume defaults               */
    deassert(cntl_valid);
    data = 0;
    deassert(esc);
    change(rcv);
    switch (dly_tap) {         /* determine pulse width         */
        case 15: i = 1; break;
        case 7 : i = 2; break;
        case 3 : i = 3; break;
        case 1 : i = 4; break;
        case 0 : assert(esc);
                 i = 5; break;
        default: return ANY_STATE; /* environmental constraint  */
    }
    NEW->dc += (rcv) ? i : -i;  /* keep track of DC-threshold   */
    if (( rcv && OLD->dc >  DCT) ||
        (!rcv && OLD->dc < -DCT)   ) {
        if (i != 1) {
            assert(error);     /* fill transition was expected */
            NEW->dc = 0;
        }
        return D_TRNS;
    }
    if (esc) {                 /* data part of an esc-sequence */
        if ((i - 1) >= RESET) {
            data = RESET;      /* perform a reset               */
            deassert(error);
            NEW->dc = 0;
        } else
            data = (i - 1);
        assert(cntl_valid);    /* cntl reception completed      */
    } else if (!error) {
        data = (i - 1);
        assert(data_valid);    /* data reception completed      */
    }
    if (error) NEW->dc = 0;    /* remain in error state         */
    return D_TRNS;
}
```

Figure 3-13: Decoder Finite State Machine

The encoder FSM is described in Figure 3-14. This circuit compiles in one conventional, clocked PLA. Again, the apparent complexity of the product machine is somewhat misleading because it decomposes into a plain toggle for the output signal *xmt*, a pulse width counter *pwc*, and a DC-threshold counter *dc*.

Intra-cluster channels are also used to synchronize the local clocks of each cluster. In order to use circuit 3-2, a recovered bit clock is desirable. This clock does not need to maintain precise phase lock to the data because it is not used for data separation. Therefore a phase locking oscillator is sufficient.

Encoder/Decoder Characteristics			
DC Threshold	Encoder (states)	Decoder (states)	Fill Loss (%)
1	418	151	5.96
2	518	183	4.47
3	618	215	3.58
4	718	247	2.97
5	818	279	2.54
6	918	311	2.22
∞	50	21	0

Table 3-3: Interface Circuitry Tradeoffs

```
#define DCT     5               /* DC threshold                */
#define RESET   3               /* Channel reset               */
%input  data(d1, d0), cntl;
%output xmt     = deasserted,
        busy    = deasserted;
%internal
        esc     = deasserted;   /* xmitting an esc-sequence    */
        int dc  = 0;            /* accumulate dc-component     */
        int pwc = 0;            /* pulse width counter         */
        int esc_data = 0;
%% {
    if (xmt) NEW->dc++;         /* keep track of DC threshold  */
    else     NEW->dc--;
    if (OLD->pwc > 0) {         /* see if current pulse is done */
        NEW->pwc = OLD->pwc - 1;
        return D_TRNS;
    }
    change(xmt);
    if (( xmt && NEW->dc < -DCT) ||
        (!xmt && NEW->dc >  DCT)   ) {
        return D_TRNS;          /* insert a fill-transition    */
    }
    deassert(busy);
    if (esc) {                  /* xmt data part of esc-sequence*/
        deassert(esc);
        NEW->pwc = OLD->esc_data;
        if (OLD->esc_data == RESET) {   /* reset: zero-DC count */
            if (!xmt) NEW->dc = -4;
            else      NEW->dc =  4;
        }
        NEW->esc_data = 0;
    } else {
        if (cntl) {             /* initiate esc-sequence       */
            assert(esc);
            NEW->pwc = 4;
            NEW->esc_data = data;
        } else                  /* send data                   */
            NEW->pwc = data;
    }
    if (NEW->pwc) assert(busy);
    return D_TRNS;
}
```

Figure 3-14: Encoder Finite State Machine

Figure 3-15 describes such an oscillator based on a delay line (ring oscillator). Circuits of this type are common in serial communication chips, for example HDLC controllers, and are

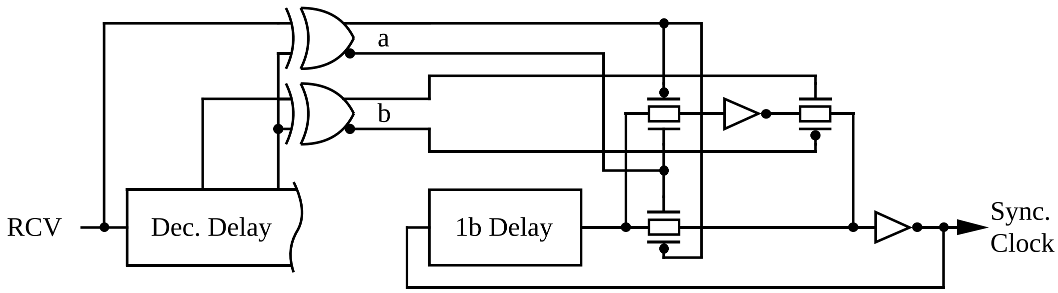


Figure 3-15: Phase Locking Ring Oscillator

occasionally mis-labeled as digital PLL's. However, there is no loop involved at all. The oscillator is set to the proper phase whenever a phase determining transition is available and runs happily at its own frequency otherwise. Due to the delay-line stabilization, the oscillator frequency is within a few percent of the system clock, so the circuit is really only inserting the missing transitions.

3.4.2.2. Transmission Protocol

Intra cluster channels are used in matched pairs running in opposing directions. Both channels operate independently and use the reverse channel for acknowledgments. Four escape sequences are defined:

1. Acknowledge successful reception of a packet.
2. Signal a data error in the received packet. Packets have a checksum postfix that is added at the sending site and removed at the receiver.
3. Signal synchronization loss. This condition can be detected after each fill transition and at the end of a packet transmission (postfix structure).
4. Signal a channel reset. The transmitting end of the channel may issue a reset at any time. Partially transmitted packets will be discarded and the receiver starts anew from the initial state.

The first three escape symbols are used by the other channel and are inserted into the current transmission at any time without disrupting the packet in transit. The last signal is used to initialize a channel and to recover from a lost synchronization.

Both transmitter and receiver are double buffered to avoid wasting time to wait for the acknowledgment. Since the wire delay is less than the transmission time, one of the three possible responses to a transmission will arrive before the 2nd transmission is completed. Otherwise, the channel is considered inoperable. A positive acknowledge will free the associated buffer. A data error will cause a retransmission of the packet in question. A synchronization-loss indication will cause a channel reset, followed by retransmission attempts of both packets.

Inter-cluster channels do not originate directly from a processing node, rather they are connected to an intra-cluster channel through an interface chip described later. Since this interface chip is only extending existing channels, it does not need to deal with virtual channels. That information is simply passed uninterpreted.

3.5. In Search of a Routing Heuristic

Initially, message-passing systems used a deterministic routing method: a given pair of source and destination addresses defined the path of the message through the network. For example, the E^3-routing [142] implies an order of the dimensions in which the source and destination address differ. Routing of messages through the network proceeds according to this order. This removes the ambiguity of the message path in networks that have multiple shortest paths between nodes.

The advantage of a deterministic routing scheme is its simplicity. The routing decision is easily implemented with a few gates that compare the source and destination address. Deadlock avoidance is easily accomplished by resource ordering [55].

The downside of deterministic routing is its performance. As network load increases so does the probability of resource conflicts. If two packets are contending for the same channel, one has to wait even if there is an alternate path of equal length. E^3 routing makes no use of alternative paths. Besides providing more flexible resource utilization under high network loads, adaptive routing may also use multiple paths between two nodes to increase the bandwidth by distributing the traffic over all available channels.

Adaptive network control is common for long haul networks, to optimize use of the expensive channels. Unfortunately, these techniques and algorithms are not directly applicable because of the amount of computation needed for one routing decision. Long haul networks feature relatively slow channels and long messages, so a router can spend considerable time to collect traffic statistics, exchange that information with other routers, and compute optimized routes through the network. The router of a message-passing system should fit in one chip along with channel interfaces, buffer memory and an interface to the node CPU. Even a high-performance CPU can execute only a few instructions during a typical packet transmission time (and would not fit on the router chip anyway). Traffic patterns of message-passing systems are short-lived and coordination among routers is either too slow or requires too much extra bandwidth. As a consequence, adaptive routing algorithms that were proposed for message-passing systems are much simpler than their long-haul counterparts.

3.5.1. Applicability of Adaptive Routing

Adaptive routing is primarily associated with path selection for the active packets through a network. In order to benefit from adaptive routing, networks should have multiple (optimal) paths between nodes. For example a K-shuffle network has exactly one shortest path for each source/destination pair. This leaves little freedom for routing decisions. Likewise, fat-tree networks and centralized routers will receive no benefit from adaptive transmission path selection.

The classes of k-ary cubes, cube-connected cycles, and random networks have sufficient alternative paths to pursue adaptive routing. For the purpose of investigating routing strategies, binary hypercubes are used. This is mainly due to their regular structure and ease of analysis. Since the strategies do not make use of specific properties of binary hypercubes, the results are expected to apply to other networks.

Besides path planning, there is a second component to adaptive routing for packet switched networks: packet reordering. Circuit-switched networks do not allow this freedom: once a path is established, all (virtual) resources are committed. Packet switched networks on the other hand can choose between storing a packet temporarily and sending it. For example, consider the simple case shown in Figure 3-16. At the front of both input queues are packets

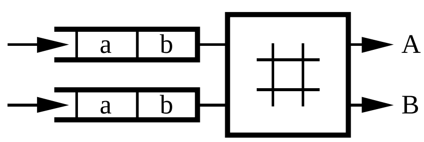


Figure 3-16: Adaptive Routing via Packet Reordering

addressed for outbound channel *B*. Without the ability to reorder the transmission sequence of packets, the router has no choice but to waste one transmission slot on channel *A*. Given the ability to reorder the sequence of transmissions, the sequence of packets in either queue may be reversed to insure optimal utilization of the outbound channels.

The decision to reorder can be made locally in each intermediate node, independent of the other nodes in the transmission path. An example would be to give priority to packets that are *almost there*. Packet reordering is available to all network topologies.

3.5.2. The I/O Model for Adaptive Routing

The input to an adaptive router is information about the state of the network. Based on this information, the router controls the disposition of packets within its domain. For the scope of this thesis, the domain of the router is one node of the network and the router is the controlling agent of all message system resources (channels, buffers, data paths, address tables, decode logic, etc.).

3.5.2.1. Input: Information Sources

Any adaptive routing heuristic needs certain information to decide how to deal with a particular packet. Basic information available to any router usually consists of:

- *Destination addresses* of all pending packets.
- The location of the router in the network.
- *Source addresses* for all pending packets. Packet header may omit the source address to save bandwidth in some systems. The system under consideration here will always require a source address.
- Static network information, such as routing tables. Any router must be able to compute which channel can be used to forward a given packet. For highly regular networks, this may be as simple as an XOR of the destination address and the router's address. A more flexible system will require tables for this purpose.
- The current status of the communication resources. This includes which channels are busy, how many buffers are available (if any), and the status of the interface to the local node.

Unlike this basic information which is required by any router, optional information sources may be added specifically to aid adaptive routing:

- The distances to the source and/or destination for all pending packets. This could be tagged onto the routing tables.
- The age of a packet. Adding a time stamp to each packet is quite expensive and is probably not justifiable solely by the expected gain in routing performance. However, packets may be time stamped for other reasons.
- Dynamic network information. Routers may monitor the traffic flow and exchange that information periodically. Again, this is quite expensive in terms of extra hardware and logic.

This survey of input information to routing heuristics is quite biased. In particular, centralized methods are not considered. Having a network control center that monitors traffic and sends directives to all routers is considered too slow. This will become more obvious once the intended mode of operation is described in Chapter 4.

Application program control of routing strategies is dismissed on similar grounds. Routing will become a very low-level task that is bound by tight timing constraints.

Finally, while [75] reports gains from dynamic network information, these methods will not be pursued further here due to their implementation cost and complexity. The claimed advantage was obtained from a system (iPSC/1) with much slower channels and larger message sizes and is not directly applicable.

3.5.2.2. Output: Switch Control

The output of the (adaptive) routing algorithm is a set of control signals to move packets through the switch. There are basically four ways to deal with a packet:

1. Ignore it. The packet remains in the receiver and the associated channel will be unusable (blocked) until the packet is removed from the receiver. The precise semantics of *ignoring* a packet depend on the implementation of the channel. Channels that lack flow control (handshaking) or have insufficient buffer space may be forced to drop a packet. As pointed out earlier, the channels considered here can block and do not drop packets.
2. Buffer it. The packet is copied to a transient storage area. The receiving channel is freed and may proceed to receive another packet. Buffer space is limited and the router is supposed to ensure that no packets are lost or duplicated on their way in and out of the transient buffer.
3. Forward it. The packet is transferred to an outbound channel that will bring it closer to its destination. Consuming terminal packets locally should be considered as a special case of forwarding.
4. Send it elsewhere. This is a generally undesirable option. "*Blind transmissions*" of packets along a path that increases the distance to the destination wastes bandwidth and increases latency. There are, however, reasons in favor of this option. Examples are recovery from broken channels and avoidance of congested parts of the network. Pathological traffic patterns can be defused by sending packets to a random intermediate node [86, 149].

Routing decisions are constrained by global network assertions. The network must be deadlock-free. Other properties such as fairness are harder to define and ensure.

3.5.3. Adaptive Routing: Prior Art

Most work on adaptive routing has been directed at telephone switches or long-haul, packet-switched networks (ARPANET, TELENET, TYMNET, etc.). Generally, adaptive routing methods are superior only if the network is subjected to highly variable traffic patterns [27, 19]. Long-haul networks are characterized by small numbers of nodes sparsely connected with costly, high-latency, low-bandwidth channels that are faced with relatively long packets. Given these time constraints, a considerable amount of CPU power is available for routing decisions. Subsequently, the proposed approaches concentrated on numerical methods to solve the underlying nonlinear multicommodity flow problem [16]. Typically, such methods have time complexities of $O(N^2)$ to $O(N^3)$ per iteration [23].

A taxonomy of routing algorithms was proposed by Rudin [120] that uses three dimensions: the *location* of routing decisions, the *information* source for routing decisions, and the *frequency* of routing decisions. Since the proposed network does not have a central controller and because the communication time to a central controller is high compared to the controlled object's rate of change, only decentralized methods are considered. Likewise, the only practical source of information is the state of the local node. The frequency of routing decisions is quite high.

Adaptive routers that exclusively use local information may become unstable in the sense that some packets may bounce back and forth because the routing tables changed during packet transmissions. This concern is valid for long haul adaptive routing due to a difference in perception of what is *adaptive*. A long haul system is usually not confined to shortest path routes and channels may differ widely in their delay, bandwidth, cost, and error rates. On the other hand, routers for a multiprocessor communication system deal with a more homogeneous collection of faster channels. The delay incurred by intermediate nodes is dominating such that routers are largely confined to shortest paths. This is similar to *delta routing* [120]: A central *network routing control center* (NRCC) distributes the globally optimized routing tables. These tables list a number of alternatives and local nodes decide which one to use based on local information.

For the system under consideration, the NRCC performs a static analysis of the network topology and distributes the tables of shortest path(s). Since the network topology won't change dynamically, the NRCC becomes obsolete once the system is started.

But even delta routing related techniques such as *hybrid, weighted routing* [118] with static global information are still too costly due to the need to accumulate local traffic statistics.

Some message-passing systems leave routing decisions to the software [22, 89, 57] and have run experiments with adaptive routing algorithms. However, these have not yet found their way into operational systems. Furthermore, any software implementation is too slow for the proposed message system: about 8 routing decisions in less than 0.5 µs are required.

The currently proposed adaptive routing methods for high performance message-passing systems are based on a greedy search for resources. For example, Dally generalizes E^3 routing for the torus network by considering all shortest distance paths [36]. In general, most work directed at adaptive routing is really aimed at deadlock avoidance methods that allow more unrestricted resouce allocation. The underlying rationale is that greedy resource allocation will result in adaptive routing once ways are found to use channels and buffers freely.

The hyperswitch system [28, 54] operates on binary hypercubes and uses a restricted form of backtracking to search for a connection (the hyperswitch system is intended to use primarily circuit switching). Backtracking is restricted to one or two levels up from the leaf of the search tree. If a connection request fails after the backtracking has exhausted its options, the source node may initiate other attempts over different routes.

The adaptive part of the routing method is trying to establish one connection at a time. If there are multiple ways to complete that particular connection, an arbitrary selection is made by the order of the search (first one is taken) without regard to the potential impact on other connections.

3.5.4. Basic Heuristic

The adaptive routing heuristic will be developed initially on a binary hypercube network. Each node of the network is uniquely addressed by a number from 0 to $N=2^n-1$, where n is the order of the cube. The message system accepts a message from an interface to the rest of the node circuitry (processor, memory, etc.) and delivers it to its destination node. A rudimentary routing algorithm is described in Figure 3-17.

```
for (j = 0; j < n; j++) {       /* scan all channels */

    if (R[j].status == FULL)    /* get a packet      */
        packet = R[j].buffer;
    else if (SOURCE.status == FULL)
        packet = SOURCE.buffer;
    else
        continue;

                                /* consume packets   */
    if  ((packet.dst_addr == NODE_ID) &&
         (DRAIN.status    == EMPTY) {
        DRAIN.buffer = packet;
        continue;
    }

    for (i = 0; i < n; i++)     /* try to send it    */
        if ((T[i].status == EMPTY) &&
            ((1 << i) & ROUTE[packet.dst_addr])) {
            T[i].buffer = packet;
            break;
        }

    if (i < N)
        continue;               /* Done!             */

    for (i = 0; i < n; i++)     /* send it blindly   */
        if (T[i].status == EMPTY) {
            T[i].buffer = packet;
            break;
        }
}
```

Figure 3-17: Routing *Scheme A*

The first loop scans all receivers *R[j]* for the receipt of packets during the last transmission cycle. The presence of a packet in the receiver buffer is indicated by a *FULL* status. If there is a packet, it will be moved to temporary storage (*packet*), otherwise a new packet may be loaded from the local node. If the address of the packet (*packet.dst_addr*) is equal to the address of this node, a local delivery attempt is made.

The second loop looks for idle outbound channels (transmitters, *T[i]*) that are along an optimal path towards the destination address of the packet. The routing table has a bitvector for each potential destination address such that a '1' in the i^{th} position identifies a suitable transmitter. In a binary hypercube, this bitvector is simply the exclusive-or between the binary representations of the node- and destination-address. A plain *XOR* doesn't require a routing table, but arbitrary network topologies have more complex paths. Certain classes of networks (mainly planar topologies) have regular routing functions that can be implemented with combinational logic and/or small tables [48]. Since routing tables are the most general approach and are within technological limits, a table-driven router is proposed.

If there is no suitable transmitter, the third loop will send the packet to a more distant node (*blind* transmission). In a binary hypercube, each of these blind transmissions causes a total of two extra packet transmissions. The waste of bandwidth in other networks can be significantly higher, for example high-ary cubes with one-directional channels need arity-times extra hops. In general, networks composed of bidirectional channels are more forgiving with respect to blind transmissions.

For the analysis of this routing scheme and others described below, an infinite supply of packets at each node is assumed. Likewise, all nodes are assumed to be ready to remove terminal packets from the network. This is essentially the saturated network case. All channels will be busy sending packets continuously. The performance measure is reduced to the question of how many unnecessary transmissions take place. Optimizing this performance measure is likely to result in heuristics that perform well under high network loads and maximize the use of the available bandwidth.

To analyze the behavior of this simple routing strategy (*scheme A*), the nodes are assumed to provide randomly addressed packets. The probability $P_S(i)$ that a new packet is addressed to a node *i* transmission steps away depends on the distribution of destinations. If all nodes are equally probable, $P_S(i)$ is a binomial distribution:

$$P_S(i) = \binom{n}{i} / 2^n \tag{3.9}$$

If there are a large number of packets and if the system is running for a while, it is reasonable to assume that the packet flow in the system will exhibit steady statistical behavior. To analyze this situation, the node can be viewed as a graph of packet flows:

Each packet flow can be characterized by distance distribution $P_{S,D,R,T}(i)$ for *Source,*

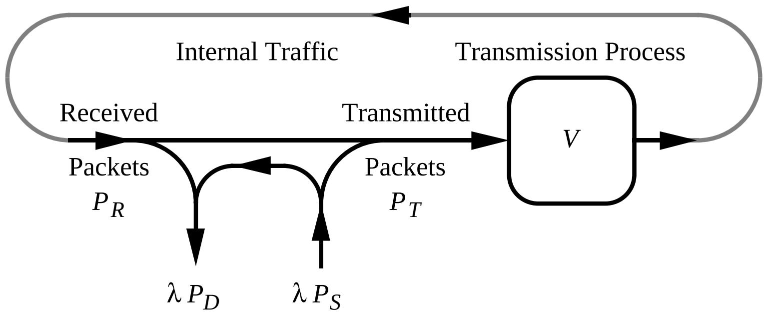


Figure 3-18: Packet Flows

Drain, Received, and *Transmitted* packets. λ is the packet rate to and from one node in terms of packet- transmission steps. Based on the assumption that the packet system does not produce or lose packets (see Figure 3-18), the following relations hold:

$$\forall\, i, 0 \le i \le n : P_R(i) + \lambda \cdot P_S(i) = P_T(i) + \lambda \cdot P_D(i) \tag{3.10}$$

$$\sum_{i=0}^{i=n} P(i) = 1$$

$$P_T(0) = 0$$

$$P_D(i) = \begin{cases} 1 & \text{if } i=0 \\ 0 & \text{otherwise} \end{cases}$$

The transmission process V will change the distance distribution of the outgoing stream to the incoming distance distribution. In the ideal case, where all packets are transmitted towards their destination, $P_R(i-1)=P_T(i)$ for $0 < i \le n$ would be true, since any transmission reduces the distance of a packet to is destination by 1. However, in the real system the probability of successful transmissions, V, is less than 1. For the routing scheme A, V depends on the packet distance. To compute V, we look at the probability $Q(i,j)$ that a packet to a node that is i dimensions away matches a transmitter in the j^{th} assignment step.

$$Q(i,j) = \begin{cases} 1 & \text{if } i \ge j \\ \left(\binom{n}{i} - \binom{j-1}{i}\right) / \binom{n}{i} & \text{otherwise} \end{cases} \tag{3.11}$$

Based on this relation, it is possible to compute the distributions for all links of a given system. However, this involves the solution of a linear equation system with $n^2 \cdot 2^n$ variables. Each link requires a separate set of variables because of the asymmetry introduced into the system by the fixed sequence used to test the receiver for packets. To restore symmetry the outer *for*-loop in the routing scheme A is modified to a loop that uses a random permutation of the numbers 1 to n each time it is executed. Therefore all links will have the same distribution. Since the scheme A handles packets one-by-one without any interaction between packets other than reducing the number of idle transmitters (which does not depend on the packet) V is a function of i:

$$V(i) = \sum_{j=1}^{n} Q(i,j)/n \tag{3.12}$$

This leads to the linear equation system:

$$
\begin{aligned}
&P_T(1) - V(2)\cdot P_T(2) = \lambda\cdot P_S(1) \\
&P_T(j) - V(j{+}1)\cdot P_T(j{+}1) - (1{-}V(j{-}1))\cdot P_T(j{-}1) = \lambda\cdot P_S(j) \text{ for } 1 < j < n \\
&P_T(n) - (1{-}V(n{-}1))\cdot P_T(2) = \lambda\cdot P_S(n)
\end{aligned} \tag{3.13}
$$

λ can be computed by normalization of the result. Figure 3-19 shows the mean number of transmission steps (which is $1/\lambda$) for various cube dimensions along with the optimal case values (V=1). It is interesting to observe that the loss in bandwidth due to blind transmission for the simple assignment scheme is less than linear.

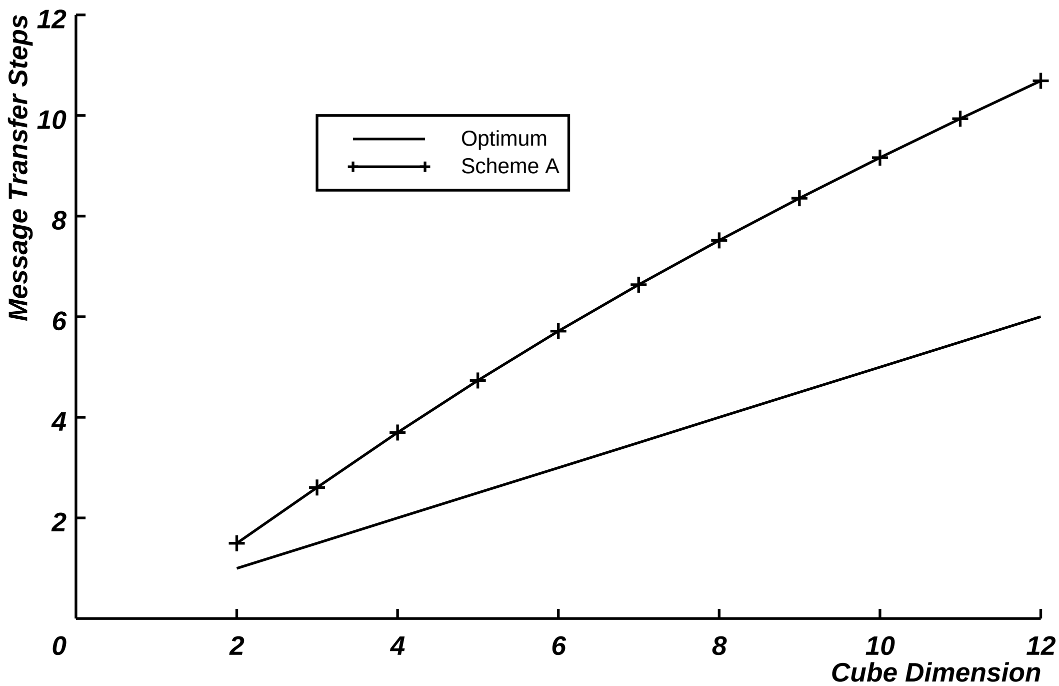


Figure 3-19: Routing *Scheme A* Performance

The distributions for a 10-cube are shown in Figure 3-20. These distributions correspond to data collected from a simulation of the packet system. The randomization of the assignment loop degrades performance by about 10% but has only minor influence on the distributions.

Figure 3-20 shows that the distance distribution for traffic within the net is significantly different from the distribution of the inserted packets. Packets within the net are typically *closer* to their destination and the most likely distance that a packet has left to go is two hops. Other network topologies and/or more realistic traffic patterns show a similar dominance in the distance distributions, which strongly suggests that an adaptive routing heuristic should expect a lot of short distance traffic and should be optimized for that case.

Unfortunately, short distance packets have fewer routing options. Therefore it should be advantageous to route packets according to their distance by giving priority to packets with

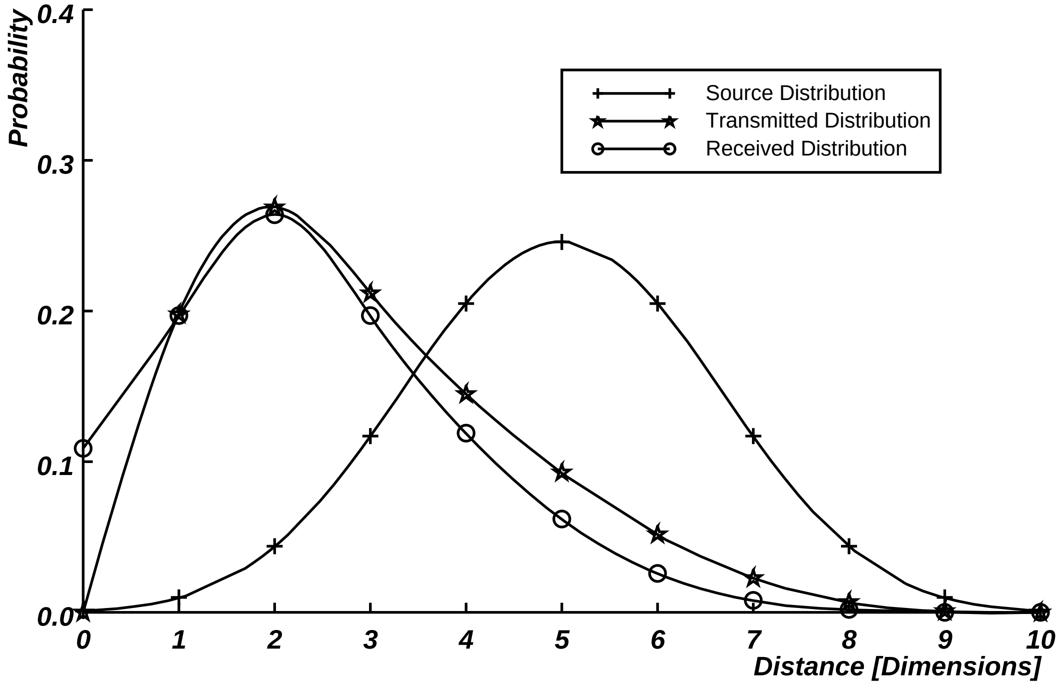


Figure 3-20: Packet Distributions for a Binary 10-Cube

the smallest distance. This leads to routing *scheme B*. A simplified description is given in Figure 3-21.

```
for (i = j = 0; i < n; i++)     /* collect packets   */
    if (R[i].status == FULL)
        packets[j++] = R[j].buffer;

....    /* deliver terminal packets              */

....    /* fill free slots with new packets      */

dst_sort(packets, j);          /* sort by distance  */

for (i = 0; i < j; i++) {      /* send packets      */

    ....    /* attempt to forward packets[i]     */

    ....    /* failed: send packets[i] somewhere */
}
```

Figure 3-21: Routing *Scheme B*

All incoming packets are collected in the array *packets[]*. After removing terminal packets and filling free slots with new packets, the collection of active packets in *packets[]* is sorted by their distance towards their destinations. Packets with the fewest hops to go are placed first. In the case of the binary hypercube, the distance to go is simply $card(packet[].dst\_addr \text{ xor } NODE\_ID)$. Topology independent routers will have to store this information in the routing table.

The analysis of this scheme is more complex because V becomes a function of P_T. To compute V the combined promotion chances of all packet compositions weighted by their probabilities (given by P_T) must be averaged. This amounts to a sum of $\binom{2n-1}{n}$ terms. For n=9 there are 24,310 items. Since V has become a rather complex function, the resulting equation system must be solved iteratively. V must be recomputed in each step. Therefore the computation time needed to solve the analytical model is comparable to a simulation approach. The situation becomes even worse with more sophisticated assignment schemes. Consequently, the discussion of these schemes is based entirely on simulations.

To get an idea of the best possible performance of any routing heuristic that *only* optimizes the number of forwarded packets, a simulation is run that uses an $O(n^3)$ algorithm [83] to find the optimal assignment. This is a special case of the linear assignment or bipartite cardinality matching problem.

Experiments with the optimal linear assignment algorithm and a number of other assignment heuristics resulted in a more effective strategy. The linear assignment algorithm finds the optimal match of resources, but it does so without regard to the underlying routing problem. While attempting to find a less complex substitute, *scheme C* was developed. Overall, about 2% better results were obtained. The failure to always find the optimal match is compensated by accounting for other factors, such as the packet priority. Figure 3-22 depicts routing scheme c. Again, all packets are first collected in a temporary storage area. In actual implementations, the assignment process will proceed in parallel with the reception of data. After taking care of terminal packets, a counter for each transmitter is set to the number of pending packets that may be routed along that route. The sequence of packets to transmitters assignments is subsequently ordered such that the transmitter with the fewest pending packets is committed to a packet first.

Assignments proceed until all transmitters with at least one matching packet are committed. If there remain idle transmitters and active packets, bind assignments are used. In actual implementations, ordering the sequence of assignments is done with a priority circuit and does not require sorting.

Each assignment is controlled by the packet priority. If there is more than one packet pending for a given transmitter, the one with the highest priority is preferred. Again, priority is a function of the destination address: packets close to their destination are given preference.

```
for (i = 0, j = 0; i < n; i++)  /* collect packets    */
    if (R[i].status == FULL)
        packets[j++] = R[j].buffer;

....    /* deliver terminal packets                   */

....    /* fill free slots with new packets           */

for (i = 0; i < n; i++)
    Tx_seq[i].cnt = 0, Tx_seq[i].id = i;

for (i = 0; i < j; i++) /* count packets per Xmitter  */
    for (k = 0; k < n; k++)
        if ((1 << k) & ROUTE[packet[i].dst_addr])
            Tx_seq[k].cnt += 1;

Tx_sort(Tx_seq); /* sort Tx_seq: fewest packets first */

for (i = 0; i < n; i++) {       /* send packets       */

    k = Tx_seq[i].id;   /* deal with most constrained
                           Xmitter first              */

    ....    /* Find packet with highest priority for
               T[k] and send it                       */

    ....    /* update Tx_seq[i+1,...,n-1] and re-sort */
}
```

Figure 3-22: Routing *Scheme C*

Observing packet priorities compensates for the loss due to the few cases in which this algorithm fails to find the optimum assignment. A linear assignment algorithm with a weight function including the packet distance might yield an even better performance. However, the extra complexity is not justifiable by the small potential gain.

3.5.4.1. Variations of the Basic Heuristic

One of the "*inventions*" of scheme C is the attempt to assign matching packets first and defer blind transmissions until no more matches can be made. This idea can be applied to schemes A and B with minor costs in complexity. The modified schemes will be named D and E respectively.

The impact of giving priority to through traffic packets in terms of the expected number of transmission steps per packet (*MTS*) or total throughput is negligible. However (as expected), the standard deviation of the MTS is decreased, which is a measure of the variation of the packet latency. Also the capability of the system to handle transmission errors causing retransmissions is increased.

3.5.4.2. The Impact of Extra Buffer Space

Schemes C, D, and E can easily incorporate a temporary buffer in each node by storing some packets to skip a routing cycle which is less costly than a blind transmission. Preferably, packets not matching an idle transmitter are stored in a small, fixed amount of memory. These packets are retrieved in the next cycle and are handled like newly received packets. They might be assigned a higher priority to reduce the latency variance.

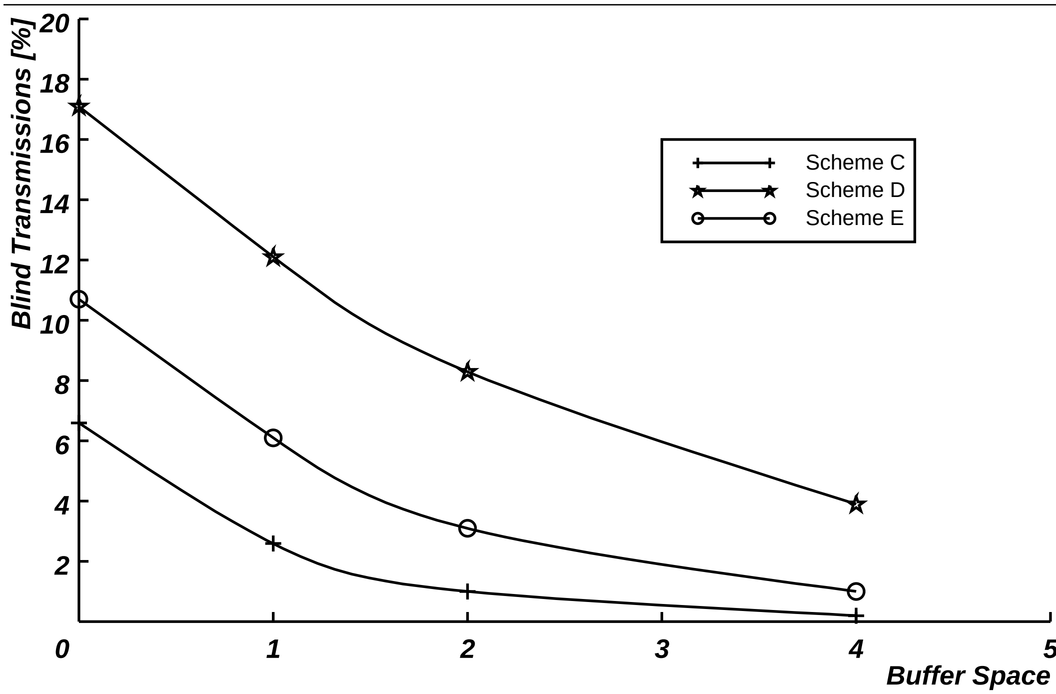


Figure 3-23: Blind Transmissions vs. Buffer Space

As expected, buffer space has a large impact on the performance. The number of blind transmissions is about halved for each additional slot of temporary storage. Therefore a very small buffer (for example 4 slots per node) allows close to optimal performance. Figure 3-23 shows the percentage of blind transmissions versus buffer space. The average number of transmission steps for one packet is plotted in Figure 3-24.

3.5.4.3. Basic Heuristics: Summary

The winning routing heuristic is *scheme C* combined with a modest amount of buffer space. Under saturated, uniform load conditions, the performance is within a fraction of a percent of the theoretical optimum. Subsequent evaluations with other traffic patterns will be confined to this method.

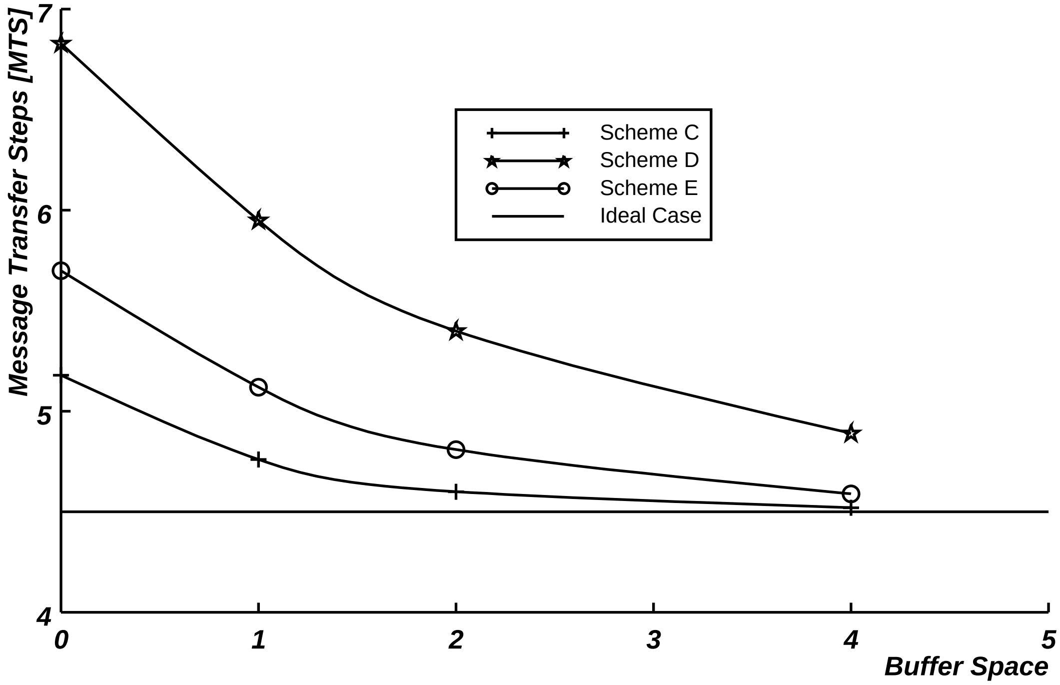


Figure 3-24: Packet Transmission Steps vs. Buffer Space

Routing Heuristics on a Binary 10-Cube								
	% Blind Transmissions vs. # of Buffers				Message Transfer Steps vs. # of Buffers			
Scheme	0	1	2	4	0	1	2	4
A	19.6	-	-	-	7.40	-	-	-
B	12.3	-	-	-	5.97	-	-	-
C	6.6	2.6	1.0	0.2	5.18	4.76	4.60	4.52
D	17.1	12.1	8.3	3.9	6.83	5.95	5.40	4.89
E	10.7	6.1	3.1	1.0	5.70	5.12	4.81	4.59
Ideal	0.0	-	-	-	4.50	-	-	-

Table 3-4: Heuristics Performance Overview

3.5.5. A Different Perspective: the Toy Cube

Another approach to the design of a routing heuristic is modeling the entire system - routers and communication channels - as one GSPN. The routers in this model are based on register files. Each channel is represented by one place that has one token if that channel is idle. These places are connected to the inputs of all packet paths that use a particular channel.

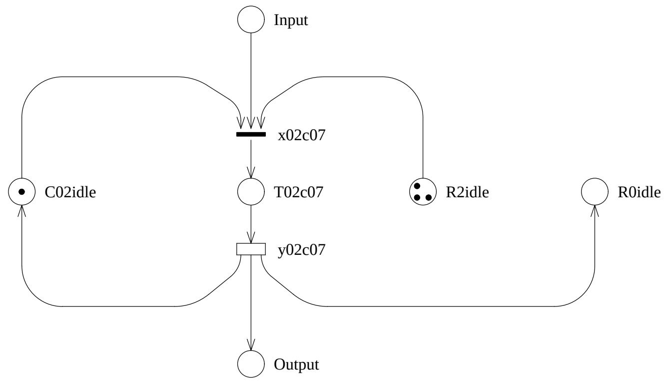


Figure 3-25: Partial Channel Subnet for Cube GSPN

The place *C02idle* in Figure 3-25 represents the channel from node 0 to node 1. This place is connected to the transition *x02c07*, which is part of a path from node 0 to node 7. Packets are inserted at each node. At packet insertion time, a destination node is chosen. This models uniformly distributed addressing. Packets retain their identity by being bound to a particular set of paths through the network. For example, a packet from node 0 to node 7 may choose node 1, 2, or 4 as the first hop. In the example, node 2 was selected and the path may proceed through either node 3 or node 6. For one packet transmission to take place, three conditions must hold:

1. The channel is idle.
2. The destination node has a register available to hold the packet.
3. A packet is pending for this path

The third condition implies that the packet occupies a register in its current place. Competing transitions and priorities are reflected in the corresponding transition rates. Once *x02c07* is fired, the transmission delay is determined by the timed transition *y02c07*.

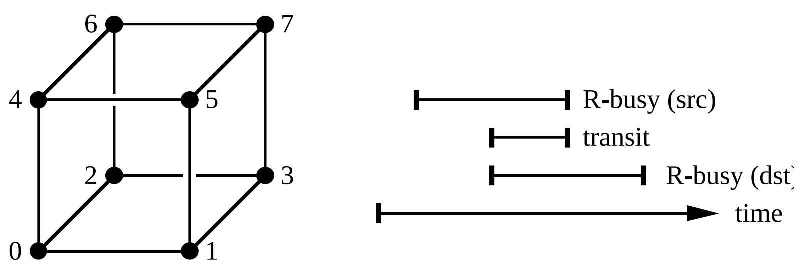


Figure 3-26: The Toy Cube

While a packet is in transit, it will occupy resources in both the source and destination nodes. Packets are removed at their destination. The simple cube of Figure 3-26 translates into a Petri Net with 432 places and 528 transitions. This is a bit on the high side of complexity for GSPN's. In fact *GreatSPN* is not capable of analyzing such networks; the accelerator described in the appendix was used for the analysis.

The heuristic being investigated with the toy cube is a probabilistic priority ordering based on the packet position in the net and its source and destination addresses. There are 6 distinct packet classes and each is given a certain priority. Priorities were determined by varying the priority and observing the impact on the saturation throughput of the entire network. After a crude gradient search of maximum throughput, a 6-dimensional quadratic polynomial was fitted through the data points generated by the gradient search. Rounding the priorities to convenient integer values yielded a set of priorities that could not be improved further. Since the optimum is rather shallow, a 10% change in the priorities is insignificant.

Total Distance (hops)	Distance to go (hops)	Relative Priority
1	1	1024
2	2	32
2	1	1024
3	3	1
3	2	32
3	1	1024

Table 3-5: Relative Packet Priorities

The optimal priorities do not depend on the total distance a packet has to traverse, rather they depend solely on the remaining distance that a packet has yet to go. Furthermore, these priorities are insensitive to the number of extra buffers that each router may use. The priorities given in Table 3-5 also confirm the earlier observation that the packet priority should increase as the remaining distance decreases.

The impact of temporary buffer space is shown in Figure 3-27. Naturally, the throughput increases as additional buffer space boosts the ability to reorder packets. This extra bandwidth is achieved at the expense of a latency increase for long distance traffic. The network is able to store more active packets and the probability increases that a packet will wait temporarily in a buffer.

It should be noted that the effect of reordering improves the throughput compared to a centralized router. The data of Figure 3-27 is directly comparable to the performance of a synchronous 8 by 8 router operating on fixed-length packets (Table 2-2).

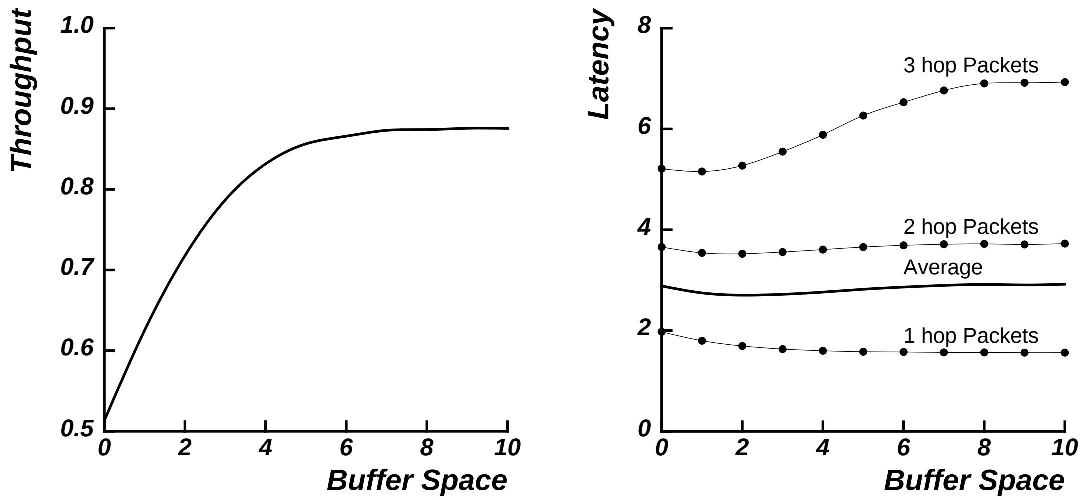


Figure 3-27: Latency / Throughput vs. Buffer Space

3.6. Routing Tables and Deadlock Avoidance

This section describes the deadlock problem and the principal method used to avoid deadlocks. Virtual channels are introduced as a facility to resolve the deadlock problem for a wide range of network topologies. The generation of routing tables is included in this section because it is closely related to the resolution of deadlocks.

Instead of defining the construction of routing tables for specific network topologies, a general construction algorithm is provided. This algorithm is applicable to the regular network families (k-ary hypercubes, cube-connected cycles, k-Shuffles) as well as to random networks.

3.6.1. About Deadlocks

Any active packet in the network binds network resources, namely the buffer used to store the packet. Successfully transmitting a packet requires an additional buffer on the receiving node. Since each node has only a fixed number of buffers, networks may deadlock. The smallest network with deadlock potential is a three node unidirectional ring: no progress can be made once each node's buffers are filled with packets addressed to nodes two hops ahead.

Besides this *Store-and-Forward Deadlock*, there are several other undesirable network problems that are related, such as *Statistical Blocking* and *Consumer Caused Deadlock*. In a statistically blocked network, the traffic of one or more nodes conspires to prevent some packets from being delivered. This is a dynamic situation that would be resolved when the network load drops, but it can cause a high latency variance. A consumer-caused deadlock arises if some node of the networks fails to remove its traffic. This can block traffic addressed to other nodes.

This section addresses the construction of routing functions for arbitrary networks that are free of store-and-forward deadlocks. Statistical blocking and consumer caused deadlocks will be dealt with in Chapter 4.

The notion of deadlock freedom rests on the assumption that packets arriving at their destination will eventually leave the network. The principal method used is based on resource ordering [55]. Only the buffers associated with the sending and receiving ends of a channel are considered. Routers may have additional buffers for temporary storage of packets, but these are not restricted to any particular packet passing through a network node and therefore are irrelevant to store-and-forward deadlocks. Additional transient buffer space reduces the probability of deadlocks, but it cannot prevent it.

There are a number of different approaches to solve the deadlock problem:

- A deadlock detection mechanism could be used if the potential for deadlocks is low. In the infrequent case that a deadlock actually occurs, a recovery procedure is initiated that resolves the problem. Error recovery may employ direct processor intervention to temporarily remove packets from the network.
- Store and forward deadlocks can be avoided in a non-blocking network. Such a network will always move packets at the cost of deviating from optimal routes.
- Store and forward deadlocks can be avoided by preventing cyclic resource conflicts.

Networks guarded by a deadlock detection mechanism allow a deadlock to occur and are prepared to recover from it. The probability of a deadlock can be reduced substantially by increasing the fan-out of the network, adding buffer space and using a topology with ample alternate paths. Deadlocks can be detected by time stamps associated with each packet: packets of a certain age are considered deadlocked and receive special treatment. The complexity eliminated by allowing deadlocks comes at the expense of potentially costly error recovery procedures. While the deadlock probability can be made quite small for typical usage, there is the danger of pathological traffic patterns.

Networks can be made intrinsically deadlock-free by guaranteeing consumption of arriving packets at each node. Each node attempts to route all packets to favorable channels and uses blind transmissions as a last resort. This approach transforms the deadlock problem into a livelock problem. Progress can be assured by using blind transmissions in a systematic fashion. For example, the topology could be collapsed into a ring for all blind traffic. This approach was used in the HEP multiprocessor [134] and was recently generalized for arbitrary topologies [100]. The disadvantage of this method is a severe degradation of the network bandwidth for pathological traffic patterns. Networks of this nature are prone to hot-spot effects.

The method of choice is the prevention of cyclic resource conflicts. In order to simplify the analysis, the transmit and receive buffers of one channel are coalesced into one entity that is uniquely associated with that channel. The deadlock problem is solved if a partial order on these channels (buffers) can be found such that all packets are routed through the network traversing channels with strictly descending order. The progress of any packet depends on the availability of a channel ranked lower then the one currently holding the packet. Since the partial order encompasses all channels throughout the network, no cycle of resource dependencies is possible.

3.6.2. Virtual Channels

Given an arbitrary network, chances are that no practical channel order exists that permits sensible routing. Obviously, it is not possible to order the three channels of the three node directional ring *and* allow all nodes to send packets to any other node. This problem can be solved by adding more channels to the system. However, channels are expensive; hence it is preferable to just increase the number of receive and transmit buffers and multiplex multiple virtual channels over one physical channel in a fair fashion. If virtual channels sharing a physical signal path are independent, they will behave as *real* channels as far as deadlocks are concerned[30]. Naturally, the combined bandwidth of all virtual channels will not exceed the bandwidth of the underlying physical channel.

The number of virtual channels needed depends on the network and on the routing function. The proposed message system uses two virtual channels per physical channel and direction. If physical channels are operated bidirectionally, each direction is regarded as one channel with two virtual channels. It will be argued on empirical grounds that these resources are sufficient to support deadlock-free operation with a routing function that provides at least one *shortest path* route for each pair of nodes[31].

Virtual channels are strictly used as a vehicle to avoid deadlocks. There are other potential uses of virtual channels, such as flow control, traffic balancing, or support of prioritized traffic, which are beyond the scope of this thesis.

---

[30] One virtual channel must not be able to interfere with the operation of others. In particular, a blocked virtual channel will not prevent traffic on other virtual channels.

[31] Two virtual channels always guarantee the existence of deadlock-free operation if the shortest path requirement is dropped: designate one node as root and route all traffic through the root. Traffic *to* the root uses virtual channel 0 and traffic *from* the root uses virtual channel 1.

3.6.3. The Routing Function

The routing function $R:(V\times V)\rightarrow E$ maps the current location of a packet and its destination to an outbound channel. This routing function is considerably simpler than Dally's $R:(E\times V)\rightarrow E$ as given in [35], which essentially requires one - potentially different - map for each incoming channel[32].

One of the advantages of a routing function that depends only on the packet's current location and its destination is the uniform treatment of all packets within one node. The router in node v_i simply uses the destination address x as an index to the routing table and retrieves two bit-vectors $(\vec{R}_i(x),\vec{S}_i(x))$, each with one bit per outbound channel. The first bit is set if the corresponding real channel can be used to transmit the packet. The second bit determines which virtual channel must be used[33]. This routing function is considered *memoryless* in the sense that it does not depend on the history of the packet.

To define the notation, it is assumed that $G:(V,E\subseteq(V\times V))$ is a strongly connected, directed graph with $n=|V|$ vertices (i.e. network nodes) and $m=|E|$ edges (i.e. real channels) representing the network topology. Edge $e_i\in E$ connects vertex $v_{\alpha(i)}$ to vertex $v_{\beta(i)}$.

The distance matrix $\mathbf{D}=[d_{ij}]_{i,j=1}^{n}$ is defined by the minimal number of edge traversals to send a packet from node v_i to node v_j. Hence the *complete*, shortest path routing function to destination node v_x is

$$\mathbf{R}(x) = [\vec{R}_1(x)\ \cdots\ \vec{R}_n(x)] \tag{3.14}$$

with:

$$\vec{R}_i(x) = [r_{i1}(x)\ \cdots\ r_{im}(x)] \qquad ; \quad r_{ij}(x)=\begin{cases}1 & \text{if}\quad \alpha(j)=i\ \wedge\ d_{ix}-d_{\beta(j)x}=1\\ 0 & \text{otherwise}\end{cases}$$

$\mathbf{R}(x)$ describes a subgraph of G that contains all possible paths of packets addressed to v_x. The virtual channel assignment function $\mathbf{S}(x)$ is structured likewise.

At this point, only shortest path routing functions are considered. Other routing functions are possible by introducing detours for some traffic. The only structural requirement on $\mathbf{R}(x)$ is that the associated subgraph of G is acyclic. A cycle in this graph would permit a packet to traverse that cycle indefinitely. Suboptimal routing functions may become necessary for certain pathological topologies for which no deadlock-free, optimal routing function exists within the two virtual channel limitation.

---

[32]Besides being unnecessarily complex, Dally's deadlock avoidance scheme also inflates the number of virtual channels, deals only with certain topologies and is restricted to E^3 routing.

[33]The encoding scheme was chosen because it is simpler to use in the virtual channel assignment algorithm. However, a hardware implementation may prefer to use one bit to enable each virtual channel. The advantage of this method is that it can enable *both* virtual channels simultaneausly. Depending on the network topology, a small fraction of channels (generally less than 5%) allows both virtual channels. In these cases, the additional buffer space gives a slight performance advantage if both virtual channels are used.

The E^3 routing function $\mathbf{R}^e(x)$ is a subset of the complete routing function. It is derived from the complete routing function such that only one bit is set in each non-terminal routing table entry:

$$r^e_{ij}(x) = \begin{cases} 1 & \text{if } \ r_{i\,p(j)}(x) = 1 \ \wedge \ \forall\, k < j : r_{i\,p(k)}(x) = 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.15}$$

where $p(i)$ is an arbitrary permutation of $1 \cdots m$. The permutation $p(i)$ establishes an order in the dimensions of the routing space. For example, traditional E^3 for hypercubes routes packets first through channels that correspond to the least significant address bits that differ. The "least significant bit first" strategy is just one of many possible orders. While the complete routing function may allow multiple channels of equal distance to the destination node, the E^3 method resolves this ambiguity or non-determinism and yields a statically defined route for each pair of nodes.

3.6.4. Constructing Deadlock-Free Routing Functions

Edge e_i *depends* on e_j if it is possible for e_i to hold a packet that directly requires e_j to proceed. This leads to the definition of the dependency matrix:

$$\mathbf{A} = [a_{ij}]^m_{i,j=1} \qquad ; \quad a_{ij} = \begin{cases} 1 & \text{if } \ \exists x : r_{\alpha(i)i}(x) = 1 \ \wedge \ r_{\beta(i)j}(x) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.16}$$

A partial order on E exists if the directed graph corresponding to $\mathbf{A}$ is acyclic. Actually, this relation is equivalent to deadlock freedom only for deterministic routing functions. If $\vec{R}_i(x)$ has more than one non-zero bit, a deadlock would require *all* permissible channels to be blocked; therefore requiring an acyclic dependency graph is overly conservative.

Since most network topologies will have cycles in the dependency graph, m virtual channels are added to the network such that $\alpha(i) = \alpha(i+m)$ and $\beta(i) = \beta(i+m)$ for $i = 1 \cdots m$. The virtual channel assignment function $\mathbf{S}(x)$ decides which edge is used for a given packet.

$\mathbf{S}(x)$ is computed by the following procedure (details follow):

1. List all dependencies required to implement the E^3 routing function defined by Equation (3.15).
2. Expand the dependency list to include all possible virtual channel assignments.
3. Sort the expanded dependency list.
4. Incrementally construct $\mathbf{A}$ such that only dependencies that do not complete a cycle in the dependency graph are added.
5. List all non-E^3 routing options.
6. Expand the optional routing list to include all possible virtual channel assignments.
7. Sort the expanded optional routing list.
8. Incrementally add routing options that do not complete a cycle in the dependency graph.

If step four succeeds, a viable and deadlock-free routing is assured. This routing is subsequently improved in step eight by gradually adding alternate paths to the E^3 routing. The heuristic components to this algorithm are confined to the two sorting steps. The algorithm is capable of generating every possible deadlock-free routing if provided with a suitable construction sequence. However, the optimal order to apply the dependencies is non-trivial and depends on the particular network topology. It turns out that a number of fairly simple sort keys *work* for a wide range of nets. Hence this procedure is an empirical approach to constructing the routing functions for networks of practical interest. This procedure - as presented in this thesis - is not guaranteed to succeed for an arbitrary network.

Step 1: The dependency list L is simply the set of edge pairs that contribute to **A**:

$$L = \{ (e_i \in E, e_j \in E, v_k \in V) \mid r^e_{\alpha(i)i}(k) = 1 \ \wedge \ r^e_{\beta(i)j}(k) = 1 \}$$

Technically, it is easiest to store the network such that each vertex has backward pointers to the inbound channels. L is computed by enumerating all possible destinations for each vertex. The E^3 routing determines the outbound edge uniquely and a loop over all permissible inbound edges produces the elements of L.

Step 2: The expanded dependency list L^e is derived from L by enumerating all ways to use virtual channels to satisfy a dependency:

$$L^e = \{ (e_i \in E, e_j \in E, v_k \in V) \mid (e_{i \bmod m}, e_{j \bmod m}, v_k) \in L \}$$

This step quadruples the number of elements in the dependency lists because there are four ways to use the two virtual channels available for each in- and out-bound edge. The implementation uses an indirect data structure for L^e so that it is easy to remove the corresponding elements once one member of the set is satisfied. Likewise, the data structure must be able to detect the failure to satisfy any of the four implementation candidates.

Step 3: Sorting L^e determines the order in which the dependencies are used to construct D. Any correct (i.e. deadlock-free) routing function can be built if the elements of L^e are arranged properly. The other steps of this construction algorithm are merely a framework to ensure correctness. The sorting step is the only part of this procedure concerned with the actual topology. It turns out that fairly simple sorting keys cover wide classes of network topologies.

For example, the routing functions for members of the family of k-ary hypercubes can be generated by sorting (the sub-keys are listed by their priority):

1. by the destination vertex index ($= k$, lowest first)
2. by the distance ($= d_{\beta(i)k}$, lowest first)
3. by the current vertex index ($= \beta(i)$, lowest first)
4. by the predecessor vertex index ($= \alpha(i)$, lowest first)

5. by the virtual channel combination (precedence is given to $0 \to 0$ over $1 \to 0$ over $0 \to 1$ over $1 \to 1$)

This particular key also solves many randomly generated networks but fails for k-Shuffles. k-Shuffles present a difficult test case by virtue of their complicated structure (see Figure 2-2). Routing functions for k-shuffle networks can be generated by sorting:

1. by the distance ($= d_{\beta(i)k}$, lowest first)
2. by the virtual channel combination (precedence is given to $0 \to 0$ over $1 \to 0$ over $1 \to 1$ over $0 \to 1$)
3. by the inbound edge index (i, highest first)
4. by the outbound edge index (j, lowest first)

Both keys (for k-ary hypercubes and k-Shuffles) rely on the vertex and edge enumeration that is defined by the generator functions in Section 2.2.2. Each key is also applicable for other (related) network topologies. Future research may address systematic ways of constructing these keys.

Step 4: **A** and **S** are initialized to 0. For each element $l^e \in L^e$ (processed in the order given in the previous step), a test is done to determine if that dependency would create a cycle in the dependency graph. If the test succeeds and if l^e is compatible[34] with **S**, then **A** and **S** are updated and all dependencies related to l^e are removed from L^e. If the test fails and if l^e was the last option to implement a particular dependency, the procedure terminates unsuccessfully. Additional sorting keys could be tried by restarting at step three.

Step 5: At this point, a viable and correct routing has been established. However this routing function covers only the E^3 routes. It is desirable to add the non-deterministic routes to allow for adaptive routing strategies. A list of alternate routes is constructed:

$$L^a = \{ (e_i \in E, v_k \in V) \mid r_{\alpha(i)i}(k) = 1 \ \wedge \ r^e_{\alpha(i)i}(k) = 0 \}$$

Step 6: Each $l^a \in L^a$ could be implemented by using either virtual channel 0 or 1. So L^r is derived from L^a by allowing both options:

$$L^r = \{ (e_i \in E, v_k \in V) \mid (e_{i \bmod m}, v_k) \in L^a \}$$

Again, provisions are made that any $l^a \in L^a$ is implemented, at most, once.

Step 7: This sorting step is similar to step three, but the key is less critical. A bad key may under-utilize the network but will still result in a solution. Generally, it is advisable to add low distance $l^r \in L^r$ first and to use virtual channel 0 whenever possible.

Step 8: Each $l^a \in L^a$ generates a number of dependencies, all of which must be satisfied before an edge is added. If all dependencies hold, **A** and **S** are updated. If both attempts to implement a particular route fail, it is removed from **R**.

---

[34]During the construction procedure, **S** must support three values: *unassigned*, *assigned to virtual channel 0*, and *assigned to virtual channel 1*.

3.6.5. Results

Figure 3-28 shows the routing function **R**(x) (left) and the virtual channel assignment function **S**(x) for a 3-dimensional, binary cube-connected cycle. Each number represents a 3-bit vector corresponding to the three outbound channels for each node. While the description of the construction algorithm uses bit vectors of length m, it is really only necessary to consider bits with potentially non-zero value. 24 nodes and 72 channels require 1920 dependencies to be satisfied for E^3 routing. 119 of the 120 additional, non-E^3 routes were implemented. This example is typical for predominant use of virtual channel 0: it is used 1520 times while only 208 routes use virtual channel 1.

```
 0: 0 1 2 4 4  4 1 1 1 5  4 4 2 2 2  6 4 4 3 2  1 7 4 4 | 0 0 0 0 0  0 0 0 0 0  0 0 0 0 0  0 0 0 0 0  0 0 0 0
 1: 2 0 1 2 2  2 4 4 4 4  6 4 1 1 1  1 3 2 4 5  4 4 7 4 | 0 0 0 2 2  2 0 0 0 0  2 0 0 0 0  0 2 2 0 0  0 0 2 0
 2: 1 2 0 1 1  1 2 2 2 2  1 3 4 4 4  4 4 5 4 4  6 4 4 7 | 0 0 0 1 1  1 2 2 2 2  1 3 0 0 0  0 0 1 0 0  2 0 0 3
 3: 4 4 4 0 1  2 5 4 4 1  1 1 6 4 4  2 2 2 7 4  4 3 2 1 | 0 4 4 0 0  0 4 4 4 0  0 0 4 4 4  0 0 0 4 4  4 0 0 0
 4: 2 2 2 2 0  1 4 6 4 4  4 4 1 3 2  1 1 1 4 7  4 4 5 4 | 0 2 2 0 0  0 0 2 4 0  0 0 0 3 2  0 0 0 0 7  4 0 0 0
 5: 1 1 1 1 2  0 2 1 3 2  2 2 4 4 5  4 4 4 4 4  7 4 4 6 | 0 1 1 0 0  0 2 1 3 2  2 2 0 4 1  0 0 0 0 4  7 0 0 2
 6: 1 1 1 5 4  4 0 1 2 4  4 4 3 2 1  7 4 4 2 2  2 6 4 4 | 0 0 1 1 0  4 0 0 0 0  0 0 1 0 1  5 0 4 0 0  0 0 0 0
 7: 4 4 4 4 6  4 2 0 1 2  2 2 4 5 4  4 7 4 1 1  1 1 3 2 | 0 0 4 4 4  4 0 0 0 0  0 0 4 4 4  4 4 4 0 0  0 0 0 0
 8: 2 2 2 2 1  3 1 2 0 1  1 1 4 4 6  4 4 7 4 4  4 4 4 5 | 0 0 2 2 1  3 0 0 0 1  1 1 0 0 2  4 0 7 0 0  0 0 0 1
 9: 5 4 4 1 1  1 4 4 4 0  1 2 7 4 4  3 2 1 6 4  4 2 2 2 | 0 0 4 0 0  1 0 0 4 0  0 0 5 4 4  1 0 1 4 4  4 0 0 0
10: 4 6 4 4 4  4 2 2 2 2  0 1 4 7 4  4 5 4 1 3  2 1 1 1 | 0 4 4 0 0  4 0 0 2 0  0 0 4 6 4  4 4 4 0 2  2 0 0 0
11: 2 1 3 2 2  2 1 1 1 1  2 0 4 4 7  4 4 6 4 4  5 4 4 4 | 0 0 3 0 0  2 0 0 1 0  0 0 0 0 3  0 0 2 0 0  1 0 0 0
12: 2 2 2 6 4  4 3 2 1 7  4 4 0 1 2  4 4 4 1 1  1 5 4 4 | 0 0 0 2 0  0 2 2 0 7  4 0 0 0 0  0 0 0 0 0  0 0 0 0
13: 1 1 1 1 3  2 4 5 4 4  5 4 2 0 1  2 2 2 4 4  4 4 6 4 | 0 0 0 1 3  2 0 1 0 4  5 0 0 0 0  2 2 2 0 0  0 0 2 0
14: 4 4 4 4 4  5 4 4 6 4  4 7 1 2 0  1 1 1 2 2  2 2 1 3 | 0 0 0 4 4  4 4 4 4 4  4 4 0 0 0  0 0 0 0 0  0 0 0 0
15: 6 4 4 2 2  2 7 4 4 3  2 1 4 4 4  0 1 2 5 4  4 1 1 1 | 0 0 0 0 0  0 6 4 4 2  2 0 0 4 0  0 0 0 4 4  4 0 0 0
16: 1 3 2 1 1  1 4 7 4 4  5 4 2 2 2  2 0 1 4 6  4 4 4 4 | 0 1 0 0 0  0 0 3 0 0  1 0 0 2 0  0 0 0 0 2  0 0 0 0
17: 4 4 5 4 4  4 4 4 7 4  4 6 1 1 1  1 2 0 2 1  3 2 2 2 | 0 4 4 0 0  0 4 4 5 4  4 4 0 1 0  0 0 0 0 1  1 0 0 0
18: 3 2 1 7 4  4 2 2 2 6  4 4 1 1 1  5 4 4 0 1  2 4 4 4 | 0 0 0 3 0  0 0 0 0 2  0 0 0 0 0  1 0 0 0 0  0 0 0 0
19: 4 5 4 4 7  4 1 1 1 1  3 2 4 4 4  4 6 4 2 0  1 2 2 2 | 0 0 0 4 5  4 0 0 0 1  1 0 0 0 0  4 4 4 0 0  0 0 0 0
20: 4 4 6 4 4  7 4 4 4 4  4 5 2 2 2  2 1 3 1 2  0 1 1 1 | 0 0 4 4 4  6 0 0 0 4  4 4 0 0 0  2 0 2 0 0  0 0 0 0
21: 7 4 4 3 2  1 6 4 4 2  2 2 5 4 4  1 1 1 4 4  4 0 1 2 | 0 0 0 0 0  0 0 0 0 0  0 0 0 0 0  0 0 0 0 0  0 0 0 0
22: 4 7 4 4 5  4 1 3 2 1  1 1 4 6 4  4 4 4 2 2  2 2 0 1 | 0 4 0 0 0  0 0 0 0 0  0 0 0 4 0  0 0 0 0 0  0 0 0 0
23: 4 4 7 4 4  6 4 4 5 4  4 4 2 1 3  2 2 2 1 1  1 1 2 0 | 0 0 4 0 0  4 0 0 4 0  0 0 0 0 0  0 0 0 0 0  0 0 0 0
```

Figure 3-28: Routing Table and Virtual Channel Assignments for a three Dimensional Binary Cube-Connected Cycle Network

It is interesting to compare the results of this new deadlock avoidance method with those obtained by known, conventional methods. In the case of k-ary hypercubes and E^3 routing,

$$s_{ij}(x) = \begin{cases} 1 & \text{if} \quad r^e_{ij}(x) = 1 \ \wedge \ i < \beta(j) \\ 0 & \text{otherwise} \end{cases} \tag{3.17}$$

yields a deadlock-free routing. This virtual channel assignment is essentially equivalent to the one described in [35]. Starting with Equation (3.17), steps five through eight were applied to add the non-deterministic routes. The results are listed in Table 3-6.

The numbers corresponding to the conventional routing function defined by Equation (3.17) are given in parentheses. The virtual channel ratio has little - if any - practical impact, but it is significant that the constructed routing function consistently offers more options for adaptive routing strategies.

Routing Function Characteristics		
Cube Type arity, dimensions	V-Channel ratio V-Ch. 0 : V-Ch. 1	Implemented Non-E^3 Routes
2, 4	926:98 (801:223)	210/272 (154/272)
2, 5	4586:534 (3999:1121)	1130/1568 (757/1568)
2, 6	21860:2716 (19190:5386)	5634/8256 (3561/8256)
2, 7	101492:13196 (89378:25310)	26798/41088 (16333/41088)
10, 2	13610:6390 (11369:8631)	4860/8100 (4104/8100)
3, 3	1768:419 (1544:643)	572/756 (380/756)
4, 3	9465:2823 (8241:4047)	3435/5184 (2469/5184)
5, 3	35165:11710 (30489:16386)	13360/22000 (10384/22000)

Table 3-6: Conventional (...) vs. Constructed Routing Function Performance

The results for k-Shuffles are relatively uninteresting because the shortest path routes in k-Shuffles are never ambiguous, thus all routing functions are of E^3 type. The ratio of virtual channel 0 to 1 usage is somewhat lower than for cubes. For example, a 3-dimensional 4-shuffle has a ratio of 14970 to 1158.

Results for random network topologies vary widely in the virtual channel ratio. Generally, around 80% of all non-E^3 routes are feasible. This is somewhat higher than the results for k-ary hypercubes, which is largely due to the fact that fewer non-E^3 routes are present in such networks.

3.6.6. Summary

An empirical procedure to construct deadlock-free routing functions that can handle a wide range of network topologies of practical interest was developed. This procedure was successfully tested on the network topologies addressed in this thesis[35] and generates less constrained routing functions than conventional methods.

[35] *Fat-Trees* are trivially deadlock-free and *Star* shaped networks don't have store-and-forward deadlock potential.

It was found that two virtual channels per physical connection are sufficient to ensure a deadlock-free, shortest path routing function.

The procedure is not guaranteed to succeed on any arbitrary topology. However, given the simple nature of the employed heuristic, it is tempting to conjecture the existence of a practical method that never fails.

3.7. Beyond a Single Cabinet

This section describes how medium distance channels are used to connect several clusters of processors. Each cluster consists of several processing nodes in a relatively compact assembly. The close proximity of these nodes allows the use of high speed - but short range - channels. Longer range - but lower speed - channels link clusters. The characteristics of these two distinct channel types were described in sections 3.4.1 and 3.4.2. The key component is an interface chip that transfers traffic between these two channel types. After describing the overall system structure and the interface chip, the remainder of this section will address practical issues on how to organize and maintain a partially distributed multiprocessor.

3.7.1. Overview

Typically, computing facilities are upgraded from time to time. Multiprocessor systems especially invite extensions by adding processors. There are natural constraints to this growth such as mechanical, thermal, and electrical limits. This leads to the desire to interconnect several multiprocessor systems. Besides expansion, processor clusters may be spatially distributed because portions are tightly tied to facilities demanding high I/O-rates, such as disk farms, graphics displays, imaging devices (video, X-ray, NMR, radar, sonar), or data acquisition equipment (high energy physics detectors).

In all cases, the reasons for distributing the system are a matter of meeting external constraints and are not due to the intended programming model and/or the preferred structure of the application code. In fact, this extra system complexity is quite unwelcome to most application programmers. Hence it is desirable to hide the extra complexity for multi-cluster systems by architectural support: the facilities provided by the message system should be uniform across the entire system.

Composite systems can be constructed by appropriate network software and conventional network technology such as Ethernet, Hyperchannel, FDDI token ring, etc. However, this local area network hardware was designed with other objectives (such as high efficiency for large, blocked data transfers, support for widely different hardware). The required software increases complexity, creates more interfaces and may require system calls or context switches. This amounts to a large decrease in available communication bandwidth and a significant increase in latency. It will be shown that a direct implementation of support for a distributed message system can both be simpler and have higher performance.

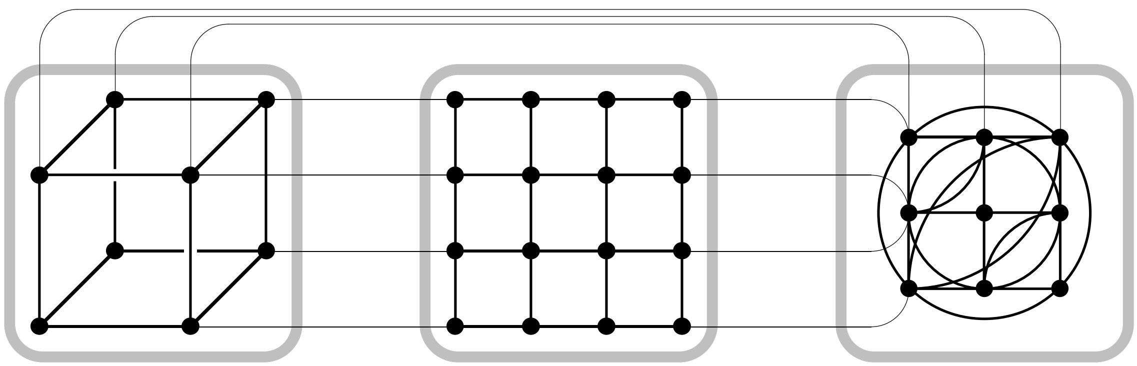


Figure 3-29: The Global Picture

A multi-cluster system may include clusters of different sizes and topologies. The example of a three cluster system in Figure 3-29 links a cube, a grid, and a shuffle exchange network to form a rather random, composite topology. Inter-cluster connections (thin lines) are likely designed with a different set of goals and constraints. For example, numerous wires connecting cabinets in arbitrary ways are inconvenient if not impractical[36].

3.7.2. Supporting Dissimilar Channels

Implementing a network of processing clusters requires support for the two types of channels described in section 3.4.1 and 3.4.2. This could be achieved by either a router that can handle both channel types or by an interface circuit that transfers packets from one channel type to the other.

Small systems are likely to form only one cluster; hence the support for intra-cluster channels is not essential. This argues for a separate interface circuit. The router of each node deals only with intra-cluster channels. Once the system is expanded to multiple clusters, interface circuits are added to translate intra-cluster channels to inter-cluster channels for those connections that cross cluster boundaries.

The alternative, designing intra-cluster channel support directly into the router, increases the complexity of the router. It is also less flexible, because the ratio of intra-cluster to inter-cluster channels would be fixed.

[36]Caltech did assemble a hypercube out of IBM-PC's using serial links for demonstration purposes [111], but even a small system caused quite a maze of wires: hardly a scalable approach to integrated multiprocessor networking.

The separate interface circuit approach has the disadvantage of an increase in latency. Protocol translation demands the complete reception of a packet. Inter-cluster channels have a higher error probability; hence they require better redundancy checks. Furthermore, the difference in transmission speeds is also greatly complicating attempts to start the transmission of a partially received packet.

The task of the interface is relatively simple because it does not require the interpretation of the translated packet. In particular, no routing functions are required. Therefore it is possible to combine the translation circuitry for multiple channels on a single integrated circuit. Besides reducing the overall component count, this approach also add some flexibility because several inter-cluster channels may be assigned to one intra-cluster channel. By keeping this assignment programmable, intra-cluster bandwidth allocation can be controlled. For example, a system that normally uses 4 to 1 channel ratio, may settle for a lower ratio if some channels fail while maintaining the same topology.

The remainder of this section describes an implementation study of an integrated circuit that supports a number of channel interfaces. The details of this design are not essential to the concept of a network with multiple processor clusters.

3.7.2.1. The Interface Chip

The interface chip is the key component to extend the network beyond one cluster. It is essentially a converter between an intra-cluster channel and an inter-cluster channel. The conversion includes a translation of the protocols and electrical signals. A certain amount of buffering is necessary because the inter-cluster channels are roughly four times slower than the intra-cluster channel. Buffer space is also needed because of the different operating modes (directional vs. bidirectional).

Figure 3-30 is a simplified block diagram of the interface chip (IFC). Twelve pairs[37] of inter-cluster channels and 6 intra-cluster channels are supported. Hence, each side of the chip has roughly the same channel capacity.

The largest block of the IFC is an array of shift registers that form the receive and transmit buffers. The bottom section consists of the local (intra-cluster) and remote (inter-cluster) receive buffers. These shift registers feature a serial input and can be read in parallel. Each receiver is associated with two buffers so that continuous back-to-back reception is possible. The shift registers are actually four bits wide so that four bits can be accepted in each cycle.

---

[37]This organization is largely influenced by the size of common cables, connectors and IC-packages. Cables with 25 twisted pair wires (50 conductors) are a standard for telephone systems. Fifty signal paths are also a common connector size and pin-grid arrays with 84 pins allow for a safe number of power and ground connections for the interface chip.

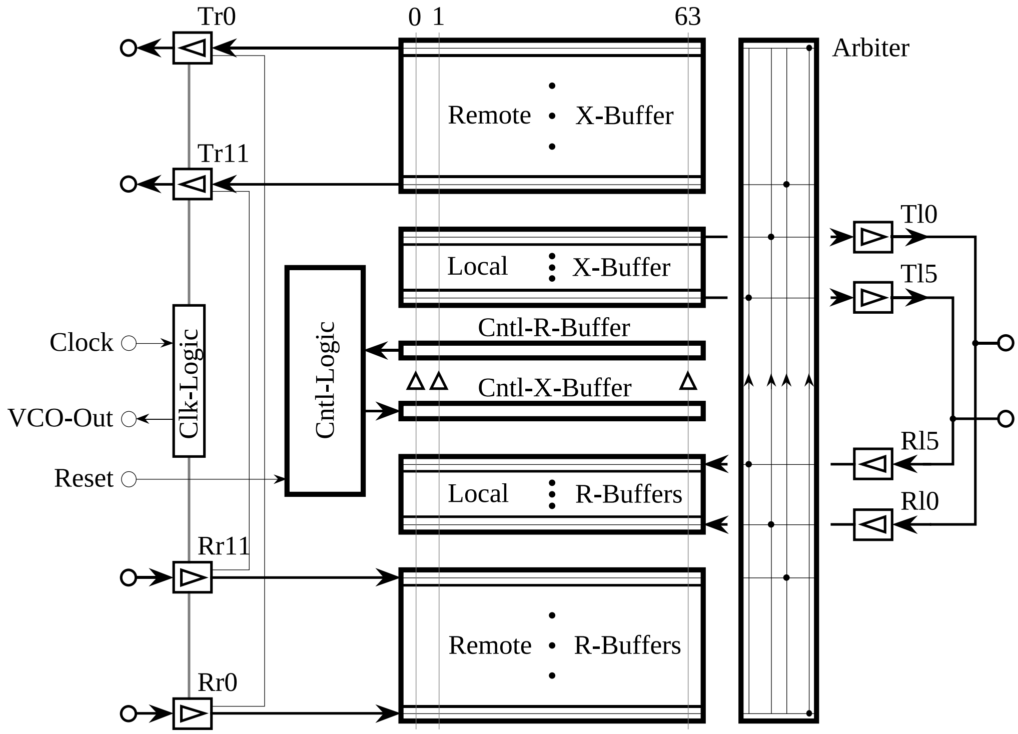


Figure 3-30: Block Diagram of the Interface Chip

The remote receive buffers are clocked by the receiver finite state machine while the local receive buffers are driven by the system clock. Synchronization of remote data is required only once per successfully received packet. Therefore, a sufficient amount of time is available for the synchronization process to minimize the metastability potential.

All receive buffers can be gated onto a common, parallel packet bus that can transfer an entire packet from a receive buffer to a transmit buffer in one system clock cycle. The packet buffer runs perpendicular to the shift registers to simplify the layout. The packet bus is directional and has buffers at the border between the receive and transmit buffers.

The local transmit buffers can receive data from the packet bus in parallel and can shift out data four bit per cycle. This structure is similar to the structure of the receive buffers. Again, two buffers per transmitter are needed to ensure continuous operation.

The *arbiter* controls the operation of the packet bus. Each receive buffer holding a packet applies for a transfer to an empty transmit buffer. The receive to transmit buffer assignment is controlled by 18 arbitration circuits. Each receiver and transmitter is attached to exactly one arbitration circuit. The assignment is programmable and may be changed at network configuration time. The default state after a *reset* is the echo-configuration: each receiver is connected to a different arbitration circuit and each transmitter is attached to the corresponding arbitration circuit. Therefore the IFC will act as a mirror after power up. For example, a packet received by the remote receiver *Rr0* will cause the arbiter to schedule a transfer to the remote transmitter *Tr0*.

Multiple receivers and/or transmitters may use the same arbitration circuit. Obviously, each arbitration circuit with an attached receiver must have at least one transmitter. All arbitration circuits with pending traffic compete for the packet bus in round robin fashion. Round robin logic is also used to decide between multiple active receivers attached to the same arbitration circuit. If there is more than one transmitter with an empty buffer attached to one arbitration circuit, the first one is used.

The IFC is not a router. The path of a packet through the IFC depends only on the arbiter configuration and the availability of resources. For all practical purposes, the data paths through the IFC are statically configured. The arbitration scheme is flexible enough to allow any input to output channel combination, just like a cross-bar. However, this flexibility is useful only to control the network topology statically. Infrequent changes to the network structure are possible, for example, to compensate for channel failures. It is not possible to change the IFC configuration on a packet-by-packet basis or to use the packet destination address to select a path.

The IFC can parallel communication resources. For example, an intra-cluster channel may be connected to four inter-cluster channels. This would allow the intra-cluster channel to run at full capacity by utilizing all four inter-cluster channels. Since a packet is always transmitted atomically over one channel, the latency of a remote transmission is not changed for the non-blocked case. Paralleling communication resources does not guarantee preservation of packet order: a transmission error may cause a retransmission of a particular packet while a subsequent packet proceeds without delay on a different channel. Since the message system is intended to use adaptive routing, it is prepared to deal with out of order transmissions.

The 12 receivers feature phase-frequency comparators that feed a common output pin that controls an external, crystal based VCO. The VCO output of several IFC's can be tied together. Only one VCO is needed for a group of IFC's.

3.7.2.2. Control Registers

There are two special registers attached to the packet bus that control the IFC. The *control receive buffer* intercepts diagnostic packets that can be used to configure the IFC. Diagnostic packets may arrive on any channel. Therefore it is possible to configure IFC's remotely.

Diagnostic packets are interpreted by the control logic. Within one system clock cycle, a response diagnostic packet, formed in the *control transmit buffer*, is sent to the channel that received the original diagnostic packet. Replies to diagnostic packets are given priority by the arbiter.

The diagnostic packet to control the IFC subdivides the data portion into 5 fields:

Diagnostic Type: Diagnostic packets are identified by the type field in the packet header. However, the type field is small and the infrequent number of diagnostic packets does not warrant the use of scarce coding space in the header. Therefore, all diagnostic packets are further classified by a type field in the data portion of the packet. The diagnostic type filed is 8 bits wide and two codes are reserved for IFC use: *IFC-execute* and *IFC-reply*. The execute type is consumed by the IFC and a reply-type packet is sent back to the originator.

Key: An 8 bit key field is used for the network configuration phase. Algorithms to explore the network topology will use this field to store markers and to *smuggle* IFC-diagnostic packets through an IFC in networks with multiple IFC-layers.

Control: Two control bits are provided for special access modes. If the *keyed* bit is asserted, the key-field is used to decide when to ignore an IFC-execute packet. If the key field does not match the key stored locally in the IFC and if the local key is not zero, the diagnostic packet will be treated as a plain packet. If the *override* bit is asserted, diagnostic packets take effect even if the IFC is locked. Locking is accomplished by one of two semaphores local to the IFC control logic. This is not a security feature, rather it is a necessary facility for a distributed cold start of a network with unknown topology.

Register Select: A 6-bit field is used to select control registers associated with each channel. Each of the 18 receive and 18 transmit channels has its own control register, whitch is merely a collection of state bits associated with that channel. Codes not used for these 36 registers will select a generic control register.

Register Operation: An 8-bit field that specifies state changes in the associated entity.

The IFC has 10 bits of global state information: the 8 bit key and two semaphores called *lock* and *signal*. This state is initialized to 0 upon power up reset. Once the IFC is running, this state can be changed by accessing the generic control register. 5 bits of the register operation are used to load the key from the key-field of the diagnostic packet and to set/reset each semaphore.

The register operations for channels provide low level control that is used for network configuration and for error recovery:

Enable: If the enable bit is set, the channel will accept packets. Channels are enabled during reset (but the IFC starts in echo-configuration). The IFC will execute a reset if all receive channels are disabled. Otherwise, a hardware reset would be required to restart an IFC.

Arbiter Number: This 4-bit field selects the arbitration circuit.

Clear: Asserting the clear bit will cause a reset of the channel controller. Any packets stored in the associated buffers are lost.

Error Reset: Asserting the error reset bit will clear the pending error flag and reset the traffic counter.

The diagnostic reply packet header is derived from the original header by moving the source address field to the destination address field. The new source address field is replaced by the destination address of the preceding diagnostic packet. The very first reply will have a source address of 0. This mechanism allows the controlling node to detect diagnostic packets from other nodes.

The data portion of the diagnostic reply packet has an invariant part that contains the diagnostic packet type (*IFC-reply*), the value of the *lock* and *signal* semaphores **before** the execution of the specified operation, and a 6-bit code that identifies a pending error condition. The 36 channel controllers are ranked statically and the address of the highest priority channel with an error condition is given. Two other error codes are used: *no pending error* and *lock-error* which has the highest priority. A lock-error occurs if the lock semaphore is set and the override bit was not asserted in the diagnostic packet. In this, case no state changes were made.

If the diagnostic packet addresses the generic control register, the value of the key prior to the current operation is returned in the specific portion of the diagnostic reply packet. Otherwise, the status of the addressed channel is returned. The status is composed of the channel controller state bits:

Enable-Bit: The enable bit is set if the channel is allowed to accept packets. If it is turned off, pending packets are allowed to be sent or moved to a transmitter.

Master-Bit: This is available only for intra-cluster channels. It is simply the value of the protocol asymmetry breaking bit.

Parity-Error-Bit: If asserted, a parity error has occurred since the last error-reset operation. Reading this bit does not clear it. Only an error-reset operation will clear error conditions.

Synchronization-Error-Bit:
A protocol problem was encountered. Both parity and synchronization errors will result in retransmissions. Perpetual retries can be detected through multiple error-reset operations. If no successful transmission occurs between successive error indications, a persistent error is detected and the channel should be disabled.

Dead-Bit: The dead-bit is set initially and cleared once the transmission protocol is established. The dead-bit disables the channel similarly to a cleared enable bit. However, both are independent. If both the enable-bit and the dead-bit are asserted, an error condition is present. If the enable bit is cleared, the entire channel cannot raise an error condition that is reflected in the invariant part of the diagnostic reply packet. The purpose of the dead-bit is to identify unused channels.

Buffer Status: Two bits indicate the presence of a packet in the corresponding buffer.

Arbiter Circuit: A 4-bit field identifies the arbitration circuit to which this channel is attached.

Packet Counter: A 5-bit field that is the number of the most significant '1'-bit of a 32-bit packet frequency counter. This is basically a network debugging and monitoring tool. This logarithmic representation reduces the number of required bits while remaining useful for both monitoring and debugging operation. For debugging, it must distinguish between small numbers of packets: 0, 1 and 2. Traffic monitoring more often concerns the in orders of magnitude; hence it is acceptable that a 5 is returned for 16 to 31 successful packet transmissions.

Diagnostic packets of *IFC-reply* type are treated like normal packets. The one and only diagnostic packet type that receives attention is the *IFC-execute* type. A more detailed discussion of packet types will be given in Chapter 4.

3.7.3. Implementing an Inter-Cluster Network

The inter-cluster network is constrained by wiring considerations. It is not practical to route arbitrary wires between clusters. However, high fan-out networks are quite attractive because of their higher bandwidth, lower latency and robustness. A compromise between conflicting goals - simple cable routes vs. rich topologies - is possible by using bundles of wires.

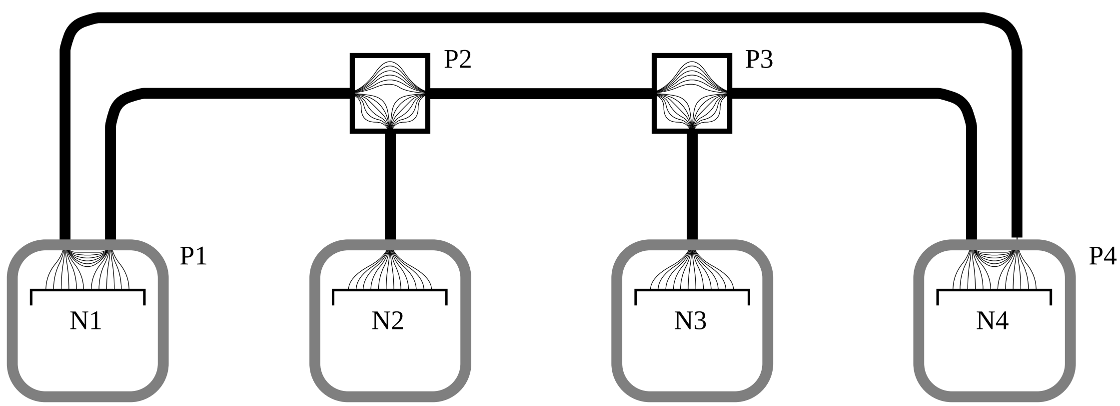


Figure 3-31: Practical Interconnections

Figure 3-31 illustrates the use of cables with multiple signal paths. A 25-pair phone cable offers 12 independent routes (two twisted pair wires must be paired to provide one directional path each way as required by the inter-cluster channel protocol). A total usable bandwidth of roughly 50 Mbytes/sec is a respectable capacity per cable. Larger systems may use multiple IFC's, larger bundles or a higher cable quality to increase the data rate.

The key feature of the 4 nodes depicted in Figure 3-31 is the difference between the cable topology and the network topology. The cables form a simple ring. Clusters may join the ring by either having two attached cables (N1, N4) or by virtue of a feed line connected to a passive switch box (N2, N3). In both cases, the wires of the cable undergo a certain permutation with some wires being connected to the cluster and others being routed through the node without any electrical connections. Hence the actual network topology can be quite different. Also, the complete failure of a single node will not sever all channels routed through that node[38].

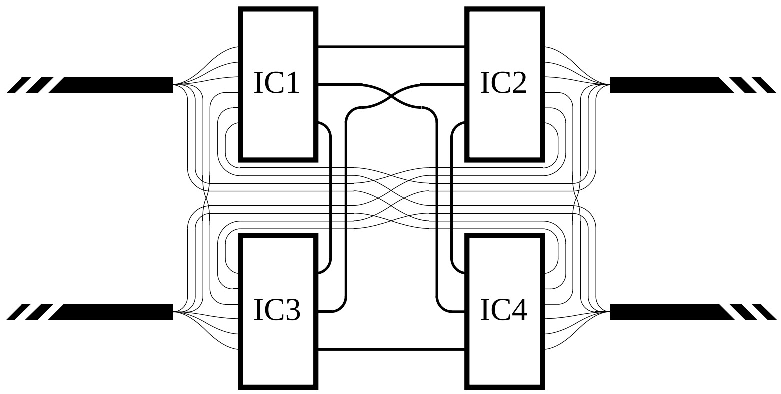


Figure 3-32: A Four-Way Repeater

Figure 3-32 shows a different use of the IFC. In this case four IFC's are used as repeaters outside of a cluster. The intra-cluster channels are either not used at all or they are just used to connect to other IFC's. Besides the IFC's, only a power supply and one crystal VCO are required. The entire repeater may be located in a remote wiring closet. Besides extending the usable length of inter-cluster channels (at the expense of an increase in latency), repeaters may also be used to reconfigure the network if the global traffic pattern changes, the network is extended or channels fail.

3.7.4. Summary

The interface chip (IFC) is a versatile inter cluster network building block. The IFC can translate packets between the fast intra-cluster network and the slower inter-cluster networks. It is also usable as stand-alone repeater to extend the range of inter-cluster channels.

The use of multi-conductor cables supports many inter-cluster channels without undue wiring complexity. The inter-cluster network topology is relatively independent of the interconnection topology and the composite network can survive complete node failures.

[38]Token-rings usually ensure continuity with a mechanical relay that closes the ring if one node loses power. However, given the number of channels involved, a completely passive system is simpler.

Another function of the IFC is synchronizing system clocks across cluster boundaries. Later sections will describe how this feature is used to build a precise global clock.

The facilities of the IFC allow a distributed network exploration procedure that can map the entire network topology without the use of any predefined hardware addresses.

3.8. Network Topology Exploration

One of the operational considerations of a communication network based system is that of maintaining a map of the topology and uniquely addressing the nodes of that network. The conventional approach is to document the network topology and assign node addresses (ID's) manually. The documentation is essentially the wiring plan that is kept in a file or on paper separately from the actual implementation. The node-ID is usually a natural enumeration scheme related to the actual topology. Nodes receive their ID during the bootstrap procedure. This ID is derived from the network documentation or hardcoded into the nodes (jumpers, EPROM, etc.).

This section describes a distributed procedure that does not depend on either an external representation of the network topology or on any method of hardcoding. Instead, the actual existing network is used as the sole source of the topology information. Furthermore, nodes will be uniquely identified without the help of any preassigned addresses. This bootstrap procedure has the advantages of being immune to documentation errors and of being able to adapt to network changes (channel failures, network extensions, reconfigurations).

3.8.1. Intra-Cluster Bootstrap Procedure

Bootstrapping starts within each cluster. Inter cluster channels are explored only after the topology, routing tables and node-ID's are established within the cluster.

The details of the bootstrap procedure will depend on the actual processor architecture in each node, which is beyond the scope of this thesis. It will be assumed that each node starts executing a bootstrap program after a system reset operation.

There are several methods to initiate this bootstrap program. A common approach is to store the bootstrap code in a small read-only memory (ROM) in each node. Since the ROM content survives power losses, each node is always ready to start. Avoiding the complexity of a ROM per node, each processor may enter a halt state after reset. In this case, the communication system must provide facilities to access the local memory and vital control registers through the network. For example, special diagnostic packets might be interpreted directly by the communication system without processor intervention. Once a minimal program is loaded into the memory, the processor can be started through network access of an appropriate control register. This method is feasible even without any knowledge of the network topology because it is always possible to perform a broadcast operation: a node that receives the *breath-of-life* message will simply forward it on all available outbound channels. Any node that is already alive will ignore such messages. Eventually, all reachable nodes will be alive. Naturally, there has to be at least one node within the network that is able to boot without outside help, say by means of a ROM and/or other peripherals.

The broadcast approach described above can be used to identify each node in the network, provided that exactly one node can boot autonomously. This node becomes the root of a broadcast tree and all other nodes are identifiable by the path of their breath-of-life message.

The remainder of this section will deal with the more challenging situation of multiple autonomous nodes: potentially two or more nodes will spring to life and will revive their neighbors. The only assumption made is that all nodes within the network will start executing the same bootstrap code within a finite amount of time after the system reset. The first step - initial program load of all nodes - may use ROMs, broadcasts or other means.

The bootstrap procedure is based on the following assumptions:

- Each processor can execute a normal, sequential program and has a certain amount of local memory. During the bootstrap procedure, this memory is private to the node.
- The processor controlling each node is able to control the attached message system. This control is used to disable all automatic routing functions and to send or receive packets directly by polling appropriate control registers. These facilities are also necessary for hardware debugging, error recovery and system reconfiguration.
- All processors will start executing the same program within a finite amount of time after reset. Initial program distribution though the network will leave the message system in a clean state, with all channel buffers empty.
- Execution of the bootstrap program need not preserve any synchronicity between different nodes. While the message system operates synchronously and all processors operate on the same clock, program execution is asynchronous. There are factors, such as dynamic memory refresh and the initial program distribution that do not preserve synchronicity.
- Communication channels are bidirectional.
- Communication channels that are unused or connected to inoperative nodes are recognized by the channel controller.
- The processor is able to read the master/slave bits of the intra-cluster channel controllers. This assumption is merely used to reduce the bootstrap time but is not essential to the principal algorithm.

All routing will be done under direct program control. In particular, it is possible to send multi-packet messages by virtue of cooperation with the remote processor in a store-and-forward fashion.

Given this system, it can be shown that no deterministic procedure exists to achieve consensus or to elect a leader [59]. Designating exactly one node as leader (root, anchor node, etc.) is a necessary first step. Even accessing the master / slave bits of the communication channels is not sufficient to break the symmetry of the system[39]. A deterministic protocol based on the master / slave bits is possible if the communication channel ports for each node are ordered (numbered) and if the topology follows certain rules for the channel assignments, for example, each cycle of the topology must have at least one channel with equal port numbers on each end. However, such procedures are complex, don't tolerate wiring errors and may constrain the topology selection.

Instead of relying on special properties of the network topology, the bootstrap procedure is based on a probabilistic protocol. The probabilistic approach is simpler and more general. The disadvantage is the lack of an upper bound on the time required to explore the network. It is possible that time to explore the network exceeds any given bound, but the probability of this event is an exponentially decreasing function of the allotted time. In practice, the time spent on the probabilistic component is a tiny fraction of the overall bootstrap time. This problem is quite similar to the decision process which generates the master / slave bits of the channel controller.

Any probabilistic protocol requires that each node has access to an independent source of random numbers. A pseudo random number generator is not sufficient because there is no seed guaranteed to be unique throughout the system. Therefore, each node needs a true source of random bits. True random bit generators have been included on integrated circuits [85], require little space and may supply one - otherwise unused - bit of status register within the message system.

The bootstrap procedure works on the principle of kingdom expansion: Initially, each processor forms its own kingdom. Subsequently, each king strives to conquer the world by taking over adjacent territory. Eventually, there will be one kingdom that covers the entire cluster.

The topology of one cluster is represented by a directed graph $G:(V, E \subseteq (V \times V))$. The directionality of G originates from the master / slave bits and does not imply a restriction of the information flow. A kingdom $K_i \subseteq V$ has exactly one king $v_{k(i)} \in V$. Obviously $i \neq j \rightarrow K_i \cap K_j = \emptyset$ and $\cup_i K_i = V$.

Let $U_i^m = \{ e_j \in E \mid v_{\alpha(j)} \in K_i \}$ be the set of unexplored channels of the kingdom K_i that have the master / slave bit set. Likewise, $U_i^s = \{ e_j \in E \mid v_{\beta(j)} \in K_i \}$ is the set of unexplored channels

[39]Consider a two-processor system with two channels. Since the master / slave bits are initialized randomly, they can result in a completely symmetric configuration.

with the master / slave bit cleared. The unexplored channel sets are maintained by the *king* (i.e. root node) $v_{k(i)}$ of each kingdom. The root node also maintains a map of the topology and deals with all messages sent from or received by K_i.

Initially $K_i = \{ v_i \}$ and $U_i^m = \{ e_j \in E \mid \alpha(j) = i \}$, $U_i^s = \{ e_j \in E \mid \beta(j) = i \}$ so that $\cup_i U_i^m = \cup_i U_i^s = E$. All channels are unexplored at this point.

After initialization, all nodes begin to execute two concurrent tasks: *send* and *receive*. Concurrency is achieved by simple co-routines driven by a loop that polls the status register of the message system to see if any messages arrive or if a channel is able to send a message. The number of iterations of this poll loop is used as a measure of time. Both tasks share one semaphore: *busy*.

The *send*-task is essentially a loop over the elements of U_i^m:

```
for (e_j ∈ U_i^m) {
    wait(random());     /* wait a random amount of time */

    P(busy);
    send_merge_request(e_j);
    if (receive_reply(e_j) == ACCEPT) {
        add(K_x : v_β(j) ∈ K_x);
        update_map(e_j);

        remove(e_j, U_i^m);

        remove(e_j, U_i^s);
    } else if (self(e_j)) {
        update_map(e_j);

        remove(e_j, U_i^m);

        remove(e_j, U_i^s);
    }
    V(busy);
}
```

The loop selects an unexplored channel $e_j \in U_i^m$. After waiting a random amount of time that was drawn uniformly from a fixed interval, the critical region guarded by the *busy* semaphore is entered. Once inside the critical region, a merge request is sent over e_j. Since U_i^m contains only channels that are connected to a viable node and because all viable nodes will eventually execute the bootstrap code, a reply is assured. There are two possible replies: *accept* or *reject* the merge request. If the merge request was accepted by K_x, K_x will cease to exist. $U_x^{m,s}$ will be added to $U_i^{m,s}$ and all nodes in K_x will become part of K_i. Likewise, all topological information of K_x is added to the maps of K_i. e_j is no longer unexplored and is removed from U_i^m.

A *reject* reply is either due to the remote kingdom being busy or due to an attempted self-merge. If $v_{\beta(j)} \in K_i$ a reject is assured because the root node remains in the *busy* state until a reply arrives and *accept* replies are returned only in the non-busy state (see below). This leaves the problem of recognizing that the reply originated from $v_{k(i)}$. The *self()* function performs this test by comparing the path of the reply message to the reversed paths of *reject* messages generated since the send-task entered the critical region. Since the normal addressing facilities of the message system are not yet functional (due to the lack of valid addresses and routing tables), addressing of messages is done by listing the channel numbers for each traversed node as part of the message. Hence, nodes are able to send a reply by reversing the sequence of channel numbers. Given the relative path of a message through the network (i.e., sequence of channel numbers) and the identity of one node along the path, all nodes of the path are uniquely identifiable. The recipient of a reply message knows its own identity (i.e., destination of the message path). Likewise, the originator of a reply message knows its identity (i.e., source of the message path). Therefore, if the *self* function locates a reply that has the same path as the received one, it concludes that the source and destination nodes are identical ($= v_{k(i)}$). Hence all nodes along the message path are in K_i and exactly one channel of this path was unexplored. This channel is added to the topology maps and removed from the unexplored channel lists.

Channels leaving K_i that yield a reject message will stay in the set of unexplored channels and will be retried in a subsequent iteration.

The *receive* task is the counterpart to the send task and takes care of replying to merge-request messages:

```
while (U_i^s ≠ ∅) {
    message = receive_merge_request();
    if (P(busy)) {  /* Non-blocking P() */
        reply_accept(message);
        send_state();
        terminate();
    } else
        reply_reject(message);
}
```

As long as there are unexplored inbound channels, the receive-task will wait for merge-request messages. Once a merge-request message is received, a non-blocking attempt is made to acquire the busy semaphore. Upon success, an accept message is returned. Subsequently, the local state (the partial topology map and $U_i^{m,s}$) is sent and the node terminates the execution of the send and receive tasks. From now on, incoming messages are simply forwarded to the remote root and messages from the root are routed to the specified channels. Essentially, the node becomes a simple router controlled by the remote root node. If the busy semaphore is not available, a reject message is returned.

The final and unique root node is distinguished by having both the send and receive task terminate by exhausting $U_i^{m,s}$.

This bootstrap procedure is correct if:

1. All channels will be explored upon termination.
2. Exactly one kingdom will eventually cover the entire cluster.
3. The procedure terminates.

Since termination depends on the set of unexplored channels being empty, the first assertion is trivially true.

Assuming more than one kingdom after a completed bootstrap procedure implies the existence of at least one node v_x that is not part of the arbitrarily chosen, final kingdom K_i. Because the network is strongly connected, there is a path from $v_{k(i)}$ to v_x, hence $\exists e_j \in E : (v_{\alpha(j)} \in K_i \wedge v_{\beta(j)} \notin K_i) \vee (v_{\alpha(j)} \notin K_i \wedge v_{\beta(j)} \in K_i)$. Therefore, e_j must have been in either U_i^m or U_i^s. In the first case, a merge-request message was sent outbound over e_j. This implies either an accept reply or a successful self-test. Either possibility would have resulted in $v_{\beta(j)} \in K_i$, which contradicts the selection of e_j. In the second case, a merge-request message was received by $v_{\beta(j)} \in K_i$. The only way for the issuing root-node to terminate is by receiving an accept-reply from a node within K_i, which contradicts $v_{k(i)}$ successfully terminating as a root node.

Being a probabilistic construction procedure, no termination is guaranteed within a bounded amount of time. For $n = |V|$ nodes, a total of $n-1$ remote merge-requests must succeed before termination occurs. Each of these can fail with probability $q = \frac{s}{s+r}$, where s is the average round trip time for a message across the network and r is the average wait period in the send task. All other contributions to the boot time are a linear functions of the number of nodes and channels in the network. The probability $P_{fail}(m)$ for failure to terminate after m remote merge-request can be computed if q is assumed to be constant. Given that the wait periods are randomly drawn from a fixed interval, r is simply half of that interval. s will increase as the kingdoms expand. However, s primarily depends only on the topological distance and does not change with the number of failed attempts. Hence it is reasonable to assume a constant q, which leads to:

$$P_{fail}(m) = \sum_{i=0}^{n-1} \binom{m}{i} q^{m-i} (1-q)^i \tag{3.18}$$

A conservative estimate of the number of attempted merge-requests is given by the time allotted to boot, in excess of the time needed for the deterministic part, divided by the average time for one iteration of the send task ($= s + r$). This approximation does not take into account the possibility that multiple merge-requests could be attempted in parallel.

Figure 3-33 shows the non-termination probability for a 1000-node system. The average wait period is assumed to be equal to the average send-merge-request operation ($s = r$). A larger r causes more rapid convergence at the expense of an increase in the deterministic part of the bootstrap procedure. Assuming 1 msec per attempt (a rather conservative assumption), the probability for not terminating within 4 sec is about $4.2 \cdot 10^{-230}$.

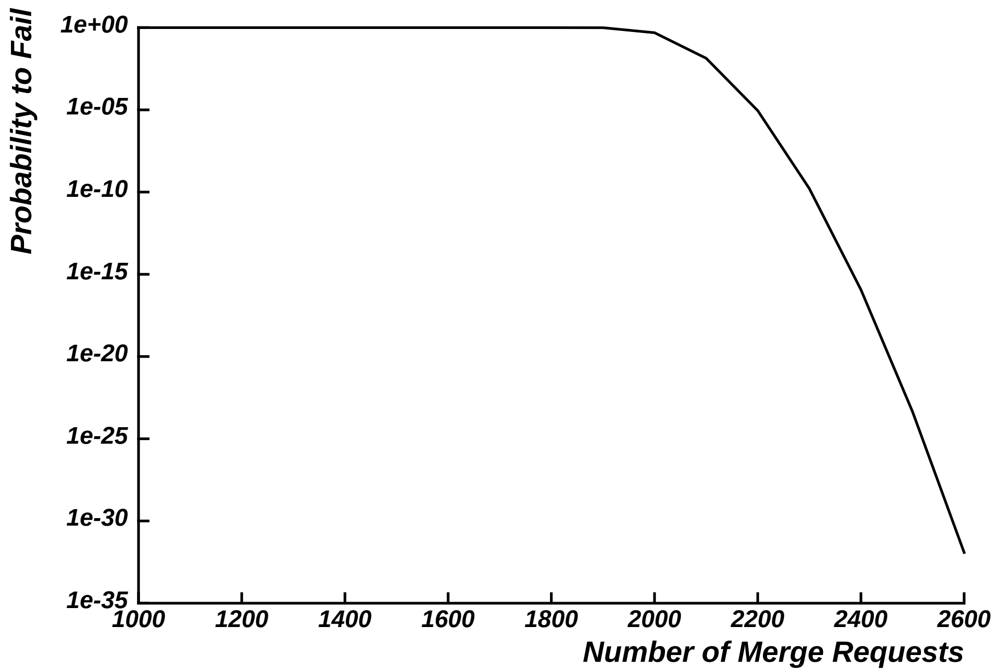


Figure 3-33: Bootstrap Failure Probability

3.8.2. Inter-Cluster Bootstrap Procedure

The inter-cluster bootstrap procedure is essentially the same algorithm. Clusters that completed the local bootstrap procedure try to capture all adjacent IFC's. The capture status of each IFC is visible by the *lock* semaphore. Successful captures cause the cluster to control the IFC. IFC'S are configured such that all unsolicited messages are forwarded to the controlling cluster. This requires continuous polling to be able to reply properly. It also allows a test to see if an IFC is really captured or if it is a repeater with a set *lock*-semaphore due to a missed reset[40].

Once a cluster encounters a captured IFC, communication is established to a potentially different cluster and the merge-request procedure is executed. The *signal* semaphore is used to decide which becomes the root-cluster.

Eventually the entire network topology is established. The root node computes the routing tables and assigns the node-ID's. After this information is distributed to all nodes, the automatic routing facilities become operational.

[40]The reset signal is only local to a cluster. Repeaters are reset only during power-on.

The assignment of ID's for a given topology may use a simple node enumeration. However, some applications are optimized for particular topologies and assume a relation between the distance between nodes and their ID's. This relation might be supplied explicitly by allowing access to the distance matrix, which was computed during the generation of the routing tables. Attempts to match the actual topology with an expected topology type (hypercubes, grids, etc.) can yield the natural enumeration if they succeed.

3.9. An Implementation Exercise

Designing and simulating an adaptive routing algorithm can lead to a system that is too complicated or cumbersome to actually be built. This section outlines an implementation to demonstrate that the proposed message system is well within the realm of current VLSI technology.

The core of the router is the data path based on the multi-ported register file approach outlined in section 2.3. There are essentially two ways to structure this register file: by using dedicated buffers or by using a uniform array of buffers. Dedicated buffers were used in the interface chip for inter-cluster channels described in Section 3.7.2. Part of the buffer array is connected to the receive channels while the other part is connected to the transmitters. This layout avoids the I/O crossbars at the expense of buffer-to-buffer copies over a parallel packet bus. While these copy cycles added little to the inter-cluster latency of the interface chip, they would require an entire transmission cycle in the router. A packet received in cycle n can leave the node no sooner than in cycle $n+2$. Moreover, the copy cycles reduce the I/O bandwidth to the local node. The bottom line is that a sequential or multiplexed register file will severely degrade the router performance.

The uniform buffer array approach is outlined in Figure 3-34. Each (horizontal) slice of the array can hold one packet. There are n channel controllers that separate the bidirectional intra-cluster channels into a receiver and transmitter. While the receivers and transmitters are logically independent, the intra-cluster protocol requires close coordination due to the sharing of the same set of bidirectional wires. The channel controllers do not have any packet buffer associated but rather store / fetch the data to / from the buffer array. Hence every channel needs its own path to the array. Access to the actual storage cells is provided by embedded crossbars (shaded areas) in the array.

A parallel packet bus is used to remove terminal packets from the buffer and to insert locally generated traffic. Besides the storage for the actual packet, each buffer has a certain amount of state and auxiliary information attached to it. The state includes the free / occupied information and the associated receiver or transmitter. The auxiliary information consists of the packet priority and an encoding of the set of viable outbound channels.

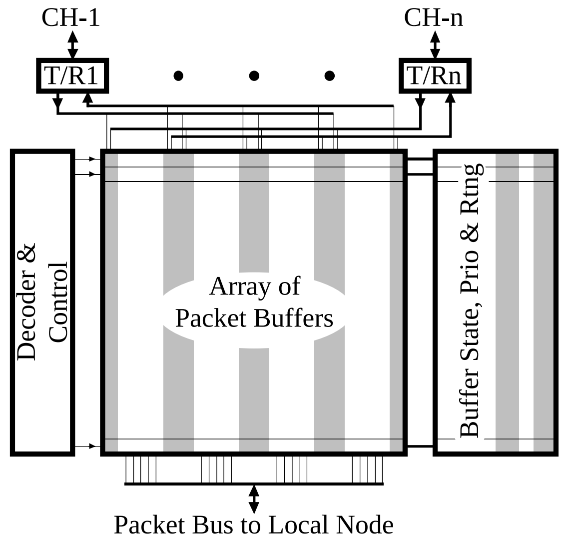


Figure 3-34: Router Data Path

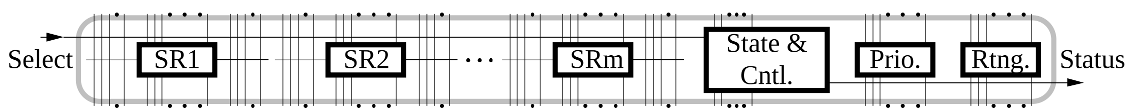


Figure 3-35: Packet Buffer Slice

Figure 3-35 is a block diagram of one entry of the packet buffer array. Each intra-cluster channel has m wires running in parallel; hence there are m shift registers ($SR1, \ldots, SRm$) in each buffer slice. The selection of m depends on the target system size, speed and number of available I/O pins. Guidelines for these design decisions were given in Chapter 2. Of practical interest are the cases $m=4$ and $m=8$.

The input of the first shift register $SR1$ can be connected to the corresponding wire of one of the n receivers. Likewise, the outputs of $SR1$ can drive the associated wire of one specific transmitter. Besides this serial access, the shift registers are readable and writable from the parallel packet bus. The length of the shift register (i.e., number of transmission cycles) is determined by the packet size in bits divided by m.

The buffer slice is selected by the decode and control logic. Selection is used to access a particular buffer from the packet bus. Each slice has a small amount of additional state information, for example to store which (if any) receiver or transmitter is attached to it or if it is currently occupied by a packet. To establish the association with a particular channel, each slice must also participate in an assignment and arbitration procedure which requires the routing and priority information. Obviously, it is not practical for each buffer to compute the routing and priority information locally. Instead, this information is added to the packet when it is entered into the array and stored in each slice in addition to the actual packet.

Several global control signals are supplied to the control logic associated with the slice state. These signals govern the buffer through the receiver/transmitter assignment process.

Serial access to a shift register is usually destructive to the stored data. However, transmitters must be prepared for a potential retransmission in case of a transmission error. The channel controller solves this problem by echoing the retrieved data on the receiver. This is possible because a channel is either receiver or transmitter but not both at one time. A further advantage is that the buffer slice doesn't need to distinguish between the receiver and transmitter associations. It simply stores the channel number and uses it to control both the input multiplexor and output driver. The signals to perform the shift operation are part of the global signals supplied to all controllers.

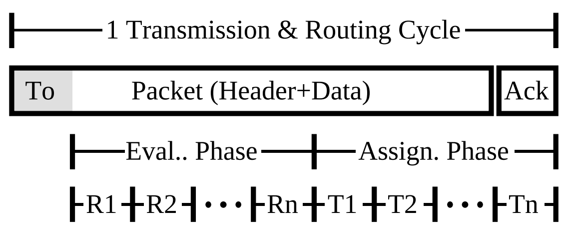


Figure 3-36: Routing: Sequence of Operations

Routing uses simple, static timing. The global control signals are oblivious to the presence of actual packets. Figure 3-36 illustrates the sequence of operations during each routing cycle. There are two phases to each cycle: the evaluation phase and the assignment phase. Each needs one cycle for each channel. The evaluation phase cannot start before the destination (*To-*) addresses of the incoming packets are received. During the evaluation phase, the routing and priority information is added to each packet. The assignment phase decides which packet is to be sent for each available transmitter.

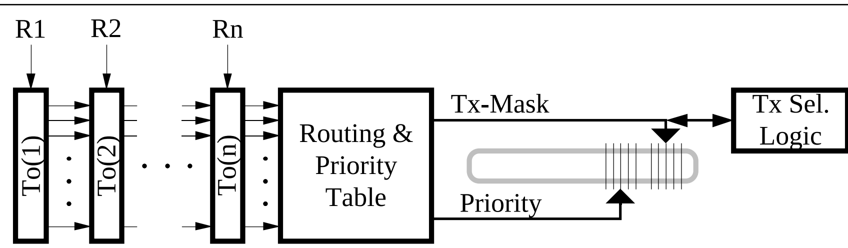


Figure 3-37: Routing Table Look-Up

Figure 3-37 is the block diagram of the routing table look-up unit. Each receiver feeds the destination address - which is transmitted first - to a dedicated *TO*-register. The *TO*-registers are not part of the buffer array, rather they simply listen to the receive - data path. Once loaded with destination addresses of the incoming packets, the *TO*-registers form a shift register that sequentially supplies these addresses to the routing table. The routing table supplies the transmitter mask which is a bit vector with a *'1'* for each potential outbound channel. The transmitter masks have $2n$ bits because of the two virtual channels provided by each physical channel. The routing table also provides a priority. The priority is part of the adaptive routing heuristic and favors packets that have few hops left to go. The priority of packets stored in the array that are not sent out during a routing cycle can change. This change is accomplished by the logic embedded in each slice (a simple counter) and is used to give priority to packets that have missed a transmission cycle.

During the evaluation phase, both the priority and transmitter mask buses are simply used to store data in the array.

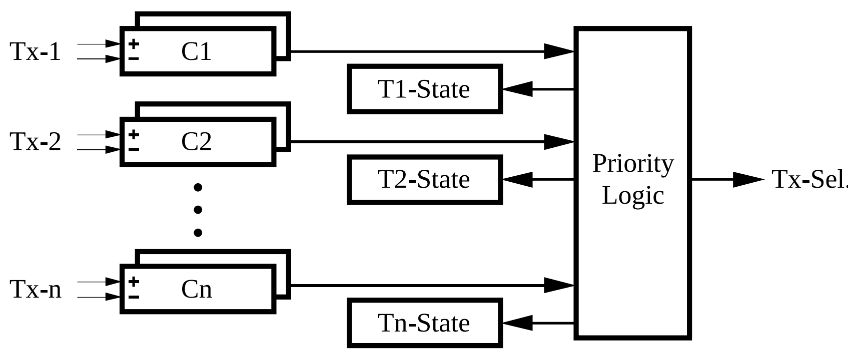


Figure 3-38: Transmitter Assignment Sequencing

While the routing table look-up scans the receivers in a fixed pattern, the transmitter sequence during the assignment phase depends on the traffic. Figure 3-38 depicts the logic controlling the transmitter assignment sequence. The routing heuristic calls for the transmitter with the fewest pending packets to be assigned first. Naturally, it is not possible to scan the array and count the number of applicable packets. Instead, this count is maintained incrementally. Whenever a packet is stored into the array, a counter for each virtual channel is connected to the transmitter mask bus such that a '1' on the corresponding wire increments the counter. Successful assignments will repeat this bit-vector at the end of the assignment cycle so that the counter will be decremented appropriately. Transmission errors could cause erroneous decrements, because a decrement took place without the associated packet leaving the buffer. This error is corrected in the next evaluation phase. Since there was a transmission failure, the corresponding channel did not receive a new packet, so there is a free slot in the evaluation phase that is used to assert the transmitter mask of the failed packet on the *Tx-Mask*-bus.

The transmitter state is part of the channel controller and is used to determine if the transmitter will be available in the forthcoming transmission cycle. Provided that the transmitter is available, the channel controller decides which virtual channel is going to be used. Normally, virtual channels simply alternate; however, if there is no traffic pending for one, the corresponding slot is skipped. Pending traffic is indicated by the state of the corresponding counters.

Available transmitters with pending traffic are presented to the priority logic which selects the one with the fewest pending packets. The output of the priority logic is fed back to the transmitter state to disable the winning channel from subsequent assignment phases.

An unary encoding of the selected transmitter is sent to the transmitter mask bus to enable the priority selection process. All packets that have the corresponding transmitter bit set will compete on the priority bus. The packet with the highest priority will store the transmitter bit. This completes the assignment of one packet to one transmitter. The selected packet gates its transmitter mask onto the *Tx-Mask*-bus, which is used to update the counters of pending packets.

Since there is only a small number of channels, the priority logic is a direct combinational circuit and is much faster than one assignment cycle. This leaves enough time to select a packet within the assignment cycle.

Ready transmitters with no pending traffic will also receive assignment cycles to allow for blind transmissions.

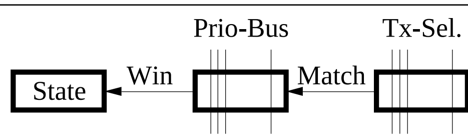


Figure 3-39: Packet Buffer Selection

The packet buffer selection starts by matching the routing bit-vector with the transmitter under consideration (Figure 3-39). A match enables a voting on the priority bus to select the buffer holding the packet with the highest priority. This is done one bit at a time, starting with the most significant bit (MSB). The priority bus is precharged to ‘0’. Buffers with a ‘1’ in the MSB will drive that wire to ‘1’. Buffers with a ‘0’ simply watch that bus-wire. If the MSB of the bus turns to ‘1’, the buffer logic detects the presence of a packet with a higher priority elsewhere and drops out of the selection process. The *Deep Thought* move-generator circuit uses this method for the move ordering [63]. That experience indicates that about 5nsec are needed per bit for this priority selection method[41]. Therefore, a 6-bit priority selection followed by a fixed priority on the buffer array can be used to uniquely select one packet buffer within one transmitter assignment cycle.

3.10. Performance

The simulation results presented in this section were obtained with CBS, which is described in the appendix. CBS simulates k-ary hypercubes that use either the new adaptive routing method or conventional E^3 routing with wormhole routing. Networks of up to 1500 nodes were simulated, a limit that is primarily due to the available physical memory (24 Mbyte). The length of the simulation runs was generally chosen to keep the statistical error below 10%. The exceptions are runs with a very high, variable load. Due to practical limits on the simulation time, the last data point presented on load graphs has a statistical error of up to 25%.

3.10.1. The Simulation Model

Independent of the routing strategy (adaptive or wormhole), two principal modes were explored: asynchronous and synchronous operation. While the proposed architecture operates synchronously, experiments were run to determine the impact of asynchronous circuit design.

In synchronous operation, all channels use the same amount of time and nodes execute their application at the same rate. Consequently, all routers will redistribute the incoming traffic at the same time. If a node receives several packets in one cycle, these packets will become available simultaneously.

In asynchronous operation, the transmission time of each channel varies slightly. The average speed of all channels throughout the network is the same as in the synchronous case, but the speeds of individual channels vary. The channel speed distribution is gaussian with a variance of 5%. In addition to this static speed variation, each individual transmission over a channel will require a varying amount of time. This dynamic channel speed variation follows a gaussian distribution with a variance of 2%. The static channel speed variation models different component speeds of an actual implementation. The dynamic speed variation models the influence of noise, temperature differences, power supply fluctuations, etc. on an asynchronous design. All these factors result in some amount of jitter in asynchronous circuits. Neither effect will cause transmission errors. Only the system speed

---

[41] This estimate is based on a conservative design using 2μm CMOS technology and MOSIS design rules.

will be slightly perturbed. The net result is a system with no fixed phase relation between the routing cycles of different nodes.

The E^3 wormhole router uses two virtual channels to avoid deadlocks. Messages are subdivided into flow control units (flits) such that a flit can pass through a node in one routing cycle, which is equivalent to the transmission time of one flit across one channel. The first flit establishes the route for subsequent transmissions. This virtual circuit is terminated with the passage of the last flit. Upon arrival of the first flit, a routing connection is established within one routing cycle according to the E^3 policy. If the required channel is busy, the connection request will block until the outbound resource becomes available. The assumption that the routing decision requires no more processing time than the passage of a mere data flit is overly optimistic. Furthermore, real systems are likely to require two pipeline stages instead of the one in this model [129].

The adaptive router operates on individual, fixed-size, minimal length packets. Each packet carries its own destination address and is handled as an independent, atomic unit of data. Packet transmissions require one cycle. Longer messages are dissolved into multiple packets at the originating node and are reassembled into one message at the destination. Sequence numbers are used to ensure correct order. Each node has four packet buffers for transient storage to avoid blind transmissions.

The routers are connected to the processing nodes with two queues that can hold up to 6 messages. Send operations on a full queue will block. The receive queue is emptied by the application process that will remove incoming traffic at a constant rate. Unsolicited messages are processed by the application process. While multiple arrivals can momentarily fill the receive queue, the application process is fast enough to ensure prompt removal of terminal traffic.

Message latencies are measured from the point a send function call is executed until the completed message is consumed at the destination with a matching receive function call. Packet latencies are measured from the time the packet is injected into the network until the time it is consumed by the message assembler at the destination. All times are expressed in multiples of the basic system cycle time, which is the time for the transmission of one packet or flit over a point-to-point channel.

Delays introduced by the application process are small compared to the network-induced delays. This is an optimistic assumption which is justified by the objective to measure the network performance. Delays introduced by the interfaces between the network and the user program executing on each node are the subject of Chapter 4.

3.10.2. The Traffic Model

CBS supports the conventional send/receive style of message-based private memory multiprocessors. Because there are numerous commercial and academic machines of this type in use, a fair amount of actual programming experience is available. It was observed that there is a certain set of communication patterns that frequently occur in different applications [46]. Some communication patterns are so common that they are directly supported by the operating system, for example various forms of grid embeddings [101]. Unfortunately, no statistic was available to single out the most common traffic pattern. Given such a statistic, a communication benchmark could be constructed based on the observation that the system performance of typical applications can be approximated by a linear combination of the performance of the characteristic components [43]. Lacking such a standardized communication benchmark of *Dhrystone* flavor, a set of six important traffic patterns is used to drive the simulations:

1. The *uniform* address distribution is the simplest and most studied traffic pattern. The destination of a message is randomly drawn from all nodes.

2. The *hot-spot* traffic pattern is derived from the uniform distribution by selecting a specific node. An x% hot-spot will receive x% of all messages. All other messages are uniformly distributed over the remaining nodes. The hot-spot pattern is notorious for degrading the performance of shared memory multiprocessors based on omega style multi stage switches [113].

3. The *normal* traffic pattern is tuned for recursive networks like hypercubes. The pattern for an n dimensional network cycles through n phases. Each phase divides the network in two equal sized sets. The nodes of one set exchange data with the corresponding node of the other set along a particular dimension. The sequence of dimensions is predefined. Normal communication is a versatile method to combine local information to a global structure, distribute global data to local nodes, perform global operations like minimum, maximum, enumeration. Various forms of broadcasting and routing are efficiently performed via normal communication [47]. Normal communication is closely related to parallel divide and conquer algorithms.

4. The *transposition* traffic pattern assumes a square number of processors that are logically organized as a two-dimensional matrix. The transposition pattern involves communication between pairs of nodes that are symmetric with respect to the major diagonal. Like hot spots, transposition traffic patterns are hard test cases.

5. The *FFT* traffic patterns use k input butterflies in each node for a k-ary cube. Fast Fourier transformations are easily coded as a normal communication operation on most logarithmic diameter networks. In order to provide a more interesting test case, the *FFT* pattern for k-Shuffles is used for a hypercube. This deliberate algorithm to topology mismatch increases the message distance from a nearest neighbor communication to one with a $\frac{diameter}{2}$ average message distance.

6. The *bit-reversal* pattern is the last stage of a FFT operation. Its properties are similar to those of the transpose and FFT pattern. All three patterns stress the network in certain, non-uniform ways.

The normal, transposition and bit-reversal patterns are all bit-permute-complement permutations (BCP), a general class of traffic patterns that covers most regular algorithms [98]. All traffic patterns have an average message distance of half the network diameter, with the exception of the normal pattern, which has a uniform distance of one.

Message lengths are either held constant or use a negative exponential distribution.

The dynamic behavior of the node process is approximated with three simple models: uncoordinated-variable, coordinated-variable and coordinated-fixed. Uncoordinated node processes proceed at their own pace. In this case, computing and communication phases alternate, and the phases of any two nodes in the network are unrelated. In the coordinated node model, the phases of all nodes throughout the network are coupled. All nodes will start communicating at the same time. This behavior is typical for barrier synchronized parallel programs. Nodes may either spend a fixed or variable amount of time computing. Variable times follow a negative exponential distribution. The amount of computation is varied to achieve different network loads.

3.10.3. Latency Distributions

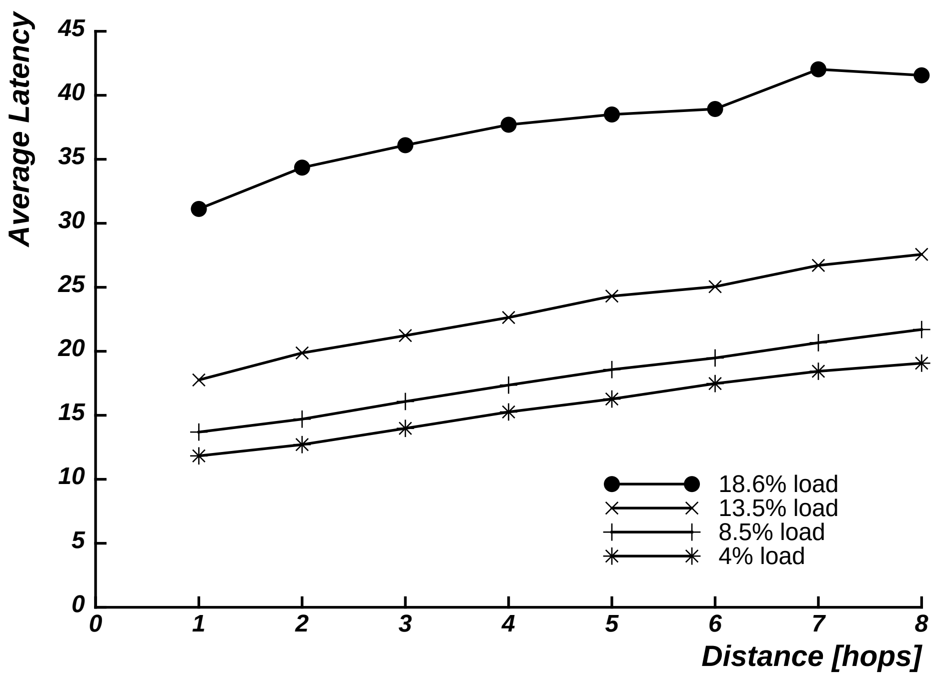


Figure 3-40: Latency vs. Distance (Wormhole Routing)

Figure 3-40 shows the average latencies for messages of length[42] 10, random traffic and

[42]The message length is expressed in the number of transmission cycles that are required to send the message to a neighbor node.

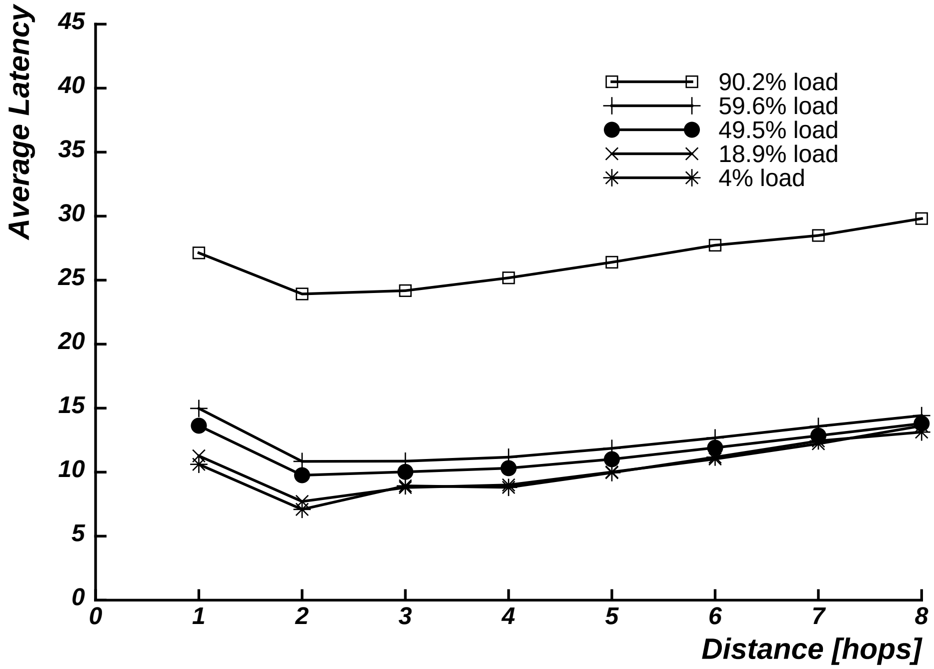


Figure 3-41: Latency vs. Distance (Adaptive Routing)

variable, uncoordinated node processes on an eight-dimensional, binary hypercube. Unlike store-and-forward packet switching, the latency increases slowly with message distance. The slope of the graphs is largely determined by the ratio of the message length to the header length. Once the header is routed through the network, transmission delay is simply proportional to the message length.

Figure 3-41 uses the same model, but adaptive routing of independent packets is used. It shows the effectiveness of using multiple channels in parallel. As the message distance increases, more independent paths through the network become available. While each packet is routed in store-and-forward fashion, the aggregate latency does not increase linearly with distance. In fact a reduction in latency is visible for 1-hop vs. 2-hop messages, which is partially due to the fact that 1-hop messages have no alternative paths.

While most parameters for Figures 3-40 and 3-41 are equivalent, the network load is not. The data for Figure 3-41 uses a much higher network load.

3.10.4. Blind Transmissions and Buffer Utilization

As described in Section 3.5, the adaptive routing strategy will send packets in an unfavorable direction if there are no other options. Furthermore, each router has 4 buffers for transient traffic.

Network Load:	4%	18.9%	59.6%
Blind Transmissions:	0.44%	0.84%	2.26%
Buffer 1 use:	1.56%	7.83%	18.5%
Buffer 2 use:	1.03%	3.78%	8.33%
Buffer 3 use:	0.55%	1.81%	4.05%
Buffer 4 use:	0.22%	1.04%	3.26%

Table 3-7: Blind Transmissions and Buffer Utilization

Table 3-7 lists the number of blind transmissions as a fraction of all useful transmissions. The data corresponds to the simulations for Figure 3-41. The variable, uncoordinated node process model causes a highly fluctuating network load. Data observed from runs with a smaller message size (for example traffic of a virtual shared memory system) indicates that Table 3-7 represents a demanding load. The typical number of blind transmissions stays below 1%.

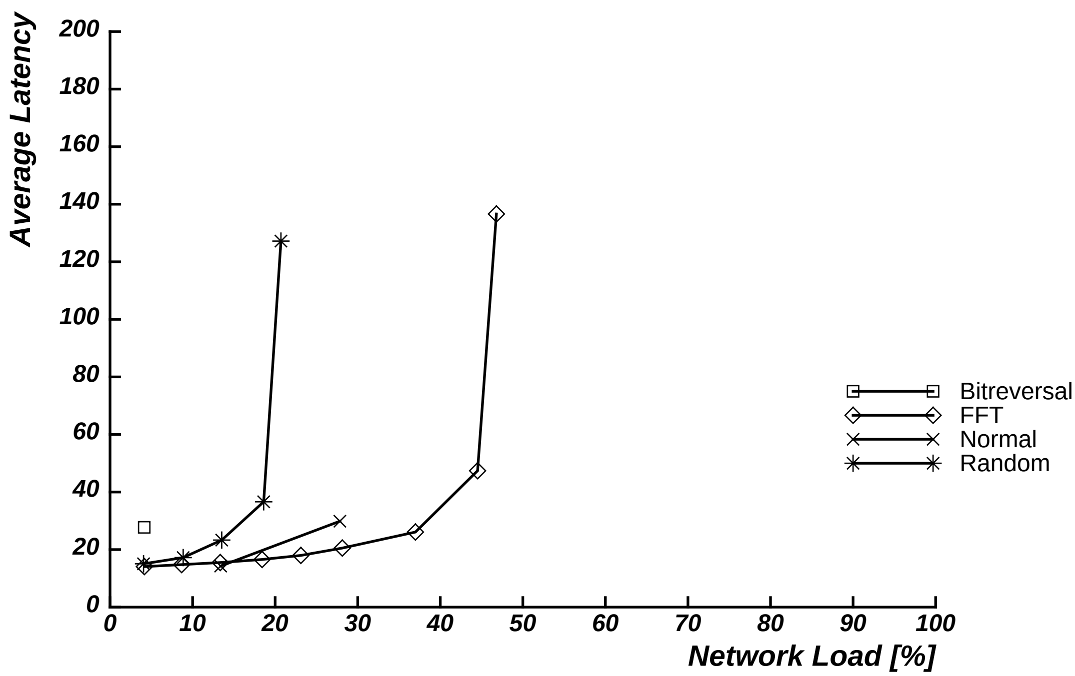


Figure 3-42: Latency vs. Network Load (Wormhole Routing)

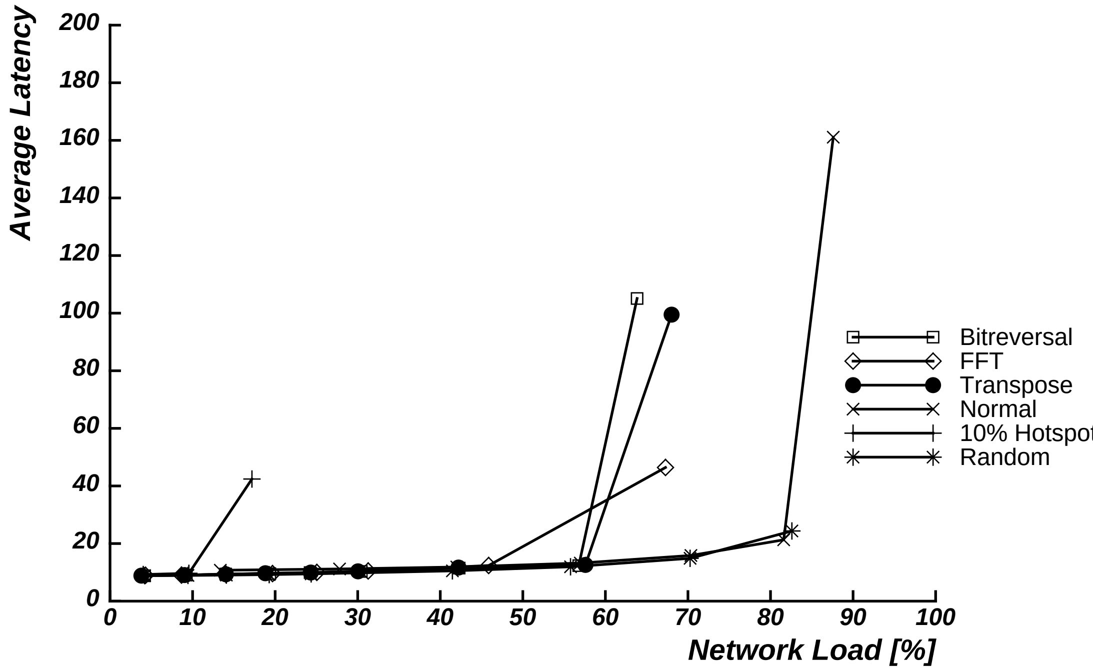


Figure 3-43: Latency vs. Network Load (Adaptive Routing)

3.10.5. Impact of Asynchronous Hardware

Simulations that used the asynchronous hardware model generally performed slightly worse than the synchronous counterpart. In the case of wormhole routing, the degradation was barely visible, considerably below the expected statistical error. Some degradation is expected because transmission paths through multiple channels run at the speed of the slowest channel in that chain and not at their average speed. This effect is best described as insignificant.

In the case of the adaptive router, the effect was more pronounced. To avoid synchronization losses in each node, routing cycles were initiated as soon as traffic arrived. This method performs reasonably under light to modest network load. At high network loads, each routing cycle deals with relatively few packets. Since packets arrive asynchronously, with most causing a routing cycle, there are more routing cycles than in the synchronous case. This leaves fewer resources for each cycle and less opportunity for traffic optimization. The net result is a degradation in the quality of the routing decision which leads to more contention.

Adding a waiting period before each routing cycle to collect more traffic and resources for a better match helps for high load traffic at the expense of a latency increase for light loads.

3.10.6. Latency vs. Network Load

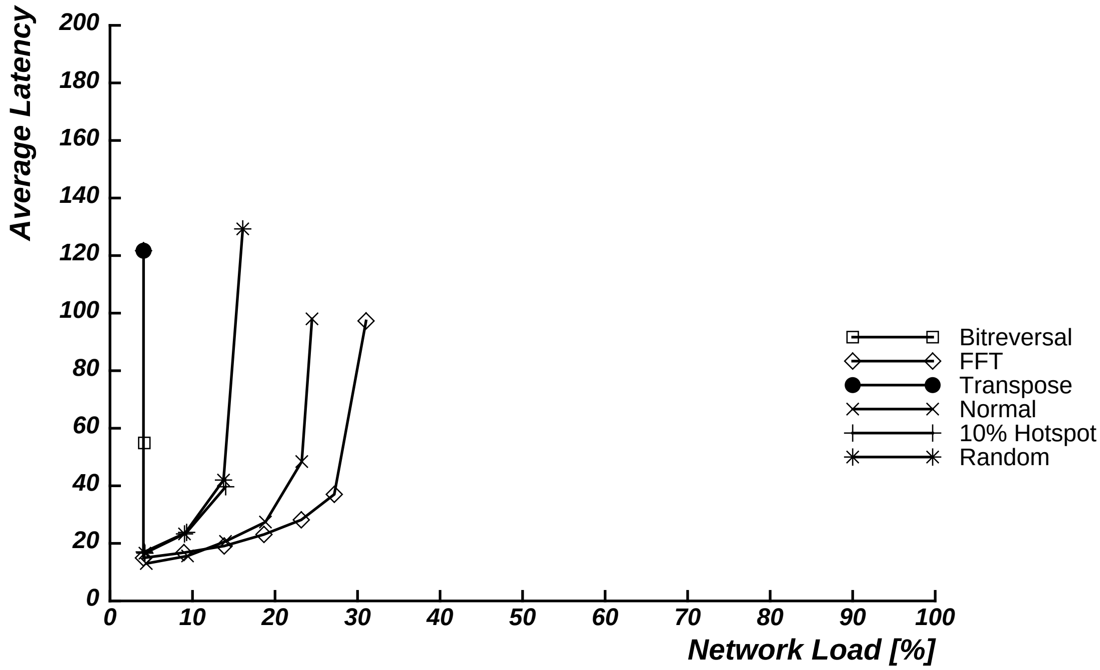


Figure 3-44: Latency vs. Network Load (Wormhole Routing)

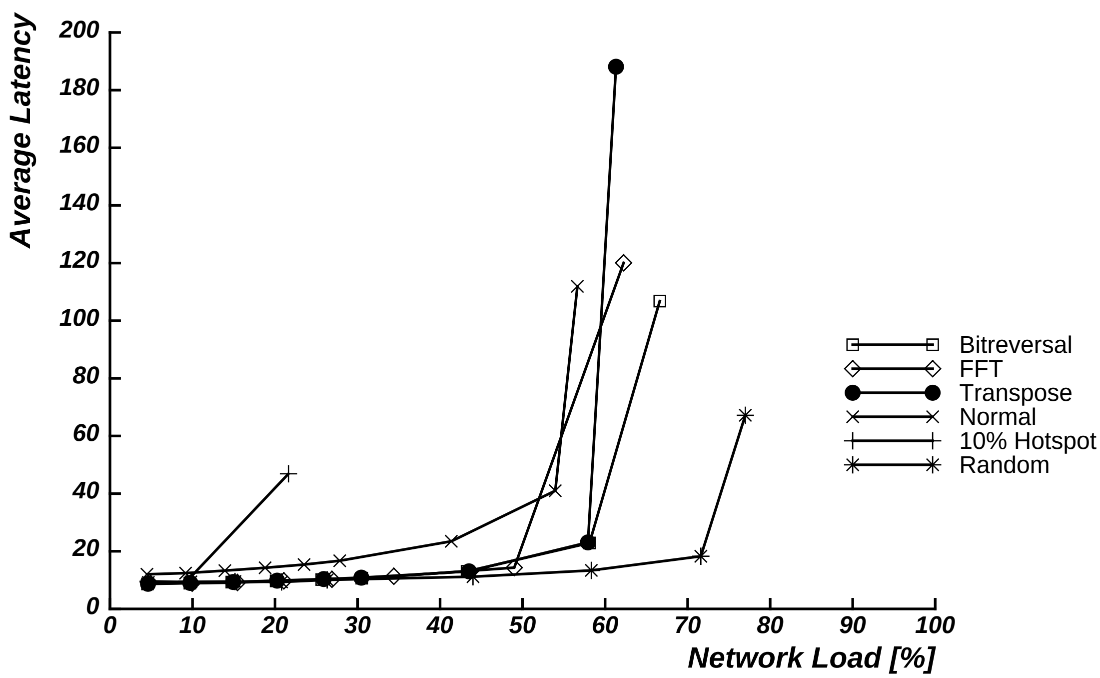


Figure 3-45: Latency vs. Network Load (Adaptive Routing)

Figures 3-42 through 3-45 use the variable, uncoordinated process model and are based on 256-node binary hypercubes. Since the adaptive routing works best with high-dimensional networks, binary cubes were used. The network load is defined as the fraction of topological bandwidth that is actually utilized. The topological bandwidth depends on the message distance distribution and was computed for each traffic pattern in use. A network load of one would indicate the best possible use of the communication resources. The network load was changed by adjusting the compute time parameter of the node process model.

A constant message size of 10 units is used for Figure 3-42 and 3-43. E^3 routing cannot handle even modest loads of the transposition and hotspot traffic types. A 10% hotspot for such a large system is a pathological case. The FFT traffic happens to minimize contention in the E^3 framework, even though a non-normal version of the FFT was used. The adaptive routing can sustain up to four times higher loads and performs better on pathological traffic patterns than wormhole routing.

Given a 10% hotspot, network loads in excess of 15% will cause unacceptable latencies. However, this averaged latency is dominated by the long delays for the traffic addressed to the hotspot. Unlike omega networks, traffic to other nodes sees only a small degradation. This is not necessarily a feature of the adaptive routing because similar observations were made on conventional hypercube machines [14].

Variable-length messages with negative exponential distribution (Figures 3-44 and 3-45) cause a more variable traffic that increases latencies. Both routing approaches degrade in roughly same proportions.

Figure 3-46 gives an overview of 64 node simulations. Only the adaptive routing results are presented. Variable-length messages uniformly degrade performance. Part of this degradation actually occurs in the insertion stage of each node: A long outbound message causes subsequent small traffic to wait until the first message is completely sent. While multiple packets can enter the network at once, each node can process only one message at a time. The simulator provides an input queue between the node process and the network.

3.11. Summary

The design of a message system that is optimized for a high network load of short messages. Unlike conventional systems, messages are broken up into small, fixed-length packets that are routed individually was presented. The increased cost for multiple packet headers is offset by the ability to use all independent network paths in parallel. Combined with a new adaptive routing scheme, useful network loads in excess of 90% were achieved. Network performance is further aided by synchronous packet transmission that gives each

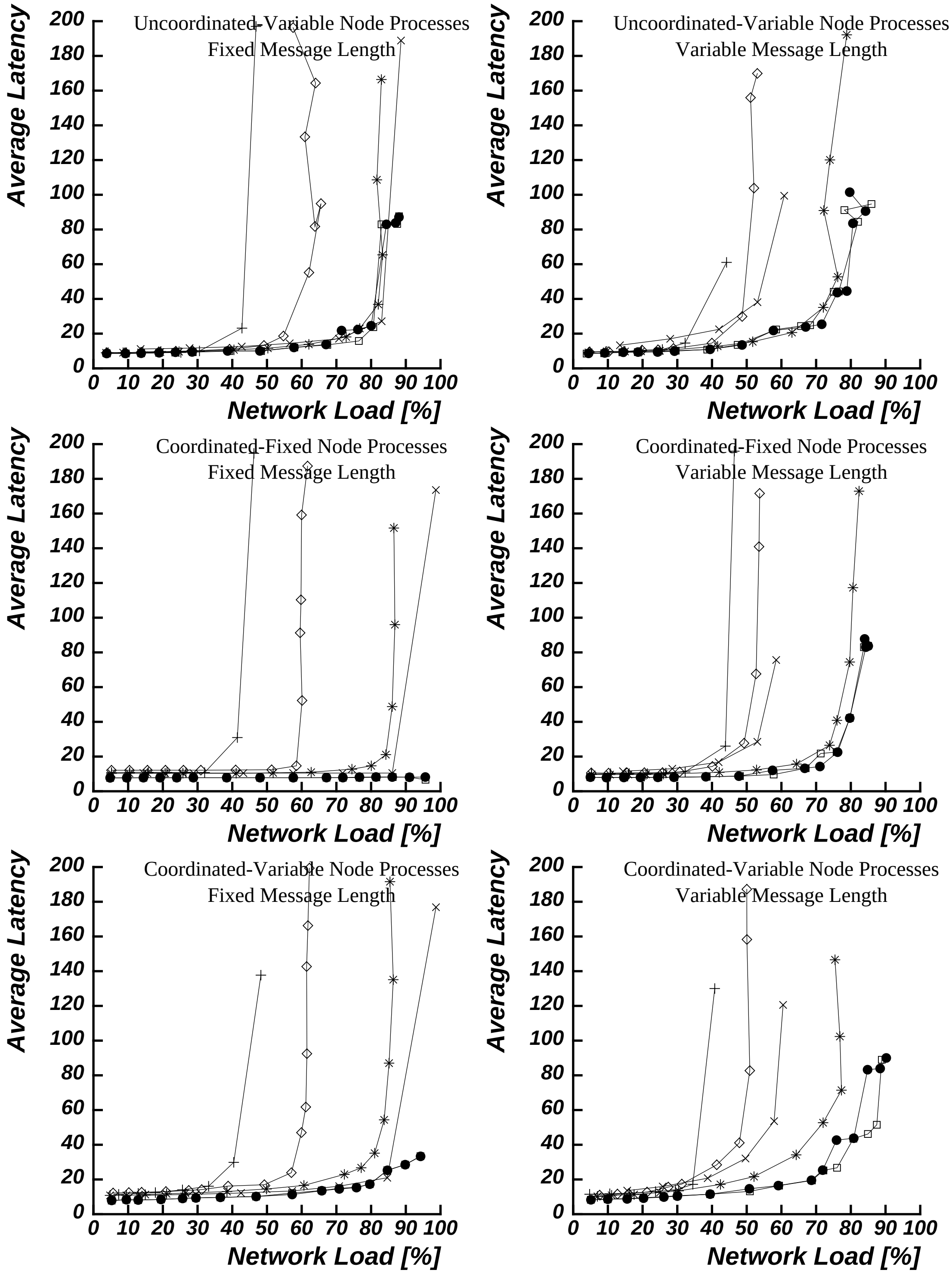


Figure 3-46: Latency vs. Network Load (Adaptive Routing)

node optimal control over the communication resource allocation and avoids synchronization losses. Short, fixed-size packets lead to a simple and efficient hardware implementation.

Adaptive routing in the context of communication networks for tightly coupled multiprocessors is normally based on a greedy search for channels and buffers. The new adaptive routing strategy outperforms simple greedy search because resource selection favors options that are least likely to constrain other traffic.

Deadlocks are avoided by resource ordering. Physical node-to-node channels were organized into two virtual channels. Each node assigns virtual channels based on the destination addresses of outbound packets. Routing and virtual channel assignment are table driven to be topology independent. A heuristic to compute deadlock-free virtual channel assignments that can handle most common network topologies was developed.

The presented clock distribution method can provide a well define clock within a cluster of processors. It is also able to maintain a coherent clock over a distributed network of clusters.

Extending the message system over a local area is possible with an interface circuit that translates the fast intra-cluster channels to signals suitable for medium distance transmission over inexpensive media. A system was outlined that links multiple processor clusters in a flexible and transparent manner.

Chapter 4
A Communication Architecture

A communication architecture fills the void between a network capable of transporting messages between processor nodes and the programs running on each node. Initially, when ensemble machines were conceived, the latencies of the early network implementations were so high that the communication architecture had few options, all amounting to forms of explicit data transfers. Given delays in excess of 1 msec., the overhead of explicit send and receive functions was small compared to the actual data transfer. With the advent of fast networks, the overhead of encapsulating data into messages became significant.

Traditional communication architectures were made more powerful by using dedicated processors to deal with communication related tasks, such as message queuing, initiating data transfers, and managing memory. The services provided by the communication system became more elaborate: messages are typically typed and can be addressed to any process throughout the system. Other services include remote procedure calls, I/O functions, object-oriented programming support, data format conversions, etc. It appears that the logical continuation of this approach will move more operating system functions to the communication processor. Ultimately, a message-based operating system will reside in these dedicated processors.

Given a fast message system, it is possible to provide a form of distributed shared memory (DSM). DSM architectures do not have physically shared memory that is accessible from all nodes. Instead the appearance of shared memory is achieved through a combination of address translation, memory management, and caching. This approach is potentially much simpler because the communication architecture has to deal with a small, well defined set of operations. From the programmer's point of view, shared memory programs are easier to write, debug and port [10].

This thesis will develop a simple, efficient communication architecture based on the shared memory model. Hence, the discussion of message-based communication architectures will be abandoned after a brief overview of alternatives in section 4.2.1. However, it will be pointed out that message-based systems can be implemented on top of the shared memory approach. Such a system is quite efficient if the message structure is laid out to match the underlying implementation.

The proposed communication architecture is based on the weakly coherent shared memory model. All processors are allowed to access memory objects in one global address space. The actual memory is composed of the private memories of the processing nodes. While access to memory is functionally uniform, access times depend on the actual location of the memory object. Access to memory objects stored locally does not differ from normal access to the private memory of an ensemble machine. Access to memory objects stored remotely will cause appropriate messages to be sent through the network.

This chapter starts with a discussion of the design objectives and rationale. The following review of related work includes a brief discussion on the virtues of conventional communication architectures. A broad overview of the entire system is intended to provide the context for the detailed technical discussion, which starts with the memory coherence problem. Memory coherence is achieved through the notion of time. All operations that change memory state are time-stamped with a globally consistent clock. A design is presented to maintain time in each node with sufficient precision to sequentialize all memory access operations. Subsequently, methods are described which use this clock to maintain memory coherence. This chapter concludes with a discussion of the expected performance and possible implementations.

4.1. Design Objectives and Rationale

The primary objective of the communication architecture is to provide a simple, efficient and general interface between a fast message system and the application programs in each node. Simplicity is required to implement the interface directly in hardware. A direct implementation is normally also a fast one. However, casting part of the architecture in hardware prevents easy modifications or extensions. Hence a rather general purpose architecture is required.

This general goal breaks down into a number of subproblems:

- How to exchange information with minimal overhead?
- What are the functions provided by the communication architecture?
- Is the set of functions orthogonal and complete?
- What are the performance and implementation consequences?

As with many questions about architectures in a new domain with relatively little hard information and experience available, fundamental design decisions require a bit of intuition:

> "Efficient multiprocessing depends on a harmonious blend of synchronization, coherence and event ordering in a system that presents the user with a simple logical model of concurrent behavior." [42].

The approach taken in this thesis rests on two basic design decisions: the use of the *shared memory* programming model and the use of *time* to achieve coherency and synchronization.

Evidence to support these decisions will be presented; however, practical experience is too limited to claim superiority over other approaches.

4.1.1. Low Overhead Communication

The use of explicit communication directives, for instance through *send* and *receive* primitives, requires the assembly of a message or the creation of a (virtual) channel. In either case some extra amount of application code is required.

Reducing this overhead while still maintaining direct control over communications are schemes where the semantic of accessing certain parts of a local, private address space has the side-effect of initiating a data transfer. For example, declaring a global queue with one write-only port in one node and a read-only port in another node could reduce the communication overhead.

The problem with this approach is defining a simple and general set of special memory objects. If the libraries of communication primitives for second generation hypercube machines are any indication, this set of primitives is large and support requires a considerable amount of processing power in the communication interface, which is typically a dedicated, programmable processor. Hence the chances for successfully implementing just the right set of primitives in hardware are slim.

A globally shared address space provides extremely low overhead communication. This simplicity comes at the expense of a certain waste of bandwidth: communication takes place even if it is not needed. It will be argued that this extra bandwidth can be provided at a reasonable cost.

There are motives to support explicit message-passing that do not relate to its function in connecting private-memory processors. Examples include the use of message-passing as a tool to structure complex programs, to provide clean and flexible interfaces, to add protection, to aid program verification, etc. There is also the need to provide a migration path for software that was developed on explicit message-passing machines.

Message-passing can be implemented efficiently on top of the shared memory model. Proper programming of the memory management unit can provide separate logical address spaces with dedicated areas for message buffers and queues. A modest set of address space mapping primitives can achieve the functionality of a message-passing system. Besides being flexible and general, there are several performance advantages: for example by allocating the buffer memory on the destination node, data transfers take place at message creation time; hence data is being transferred while the message is assembled. This results in a substantial reduction of message latencies. Therefore, using the DSM model as primary communication architecture does not impair the efficient implementation of a message-passing system.

4.1.2. The Programming Model

The programming model provided by the communication architecture is that of a weakly coherent PRAM according to Definition 3. The weak coherency is used to provided some breathing room for the hardware implementation by allowing concurrency between program execution and memory updates which will lag due to the network delays.

The definition of weak coherency used time mainly out of convenience to express causality. No statement was made on how fast time advances and any function that establishes an (partial) order could be used. In the proposed architecture, the notion of time becomes concrete: it is exactly the physical time, advanced in constant increments fine enough to distinguish between consecutive memory cycles. This differs from the concept of virtual time in the *Time Warp* operating system [64].

Models of weak coherency differ in the way the parameter t_s is advanced. The two fundamental options are *lazy* or *eager* consistency updates. Implementations with a lazy update policy will restore coherency on demand. t_s is advanced by an explicit or implicit synchronization instruction. Without such operations, the incoherent state can persist for indefinite duration. Hence there is no bound on $t-t_s$.

Machines with the eager update policy strive to keep $t-t_s$ as small as possible.

The proposed architecture uses the eager update policy. In addition, the current t_s is available to the application program, as is the current time t. t_s will be referred to as "*past*" while t is "*present*."

This PRAM programming model guarantees strong coherency for all *past* operations. Inconsistent state is confined to the brief period between *past* and *present*. This period of uncertain state will be referred to as the *gray zone*.

4.1.3. Performance Considerations

Because the physical memory is distributed among all nodes, access time will vary depending on the memory address. The message system is not a direct replacement for the switch of a conventional shared memory system like IBM's RP^3 or BBN's Butterflies, due to higher latencies. Therefore, techniques to reduce access delays are an integral part of the architecture.

Two methods of access time reduction are used: caching and memory replication. Caching differs from conventional caches in the way coherency is achieved. Because the message system does not support broadcasting, the snoopy cache approach is not feasible. In avoiding the complexity of directory based invalidation, caching distinguishes between local and remote memory. Local memory is cached normally. Caching of remote memory differs by limiting the lifespan of the cached entry.

Memory replication permits copies of the same data in multiple places. Thus access times for multiple nodes can be reduced at the expense of a reduction of total physical memory space.

4.2. Related Work

Increasing the speed of the message system moves overall system characteristics closer toward true shared memory machines. Hence the experience gained with both bus-based and switch-based true shared memory machines is relevant.

To a lesser extent, the experience with implementations of message-passing private memory machines is helpful. These machines serve mainly as a reference point to illustrate the considerable complexity and poor cost-effectiveness of high end implementations of this type of architecture. For example, JPL's latest Hyperswitch requires a large (>1200 cm^2) and extremely complex (15-layer) PC-board, fully populated with fast, hot and expensive chips to implement a high-end version of their communication architecture. Even with several custom VLSI components, this microcoded controller is fully utilized handling the message traffic and does not add to the processing power of each node.

Since the message system described in Chapter 3 can be extended over multiple clusters, the communication crosses the boundary to local area networks. Again, due to reasons rooted in the early implementation technology, the prevailing communication architecture is based on explicit message-passing. Hence the direct applicability of this vast body of experience is limited.

4.2.1. The Message-Passing Alternative

A DSM-based communication architecture that spans several clusters distributed over a local area performs some of the functions of a high speed local area network. However, the former approach is motivated by combining multiple processors into one, larger system with the potential for using the aggregate resources on one application. This differs from the motivation for high speed local area networks which strive to facilitate data transfer between multiple, autonomous units.

While there is a substantial amount of overlap, the main emphasis and motivation are quite different. The main differences include:

- Support for long haul networks.
- Support for heterogeneous systems.
- Concern for system security and fault tolerance.

While some proposals for future high-speed long-haul networks are considering DSM-like systems, the approach presented in this thesis is not intended to span more than a local area (for example a research lab). The long latencies inherent in long-haul networks pose numerous problems that were not considered here; hence it appears that the traditional message-passing approach is more suitable.

The networking perspective is concerned with systems composed of very different machines. Hence interface standards and well-defined communication protocols are of foremost importance to allow multiple, independent implementations to operate together. Therefore, an important function of local area networks is that of a match-maker between dissimilar systems. This task is much easier with traditional message-passing than it is with a DSM system. Again, the motivation of the proposed communication architecture is different: while the individual node may vary in size, capabilities and peripherals, they all must conform to the same basic architecture. For example, all nodes must address the shared portion of the memory using the same byte-order. Furthermore, since memory becomes a common resource, its allocation and access control requires cooperation of all nodes, which rules out dissimilar operating systems. In fact, it is ultimately intended that the entire system be run by *one* distributed operating system as opposed to a collection of autonomous operating systems.

A message-passing system provides very natural means to enforce system security and may be easier to design in a fault tolerant fashion. While both issues are also important for a DSM implementation, it is certainly not the architecture of choice when extreme security and/or fault-tolerance is required.

The bottom line is that there are several areas where explicit message-passing is indispensable. The proposed DSM-based architecture is not billed as the universal solution, rather it is specifically aimed at cost-effective multiprocessing with the potential of modest spatial distribution.

4.2.2. Characteristics of DSM Systems

DSM systems can be characterized along six dimensions:

1. Coherency support
2. Caching method
3. Memory replication
4. Memory ownership
5. Update policy
6. Remote operations

Coherency support ranges from no support at all, through support for weak coherency, to support for strong coherency. Systems that provide no support delegate the coherency problem to higher levels (programming conventions and/or advanced compiler techniques). In the extreme case, a read following a write from the same processor to the same location may return the old value unless the operations are separated by a certain number of instructions. More commonly, reads following writes within the same processor are guaranteed to return the new values. In both cases, the value seen by other processors is undefined. Sharing data though the use of global memory requires special precautions, such as explicitly flushing the cache, marking pages non cacheable and organizing memory such that no two processors write to the same memory location. The exclusive write rights can be changed with proper synchronization, which is provided by special facilities or instructions. Frequently, resources outside of the memory system are provided for synchronization. This may include special semaphores (Sequent), hardware support for messages (Elxsi), inter-processor interrupt logic, hardwired barrier synchronization primitives, etc.

Weakly coherent systems allow inconsistent states but are prepared to restore consistency on demand. Strongly coherent systems guarantee that all read operations will return the most recently written value.

Caching is crucial to performance. However, maintaining cache coherence is difficult. Machines that broadcast all memory transactions can monitor the memory to maintain consistency [70]. However, broadcasting requires vast amounts of bandwidth, most of which is wasted. Therefore, more efficient caching methods are required. The problem can be deferred to the application level by mandating software controlled caching. Programs need to declare their access pattern in advance to ensure that memory locations are not cached in multiple places.

Common to all DSM systems is a global address space that maps into the collection of distributed physical memories. If memory replication is allowed, this map may allocate one part of the address space in multiple nodes. Such allocation can support a higher access frequency and/or reduce access delays. However, replication complicates the coherency problem.

Related to the issue of memory replication is the question of memory ownership. One approach to memory replication is to designate a master copy. Only the node holding the master copy is allowed to alter the state. Slave copies are either updated or left inconsistent. In the latter case, the node holding the master copy invalidates all slave copies on demand. In this context, eligible owners are the communication controllers of the processor nodes and not the process running on a node. Master copies may be write accessible by remote nodes. In this case, the remote write request has to be sent to the master copy holder. The alternative to memory ownership is a DSM system that allows multiple copies, all created equal.

Update policies for replicated memory range from no updates to immediate updates. The "*no update*" policies require a designated master copy to achieve coherency. Slave copies will be invalidated once restoration of consistency is required. Symmetric, replicated systems propagate changes made to one copy be made to all others.

Operations on remote memory objects may include:

Read	Reading a remote location may either retrieve the address data or an entire cache line. Some coherency methods require that the initiating node address is recorded for future invalidations.
Write	Writing to a remote location may require invalidation of cached copies of that location.
Read-Op-Write	Atomic operations, such as *fetch-and-add*, may provide efficient synchronization.

4.2.3. Proposed DSM Systems

Early experiments on DSM systems were conducted on machines linked by a local area network [45, 87]. Strong coherency is supported by allowing at most one writable page. Multiple read-only pages may exist. Processors may cache memory local to that node (the systems used for these experiments were not equipped with caches). Replication is restricted to read-only pages. Due to the slow media, no remote memory access is feasible. To attain acceptable performance, a number of bandwidth conserving techniques were developed. Among these are copy-on-reference and process migration. Process migration replaces remote procedure calls [99], which were constrained by the lack of a global address space. To attain reasonable performance, applications must exhibit medium-grained parallelism with good locality of reference.

The SVM system [125] replaces the local area network with the much faster message system of the Intel iPSC/2 but retains the overall structure. Memory pages are owned by processor nodes. SVM is strongly coherent and allows read-only page replication with designated owners in charge of invalidation. A page fault requires about 3 to 4 msec for 4Kbyte pages.

While using similar technology as the SVM system, the Clouds system [115] provides a user-controlled, segmented memory. Memory objects are made accessible by explicit function calls that also provide synchronization facilities.

The significant latencies to move pages or segments restrict DSM systems to applications with relatively high granularity. Hence it is desirable to reduce the memory object size to single words or cache lines. In such systems, the processor cache becomes part of the DSM system. Maintaining the directories of a paged system is insignificant compared to the page transmission times; hence it is done by the page fault handler in software. Once the memory object size drops to words, directory access becomes critical. Therefore, fine grained, directory-based DSM systems implement directories in hardware as part of the memory subsystems. Studies on the expected performance are given in [4].

MERLIN [152] is based on the notion of *reflective memory*. Parts of the address space of one node are mapped into the address spaces of remote nodes. Local memory traffic is monitored and all write operation to the designated memory regions are sent over a network to memory attached to remote nodes. Merlin is intended to interconnect conventional high performance machines. The Merlin project has no control over the structure of its processing nodes; therefore, it is constrained to appear like a peripheral device with extra memory to the host processor. In particular, no change of the memory and cache structure of the host processor is possible. Therefore, synchronization and coherency are provided by facilities outside of the DSM system. Replication with a statically owned master copy is provided. Only the owner node can write while slave copies are read-only. Slave copies are updated whenever the master copy is changed.

Attempts to extend broadcast-based DSM systems that are not limited to a single bus implementation resulted in multi dimensional structures of buses [52]. Nodes attached to multiple buses have tables that control selectively any forwarding of transactions across multiple buses so that snooping cache controllers can monitor all relevant memory traffic. Strong coherency is achieved via snooping caches. Memory is not replicated and remote memory access is supported.

4.2.4. Programming Models

The basic programming model is that of a parallel random access machine (PRAM): each element of a finite set of processors P executes an independent sequential program accessing a common memory.

Each memory read operation $r_i = (a_i, v_i, p_i, t_i) \in R \subset A \times V \times P \times T$ within the execution of PRAM programs is described by the memory address a_i, the retrieved value v_i, the processor number p_i and the execution time t_i. Likewise, each memory write operation $w_i = (a_i, v_i, p_i, t_i) \in W \subset A \times V \times P \times T$ is described by the memory address a_i, the stored value v_i, the processor number p_i and the execution time t_i.

Memory operations are atomic and are separated by a finite amount of time such that operations are ordered in time. This is equivalent to assuming processors of finite speed that cannot read and write simultaneously. It is further assumed that time has enough resolution to sequentialize any two operations throughout the system[43].

Ideally, memory access is *strongly coherent*:

Definition 1: A PRAM is strongly coherent iff:

$$\forall r_i \in R : \exists w_j \in W : v_i = v_j \wedge a_i = a_j \wedge t_i > t_j \;\wedge$$
$$\forall w_k \in W : k = j \vee a_k \neq a_i \vee t_k > t_i \vee t_k < t_j$$

All read operations always return the most recently written value to that memory location.

Ideal PRAM are difficult to implement due to finite switching delays and limited memory bandwidth. Therefore, the less rigid model of a weakly coherent system was introduced:

Definition 2: A PRAM is weakly coherent iff:

$$\forall r_i \in R : \exists w_j \in W : v_i = v_j \wedge a_i = a_j \wedge t_i > t_j \;\wedge$$
$$\forall w_k \in W : k = j \vee a_k \neq a_i \vee t_k > t_i \vee t_k < t_j \vee t_k > t_s$$

The system is strongly coherent with respect to all operations that precede t_s.

Definition 2 is equivalent to definition 1 if $t_s = t$, where t is the current execution time. Programming a weakly coherent PRAM requires some means of reasoning about t_s. For example, t_s could be made accessible through explicit synchronization directives. Whenever a strongly coherent state is required, a synchronization operation is issued.

Frequently, definition 2 is strengthened such that processors remain strongly coherent with respect to their own operation:

Definition 3: A PRAM is weakly (self-) coherent iff:

---

[43]This assumption merely simplifies the notation. It is not strictly necessary because concurrent operations of different processors can be distinguished by the unique processor number.

$$\forall r_i \in R : \exists w_j \in W : v_i = v_j \wedge a_i = a_j \wedge t_i > t_j \;\; \wedge$$
$$\forall w_k \in W : k = j \;\vee\; a_k \neq a_i \;\vee\; t_k > t_i \;\vee\; t_k < t_j \;\vee\; (t_k > t_s \wedge p_k \neq p_i)$$

Definition 3 will be used in this thesis unless noted otherwise. It is not necessarily true that $t - t_s$ is bounded. For example, a system with an explicit synchronization directive may allow multiple copies of the same memory object to exist in different processors. A change to one copy may stay invisible to other nodes indefinitely.

Caveat: Let x, y be two variables that were initialized to 0 and that have been changed to 1 by exactly one processor that writes to x and then updates y. Given Definition 3 with a bounded $t - t_s$ then reading a 1 from y by some other processor does not imply that a subsequent read from x would return a 1.

The reason for this apparent violation of causality is that the variables x and y do not have to reside in the same node, hence the delay between issuing the write operations does not directly relate to the time the new value becomes visible. This is really a problem only for DSM systems. Bus-based, weakly coherent systems ensure visible causality due to the sequential nature of the memory access. Most weakly coherent DSM systems can make this guarantee only if x and y reside on the same node.

4.3. System Overview

Cost effectiveness is one of the primary objectives of this communication architecture. However, cost effectiveness is also assumed to be important to the overall system. Given today's technology tradeoffs, this excludes high-end implementations such as GaAs components, immersion cooling, etc. The nodes of the intended system will not use Cray-style technology, rather they will use high-end CPU's such as Motorola's 88000, MIPS's R3000 or other *"Killer Micros."* To maintain system balance and overall cost-effectiveness, the communication architecture must be easily integrateable, may not use expensive technology and must fit in a small (compared to CPU and memory) space.

Because the communication architecture is closely associated with the memory, it is integrated into the memory controller. Figure 4-1 outlines the structure of one node. A conventional processor, for example Motorola's MC88100, is attached to a cache and address translation unit. Contemporary CPU's either have integrated cache and address translation facilities or offer these functions as a separate chip that is specifically designed for the CPU architecture. In either case, the CPU, memory management and cache form a black box building block.

The CPU block is attached to a conventional bus. This bus may be used for optional I/O devices. Unlike other message-passing machines, the interconnection network is not a

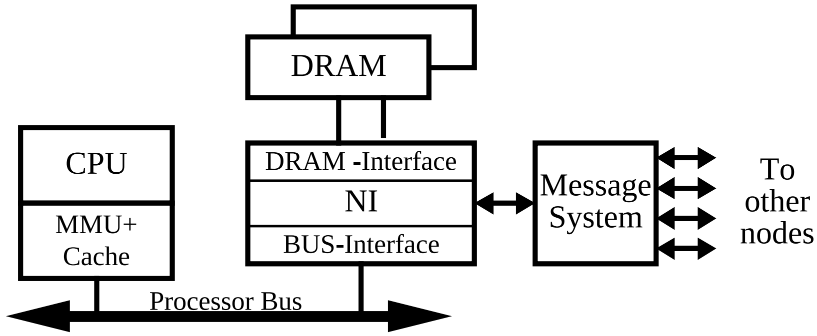


Figure 4-1: Processor Node Organization

peripheral device, rather it is integrated into a memory controller. Conventional memory controllers simply drive dynamic memory chips which require an address multiplexor, a refresh counter, a decoder, a few delay elements, and perhaps a parity or ECC circuit. Such devices are largely dominated by the I/O pin drivers.

Since the memory controller is, by today's standards, a vastly underutilized chip, and because all the required signals are accessible, it is the ideal place for the network interface (NI). Therefore, the entire message system and network interface form one building block that appears as plain memory to the processor bus.

The entire assembly - CPU bloc, DRAM and NI - could be integrated on a relatively small PC-board or ceramic carrier. Such network building blocks could be added incrementally in much the same fashion as transputer modules are used. The characteristics of such building blocks are small size and minimal amount of glue circuitry. The module is largely dominated by the CPU, its cache and the memory array. The message system has a number of ports that can be connected to any similar port on another module. These building blocks could be added incrementally to a basic system that provides little more than power, cooling, a clock and some peripheral divides.

The interface to the communication architecture is equivalent to that of a block of memory. There are a number of special registers to initialize the system and to control the memory mapping, but basically the communication architecture behaves like a bank of memory. The initialization and control functions can be performed through the network. Therefore, it is not strictly necessary that each node has a CPU. Passive, non-CPU nodes may be added to a system for special purposes. For example, a node gathering video-data may only feature a fram-grabber with a DMA controller that writes the image into a certain area of memory. Processing nodes interested in that data can map to that address space and simply read the data. Other uses of non-CPU nodes include bit-map displays, audio-I/O, disk-drives, etc.

By default, the connections between nodes use fast intra-cluster channels. Extending the architecture beyond one cluster does not require changing the nodes, rather it employs separate interfaces that translate between intra-cluster channels and inter-cluster channels. Therefore, the ability to integrate multiple, physically distributed clusters has no direct impact on the node structure.

4.4. The Notion of Time

Time is the basic mechanism for the proposed communication architecture. Since time is used to resolve the order of memory operations, the unit of time needs to be roughly equivalent to the time needed for one memory operation. Currently DRAM access times are in the 50-100 nsec range. Hence one time unit is in this region. The exact choice depends on the system clock and the basic message transmission cycle time. For reasons discussed later, the transmission cycle time must be an integral multiple of the time unit.

4.4.1. Keeping Time

Providing each processor in the system with an accurate clock is relatively easy to accomplish if the system operates synchronously, that is all processors receive their clock signals from one central source. In this case, the clock in each node is simply a counter that is reset at system initialization time and continuously incremented on every clock cycle.

This method has to deal with two problems: the global distribution of a common clock signal and the detection of and recovery from inconsistencies. For systems that are physically distributed, a common clock becomes inconvenient and a method to synchronize multiple clock sources is needed.

Ways to distribute a common clock were discussed in section 3.3. However, given a common clock, the absolute time keeping problem is only half solved. No serious digital design should be unprepared for a glitch in a critical component. If the proposed system were to rely on a large number of counters to advance in lock step fashion after reset - with no way of verifying consistency - extremely difficult diagnostic problems would arise. In the (however rare) event that one time keeping counter in a large system would be off by a small amount - for instance due to a power supply glitch, an alpha-particle hitting a critical gate of the underlying VLSI circuit, aging or a marginal manufacturing defect - no immediate problem would show up. However, coherency would be compromised for certain access patterns and programs would fail occasionally. Even worse, many applications would happily finish with incorrect results.

The severity of the timer problem arises from its cumulative nature. In the absence of the mechanism describe below, any glitch would be preserved for as long as the system is running: potentially months or years. So while the probability for a single CMOS/VLSI counter failure is quite low, having many of them count correctly for a year at a high rate is woth consideration. Given that every part of the logic implementation is subject to some failure mechanisms, there is no perfect solution to this problem. Periodic consistency checks and ways to correct for transient errors go a long way to make time keeping a non-issue.

4.4.1.1. The Local Time

The local clock can be read by the processor. Since the unit of time may exceed one processor cycle, a circuit is required to inhibit the time from returning the same value twice. This can be done by stalling the processor on the second access attempt until the timer has advanced by one tick.

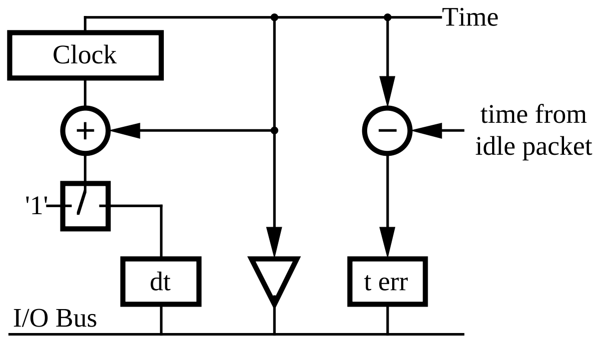


Figure 4-2: Time Keeping Circuit

To compare the state of the timers of two processors that are connected by a communication link, the idle packet is used to transmit the timer state. Idle packets are exchanged in the absence of real traffic on every routing cycle. This ping-pong protocol maintains synchronization of the send/receive protocol engines, verifies the operational status of a communication link and is essential to the oldest-message locating protocol. Since the communication delay between adjacent processors is known, a direct comparison of the timers is possible. Upon detection of a discrepancy, an exception is raised. Timers and all other functions are maintained as if no problem occurred. In response to the exception, the attached processor can force idle packets to be exchanged on any channel. As part of the exception, the difference between the expected and reported values is made available to the processor. It is therefore able to determine how far off its timer is with respect to all adjacent timers and whether that difference is changing or not. A changing difference would indicate a permanent problem. If only one channel reports a difference, the problem is assumed to be with the remote timer. If all differences are the same, the local timer is presumed to be in error.

To recover, the processor is able to add or subtract a value to the timer. This avoids all problems associated with a direct set ability, namely the need for precise synchronization of the processor instruction execution and the timer update interval. Permanent errors or cases not mentioned above, for example the channels reporting different error offsets, probably require a non-graceful exit, although there is little that clever software can't deal with.

The error detection depends on idle-messages being sent. Paranoid processors may force idle messages periodically, but this isn't really necessary because truly 100% network utilization is hard to achieve. Even given an infinite supply of messages to be sent on each processor, about 1% idle messages will prevail unless the address patterns are carefully matched to the topology.

4.4.1.2. The Global Time

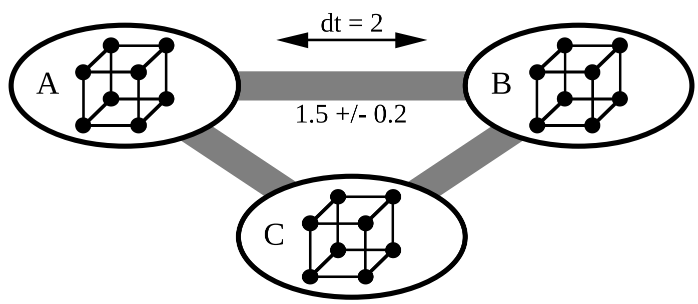


Figure 4-3: Inter Cluster Time Maintenance

All clocks within one cluster run off the same clock and are synchronized through the cluster reset. Maintaining time in a network of multiple clusters involves the problem of longer and less precise delays. The clock distribution system described in Section 3.3 guarantees that the clocks in multiple clusters will stay tightly synchronized. However, there may be start up transients and perhaps errors due to electromagnetic interference.

In order of allow flexibility in dealing with such problems, the inter-cluster delays are overestimated (Figure 4-3). For example, if the actual delay of an inter-cluster channel is 1.5 units, it is assumed to be 2 units. Packets that arrive earlier may have to wait 0.5 units, a small fraction of the packet transmission time. This allows individual clusters to drift by +/-0.5 units without causing inconsistent clocks.

4.4.2. Time-Stamping Packets

For the purpose of maintaining coherency, message origination time is needed at the receiving node. However, including the current clock in the header is quite wasteful. Therefore, the header contains only a relative time-stamp that is initialized to 0 at packet creation time. Whenever a packet is passed through a channel, the transmission delay is added to the age field. At the receiving site, the age is subtracted from the time of reception to yield the time of packet creation.

4.4.2.1. Maintaining Time-Stamps

The router - as described in Section 3.9 - is built around a register file with serial access capability. Essentially, each word in the register file is a shift-register that can store one entire packet plus some transient information (such as the subset of transmitters that can be used to advance the packet toward its destination). Serial access[44] is supported so that the transmitter can operate directly out of the register file. This eliminates the need for special parallel to serial converters in each transmitter and their counterpart in each receiver. This also reduces the need for data transfers if uniform access to the register file is provided.

Each transmitter needs two locations in the register file (one for each virtual channel). Likewise, two registers are necessary for each receiver. A small number of buffers for transient packets greatly improves the efficiency of the routing heuristic. *Small* is meant quite literally: one or two transient buffers achieve most of the potential gain. A design with 4 transient buffers appears optimal.

In addition to the storage locations mentioned above, the input and output queues are also located in this register file. Each queue has several entries so that multiple packets can be entered into and/or removed from the network in one transmission cycle.

A router that has 8 channels, 4 transient buffers and I/O queues of 8 packets needs a register file with 52 entries, all of which may contain active packets that are subject to the time keeping mechanism. All active packets stored in the register file have to be aged on every transmission cycle. Furthermore, the router must compute the time of the oldest packet in this area and has to expire packets that exceed the max age.

One approach to the aging problem is to convert the packet age from the relative representation to an absolute representation whenever a packet is stored into the register file. As part of the retrieval operation, the age is subsequently converted back to its relative

---

[44] Serial access does not imply bit-serial access: there can be more than one tap into the shift register so that 4 or 8 bits can be moved in/out in parallel.

representation. The time-stamp of the least recent packet (t_{lrp}) is easily computed with a priority network that operates on the age-field of the stored packet[45].

Alternatively, the aging problem can be solved by attaching a small finite state machine to each location in the register file. The need for serial access could be met with a dynamic shift register implementation that is less chip-area intensive than an array of static memory cells. Each word is recirculating in place during each routing cycle. Since the transmission protocol sends or receives a number of bits in parallel, each word in the register files is subdivided into several recirculating shift registers. The implementation is free to choose a partition that is based on functional considerations. In this case, the bits of the age-field are allocated in one shift-register such that the least significant bit is moved out first. The bit is passed through the FSM before it is reinserted and the FSM simply implements a bit-serial incrementer. It is easy to detect overflows this way. Also, the maximum detection logic is quite simple. A single arbitration wire connecting all words in the register file is needed. During each bit-cycle - starting with the most significant bit - all FSM's assert their value onto the wire, which is acting as a large OR gate. If the wire settles on the correct value, the FSM continues to participate in the process. If the FSM sees an incorrect value, it will stop participating in the minimum search. Using this method, the arbitration wire will cycle through the binary representation of the maximum packet age.

Note that this approach is quite similar to the router logic, which operates on the priority and transmitter assignment fields that are added to each packet upon insertion into the register file.

4.4.2.2. Contribution to the Routing Heuristic

The routing heuristic developed in Section 3.5 did not base routing decisions on the history of a packet. This is largely because such information is not easily available. Priority is given only to packets that are temporarily stored in transient buffers, a fact that is available within the scope of the router. However, the next router cannot distinguish between packets that were delayed in transient buffers and those that were not.

By adding the age field to the header, the accumulated delay becomes visible to the router and can be used to give priority to older packets. This has the net effect of reducing the variance of the latency distribution. Reducing the latency variance is desirable because it renders the message system more predictable and uniform.

---

[45]This assumes a custom VLSI design. Otherwise, the entire implementation of the register file, router and associated circuitry is quite complex and expensive.

4.4.2.3. The Sequencing Problem

Adaptive routing does not preserve order. Packets are routed independently through the network and may encounter different delays. Because of this, two packets from the same source node and addressed to the same destination may arrive in reverse order.

It is necessary to preserve the order of transmissions between any two nodes in the network. The traditional approach is to use sequence numbering. The originating node for each source / destination pair assigns sequence numbers to each packet. At the receiving end, a counter is maintained and packets are ordered by their sequence numbers. Sequence numbers can be re-used once a packet is removed from the network. Using a cyclic counter, the destination node sends a flow-control packet back to the originating node once the first half of sequence numbers is consumed. This enables the originating node to reuse that block. This method achieves sequencing and a form of flow control.

There are two disadvantages to this method: the sequence number requires room in the packet header, thus decreasing useful bandwidth and each node must maintain two sequence counters for every other node in the network (one to send and one to receive packets).

The age field introduced for time-stamping *almost* serves the same function. If receiving traffic is ordered by the time it was sent, the original sequence is restored. A minor problem with this approach is that all packets for each source destination pair must have unique origination times. Inserting two packets into the network in one cycle is therefore prohibited. Multiple insertions, however, are essential to reduce latencies; hence the unit of time for the age should be less than one transmission cycle length. In this case, multiple packets inserted in the same cycle may differ in age as required to ensure the proper sequence.

The major problem is associated with gaps in the traffic. Assuming that node A did not send packets to node B for some time, node B will have trouble recognizing the first packet after this pause. Node B is not able to determine whether the received packet is preceded by a less recent one or not. The consecutive sequence numbers don't have gaps; hence node B could immediately decide on the proper sequence.

It turns out that the proposed communication architecture is insensitive to message transposition because memory coherency is achieved through time-stamps. Hence it is sufficient to ensure that all packets for one message are delivered in the correct sequence. This is easily accomplished with three conventions:

1. The first packet of a message is identified by its type field as the start of a message. Subsequent packets of the same message are simply identified as data items.
2. The type of the first package defines the message length.
3. All packets of one message must carry consecutive time-stamps.

The number of different message types will be very small (*read*, *write*, *sync-operation*, *page-copy*, *diagnostic*), therefore a 4-bit type field is sufficient to distinguish the first packet of a message and define the message length. Insertion of messages with running time-stamps is also trivial because all but the page-transfer message consists of only one or two packets.

The receiving node temporarily stores data-packets. Once the matching header arrives, the message assembler can verify completion. Completed messages are forwarded to NI.

4.4.2.4. Error Recovery

Since the age field is of fixed size, the lifetime of a packet is limited. Non-terminal packets that have exceeded their lifetimes are converted into error-packets and sent back to the originating node. Error packets have indefinite life times.

4.4.3. The *Gray Zone*

A coherent and reliable time distribution mechanism, such as the one outlined in the previous section, is not sufficient to build a coherent memory system. Proper time-stamping can resolve any ambiguity among competing operations, but it is unable to finalize any operation without additional assumptions about the underlying message-passing layer. Typical communication systems for private memory multiprocessors have variable and potentially unbounded message latencies that depend on the traffic pattern, network load, routing and scheduling strategies. As a consequence, the receiver of a message has no way of knowing whether there are other messages in transit with older time-stamps. Operations will eventually cause a globally coherent state change, but they are of undetermined duration.

Assuming that the message-passing layer is well behaved[46], all operations will settle into a coherent final state in a finite time span. It is useful to define an upper bound on this time span:

> **Definition 4:** The *gray zone* is the smallest time interval $[t - t_{gzw}(t), t]$ that contains the time-stamps of all outstanding messages at time t. The gray zone width t_{gzw} is a function of the system state at time t.

Given knowledge of the gray zones for each memory location[47], an optimal system could be built. Operations that require the coherent value of a memory location will stall until the

---

[46]A message-passing layer is considered well behaved if it is free of deadlocks, and if the resource allocation is fair so that all active nodes will receive a non-zero share of the available bandwidth (absence of starvation and livelocks). Ideally, the communication bandwidth is equally distributed.

[47]A *gray zone* that is specific to a particular object considers only messages that can result in a state change of that object

gray zone for that memory location has expired, while all other operations proceed immediately. Any stall time would be the direct consequence of the propagation delay of a certain message critical to the pending operation and is therefore unavoidable.

Intuitively, the gray zone extends from the recent *past* to the *present* state of the system. The effects of current operations may not be visible immediately throughout the entire system, but they will finalize before the gray zone expires. In a conventional single processor system no gray zone exists: each instruction is completed before the next starts. Highly pipelined systems can have *"gray zones"* because multiple instructions can be executed concurrently. This is usually dealt with in software (dependency analysis at compile time controls appropriate instruction scheduling) or in hardware (scoreboarding). Perhaps the notion of an imprecise interrupt pioneered in the IBM 360/91 could be related to the concept of a gray zone: an exception could refer to one of several active instructions. In this case, the gray zone includes all active instructions in various execution units. For the purpose of this discussion, the gray zone is considered a feature of the memory system of a multiprocessor.

Messages are the carrier of memory state changes. Therefore, the gray zone is directly related to the definition of weak coherency: $t_s = t - t_{gzw}$.

In general, the precise gray zone width is only available from inspection of traces after program completion. A practical system needs inexpensive methods to produce tight and timely bounds on the gray zone at run time. Obviously, it is easier (and less efficient) to give global bounds on the gray zone rather than associate an individual gray zone to each memory location. Unless noted otherwise, the following discussion deals with global gray zones.

One approach to solve this problem is to expire messages after a certain period. If a message is not delivered within t_e cycles it is removed from the system and an exception is raised. This t_e becomes a conservative bound on t_{gzw}: the gray zone cannot exceed the lifetime of a message. The expiration time t_e is critical to the performance of the system because it is subjected to 2 opposing goals: a small t_e will minimize the time required to settle into a coherent state, but it also will increase the number of messages that expire.

The impact of a wide gray zone - the consequence of a large expiration time - depends on the frequency of coherent operations in a given program. A coherent operation needs at least t_{gzw} more time to complete than its incoherent counterpart. Assuming that a memory reference takes t_{mem} time and that P_{rmt} of all memory references are made to remote locations and require an additional mean message delay of t_{msg}, the average memory access time is $t_{mem} + P_{rmt} t_{msg}$. Therefore, if P_{sync} is the fraction of synchronization operations in a given program, the total program execution time is increased by a factor of K_{gzw}:

$$K_{gzw} = \frac{P_{rmt} t_{msg} + t_{mem} + P_{syn} t_e}{t_{mem} + P_{rmt} t_{msg}} \qquad (4.1)$$

The frequency of message expirations can be derived from the actual message latency distributions. The cost - in units of time - for one message expiration can be quite high. In the best possible case, a subsequent retry will succeed and only t_e time plus one partial round-trip message delay is wasted. Realizing that expirations are most likely due to network congestion, an exponential backoff strategy similar to the one used in the Ethernet protocol is appropriate. Further increasing the cost of message expirations is the fact that some operations cannot be retried easily. Remote read operations pose no problems since the originating processor has to wait for the reply anyway, so blocking is an inherent part of the operation. Remote writes can cause more problems because the processor can proceed in parallel with the pending write operation. Subsequent retries will advance the apparent time of the write and may violate memory causality. For instance, a read after the failed write to the same location may succeed before the retried write operation, thus yielding the old memory content. The sequencing mechanism must be capable of resolving this problem. Memory causality isn't strictly necessary for weak coherent systems, but its absence is counter intuitive and is bound to cause incomprehensible bugs. The most troublesome case arises if an update operation fails. Update operations are side effects to write operations if the target memory location exists in more than one processor node. This memory replication will be described later. A failed update leaves the system in an incoherent state and an error recovery involving all processors that hold a copy of the partially updated memory location is required. The amount of time required for this is roughly equal to a page replication or deletion operation and requires operating system intervention.

An approximation to the performance loss due to message expiration is based on several simplifying assumptions:

- Message latencies follow a negative exponential distribution. In reality this isn't true, but only two properties of this distribution are used: the average latency and the probability for the latency to be greater than a given value. The relation between these two properties is a reasonable approximation to real distributions that cannot be expressed analytically.
- The time needed to recover from a message expiration t_{rec} can be expressed as a single quantity. In reality, this depends on the ratio of read and write operations, the amount of replication, the particular access pattern, etc. For a given environment, a single average is a reasonable first order approximation.
- Memory access operations require a message transmission with probability P_{rmt}. In reality, this depends on the ratios of read, write and synchronization operations to local / remote locations, which in turn depend on the properties of the application programs. Again, this simplification is justifiable for a given application / system combination.

Given these assumptions, the expiration of messages will increase the total execution time by a factor of K_{exp}:

$$K_{exp} = \frac{t_{mem} + P_{rmt}(t_{msg} + e^{-t_e / t_{msg}} t_{rec})}{t_{mem} + P_{rmt} t_{msg}} \qquad (4.2)$$

The product of K_{exp} and K_{gzw} vs. t_e is plotted in Figure 4-4. The graph is based on $t_{mem} = 150ns$, $t_{msg} = 2\mu s$ and $t_{rec} = 200\mu s$. The fraction of remote references is set to 50% ($P_{rmt} = 0.5$). The overall performance loss is quite severe if the frequency of synchronization operations is high.

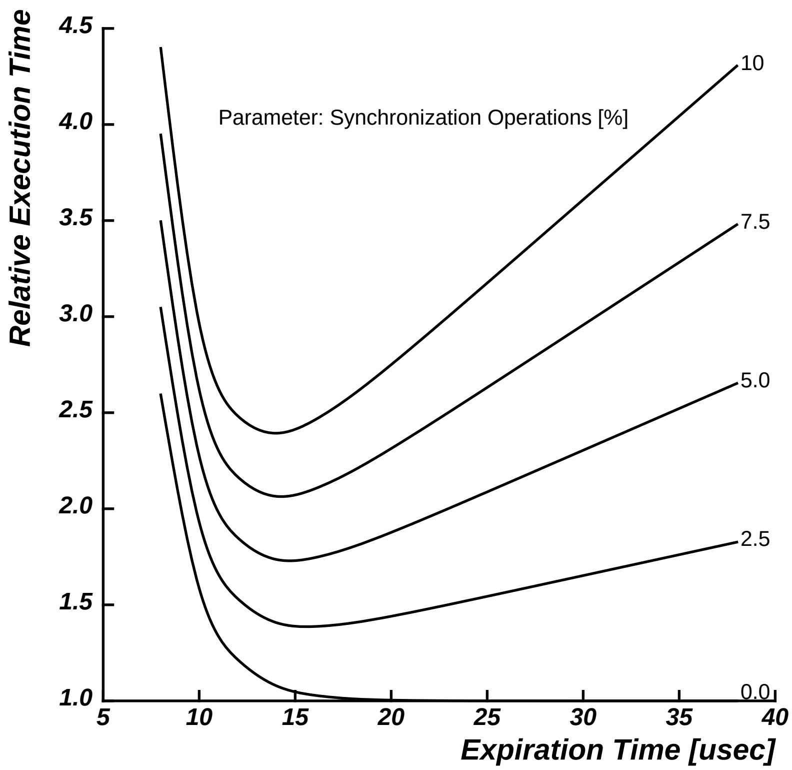


Figure 4-4: Optimal Expiration Times

It turns out that the optimal value for t_e is quite insensitive to changes in either P_{sync} or P_{rec} and can be optimized for a particular t_{msg} / t_{rec} combination:

$$\frac{\partial}{\partial t_e} K_{gzw} \cdot K_{exp} = 0$$

yields

$$e^{a \cdot t_e} + b \cdot t_e + c = 0 \qquad (4.3)$$

with

$$a = \frac{1}{t_{msg}}$$

$$b = -\frac{P_{syn} P_{rmt} t_{rec}}{P_{rmt} P_{syn} t_{msg}^2 + P_{syn} t_{mem} t_{msg}}$$

$$c = t_{rec} \frac{(P_{syn} P_{rmt} - P_{rmt}^2) t_{msg} - P_{rmt} t_{mem}}{P_{rmt} P_{syn} t_{msg}^2 + P_{syn} t_{mem} t_{msg}}$$

t_e can be derived from Equation (4.3) by means of numerical methods. Some solutions for typical message latencies and recovery times are given in Figure 4-5.

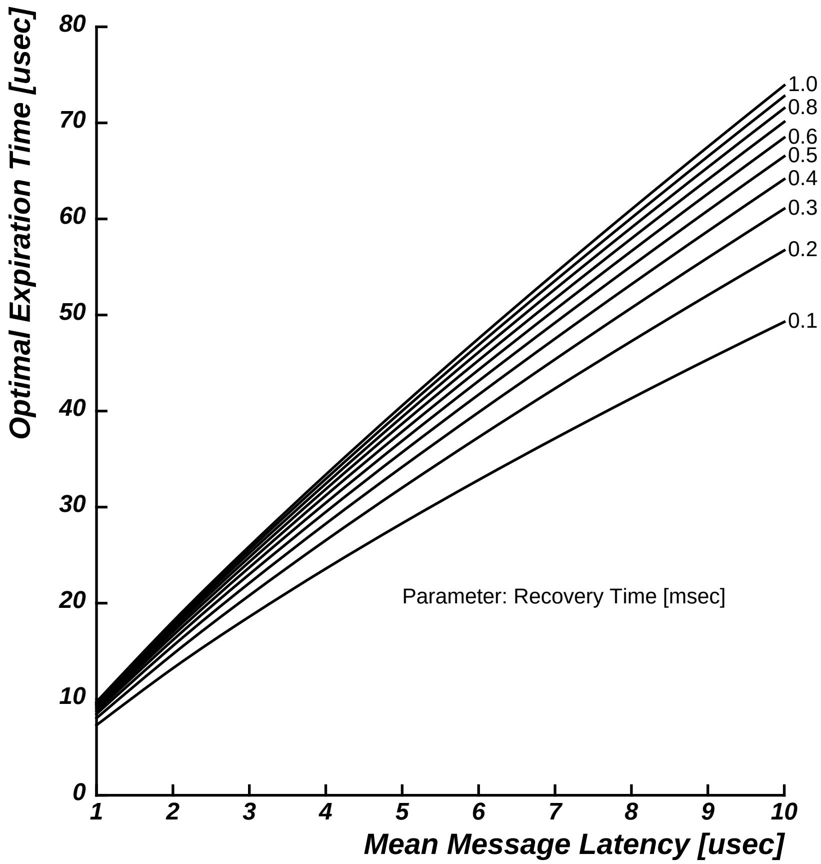


Figure 4-5: Optimal Expiration Times

4.4.3.1. Practical Gray Zone Maintenance

Because the use of a fixed gray zone compromises overall system performance, a method of dynamically computing the current t_{gzw} at runtime is needed. The gray zone width is an important parameter to many higher level functions of the system under consideration so its timely and continuous update is worth a direct implementation in hardware.

Conceptually, the gray zone could be computed by periodically freezing the state of the communication system. In such a *snap-shot dump* of the network state, each active packet[48] is associated with precisely one node. Therefore, each node of the network can compute the time of the least recent packet (t_{lrp}) in its possession. Each packet in transit will contribute to one and only one local t_{lrp}. Naturally, freezing the network isn't really necessary as long as there are well defined points in time for which an unique message-to-node relation can be established. This leads to an *interesting*[49] design constraint for the implementation of the data link layer:

> **Characteristic 1:** The inter-node communication is organized into global message transmission cycles so that at the end of each cycle all packets are associated with precisely one network node.

Systems that exchange more than one packet over a channel during one message transmission cycle are possible, but it is desirable to keep the message transmission cycle as short as possible because its duration will determine the frequency of gray zone updates: infrequent updates necessitate longer gray zones and thus cause slower coherent operations.

For the initial discussion, message transmission cycles are considered synchronized throughout the system. This is easily realizable given that the system runs synchronously, driven by a central clock. In the actual implementation, this restriction will be relaxed to accommodate physically distributed systems.

At the end of each transmission cycle, each processor will update its estimate of the gray zone width. It will update the lower bound of the time-stamp associated with any message addressed to this node. Any specific node - say PE0 - is considered the root of a communication tree, such as outlined in Figure 4-6.

Each node computes the age of the least recent packet in its possession (t_{lrp}). The minimum of this value and values from nodes located lower in the tree is passed to the next node up. Each transmission step incurs one transmission cycle worth of delay. Therefore, the t_{lrp} reported at PE0 for an empty network is lagging d transmission cycles behind *present*, where d is the network diameter or the depth of the tree.

Because t_{lrp} is monotonically increasing over time, it is not necessary to transmit the full value each cycle. To preserve bandwidth, only a 2-bit number is exchanged, which is piggy backed onto the handshake protocol of the intra-cluster channels. Transmitting a "0"

---

[48]Messages may consist of several independent packets: for example, a write-message can have two packets, one carrying the address and one the data. The sequencing and time-stamping mechanism at the network layer is designed to disassemble or assemble messages into or from packets

[49]This restriction is interesting because it is at odds with most current designs. This feature is difficult to implement in communication systems that operate on variable-length messages or that use self-timed and/or asynchronous hardware.

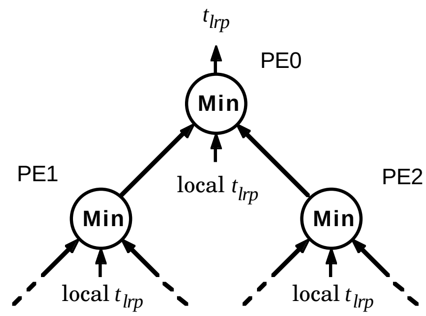


Figure 4-6: Locating the Oldest Message in Transit

indicates stagnation of t_{lrp}, "1" advances with time and "2" and "3" are used to catch up. The limited value range implies that changes to t_{lrp} are rate limited. If an old packet is removed from the network, it may be several cycles before the gray zone is properly updated.

4.4.3.2. Constructing Min-Trees

Each communication channel allows one 2-bit increment to be transmitted in each direction during each cycle. Therefore, each channel of the network represents two potential gray zone propagation paths in opposing directions.

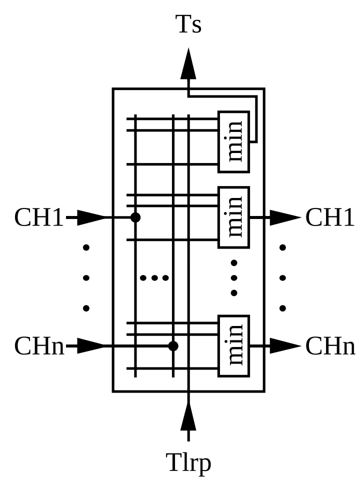


Figure 4-7: Gray Zone Logic

Each node has a programmable matrix that can combine the received t_{lrp} estimates with the local t_{lrp} to compute the local and transmitted bounds. This matrix is programmed according to the topology in use. The minimum computation trees for each node may share common sub-trees, and all trees must cover the entire network. Obviously, there can be no cycles in these trees, otherwise time could not advance.

It is always possible to construct such an assignment:

1. Construct a minimal diameter spanning tree for the network. This is a trivial by-product of the routing table computation.
2. Designate one node as *root* so that the maximum distance to all leaf nodes is minimized for all other nodes, using only paths of the spanning tree. The worst case distance to any node will be no worse than the network diameter.
3. Complete the min-tree for the root node.
4. Propagate the t_{lrp} of the root node to all other nodes using the reverse min-tree.

This *if-all-else-fails* procedure results in min-tree depths of no worse than twice the network diameter.

In practice however, most regular network topologies feature min-trees that are no deeper than the network diameter. For k-ary hypercubes, two basic construction steps allow optimal min-trees for the entire topology class.

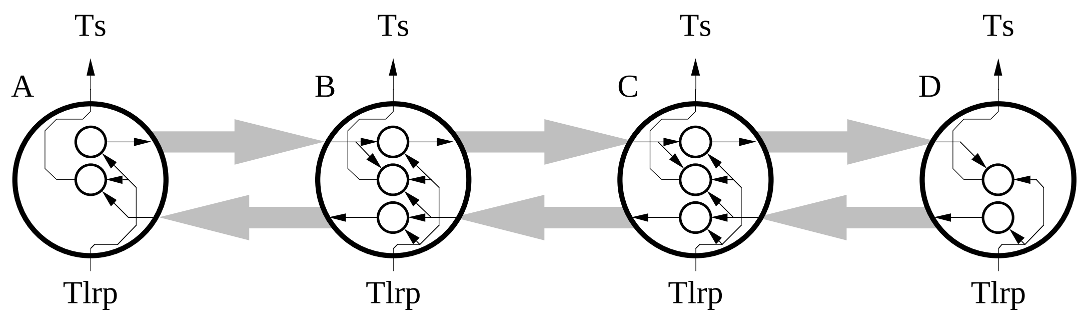


Figure 4-8: Gray Zone Computation for Linear Arrays

Figure 4-8 depicts the assignment for a linear array or ring of nodes. In cases of rings, the channel between nodes A and D is not used in the min-tree. This structure is required for each dimension of a k-ary cube and for the cycles of the cube--connected cycle topology.

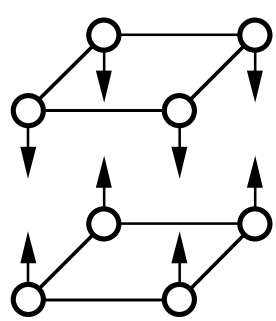


Figure 4-9: Gray Zone Computation for k-ary Hypercubes

Once the min-trees are realized for a cube of dimension d, the min-trees for a $d+1$ cube are constructed by joining the k sub-cubes with linear array structures along the next dimension (Figure 4-9). This recursive construction of min-trees has the benefit that there is an E^3 routing function such that all E^3 routes are embedded in the min-trees of the destination nodes.

4.4.3.3. Pathological Cases

So far the t_{lrp} computation simply assumed a static distribution of packets. It turns out that moving traffic normally presents no problem because packets tend to move towards their destination and this path usually coincides with the min-tree for that destination.

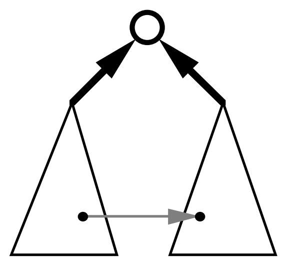


Figure 4-10: The Monotonicity Problem

However, packets are not confined to the min-tree path and may move to another branch of the tree (Figure 4-10). If that packet happens to be an old packet and if the other branch of the tree is only populated with more recent packets, departure and re-appearance in a different branch can cause a non-monotonic change in t_{lrp} for some nodes of the network.

Lack of monotonicity is not acceptable, therefore the maximum exported value of t_{lrp} is used to reject packets which otherwise could cause inconsistencies. Hence the route of old packets will be confined to the min-tree path to the destination.

4.4.4. Summary

Using the methods developed in Chapter 3, a viable time distribution system was presented that can supply a fine-grained and globally consistent clock. The resolution is sufficient to distinguish individual memory cycles.

The global clock will be used to time-stamp messages. While these time-stamps are primarily motivated by the coherency mechanism described below, they are also used to solve the sequencing problem and to reduce the latency variation of adaptive routing.

The message system is extended such that each node can provide a monotonically increasing, lower bound on the time-stamp of the least recent message in transit. This function provides a tight and timely bound on the width of the *gray zone*, which defines the temporal interval that includes the time-stamps of all outstanding messages.

An efficient and inexpensive method to compute the gray zone was developed. For most network topologies, the minimal gray-zone width equals the network diameter due to optimal embedding of the minimum computation trees. Furthermore, these trees can be design to coincide with E^3 routing paths, which help to minimize the gray zone width.

4.5. Coherency Support

The CPU-block (CPU, memory management and cache) in each node addresses *physical* memory. The virtual address space of the application process is mapped onto a physical address space by conventional means. However, a physical address may refer to a local or remote memory location. Furthermore, one physical memory object may reside on several different nodes.

Coherency throughout this system is maintained by using the previously described facilities: time stamps and the gray zone. Besides providing weak coherency for the entire memory, a set of strongly coherent operations is provided.

4.5.1. The *Time-Buffer* Approach

For illustration purposes, each memory location is assumed to carry a tag with the time of the last write operation. Read operations on such a memory are not affected. For each write operation, the initiation time is compared to time associated with the target memory location. If the write was initiated more recently than the last write to the target memory location, the write is performed as usual. If it turns out that the contents of the target memory location are more recent that the attempted write, the write operation is discarded and has no effect on the state of the memory. Successful writes will update the time-tags accordingly.

This system has the property of converging to the most recent state. It is insensitive to the delay between issuing a write operation and the time it actually arrives at the memory controller. Furthermore, multiple copies of the same memory objects will stay consistent as long as successful write operations are propagated to all copies.

This approach is equivalent to Z-buffered image generation, except with time replacing the third spatial dimension.

Tagging each memory location is quite wasteful and actually unnecessary. The time tags that are older than the current end of the gray zone are obsolete. They could be replaced with time 0 without changing the behavior of the system because there are no pending write operations outside of the gray zone. Therefore, tags can be discarded once they become older than t_s.

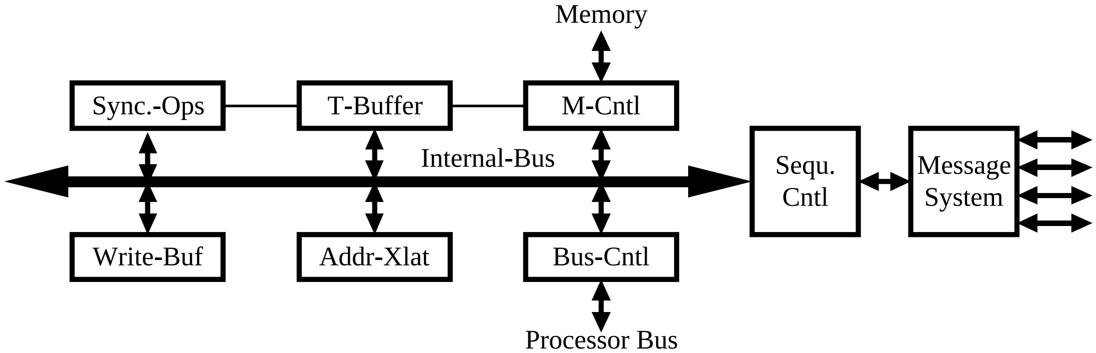


Figure 4-11: Network Interface Structure

Time tags associated with the memory of a node are stored in a separate time-buffer that is organized like a cache memory. For each write operation to memory, the address is presented to the time-buffer. In case of a hit, the time stamps are compared. Outdated write operations will prevent a memory write cycle. All successful write operations cause the address and time to be added to the time-buffer. Time buffer entries are recycled once they leave the gray zone. The size of the time-buffer is determined by the memory speed and by the maximum gray zone width (which is limited due to the finite size of the age field in the packet headers). Assuming a 150nsec cycle time and a maximum gray-zone of 32 μsec, a maximum of 200 entries are required.

4.5.2. Synchronization Operations

The vast majority of all memory accesses in parallel programs do not require strong coherency. There is, however, a significant fraction of operations requireing strongly coherent memory access. These operations are generally related to the control structures of the programs and not to the actual data processing, which explains the observed ration of access types in the programs used for this research. The fraction of strongly coherent operations relates to the origin of the code. Programs that were developed on true shared memory machines tend to use more strongly coherent operations.

Support for strong coherency is required to port parallel applications from other systems. They also permit a number of other programming styles that depend on certain atomic operations.

Strongly coherent operations are implemented by monitoring the memory traffic. Four forms of synchronization operations were used in the experiments:

- *Coherent read* returns the strongly coherent value of a memory location.
- *Exchange* returns the strongly coherent value of a memory location and atomically replaces it with a new value.
- *Test-and-Set* is equivalent to an exchange with '1' as the new value.
- *Fetch-and-Add* returns the strongly coherent value of a memory location before atomically adding a new value to it.

Each of these operations requires waiting the full duration of the current gray zone, hence these are relatively slow functions. In addition, the architecture provides fast access to the precise global clock. The access to the clock (*present*) and to the end of the gray zone (*past*) could be used in time based synchronization algorithms. For example, processes could negotiate the time of a rendezvous. However, this line of research was not actively pursued within this thesis.

In some cases where strong coherence is required, the program is able to initiate the synchronization operation ahead of time. To allow for concurrency between executing a synchronization and execution of unrelated code, the synchronization operations are split into two parts: an initiation and a verification operation. Initiation proceeds without blocking and returns the weakly coherent result. Subsequent verification will block until the strongly coherent result becomes available.

A potential use for the incoherent result of the initiation part arises in situations where the probability for differences between weak and strong coherence is low. For example, infrequent updates to parts of a large, shared data structure require locking for correctness, but locking usually succeeds. Optimistic programming may proceed before the lock is verified if it is possible to recover from a denied lock.

Besides its use in optimistic algorithms, the immediate incoherent result also serves as a flow-control mechanism. Remote synchronization operations depend on the availability of resources at the remote node. To avoid excessive inflation of the gray zone and to prevent deadlocks, the execution time of a remote synchronization operation is the time when the remote node started processing the synchronization request. The return of the incoherent result is therefore the acknowledgement that execution of the strongly coherent operation started.

The chosen set of strongly coherent operations is fairly common for shared memory multiprocessors. *Test-and-Set*, *Exchange* and *Fetch-and-Add* operations were proposed and are implemented in numerous systems. Given a specific application, there is a significant temptation to implement more complex operations. However, complex synchronization operations are at odds with high-level compiler technology, which has problems generating optimal code for complex instructions. Moreover, the set of desired instructions depends highly on the given application. A *Compare-and-update-pointer-if-less-or-equal* instruction may be perfect for one application but usually is not quite right and therefore useless. Besides complicating the implementation and the compiler, baroque features lead to non-portable code.

Given the lack of convincing applications of compound atomic operations that lead to significant performance advantages and that cannot be replaced with primitive synchronization directives, the proposed architecture implements only a basic set of strongly coherent operations.

4.5.3. Alternate Synchronization Approaches

Process synchronization does not necessarily require hardware-supported atomic operations (such as *Test-and-set*). Instead, understanding of the precise semantics of weakly coherent memory operations can lead to simpler and more efficient programs. The barrier synchronization algorithm discussed below is such an example. In this case, each variable is changed by only one process. The algorithm relies on the fact that a write will become visible to all processors since the underlying communication mechanism is eagerly striving for coherency.

Barrier synchronization, a common primitive for iterative concurrent algorithms, is an instructive example of how to use the properties of DSM efficiently. Brook's *Butterfly Barrier* [20] is the DSM equivalent of the normal algorithm for barrier synchronization in binary hypercube systems. The exchange of messages is replaced by access to shared variables.

Synchronization is achieved by synchronizing two processors at a time. For each pair, a set of two variables is needed, one associated with each processor. Let *bv1* and *bv2* be the variables used for the barrier, then the algorithm is as follows:

```
       Processor 1         Processor 2

(1)    while (bv1);        while (bv2);
(2)    bv1 = 1;            bv2 = 1;
(3)    while (!bv2);       while (!bv1);
(4)    bv2 = 0;            bv1 = 0;
```

Initially, both variables are initialized to 0. The processor that arrives at the barrier first will wait in the second spin lock. Once the other processor executes step two of the barrier, both processors can proceed. The first spin lock completes a two-phase hand-shake protocol.

The two spin locks can be combined if the boolean flag is replaced by an epoch counter:

```
       Processor 1         Processor 2

(1)    bv1++;              bv2++;
(2)    while (bv1 > bv2);  while (bv2 > bv1);
```

In this formulation, variables are written exclusively by one processor. Restricting the set of processors that may write to a particular variable at one time is a fairly general strategy to construct parallel algorithms.

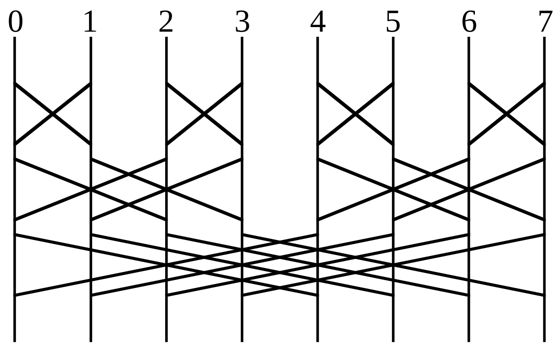


Figure 4-12: Brook's Butterfly Barrier

The two-processor barrier is a building block for logarithmic depth synchronization structures. The message-passing equivalent would exchange messages along the dimensions of a binary hypercube. The access pattern for the DSM version is given in Figure 4-12 for an eight-processor example. For 2^n processors, n sub barriers are necessary. Hence a total of n variables are necessary per processor.

Once a process has passed the barrier, it is guaranteed that all other processes have reached the barrier. However, it is not guaranteed that the results of all operations performed by the other processes prior to reaching the barrier are visible. To guarantee the validity of all results, each process may not rely on variables updated by different processors for the duration of one gray-zone width after passing the barrier. After this period, all side-effects caused by any operation before the barrier are guaranteed to be visible. This delay is easily accomplished by recording the time of passing the last pairwise barrier. Before reading remote data, this time is compared to the *past* variable. Once *past* is greater than the recorded time, all data is guaranteed to be correct. It is usually possible to hide this delay by performing some unrelated computations, for example initializing the next iteration.

Brook's butterfly barrier was refined in [58]. The *dissemination* method is more efficient if the number of processors is not a power of two. However, the *tournament* algorithm, also described in [58], is less suited for a DSM system due to its asymmetry. The butterfly barrier's symmetry avoids hotspots and distributes memory references evenly. Both are desirable properties for good DSM algorithms.

Lubachevsky [90] refines the algorithm further by reducing the number of variables to one per processor. He also gives practical implementations of related synchronization primitives.

4.5.4. Multiple Synchronization Domains

The gray zone mechanism - as presented so far - encompasses the entire system. All nodes throughout the network are considered, as are all messages. This behavior is appropriate if the system is relatively small or if the majority of all nodes are working on one problem. In case of large systems with a high network diameter, the global gray zone is rather large. Furthermore, localized congestion in one part of the network is universally increasing the latency everywhere.

Systems that typically run multiple, independent tasks may not require this global gray zone and could run more efficiently with a gray zone computed over the subset of the active nodes that contribute to one task. Since multiple, parallel, but independen,t tasks will have disjunct address spaces with mutual access restriction and protection, each subset of processes is not concerned with the messages belonging to another task.

Ideally, these properties could increase efficiency by useing multiple synchronization domains. There are two ways to implement multiple synchronization domains: *spatial* and *logical* decomposition.

The spatial decomposition does not require any changes to the system hardware. For each region of the network, a separate set of min-trees for the gray zone computation is constructed. Channels that cross region boundaries are simply ignored. This requires that the decomposition in independent regions matches the network topology. Given such a decomposition, the implementation is a simple matter of programming the min-tree computation.

The logical decomposition requires additional circuitry that recognizes what logical domain a particular message belongs to. Furthermore, each domain requires an independent set of min-trees. This either requires more bandwidth dedicated to the gray zone maintenance or less frequent gray zone updates that are time multiplexed. Both approaches represent a significant increase in circuit complexity.

Neither method has an effect on the basic memory coherency mechanism. Only the scope of strongly coherent operation is reduced. Hence synchronization scope can be traded against synchronization speed: local synchronization is faster than global synchronization. Global synchronization is still feasible through the use of gate-ways; thus it is possible to establish a synchronization hierarchy.

4.5.5. Summary

Time-stamped messages to execute remote memory access operations plus a temporary time-tag buffer associated with the local memory of each node are sufficient to implement a weakly coherent memory system. Memory replication does not compromise this weak coherency mechanism. Furthermore, it is insensitive to message reordering, thus the sequencing problem caused by adaptive routing is greatly simplified.

Given a mechanism to provide a tight and timely bound on the actual message latency, strongly coherent operation become feasible. An efficient method to dynamically compute this bound was presented. The concept of the gray zone was introduced to illustrate the nature of the weak coherency. The gray zone is the temporal interval that is guaranteed to contain all incomplete memory operations. Synchronization delays are dominated by the gray zone width.

4.6. On Caching

The communication architecture does not restrict access to the DSM in any way. Nodes accessing a remote memory object will send a message to the nearest node holding a copy of the address memory. The network interface (NI) of the remote node will subsequently perform the memory access on behalf of the originating node. Thus one NI function is the interpretation and service of unsolicited requests through the network.

Obviously, remote memory access is considerably slower than local memory access. A conservative design with contemporary components will need about 150 nsec for a local memory access. Remote access adds the round-trip delay of a message to the remote node, or about 2 times 450 nsec in the best case for a memory one hop away. Generally, remote access will be 10 to 20 times slower than a local reference.

Given CPUs running at 25 to 50 MHz with cycle per instruction ratios that are approaching 1, it is obvious that even local memory references are too slow to match the CPU speed. The traditional answer to this problem is caches. Any viable DSM system must allow some form of caching.

Besides the cache memory associated with the CPU, DSM systems can reduce the time required for remote memory accesses by replicating frequently accessed memory objects in multiple nodes.

Instruction caching is taken for granted in this discussion. Instructions are only read; and for performance reasons, it will be necessary to keep copies of the code in all nodes. Similar considerations apply to the stack. However, languages like C allow pointers to stack objects and concurrent C-programs will be able to pass pointers to automatic variables to other threads. Therefore, stack space must be accessible for read and write by remote nodes.

4.6.1. Replication of Data

Reducing memory latency and increasing memory bandwidth without using *true* multiported memories[50] are achieved by using several memory modules in parallel. Read operations can be directed to any module while write operations must be performed on all memory modules.

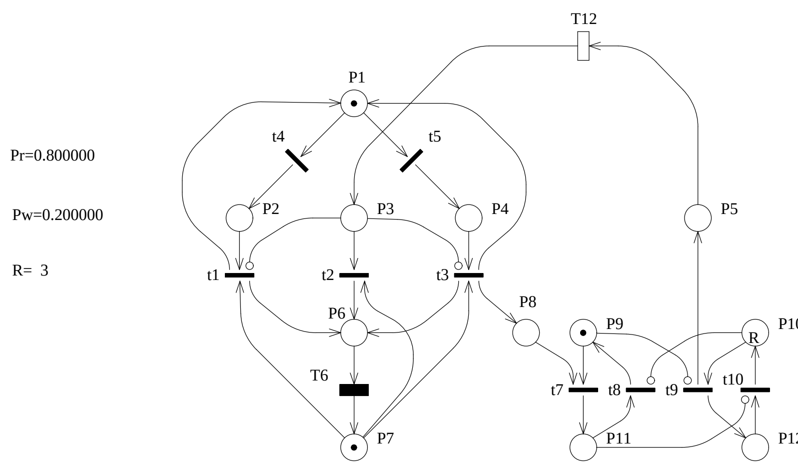


Figure 4-13: GSPN Model for Memory Replication

Figure 4-13 is a simple model for a replicated memory system. Memory operations are initiated by the token in place *P1*. Transition *t4* initiates a *read* operation with probability P_r. Likewise, transition *t5* is fired for write operations with probability $P_w = 1 - P_r$. A token in *P7* indicates that the memory is idle, while a token in *P6* signals a memory operation in progress. The deterministically timed transition *T6* models the memory access time. The arc from transitions *t1* and *t3* to place *P1* cause the memory to be used at the highest possible rate; that is, an infinitely fast processor is continuously accessing memory. Each write operation requires write cycles in all copies of the replicated memory. Since all copies of the memory behave in like fashion, only one copy is modeled. Places *P8* through *P12* form a token multiplier that places R tokens in place *P5* for every write operation. This update traffic is fed through the communication network. The network delay is modeled by transition *T12*. Update requests arrive in place *P3*.

[50]True multiported memories have several ports that can concurrently access different locations. Such devices are very expensive and are used only for special applications, such as the CPU register files. Multiported bulk memory uses multiple independent banks and resolves conflicts by blocking.

The firing rates of transitions *T4* and *t5* represent the sustained memory traffic to this replicated memory object for each node. The total memory traffic is a factor of $R+1$ higher.

The GSPN in Figure 4-13 is not structurally bounded, unless transition *T12* has bounded delay. The later property requires the inhibition arc from place *P3* to transitions *t1* and *t3*. This ensures that update traffic takes precedence over local memory access. Structural boundedness is required to ensure that only a finite number of update messages are present in the network.

Under these conditions, the local memory latency becomes

$$t_{acc} = (1-P_r)R+1 \tag{4.4}$$

where P_r is the fraction of read operation and R is the number of extra copies of the replicated memory object. Equation (4.4) expresses time in terms of the basic memory cycle time. The total memory bandwidth becomes

$$B_{mem} = \frac{R+1}{(1-P_r)R+1} \tag{4.5}$$

times the bandwidth of a single memory module. Ideally, the network delay has no impact on the memory access times because update traffic proceeds concurrently to normal operation. While this is true for purely local memory references, it is not true as far as the entire system is concerned. Nodes that do not hold a copy will have to use the network to perform remote accesses, hence remote traffic competes with update traffic and the latter increases with higher replications.

Each write operation causes R update messages to be sent through the network. Therefore, each copy of a replicated memory object must know the locations of the other copies. Naturally, it is undesirable to have a long list of pointers associated with each memory copy and inserting R update messages all at once in one node would disturb both the network and the originating node if R is large. Hence it is desirable to limit the number of pointers per copies. If this number is limited to one, all copies of a memory object form a ring. Update messages are inserted at any point along the ring and carry a simple counter that is initialized to R. The counter is decremented while the update message passes through one node. Once it reaches zero, the message has visited all nodes and is removed from the ring.

The drawback of the ring structure is the linear diameter that increases update times. Since update messages carry the time stamp of the associated write operation, this delay directly contributes to the gray zone width. Therefore, a logarithmic diameter dissemination scheme is preferable. Hence a topology is required that:

- allows insertion of updates at any node.
- has a low and bounded fan-out.
- is easily controlled by a depth counter in the update messages.

It turns out that 2-shuffles meet these requirements. 2-Shuffles have a bounded fan-out of two. Each node is the root of a fixed-depth tree that includes a unique path to all nodes; hence a depth counter can be used to avoid unnecessary transmissions to nodes that were already updated. The depth counter is used in the same way as in rings. The self-references of the diagonal nodes can be used to extend the update graph to networks sizes that are not powers of two.

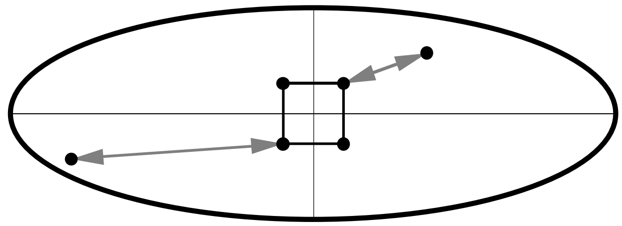


Figure 4-14: Replication Strategy

Generally, write operations can proceed concurrently with program execution because the CPU does not need any response. It is sufficient to ensure that the write takes place and that self-coherence is guaranteed. Remote write operations may cause subsequent update traffic and may face delays due to tree saturation. Therefore, a form of flow control is needed to limit the number of outstanding write operations. The simplest way to accomplish this is by returning an acknowledge message from the remote node, supplying the new value. Thus all writes are implicitly followed by a read and flow control is established. The time of the write operation is allowed to be the time of the associated memory cycle at the remote node, thus avoiding an inflation of the gray zone in case of contention. Keeping track of outstanding remote writes permits simple retries in case of expiration.

Memory replication may be either automatic or application-program controlled. This is an operating system issue and is not dictated by the hardware. Automatic allocation strategies are transparent to the application program and may employ adaptive migration policies to minimize latencies. One example of a competitive migration strategy is given in [18]. These methods require some instrumentation of the memory system, for example access counters associated with memory pages.

Application program controlled memory replication uses the access patterns of an application program to optimize the memory allocation and replication.

In either case, it was found that good replication patterns minimize the distance between copies (Figure 4-14). This approach minimizes update traffic delays, which in turn result in faster synchronization operations.

The expiration of update messages causes significant recovery problems; hence such events must be minimized. Requiring structural boundedness of the memory replication system is a partial solution. No expiration would occur when the storage provided by the message system can accommodate all update messages. Hence bounding the latency of update messages and providing adequate queue space can eliminate the update expiration problem. This requires that update messages have higher priority than other messages in the network. Since priority increases with message age, non update messages are expired when they exceed one half of the age range.

4.6.2. Processor Caches

Caching of memory objects that are local to a node does not differ from a single processor system. The black box cache of the CPU block reduces access times and memory traffic in conventional manner. This covers caching of instructions, purely local references and the processor stack.

Most of the actual cache organization is independent of the DSM organization. The features that do relate to the communication architecture are:

- The write back policy: It is required that the cache is used in *write through* mode so that memory objects can be updated as soon as possible.
- The cache must support external invalidation. This is normally accomplished by a snooping mode intended for bus-based multiprocessors. This feature will be used to remove or update cached entries from other nodes.
- Prefetching, such as loading the entire cached line, is helpful to reduce latencies. CPU blocks that can issue multiple, overlapping memory requests are ideal for DSM systems.
- Visible cache hits would benefit access counters that are used in some migration strategies. Otherwise, access frequency based migration policies could become ineffective because caches mask out some of the actual memory traffic.

Cached local memory objects are easily kept coherent by broadcasting changes to the memory object on the processor bus. However, this is not so easy once the memory object is remote. The conventional approach to this problem is to keep a directory of nodes that have a cached copy at the site of the original. Changes are subsequently forwarded to the copy holders or outstanding copies are invalidated.

Shying away from the cost and complexity of such fine-grained directories, the gray zone is used for an inexpensive but less efficient approximation. Accessing a remote memory location retrieves the associated cache line which is placed in the local cache. This copy is known to become potentially incoherent once the time of retrieval is less than *past*. Therefore, the cached copy is simply invalidated when it becomes too old.

Writing to remote locations leaves a copy in the local cache. Write operations will also return the contents of the remote memory, hence extending the lifetime of that cache entry.

Time limited caching also aids the use of spin locks. While the CPU is fast enough to swamp the network with read requests for a remote location, this rate is reduced through caching to a rate inversely proportional to the width of the gray zone. As the gray zone increases, typically a sign of contention, the access rate is reduced as caches become more effective due to longer cache line lifetimes. This self-regulation helps to stabilize the entire system under high load conditions.

4.7. Address Space Management

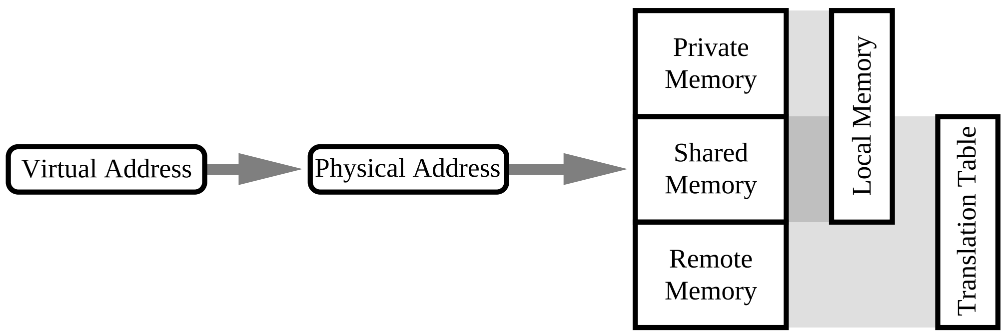


Figure 4-15: Node Address Space

A DSM system introduces one more dimension to the virtual address management: the spatial memory location. Controlling the spatial allocation of memory could be merged into a conventional memory management unit (MMU). However, the MMU functions are an integral part of the CPU block and are not easily accessible. Furthermore, managing memory allocation is quite separate from the task of maintaining virtual memory; therefore a second, independent translation mechanism is preferable.

The MMU maps the virtual address space into a physical address. This function includes the protection of address spaces, detection of page faults, etc. The virtual address space is generally larger than the available physical memory.

Mapping the physical address onto the existing memory is a simpler task because all issues related to virtual memory, protection and relocation were taken care of by the MMU. What remains is to actually locate the memory and perform the access.

The physical address space is divided into three regions (Figure 4-15). *Private* memory simply maps the physical address directly onto the local memory. *Shared* memory also maps directly onto local memory. However, shared memory is accessible from other nodes. *Remote* memory is a reference to the shared memory of another node.

The private memory space consists of *all* local memory, even though some of it may be shared. Access to private memory simply uses the physical address without further interpretation. No bound checking is necessary because this was done by the MMU.

The shared memory space is the same as the private memory space. Again, no modification of the physical address takes place. However, shared memory is organized in a number of physical pages. Each page carries a descriptor that includes a reference counter and two remote pointers. In addition to the untranslated memory access, the associated page descriptor is retrieved. The reference counter is updated for the benefit of adaptive migration or replication methods. In case of write operations, the remote pointers are examined. If any of them are non-zero, an update message is sent to that node.

Since no local translation takes place, memory reference and descriptor processing can proceed in parallel. Private and shared memory are aliases for the same storage, but descriptor processing is performed only for remote references.

Remote memory references cause a table look up to retrieve the node holding the copy. In addition, the local page number is translated to the remote page number. The burden of translating a local page number into the page number of the remote node rests with the originating node. The same is true for update traffic. Incoming messages will always have a correct local address.

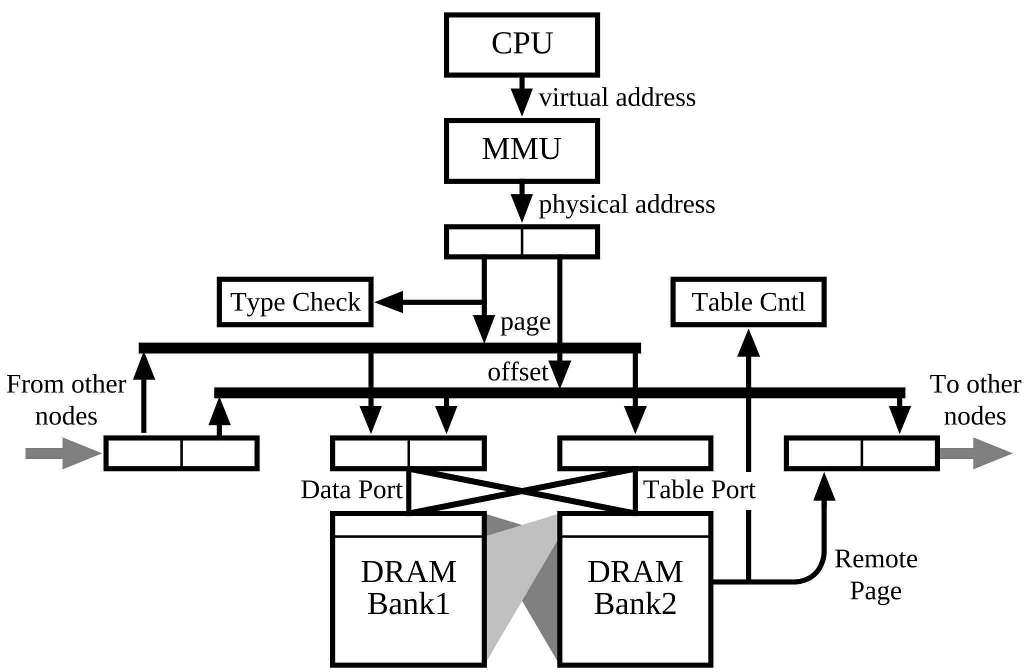


Figure 4-16: Address Translation

A block level schematic of the physical address to hardware address translation process is given in Figure 4-16. A physical address generated by the MMU is type checked with two comparators. This simple test determines whether the access is to private, shared or remote memory.

The physical memory is organized into two banks that can operate concurrently. These banks are accessible through two ports: one for normal data access (read, write) and one to access the table of physical page descriptors. A multiplexor permits swapping the port to bank assignment. The page descriptors are stored in DRAM such that the descriptors for pages in bank 1 are stored in bank 2 and vice versa. Storing the page descriptors in DRAM does not impair access speed because the descriptor is not necessary to perform the access. It only contributes to sending update messages, a process that is slower than the memory access and proceeds concurrently.

Private memory access only uses the data port. Shared memory access uses both ports concurrently and remote memory access only uses the table port. Update messages received from other nodes are always treated as shared memory access. However, the forwarding pointer is only used if the depth count of the update message is non-zero. In that case the depth count is decremented and copied to new update messages. Update messages caused by local shared memory access are initialized with the full depth count.

4.8. Performance and Evaluation

Ideally, an architecture is evaluated without assuming a specific implementation. Given a formal description of an architecture, abstract measures could be defined that can compare competing designs. Unfortunately, this ideal is rather difficult to achieve because of the complexity of the entire system.

The actual user of a computing system is concerned with the time required to complete his application program. However, this includes the effect of the operating system, the compiler, the algorithms used, the programming language, the degree of optimization, the effort to fine-tune the application for the actual architecture, the system configuration, and so on. Separating the contribution of each element is not really feasible; hence it is common to make simplifying assumptions or to use abstract models for the application and/or the architecture.

Algorithm analysis for PRAM programs assumes uniform cost to access memory. However, the access delays for DSM implementations vary widely. A cached memory location can be accessed within one or two processor cycles, while a paged DSM may require up to 3 msec for a fault, roughly 75000 times longer [125].

Predicting the performance of applications and assessing the architectural merits of a DSM implementation requires a better cost model. Ideally, the cost model of a particular architecture would accept a number of implementation-specific parameters, such as memory speed, message system latency and bandwidth, cache sizes, etc. Given the characteristics of the application, such as memory access patterns, the average cost for memory references could be computed.

Agarwal [4] evaluates the memory access transactions in terms of the required data transfers. One bus transaction, in the case of a bus-based shared memory system, is the cost unit. The frequency and type of memory access for particular applications is extracted from application address traces.

This approach cannot be used to evaluate a network-based DSM because the message system latency is much more dynamic and depends on the access pattern. Therefore, direct simulation of synthetic and real applications is used to assess the performance of a particular instance of the proposed architecture.

Evaluating a communication architecture based on the simulation of a particular implementation requires that several implementation parameters are fixed. The following set of implementation assumptions was used:

Memory speed: Dynamic memory with a cycle time of 150 nsec was modeled. Access times were assumed to be 100 nsec. Cache hits are assumed to require one clock cycle of 50 nsec (using a 20 MHz clock). The data cache is two way set-associative with a total of 256 words or 1024 bytes.

CPU: Because all experiments were run on SUN 3 workstations, a 68020 CPU was modeled. This greatly simplified the simulator (see Appendix A.3). The target CPU was assumed to run at 20 MHz.

Message system speed:
All channels were assumed to transfer one byte per clock cycle (50 nsec). This results in a message cycle time of 450 nsec. Four transient buffers were provided.

These assumptions are fairly conservative. In most cases, the absolute performance is less interesting than the relative changes in performance when various system parameters are varied (for example speed-up vs. number of processors).

The simulator is able to alter the relative speeds of CPU, memory and message system freely, but there are simply too many degrees of freedom to explore these tradeoffs.

Bisiani and Ravishankar [17] performed an independent evaluation of an earlier version of the proposed communication architecture. This research used a different set of applications.

4.8.1. Strategy

The proposed communication architecture was evaluated with four different applications. Each was chosen to illustrate a particular aspect of the architecture.

A *synthetic workload generator* was used to exercise the system with a randomly generated sequence of memory references. The objective was to give rough estimates of the speed of each operation type.

The *matrix multiplication* example demonstrates how a message-passing implementation can be translated efficiently into a DSM implementation.

The *c2fsm* example demonstrates one approach to the common problem of distributing a workload composed of many small subtasks that are dynamically generated. The use of a distributed task-queue avoids the hot-spot of a centralized resource.

The *printed circuit router* application uses shared memory to store a common database and private memory to work on sub-problems. Easy access to a common data-structure is one of the virtues of the DSM approach.

4.8.2. Synthetic Workload Generation

The primary measures of interest are the latencies for the read and synchronization operations. The latencies for write operations is usually zero due to the effect of write buffers. Delays during write operations were observed only once when more than four write operations were issued back-to-back. Since this happens rather infrequently, these statistics are omitted.

Caching was turned off, because it is the communication architecture that is being studied. The net effect of caching is to reduce the memory access frequency or to allow the CPU to execute more memory references. The impact of memory caches is relatively well-studied and it is beyond the scope of this thesis. Instruction fetches and stack references were considered to be cache hits at all times.

The workload generation is controlled by several parameters:

Load: A random delay that follows a negative exponential distribution was used to model CPU time spent between memory operations. This parameter could be related to the cache hit rate in a real system.

Fraction of write operations: It is common to assume 20% writes and 80% reads. All results shown below are based on this ratio.

Fraction of remote operations:

This parameter controls the ratio of local and remote memory access operations.

Fraction of synchronization operations:
A certain fraction of synchronization operations is inserted. The reported time for these strongly coherent operations (for example a *test-and-set*) include the verify part. No operations were issued between the origination and the verification part.

Access pattern: The access patterns used to exercise the message system were implemented. However, due to the uncoordinated nature of this experiment, most results were quite similar. Therefore, only the random access and the hot-spot pattern results are included.

Replication pattern:
Memory was replicated such that the copies minimize their topological separation. This strategy achieved the best results compared to other allocations. The number of memory copies was varied.

Network topology: While all topologies mentioned in Chapter 2 were implemented, only the results for binary hypercubes and k-shuffles are presented. Experiments with other topologies resulted in similar data.

The first two graphs in Figure 4-17 show the impact of network load on the read latency to remote memory locations. The local memory traffic is part of the simulation, but its characteristics are very similar to those of a conventional memory subsystem; hence it is not presented here.

Increasing the network load is done by reducing the delay between memory operations. Due to the random access pattern, the memory system will saturate before its nominal capacity. Some node memories will receive multiple requests while others remain idle. This temporal load imbalance is common to all irregular, dynamic access patterns and lead to a reduction in useful memory bandwidth.

Evidence for graceful degradation under load saturation is the limited latency increase as memory saturation is approached. The adaptive communication resource allocation has a limited load balancing effect that is demonstrated by the slope reversal of the *average* latency as memory saturation occurs. Figure 4-20 shows the actual latency distribution for a 16 node binary hypercube system. The worst average latency occurs at an offered remote load of 400.000 remote references per second by each node. The average latency for an offered load of 1M remote references has a slightly lower average[51]. However, the total number of operations is slightly lower.

The hot-spot experiment results shown in Figure 4-17 fail to detect any significant degradation. Even at very high hot-spot concentration, no network induced effects are visible.

---

[51]An offered load of 1M references is practically equivalent to an infinitely fast processors that causes a continuous stream of back-to-back operations.

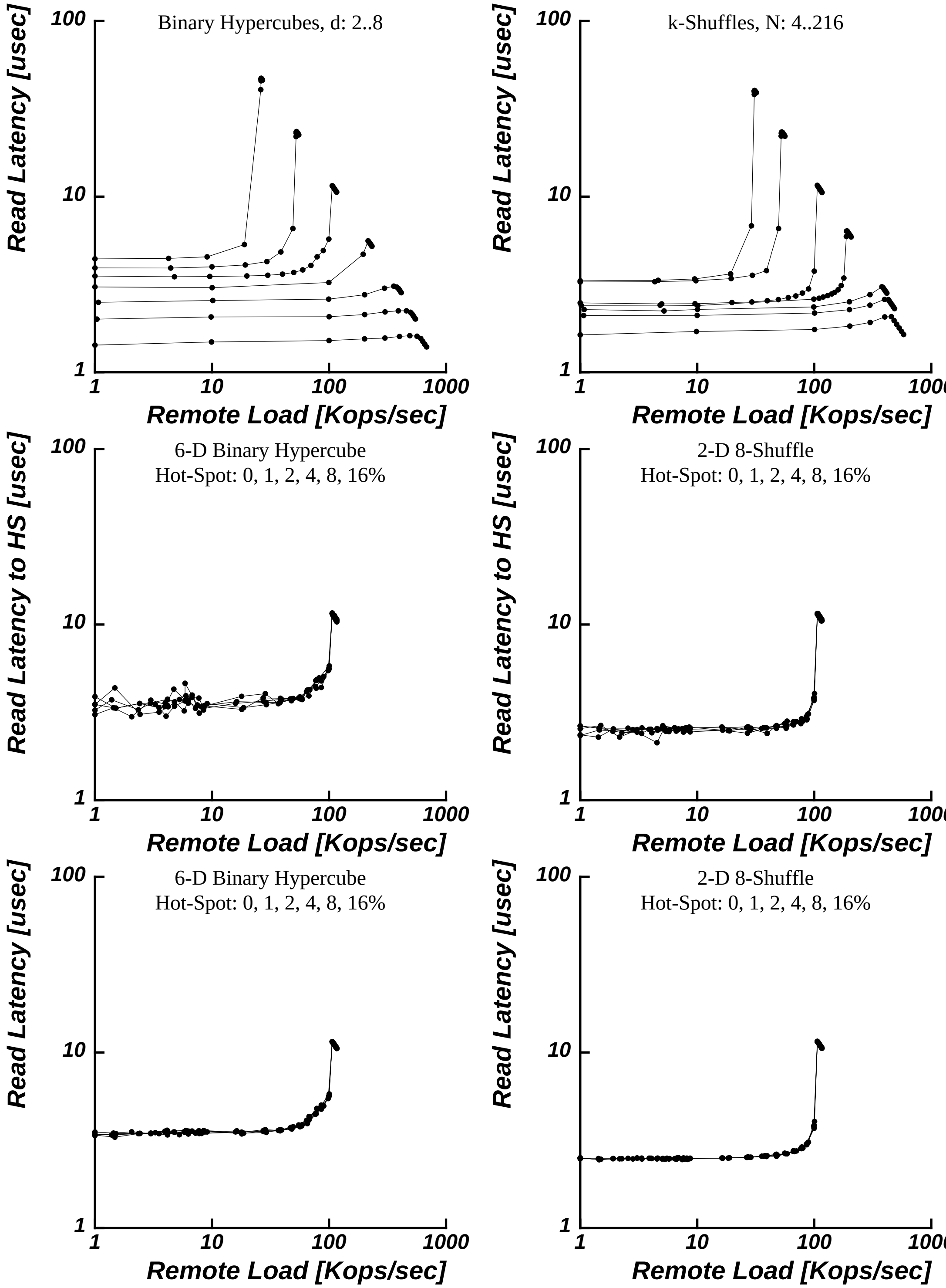


Figure 4-17: System Performance: Topology and Hot-Spot Effects

Replication is remarkably effective at reducing latencies at high loads (Figure 4-18). In fact,

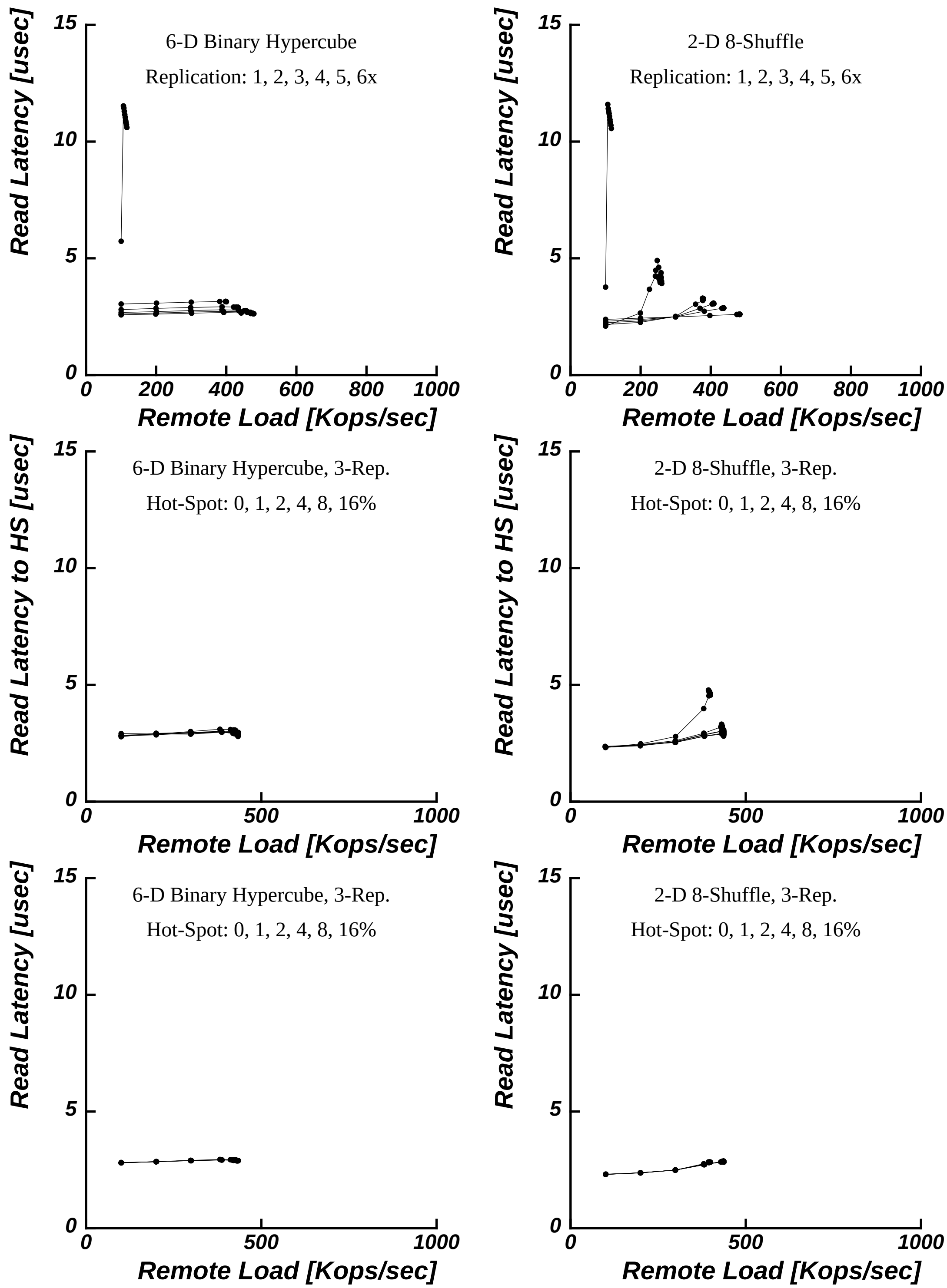


Figure 4-18: System Performance: Impact of Memory Replication

a replication factor of 2 was sufficient to avoid memory saturation in a 64 node binary hypercube system. In this case, the latency is ultimately limited by the transmission delays,

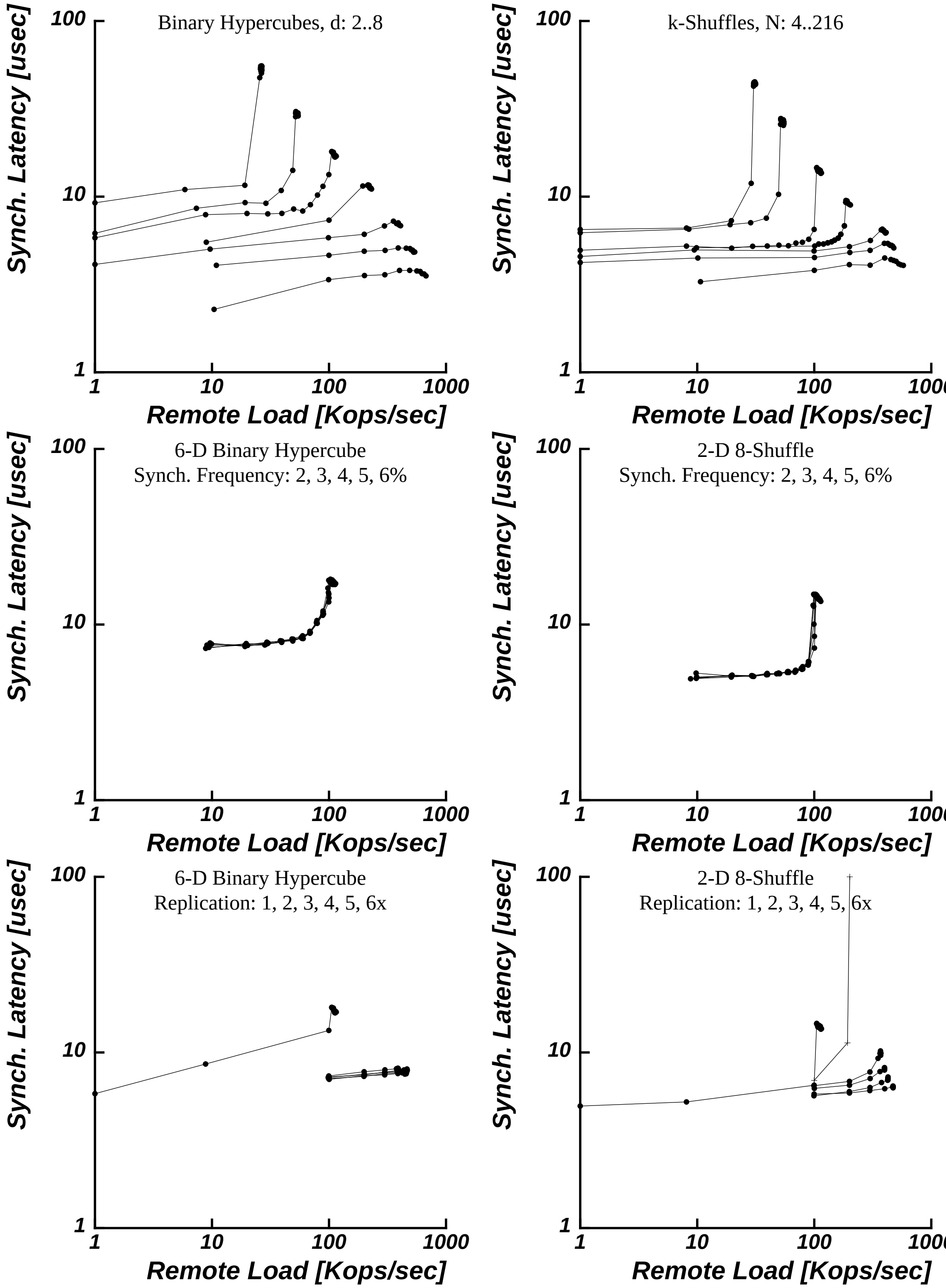


Figure 4-19: System Performance: Synchronization Speed

which did not change noticeably even when references were issued back-to-back with no processing delay. Replication is less effective in the k-Shuffle due to limited impact of

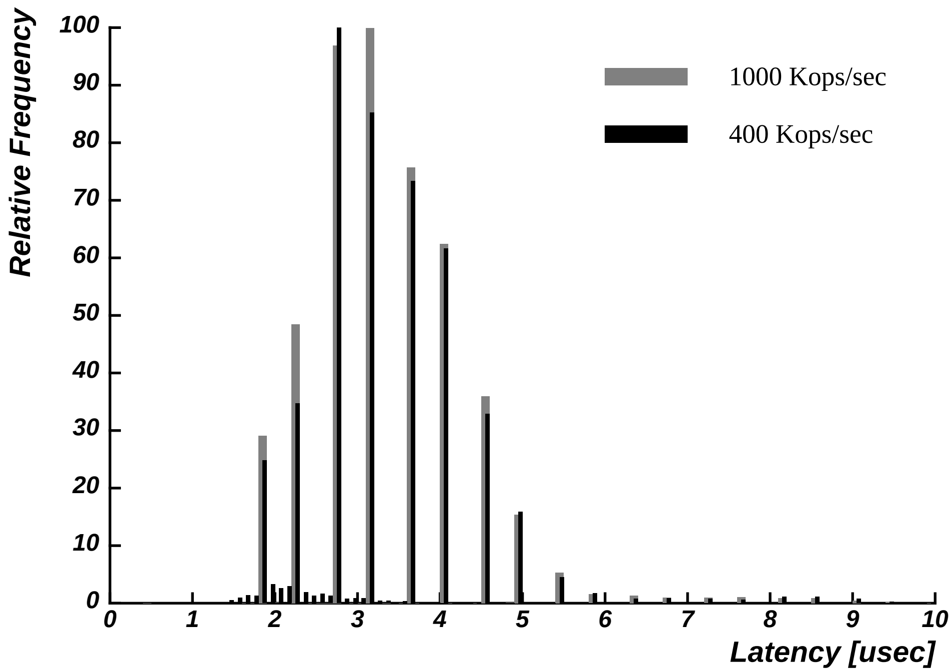


Figure 4-20: Remote Read Latency Distribution

adaptive routing in this topology family. In fact, the update traffic associated with the write operations causes higher replications to perform worse at high loads in k-Shuffles.

Since replication allowed a higher traffic at saturation and because of the extra update traffic, degradation with a 16% hot-spot traffic pattern became noticeable. Figure 4-18 shows that k-Shuffle are more susceptible to this problem, which is due to the fact that this topology has no alternative paths that could be used in adaptive routing. Even in this case, only the traffic directed to the hot-spot sees a change in the latency. Traffic to other nodes remains unchanged.

The latencies for synchronization operations follow are similar to normal references. Again, hot-spots are a non-issue for modest sized systems and replication is effective is reducing latencies. However, high replication factors cause update traffic that increases the gray-zone and hence synchronization times. In Figure 4-19, this effect is visible for the 8-Shuffle topology: at 6-fold replication, synchronization latencies increase dramatically (a '+' mark denotes this graph).

4.8.3. Matrix Multiplication

Dense matrix multiplication is an *embarrassingly* parallel problem that is easily ported to any parallel architecture and runs with near perfect speedup. Because of its simplicity, matrix multiplication is useful to illustrate optimization techniques for DSM systems.

The basic $O(n^3)$ algorithm runs directly on a DSM system without modification:

```
double A[n][n], B[n][n], C[n][n];

for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        t = 0.0;
        for (k = 0; k < n; k++)
            t += A[i][k] * B[k][j];
        C[i][j] = t;
    }
}
```

To exploit parallelism, either outer loop would be distributed over all available processors. Since all matrices reside in shared memory, no explicit communication is necessary. However, the performance of this naive parallel implementation is impaired by the frequency of read operations to remote structures. A directory-based DSM would suffer the full latency for remote access on each read because the probability of data reuse decreases inversely proportional to the square of the number of processors, rendering caching ineffective.

DSM with support for memory replication may simply allocate one copy of the matrices in each processor. However, this approach is quite wasteful in both memory and bandwidth usage. Memory demand becomes proportional to the number of processors and bandwidth is required to update all copies. The total required memory write bandwidth per node would be equal to that of a single processor implementation of the entire problem.

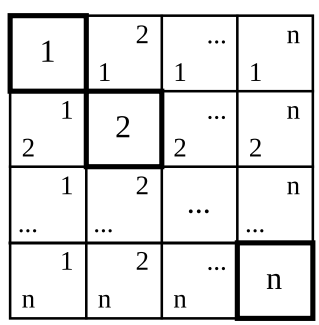


Figure 4-21: Matrix Allocation for Replicated Memory DSM Systems

An optimal allocation pattern for replicated memory DSM systems is given in Figure 4-21. Each part of the matrix is stored in, at most, two nodes. The matrix is partitioned in *N* rows and columns, where *m* is the number of available processors. In addition, each processor has two scratch arrays to store one column each.

Each processor starts by computing the diagonal submatricies. All required data is local; hence processors can proceed at their nominal speed. Processors then proceed with the next submatrix in the same row. Submatrix enumeration wraps around so that after N steps each processor has computed all submatricies in one row. The code is modified such that iterations $2 \cdots N$ operate out of the scratch array. Using double buffering, each processor receives the column data from its left neighbor, uses it, and passes it on to the right neighbor. Thus processors are organized in a ring. The scratch arrays are initialized in the first iteration:

```
for (i = ...) {
    for (j = ...) {
        t = 0.0;
        for (k = 0; k < n; k++) {
            tmp = *right++;
            *left++ = tmp;
            t += tmp * B[k][j];
        }
        C[i][j] = t;
    }
    ....   /* producer/consumer synchronization
              omitted */
}
```

In this version, communication uses only remote write operations to remotely stored memory. Since writes can proceed concurrently with program execution, the algorithm is not slowed by network latencies. Essentially, the message-passing version of matrix multiplication is used. Only pairwise, producer/consumer style synchronization is require, which is easily accomplished without special synchronization operations. The pairwise synchronization implicitly barrier synchronizes the algorithm.

This example demonstrate how methods developed in a message-passing environment can be employed in DSM systems. It also shows that control over memory allocation, in particular if replication is supported, can be used to match the algorithm structure. Finally, program development is aided by the ability to run the naive implementation directly. Profiling may reveal that this particular operation is too infrequent to warrant parallelization at all.

In the case of a 256 by 256 matrix (Figure 4-22), the speed-up of the efficient memory access implementation is eventually degraded by the overhead for the necessary address computation. Given that there is very little work performed in each loop iteration, the loop overhead and related address computation degrade performance severely. In the 256 processor case, the performance actually drops below that of a simple implementation that uses the conventional shared memory algorithm. Increasing the problem size alleviates this problem (Figure 4-23).

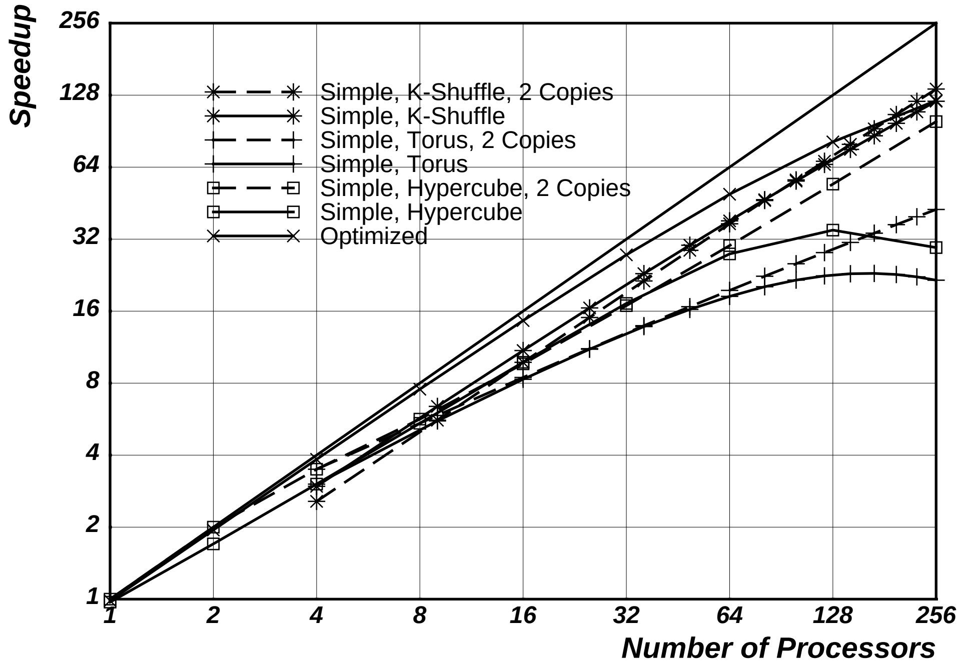


Figure 4-22: 256 by 256 Matrix Multiplication Speed-up

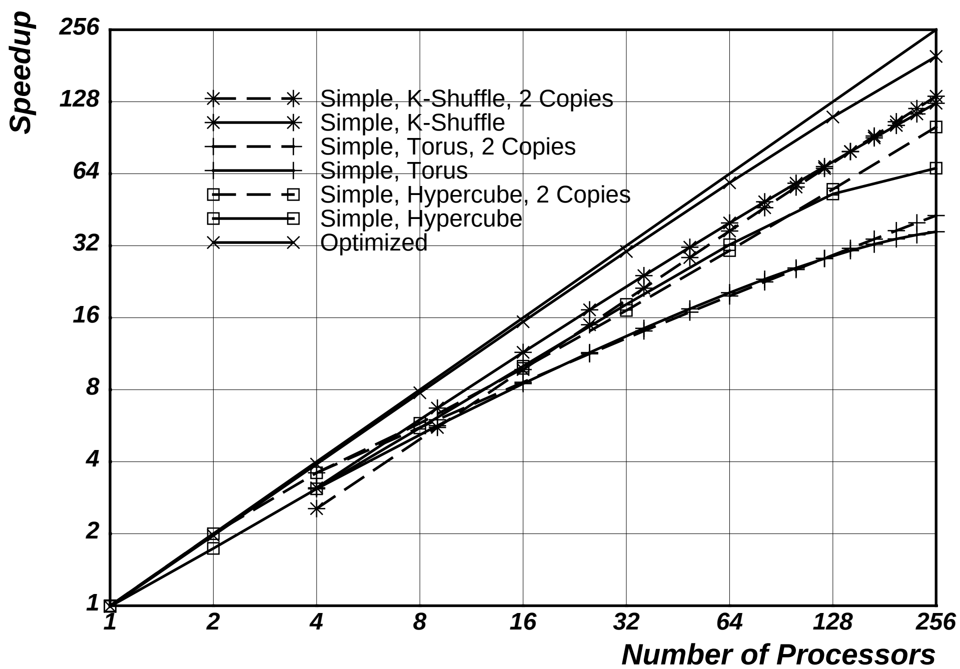


Figure 4-23: 1024 by 1024 Matrix Multiplication Speed-up

4.8.4. C2FSM

C2FSM [102] translates a C-language description of a finite state machine into a boolean representation, suitable for direct implementation in hardware. The translation process starts with an initial state description and applies a specified transition function. Given a particular set of input signals and the current state of the machines, evaluating the transition function produces a new state. For each state, the C2FSM core calls the transition function systematically with all possible input combinations and generates the set of possible next states. The enumeration of input combinations to the transition function employs an algorithm that senses whether a particular input signal is significant in a particular state. Input combinations that don't affect the state transition are not evaluated; hence the state expansion complexity is usually much less that $O(2^n)$, where n is the number of input signals.

The expansion step is moderately complex and involves a number of boolean operations, plus it uses the externally defined transition function which is a black box. Calling the transition function has the side effect of partitioning the input signal space. While this procedure is greatly reducing the number of required calls to the transition function, it is also inherently sequential. Therefore, parallelizing the expansion step is not feasible.

Expansion produces a number of new states which are added to a data structure of reachable states. The C2FSM core proceeds to expand each state in this structure, thereby potentially adding new states to it. The process terminates once all states are expanded. The total number of states is not known when the process is started.

The obvious source of parallelism is the concurrent expansion of multiple states. Therefore, a parallel DSM implementation of C2FSM faces two problems:

- The maintenance of a shared task queue of states that are awaiting expansion.
- A global data structure of all known states that is accessible from all processors. Each processor must be able to add states to this structure.

Multiple reader, multiple writer task queues are a recurring problem and most parallel algorithms employ locks to ensure mutually exclusive access to a critical region or use special atomic operations (*fetch-and-add*).

In this case, a distributed task queue composed of single writer, single reader queues is used. Figure 4-24 depicts a network of nodes connected by queues. Each node reads from one or two queues and writes to one or two queues. The underlying strongly connected, directed graph is optimized for a low diameter. Each processor reads from the queue with the most tasks and writes to queues with the fewest tasks. During de-queue operations, the sizes of the input queues are compared to the sizes of the output queues. If the difference exceeds a threshold, work is directly passed on to the output queues.

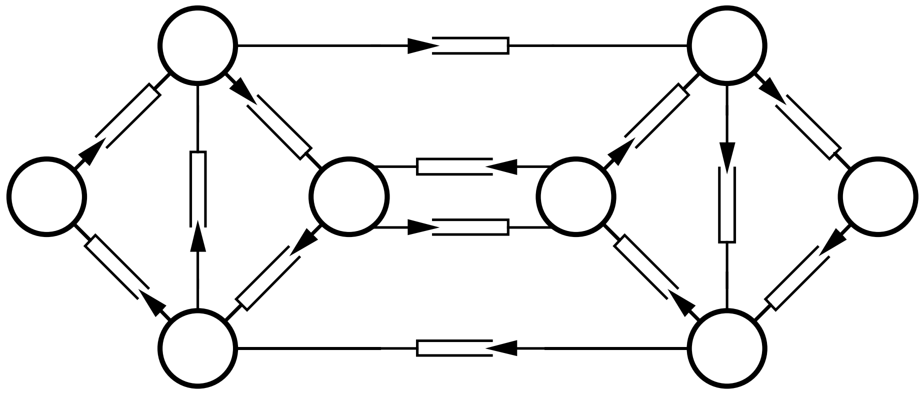


Figure 4-24: Distributed, Load Balancing Task Queue

In general, this diffusion balancing mechanism may sort tasks by the location of the associated data structure and prefer those that are local. C2FSM state descriptions are small and the cost for accessing them is negligible compared to the expansion processing time.

This distributed task queue does not require special synchronization operations: simple *one-producer-one-consumer* queues are sufficient. Furthermore, the distributed nature avoids memory access contention.

C2FSM uses a binary tree for the state data structure. The parallel implementation uses a hash table. Most states are found in the hash table; hence updates are relatively infrequent. Therefore, the hash table space is subdivided into a number of regions with write locks to prevent concurrent updates in the same region. Since updating is done by changing a single point, read access is not blocked during updates.

Figure 4-25 is a typical example for the parallelism found in C2FSM. The benchmark problem is the encoder finite state machine given in 3-14 where *DCT* was set to 8 in order to increase the problem size. This results is a FSM of 1118 states. Beginning with one unexpanded state, the number of states that could be processed in parallel quickly rises to about 60. Hence at most 60 processors could be used for this problem.

The ideal performance of a parallel expansion implementation is given by the solid line in Figure 4-26. This is based on a static analysis of the state expansion process without taking any synchronization and/or memory contention losses into account. The actual performance is given by the lower solid line. A binary hypercube implementation was used and the task-queue topology is that of a 2-Shuffle. The grain-size of this problem is fairly large (average task is 2 msec between synchronizations).

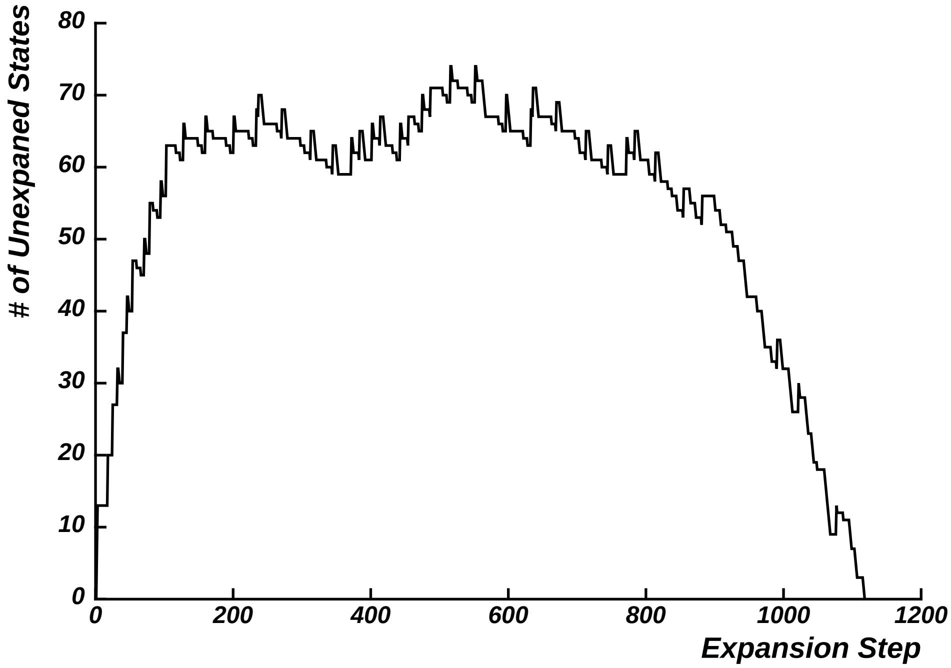


Figure 4-25: Parallelism in C2FSM

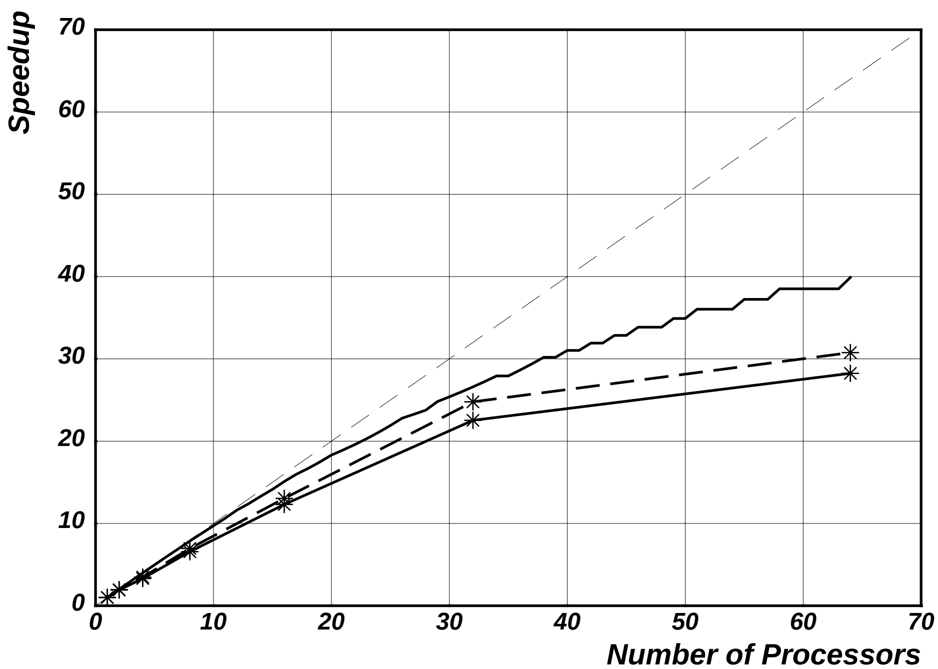


Figure 4-26: Distributed C2FSM Speed-up

The dashed line in Figure 4-26 is the result of simulation with a changed processor to network speed ratio. Increasing network speed results in a reduced degradation due to communication delays. Increasing network speed by almost a factor of 10 results in a 7% to 11% performance increase. This indicates that this benchmark is largely limited by processor and memory speed and not by the communication architecture.

4.8.5. PCB

PCB is a bit-map based printed circuit board editor and router [103]. The primary data structure is a set of bitmaps, organized as an array of bytes where each bit corresponds to one of eight different bitmaps. Circuit traces are represented by setting the appropriate bits in one of the signal planes. If adjacent bits are set, they are connected. In addition to the bitmap, a database of the component shapes, locations and the desired connections is maintained. PCB enforces design rules such that no unrelated signals will be connected.

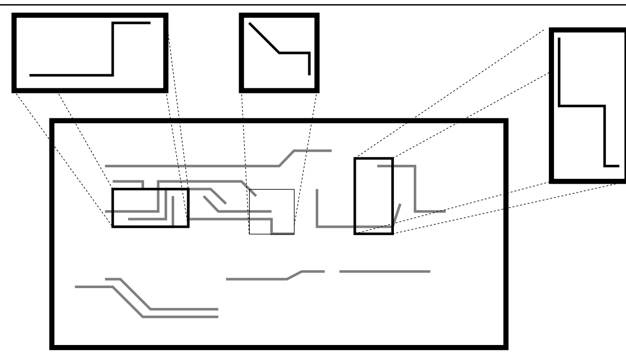


Figure 4-27:

PCB's router is a fairly conventional maze runner. For each desired connection, a sub-bitmap is created from the main bit map that encloses the end points of the net to be routed plus a programmable border. The sub-bitmap is not a direct copy of the main bitmap, rather a translation process takes place that identifies the potential space for a new trace, finds all legal places for through holes, and marks areas that are connected to the net under construction. The sub-bitmap is subsequently expanded under control of an evaluation function that takes the preferred trace orientation, global congestion etc. into account. Once the wavefront expansion finds a viable trace, a backtracking path updates the main bitmap.

The parallel implementation assigns different nets to different processors (Figure 4-27). Each processor translates the relevant section of the main bit map and proceeds to execute the same algorithm. The scheduling process tries to minimize overlap of individual sub-problems. If overlap is detected, the associated sub-processors use locks to prevent concurrent update. Since the sub-problems are processed independently, there is a potential for conflicts in overlapped regions. Therefore, updates include a verification path before the main bitmap is altered. If the new trace is found to conflict with the altered main bit map, the route is discarded and a new routing attempt is made.

The implementation is asymmetric in the sense that one node coordinates the work by distributing sub-problems. Hence the minimal configuration requires two processors.

PCB is a fairly large-grained test case. Given the limited parallelism in this example plus the memory size limitations of the simulator, only 16 node simulations were feasible. The simulation was driven by layout data from a 568 net design[52]. Direct simulation of the routing process was not feasible because the simulator slows down the execution of application code by about 100.000 (about 10 msec simulator CPU time per remote memory reference).

# of Nodes	Relative Speed	Read Latency	TAS Latency	FADD Latency	VFY Time
2	1.0	0.71	1.3	0.60	1.0
4	2.3	0.99	1.5	0.84	1.4
8	4.9	1.3	1.9	1.2	2.1
16	9.7	1.5	2.4	1.4	2.9

Table 4-1: PCB Simulation Summary

The PCB simulations are only of limited use for the evaluation of the communication architecture because they depend on numerous unrelated factors, such as algorithm design and implementation, problem selection, decomposition. Hence the summary in Table 4-1 gives only a brief overview.

The real value of this work was in the development and the refinement of the architecture. Simulator configurations and instrumentation were frequently changed to track down unexpected delays and interactions. A complex code like PCB was stressing the limits of RTL-type simulations on conventional machines. In particular, it proved impossible to study dynamic page migration algorithms due to prohibitive simulation times and memory requirements.

[52]A meta-stability tester that was layed out with PCB and fabricated by the MOSIS PC-board service [104].

4.9. Implementation Issues

The network interface (NI) is largely a collection of finite state machines clustered around straight forward data path (Figure 4-11). The simulator of the communication architecture actually implements these finite state machines at the register transfer level (Appendix A).

The largest components of the NI are the T-buffer and the synchronization support circuitry. The data path, memory controller, bus interface and glue logic are relatively mundane, probably within reach of contemporary logic synthesis tools.

4.9.1. System Integration

Ideally, the processor for a DSM system would be specifically designed to match the communication architecture. However, given the design cost and complexity of contemporary processors, this approach is hardly feasible. Instead the communication architecture must match the chosen processor. Potential node CPU's include the Motorola 88000 series, Mips's R3000, and SUN-Microsystem's Sparc.

Problems arise in these areas:

- The instruction set does not support specialized memory operation. For example the distinction between strongly and weakly coherent memory access is not part of any instruction set.
- The cache organization does not match. The high integration levels of current VLSI resulted in black box cache chips that included practically the entire cache circuitry, memory, address translation, etc. Unfortunately, it also hides all internal states.
- The processor architecture and its memory interface demand very small access times.

Most CPU architectures include atomic *Test-and-Set* or *Exchange* instructions that cause a special bus cycle. However, the communication system may prefer different semantics. For example, efficient synchronization with a snoopy cache system requires a *test-and-test-and-set* instruction. Communication architectures based on weak coherence will require special synchronization instructions. The proposed communication architecture can be modified easily to support a specific set of synchronization primitives. However, maintaining the division of these operations into a non-blocking initiation phase and a blocking verification phase could boost performance.

One method to overcome the instruction set mismatch problem is the use of special memory locations that have the desired access semantics. This requires bypassing caching and address translation, dedicated registers and decoders, plus at least two operations: one to provide the properly translated address and one to store or load the data. Interrupts may not separate this sequence and memory exceptions can complicate matters. The entire operation is significantly slower than normal accesses, a penalty that is acceptable only in infrequent cases.

Since the native *synchronization* instruction is visible on the bus, external circuitry can recognize this sequence and alter the semantics. For example, Motorola's MC88100 provides an atomic exchange instruction that could be used as a vehicle for customized instruction. Issuing the *exchange* with an operation code as *data* will cause a properly translated address on the processor bus along with special signals that identify the atomic bus operation. The external circuitry can subsequently interpret the data on the bus as an operation code or as a combination of op-code and data sharing one word. The bus cycle is completed with data supplied by the external circuit.

The most difficult situation exists if the integrated cache controller has no provisions for external invalidation. Fortunately, snooping cache protocols are sufficiently well-established that recent cache controllers include a snooping option. For example the MC88200 cache and address translation unit support snooping, albeit at the expense of processor access bandwidth.

Even with support for external invalidation, black-box caches may still cause problems. For example, a memory system that uses reference counters to assist page migration strategies will be unable to count references to cached items. This severely degrades the competitive page migration method given in [18].

Remote memory reference latencies may exceed 1 μsec, a long time for a fast processor, but not long enough to justify a normal context swap. Frequently, the memory system could handle multiple requests concurrently. Therefore, processors that issue multiple addresses in advance could greatly ease the memory system complexity. Apparently such support is planned for the next generation of the 88K series. However, given a single instruction stream, the number of outstanding memory requests is limited by data dependencies.

The ideal processor for a DSM system has multiple sets of registers, program counters and is able to switch between a small number of active threads. General purpose multi-micro tasking was used in the HEP multiprocessor [134]. Built with off-the-shelf components, HEP's cost performance ratio was not competitive. The principle of low-overhead context switching as a means to cope with memory and network latencies is sound if the application has sufficient parallelism.

4.9.2. The T-Buffer

The *T-buffer* of the NI (Figure 4-11) maintains the set of time-stamps associated with the most recently accessed memory locations. It is connected to the internal bus of the NI to monitor all memory references. Since there is an upper limit on the gray-zone and because there are only a finite number of memory cycles possible within that period, the total number of entries in the T-buffer is limited to about 200 entries.

Each entry contains the time of the most recent write operation and the processor ID to disambiguate multiple writes issued during one clock cycle. Depending on the actual system parameters, this amounts to about 16 bits per entry.

T-buffer entries may be associated with any memory location within the scope of one NI. This implies an organization similar to that of a tag-store for a data cache. However, all write operations must be recorded in the T-buffer for the duration of the gray zone. A fully associative organization solves this problem, but it is rather costly. Alternatively, a potentially larger T-buffer may partially direct mapping with multiple associative sets. Implementations that cannot hold an arbitrary set of memory locations must have facilities to deal with overflows. Since overflows are rare, a sequential search of an overflow area is viable.

Given proper pipelining, the T-buffer can operate in parallel with the DRAM. Therefore, it can be as slow as the DRAM, which is much less than the speed of typical tag-stores for data caches. The design of the T-buffer may use this time for sequential tag-compares, thus using only one comparator for all sets.

4.9.3. Synchronization Support

Synchronization operations require a *snooper* on the internal bus of the NI that monitors the memory access for the duration of the gray zone. Once a synchronization operation is started, a snooper is assigned to the accessed memory location. It records the final state of that memory location just prior to the instruction execution time. All write operation that predates the synchronization operation will update the snooper but not the memory location. Once the gray zone reaches the initiation time, the snooper returns the final result and the synchronization operation is completed.

One snooper is required for each concurrent synchronization operation within the gray zone. At least two snoopers are required, one for locally issued operations and one to serve remote requests. It is desirable to support more than one concurrent local synchronization operation. A total of 8 to 16 snoopers appear appropriate (the experiments were run with one local and an unlimited number of remote snoopers).

Each snooper contains the address being monitored, the data word, the operation type, the time of execution and a time-window information. This amounts to about 80 bits of state plus a small finite state machine.

4.10. Summary

Conventional private memory machines do not fully benefit from faster communication channels and improved routing technology due to the high overhead of explicit *send* and *receive* operations. Attempts to reduce this overhead to match the speed of the message system resulted in complex and expensive I/O processors.

The presented architecture achieves low communication overhead through implicit message generation to support distributed shared memory (DSM). Unlike true shared memory machines, processors are not able to access the memory of a remote node directly, thus preserving the simplicity, scalability and cost-effectiveness of private memory machines. Like true shared memory machines, the memories of all nodes are part of one global address space, thus keeping the convenient PRAM programming model.

The key to the performance of this DSM architecture is close integration with a message system that supports highly dynamic traffic of predominantly short messages. Given the message system described in Chapter 3, good performance for remote memory references was achieved, which is unlike most network-based DSM systems that cannot support this low-grained communication.

Other significant characteristics include:

Coherency support: Memory access normally is weakly coherent. This allows the system to hide the latency of the underlying communication network. Evidence was presented to show that most data references of common application programs only require weak coherency. A set of strongly coherent operations is supported for synchronization and program control. These functions are slower because they cannot hide network delays.

Symmetric memory replication
Memory can be replicated to reduce latency and increase access bandwidth in a non-hierarchical fashion without designated ownership.

Compatibility with message-passing systems:
It was shown that algorithms developed for message-passing machines translate easily into code for a DSM system by substituting *send*-operations with remote write operations.

Simplicity: The set of operations required for this DSM system is small and well-defined. It is therefore possible to implement them directly in hardware.

Time: Memory coherency is based on time-stamping all state changing transactions with a global, precise clock. Consequently, the coherent state of the system is always define. The clock is accessible by all processors and may be used for synchronization purposes. Furthermore, all processors are supplied with a tight bound on all pending memory transactions. Operations beyond this bound are guaranteed to have finalized.

Based on conservative technology assumptions, simulations of synthetic and realistic applications were used to evaluate the expected performance. The experiments show that most of the message system performance is made available to the application program. Overall performance is largely limited by the speed of the memory. As expected, the speed of synchronization operations depend primarily on the network latencies.

Chapter 5
Conclusion

As circuit and implementation technologies slowly approach physical and economic limits, parallelism will become important for future performance improvements. Two broad classes of parallel machines have emerged: shared memory systems that use switching networks between sets of processors and memories, and private memory systems that use a message-passing network between processing nodes. The former class of systems appeals by virtue of its simple programming model, which is a natural extension of conventional uniprocessors. The latter class features more cost-effective and scalable implementation.

This thesis research attempted to combine the advantages of both approaches. Preserving the shared memory model with its implicit, low-overhead communication properties leads to an increased demand for communication bandwidth. Therefore, most efforts were directed toward the development of a message system for optimal communication channel utilization under high-load conditions.

The design of the message system was influenced by the opportunities and constraints of custom VLSI technology, namely the ability to integrate complex circuits that are limited by the number and speed of the I/O signals. It was further optimized for the irregular traffic of very short messages that can be expected from a distributed shared memory system.

Coherence of a distributed shared memory system can be achieved through the notion of time. Time stamps provided by a precise clock maintained by the message system are used to ensure a well-defined global state. Essentially, all operations of the proposed system are ordered by time and processor numbers. A form of weak memory coherence was introduced to allow concurrent communication and computation. Strongly coherent operations are provided at the cost of full communication delays.

5.1. Contributions

A systematic review of the bandwidth and latency tradeoffs for a representative collection of network topologies was presented. Optimizing networks for optimal bandwidth conflicts with minimizing latencies. This conflict can be avoided if traffic is broken into small packets that are allowed to traverse the network independently. By concurrently utilizing multiple, independent paths through the network, high fan-out / low-diameter topologies have both maximal bandwidth and minimal latencies.

A message system using synchronous communications of minimal size, fixed-length packets was conceived. Aided by a new adaptive routing strategy, useful network loads in excess of 90% of the network capacity were achieved. Unlike other adaptive routing methods, an anticipatory evaluation of resource assignments is used. Routing decisions that are likely to constrain other traffic are avoided.

Using inexpensive media, the message system was generalized to link clusters of tightly coupled processors over a local area in a functionally transparent fashion.

Adaptive routing is supported by a practical topology-independent method of deadlock avoidance through resource ordering. This heuristic is not guaranteed to find a deadlock-free virtual channel assignment but was successful on all networks of practical interest.

Methods for precise clock distribution were presented. This clock distribution can be extended over a local area in a symmetric fashion. All nodes that join the network will synchronize to one common clock that does not depend on any particular node.

A simple, time-based programming model for weak shared memory coherence was developed. Transient incoherence was allowed to enable overlapping of communication and computation. Strongly coherent operations are supported but require more time as the network delays become visible.

5.2. Future Work

The actual implementation of a multiprocessor system based on the proposed communication architecture is a natural continuation of the research presented in this thesis. Such a project will face a number of problems that were not addressed. Among these are issues related to the actual processor, the operating system, the I/O devices, etc. However, there are a number of other loose ends that deserve attention in their own right.

The area of performance evaluation of communication architectures lacks standardized benchmarks and measurement methods. Measured and simulated results of proposed and implemented architectures are frequently not comparable due to different assumptions and methods. While several synthetic and actual benchmarks are now in common use to assess the performance of processor architectures, caches and memory systems, no such suites exist to compare multiprocessor communication designs.

The use of virtual channels for deadlock avoidance through resource ordering is used in many networks. Constructive methods to find deadlock-free virtual channel assignments are known for specific topologies and this thesis presented a heuristic to deal with arbitrary topologies. It is not known, however, how many virtual channels are minimally required for a deadlock-free assignment of an arbitrary graph. Conversely, the set of network topologies that have deadlock-free, minimum-path assignments with no more than two virtual channels is not known.

Research on efficient synchronization operations has not yet found universally satisfactory solutions. This thesis added another option: time. The primary function of the precise maintenance of time is to ensure memory coherency. By allowing each process to access an accurate, global clock, it should be possible to design efficient, high-level synchronization algorithms without widely shared variables.

Appendix A
On Simulation Tools

Preliminary studies were carried out with normal C-programs [73]. However, it became obvious that using plain C would result in large, ill-structured programs. Before writing a general purpose simulation environment (scheduler, light weight process support, etc.), possible alternatives were considered. It was clear from the beginning that simulation would be a rather CPU-intensive task; hence efficiency was of primary concern. In the end, using the C++ language [141] appeared to be the most promising approach. Despite several compiler bugs and some shortcoming of the language, C++ turned out to be a blessing.

A.1. The C++ Task Package

Unlike Simula, the bare C++ language has no notion of concurrency. However, a separate library of classes is available to support co-routine style programming. Each object of the *task* class maintains its own context and stack. The maximal stack size of each task must be defined at construction time. This storage is allocated statically for the lifetime of the task object. The implementation of the task package uses a set of assembler functions to manipulate the stack pointer. As long as each task stays within its stack limit, each task safely maintains its own thread of control. Virtual concurrency of task execution is achieved by non-preemptive scheduling. Tasks release control by either explicitly suspending operation (e.g., to wait for some event) or implicitly by accessing objects that may block (e.g., reading from an empty queue).

The task package maintains the notion of virtual time. The scheduler orders executable tasks by time. Time advances when there are no active tasks scheduled for the current time. Tasks can read the virtual clock, can suspend execution for certain durations, or may wait for a specific time. For simulations, the virtual time models the time experienced by the simulated system.

The original code of the task package used a rather simple scheduler. The time order priority queue of tasks waiting for execution was maintained as a linked list. Insertions and deletions (un-scheduling of events is supported and quite common) into the event queue involved linear searches. While this is acceptable for small numbers of tasks, the simulations performed for this thesis used up to 5000 tasks. Profiling revealed that more than 90% of the CPU time went into linear searches of the event queue. Subsequently, the event queue data structure was changed to a splay tree [67, 133]. This change reduced the scheduling overhead to about 3% of the total CPU time. Part of the improvement was due to maintaining a pointer to the least recent object.

The notion of queues employed by the task package did not match the intended structure of the simulator. Writing a task that sends and receives data over several queues concurrently was cumbersome and inefficient. It became apparent that some sort of interrupt facility for tasks was needed. Queuing one or more events for one task was desirable. Likewise, the task should be able to deal with all current events once control is granted. While context switching is not very expensive (equivalent to about 9 function calls), it is still significant and using the main event queue would result in a large number of otherwise unnecessary context swaps. This problem was solved by adding an *event-queue* object. Task objects may use private event queues to coordinate internal operations. The prevalent style of inter-task synchronization and communication distributes pointers to other tasks. Hence tasks are able to schedule local events for a remote task. This mechanism also allows tasks to be partitioned into several semi-concurrent parts. Essentially each task can become a mini-simulator of its own.

A.2. CBS

This section is a brief description of the CBS network simulator. After describing the design goals of CBS, the basic control structure and the main data structures are outlined. CBS served as a stepping stone for the construction of NS2 and the experience gained from CBS that contributed to NS2 is summarized at the end of this section.

A.2.1. CBS Objectives and Design Rationale

CBS (cube simulator) was primarily written to experiment with different message systems. The *experimental* nature translates into a program that underwent many modifications. It also means that the final version is moderately complex and largely undocumented.

Experimentation requires good instrumentation; and because the object of the research is the message system, all parts of the simulator relating to the transport of messages through a communication network were heavily instrumented. Instrumentation is accomplished by adding code to the functions of interest, usually accessing global counters, histograms, etc.

Given that k-ary hypercubes cover a fairly wide and diverse family of topologies, this structure was hardcoded into CBS. This slightly simplifies routing.

The difference of asynchronous vs. synchronous operation was one area of interest that was covered by CBS. To simulate asynchronous behavior, CBS has facilities to vary the speed of each node so that the speed distribution is gaussian with a selectable variance. This feature models systems like JPL's Mark-III hypercube that run each node with a separate crystal oscillator. The operating frequencies vary slightly from node to node but are very stable over time. CBS can also model dynamic timing variations (jitter). Both features operate by changing the clock period for each node, either statically (constant over the entire simulation) or dynamically (each cycle may differ).

While there was plenty of CPU time available for this research (many SUN-workstations are quite idle over night at CMU), single runs of several days occurred. Given that simulations always end up being CPU-intensive, coding efficiency was emphasized throughout. Given the choice between following good software practice (such as proper encapsulation, information hiding, side-effect-free functions, etc.) and fast short cuts (such as global variables, tricky pointer arithmetic, etc.), the latter always won. Besides producing many subtle bugs, this approach resulted in a very fast, low-overhead simulator.

A.2.2. The CBS Structure

CBS uses the modified C++ task package. The simulator is controlled by arguments supplied on invocation so that it can be used from shell scripts. The main program parses the arguments and dynamically constructs the network. The arity and the number of dimensions are supplied from the command line.

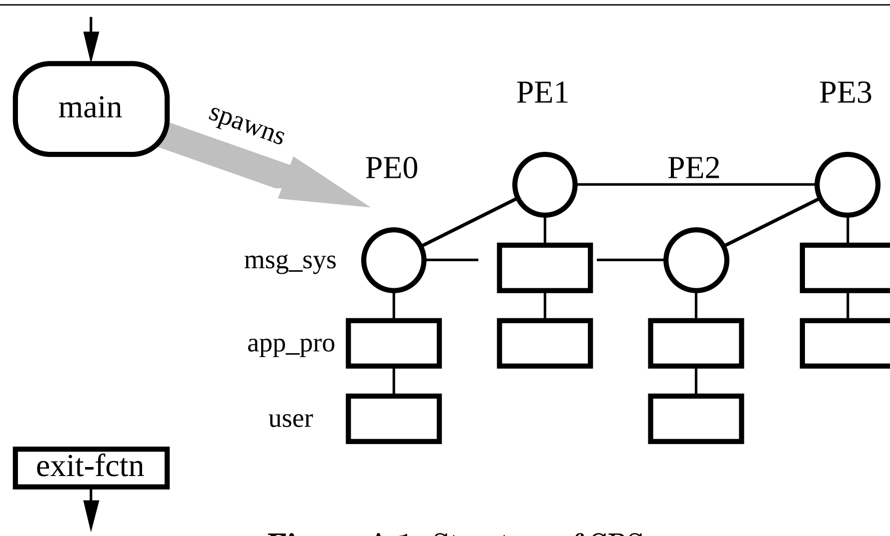


Figure A-1: Structure of CBS

The basic structure of CBS is given in Figure A-1. For each node of the network, three tasks are created:

The *message-system*:
All functions related to the routing of packets through the network are handled by the message system task. It also maintains a set of transient packet buffers if adaptive routing is used. In case of virtual cut-through routing, virtual channels are maintained to ensure deadlock freedom. All communication channels connecting different nodes of the network originate from and terminate in message system tasks.

The *application process*:
The application process is the interface between a program running on a node and the message system. It is responsible for disassembling messages into packets and vice versa. Other functions include the queuing of unsolicited messages, enforcing message order and hiding the low level details from the user process.

The *user program*: The user program drives the simulator. It is provided with an environment that resembles the one found on a private memory multiprocessor. Technically, it is just a C++ task with it's own threat of control. However, a number of functions and macros are provided so that this code may pretend to run on a parallel machine.

The message-system tasks are interconnected with channel-objects. Two channels are also used to connect it to the application process.

Channel objects are not tasks and have no individual control flow. Hence they live outside of the task package. Channels replace the *queue*-object of the task package. The major differences are that the transmission through a channel takes time and that channels use the local event-queue mechanism. Because of this, it is easy to have multiple active channels operating *concurrently*, all attached to one task.

On creation, the channel object receives pointers to the sources and destination tasks it is attached to. To send a packet across a channel, the originating task executes the transmit-function of the channel object. The packet object is stored in the channel structure and an event for the destination task is scheduled. That event will become visible to the destination task after the packet transmission time expires. This transfer mechanism does not cause any context changes as the originating task stays in control. Thus sending a packet is no more expensive than executing a function call. Channels use the lazy evaluation principal. If channels are not referred to, their state cannot change because they are not task objects. However, an ongoing transmission is supposed to complete after a defined time. This is done by re-evaluating the channel status upon request. Thus the channel-busy test can have the side-effect of changing the channel status from busy to idle. The event-queue mechanism practically eliminates the need for status tests; hence inefficient polling is avoided.

The adaptive and the virtual-cut-through routing policies are sufficiently different that two different versions of the message-system tasks are provided by CBS. A switch on the command line decides which one is used to construct the network.

A.2.3. Running Programs in CBS

CBS simulates a private memory multiprocessor that can run C or C++ programs in each node. These user programs will become the core of the user-program task. It is therefore necessary to recompile this task whenever the user program is changed. Changing the number of processors, the network structure or the routing policy does not require recompilations.

CBS provides the user-program with a generic set of send and receive functions that are functionally equivalent to those found on real ensemble machines such as the Ncube/10 or JPL's Mark-III running the Mercury operating system. Typed, non-blocking sends to any node throughout the network are supported. Typed receive operations may either block or return a success/fail status. Unsolicited messages are queued for future consumption.

Since the user programs become part of a predefined task, a few coding conventions are in effect. Memory local to each node must come from the stack (standard automatic storage), the dynamic memory allocator (available via the *new* function of C++), or it must be part of the user-class declaration. An upper bound of the stack use must be declared and observed because the task package maintains multiple stacks via static allocation. Actual stack usage is reported, but stack overruns result in bizarre errors. New code should use generous stack limits. Runs with many tasks will require tighter limits as the allocated stack space increases the total memory requirement for CBS.

Dynamically allocated memory is quite similar to typical *malloc* use in conventional C-programs. Messages transport pointers to data, not the actual data itself. This avoids unnecessary copying. However, this performance short-cut is quite unlike programming on a real private memory machine, where local pointers are invalid in remote nodes.

Data structures declared in the user-program class serve as the equivalent to storage that is global within one node. However, reference to this data from functions other than the actual user-task requires a pointer to the user-task.

Execution of local code takes no time. Essentially, the node CPU is infinitely fast. The user-program must use explicit calls to the *delay()* function to consume time.

The "infinitely" fast node CPU and the availability of real global memory eases instrumentation. User programs may simply add code to measure the time for certain operations and record the result in global structures. No locking is necessary due to the non-preemptive scheduling. Only send, receive and delay function calls can result in context changes.

A.2.4. Lessons Learned

Using the C++ language and the modified task-package saved a great deal of work. C++ is low level enough to allow fast and efficient code. There is plenty of rope were needed, and surely enough to hang oneself. The class construct was indispensable for CBS. The only disadvantage was the compilation process, which uses C as an intermediate language. Since the symbolic debugger refers to the intermediate C-code identifiers, familiarity with the translation process is required. Debugging is further complicated by the multi-threaded control flow. The multitude of stacks destroys context information so that the debugger cannot provide the correct call stack information.

The static stack allocation without a secure way to check for overruns is a frequent source of trouble. Users must be aware that print statements use a considerable amount of stack space (about 1600 words on SUN-3 implementations). It was essential to verify the actual stack usage. Any successful CBS run should report at least a few words of unused stack space for each task. Consuming all stack space results in a warning and the simulation results cannot be trusted. However, the absence of a stack warning is no guarantee of correct stack use. If the user task allocates a large automatic data structure that exceeds the declared stack size and if that data structure is only partially used, the stack verification procedure may fail to detect the problem. Stacks are verified by initializing the stack with a special bit pattern. At termination, the number of words at the end of the allocated stack area that still have the initialization pattern are counted. If this end falls into a gap of the user data structure, no error is detected. Hence it is necessary to check the possible size of the automatic data structure when the stack size declaration is made.

The overall structure of CBS was adequate and proved flexible enough for fast experimentation. The encapsulation of the user code as a separate task was successful and collapsing the message system with the application process would have increased the complexity too much. Therefore, using three tasks per processing node was the optimal structure.

The local event queue mechanism proved to be robust, efficient and flexible. However, the structures of the channel and fifo objects were too tricky. Both objects avoided overhead by being driven by the environment. This code turned out to be quite fragile. A cleaner structure is needed.

The hardwired nature of both the network topology and the instrumentation resulted in fast execution; however, instrumentation turned out quite ad-hoc. A better abstraction for the instrumentation part is clearly called for.

The need to supply execution times explicitly in the user code is awkward. This procedure is error prone and tedious.

A.3. NS2

Based on the experience with CBS, NS2 has the same overall structure. Again, the C++ task package provides the framework. Like CBS, NS2 uses three tasks per node, the *message system*, the *network interface* and the *user* class. Unlike CBS, NS2 was structured to be of more general use; hence the code was organized into 18 modules that are integrated by a skeleton system. For performance reasons, there are still a few short-cuts between modules, but these are better documented and caused no problems.

Since the results from asynchronous operations of CBS caused few surprises, the ability to simulate asynchronous behavior at the hardware level was dropped. Likewise, since wormhole routing was not beneficial, only adaptive routing of fixed-length, minimal-sized packets is supported. Due to the modular structure of NS2, changing the router is much less of a problem than in CBS.

As in CBS, networks are created at run time. However, the topology is no longer hardwired into the system. The simulator skeleton calls a network generator module that supplies the bare topology. Different generators can be used by an appropriate switch on the command line. It is possible to use a topology generator that simply reads in the connectivity information from a file. NS2 verifies the topology and builds the routing tables. Routing is table driven.

A.3.1. The NS2 User Code Interface

CBS simply provided the usual compliment of *send* and *receive* type primitives. However, the objective of NS2 is to provide the appearance of virtual shared memory. Hence access to certain memory locations by the user program causes implicit communication to occur.

Inspired by exercise 8 of Chapter 6 of *The C++ Programming Language* [141], the class *INT* was defined to represent shared virtual memory. Besides the switch from *int* to *INT* variables, no change of conventional code should be required. The alternative of requiring some explicit function call for every access to memory was simply too appalling to contemplate.

NS2 succeeded in creating a syntactic and semantic equivalent to *int*-variables. However, this exercise is really beyond the capability of the C++ language. Class objects differ from the built-in types in many subtle ways and operator overloading is only a partial answer. Aggravating the problem is the need for memory allocation under the control of the NS2 simulator. Memory allocation is not operational before the network structure is in place and all nodes are allocated. Explicit assignment to the *this* pointer was required, but that precludes static allocation[53]. The cost of memory allocation outlaws the destruction and construction of INT objects during assignments. Overloading assignments by reference - suggested by *the book* - is only a partial solution. The situation becomes nearly hopeless when combinations of INT arrays, composite structures, pointers to INT arrays and associated pointer arithmetic are all required to work as their *int* counterpart. Solving this problem correctly would require modifying the C++ compiler or building a preprocessor that can completely parse a C++ program.

The *incorrect* solution used by NS2 is a set of filters that partially parse the declarations and the user code to translate statements dealing with *INT*'s into legal C++. Most interventions deal with the allocation of INT's, arrays of INT's and structures that contain INT's. The process is quite complex and likely to be confused by overly complex data structures.

The inability to substitute built-in data types with user-defined objects in a clean and efficient manner is a deficiency of C++. However, this defect is largely due to the desire to stay compatible with C whenever possible. The multi-stage compilation processes that uses C as the intermediate language were instrumental in overcoming this problem. A monolithic compiler such as GNU's G++ would have require substantial compiler modification.

NS2's user code interface is stretching the limit of the C++ language. While the desired objective is met, the solution is complex and fragile. This is probably the weakest part of NS2.

NS2 provides detailed control on how memory is allocated. Memory can be allocated on one node, all nodes and any subset of nodes.

A.3.2. Network Interface Structure

The network interface task (*NI*) of NS2 has a lot more functionality than the application process task of CBS. Initially, the NI used the same structure as the message system task and the old application process, that is basically a switch-statement that decodes the event returned by the local event queue. Each event, for example the reception of messages, causes certain actions, for example the insertion of the data into a queue. Given that there were few event types and that little context information was required, a simple switch statement was quite effective and of manageable complexity.

However, the complexity of the NI is way too high, resulting in incomprehensible spaghetti

[53]Static memory allocation is achieved by means outside the C++ language

```
while (1) {
    waitlist(ev_queue, 0);                  // wait for something to happen

    ev_ty = ev_queue->E_deq(&info);         // dequeue event
    if (ev_ty < 0 || ev_ty > PAGE_SIG)
        ERR "NI-mngr%d: undefined event encountered ev-type=%d", id, ev_ty RRE;

    res_vect |= 1 << ev_ty;                 // update resource vector
    if (ev_ty == MEM_RDY || ev_ty == PAGE_SIG)
        res_vect |= info;

    DBP(("NI%d: event %s inf=%d time=%4d res_vect=%08x\n", id, E_MEMO[ev_ty], info, clock, res_vect));

    for (re_scan = 1; re_scan;) {           // execute sub-FSMs
        re_scan = 0;
        while ((res_vect & WU_req) || !WU_req) WU_state = WU_fsm(WU_state, &WU_req);
        while ((res_vect & NQ_req) || !NQ_req) NQ_state = NQ_fsm(NQ_state, &NQ_req);
        while ((res_vect & NS_req) || !NS_req) NS_state = NS_fsm(NS_state, &NS_req);
        while ((res_vect & US_req) || !US_req) US_state = US_fsm(US_state, &US_req);
        while ((res_vect & RV_req) || !RV_req) RV_state = RV_fsm(RV_state, &RV_req);
        while ((res_vect & PC_req) || !PC_req) PC_state = PC_fsm(PC_state, &PC_req);
    }
}
```

Figure A-2: Network Interface Core Loop

code. Therefore, the NI was restructured into a generic core and a number of specialized functions (Figure A-2). The NI-core loop waits for events to happen. Events are the expiration of timers, requests by the user task and the completion of I/O operations by the message system (transmission and reception of messages). The core loop does not act on these events, rather it updates a set of resource flags. 32 resource bits that reflect the status of the input and output signals of the network interface are provided. These bits are very much like the logic value of a wire in a digital circuit. Besides bits related to events, a number of these state bits are used internally by the NI.

After the de-queuing of an event and after updating the resource bits, the core loop proceeds to execute several finite state machines (FSM). These FSM's are defined by their transition function, the same approach that is used in C2FSM/AFC [102]. The input signals to these FSM's are the resource bits maintained by the core loop. FSM's declare which inputs are needed for the next state transition by setting a corresponding bit in a request vector. The core loop calls the transition function until the request and resource vectors don't match. The NI uses six FSM's for various sub-functions such as interpreting incoming traffic, maintaining the memory of the node, keeping track of user-task operations and performing some node operating functions such as allocating local memory, copying pages across the network, etc.

The FSM's communicate with each other by setting or clearing bits in the resource vector. Hence the NI is best described as a set of concurrent FSM's connected by various wires. The NI core is simply simulating this set of concurrent FSM's by executing its transition functions until no more state transition is possible without an external event. At this point control is returned to the simulation scheduler that will revive the NI when the next event becomes available.

The interpretative nature of this NI structure is less efficient than a monster switch statement, but it did wonders for debugging. A framework is provided that can emit messages on every state change and for every event. This sequence of state changes was instrumental to pinpoint problems.

It is no accident that the software complexity was handled by *reducing* the abstraction level: The NI is essentially a little logic simulator. The individual FSM's are quite similar to the C2FSM description of actual circuitry.

A.3.3. Instrumentation

Instrumenting the simulation was one of the deficiencies of CBS. NS2 addressed this problem by defining the two classes *meter* and *multimeter* that are generic instrumentation devices. The basic idea is to separate the phenomenon under observation from the tool used to measure the effect. Once a probe is deployed in the simulator, different observation devices can be attached to this data source.

Due to the uniform structure of the instrumentation in NS2, it became possible to selectively collect data under certain conditions and in different ways. For example, a common problem with the data acquisition in CBS was the appropriate handling of non-steady state behavior. The start up transient and the termination phase introduced substantial errors because CBS simply averaged over the entire run. Because data collection was distributed and non-uniform, there was no simple place to control data collection. In NS2, the constructor of the instrumentation classes maintains a list of all active meters which is used to prepare result summaries, dump data, record traces, etc.

The meter class defines a common interface for all instrumentation. Subclasses are derived from the meter-class to provide specific measures, such as averages, histograms, etc. Derived classes may have extra storage to record derived information, for example to compute the simulation error [82].

```
meter (
    char           *name,     // name of this meter
    unsigned char   opt,      // option vector
    int             sf  = 1,  // scale factor
    unsigned short  aff = 0,  // affiliation (opt.)
    int            *var = 0,  // variable to be monitored (opt.)
    multimeter     *mmp = 0   // multimeter pointer (opt.)
);
```

Figure A-3: The *Meter* Declaration

The generic meter constructor is given in Figure A-3. Meters can be constructed dynamically. However, in most cases they are simply defined before the simulation is started. The *name* given at construction time will be used to report the measured entity. The *option vector* is constructed by OR-ing predefined constants. Available options include:

Discrete: The observed quantity is defined only for discrete points in time. Examples of discrete observables are the reception of a message, cache misses, initiations of synchronization operations. Averaging a discrete meter over time is equivalent to computing the rate or frequency of events. If the *discrete* option is not specified, the observable will be treated as a continuously defined variable. For example the number of items waiting in a queue is continuously defined.

Report: The meter will show up in the simulation report. This option can be used to unclutter the simulation report by suppressing some measures. Normally, un-reported meters are grouped into a multimeter and contribute to a measure that is averaged over a larger group of objects, such as all nodes in the simulated system.

Node / Channel: Meters can be affiliated with the primary simulation objects, the processing nodes and the communication channels.

Trace: The *Trace* option enables the inclusion of the metered data in a simulation trace. Selective inclusion is frequently needed because traces tend to be large.

The scale factor is mainly used for entities that depend on the simulation size. This parameter is simply passed to subsequent processing stages that can use it to normalize results.

Normally, instrumentation is done explicitly. Whenever a metered entity is changed during the course of the simulation, a *set* or *change* function call is needed. Hence instrumentation still requires insertion of instrumentation code. However, meters can be used to monitor the state of a variable without explicit instrumentation. If a pointer to an integer variable is specified in the meter declaration, a monitor task is created that is executed whenever the simulated time advances. The monitor task scans a list of variables under investigation and will update the appropriate meter whenever a change occurs. This instrumentation method is convenient for a variable that is changed from many places; however, it incurs an extra task context change whenever the scheduler advances time plus the state change compares for each monitored object. This can translate into a 10% increase in simulation time.

Multimeters are a collection of meters. For example, the number of cache misses is recorded in one meter for each node of the simulated system. A multimeter is defined that combines all cache miss monitors and reports the average, minimum, maximum, variance, etc. of all cache miss meters.

A.3.4. Execution Delay Computation

CBS only provided an explicit *delay* function. Writing application code required explicit estimation of how much time is used by the various parts. NS2 still provides the explicit delay function, but the primary means of CPU time estimation of the client application was automated. The user code is compiled only to the assembly code level. At this point, a filter is inserted into the compilation process that parses the code generated by the compiler. For each instruction, the number of required clock cycles are computed based on tables for the instruction and the used addressing modes. The total number of clock cycles for straight code sequences is accumulated in the *A2* register[54] before each branch instruction and before all labels and function calls. It is necessary to reconstruct the entire execution flow graph to properly deal with some of the code motion caused by the use of the optimizer of the C-compiler.

The delay estimator detects illegal use of the A2 register that may arise from too many register declarations of pointer variables in the user code. Assembly directives are recognized to zero the impact of code instrumentation. The code that is generated for each use of a *meter* is considered to execute in 0 clock cycles; hence instrumentation is non-intrusive.

Once a memory reference to the simulated memory is encountered, the accumulated number of clock cycles is translated into elapsed run time and an appropriate delay is executed.

A.3.5. Trace Facility

A trace facility was added to NS2 that was partially inspired by the *SCHEDULE* package [41]. When tracing is enabled, NS2 writes a write trace record to a file whenever certain events occur. Events that can cause trace records are changes of a meter, the encounter of an event declaration and access to the virtual shared memory.

The header of the trace file contains supplementary information associated with a simulation run. This information includes a plain text description of the user code, optional comments, time and date of execution, name of the system executing the simulation, the network topology, the simulation parameters, etc. Besides these documentary entries, the header also contains tables describing all meters and multimeters that participate in the trace, geometry information to display the network, and a table describing each location in the user program that can access virtual shared memory. To ensure that the source code line numbers are valid, a check sum of the source code is included.

---

[54] Due to the manipulation of the assembly code, NS2 is specific to the Motorola 68000 family, which is used in many SUN-workstations.

Meters that have the trace option enabled will write a record containing a time stamp, the meter ID and the new value to the trace file whenever the metered variable is changed.

User-defined event records carry a brief mnemonic and the time when they are encountered during the simulation.

Virtual shared memory access can be traced. Memory access records include the time of access, the processor number, the index into the table of memory references and the consumed CPU time since the last memory reference. The time consumed by the preceding memory reference is computed by subtracting the used CPU time from the elapsed time since the last reference. The trace analyzer finds the type of memory access and the source code line number from the table in the header.

While prone to produce large trace files, recording memory references are a useful debugging tool and provide multiprocessor address traces for cache performance studies.

Because of the non-intrusive instrumentation, the execution of NS2 is deterministic and does not change by turning trace components on or off. Therefore, it is possible to rerun a simulation with a different set of enabled trace components if the first run did not contain the desired information.

A.3.6. Lessons Learned

NS2 corrected most of the problems encountered by CBS. However, these two simulators are not directly comparable. CBS's mission was to study the message system while NS2 was focused on the communication architecture. Hence NS2 explored new territory and promptly encountered a few problems.

The main problem was related to C++'s inability to truly substitute a data type of the C++ language with a user-defined object. The current solution in NS2 is workable but it is not satisfactory.

Another problem relates to the implementation of addresses within NS2. For performance reasons, addresses were restricted to 32 bits so that they could be held in a register. The address must serve two functions, to uniquely address virtual shared memory throughout the entire simulated system and to address the memory on a local node. Since multiple copies of a page of memory may exist, these two address components are not identical. The net result is a limit on the amount of memory that can be resident in each node. 128Kbytes appeared to be plenty when the structure of NS2 was conceived because the combined amount of all simulated nodes must fit into the available physical memory[55]. This limitation turned out to be one of the first complaints of NS2 users.

The use of interpreted finite state machines to structure the network interface was successful. Many modifications were made in this area and debugging proved to be relatively easy.

Instrumentation was entirely satisfactory. The ability to monitor each virtual shared memory reference was used in NS2 to print an annotated source listing. Each line that contains a memory reference is preceded by the total number of time it was executed, the average access time and the accumulated CPU time since the last access. This level of detailed profiling of parallel programs is also helpful for debugging and fine-tuning of parallel programs.

A.4. TA2

TA2 is the trace analyzer for the NS2 simulator. TA2 is an interactive Suntool[56] application. It was separated from NS2 because simulations are too slow for interactive use and because of the memory demand for the graphic subsystem.

Figure A-4: The TA2 Control Panel

Once started, TA2 displays a control panel on the screen (Figure A-4). Most TA2 functions are activated by moving the cursor near a panel object and pressing an appropriate mouse button. In general, standard Suntool conventions are used.

The load function reads a trace file into memory. Each load resets all internal data

[55]Experience with CBS showed that thrashing is fatal. Even a 10% excess over the available physical memory degraded the SUN's to less than 1% of the normal performance.

[56]Suntools is a window manager for SUN workstations.

structures and multiple traces can be processed without restarting TA2. Reading a trace file can cause a noticeable delay because trace files tend to be large. Currently, the entire file is read into memory. The header of the trace file is interpreted and the vital statistics are displayed on the control panel.

Typical trace files can have several million entries. Therefore, it is of little use to display individual trace records and the main control structure of TA2 is an uncommitted driver that scans through the trace data. The trace data is displayed through various presentation functions that can be attached to the driver.

A.4.1. The TA2 Driver

The TA2 driver maintains a pointer to the trace data that corresponds to the *current* simulation time. The trace record being pointed at is the next event that happened during the simulation. Initially, the trace pointer is positioned at the start of the trace data. This corresponds to simulated time 0. Advancing the pointer will cause the driver to decode the trace record and present the data to an optional presentation function. Presentation functions are attached dynamically by user commands to TA2.

The simulated time may advance as a side-effect of advancing the trace pointer. There is no one-to-one relationship between simulated time and the trace pointer position because trace records correspond to simulation events and not to the passage of time. Each record contains the amount of simulated time that passed since the last record. Multiple events can happen at the same time because NS2 simulates a concurrent system. In this case, the incremental time stamp of the trace record is 0.

The current position in the trace file and the corresponding simulated time are displayed in the control panel in the event and time sliders, a horizontal bar that indicates the relative position.

The trace pointer can be moved interactively. The *Begin* and *End* objects of the control panel will move the trace pointer to either end of the trace. The event and time sliders are panel objects that can be manipulated with the mouse. Dragging the time-slider to a certain position will cause the TA2 driver to move the trace pointer to that position and update the event slider accordingly. Both sliders can be used interchangeably. The *Forward* and *Backward* objects cause the driver to continuously move the trace pointer in the desired direction. The speed of this animation mode is controlled by a separate slider at the bottom of the control panel. The *Stop* panel object will stop the animation and *Step* can be used to move the trace pointer one step at a time.

Except for the *begin* and *end* functions, the TA2 driver moves the trace pointer by incrementally stepping through all trace records and calling the appropriate presentation functions.

There are two principal positioning modes controlled by the *buttons* in front of the event and time sliders. In event-mode, the driver is stepping through the trace file one record at a time while in time mode a step includes all records that happened at the same time.

Besides the trace point, two auxiliary pointers that designate a symmetric window around the current trace pointer are maintained. This window is of a selectable width that is specified in units of simulation time. One unit of simulation time is assumed to be one nsec. This window is typically used by presentation functions to average some measure.

A.4.2. TA2 Presentation Functions

Presentation functions are attached to the TA2 driver to visualize the trace data. There are three types of data records in a trace file: meter change records, event records and profile records.

Meter records simply contain the new value for a given meter. A presentation function can be attached to a specific meter. Multimeters are a collection of meters and are defined in the header of the trace file. There are no specific records for multimeters, rather the TA2 driver will call the multimeter presentation function whenever one meter of that group changes.

Event records serve as markers that are specified by the author of a user program. Whenever the execution of that program reaches the event statement, the time, node-number and event-mnemonic are added to the trace file.

Profile records can be generated whenever a program executes a reference to the virtual shared memory. The time, node-number and consumed CPU time of that node since the last reference are recorded along with a static identification that relates the reference to the corresponding statement in the source code. In addition, the static identification indicates the type of reference operation. Delay statements can also cause profile records.

Presentation functions are activated by selecting a panel item with the mouse. Pop-up menus will query for parameters, for example what data source to use. All presentation functions will open a separate window in addition to the control panel on activation. Presentation functions are closed by destroying their window. They are also closed when a new trace file is loaded.

The plot function can be used to view the value of a meter or multimeter as a function of time (Figure A-5). Discrete meters will result in a bar graph where vertical lines represent

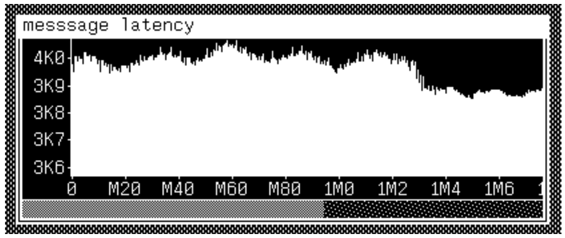


Figure A-5: The *Plot* Presentation Function

the value of that meter. Continuous meters will result in a continuous plot. In case of non-zero windows, the meter value will be the average over that period for each point in the display. Discrete meters will add the values within the window and divide by the window width. For example, if the meter is set to one for each memory reference, the windowed value will become the reference rate over the window period. Continuous meters will cause proper integration over the window period.

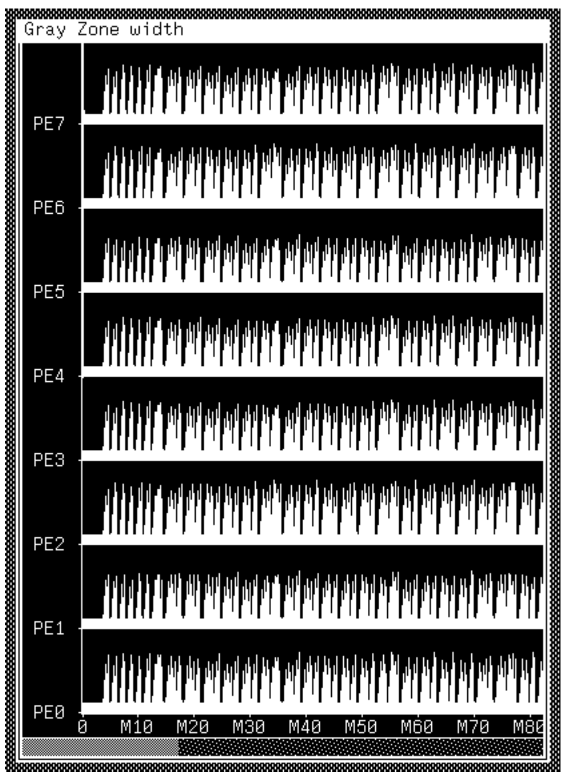


Figure A-6: The *Multi-Plot* Presentation Function

The multi-plot presentation function is similar to the plot function, but applies only to multimeters. Each associated meter is plotted separately and is identified by the node or channel number it is affiliated with (Figure A-6). The multiplot function is used to give an overview of one particular measure throughout the system. Load imbalances or irregular behavior of one node are easily spotted in this presentation.

The histogram function can either be cumulative or can operate over the window width. It can be applied to meters and multimeters. Histograms are typically used on discrete meters. If applied to a continuous meter, they become the density function of that meter.

The event window operates only on the event records. Each event is described by a horizontal line of text. The current event record is positioned at the center. The text scrolls up or down according to the movement of the trace pointer.

The profile function opens a text window with the source code. Pointers indicate the current location of execution for each node. The accumulated CPU and access times are shown as cumulative bar-graphs alongside the code. Cumulation is restarted whenever the direction of movement of the trace pointer is changed.

The display function lists the current values for a selected set of meters and multimeters. The current event records are displayed at the top of the window.

The *Miscellaneous* panel object pops up a menu of auxiliary functions and commands:

- The *hardcopy* function can produce a spoof[57] files for currently opened windows of presentation functions.
- The *differentiate* and *integrate* functions can perform numeric differentiation and integration of meters and multimeters vs. time. This can be used to convert between cumulative and rate meters.
- The *network* function opens a graphic representation of the network being simulated. The geometric information for this plot is taken from the trace file header. NS2 has optional geometry generators for each network topology that can generate reasonable layouts for some networks. The nodes and channels can represent the value of a multimeter. The network nodes become tiny pie charts for meters that are affiliated with nodes, for example synchronization wait time. The rectangular channels symbol is filled according to the value of a meter affiliated with a channel, for example utilization.
- The *quit* function terminates TA2.

Presentation functions are relatively easy to write and attach to the skeleton of TA2.

[57]*Spoof* is a graphics interpreter and plotting program. It can produce line drawings and/or graphical representations of data in various forms. Many figures in this thesis were generated with spoof.

Appendix B
Generalized Petri-Nets as a Design Tool

Petri nets were initially conceived as a tool to analyze the correctness of asynchronous concurrent systems. They were used to verify circuit design such as bus arbiters, I/O interfaces, etc. At a higher level, Petri nets were used to certify protocols and software systems. Petri nets used in this capability have no notion of time. They can prove the absence of deadlocks but they don't reveal any performance information.

Since Petri nets are a relatively convenient and intuitive method for expressing concurrent systems, generalizations have been proposed that add timing information [96, 91]. Timed Petri nets become a more realistic model and analysis can yield performance estimates. Transitions of a stochastically timed Petri net (SPN) require a random amount of time to fire a transition. If the transition times are exponentially or geometrically distributed, the expressive power of SPN is equivalent to that of homogeneous Markov chains. States of the Markov chain correspond to valid markings of the SPN; hence the Markov chain tends to have a large number of states and is a much less convenient modeling tool.

Since the delays of real systems are not necessarily exponentially distributed, SPN's were generalized to include transitions with fixed delays (GSPN). GSPN's are becoming popular tools to estimate the performance of concurrent systems, both at the hardware and software level [116].

Molloy wrote an interactive tool to enter, debug and analyze GSPN's [95]. This system inspired Chiloa's *GreatSPN* [26] that was used in this thesis.

GreatSPN is centered around a specialized, interactive graphic editor for GSPN's. Once a net is entered, it can be executed manually by using the mouse to fire enabled transitions. Besides this debugging and documentation support, a number of structural and dynamic analysis tools are available that can be executed from within the graphic editor. Results are back-annotated into the graphical representation of the net.

Complex nets frequently exceed the capability of direct numerical analysis. Therefore, a Monte Carlo simulator is part of the GreatSPN package. Simulation can deal with all nets but result in less precise results.

Unfortunately, the GreatSPN simulator is rather slow. Internal data structures are inefficient and statically allocated, severely degrading its utility. To overcome this limitation, an equivalent simulator (GS) was written with emphasis on execution speed.

Unlike GreatSPN, GS compiles the GSPN into a C-program, thus eliminating many address computations. Using lazy evaluation methods, carefully allocated registers and other C performance tricks, a 500 to 1000-fold speedup over the original Pascal code was achieved.

Unlike the GreatSPN package, GS defines a human readable input language that is compatible with the C preprocessor. Using CPP macros, much larger nets become feasible. There simply is no way to draw nets with 1000 nodes.

Appendix C
On the Generation of Random Numbers

The synthetic load generation, the asynchronous modeling, and the Monte-Carlo simulator for Generalized Stochastic Petri Nets all required a source of random numbers. *rand()* from the standard UNIX-library is known for its unacceptable statistical properties and was never used throughout this thesis. Cohen's *random()* - as supplied with more recent versions of Berkeley's UNIX - was found to have good statistical properties, but it was not available in source form. Since *random()* was missing on several systems used for this research, an independent, portable pseudo random generator (*xrand()*) was written, verified and used throughout this thesis. News of Park's standard random number generator [109] was a bit too late for this work and it may have been too computationally expensive.

xrand() is composed of two parts: a linear congruential generator (LCA) that is permuted with a Fibonacci additive congruential generator. The LCA was chosen because of the readily available theory [79, 97, 1] that governs the parameter selection to ensure good uniformity. Verification by means of Chi-squared tests on bits, bytes, words and other sub-range histograms of more than $7 \cdot 10^8$ numbers sustains the uniformity assumption with a probability exceeding 0.999. LCA's do not score well on independence tests, hence the need for the second component to *xrand()*.

xrand() uses Knuth's algorithm *M* [79] to permute the numbers produced by the LCA. The algorithm was extended to permute the 4 bytes within a word because of an asymmetric uniformity test results: without the additional permutation step, the uniformity score of the low-order byte was found to be better than that of the high-order byte. The additive congruential generator is based on the polynomial $x_n = x_{n-55} - x_{n-24}$.

Independence tests on *xrand()* included *gap*, *run-up*, spectral and auto-correlation tests. The independence assumption holds with a probability of better than 0.97. Scores for individual tests were compared to results obtained with *random()* and *xrand()* produced better results in most tests. *xrand()* requires about 19.4 µsec per call on a SUN-3/60, which is about 44% slower than *random()*.

Appendix D
Characterizing Telephone-Cable

The cable used for the inter-cluster channel experiments is a plain 25-pair telephone cable similar to Belden's type 1232A. This cable was selected because it is the cheapest available medium; hence it provides a good worst case test. There are plenty of higher quality grades available that have larger gauge conductors, insulation with better dielectric properties (like Teflon) and various degrees of shielding.

Telephone cables are well characterized for their typical application. Data sheets include loss, impedance and crosstalk data for the DC to 100Khz range and over long distances. Occasionally, attenuation is specified for 1 MHz (Belden claims 0.21 db/100m at 1 MHz for the 1232A). Unfortunately, the inter-cluster channel requires data for higher frequencies at shorter distances. Hence actual measurements were necessary.

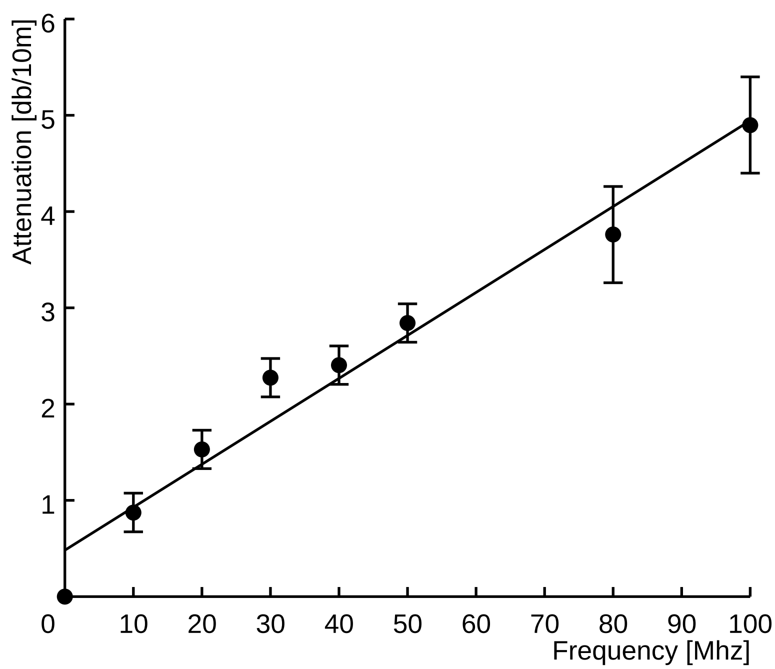


Figure D-1: Telephone Cable Loss

Figure D-1 shows the loss of one pair that is driven as a transmission line. The impedance was measured to be 91 ohms. At 12.5 MHz, the highest fundamental frequency of the intercluster channel, an attenuation of roughly 1 db/10 meter is found. The wire is driven with a 5Vpp signal and reliable reception requires at least 100mVpp. Therefore, the maximum acceptable loss is 33db or about 330 m, well above the design goal of 200m operation.

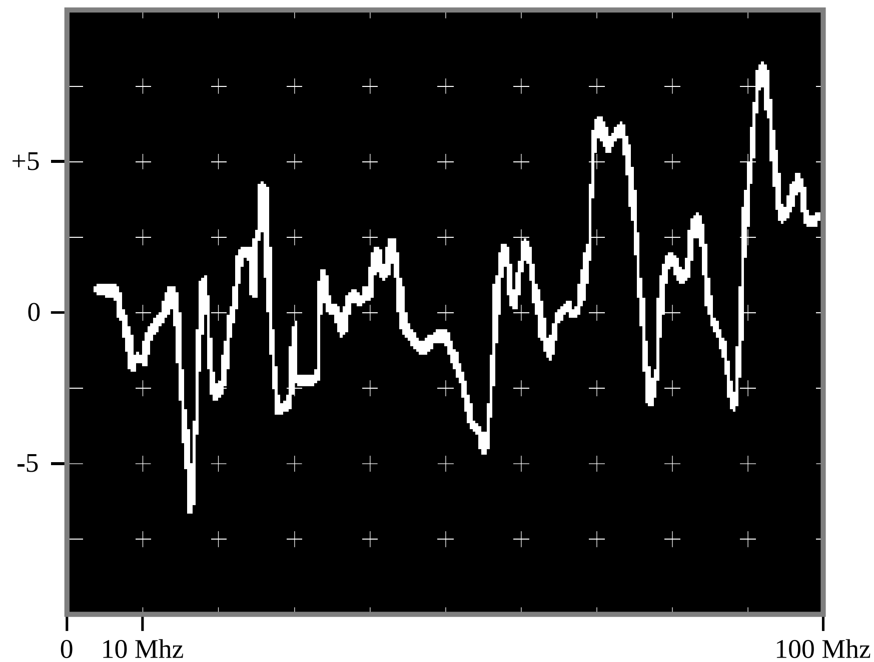


Figure D-2: Telephone Cable Dispersion

The second critical parameter is the dispersion, usually expressed as the change of the group delay ($=\frac{\partial\, phase}{\partial\, frequency}$) vs. frequency. Dispersion causes signal distortion because signals of different frequencies propagate at different speeds. Figure D-2 is a scanned screen picture from a network analyzer. A 25m phone cable is tested by measuring the phase difference between it and a RG-58 coaxial cable of equal average delay. The group delay over the 0-15 MHz range varies by about 1.1ns for 10m. Hence a 200m cable would have a group delay variation of about 22ns or almost 50% of one bit-time, which is a bit on the high side. However, part of the distortion is due to the RG-58 reference, which was assembled from several laboratory cables. Reflections from the connectors and barrels contributed to some of the variation seen in Figure D-2, which was taken with the maximum available phase sensitivity. Eye-patterns from random signals suggest that the group delay for 200m is lower, perhaps by a factor of two.

Tests showed that crosstalk is greatly reduced by driving the wire pair in a truly balanced way by means of transformer coupling. Crosstalk was measured indirectly during random pattern transmission tests by driving adjacent wires with fast square waves. Again, interference from asymmetrically driven wires is much higher than that from balanced and terminated wires.

References

[1] M. Abramowitz, I.A. Stegun.
Handbook of Mthaematical Functions.
Dover Publications, Inc., 1972.

[2] D. W. Anderson, F. J. Sparacio, R. M. Tomasulo.
The IBM System/360 Model 91: Machine Philosophy and Instruction Handling.
IBM Journal of Research and Development 11(1):8-24, January 1967.

[3] T.E. Anderson.
The Performance Implications of Spin-Waiting Alternatives for Shared-Memory Multiprocessors.
In *Proceedings of the 18th International Conference on Parallel Processing*. Pennsylvania State University, August, 1989.

[4] A. Agarwal, R. Simoni, J. Hennessy, M. Horowitz.
An Evaluation of Directory Schemes for Cache Coherence.
International Symposium on Computer Architecture , May, 1988.

[5] James Archibald, Jean-Loup Baer.
An Economical Solution to the Cache Coherence Problem.
Proceedings of the 11th International Symposium on Computer Architecture, SIGARCH Newsletter; IEEE 12(3):355-362, June 1984.

[6] James Archibald.
A Cache Coherence Approach for Large Multiprocessor Systems.
In *1988 International Conference on Supercomputing (ICS); ACM Press*, pages 337-345. St. Malo, France, July 1988.

[7] R. Arlauskas.
iPSC/2 System: A Second Generation Hypercube.
In G.Fox (editor), *The Third Conference on Hypercube Concurrent Computers and Applications*, pages 38. JPL/Caltech, ACM Press, January, 1988.

[8] Arvind, R.A. Iannucci.
A Critique of Multiprocessing von Neumann Style.
In *Proceedings of 10th Annual International Symposium on Computer Architecture*. June, 1983.

[9] William C. Athas, Charles L. Seitz.
Multicomputers: Message-Passing Concurrent Computers.
Computer 21(8):9-24, August 1988.

[10] Clive F. Baillie.
Comparing Shared and Distributed Memory Computers.
Parallel Computing 8(1-3):101-110, October 1988.

[11] George H. Barnes, Richard M. Brown, Maso Kato, David J. Kuck, Daniel L. Slotnick, Richard A. Stokes.
The ILLIAC IV Computer.
IEEE Transactions on Computers C-17(8):746-757, August 1968.

[12] K. E. Batcher.
The Massively Parallel Processor (MPP).
Digest of Papers, Compcon85 Thirtieth IEEE Computer society International Conference; IEEE Computer Society :21-24, February 25-28, 1985.

[13] John Beetem, Monty Denneau, Don Weingarten.
The GF11 Supercomputer.
In *11th Ann. Symp. on Computer Architecture*, pages 290-296. ACM and IEEE, June, 1985.

[14] A. Beguelin, D. Vasicek.
Communication Properties of NCUBE Hypercube Interprocessor Network.
In M. Heath (editor), *Proceedings of the Second Conference on Hypercube Multiprocessors*. SIAM, October, 1986.

[15] A. Beguelin, D.J. Vasicek.
Communication between Nodes of a Hypercube.
In M.T. Heath (editor), *Proceedings of the 2nd Conference on Hypercube Multiprocessors*, pages 162-168. Oak Ridge National Laboratory, SIAM, October, 1986.

[16] D.P. Bertsekas, E.M. Gafni, R.G. Gallager.
Second Derivative Algorithms for Minimum Delay Distributed Routing in Networks.
IEEE Transactions on Communications COM-32(8):911-919, August, 1984.

[17] R. Bisiani, A. Nowatzyk, M. Ravishankar.
Coherent Shared Memory on a Message Passing Machine.
Technical Report CMU-CS-88-204, Carnegie Mellon University, December, 1988.

[18] D.L. Black, A. Gupta, W. Weber.
Competitive Management of Distributed Shared Memory.
In *Compcon*. IEEE, Spring, 1988.

[19] R.R. Boorstyn, A. Livne.
A Technique for Adaptive Routing in Networks.
IEEE Transactions on Communications COM-29(4):474-480, April, 1981.

[20] E.D. Brooks.
The Butterfly Barrier.
International Journal of Parallel Programming 15(4):295-307, 1986.

[21] Eugene D. Brooks III.
The Shared Memory Hypercube.
UCRL 92479, Lawrence Livermore National Laboratory, March, 1985.

[22] P. Burns, J. Crinchton, D. Curkendall, B. Eng, C. Goodhart, R. Lee, R. Livingston, J. Peterson, M. Pniel, J. Tuazon, B. Zimmerman.
The JPL/Caltech Mark IIIfp Hypercube.
In G.Fox (editor), *The Third Conference on Hypercube Concurrent Computers and Applications*, pages 872. JPL/Caltech, ACM Press, January, 1988.

[23] David G. Cantor.
Optimal Routing in a Packet-Switched Computer Network.
IEEE Transactions on Computers C-23(10):1062-1068, October 1974.

[24] T.J. Chaney.
A Comprehensive Bibiliography on Synchronizers and Arbiters.
Technical Report Technical Memorandum No. 306, Institute for Biomedical Computing, Computer Systems Laboratory, Washington University, St. Louis, Missouri, March, 1985.

[25] Alan E. Charlesworth.
An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family.
Computer 14(9):18-27, September 1981.

[26] G.Chiola.
GreatSPN User's Manual
Version 1.3 edition, Dipartimento di Informatica, Universita‘ degli Studi di Torino, corso Svizzera 185, 10149 Torino, Italy, 1987.

[27] W. Chou, A.W. Bragg, A.N. Nilsson.
The Need for Adaptive Routing in the Chaotic and Unbalanced Traffic Environment.
IEEE Transactions on Communications COM-29(4):481-490, April, 1981.

[28] E. Chow, H. Maden, J. Peterson, D.C. Grunwald, D.A. Reed.
Hyperswitch Network for the Hypercube Computer.
In *Proceedings of the 14th International Symposium on Computer Architecture*. June, 1988.

[29] R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B. Papworth, P.K. Rodman.
A VLIW Architecture for a Trace Scheduling Compiler.
In *ASPLOS II*, pages 180-192. IEEE, October, 1987.

[30] *The Connection Machine: A More Detailed Look*
Thinking Machine Corp., 1985.

[31] Christoph von Conta.
Torus and other networks as communication networks with up to some one hundred points.
IEEE Transactions on Computers C-32(7):657-666, July 1983.

[32] R.R. Cordell.
A 45-Mbit/s CMOS VLSI Digital Phase Aligner.
IEEE-Journal of Solid State Circuits 23(2):323-328, April, 1988.

[33] S. Cray.
What's This About Gallium Arsenide?
In *Supercomputing'88*. Orlando, FL, November, 1988.

[34] W.J. Dally, C.L. Seitz.
The Torus Routing Chip.
J. Distributed Systems 1(3), 1986.

[35] W.J. Dally, C.L. Seitz.
Deadlock-Free Message Routing in Multiprocessor Interconnection Networks.
IEE Transactions on Computers C-36(5):547, May, 1987.

[36] W.J. Dally.
A VLSI Arcitecture for Concurrent Data Structures.
PhD thesis, California Institute of Technology, March, 1986.

[37] C. Delrome, G. Farhi.
Large Graphs with Given Degree and Diameter.
IEEE Transactions on Computers C-33(9):857-859, September, 1984.

[38] Monty M. Denneau, Peter H. Hochschild, Gideon Shichman.
The Switching Network of the TF-1 Parallel Supercomputer.
Supercomputing 1988 , Winter 1988.

[39] A.M. Despain.
X-Tree: a multiple microcomputer system.
Spring 1980 Compcon; IEEE :324-327, 1980.

[40] D.M. Dias, M. Kumar.
Preventing Congestion in Multistage Networks in the Presence of Hostspots.
In *Proceedings of the 18th International Conference on Parallel Processing*. Pennsylvania State University, August, 1989.

[41] J.J. Dongarra, D.C. Sorensen.
Schedule User Guide
Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Class Avenue, Argonne, IL 60439-4844, 1987.

[42] Michel Dubois, Christoph Scheurich, Faye A. Briggs.
Synchronization, Coherence, and Events Ordering in Multiprocessors.
Computer 21(2):9-21, February 1988.

[43] D. Ferrari.
On the Foundation of Artificial Workload Design.
In *Proceedings of the ACM SIGMETRICS Conference*, pages 8-14. ACM, August, 1984.

[44] Joseph A. Fisher.
Wide Instruction Word Architectures: Solving the Supercomputer Software Problem.
In *Proc. INRIA International Symposium on Scientific Supercomputers*. Paris, France, February 1987.

[45] Brett D. Fleisch.
Distributed Shared Memory in a Loosely Coupled Distributed System.
In *Compcon '88; IEEE*, pages 182-184. San Francisco, CA., February -March, 1988.

[46] G.C. Fox, W. Furmanski.
Communication Algorithms for Regular Convolutions and Matrix Problems on the Hypercube.
In M.T. Heath (editor), *Proceedings of the 2nd Conference on Hypercube Multiprocessors*, pages 223-238. Oak Ridge National Laboratory, SIAM, October, 1986.

[47] G. C. Fox, W. Furmanski.
Hypercube Algorithms for Neural Network Simulation: The Crystal_Accumulator and the Crystal_Router.
In *3rd Conference on Hypercube Concurrent Computers and Applications; ACM*, pages 714-724. Pasadena, CA, January, 1988.

[48] Greg N. Frederickson, Ravi Janardan.
Optimal Message Routing Without Complete Routing Tables.
In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 88-97. Calgary, Alberta, Canada, August 1986.

[49] R.M. Fujimoto.
VLSI Communication Components for Multicomputer Networks.
PhD thesis, UCB, September, 1983.

[50] Edward F. Gehringer, Janne Abullarade, Michael H. Gulyn.
A Survey of Commercial Parallel Processors.
Computer Architecture News; ACM 16(4):75-107, September 1988.

[51] J. R. Goodman, M.-C. Chiang.
The Use of Static Column RAM as a Memory Hierarchy.
Proceedings of the 11th International Symposium on Computer Architecture, SIGARCH Newsletter; IEEE 12(3):167-174, June 1984.

[52] James R. Goodman, Philip J. Woest.
The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor.
Technical Report TR 766, Computer Sciences Department, University of Wisconsin, Madison, WI, April 1988.

[53] R. W. Gostick.
Software and Hardware technology for ICL DAP.
Australia Computer Journal; ICL 13(1), 1981.

[54] D.C Grunwald, D.A. Reed.
Networks for Parallel Processors: Measurements and Prognostications.
In G. Fox (editor), *The Third Conference on Hypercube Concurrent Computers and Applications*, pages 610-619. Caltech/JPL, ACM Press, January, 1988.

[55] K.D. Guenter.
Prevention of Deadlocks in Packet-Switched Data Transport Systems.
IEEE Transactions on Communications C-29(4):512, April, 1981.

[56] John L. Gustafson, Stuart Hawkinson, Ken Scott.
The Architecture of a Homogeneous Vector Supercomputer.
Proceedings of the 1986 International Conference on Parallel Processing; IEEE :649-652, August 1986.

[57] John P. Hayes, Trevor N. Mudge, Quentin F. Stout, Stephen Colley, John Palmer.
Architecture of a Hypercube Supercomputer.
Proceedings of the 1986 International Conference on Parallel Processing; IEEE :653-660, August 1986.

[58] D. Hensgen, R. Finkel, U. Manber.
Two Algorithms for Barrier Synchronization.
International Journal of Parallel Programming 17(1):1-17, 1988.

[59] M.P. Herlihy.
Impossibility and Universality Results for Wait-Free Synchronization.
In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 276-290. Aug, 1988.

[60] Anthony J. G. Hey.
Practical Parallel Processing with Transputers.
January 1988

[61] W.S. Ho, D.L. Eager.
A Novel Strategy for Controlling Hot Spot Congestion.
In *Proceedings of the 18th International Conference on Parallel Processing*. Pennsylvania State University, August, 1989.

[62] C.A.R. Hoare.
Communicating Sequential Processes.
Prentice/Hall International, Englewood Cliffs, N.J., 1985.

[63] F.H. Hsu.
A Two-Million Moves/s CMOS Single-Chip Chess Move Generator.
IEEE Journal of Solid-State Circuits SC-22(5):841-846, October, 1987.

[64] Jefferson, et al.
Distributed Simulation and the Time Warp Operating System.
Technical Report CSD-870042, UCLA, Los Angles, CA, August, 1987.

[65] M.R. Jerrum, S. Skyum.
Families of Fixed Degree Graphs for Processor Interconnection.
IEEE Transactions on Computers C-33(2):190-194, February, 1984.

[66] Chris Jesshope.
Reconfigurable Transputer Systems.
January 1988

[67] D.W.Jones.
An Empirical Comparison of Priority-Queue and Event-Set Implementations.
Communications of the ACM 29(4):300-311, April, 1986.

[68] Marta Kallstrom, Shreekant S. Thakkar.
Experience with Three Parallel Programming Systems.
Spring COMPCON '87, Digest of Papers; IEEE :344-349, February 1987.

[69] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, Daniel D. Sleator.
Competitive Snoopy Caching.
Technical Report CMU-CS-86-164, Carnegie-Mellon University, 1986.

[70] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkinsk, R. G. Sheldon.
Implementing a Cache Consistency Protocol.
In *Proceedings of the 12th International Symposium on Computer Architecture; IEEE*, pages 276-283. Boston, MA, June 1985.

[71] P. Kermani, L. Kleinrock.
Virtual Cut-Through: A New Computer Communication Switching Technique.
Computer Networks 3:267-286, 1979.

[72] Parviz Kermani, Leonard Kleinrock.
A Tradeoff Study of Switching Systems in Computer Communication Networks.
IEEE Transactions on Computers C-29(12):1052-1060, December 1980.

[73] B.W. Kerningham, D.M. Ritchie.
Software Series: *The C Programming Language.*
Prentice-Hall, 1978.

[74] R.E. Kessler, M. Livny.
An Analysis of Distributed Shared Memory Algorithms.
In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 498-505. IEEE, June, 1989.

[75] C-K. Kim, D.A. Reed.
Adaptive Packet Routing in a Hypercube.
In G.Fox (editor), *The Third Conference on Hypercube Concurrent Computers and Applications*, pages 625. JPL/Caltech, ACM Press, January, 1988.

[76] L. Kleinrock.
Communication Nets: Stochastic Message Flow and Delay.
McGraw-Hill, Inc., 1964.

[77] Leonard Kleinrock.
Analytic and Simulation Methods in Computer Network Design.
AFIPS Conference Proceedings, SJCC 1970; AFIPS :569-579, 1970.

[78] T.F. Knight Jr., A. Krymm.
A Selfterminating Low-Voltage Swing CMOS Output Driver.
IEEE Journal of Solid-State Circuits 23(2):457-464, April, 1988.

[79] D.E. Knuth.
The Art of Computer Programming.
Addison-Wesley Publishing Co., 1979.

[80] O.K. Kwon, B.W. Langley, R.F.W. Pease, M.R. Beasley.
Superconductors as Very High-Speed System-Level Interconnects.
Electron Device Letters EDL-8(12):582, December, 1987.

[81] P. Lancaster, M. Tismenetsky.
The Theory of Matrices.
Academic Press, 1985.

[82] S.S. Lavenberg.
Statistical Analysis of Simulation Outputs.
Technical Report, IBM, Yorktown Heights, NY, 1980.

[83] E.L. Lawler.
Combinatorial Optimization.
Holt, Reinhard and Winston, 1976.

[84] C.E. Leiserson.
Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing.
IEEE Transactions on Computers C-34(10):892-901, October, 1985.

[85] L. Letham, D. Wolf, A. Folmsbee.
A 128K EPROM Using Encryption of Pseudorandom Numbers to Enable Read Access.
IEEE Journal of Solid-State Circuits SC-21(5):881-888, October, 1986.

[86] Gavriela Freund Lev, Nicholas Pippenger, Leslie G. Valiant.
A Fast Parallel Algorithm for Routing in Permutation Networks.
IEEE Transactions on Computers C-30(2):93-100, February 1981.

[87] K. Li.
Shared Virtual Memory on Loosely-Coupled Multiprocessors.
PhD thesis, Yale University, October, 1986.

[88] H. Li, M. Maresca.
Polymorphic-Torus Network.
In *Proceedings of the 1987 International Conference on Parallel Processing; Penn State*, pages 411-413. University Park, Penn., August 1987.

[89] J.R. Lineback.
A Route to denser Multiprocessors.
Electronics :24-25, November 18, 1985.

[90] B.D. Lubachevsky.
Synchronization Barrier and Related Tools for Shared Memory Parallel Programming.
In *Proceedings of the 18th International Conference on Parallel Processing*. Pennsylvania State University, August, 1989.

[91] M.A. Marsan, G. Conte.
A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems.
ACM Transactions on Computer Systems 2(2):93, May, 1984.

[92] M. Martonosi, A. Gupta.
Tradeoffs in a Message-Passing Implementation of a Standard Cell Router.
In *Proceedings of the 18th International Conference on Parallel Processing*. Pennsylvania State University, August, 1989.

[93] W.R.Blood,Jr.
MECL System Design Handbook
4 edition, MOTOROLA Semiconductor Products Inc., 1983.

[94] G. Memmi, Y. Raillard.
Some New Results about the (d,k) Graph Problem.
IEEE Transactions on Computers C-31(8):784-791, August, 1982.

[95] M.K. Molloy.
On the Integration of Delay and Throughput Measures in Distributed Processing Models.
PhD thesis, UCLA, 1981.

[96] M.K. Molloy.
Discrete Time Stochastic Petri Nets.
IEEE Transactions on Software Engineering SE-11(4):417, April, 1985.

[97] M.K. Molloy.
Fundamentals of Performance Modeling.
In prepareation to be published, Computer Science Dept. Carnegie-Mellon University, Pittsburgh, PA 15213, 1988.

[98] D. Nassimi, S. Sahni.
An Optimal Routing Algorithm for Mesh-Connected Parallel Computers.
Journal of the ACM 27(1):6-29, January, 1980.

[99] Bruce Jay Nelson.
Remote Procedure Call.
Technical Report CSL-81-9 (Also CMU-CS-81-119), Xerox Palo Alto Research Center, Palo Alto, CA 94306, May 1981.

[100] J.Y. Ngai, C.S. Seitz.
A Framework for Adaptive Routing in Multicomputer Networks.
Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures , 1989.

[101] *NNCP Programmer's Manual, Mark III*
A edition, JPL/Caltech, 1986.

[102] A. Nowatzyk.
Advanced Design Tools for Programmable Logic Devices.
Technical Report CMU-CS-86-121, Carnegie-Mellon University, November, 1985.

[103] A. Nowatzyk.
PCB - A Printed Circuit Board Editor.
Technical Report CMU-CS-85-167, Carnegie Mellon University, June, 1985.

[104] A. Nowatzyk.
A Metastability Tester.
Technical Report, CMU Computer Science Dept., 1988.

[105] S.F.Nugent.
The iPSC/2 Direct-Connect Communication Technology.
In *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications*. JPL/Caltech, ACM, January, 1988.

[106] John F. Palmer.
A VLSI Parallel Supercomputer.
In Michael T. Heath (editor), *Hypercube Multiprocessors 1986*, pages 19-26. siam, Philadelphia, PA, 1986.

[107] John F. Palmer.
The NCUBE Family of High Performance Parallel Computer Systems.
In *3rd Conference on Hypercube Concurrent Computers and Applications; ACM*, pages 847-851. Pasadena, CA, January, 1988.

[108] Mark D. Papamarcos, Janak H. Patel.
A Low-Overhead Coherence Solution for Multi-Processors with Private Cache Memories.
Proceedings of the 11th International Symposium on Computer Architecture, SIGARCH Newsletter; IEEE 12(3):348-354, June 1984.

[109] S.K. Park, K.W. Miller.
Random Number Generators: Good Ones Are Hard To Find.
Communications of the ACM 31(10):1192-1201, October, 1988.

[110] N.M.Patel, P.G.Harrison.
On Hot-Spot Contention in Interconnection Networks.
In *Sigmetics*, pages 114-123. ACM, Santa Fe, May, 1988.

[111] PC-Cube.
Distributed by ParaSoft Corp. 27415 Trabuco Circle, Mission Viejo, CA 92692, (714) 380-9739, 1988.

[112] G.F. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, V.A. Norton, J. Weiss.
An Introduction to the IBM Research Parallel Processor Prototype (RP3).
In J. J. Dongarra (editor), *Special Topics in Supercomputing*. Volume 1: *Experimental Parallel Computing Architectures*, pages 123-140. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1987.

[113] G.F.Pfister, V.A.Norton.
"Hot Spot" Contention and Combining in Multistage Interconnection Networks.
In *Proc. of the 1985 Conference on Parallel Processing*, pages 790-797. IEEE, 1985.

[114] Raetz, G.M.
Sequent general purpose parallel processing system.
In *Northcon/87. Conference Record*, pages 22-24. IEEE; ERA; Electron. Manuf. Assoc, Western Periodicals Co., North Hollywood, CA, USA, September, 1987.

[115] U. Ramachandran, M. Ahamad, M.Y.A. Khalidi.
Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer.
In *Proceedings of the International Conference on Parallel Processing*, pages 160-169. August, 1989.

[116] C.V. Ramamoorthy, G.S. Ho.
Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets.
IEEE Transactions on Software Engineering SE-6(5):440, September, 1980.

[117] Daniel A. Reed.
Optimal Routing in the Cube-Connected Cycles Interconnection Network.
Technical Report CSD-TR-412, University of North Carolina, June 27, 1983.

[118] D.A. Reed, C.K. Kim.
Packet Routing Algorithms for Integrated Switching Networks.
In *Sigmetrics*, pages 7-15. ACM, 1987.
Not really interesting.

[119] U.L. Rohde.
ISBN 0-13-214239-2: *Digital PLL Frequency Synthesizers - theory and Design.*
Prentice Hall, Inc., Englewood Cliffs, NJ 07632, 1983.

[120] H. Rudin.
On Routing and "Delta Routing": A Taxonomy and Performance Comparison of Techiques for Packet-Switched Networks.
IEEE Transactions on Communications COM-24(1):43-59, January, 1976.

[121] L.S. Rudolph.
Software Structures for Ultraparallel Computing.
PhD thesis, NYU, December, 1981.

[122] Larry Rudolph, Zary Segall.
Dynamic Decentralized Cache Schemes for an MIMD Parallel Processor.
In *Proceedings of the 11th International Symposium on Computer Architecture*, pages 340-347. June 1984.

[123] R.M. Russell.
The CRAY-1 Computer System.
Communications of the ACM 21(1):63-72, January, 1978.

[124] R.M. Russell.
The CRAY-1 Computer System.
Communications of the ACM 21(1):63, January, 1978.

[125] K. Li, R. Schaefer.
A Hypercube Shared Virtual Memory System.
In *Proceedings of the 18th International Conference on Parallel Processing*. Pennsylvania State University, August, 1989.

[126] Gary E. Schmidt.
The Butterfly Parallel Processor.
Proceedings of the Second International Conference on Supercomputing :362-365, 1987.

[127] C.L. Seitz.
The Cosmic Cube.
Communications of the ACM 28(1):22-33, January, 1985.

[128] C. Seitz et al.
Wormhole Chip Project Report.
Technical Report, Caltech, Winter, 1985.

[129] C.L. Seitz, W.C. Athas, C.M. Flaig, A.J. Martin, J. Seizovic, C.S. Steele, W-K. Su.
The Arcitecture and Programming of the Ametek Series 2010 Multicomputer.
In G.Fox (editor), *The Third Conference on Hypercube Concurrent Computers and Applications*, pages 33. JPL/Caltech, ACM Press, January, 1988.

[130] C. H. Sequin.
Doubly Twisted Torus Networks for VLSI Processing Arrays.
Proceedings of 8th Annual International Symposium on Computer Architecture, SIGARCH Newsletter 9(3):471-479, May 1981.

[131] Andre Seznec, Yvon Jegou.
Towards a large number of pipeline processors in a tightly coupled multiprocessor using no cache?
In *1988 International Conference on Supercomputing (ICS); ACM Press*, pages 611-620. St. Malo, France, July 1988.

[132] David Elliot Shaw.
NON-VON: A Parallel Machine Architecture for Knowledge-Based Information Processing.
IJCAI-81 2:961-963, 1981.

[133] D.D. Sleator, R.E. Tarjan.
Self-Adjusting Binary Search Trees.
Journal of the ACM 32(3):652-686, July, 1985.

[134] B.J. Smith.
Architecture and Applications of the HEP Multiprocessor Computer System.
In *Proceedings of SPIE*. August, 1981.

[135] A. J. Smith.
Cache Memories.
ACM Computing Surveys 14(3):473-530, September 1982.

[136] Alan Jay Smith.
Bibliography and Readings on CPU Cache Memories and Related Topics.
Computer Architecture News 14(1):22-42, January 1986.

[137] Per Stenstrom.
Reducing Contention in Shared-Memory Multiprocessors.
Computer 21(11):26-37, November 1988.

[138] K.S. Stevens, S.V. Robison, A.L. Davis.
The Post Office - Communication Support for Distributed Ensemble Architectures.
6th International Conference on Distributed Systems , 1986.

[139] K.S. Stevens.
The Communication Framework for a Distributed Ensemble Architecture.
AI Technical Report 47, SRI, February, 1986.

[140] B. Stroustrup.
A Set of C++ Classes for Co-routine Style Programming.
Technical Report CSTR-108, AT&T Bell Laboratory, Murray Hill, NJ, November, 1984.

[141] B. Stroustrup.
The C++ Programming Language.
Addison-Wesley Publishing Company, 1986.

[142] H. Sullivan, T.R. Bashkow.
A Large Scale Homogenious Machine.
Proc. 4th Annual Symposium on Computer Architecture :105-124, 1977.

[143] T. Szymanski.
On The Permutation Capability of a Circuit Switched Hypercube.
In *Proceedings of the 18th International Conference on Parallel Processing*. Pennsylvania State University, August, 1989.

[144] *Product description: IMS T414 Transputer*
INMOS Corp., PO Box 16000, Colorado Springs, CO 80935, 1985.

[145] Texas Instruments.
TI develops quantum effect transistor.
Electronic News 34(1737):20-21, December, 1988.

[146] R.H. Thomas.
Behavoir of the Butterfly Parallel Processor in the Presence of Memory Hot Spots.
In *Proceedings of the International Conference on Parallel Processors*. BBN Labs. Inc., 1986.
Claims that there are essentionally no hot spots.

[147] J. E. Thornton.
Design of a Computer: The CDC 6600.
Scott, Foresman & Co., Glenview, IL, 1970.

[148] N.F. Tzeng.
Design and Analysis of a Novel Combining Structure for Shared Memory Multiprocessors.
In *Proceedings of the 18th International Conference on Parallel Processing*. Pennsylvania State University, August, 1989.

[149] L. G. Valiant.
Optimality of a two-phase strategy for routing in interconnection networks.
IEEE Transactions on Computers C-32(9):861-863, September 1983.

[150] Alexander V. Veidenbaum.
A Compiler-Assisted Cache Coherence Solution for Multiprocessors.
Proceedings of the 1986 International Conference on Parallel Processing; IEEE :1029-1036, August 1986.

[151] Colin Whitby-Strevens.
The Transputer.
In *Proceedings of the 12th International Symposium on Computer Architecture; IEEE*, pages 292-300. Boston, MA, June 1985.

[152] L. Wittie, C. Maples.
MERLIN: Massively Parallel Heterogenious Computing.
In *Proceedings of the 18th International Conference on Parallel Processing*. Pennsylvania State University, August, 1989.

[153] Chuan-Lin Wu, Tse-Yun Feng.
The Universality of the Shuttle-Exchange Network.
IEEE Transactions on Computers C-30(5):324-332, May 1981.

[154] C. Wu, T. Feng.
Tutorial: Interconnection Networks for Parallel and Distributed Processing.
IEEE Computer Society Press, 1984.

[155] William A. Wulf, Samuel P. Harbison.
Reflections in a Pool of Processors -- an experience report on C.mmp/Hydra.
In *AFIPS Proc. of the NCC*, pages 939-951. February, 1978.

[156] *The Programmable Gate Array Design Handbook*
1 edition, Xilinx Inc., 2069 Hamilton Avenue, San Jose, CA 95125, 1986.

Table of Contents

List of Figures

List of Tables