VLSI COMMUNICATION COMPONENTS FOR MULTICOMPUTER NETWORKS

Richard Masao Fujimoto

ABSTRACT

Advances in microprocessor technology will soon make general purpose computing systems composed of thousands of VLSI processors economically feasible. A high-performance communication system to interconnect these processors is of crucial importance to exploit the parallelism inherent in applications such as circuit simulation and signal processing. This thesis discusses issues in the design of universal VLSI communication components to be used as the building blocks for constructing robust, high-bandwidth, point-to-point networks. The components provide enough flexibility to serve a wide variety of multicomputer configurations and applications. They feature special purpose hardware to implement communication functions traditionally implemented with network software.

A communication network constructed from the proposed components is modeled as a set of nodes (components) connected by bidirectional communication links. Because of technological constraints, the total I/O bandwidth of each node is limited to some fixed value, and assumed to be equally divided among the attached links. Increasing the number of links per component leads to a reduction in the average number of hops between nodes, but at the cost of reduced link bandwidth. This "hop count / link bandwidth" tradeoff is examined in great detail through M/M/1 queueing models and simulations using traffic loads generated by parallel application programs. These results indicate that a small number of links should be used. It is also found that a significant improvement in performance is obtained if a component is allowed to *immediately* begin forwarding a message when the selected output link becomes idle, regardless of whether or not the end of the message has arrived. Finally, mechanisms which efficiently transmit a single message to multiple destinations are seen to have a significant impact on performance in programs relying on global information.

The complexity of the circuitry required to implement a communication component is examined. Schemes for providing hardware support for communication functions — routing, buffer management, and flow control — are presented. Estimates of the number of buffers and the degree of multiplexing on each communication link are determined. The amount of circuitry to implement a communication component is computed, and it is seen that the proposed communication component could be implemented with technology available today. Design recommendations for the implementation of such a component are made.

Richard M. Fujimoto 8/25/83

Richard M. Fujimoto

Carlo H. Séquin 8/25/83

Carlo H. Séquin
(committee chairman)

TABLE OF CONTENTS

CHAPTER ONE

INTRODUCTION

The processing power of a general purpose computing system can be increased in two ways. One approach, which has the advantage that old software can be re-used, is to increase the speed of an existing computer system by technological means without altering the basic organization of hardware components. Much of this effort focuses on the development of very high speed electrical circuits through the use of new materials, e.g. Joseph junctions [Ghee82] or gallium arsenide [Long80]. The primary mode of operation in such a system is *sequential*, although limited amounts of parallelism may be employed in certain portions of the processor. The huge investment in existing software fuels the effort to make this approach commercially viable.

The second approach to building high-performance computer systems relies on a more general exploitation of parallelism, e.g. by using a large pool of relatively inexpensive computers that operate in *parallel* to solve a large problem which has been decomposed into a number of smaller subproblems. Advances in integrated circuit technology have made this approach feasible by allowing the construction of chips using a very large scale of integration (VLSI) to pack hundreds of thousands of transistors onto a small piece of silicon. It is in this latter approach that VLSI technology can have a truly dramatic impact in the structure of tomorrow's computing systems. This thesis will focus on the exploitation of parallelism to achieve high performance, and in particular, on the hardware necessary to support high bandwidth communications among thousands of processors.

A key design parameter of multicomputer systems (systems composed of more than one processor interconnected by a communication network) is processor *granularity*, i.e. the size and capability of the individual processing elements. At one end of the spectrum, each processing element is very small and limited in capability, allowing an entire multiprocessor system to be placed on a single chip. Examples are the special purpose systolic array processors which are particularly suitable for high-throughput signal processing applications [Kung80, Kung82], the 'tree-machine' developed at Caltech [Brow80], and the Boolean Vector Machine proposed by Wagner [Wagn83]. Since the unit to be replicated is small, often consisting of only an arithmetic unit and a few data registers, the granularity of the system is very fine. The other extreme, using very large granules, is exemplified by such supercomputers as the S1 which employs a few large, high-performance processors [Widd80]. Each processor consists of thousands of integrated circuit chips. Commercially available multiprocessor systems built by IBM [Ensl74a] or UNIVAC [Ensl74b] also belong to this category.

Earlier work in the X-tree project [Desp78, Sequ78] advocated an intermediate granule size equal to that of a single VLSI chip. For a *general purpose* system, some minimum complexity is required in each processing element to allow enough flexibility to enable several to cooperate productively across a wide range of applications. The simple processor advocated by the "small granule" approach is too small a building block for a general purpose computer. On the other hand, a very large granule size forces closely coupled components such as a processor and its associated memory to be implemented on separate chips, thus increasing the performance penalties resulting from off-chip communications. An intermediate granule size equivalent to a single-chip microprocessor and its memory forms an entity with enough processing power for general-purpose computing, but is still small enough to be implemented on a single chip.

Advances in VLSI technology are making general-purpose computing systems composed of thousands of processors economically feasible. The processors, however, comprise only a portion of the system. The communication system that interconnects the processors is of equal importance. The performance of many multiprocessor systems has been limited by insufficient inter-processor input/output (I/O) bandwidth. Furthermore, the communication system may dominate the hardware cost. In Cm*, for example, the hardware responsible for setting up the communication paths (i.e. the k-maps) was considerably more expensive than that used in the processors [Swan77b]. It is clearly desirable to also exploit VLSI technology to reduce the cost of the switching hardware. This thesis discusses issues in the design of universal VLSI switching components to be used as the building blocks for robust, high-bandwidth, communication networks with enough flexibility to serve a wide variety of multicomputer configurations and applications.

1.1. The Concept: Modular, High-Bandwidth Communication Networks

A collection of VLSI communication components that can be combined into networks of high bandwidth and arbitrary topology is envisioned. Any processor with the proper interface can be attached to this communication system. Only a few types of VLSI building blocks are required, providing modularity and incremental expansibility (the ability to create a larger computing system by adding hardware to an existing system). The goal is to develop components that plug together easily and completely hide from the user the details of the information transfer within the network. Just as the telephone system hides from the user the details of routing calls and transferring voice information, these new communication modules handle the low-level details of transferring data by providing circuitry to perform communication functions such as handshaking, message routing, buffering, and flow control. For the system designer, the lowest level primitive that must be dealt with is the information packet or the block of data to be transmitted. For the user of the final system, the network provides end-to-end communications much like the telephone system.

Figure 1.1 gives a conceptual view of such a system, divided into a communication domain (C) using these VLSI communication components, and a processor domain (P) dedicated primarily to the user's computations. Required properties of the communication domain include unrestricted network topology, modularity, incremental expansibility, decentralized control, and the ability to recover from certain classes of failures. Low-latency, high-bandwidth communications are required to achieve good performance in applications such as circuit simulation and signal processing. This research will focus on networks using

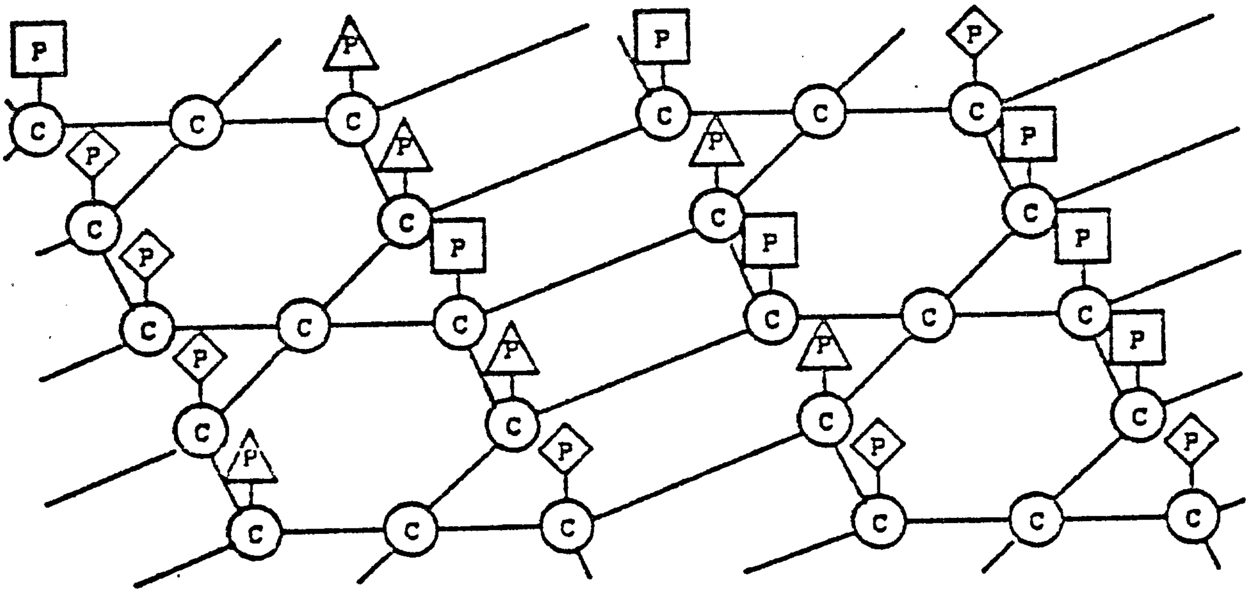


Figure 1.1. *Separation of a Multicomputer into a Communications Domain (C) and a Processor Domain (P).*

dedicated links. The proposed communication domain is designed to support high-performance communications among a large number, possibly thousands, of processors.

The proposed components perform several basic "store-and-forward" communication functions. Each component receives messages from any attached processor(s) and from other communication components. Before a message can be forwarded to the next component/processor, an output link must be selected via some *routing algorithm*. After an output link has been selected, the message is forwarded over this link. To handle conflicts which occur when more than one message is routed over the same output link at the same time, buffers are provided to hold waiting messages. Each communication component must provide circuitry for managing these buffers. Finally, to avoid loosing messages when the buffer space in a component is exhausted, a *flow control* mechanism is required to throttle arriving traffic. Details of mechanisms which perform these functions are discussed in chapter 4, as well as estimates of the amount of circuitry required to implement them.

The types of processors used in the processor domain may vary depending on the application, but the interface to the communication system is standardized. This separation of the communication domain and the computation domain relieves the processors of much of the overhead associated with the forwarding of messages destined for different nodes. It makes possible the development of general purpose communication hardware that is suitable for a wide range of applications, and also provides the flexibility to construct heterogeneous systems containing many different types of specialized processors.

One may note the similarity between the components described here and communication processors used in loosely coupled computer networks. An example of such a processor is the Interface Message Processor (IMP) used in the ARPANET, a computer network linking several major universities and institutions around the world [Hear70]. Indeed, many problems associated with loosely coupled computer networks (e.g. routing, buffering, flow control) also appear in this context. However, our design is not merely a scaled down version of the ARPANET. The key differences arise from the aim at higher bandwidth and lower latency, intrinsically lower error and failure rates within the communication hardware, and the envisioned implementation in VLSI.

1.2. Definition of Terms

A number of terms will be used throughout this thesis. In order to avoid confusion, their meaning in the context of this report will now be defined.

First, each *message* sent into the communication domain consists of some number of fixed length *packets*. Communication components deal exclusively with packets. Here, it is often the case that each message fits into a single packet, so the two terms will be used interchangeably when no confusion arises.

Each communication component contains some number of *ports* and *links*. A link refers to the collection of wires connecting a communication component to another such component or to a computation processor. The link is external to the chip. A port is the circuitry within the chip which drives data onto, and receives data from the link. When necessary, the distinction is made between an *input port*, which receives data entering the chip, and an *output port*, which sends data away from the chip. Each link is bidirectional and full duplex, i.e. each may simultaneously carry traffic in both directions. There is exactly one link attached to each port, so when referring to the "number of ports/links", the two terms are used interchangeably.

A *virtual circuit* refers to an end-to-end connection from one processor (say A) through a certain number of switching components to another processor

(B). Here, a virtual circuit (or circuit for short) refers to the directed path through the network from A to B. As will be discussed in chapter 4, virtual circuits must be "established" before messages can be sent, and all data sent on the same circuit follow the same path. In order to distinguish data on different virtual circuits which are using the same physical link, each link is divided into some number of *virtual channels*, with each channel carrying data for one circuit. Thus, a virtual circuit is a sequence of channels from one processor to another.

In the discussion presented above, virtual circuits were defined to have a single source and destination processor. An exception to this is a *multicast circuit* which has a single source, but more than one destination. A message sent on a multicast circuit is replicated within the network, and a separate copy is received by each destination processor. Such a mechanism is useful in applications requiring the same data to be distributed to several other processors, as will be discussed in chapter 3.

Another term used extensively in this thesis is *virtual cut-through* [Kerm79]. This refers to a mechanism in which the forwarding of data packets can begin as soon as the packet *header* (here, the first byte) arrives, if the proper outgoing link is idle. Without cut-through, forwarding would have to be delayed until the entire packet has arrived in its buffer. It will be seen that this immediate forwarding mechanism can lead to a significant improvement in performance.

Finally, several terms are used regarding the performance of the multicomputer and the communication network. *Bandwidth* refers to the amount of traffic a communication medium can carry over a fixed period of time, typically measured in bits per second. The medium may be a single communication link or the entire network as a whole. *Delay* refers to the amount of time which elapses from when a message/packet enters the communication medium, until it leaves. A more precise definition will be given later. *Latency* is another term which is used interchangeably with delay. Finally, *speedup* refers to the ratio of the execution time of an application program on a single-processor computer system to the corresponding time on a multicomputer system. Intuitively, it indicates how much faster the program executes on the multicomputer.

1.3. Previous Work in Communication Networks

The research most applicable to the work reported here may be broadly divided into two categories: loosely coupled computer networks, and interconnection networks for tightly coupled multiprocessors. Each of these will now be discussed in turn.

1.3.1. Loosely Coupled Computer Networks

A great deal of research has been carried out in loosely coupled communication networks. Although many of the constraints and goals in the design of these networks are different from those discussed here, much of this research is still applicable. A complete overview of the literature in this field is beyond the scope of the present discussion. Textbooks such as [Davi73, Tane81, Kuo81, Ahuj82] provide excellent introductions to the field as well as extensive bibliographies. The research most relevant to the communication component networks discussed here deals with message routing techniques and protocols for error free transmission. Other research in relevant areas (e.g. deadlock prevention) will be described later as the need arises.

Message routing is the process of selecting a route, i.e. a path, through a network from a processor sending a message to the processor receiving it. Research in this area is usually concerned with developing general techniques which are applicable to networks of arbitrary topology. An overview and taxonomy of practical routing algorithms is described in [McQu74, Gerl81]. Practical routing algorithms used by specific networks have been described for several networks, e.g. Arpanet [McQu74, McQu80], Datapac [Spro81], Tymnet [Tyme81, Rind77], and IBM's SNA [Juen76]. Other heuristic routing schemes have been proposed, among them [Floy62, Fran71, Chou81]. Finally, routing techniques based on more rigorous mathematical performance models include [Cant74, Gall77, Sega77]. Networks constructed from communication components must also use some routing algorithm to establish virtual circuits, so much of the work described above is applicable here.

Another significant area of research in loosely coupled networks is in the design of protocols to ensure reliable transmission of data through the network. A good survey of work in this area and an extensive bibliography is reported in [Pouz78]. Much of the work in protocols has centered around the development of a layered structure of communication protocols, and defining standard protocols within each layer. As a result of this work, a standard has been defined by the International Standards Organization (ISO), and is now widely used by many computer manufacturers [Zimm80].

Of special interest here are protocols for flow control, i.e. mechanisms which control the flow of traffic through the network. Good overviews of work in this area are presented in [Pouz81, Pouz76, Kahn72]. Flow control procedures in Datapac and Tymnet are described in [Spro81, Tyme81, Rind77], while a hierarchical flow control scheme is presented and analyzed in [Chu77].

Most of the protocols developed for loosely coupled networks are inappropriate for the networks discussed here. This is because these protocols make assumptions which are not valid in closely coupled multicomputer networks. In particular, loosely coupled networks cover wide geographic areas and are subject to adverse environmental conditions, so error rates can be expected to be much higher than in networks constructed from communication components. As a result, protocols in loosely coupled networks typically pay close attention to detecting and retransmitting corrupted messages at *all* levels of the layered structure. With low error rates however, transmission errors can be handled by high-level (i.e. end-to-end) protocols, freeing lower level mechanisms within the network to incorporate such performance improving techniques as virtual cut-through. Thus, the protocols used in loosely coupled networks are normally too inefficient for the networks discussed here.

1.3.2. Interconnection Networks for Closely Coupled Multiprocessors

A great deal of research has also been done in the area of interconnection networks for closely coupled multiprocessor systems. "Classical" research in interconnection networks examines single- and multistage-interconnection networks constructed from small (typically 2 by 2) crossbar switches. These networks are discussed in the context of establishing processor to memory or processor to processor communications. The bulk of the remaining research in the field focuses on interconnection topologies. A good survey of work in both of these areas is given in [Feng81].

The work in single- and multiple-stage interconnection networks can be partitioned into two categories: networks for SIMD (single-instruction stream, multiple-data stream) computers, and networks for MIMD (multiple-instruction stream, multiple-data stream) computers. A survey of interconnection networks for SIMD computers is given in [Sieg79a]. Many of the SIMD networks are also applicable to MIMD machines.

SIMD systems are special purpose computers typically used for large computational tasks requiring many vector operations. A "typical" SIMD computer is shown in figure 1.2. Here, a number of processors are connected to memory modules through an interconnection network. The controller broadcasts instructions to the various processors. All processors execute the same instruction on each clock cycle. Each performs some computation using data from one of the memory modules. If data (e.g. elements of a vector) are properly distributed across the memory modules, then conflicts in accessing the memories can be avoided.

In the scenario described above, the interconnection network effectively *aligns* data scattered across the memory modules. Alternatively, the network can be thought of as performing some permutation of input lines to output lines. Thus, these networks are sometimes referred to as alignment or permutation networks. Networks which support any permutation of input to output lines are

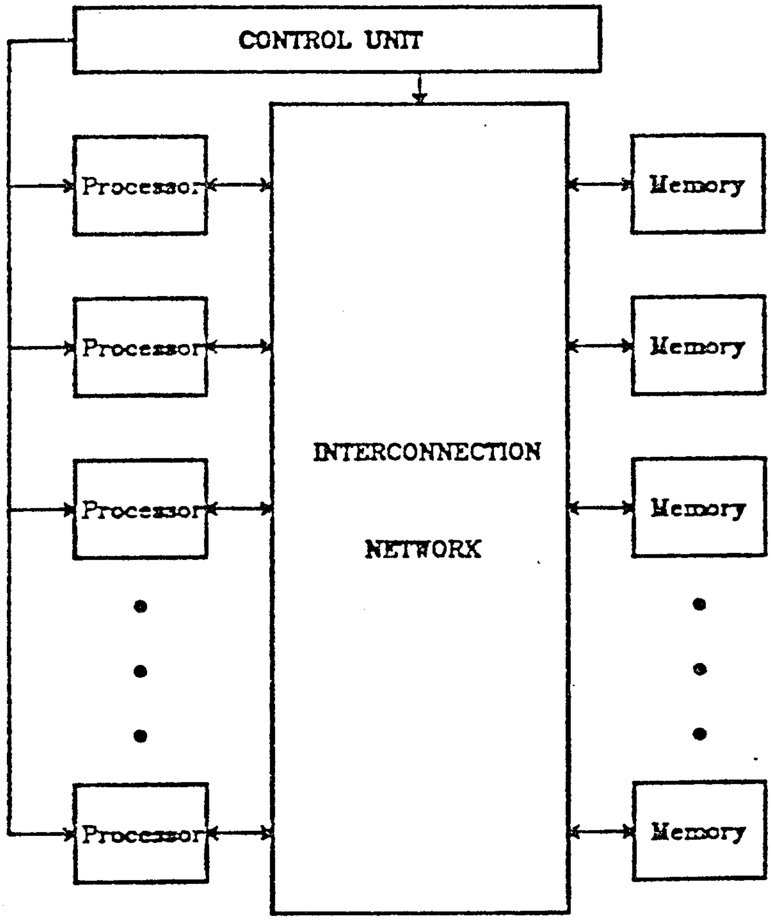


Figure 1.2. *A typical SIMD Machine.*

called "nonblocking". The crossbar switch [Wulf72] and the Clos network [Clos53] are examples of nonblocking networks. Nonblocking networks become prohibitively expensive as the number of processors and memory modules grows, so less expensive networks which support some subset of all possible permutations (called "blocking" networks) have been explored. Examples of blocking permutation networks include the shuffle exchange [Ston71], banyan networks [Goke73], the omega network [Lawr75], the flip network used in the STARAN processor [Batc76], the indirect binary n-cube [Peas77], the baseline [Wu80a], and the reverse-exchange network [Wu80b]. An introduction and overview of this work is presented in [Chen81]. Classes of networks which subsume many of the specific networks listed above have also been discovered, e.g. the delta network class [Pate81] and the multistage cube [Sieg81]. Thus it is not surprising that many of the variations described above have no, or only slightly different, performance characteristics.

Extensive analyses and comparisons of different permutation networks have been performed. For example, in [Sieg79b] bounds are derived for the time required for some networks to simulate others. Parker shows that the inverse omega network and the indirect binary n-cube have identical switching characteristics [Park80], while in [Wu80a] it is shown that the flip network, omega network, indirect binary n-cube, and one form of the banyan network are topologically isomorphic. Equivalence classes among permutation networks are defined in [Prad80]. Other analyses describing performance and permutation properties include [Fran81, Nass81, Than81]. Extensions which allow the set of performable permutations to be expanded, typically by cascading more than one network or allowing multiple iterations through the same network, are discussed in [Yew81, Wu81a]. A theory for composing the permutations performed by the omega network is discussed in [Stei83]. Finally, parallel algorithms for setting up the switches in permutation networks are described in [Lev81, Nass82]. Although one disadvantage of the permutation networks described above is that the time complexity to setup an n input network given some permutation is $O(n \log n)$ with the fastest known serial algorithms, these papers allow settings to be determined in as little as $O((log\ n)^2)$ time in some situations when n processors are used to perform the computation.

The topologies of the networks described above can also be applied to networks for MIMD machines. Here however, average message delay and network bandwidth are used as performance measures rather than the number of permutations performed. In this context, it has been shown that most of the networks described above yield virtually the same performance [Pate81].

These interconnection networks represent one class of networks which could be implemented with the communication components described here. Special switches designed specifically for these permutation networks (typically, 2 by 2 crossbar switches) have two *apparent* advantages over the general purpose components proposed in this thesis. First, since the network topology is fixed, they may be optimized for efficient message routing. However, with a virtual circuit transport mechanism (described in chapter 4), message routing is reduced to a single read from a relatively small (a few hundred entries at most) table. In current technology, this can be performed in a single clock cycle, where the clock cycle is determined by the rate at which data can be clocked into a chip, so any advantage derived from optimized routing is minimized.

Second, current implementations of the simple 2 by 2 switches require less circuitry than the components described here. However this difference is largely due to the improved functionality of our communication component, rather than some fundamental increase in complexity. The components described here use more sophisticated buffer management strategies than are typically used in the 2 by 2 switches, and a microcoded engine is provided for implementing failure recovery protocols. Since the performance of a switching node is limited by I/O bandwidth (i.e. there is some maximum number of pins on each chip and some maximum rate at which each pin can be driven) and since off-chip communications are typically an order of magnitude slower than on-chip speeds [Sequ78], this additional complexity is not detrimental to the clock rate. In addition, general purpose components provide enough flexibility to allow networks to be tailored to the communication needs of the system. For example, more bandwidth could be placed near expected points of congestion, e.g. around the disks. Thus, the communication components described here can achieve at least as much performance as the switching nodes in the permutation networks, if not more, as well as provide additional flexibility to the system designer.

Other research in interconnection networks focuses on defining attractive network topologies. This work can be classified into two categories: networks for special purpose computation, and networks for general purpose computation. Special purpose network topologies are aimed toward achieving an efficient mapping of some class of algorithms onto the network. General purpose networks cannot assume any specific algorithm, so they try to optimize some general criteria for goodness, e.g. average hop count between pairs of nodes.

An introduction to research in special purpose networks designed for efficient execution of specific algorithms is presented in [Gott82]. In [Thom80] a theory of VLSI is introduced and bounds for area/time tradeoffs in implementing VLSI chips for specific computations (e.g. the FFT) are derived. Also, the work in systolic architectures examines two dimensional networks suitable for executing certain numerical algorithms for signal processing and matrix manipulation problems [Kung80, Kung82]. There has also been an extensive amount of work in matching problems to such well known topologies as the perfect shuffle

[Ston71], the mesh [Nass79, Prep83], and tree networks [Deke83, Nath83]. The cube-connected-cycles network is another network which exhibits properties favorable for the efficient implementation of certain parallel algorithms [Prep81].

Much of the work in topologies for general purpose computation focuses on defining networks which achieve some characteristic expected to lead to good performance (e.g. small average hop count). One problem in this domain which has received some attention is the "(d,k) graph problem", in which the goal is to maximize the number of nodes in a graph of degree d, and diameter k [Acke65, Frie66, Korn67, Stor70, Toue79, Imas81, Memm82, Amar83]. Other topologies recently proposed for communication networks include ringed trees [Desp78], snowflakes [Fink80], clusters [Wu81b], chordal rings [Arde81], C_S' graphs [Farh81], binary trees [Horo81], cube connected cycles [Prep81], hypertrees [Good81], lens networks [Fink81], multiple tree structures [Arde82], and mobius graphs [Lela82a, Lela82b]. Comparisons of some of these structures are reported in [Witt81, Swar82, Reed83]. Finally, other research examines topologies which are attractive for fault tolerance, e.g. [Prad82, Adam82].

Most of this work is directly applicable to the networks studied here, since it refers to topologies which can be constructed from the proposed components. These earlier studies are at a higher level of abstraction than those presented here however. While the work reported above focuses entirely on system performance, the work described here is aimed at low level design decisions, e.g. the number of I/O ports on each chip, and considers the constraints imposed by a VLSI implementation. The impact of these constraints on overall performance is examined. Some work in implementation issues has been performed by Franklin, however this has been restricted to studies of partitioning certain switching structures, e.g. crossbar switches and banyan networks, into modules suitable for VLSI implementation [Fran82].

In the area of communication components, some building block modules have been proposed. In [Hopp79] a packet switched 2 by 2 crossbar node using unidirectional links is proposed as a switching node. Routing information is carried with each packet as a sequence of bits, with each bit indicating the direction the packet is to follow at intermediate nodes. One disadvantage of this scheme is that the destination address of each node varies according to the location of the sender of the message, and senders are required to generate this routing information themselves. With arbitrary networks, this computation is somewhat complex, and recomputations are necessary if the topology changes because of component failures or network expansion. Simple flow control and buffering scheme are provided, although they do not prevent some of the buffer hogging and deadlock problems discussed in chapter 4. Unlike the design presented here, no processor is provided in each switching node. Overall, this component can be regarded as similar in intent to the components described here, however much less sophisticated in functionality.

A component similar to that described above is the Dual Interconnecting Modular Network Device, or DIMOND [Jans80]. Again, this component has two input and two output links, and each message carries detailed routing information with it. In [Jans80] details of the implementation of the DIMOND are explained, as well as its use in constructing networks such as rings and trees. A minimal amount of buffering is provided in each component (a single register on each output port).

Finally, a 3 input, 3 output link component called STICS (Synchronous Triangular Interprocessor Connection Scheme) has also been proposed [Rile82]. These components can only be applied to a very restricted class of topologies however, and thus are not as general as the components described here.

To the author's knowledge, all of the previous work in VLSI communication components has emphasized simplicity at the expense of generality, functionality, and/or performance. With advances in VLSI however, chip densities are increasing at a rapid rate, and more functionality can readily be integrated onto a single chip. Thus, more sophisticated designs achieving greater functionality and performance are becoming practical. The communication components described here represent one attempt to design and analyze the performance of such a switching chip.

The work presented here is a continuation and extension of the work in the communication switch for the X-tree project [Sequ78, Desp78]. Work in the low-level design of the internal structure of an X-tree node are described in [Laur79, Grif79, Fuji80, Wong81]. Perspectives and lessons learned from these designs and the X-tree project as a whole are described in [Sequ82]. While the work in X-tree focussed on a particular topology, the components proposed here provide more flexibility, allowing construction of arbitrary high-performance communication networks.

1.4. Overview of Thesis

This thesis focuses on the design of VLSI communication components, and the impact of certain design decisions on system performance. The remainder of this thesis is organized as follows: In chapter 2, the tradeoff between the number and bandwidth of the communication links is discussed in the context of a single-chip implementation of the proposed communication component. It is seen that the I/O bandwidth of each component is fixed, and assumed to be equally divided among the attached links. The communication network is modeled as a set of nodes (components) interconnected by communication links. The question of whether each node should have a large number of low bandwidth links (implying relatively few "hops" between a given pair of nodes) or a small number of high bandwidth links (implying many hops) is addressed. Each node of a topology requiring b "branches" or links to neighboring nodes can be implemented by a cluster of p-port communication components. M/M/1 queueing models are used to analyze optimal value of p, using average delay and total bandwidth of the "cluster node" as performance measures. It is found that components with a small number of ports yield cluster nodes with the most bandwidth, and smallest average delay.

Cluster nodes using components with a small number of ports require more chips than those using a larger number of ports. Thus, the cluster node studies do not consider differing chip counts. Networks with the same number of components are compared within certain classes of network topologies (e.g. lattices and trees). It is found that components using a small number of ports, e.g. from 3 to 5, tend to yield networks with lower average delay, but less bandwidth than networks using components with a larger number of ports. It is argued however, that while network bandwidth can be increased by using more communication chips, average delay cannot be reduced so easily. Thus, components with a small number of ports should be used.

In chapter 3, results of simulation studies are presented. Here, parallel application programs are used to create traffic loads for networks constructed from communication components. The traffic loads cover a wide variety of different communication patterns. Both cluster node networks and networks using approximately the same number of components are examined. The simulation results support the conclusion of the previous chapter that a small number of ports should be used.

Chapter 4 examines the design of a communication component in greater detail, and discusses the complexity of the required circuitry. Various mechanisms for transporting data through any communication network are discussed, and a mechanism based on virtual circuits is argued to be the most appropriate for the networks discussed here. Schemes for providing hardware support for communication functions — routing, buffer management, and flow control — are presented, and estimates of the number of buffers and virtual channels are determined. Based on these estimates, the amount of circuitry to implement a communication component is estimated, and a floorplan for one implementation is shown.

Finally, chapter 5 presents concluding remarks, and a summary of design recommendations for implementing general purpose, high-performance VLSI communication components.

CHAPTER TWO

PERFORMANCE EVALUATION STUDIES

In this chapter, the performance of networks constructed from VLSI communication components is evaluated. The optimal number of communication ports for each chip is discussed in detail. The performance improvement resulting from incorporating a virtual cut-through mechanism into the communication hardware is also studied.

The first section discusses constraints imposed by a single-chip implementation of the communication components. These constraints lead to a tradeoff between the number and bandwidth of the communication links (or ports since there is one link per port). The following section discusses analytical studies evaluating the performance of various networks constructed with VLSI communication components as a function of the number, and thus of the bandwidth of the communication links.

2.1. VLSI Constraints

A VLSI chip is subject to a number of technological constraints. Violation of these constraints will result in a chip which cannot be manufactured in large quantities, or which cannot be depended upon for reliable operation. For this study, the three most important constraints are:

(1) Limited amount of silicon area.

(2) Limited allowable power dissipation.

(3) Limited number of pins for off-chip communications.

We will consider the implications of each of these constraints on the design of a VLSI communication component, and in particular, on the number and bandwidth of the communication links.

2.1.1. Area

Beyond a certain die size, the yield, i.e. the fraction of manufactured chips which function correctly, decreases dramatically with increased area [Glas78]. Current technology allows approximately 500,000 transistors to be placed on a single chip. It is projected that chips with 1,000,000 device will be possible by 1985 [Patt80]. It will be demonstrated in chapter 4 that this is more than adequate for the communication components described here, so limited amounts of silicon area do not severely constrain the design of the chip.

2.1.2. Power

The total amount of power generated by the chip must not exceed some upper bound determined by the power dissipation capacity of the integrated circuit package. Since the average power dissipation determines the amount of heat generated by the chip, violation of this constraint will result in a component which will overheat and fail during operation. We will assume that the amount of power dissipated by the chip varies linearly with the number of links, i.e. the total amount of power consumed by a p-port component is $(P_p \times p)+C_r$, where P_p is the average amount of power consumed by each port, and C_r is the power dissipated by circuitry which is not affected by the number of ports (e.g. portions of the control and routing circuitry). The power dissipated by this "link independent" circuitry is assumed to be constant; it thus reduces the total amount of power the port circuitry can dissipate, but does not enter into the tradeoff to be discussed.

If, for the moment, we neglect static power dissipation, then the power dissipated by the link circuitry is proportional to the clock rate [Carr72]. Doubling the number of links doubles the amount of circuitry, and thus the power dissipated by the chip. This can be offset by halving the clock rate, which in turn, halves the bandwidth of each communication link. Thus, increasing the number of links requires a proportional decrease in the bandwidth of each one.

Let us now consider static power dissipation. If it is assumed that the static power dissipation of each transistor remains constant as the clock rate is varied, then increasing the number of ports increases the number of transistors, which in turn increases *both* the static and dynamic power dissipation of the chip. However, reducing clock speed only reduces dynamic dissipation. Thus, increasing the number of ports really implies a more than proportional decrease in link speed. Therefore, the linearity assumption is biased to favor a large number of ports.

On the other hand, a slower clock rate implies that smaller transistors may be used, resulting in a reduction in static, as well as dynamic, power dissipation. If the clock rate is cut in half, the current driving capabilities of (say) an NMOS transistor may also be cut in half, which in turn halves the static power dissipation. In other words, both static and dynamic power dissipation are proportional to the clock rate. This is in agreement with the original model which only considered dynamic power dissipation, so link bandwidth is again a linear function of the number of links.

Therefore, when power dissipation is considered, the linearity assumption can only be biased to favor a large number of ports. In the analysis which follows, it will be seen that a small number of ports yields better performance under the linearity assumption. A more complex model which accounts for the bias will only add further support for this conclusion. Here, it will be assumed that link bandwidth is inversely proportional to the number of communication links if power restrictions constrain the design of the chip.

2.1.3. Pins

The number of interconnections to the chip's periphery is limited, and will increase much more slowly than the number of transistors per chip [Keye79]. Given N pins for p communication links, there are N/p pins per link. Bandwidth per link is thus proportional to N/p, assuming a constant bandwidth for each pin. Doubling the number of links halves the number of pins, and thus the total bandwidth, of each link. Thus, due to pin limitations, bandwidth per link also varies inversely with the number of links.

In the analysis presented above, it was assumed that all of the pins of each link are used for transmitting data. In a real implementation, some of the external connections may be used for control lines. These control lines represent an overhead which increases with the number of links. Doubling the number of links doubles the number of control lines, implying fewer pins are available for transmitting data. This results in a more than proportional decrease in link speed. A more precise model which includes control pins will lead to better performance for networks with a small number of ports, since the simplified model described above does not include this "per link" overhead. Again, the more precise model strengthens the conclusions which follow.

Finally, this model neglects the effects of data skew. In a traditional implementation of a parallel communication link, the receiver must wait for all of the arriving bits to reach a stable value before clocking the data. Due to possible variations in propagation delay along the different wires of the link, a parallel link must usually operate at a slower clock rate than the corresponding serial link, an effect not accounted for in the analysis presented above. These data skew problems can be alleviated by implementing the parallel link as a number of autonomous serial links, allowing the link to operate at the highest possible clock rate. This latter implementation leads to link speeds which are proportional to the number of pins per link, in accordance with the linear model presented above.

2.1.4. Summary of Constraints

A tradeoff exists between the number of links per chip and the speed of each link. If the chip design is constrained by either power or pin limitations, then doubling the number of links either halves the clock rate or halves the number of pins allocated to each one. In either case, the link bandwidth is halved. In effect, each chip has some total amount of I/O bandwidth which is equally divided among the existing communication links. This "constant bandwidth per chip" model will be used in all of the studies which follow.

In addition to its effect on link speed, the number of ports also affects the average hop count between two nodes in the network (e.g. a ternary tree could be used instead of a binary tree if one more port were available for each node). As the number of links on each chip is increased, the average hop count between pairs of nodes is reduced. The sections which follow present analytical and simulation results exploring this tradeoff between link speed and hop count.

2.2. Analytical Studies

In this section, the performance of networks constructed from p-port communication components is evaluated through analytical models. Average "end-to-end" delay and total network bandwidth are used as performance measures. The delay from point A to point B in a network is defined as the time which elapses from when the packet header begins to leave A to when the entire packet arrives at B. The "hop count" from A to B is defined as the length, i.e. the number of links, of the minimum length path from A to B. Network bandwidth is the amount of traffic the network can carry over some fixed time interval. A more precise definition for bandwidth will be given later.

In a real network carrying traffic generated by a parallel application program, average message delay may not be an appropriate performance measure. If a data value is generated in one processor long before it is used by another, then delays encountered by the message carrying this data do not affect the execution time of the program, so long as the data arrives before it is needed. However, since we cannot know a priori which message delays affect performance, *average* delays will be used. Also, averages are simpler to compute than other measures, e.g. maximum delay. A more detailed simulation study will be discussed in chapter 3 which uses execution time (actually, speedup) as the performance measure.

In order to evaluate the tradeoff between hop count and link bandwidth, two multicomputer network models are developed. In the first model, the implementation of a topology requiring b "branches" per node with p-port communication components ($p \leq b$) is considered. To achieve the necessary fanout, several components are interconnected to form a "cluster node" with b external branches. Each cluster node forms a single conceptual node of the desired topology. Delay and bandwidth are compared for various values of p. In general, a cluster node using components with a small number of ports will require more components than one using a larger number of ports. Thus, comparisons under this model neglect chip count.

A second analysis compares networks using the same number of components. In this model, the hop count/link bandwidth tradeoff is evaluated within individual classes of network topologies, such as trees or lattices.

In each case, a queueing model is used to evaluate network performance. The assumptions made by this model are outlined in the next section. Performance with and without a virtual cut-through mechanism is explored. Delay in a lightly loaded network and overall network bandwidth are computed and compared for the different approaches.

2.2.1. Assumptions

As discussed earlier, it is assumed that the bandwidth of each communication link is a linear function of the number of links on each chip. In the analysis which follows, a queueing model is used, and a number of other assumptions must be made:

(1) Message arrivals at different nodes are independent.

(2) Message arrival times have a Poisson distribution.

(3) Message lengths have an exponential distribution.

(4) Each node contains unlimited buffer space.

(5) Routing through each node is deterministic.

(6) Electrical propagation delays are negligible.

(7) Transmission error rates are negligible.

The first three assumptions are necessary to solve the queueing model. In particular, the first assumption, often referred to as the "independence assumption", states that "the exponential distribution [for message length] is used in generating a new length each time a message is received by a node ..." [Klei76]. This is clearly false since messages maintain their length as they pass through the network, but the effect of the assumption on the accuracy of message delay computations is negligible so long as the network does not contain long chains with no interfering traffic [Kerm79]. The assumption is a reasonable one for the networks examined here because the traffic loads used in these studies lead to output links which carry traffic arriving from several different input links, eliminating the long chains described above.

Similarly, the Poisson arrival time and the exponentially distributed message length assumptions (the latter implies exponential service times) allow the use of M/M/1 queues which can be easily solved. Relaxing each of these assumptions results in G/M/1 and M/G/1 queues respectively. If these queues are used however, Jackson's theorem [Jack57] cannot be applied, since the arrival times at each node no longer follow a Poisson distribution. The resulting queueing models are difficult to solve for the large, complex networks studied here. These assumptions are simplifications since traffic in the actual network need not be Poisson, and the networks considered here use fixed length packets, as will be discussed in chapter 4. Simulation studies will be discussed later which remove these restrictions. Further, a second approximate queueing model using M/G/1 queues will also be discussed [Klei76]. Here, the approximating assumption that Jackson's theorem still applies is made. It will be seen that although this second approximate model yields performance curves somewhat different from the first, the final conclusions drawn from the two models are identical.

The remaining assumptions listed above are appropriate for the networks examined here. The fourth assumption, unlimited buffer space, will be addressed in chapter 4. It will be seen that components with a limited number of buffers can achieve virtually the same performance as components with unlimited buffering capacity. The deterministic routing assumption is appropriate because packets traveling along the same virtual circuit follow the same path from source to destination. As discussed in chapter 4, this is necessary to ensure that packets sent on the same virtual circuit arrive in the order in which they were sent, thus avoiding much of the overhead associated with reassembling messages from their constituent packets. Since communication links are short, electrical propagation delays are negligible (a few nanoseconds at most) compared to the time required to transmit a single packet (hundreds or thousands of nanoseconds). Finally, the assumption concerning error rates is justified by the extremely low error rates measured in local communication networks [Shoc80]. Since the communication system described here is confined to an even smaller physical area than these local networks, it is less susceptible to noise in the operating environment, making this final assumption even more appropriate.

In addition to the "queueing model assumptions" described above, it is assumed that the internal structure of each cluster node is a balanced tree topology (a tree with minimal average path length between the root and leaf nodes [Knut73]). This minimizes the average hop count through the cluster node, as well as the number of components required to implement a node with a fixed number of branches.

Finally, in order to evaluate the performance of any communication network, traffic distribution assumptions, i.e. which processors send messages to which other processors and how frequently, must be made. These will be explained during the analysis as the need arises. In general, these assumptions are made to simplify the analysis. Simulations using a wide variety of traffic distributions are discussed in chapter 3.

2.2.2. Model I: Cluster Nodes

Consider the implementation of a network topology requiring b branches, i.e. communication links, for each node. Each node could be implemented with a single communication component requiring $b+1$ ports, assuming one port is used to connect to the computation processor attached to that node. Alternatively, each node could be implemented with a "cluster" of p-port communication components, where $3 \le p \le b$. As discussed earlier, it will be assumed that the components within each cluster node are interconnected by a balanced tree topology. Figure 2.1 for example, shows a node with 4 branches ($b=4$) implemented with 3-port communication components called "Y-components". This "cluster node" implementation implies a larger hop count between processors, however it also uses links of higher bandwidth, since fewer ports are required on each VLSI chip.

Adding a p-port component to an already existing cluster node adds $p-2$ branches. Since the one component cluster node has $p-1$ branches, an n component cluster node has $(p-1)+(p-2)(n-1)$ branches. Thus, a b-branch cluster node uses

$$n = ceiling\left[\frac{b-p+1}{p-2}+1\right] = ceiling\ \frac{b-1}{p-2}$$

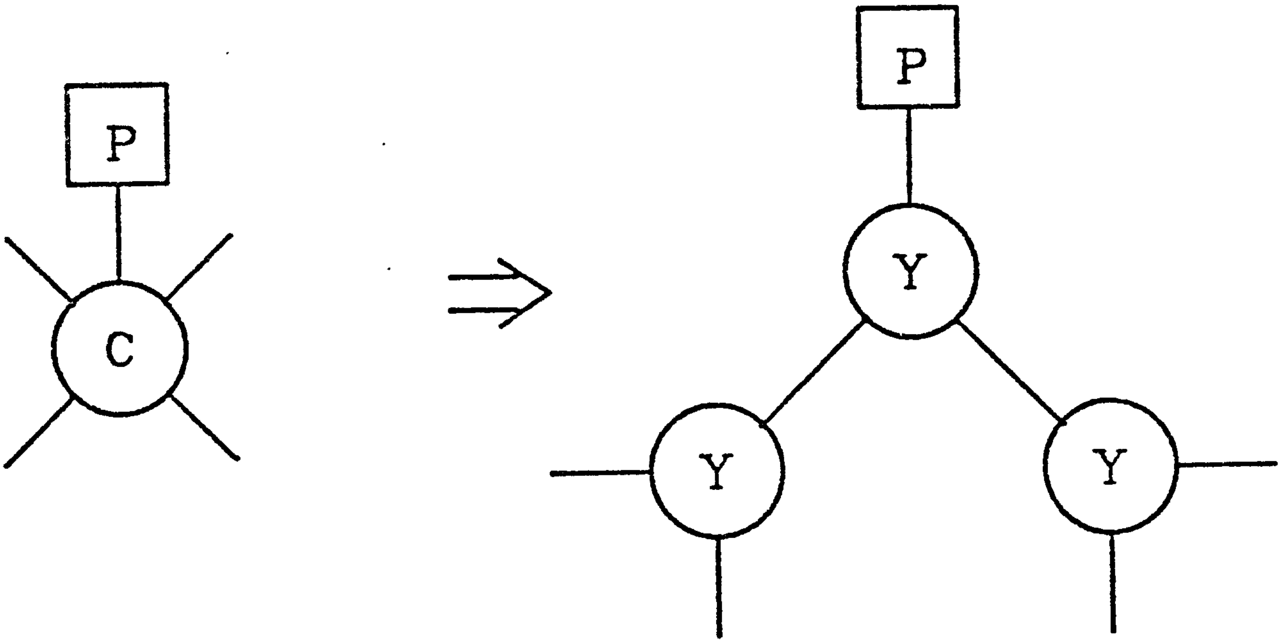


Figure 2.1. *4-branch node built from Y-components.*

components, where *ceiling* (x) is defined as the smallest integer greater than or equal to x.

2.2.2.1. Queueing Model

The queueing model presented in [Klei76] is used to evaluate the performance of a b-branch "cluster node". In order to evaluate these models, traffic distribution assumptions must be made. For the cluster node network, it is assumed that there are two virtual circuits between every pair of branches in the cluster node, one in each direction. In order to simplify the analysis, traffic to and from the processors attached to the cluster node will be ignored, and only traffic between branches will be considered. Since there are b branches, there are $b(b-1)$ virtual circuits through such a node. Assume that a traffic load of l messages per second exists on each of these virtual circuits, and each message consists of a single packet of data.

The average delay T through a store and forward communication network is defined as:

$$T = \sum_{i,j} \frac{\gamma_{ij}}{\gamma} Z_{ij}$$

where γ_{ij} is the average number of messages per second entering the virtual circuit from branch i to branch j, while γ is the total arrival rate on all virtual circuits. Z_{ij} is the average delay for messages along the virtual circuit from i to j. It is assumed that $\gamma_{ij} = 0$ if $i = j$. Since it is assumed that each of the $b(b-1)$ virtual circuits has the same external load, l messages per second, $\gamma = b\ (b-1)\ l$, and $\gamma_{ij} = l$. Thus,

$$T = \frac{1}{b(b-1)} \sum_{i,j} Z_{ij}. \qquad (1)$$

Let us now examine Z_{ij}.

Consider the path taken by the virtual circuit from branch X to branch Y, as shown in figure 2.2. The average delay Z_{xy} along this path is equal to

$$Z_{xy} = \sum_{i=1}^{n_l} T_i$$

where T_i is the average delay at link i. Assume links are numbered sequentially from 1 to n_l, as shown in figure 2.2. With cut-through,

$$T_i = \frac{m_l}{C(1-\rho_i)} - (1-\rho_{i+1})(t_m - t_h) \qquad (2)$$

as discussed below and in [Kerm79], where

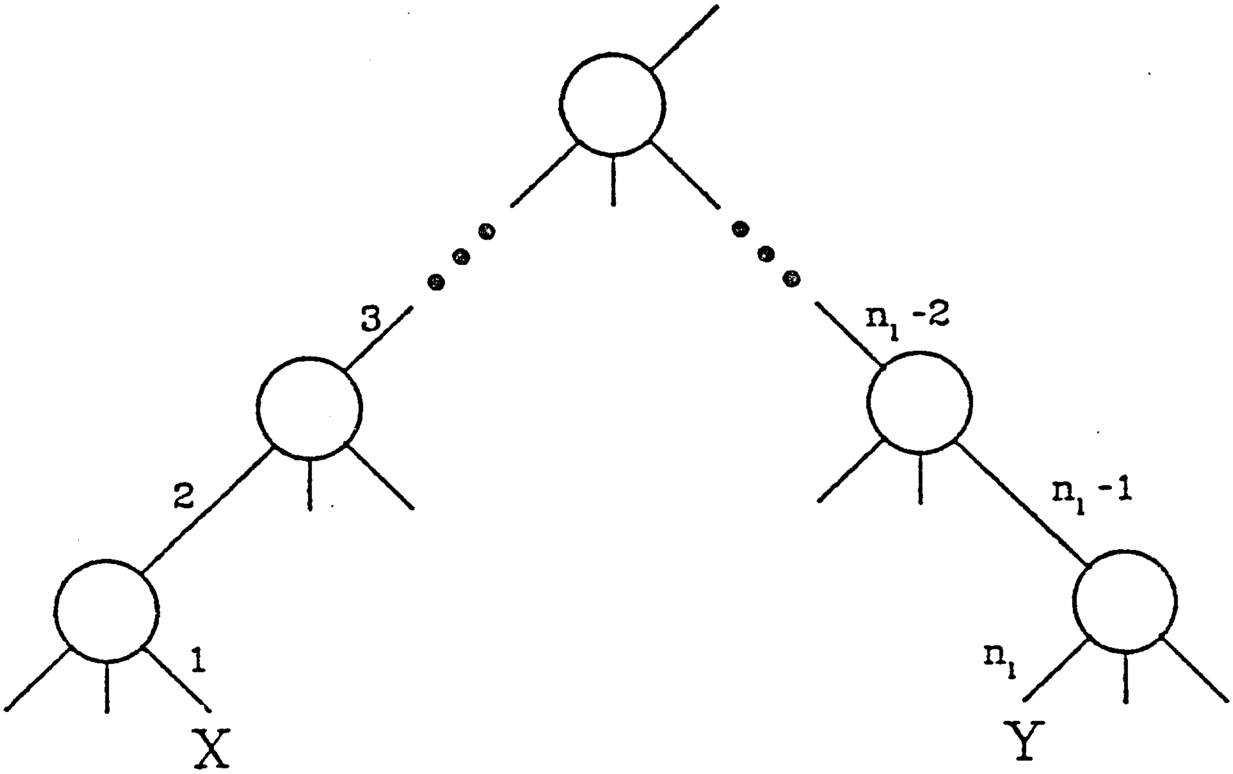


Figure 2.2. *Virtual circuit from X to Y.*

m_l = average message length
C = capacity (bandwidth) of the links
ρ_i = utilization of link i
t_h = time to transmit message header over the link
t_m = time to transmit message over the link

The message transmission time, t_m, includes the time to send both the data and header portions of the message. Assuming the total I/O bandwidth of each p-port component is B bits per second, C is equal to B/p. The first term of equation (2) is the solution of an M/M/1 queueing model, and represents the amount of time required to obtain and transmit a message over the link. The second term considers the effect of cut-through. $(1-\rho_{i+1})$ is the probability that a cut-through occurs, and $t_m - t_h$ is the amount of time "saved" by beginning to forward the message as soon as the header has arrived. It is assumed that no "partial" cut-throughs occur, i.e. forwarding begins either immediately after the header arrives or after the entire packet is received.

Thus the delay through the virtual circuit from X to Y is

$$Z_{xy} = \sum_{i=1}^{n_l} \left[\frac{p\, m_l}{B(1-\rho_i)} - (1-\rho_{i+1})\,(t_m - t_h) \right].$$

Z_{xy} measures the time from when the header of a message arrives at branch X to when the entire message has been forwarded over branch Y.

The total traffic load on link k is equal to the number of virtual circuits using the link, say v_k, times l, the load on each virtual circuit in messages per second. Thus, link utilization is

$$\begin{aligned} \rho_k &= \frac{m_l\, l\, v_k\, p}{B} && \text{if } k \le n_l \\ &= 1 && \text{if } k = n_l + 1 \end{aligned} \qquad (3)$$

Assigning ρ_{n_l+1} to 1 forces the $(1-\rho_{i+1})\,(t_m - t_h)$ portion of the last term in the summation for Z_{xy} to be zero. This is necessary to fulfill the definition for delay given above, i.e. the time which transpires from when the head of the message enters the cluster node until the time at which the *end* of the message leaves. The equation for Z_{xy} measures the time from when the head of the packet enters until the time at which the *head* begins to leave. Thus we must also add the time which elapses until the *end* of the packet leaves the cluster node. Setting ρ_{n_q+1} to 1 accomplishes this by in effect, eliminating the "saved time" resulting from cut-through in the final node.

Since v_k can be easily computed for each link of a given cluster node, the delay Z_{ij} for each virtual circuit can be found. Once Z_{ij} is known, equation (1) can be used to compute the average delay among all virtual circuits using the cluster node. Figure 2.3 shows the results of this computation for a 20-branch ($b = 20$) cluster node. The optimal number of ports as a function of b will be studied in a later section.

The various curves correspond to implementations that differ in two respects:

(1) the number of ports on each communication component

(2) whether or not a cut-through mechanism is used

The "without cut-through" curves are obtained by deleting the $(1-\rho_{i+1})\,(t_m - t_h)$ term in equation (2). Average delay is plotted in figure 2.3 as a function of the external load applied to each virtual circuit.

The computations assume that the average packet length m_l is 17 bytes, consisting of 16 data bytes and a one byte header. The total I/O bandwidth of each chip, B, is assumed to be 100 Mbits/chip-second, and is equally divided among the existing links. This latter value was chosen arbitrarily but does not affect the relative ordering of the curves. These numerical values will be used in all subsequent computations unless indicated otherwise. From figure 2.3, it is seen that network performance deteriorates as p is increased for this particular cluster node.

To a first order approximation, each of the curves in figure 2.3 can be represented by two performance measures:

(1) T^*, the delay in a lightly loaded network.

(2) l^*, the maximum traffic load the network can support.

T^* is the delay when l, the traffic load on each virtual circuit, is zero, and l^* is the asymptotic value for traffic load at which the delay approaches infinity. This latter quantity reflects the point at which some link(s) in the network approach 100% utilization, leading (mathematically) to queues which become infinitely long. In the real network, a flow control mechanism limits the actual queue size on each link, as will be discussed later. We will now examine delay and throughput in turn to determine the optimal number of ports for implementing

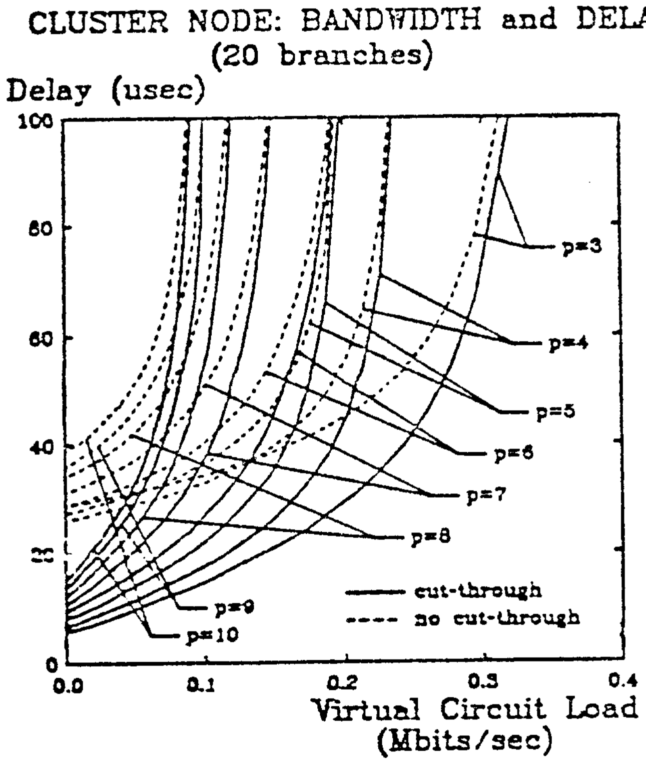


Figure 2.3. *Queueing delay for 20-branch cluster node.*

cluster nodes of any size.

2.2.2.2. Delay

T^*, the delay through a lightly loaded cluster node, is obtained by setting the traffic load, l, or equivalently the link utilization, ρ_i, equal to 0 (except if $i=n_l+1$, in which case $\rho_i=1$). Thus, from equation (2), the delay at each hop is

$$\begin{aligned} T_i^* &= \frac{p\, m_l}{B} - (t_m - t_h) && \text{if } i \le n_l \\ &= \frac{p\, m_l}{B} && \text{if } i = n_l + 1 \end{aligned}$$

when cut-through is used. A graph of T^* as the number of branches increases is shown in figure 2.4. It is seen that cluster nodes implemented using components with the minimum number of ports yield the smallest delay. The "bumps" in figure 2.4 occur when a new component is added to the cluster node as the number of branches is increased. This leads to a discontinuity in the average hop count, which in turn causes a discontinuity in the delay.

Without cut-through, the delay of each hop through a lightly loaded b-branch cluster node implemented with p-port communication components is simply $p\, m_l / B$. If $\bar{H}$ is the average number of hops through the cluster node, then $T^* = \bar{H}\, m_l\, p/B$. This function is also plotted in figure 2.4. It is seen that the optimal number of ports is again never larger than 4. The curves also demonstrate that virtual cut-through can significantly improve message delays.

Assuming the cluster node is implemented by a balanced (p-1)-ary tree, at most $2 \log_{p-1} b$ hops are required. Thus, the delay through a cluster node without cut-through is

$$T^* = \frac{2\, m_l\, p\, \log_{p-1} b}{B} .$$

Differentiating with respect to p and setting the result equal to 0 reveals that minimum delay is achieved with approximately 4.6 ports per component,

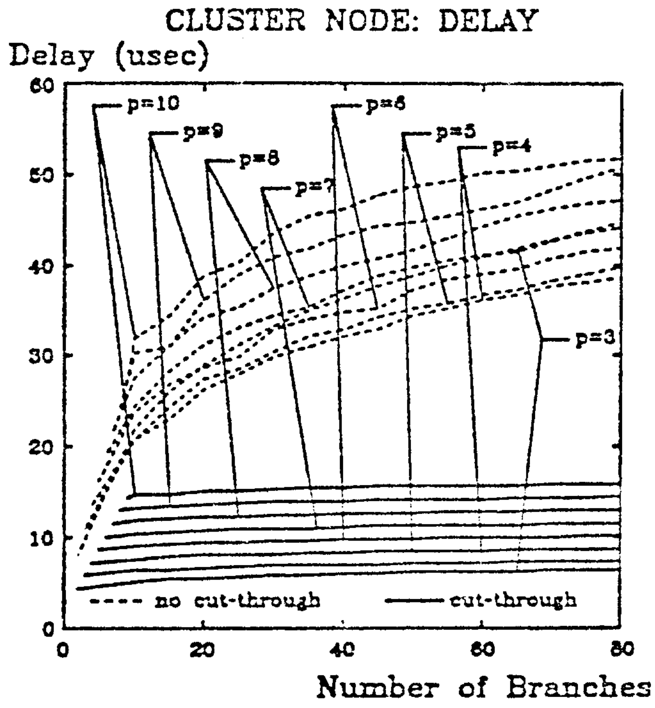


Figure 2.4. *Delay through cluster node under light traffic loads.*

agreeing with the curves in figure 2.4.

Thus we see that the optimal number of ports is relatively small when considering delay through lightly loaded networks. Delay through a lightly loaded cluster node is minimized when the number of ports is between 3 and 5.

2.2.2.3. Bandwidth

L^* is defined as the total network load when ρ_i approaches 1 on the most heavily utilized link in the cluster node. Since the links around the root of the clustèr node carry the most virtual circuits, they will saturate first. If the load on each virtual circuit at saturation is l^*, then L^* is $b(b-1)l^*$. If the most heavily utilized link has bandwidth B/p and carries v virtual circuits, then equation (3) suggests that saturation occurs at $l^*m_i = B/v\,p$ bits per second. Thus,

$$L^{*}m_{l} = \frac{b(b-1)B}{v\,p}$$

bits per second. A plot of $L^{*}m_{l}$ as a function of the number of branches is shown in figure 2.5a. The curves indicate that cluster nodes constructed with the minimum number of ports yield the most bandwidth.

The irregular behavior of the curves is an artifact of the manner in which branches are added to the cluster node, and does not represent a general behavior of communication networks. It is best explained by examining the individual components from which the curves are derived. For a given value of p, the behavior of L^{*} can be characterized qualitatively by the quantity $b(b-1)/v$, i.e. the number of virtual circuits using the cluster node divided by the number of circuits using the most heavily loaded link. Figures 2.5b and 2.5c show plots of these two quantities as a function of b. For clarity, only curves for p equal to 3, 4, and 5 are shown in figure 2.5c. The remaining curves demonstrate a similar behavior. It is seen that while the function $b(b-1)$ yields a smooth curve, the curve for v contains a number of discontinuities. These discontinuities give rise to the peaks in figure 2.5a.

The location of the discontinuities in figure 2.5c is a consequence of the manner in which components (i.e. branches) are added to the cluster node. The number of branches is increased by adding components "from left-to-right" at the leaves of the cluster node. Under this scheme, the most heavily utilized link is always be the "leftmost" link attached to the root of the cluster node (see figure 2.5d). The number of virtual circuits using this link, v, is simply $b_l(b-b_l)$, where b_l is the number of branches in the leftmost subtree of the root. If a new branch is added to the cluster node, one of two situations occurs:

(1) The branch is added to the leftmost subtree, causing both b and b_l to increase by 1, and v to increase by $(b - b_l)$.

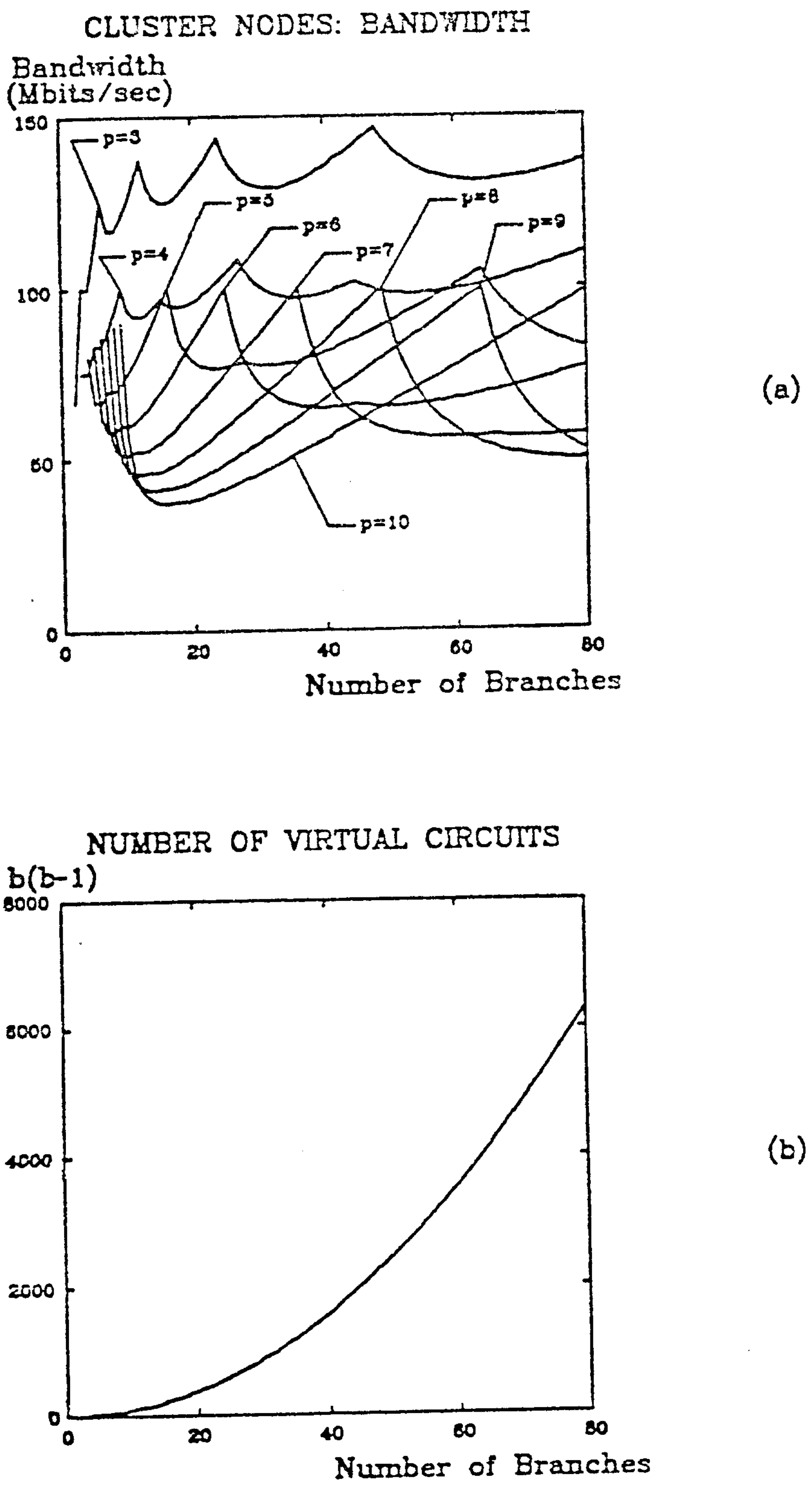


Figure 2.5. *(a) Bandwidth of cluster node. (b) Circuits in cluster node.*

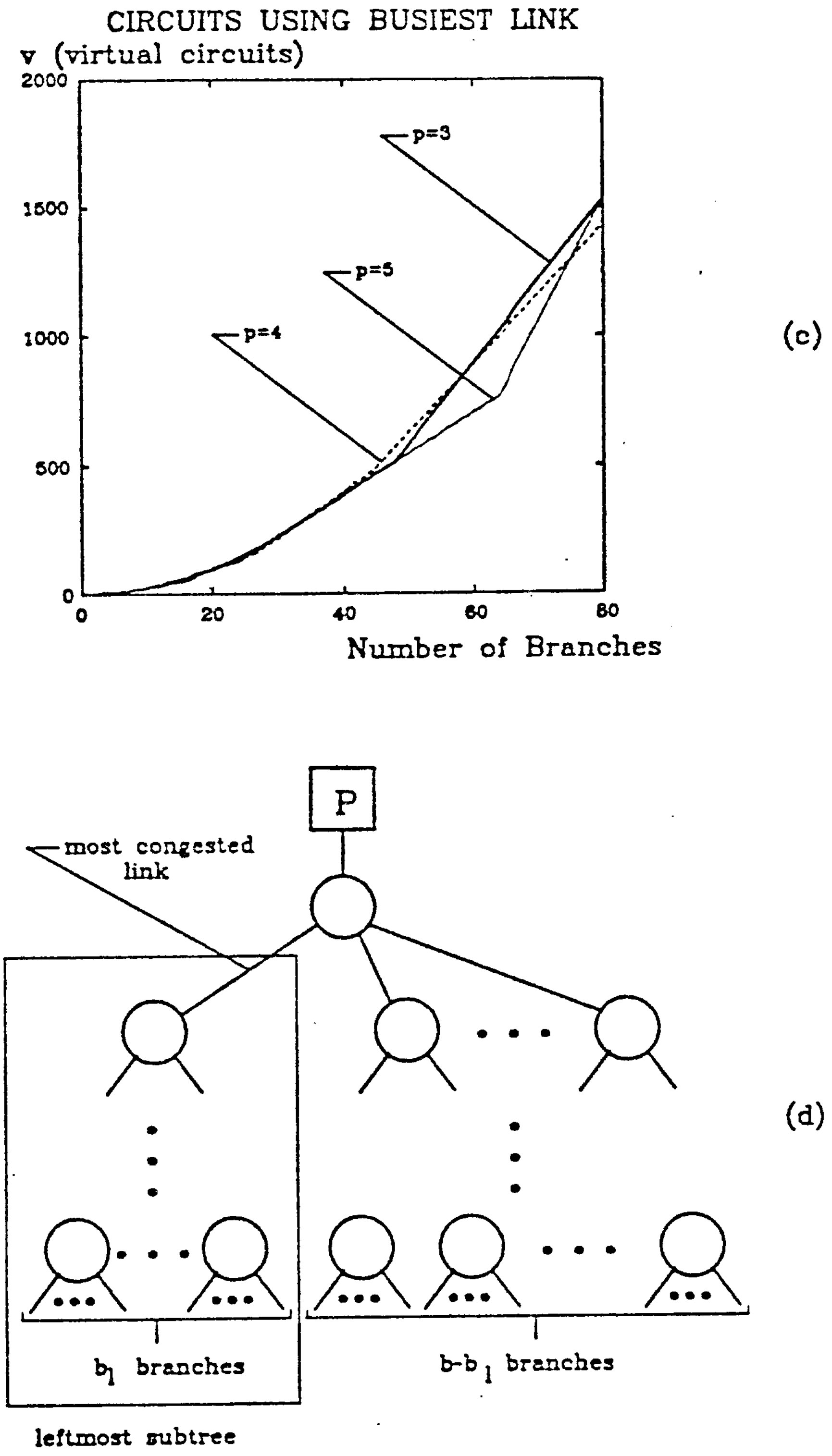


Figure 2.5. *(c) Most heavily loaded link. (d) Sample cluster node.*

(2) The branch is added somewhere other than the leftmost subtree, causing only b increases by 1, and v to increase by b_l.

As branches are added to the cluster node, discontinuities occur when the transition is made between these two situations, since the rate at which v is increasing suddenly changes. This transition occurs when the leftmost subtree becomes full, and when a new level is added to the cluster node. An exception to this rule occurs for p equal to 3, where adding a new level does *not* cause a discontinuity. This is a consequence of the symmetric nature of the binary tree. When a new level is added, the number of branches in the left subtree b_l is equal to $b - b_l$, the number in the right subtree, so the rate at which v is increasing remains the same, and the transition causes no discontinuity. Thus, in the binary tree, discontinuities occur only when the left subtree becomes full, and new branches begin filling the right subtree.

Each discontinuity results in a peak in the L^* curve. As b is increased, L^* increases if $b(b-1)$ is growing faster than v, but falls if v is growing faster. Each discontinuity represents a point at which the growth of v becomes accelerated, causing L^* to fall.

The curves in figure 2.5a indicate that the bandwidth provided by the cluster node does not increase significantly as the number of branches, and thus the number of components increases. This is due partially to the fact that the cluster node is implemented as a tree, and partially to the traffic model presented above. The traffic model assumes that there is a virtual circuit between every pair of branches, and that all of the virtual circuits are equally loaded. Thus, the I/O bandwidth of the root node limits the total bandwidth of the cluster node; increasing the number of components does not significantly increase the total bandwidth provided by the cluster node.

When the root node links become congested, most of the links of the cluster node, i.e. those near the leaves, are underutilized. Thus, virtual circuits which only use these links, i.e. circuits which do not go through the root node, can actually handle much more traffic. Let us consider the total bandwidth of the cluster node when traffic on these "underutilized" virtual circuits is allowed to increase. In particular, let us uniformly increase the traffic load on all virtual circuits which do not go through the root node until more links begin to saturate. The links "highest" in the cluster node tree will saturate first. Now repeat this process, i.e. increase the load on all virtual circuits which do not use saturated links, until all of the links are saturated. The total load on all of the virtual circuits gives the maximum traffic load the cluster node will support.

With the traffic load just described, it is clear that all of the links of the cluster node will be equally utilized. Such a network is said to be "balanced". The bandwidth of a balanced network is equal to the sum of the bandwidths of all of the communication links divided by the average hop count through the network. Intuitively, each link adds some fixed amount bandwidth to the network, and each virtual circuit uses bandwidth proportional to the number of hops it requires. Thus, this figure is indicative of the number of active (i.e. transmitting data) virtual circuits the network can support at one time, or alternatively, it is indicative of the total bandwidth allocated to a fixed set of virtual circuits. It will be seen later that this intuitive measure of bandwidth can also be derived from a queueing model for balanced networks.

A b-branch cluster node built from p-port communication components provides bandwidth (see section 2.2.2):

$$\frac{B \; ceiling \left\lceil \frac{b-1}{p-2} \right\rceil}{\overline{H}} \quad \text{for} \quad p \geq 3.$$

A graph of this measure of bandwidth for various values of p is shown in figure

2.6. Since a cluster node of n chips has a total link bandwidth which increases linearly with n, and the hop count increases only logarithmically in n (assuming a tree topology for the cluster node), one would expect the cluster node with the most chips to provide the most bandwidth. This corresponds to cluster nodes constructed with components using the minimum number of ports, or here, 3. The graphs confirm this intuitive result. Note that virtual cut-through does not impact the bandwidth provided by a network.

When constructing multicomputer systems with cluster nodes, congestion at the root can usually be alleviated through the use of an appropriate routing algorithm. For example, figure 2.7 shows a grid topology implemented with Y-components. An appropriate routing algorithm for this topology is to route packets along one direction, say north/south, and then the other, east/west,

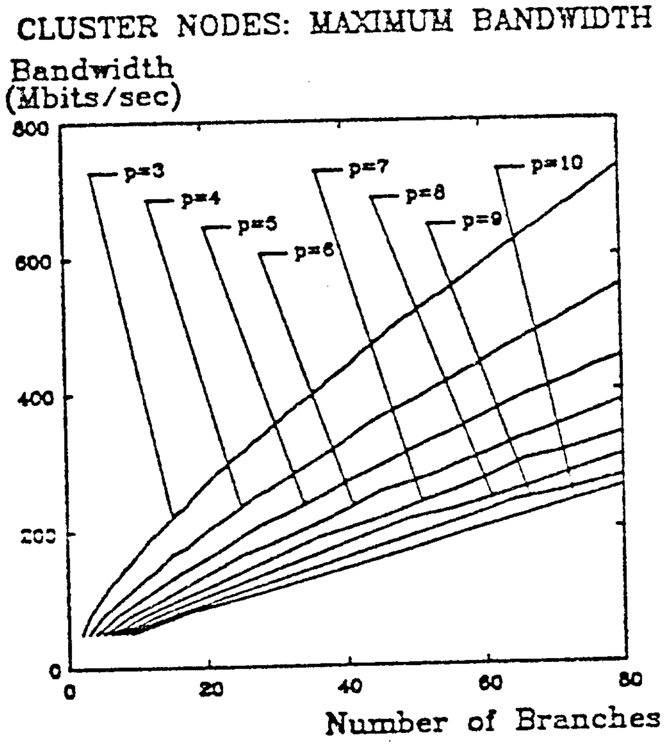


Figure 2.6. *Maximum bandwidth (#chips/hop count) of cluster node.*

using only one "90 degree turn". With this scheme, each packet travels through the root of a cluster node at most three times —at the source node, at the destination node, and at the node in which the 90 degree turn is made. In general, this type of behavior can be exploited for any topology (except trees where there is only one path between any pair of processors) by using a "global" shortest path routing algorithm through the network to increase usage of the shorter paths through the cluster node which do not go through the root.

Thus cluster nodes built with communication components with a small number of ports, say from 3 to 5, yield the least delay, and cluster nodes built with 3-port components yield the most bandwidth.

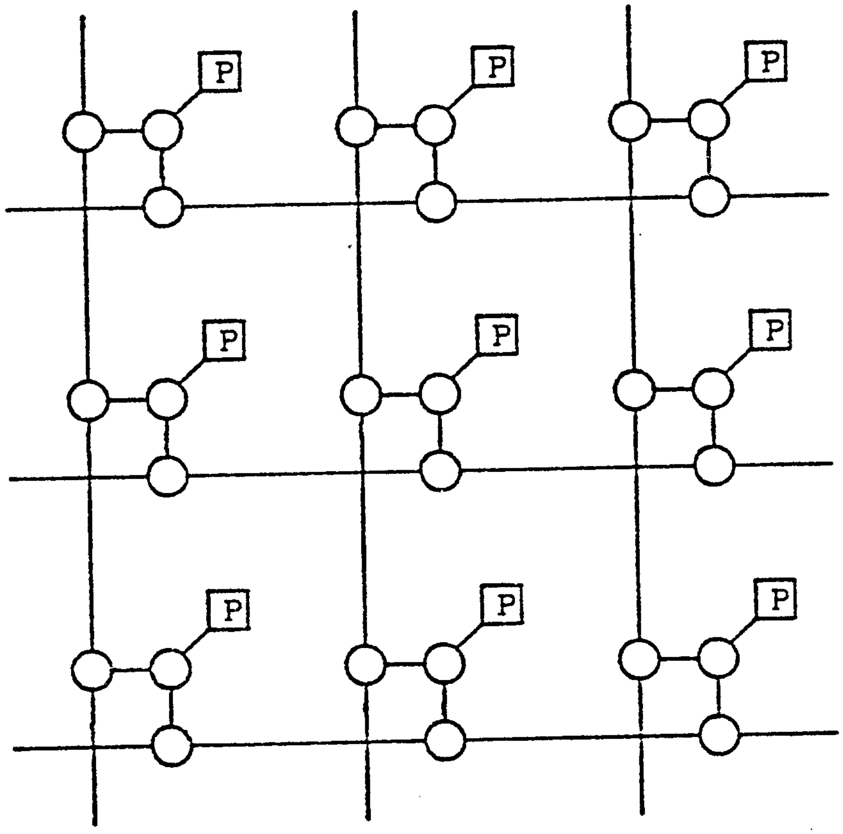


Figure 2.7. *Grid topology built with Y-components*

2.2.3. Model II: Networks with a Fixed Number of Components

The models in the previous section demonstrated that higher bandwidth and lower delays can be achieved by implementing b-branch cluster nodes with communication components using relatively few ports. Such networks require more chips than networks constructed from components with a larger number of ports. In this section, we explore the tradeoff between hop count and link bandwidth for networks with the same number of switching components.

Consider a large, unbounded network constructed from p-port communication components. As before, assume that each port has a bandwidth proportional to $1/p$. It will be assumed that there is one processor attached to each communication component in the network, using one of its p ports. Thus, in this model, p must be at least 4, since a 3-port component can only implement a ring topology.

Suppose that for some application, each node must communicate with all nodes within R hops of it. Assume that there are M such nodes. A small value for R, or equivalently M, indicates that traffic from each node is very localized, while a larger value indicates more global communications. Consider one specific node in the network, say X, and let us number the M nodes it sends messages to: 1,2, $\cdots$ M. The average distance (i. e. hop count) from X to these M nodes is

$$H = \frac{1}{M}\sum_{i=1}^{M} d_i$$

where d_i is the number of links traversed in the shortest path from X to i. Assume that traffic from X is uniformly distributed among the M nodes it communicates with. As before, increasing p will reduce the average distance, but at the cost of slower links. Conversely, reducing p implies faster links, but longer distances.

The average distance $\overline{H}$ is clearly dependent on the topology of the network. In general, more redundancy (i. e. distinct paths between pairs of nodes) implies a larger $\overline{H}$, assuming constant p. For the purposes of this section, different classes of network topologies will be characterized by a function, $m(i)$ $(i=1,2, \cdots ,k)$, with $M=\sum_{i=1}^{k} m(i)$ and $m(i)$ equal to the number of nodes whose minimum length path to node X is exactly i hops. The networks discussed here are assumed to be symmetric and unbounded. Since each node has $p-1$ ports for communicating with other nodes (one port leads to the processor attached to that node), $m(1) = p-1$. Two abstract cases will be discussed here:

$$\left.\begin{array}{ll} \textit{lattices}: & m(i) = m(i-1)+(p-1) \\ \textit{trees}: & m(i) = (p-2)\times m(i-1) \end{array}\right\} \qquad i=2,\ldots,k \text{ and } p\geq 4$$

The first represents regular two-dimensional lattices (see figure 2.8) and the second trees. Note that the latter case has no redundant links and thus gives minimal $\overline{H}$ for any topology with p ports per node. It is thus a favorable topology for components with a large number of ports, since networks using these components depend on small hop counts to overcome the handicap of having slower links. The lattice networks represent an alternative class of topologies with less favorable hop count averages, but redundant paths between pairs of nodes.

2.2.3.1. Queueing Model

The cut-through queueing model discussed earlier can also be applied to the networks presented in this section. The symmetric nature of the traffic load and the network topology leads to links which are equally loaded, i.e. the network is balanced. As before, we will consider only traffic within the network itself. Delays on the links between the processors and communication components are ignored.

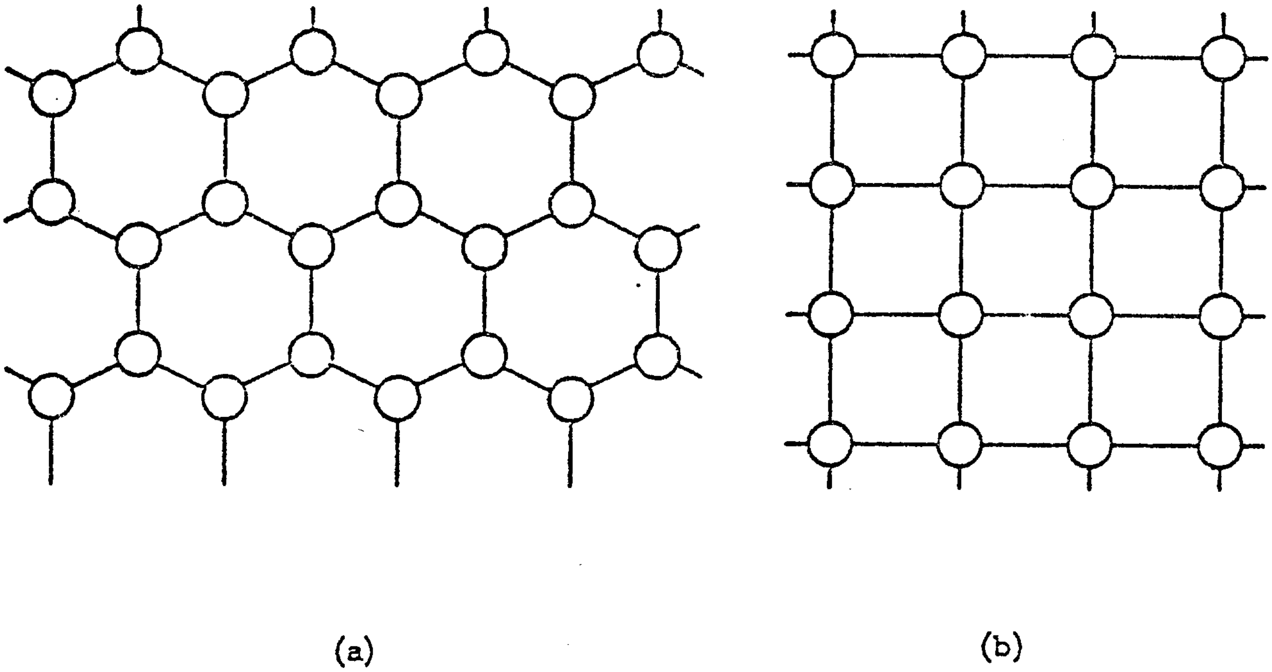


Figure 2.8. *Two regular two-dimensional lattices. (a) $p=4$. (b) $p=5$.*

A closed form solution for estimating network delay, including the effects of virtual cut-through, is known [Kerm79]. Using the same assumptions discussed in section 2.2.1, it can be shown that the average delay to send a message through a balanced network is

$$T = \frac{m_l \, p \, \overline{H}}{B\,(1-\rho)} - (\overline{H}-1)\,(1-\rho)\,(t_m - t_h) \qquad (4)$$

where:

$\overline{H}$ = average hop count
m_l = average message length
B = total I/O bandwidth of each communication component
p = number of ports
ρ = utilization of each link
t_h = time to transmit message header over the link
t_m = time to transmit message over the link

The first term of this equation is the delay when no cut-through is used. The second term is the improvement when cut-through is added. The effectiveness of cut-through in reducing delay increases with $\overline{H}$ because there are more chances for cut-through to occur if the number of hops required is large. The dependence on ρ arises from the fact that the probability that the outgoing link is free, i.e. the probability that a cut-through will occur, depends on how heavily the link is utilized. As before, the model assumes that no "partial" cut-throughs occur, i.e. forwarding begins either immediately after the header arrives or after the entire packet is received. The cut-through mechanism has greater impact in lightly loaded networks (small ρ).

Consider a network with N processors (and thus N communication components), with each processor sending messages to the M processors closest to it. If $\overline{H}$ is the average hop count to reach another processor, then there are $N \times M$ virtual circuits, each using $\overline{H}$ links. Assume the load on each virtual circuit is l messages per second, or $l\ m_l$ bits per second. Since the network has $N \times (p-1)$ links (excluding the one connecting to the processor), the average load on each link is $N\ M\ m_l\ l\ \overline{H} / N(p-1)$ bits per second. Therefore,

$$\rho = \frac{M\ m_l\ l\ \overline{H}}{B} \frac{p}{p-1}. \tag{5}$$

Since $\overline{H}$ can be computed numerically, given M and p, we can use equations (4) and (5) to compute message delays.

Figure 2.9a shows delay in lattice topologies with and without a cut-through mechanism as a function of the load applied to each virtual circuit. Table 2.1 lists the number of virtual circuits using each link. M is fixed at 50 nodes. The optimal number of ports as a function of M will be studied in a later section. Under light traffic loads, networks with a smaller number of ports achieve lower delays, regardless of whether or not a cut-through mechanism is used. Figure 2.9a indicates however, that the "knee" for curves with a large number of ports is further to the right than that of those with a small number of ports. This indicates that networks with a large number of ports can maintain reasonable delays for larger traffic loads than networks with a small number of ports. In other words, these curves indicate that components with a small number of links yield networks with shorter delay, but less overall bandwidth.

Table 2.1.
Link Usage

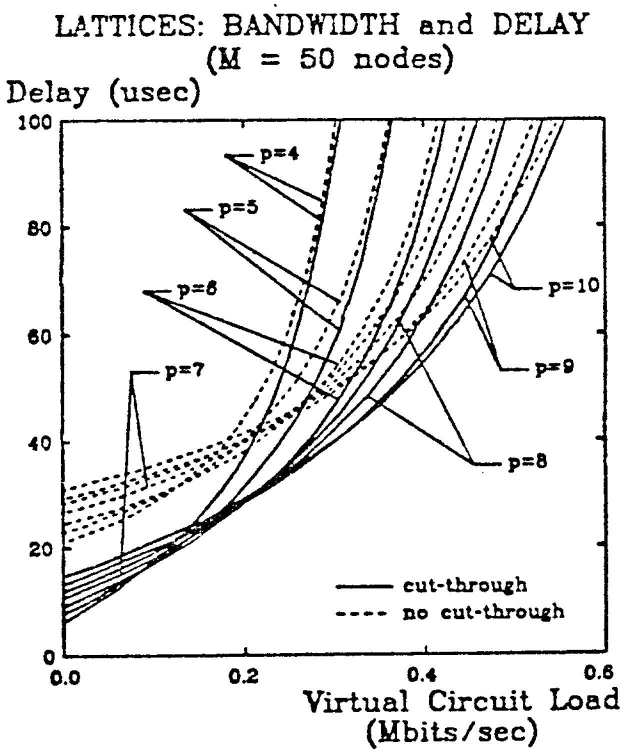
p	Link Bandwidth (Mbits/sec)	Circuits per Link (Lattices)	Circuits per Link (Trees)
4	25.00	65.00	57.33
5	20.00	42.50	32.50
6	16.67	30.00	24.00
7	14.29	23.33	18.00
8	12.50	18.57	13.43
9	11.11	15.00	11.50
10	10.00	12.67	10.11

Figure 2.9b and table 2.1 present the same analysis for tree topology networks, also with M fixed at 50 nodes. Again, networks with a small number of ports yield better delay under light traffic loads, but poorer overall bandwidth. The minimum number of ports achieves the least delay when a cut-through mechanism is used, as would be expected since cut-through diminishes the penalties of traversing extra hops. Networks without cut-through achieve minimal delay when 5 ports are used, for this particular value of M.

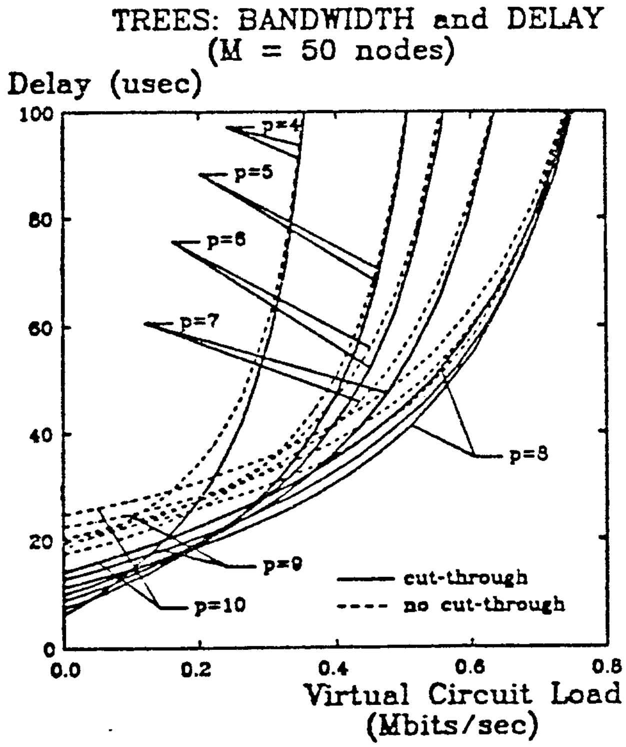
We will now analyze the optimal number of ports as a function of traffic locality, or here, M. As before, T^*, the delay in a lightly loaded network, and l^*, the maximum virtual circuit traffic load supported by the network, will be evaluated and compared.

2.2.3.2. Delay

T^*, the delay in a lightly loaded network is again found by setting ρ equal to 0. Thus, from equation (4), one obtains:



(a)



(b)

Figure 2.9. *Queueing delay, M=50. (a) lattices. (b) trees.*

$$T^{\bullet} = \frac{m_t \, p \, \overline{H}}{B} - (\overline{H}-1)(t_m - t_h) \qquad \text{with cut-through}$$

$$T^{\bullet} = \frac{m_t \, p \, \overline{H}}{B} \qquad \text{without cut-through}$$

These quantities are plotted in figure 2.10 as a function of M, the number of processors to which each processor sends messages (which determines $\overline{H}$). When cut-through is used, it is seen that networks constructed with the smallest number of ports yield the least delay for both lattice and tree topologies. The same is true for lattices without cut-through, indicating that the reduction in hop count caused by increasing the number of ports is not enough to adequately offset the lost bandwidth per port. The final case, tree topologies without cut-through, is somewhat more complex.

In tree topologies without cut-through (figure 2.10b) it is seen that the smallest number of ports ($p=4$) does not give minimum delay beyond $M=32$ nodes. Similarly, as M is increased further, larger values of p appear more attractive (see figure 2.11), although the optimal number never rises beyond 6. Given some value of M, the growth of $m(i)$ (as i increases) determines the average hop count, $\overline{H}$. The faster $m(i)$ grows, the smaller $\overline{H}$ becomes. In tree networks, $m(i)$ is an exponential function of p, implying its growth will be accelerated substantially if p is increased. This acceleration is so substantial that, to a certain extent, the associated reduction in hop count effectively offsets the bandwidth loss which results when more ports are used.

For both classes of networks, these results favor a communication component with relatively few ports, say from 4 to 6. A cut-through mechanism makes the optimal number closer to 4. Under the conditions stated above, tree topologies will always yield lower delays than lattices because of lower hop count averages. This in turn results from the lack of redundant paths in tree topologies and is in agreement with results already discovered by other researchers

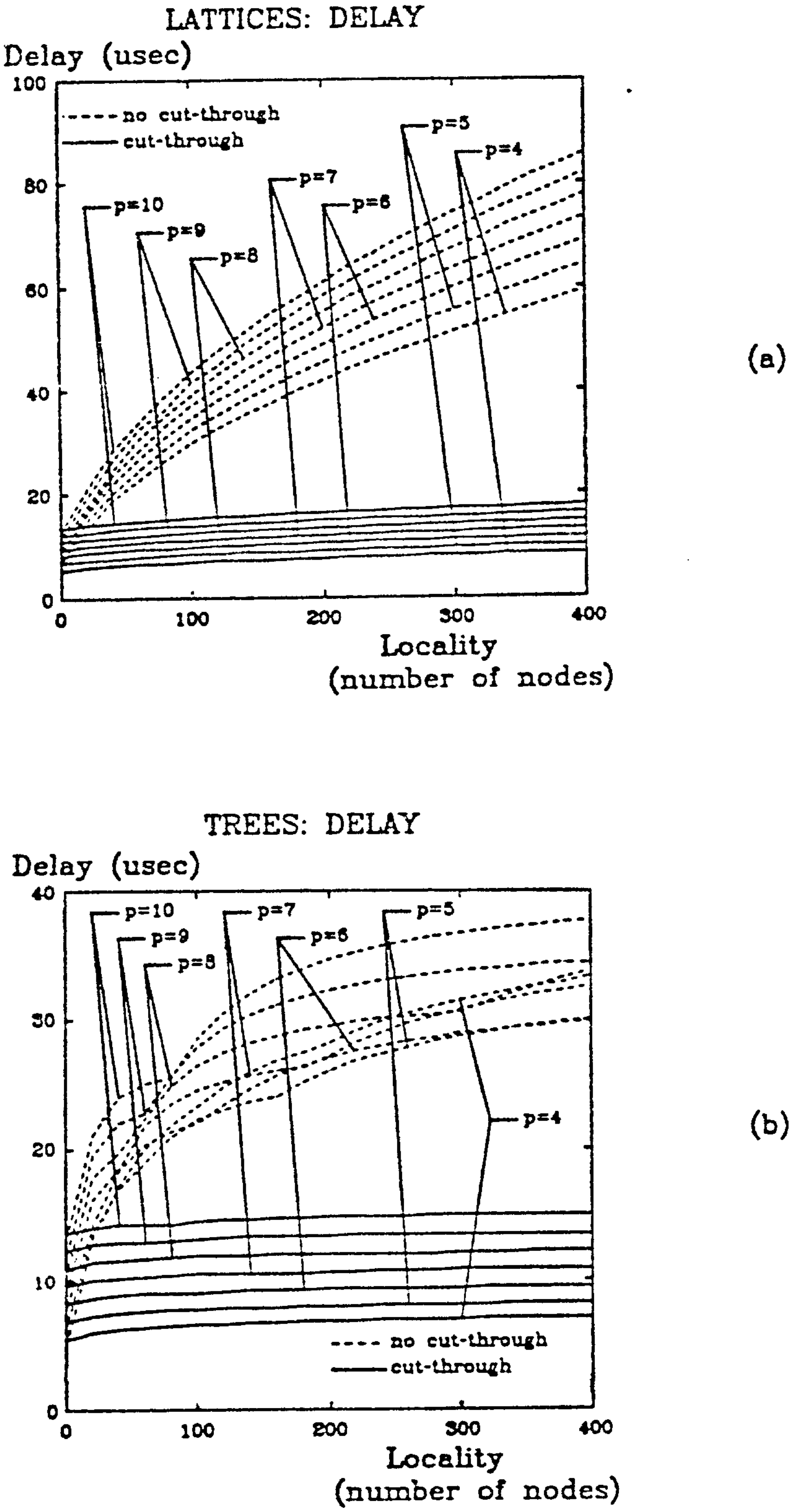


Figure 2.10. *Delay under light traffic loads. (a) lattices. (b) trees.*

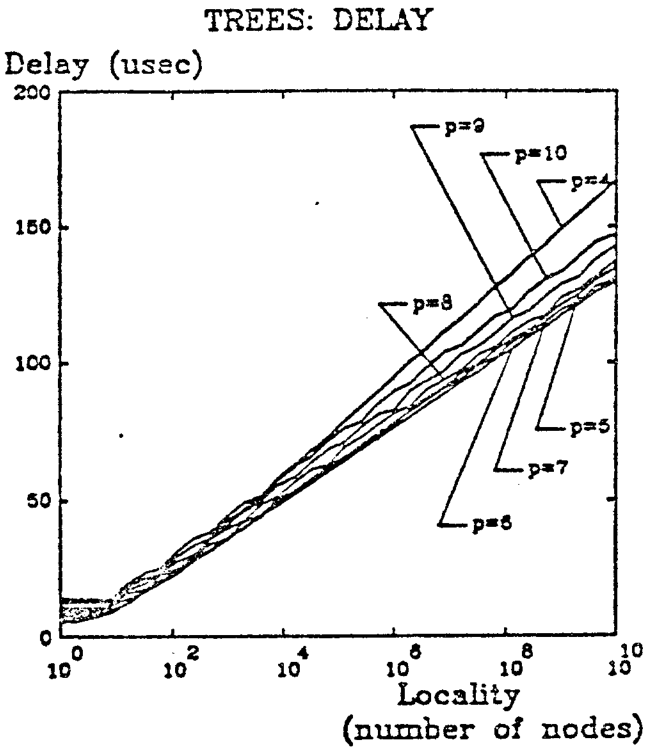


Figure 2.11. *Delay under light traffic loads, trees (no cut-through).*

[Desp78]. Asymptotic values of $m_i\, p\, \overline{H}/B$ as a function of M will now be derived, in order to determine an optimum number of ports when no cut-through mechanism is provided.

Given such an abstract topology, we can treat M as a function of the continuous variable r, the distance of a node from the other nodes to which it is sending messages, (previously, $m(i)$ was a function of the discrete variable i). As M grows toward infinity, $m(r)$ is asymptotically equivalent to $m(i)$. With this perspective, $\overline{H}$ is no longer a sum, but rather an integral. Thus we have

$$\overline{H} = \frac{1}{M}\int_0^R r\, m(r)dr \qquad \textit{with} \qquad M = \int_0^R m(r)dr$$

And the two cases disussed above reduce to:

$$\left.\begin{array}{ll} \textit{lattices}: & m(r) = (p-1)r \\ \textit{trees}: & m(r) = (p-1)(p-2)^{r-1} \end{array}\right\} \qquad p \geq 4$$

Evaluation of the above integrals for the two cases results in the following equations for delay:

lattices: $$T^* = \frac{m_l \overline{H} p}{B} = \frac{m_l}{B}\sqrt{\frac{8(p-1)M}{9}}$$

trees: $$T^* = \frac{m_l \overline{H} p}{B} = \frac{m_l}{B}\left[\frac{p\,R(p-2)^R}{(p-2)^R-1} - \frac{p}{\ln(p-2)}\right]$$

with $$R = \frac{\ln[M(p-2)\ln(p-2)+p-1] - \ln(p-1)}{\ln(p-2)}$$

The equation for the first case again demonstrates that for any given M, a lower delay results if fewer ports are used. The equation for the second case however, requires a more detailed analysis.

Minimizing $\overline{H}p$ by taking the derivative with respect to p, and solving this equation numerically yields the curve in figure 2.12. This curve gives the optimal number of ports (optimal in that it minimizes the average delay) as a function of M, the number of nodes communicated with.

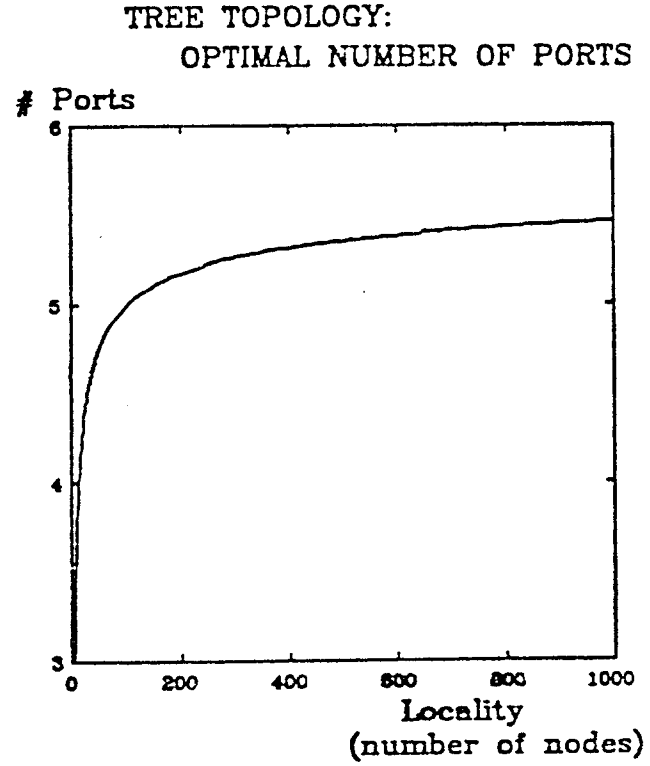


Figure 2.12. *Optimal number of ports, tree topologies (no cut-through).*

The above derivations assume that traffic from a node is uniformly distributed among the nodes it communicates with. In practice, one would try to map a specific problem onto the multicomputer in such a way that there is more traffic with nearby nodes than with those which are further away. Traffic between neighboring nodes should then be weighted more heavily. If one takes this into account, the case for the use of a few high-bandwidth links rather than many slower links becomes even stronger.

Thus, based on these studies, it appears that a communication component with relatively few ports, say from 4 to 6, is the most desirable. If cut-through is considered, the argument for a small number of ports also becomes stronger.

2.2.3.3. Bandwidth

Let us consider increasing the load on all virtual circuits of the network. As before, network bandwidth is defined as the asymptotic traffic load supported by the network as it approaches saturation, i.e. as link utilization ρ approaches 1. From equation (5), the load per virtual circuit at saturation l^* is

$$l^* = \frac{B}{m_l M\bar{H}} \frac{p-1}{p}$$

messages per second. The total network bandwidth at saturation is

$$L^* m_l = \frac{BN}{\bar{H}} \frac{p-1}{p}$$

bits per second, since there are $M \times N$ virtual circuits, where N is the number of nodes in the network. Thus, neglecting the $(p-1)/p$ term, the total bandwidth of a network is approximated by the sum of the link bandwidths divided by the average hop count, agreeing with the maximum bandwidth figure of merit derived intuitively for the cluster node model. This figure is indicative of the maximum number of active virtual circuits the network can support at one time.

When comparing networks with the same number of chips, the sum of the link bandwidths is constant, so the topology with the smallest average hop count will achieve the highest bandwidth. Thus, in this case, networks constructed with the largest number of links per node yield the most bandwidth. Bandwidths for tree and lattice networks are shown in figure 2.13 for various values of p, confirming this intuitive result. The curves also demonstrate how rapidly network bandwidth diminishes as traffic becomes less localized.

2.2.4. M/G/1 Queueing Models

The queueing models presented thus far have assumed that message lengths are exponentially distributed. This allows one to use M/M/1 queueing models which can be easily solved. Since the networks described here use fixed length packets, an M/G/1 model is more appropriate. Unfortunately, the exact solution of complex networks of M/G/1 queues is unknown, since Jackson's theorem can no longer be applied. Briefly, Jackson's theorem allows one to solve a network of queues with Markov arrival rates by examining each queue independently, isolated from the rest of the network. Fixed length packets imply non-exponential service times which leads to non-Markovian behavior.

An alternative approach to resolving this dilemma is to use M/G/1 queues, but to make the approximating assumption that Jackson's theorem can still be applied. Other studies have indicated good correspondence between this model and simulation results [Klei76]. The Pollaczek-Khinchin mean value formula indicates that replacing an M/M/1 queue with one using fixed service times reduces the waiting time (i.e. the time spent waiting for the link to become free) by a factor of two [Klei75]. In the analysis presented thus far, this implies that equation (2) for the cluster node model becomes

$$T_i = \frac{1}{2}\left[\frac{m_i}{C\,(1-\rho_i)} + \frac{m_i}{C}\right] - (1-\rho_{i+1})\,(t_m - t_h)$$

while equation (4) for the second model becomes

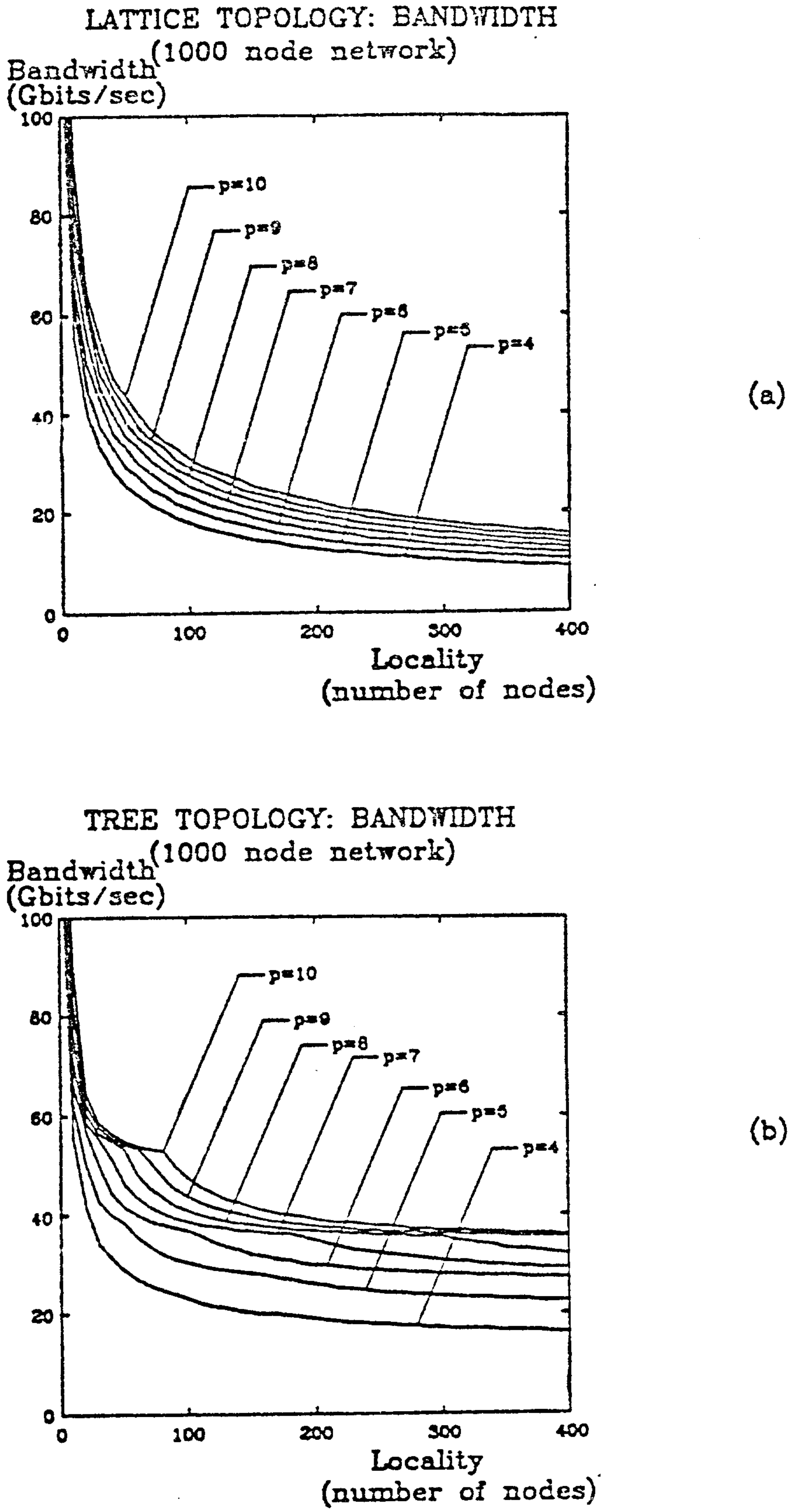


Figure 2.13. *Bandwidth (#chips/hop count) (a) lattices (b) trees.*

$$T = \frac{\overline{H}}{2}\left[\frac{m_l\,\rho}{B\,(1-\rho)} + \frac{m_l\,p}{B}\right] - (\overline{H}-1)\,(1-\rho)\,(t_m - t_h)\,.$$

Closer examination of these equations reveals however, that delay in lightly loaded networks (delay as ρ approaches 0) and network bandwidth (traffic load as ρ approaches 1) are identical to that in the M/M/1 model. Thus, the M/G/1 queueing models yield curves with the same relative orderings as those derived for the M/M/1 models.

2.2.5. Summary of Analytic Results

The analytic results for the optimal number of ports are summarized in table 2.2 below.

Table 2.2.
Optimal Number of Ports

model	delay	bandwidth
cluster nodes	small	small
fixed number of components	small	large

When considering delay, all of the analytical models presented here indicate that better performance is achieved with communication components with a relatively small number of ports, say from 3 to 6. Virtual cut-through reduces the impact of larger hop counts, and thus pushes the optimal number of ports closer to 3. It is seen that a cut-through mechanism can substantially reduce transmission delays in the network, so it is unreasonable to exclude it from any communication component design.

When considering bandwidth, the cluster node model favors components with the minimum number of ports, while the "one communication component per processor" model favors a large number of links. It is important to realize however, that the overall bandwidth of a network can be increased by adding more chips, since the sum of the link bandwidths grows faster than average hop count in most topologies (rings, which are generally considered to be unsuitable for networks with a large numbers of processors, are an exception). This is verified by the cluster node model, where networks constructed from components with a small number of ports achieved greater bandwidth. Thus, achieving low latency appears to be the more important problem, leading to further support of communication components with a small number of ports.

It is important to remember that these bandwidth studies measure maximum network bandwidth, and thus only consider performance under heavy traffic loads. In a lightly loaded network, the bandwidth available to individual virtual circuits is equal to the bandwidth of the communication links it uses and thus will be larger if components with a small number of ports are used. Therefore, when combined with the analytic results presented in this section, one must conclude that providing general purpose communication components with a small number of ports, say from 3 to 5, is the best choice.

The analysis presented above made a number of simplifying assumptions. The strongest assumption concerned the traffic distributions among processors. Simulation studies which explore a number of different traffic distributions will be discussed next. It will be seen that for the most part, these simulations support the conclusions derived analytically. When discrepancies do occur, the simulations indicate better performance for components with a small number of ports, thus strengthening the conclusion that a small number of ports should be used.

CHAPTER THREE

SIMULATION STUDIES

The analytical models presented earlier made some simplifying assumptions. In particular, traffic distributions were assumed to be such that links are equally utilized, message arrivals were assumed to follow a Poisson distribution, and message lengths were assumed to follow an exponential distribution. To evaluate the conclusions derived by the analytical models when these assumptions are relaxed, and to gain deeper insight into the tradeoffs between various network topologies and realizations of the communication components, a simulation program was developed. The results of these simulation studies are discussed in this section. An instruction level simulator called Simon is described, and the respective speedups resulting from executing several parallel application programs on various network structures are reported.

The first two sections describe the simulator and the assumptions made about the multicomputer system. Following this, the application programs are described, and simulation results are presented. Some of the issues evaluated by this study include the optimal number of ports and the effect of incorporating a mechanism in the communication hardware for efficiently handling multiple-destination messages.

3.1. The Simulator: Simon

Simon (Simulator of Multicomputer Networks) is a discrete-time, event-driven simulation program designed to facilitate comparison of alternate switching structures [Fuji83]. The most important features of Simon are:

(1) Traffic in the communication domain is generated by application programs executing some parallel algorithm. This is in contrast to the analytical studies which made the simplifying assumptions that links are equally loaded and message arrivals follow a Poisson distribution.

(2) The software modeling the interconnection network is contained in a separate module called the "switch model", allowing easy comparison of different switching structures.

The simulator consists of three components (see figure 3.1): the application program, the simulator base, and the switch model. The application program consists of a number of tasks, or equivalently, processes, which execute in parallel and communicate by exchanging messages. The simulator base time-multiplexes execution of the tasks on the host computer, in this case a VAX-11/780. The base also keeps track of time for each task (each task has a clock which advances as the task executes) to ensure that interactions among tasks (e.g. message transmissions) are simulated in the proper time sequence. Finally, the switch model provides a fixed virtual circuit interface for the tasks and simulates message passing between processors. A detailed description of the simulator is given in [Fuji83].

3.2. Assumptions

A number of assumptions are made in the simulation experiments reported here. These include:

(1) negligible operating system overhead

(2) VAX 11/780 processing elements

(3) one-to-one mapping of tasks to processors

(4) fixed length packets (1 byte header, 16 data bytes)

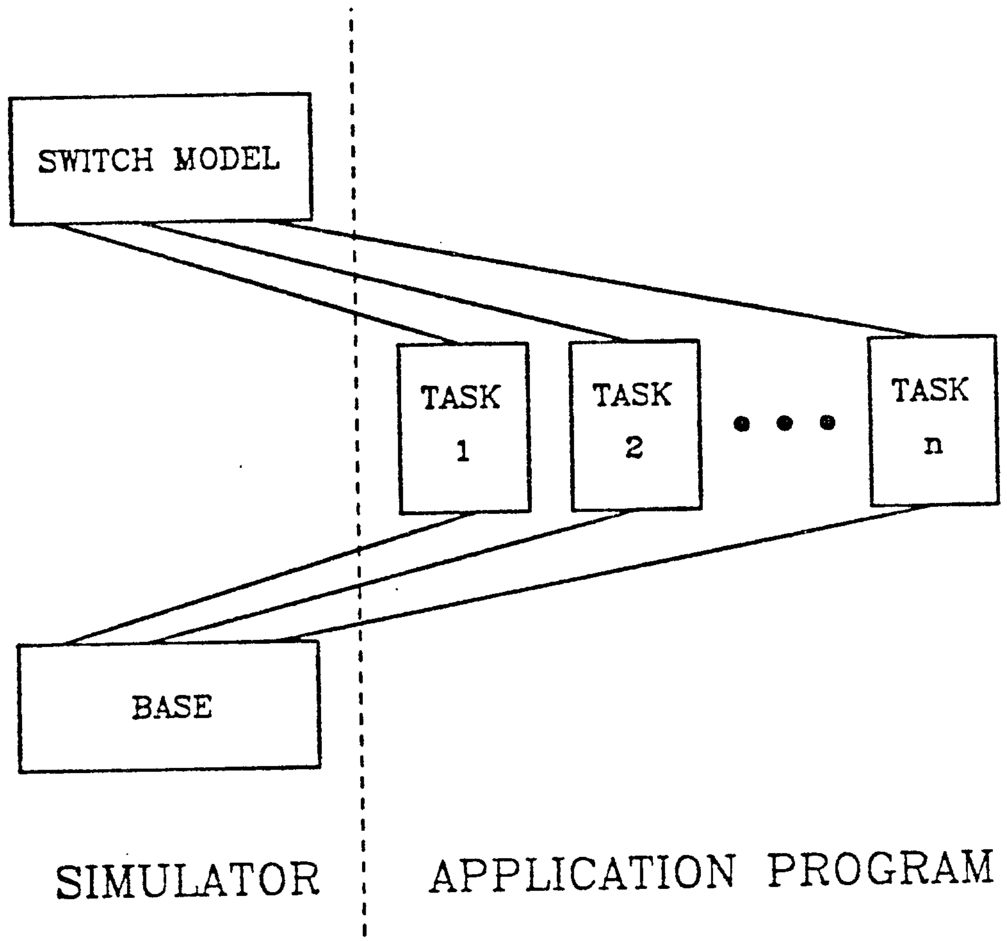


Figure 3.1. *Block diagram of simulator.*

(5) unlimited buffering within each communication component

(6) error free transmission

(7) virtual cut-through

(8) virtual circuits set up in advance

(9) shortest path routing

Each of these assumptions will now be discussed in turn.

(1) The bulk of the simulation studies assume that the time to execute an operating system routine for invoking a communication mechanism (e.g.

sending a message) is negligible. This allows separation of the penalty due to operating system overhead from that inherent in the communication switch. Studies which analyze the impact of operating systems overhead alone, i.e. which assume negligible communication delays, will also be discussed.

(2) The speed of the processing elements is fixed throughout the simulations to that of a VAX-11/780. By the mid 1980's, 32-bit microprocessors will have achieved this performance level [Patt82]. However, the absolute speed of the processors is not so important as the ratio of processor speed to communication delays, since this affects the fraction of time each processor spends performing computations relative to the time required for communications. As technology improves, the computational speedup due to the use of multiple processors is unchanged if this ratio remains the same, since both uniprocessor and multicomputer execution times decrease by the same factor. If however, processor speed increases at a faster rate than communication speed, the ratio changes. Communication delays will prevent the multicomputer execution time from decreasing in proportion to that of the uniprocessor, and speedup actually decreases. Here, since the "VAX" assumption implies a constant processor speed, this ratio is changed by varying communication bandwidth. This provides the flexibility of determining performance under 1983 technology, as well as predicting the effect of technological changes.

(3) It is assumed that each task executes on a separate processor. In other words, it is assumed that the system contains enough processors to accommodate the application program. The programs studied here use at most 32 processors, so this is a reasonable assumption. Indeed, general purpose systems using more than 32 processors have already been constructed

[Swan77a, Stri83, Kush82, Hosh83].

(4) Packets consist of a single control byte followed by 16 data bytes. Fixed-size packets are used because of the difficulties associated with managing variable sized buffers, as discussed in chapter 4. This is in contrast to the analytic models described in chapter 2 which made the simplifying assumption that message lengths follow an exponential distribution. The control byte is used to specify a virtual channel number, as will be discussed in chapter 4. In the application programs discussed here, messages are short, typically consisting of only a single floating point number, and fit within a single packet. An area of future research is to consider workloads which include large, multi-packet messages, e.g. paging traffic and/or file transfers.

(5) It is assumed that adequate buffer space is available in each component for holding packets waiting to be forwarded. It will be shown later that chip densities now allow each component to provide enough buffer space to achieve approximately the same performance as a component with an unlimited amount of buffering.

(6) The simulator assumes that no errors occur during data transmissions. This assumption was also used in the analytical models, and was justified in the discussion there.

(7) Virtual cut-through is used in all networks. Partial cut-throughs are allowed, i.e. if the outgoing link used by a packet is busy when the header arrives, but becomes free before the tail arrives, the packet need not wait for the latter event before it begins using the link. The analytical results presented earlier indicated that substantial improvements can be achieved with virtual cut-through, so it is unreasonable to exclude it from any design.

(8) It is assumed that all virtual circuits are set up before the tasks begin execution. All of the application programs studied are static in the sense that new tasks are not created after execution begins. Since the programs execute for long periods of time, the set-up time is negligible relative to the total execution time. Thus, it's effect on overall performance can be neglected.

(9) Finally, the simulator uses a shortest path routing algorithm to set up its virtual circuits. Within the simulator, Floyd's algorithm [Floy62] is used to perform this computation. To prevent unfair comparisons, one routing algorithm was used throughout all of the simulation studies. Shortest path routing was selected because it has a simple implementation and because it has some prospect of achieving good performance since it minimizes the amount of network resources, i.e. bandwidth, required for each virtual circuit. Evaluation of more sophisticated routing algorithms is a topic of future research.

3.3. The Application Programs

Traffic distributions are generated by application programs executing parallel algorithms. For the purposes of this study, an application program is characterized by the communication pattern it generates. In particular, communications are characterized by the structure of communications between the program and its surrounding environment, and the pattern of communications within the program, i.e. among its tasks.

External communications between the parallel program and its environment are assumed to fall into one of two categories:

(1) serial input, serial output (SISO).

(2) parallel input, parallel output (PIPO).

These two communication patterns are shown in figure 3.2. In SISO, the input data arrives from (is sent to) a single source (destination). In PIPO, the data arrives (leaves) in parallel from (to) several sources (destinations).

Several of the application programs implement signal processing functions which use an SISO communication pattern. A single processor samples the input waveform and distributes the data values to a number of the other processors

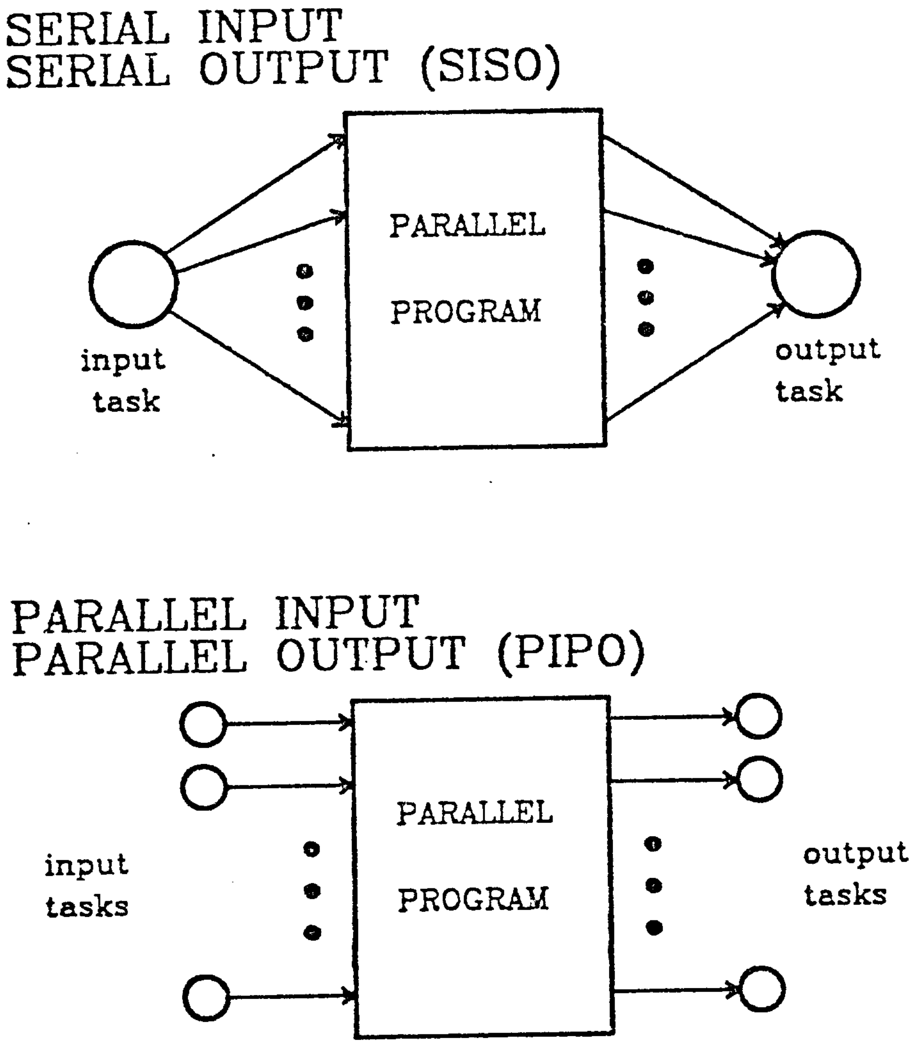


Figure 3.2. *Communication patterns for application programs.*

which collectively compute results. Another processor collects the output waveform. In other situations, a PIPO structure might arise. For example, the application program could be one of several job steps, each of which is implemented as a separate parallel program. Since the input (output) of each job step comes from (goes to) another parallel program, one can expect data to arrive (leave) in parallel. While other communication patterns are possible, e.g. SIPO or PISO, these are only combinations of the patterns presented above, and are not fundamentally different.

The internal communication paths are also partitioned into two categories:

(1) global

(2) local

As the name implies, global communications implies that each task communicates with all, or nearly all of the other tasks. Local communications implies each task communicates with a small subset of the other tasks. The programs studied here that use local communications are pipelined. Thus, the communications are local in the sense that each stage of the pipeline sends messages only to the next stage, and not to previous or subsequent stages. Although this communication structure does not exhibit loops among tasks in different stages of the pipeline, loops may exist among tasks within the same stage. Programs that exhibit loops among tasks in different stages are considered to belong to the class with global communications.

Six application programs demonstrating several different traffic patterns were run on Simon. Each uses one of the four combinations of the parameters described above. These are:

(1) Barnwell, a signal processing program using Barnwell's algorithm (global SISO)

(2) Block I/O, a signal processing program using block filters (local SISO)

(3) Block State, a second program also using block filters (local SISO)

(4) FFT, a program for computing Fast Fourier Transforms (local PIPO)

(5) LU, a program for performing LU decomposition on a sparse matrix (global PIPO)

(6) Random, a program generating artificial traffic loads (global PIPO)

The communication patterns exhibited by these programs are summarized in table 3.1 below.

Table 3.1
Communication Structures
Used by the Test Programs

	SISO	PIPO
global	Barnwell (12 tasks)	Random (12 tasks) LU (15 tasks)
local	Block I/O (23 tasks) Block State (20 tasks)	FFT (32 tasks)

All of these programs communicate relatively small amounts of data frequently. Typically, a task waits for data values to arrive from other task(s), performs some floating point operations on them, and then generates a result which is passed on to another task(s). The number of processors ranges from 12 in the Barnwell program to 32 in the FFT. Each of these programs will now be discussed in greater detail.

3.3.1. Barnwell Filter Program (global SISO, 12 tasks)

The Barnwell filter, and the two programs which follow, implement the digital filter defined by the equation:

$$Y_n = \sum_{i=1}^{N-1} b_i Y_{n-i} + \sum_{i=0}^{M-1} a_i X_{n-i}$$

Vectors X and Y are the input and output waveforms, A and B characterize the filter being implemented, and N and M are the number of poles and zeros in the filter respectively. The programs presented here use $M = N = 7$.

An "input task" distributes a total of 400 samples of the input waveform (the real multicomputer would collect this data from a sensor at some sampling frequency) to some number of "computation tasks". An "output task" collects the output waveform computed by the computation tasks. Thus, all three programs have an SISO communication pattern. When all of the 400 input samples have been processed, execution terminates. It is assumed that the sampling frequency is large compared to the rate at which data points can be processed. This ensures that the execution time is not limited by the input data rate. Thus, at time 0, the input processor begins distributing the 400 data points to the computation processors and never waits for input data.

The Barnwell program computes the filtering function using Barnwell's algorithm [Barn78, Barn79, Hodg80, Barn82, Lu83]. The two signal processing programs which follow use a different technique for performing the calculations. In Barnwell, twelve tasks are used, as shown in figure 3.3. Each node in figure 3.3 represents a task, and each arc a virtual circuit. An arc which fans out to several destinations represents a broadcast communication.

The Barnwell program uses ten tasks to execute the signal processing algorithm. This is the maximum number of processors the algorithm can effectively use in performing the computation, assuming small communication delays. This number is a function of the number of poles in the filter being implemented. Each computation processor receives 40 input samples.

Each data point received by a computation processor is combined with data generated by other processors. The result is then broadcast to the six processors immediately "to the right" of that processor. These communication paths are shown in figure 3.3. The communication pattern for the Barnwell program is

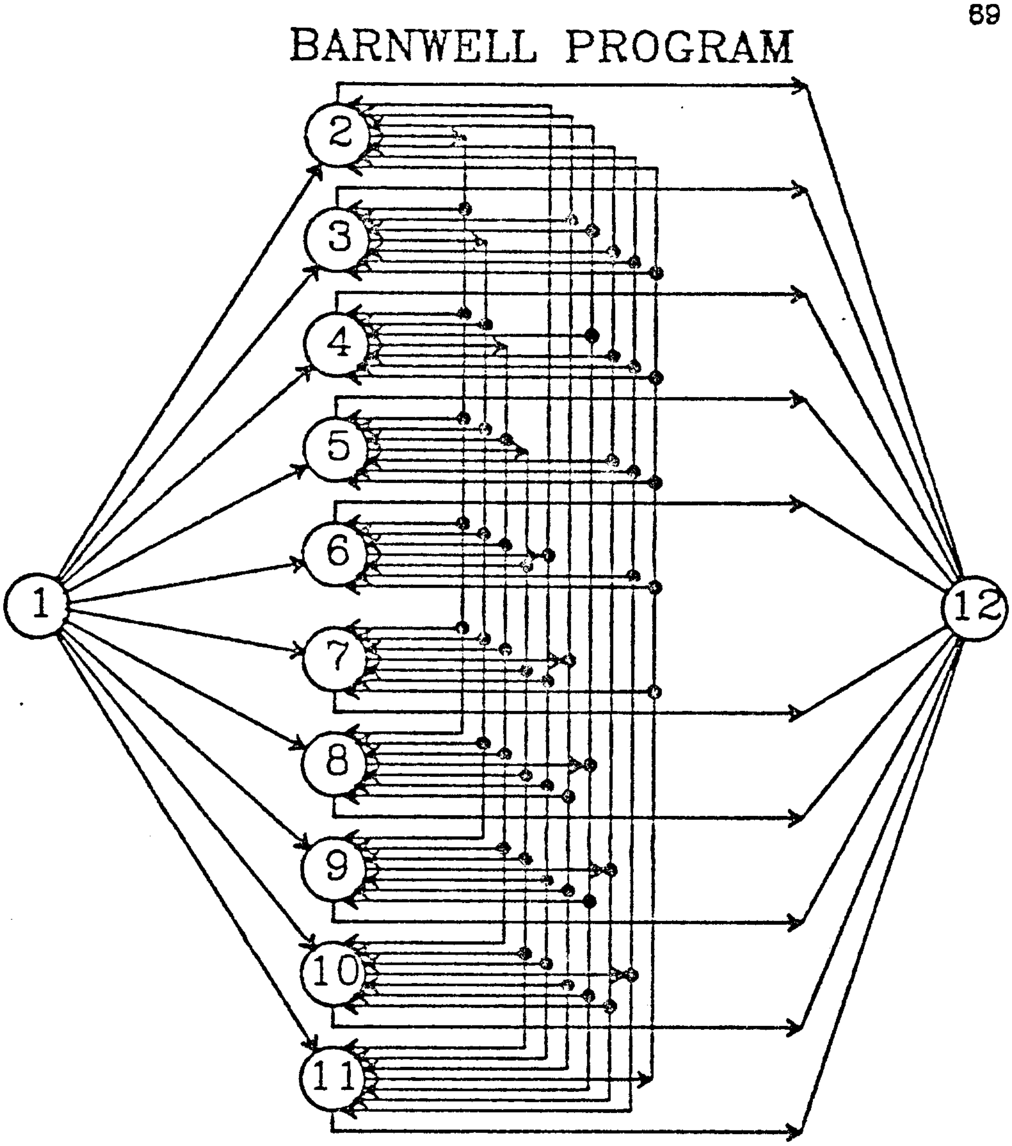


Figure 3.3. *Communication paths for Barnwell program.*

classified as global SISO, although communications are really only approximately global since each computation processor does not communicate with all others.

3.3.2. Block I/O Filter Program (local SISO, 23 tasks)

The Block State and Block I/O programs perform the filtering function described above by grouping the input samples into blocks and then processing each block as a single unit. The resulting communication patterns are local SISO. These algorithms have the advantage that the block size can be varied to change the performance of the system. A larger block size requires a larger number of processors, but increases the rate at which input samples can be processed. Increasing block size does incur a latency penalty however. The amount of time between reception of the first input sample and the generation of an output waveform increases. In practice, one would use the minimum block size which allows the input samples to be processed in real time; this minimizes the latency as well as the number of processors.

Here, the Block I/O and Block State programs use the minimum block size in order to minimize latency. This minimum size is related to the number of poles in the filter. Given this block size, the computation is structured to use as many processors as required to exploit the parallelism inherent in the computation. The Block I/O program uses 23 tasks which are structured as a two-stage pipeline, as shown in figure 3.4. Communications within the second stage are global, so the program is actually somewhat intermediate between local and global SISO. Note that input samples must be broadcast to several other tasks. Details of the algorithms implemented by this program can be found in [Burr71, Burr72, Mitr78, Lu83].

3.3.3. Block State Filter Program (local SISO, 20 tasks)

The Block State program uses the same "blocking" techniques discussed in Block I/O. This program however, uses a somewhat different approach to perform the computation, and as a result includes information of the internal behavior of the filter as well as the input-output relationships. Thus, it allows the determination of some intermediate values which the Block I/O program does not compute. As before, the minimum block size is used, resulting in a computation which requires 20 tasks. The communication paths for this

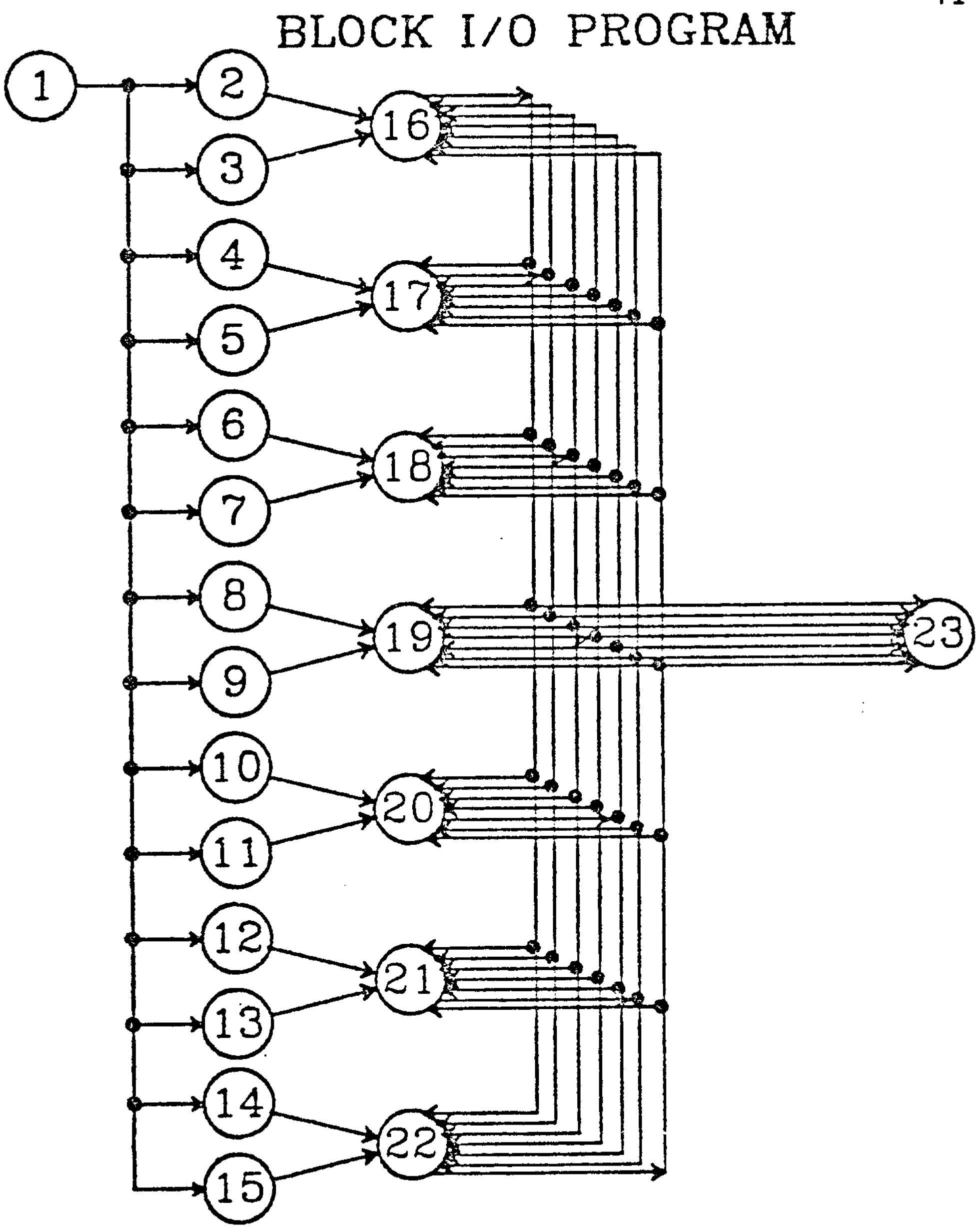


Figure 3.4. *Communication paths for Block I/O program.*

program are shown in figure 3.5. It is seen that the computation uses a 4 stage pipeline, and thus exhibits a local SISO communication pattern. Again, input samples are distributed via multiple-destination messages. Further details of the algorithms used in the Block State program can be found in [Barn80a, Barn80b, Zema81, Lu83].

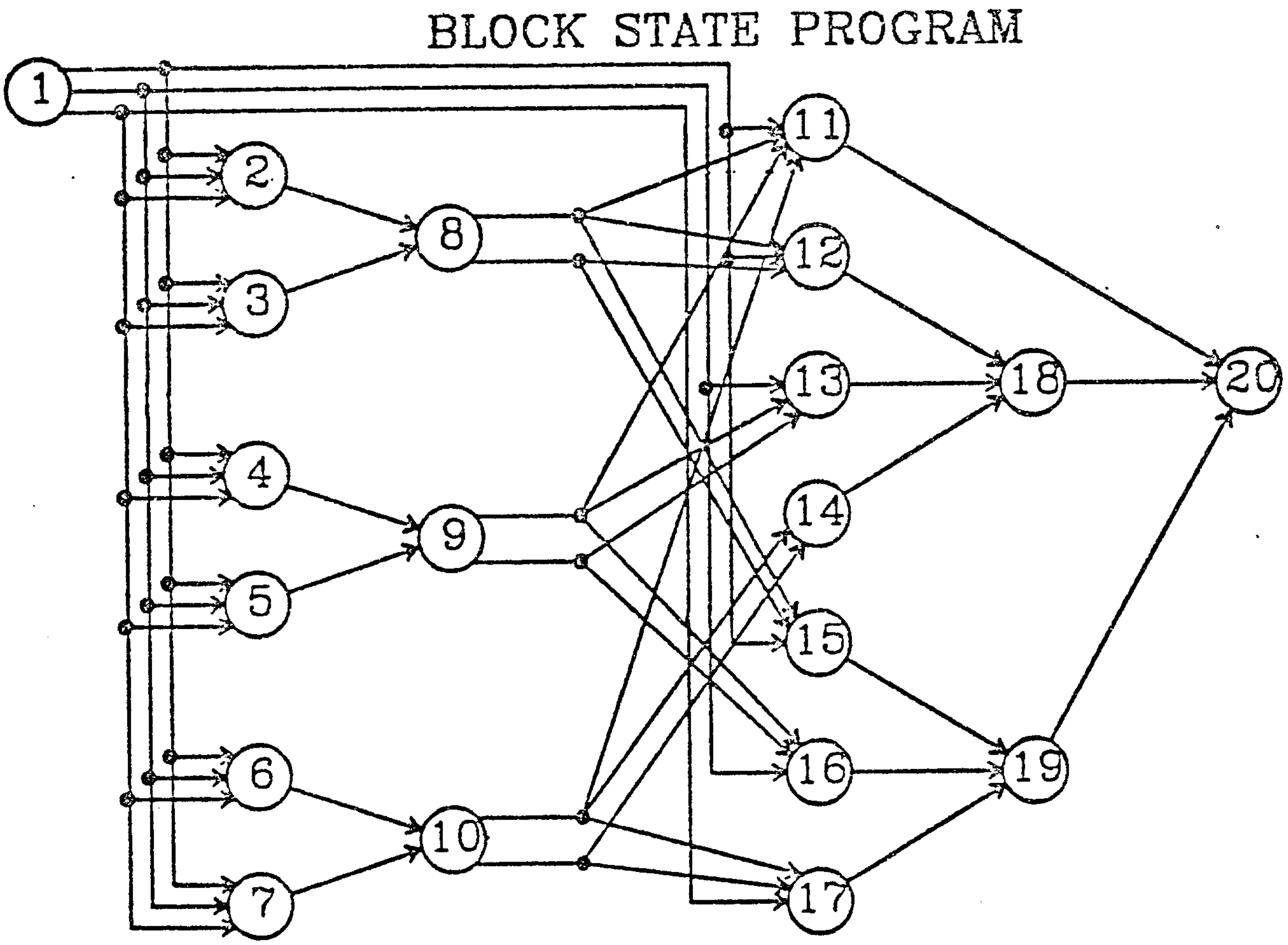


Figure 3.5. *Communication Paths for Block State program.*

3.3.4. FFT Program (local PIPO, 32 tasks)

This program performs a complex 16 point Fast Fourier Transforms on sets of input values. The FFT algorithm is used to compute the Fourier coefficients for an analog signal. The input consists of 400 sets of complex input values, $x_0 \cdots x_{15}$. The output consists of 400 sets of complex numbers $y_0 \cdots y_{15}$, such that

$$y_i = \sum_{k=0}^{15} x_k \exp((-2\pi / 16)ik)$$

Details of the algorithm used to perform this computation in time proportional to $N\log N$ (here, N=16) are discussed in e.g. [Baas78].

The communication paths used by this program are shown in figure 3.6. Since the same computation is performed on several sets of input data, the computation can be pipelined. The input data are assumed to reside in the processors comprising the first stage of the pipeline, so the resulting communication paths are local PIPO.

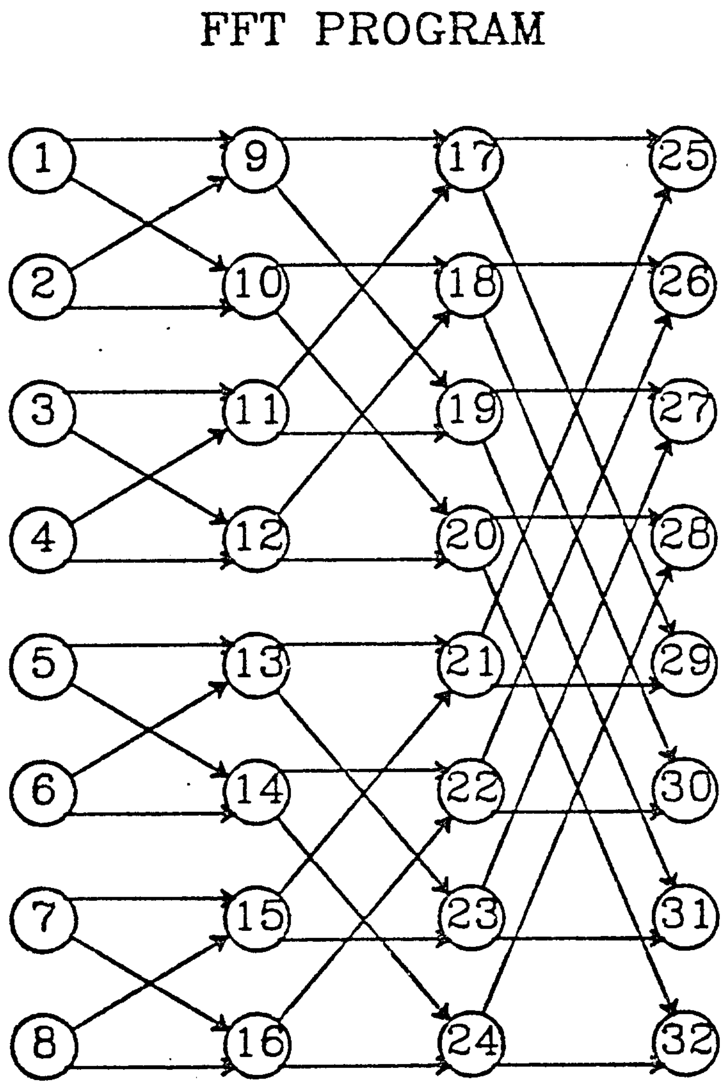


Figure 3.6. *Communication paths for FFT program.*

3.3.5. LU Decomposition (global FIFO, 15 tasks)

This program performs LU decomposition on a sparse matrix. LU decomposition is a well known technique for solving a set of linear equations. Suppose a set of equations is specified as

$$AX = Y$$

where A is a known n by n matrix, Y is a known column vector of length n, and X is an unknown column vector also of length n. The solution to this equation can be found by factoring the A matrix into two components, L and U, and then solving the equations

$$LB = Y \quad \text{and} \quad UX = B$$

in turn for B and then for X. L and U are upper and lower diagonal matrices respectively, i.e. all of the elements above (for L) or below (for U) the main diagonal are 0, so these two equations can be easily solved by forward and backward substitution respectively. If the equation $AX = Y$ is solved many times with different values for Y, then this method is more efficient than solving the original equation ($AX=Y$) repeatedly by say, gaussian elimination [Dahl74].

LU decomposition is one step in the inner loop of the circuit simulation program SPICE, so it must be executed repeatedly on each circuit simulation run [Nage75]. The parallel program used in these experiments performs the decomposition by using Doolittle's algorithm [Chua75]. The matrices used in this application are sparse, but not necessarily banded, making other techniques, e.g. systolic methods [Mead80], less attractive.

Given a sparse matrix, the parallel program was generated by first creating uniprocessor code for performing the computation, analyzing the data dependencies within this code, and then creating a parallel program from the data dependency graph [Yu84, Wing80]. For the program in question, the communication pattern which results from this process is global, i.e. every task sends messages to every other task. The program is PIPO since LU decomposition is only one of several parallel job steps in the inner loop for SPICE. Input (output) values can be expected to arrive from (be sent to) another parallel program executing the previous (subsequent) step of the inner loop.

3.3.6. Artificial Traffic Loads (global PIPO, 12 tasks)

A program creating synthetic traffic loads using random number generators was also studied. In the discussion which follows, this program is referred to as the "Random" program. In contrast to the other application programs, this program does not perform any useful computation. Its only function is to generate traffic for the communication network. The program consists of 12 tasks, each of which sends a total of 500 single-packet messages. Messages are uniformly distributed among other tasks, implying global communications. Since each processor originates its own messages, in contrast to a single processor generating all messages, the external I/O structure is PIPO. The mean time between messages is chosen from an exponential distribution. Loading on the network is increased by reducing the average time between messages.

3.4. Communication Delays

Figure 3.7a shows the performance of these application programs using a fixed-delay, infinite-bandwidth switch. Speedup, which is defined as the execution time of the program on a uniprocessor divided by the execution time on the multicomputer, is plotted as a function of communication delay. Here, delay refers to the end-to-end delay to send a message along a virtual circuit. It is assumed that this delay is the same along all circuits. Since the switch provides unlimited bandwidth, any number of processors may simultaneously send messages.

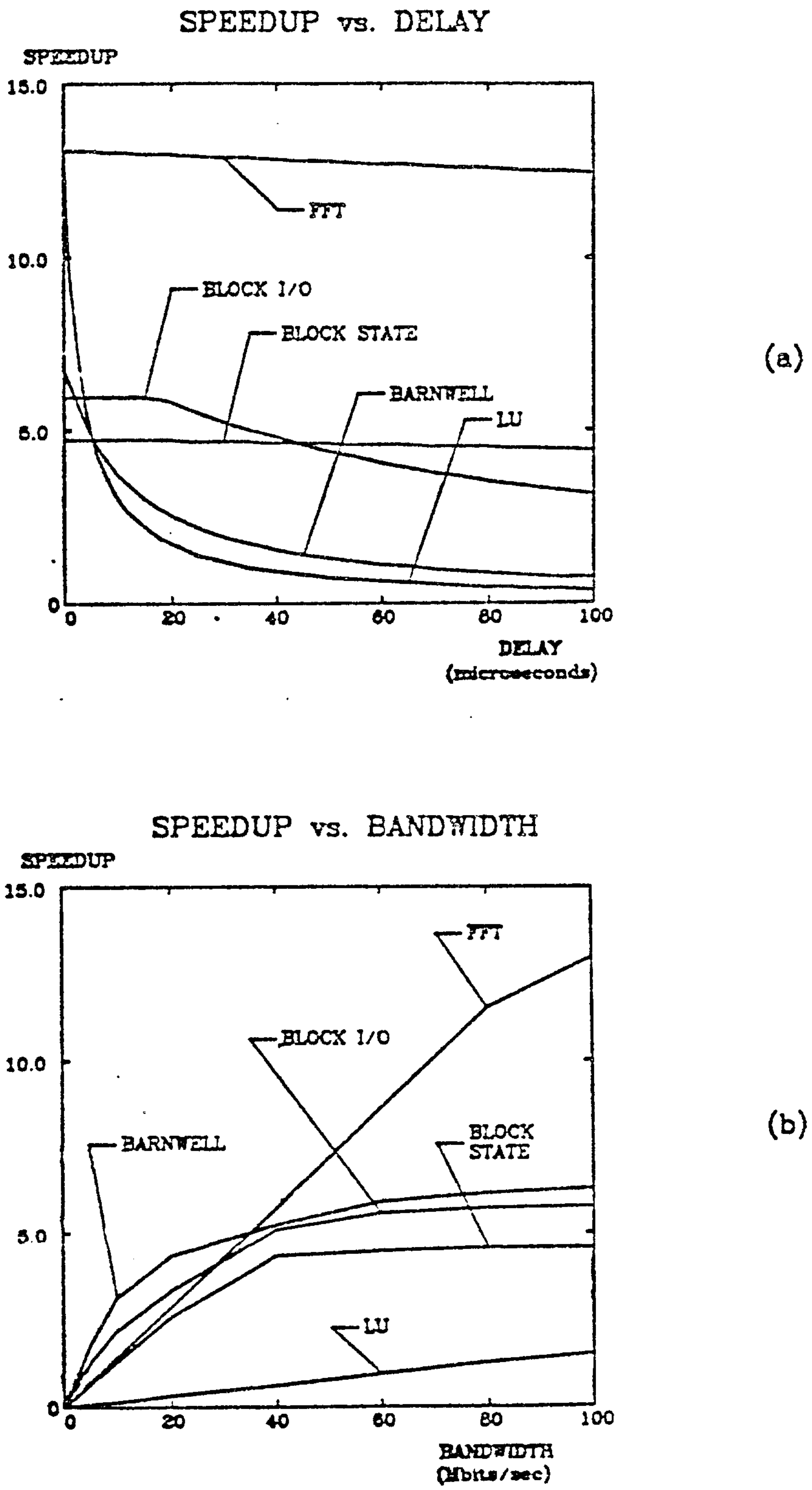


Figure 3.7. *Speedup vs. (a) delay. (b) bandwidth.*

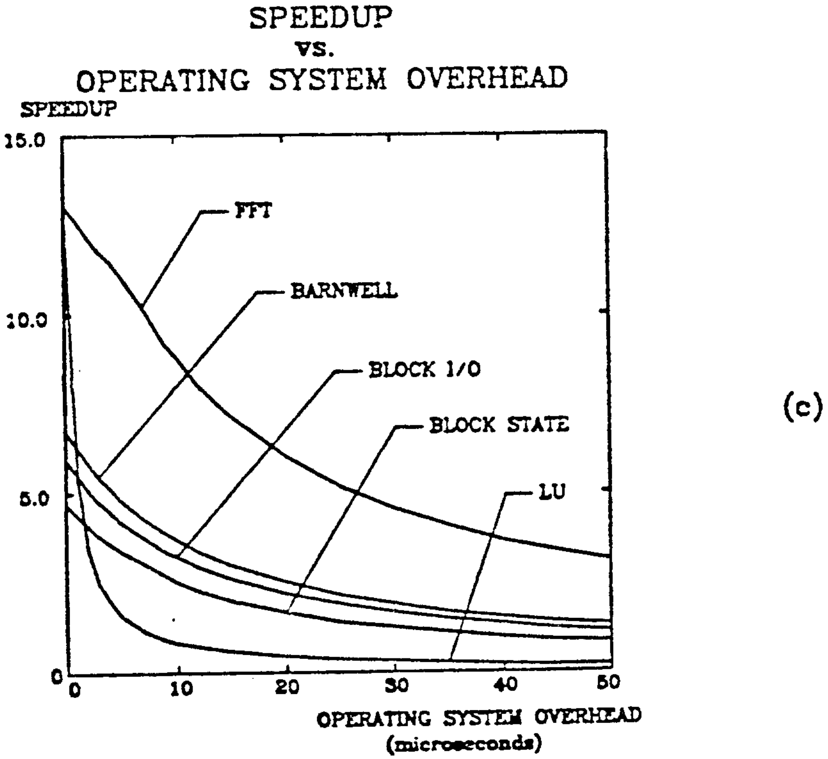


Figure 3.7. *(c) Speedup vs. operating system overhead.*

The two programs using global communications, Barnwell and LU, experience a severe degradation in performance as communication delays increase. This results from the relatively fine "granularity" of the computation, in which communications are frequent and delays have a significant impact on total execution time. On the other hand, the FFT and Block State programs exhibit little performance degradation as delays increase. These programs are pipelined, so delays only affect the amount cf time required to fill and empty the pipe. Once the pipeline is filled, data arrives at each processor at a constant rate, independent of communication delay, so all of the processors remain busy. It is erroneous however, to conclude that the interconnection switch does not impact the performance of these programs, since the curves in figure 3.7a assume unlimited network bandwidth.

The curves in figure 3.7b show the performance of the programs as a function of network bandwidth. Conceptually, the network can be viewed as an entity which provides a certain amount of bandwidth (this quantity is plotted on the horizontal axis in figure 3.7b) for transmitting messages. The optimistic assumption is made that all of the network's bandwidth can be allocated to a single virtual circuit on demand. In the simulator, this is implemented by using an "ideal bus" switch model. The communication network consists of a single bus of the indicated bandwidth. The full bandwidth of the bus is allocated to messages as they are generated. Conflicts to access the bus are queued in FIFO sequence, and propagation delays along the bus are assumed to be zero. The curves indicate that although the performance of the pipelined programs is insensitive to communication delay, adequate network bandwidth is required to achieve good performance. The programs exhibiting global communication patterns behave similarly. The LU program in particular, is seen to require very large amounts of bandwidth before achieving good performance. Simulations at higher bandwidths indicate that a 500 Mbit/second network is required to achieve a speedup of 10.0 (speedup with an infinite bandwidth, zero delay switch is 12.7).

Finally, the curves in figure 3.7c indicate performance as a function of operating system overhead. Here, overhead is measured as the time required to execute an operating system routine for sending or receiving a message. Transmission delays are assumed to be zero. It is seen that degradation is severe when delays in the operating system are only a few tens of microseconds. This result again is a consequence of the relatively fine granularity of the computation. It points out that hardware support for operating system primitives (here sending and receiving messages) is required to allow full exploitation of the parallelism inherent in many programs. With a traditional software implementation, the time spent in the operating system will dominate the transmission time, negating the benefits of incorporating a high-performance communication network. In particular, since recovery from transmission errors is left to an end-to-end protocol, hardware support should be employed in the computation processor to keep these checks from degrading performance. Hardware support for communication primitives thus represents an important area of future research.

3.5. Issues Under Investigation

Four separate issues are studied in these simulation experiments. The first explores the optimal number of ports, and compares simulation results with those predicted by the analytical models presented earlier. Next, an alternative model in which processor and communications are integrated onto the same chip is studied. Third, since many of the application programs send the same message to several different destinations, the impact of incorporating a mechanism which efficiently handles such messages, i.e. a multicast mechanism, is investigated. Finally, the particular mapping of tasks to processors which was used in these experiments is examined, as well as its impact on the simulation results.

In order to evaluate the optimal number of ports, two types of switch models were implemented: cluster nodes and networks with a fixed number of components. These switch models correspond to the networks discussed in the analytical studies presented earlier. In the first, each node of a topology requiring b branches per node is implemented with a cluster of p-port communication components. As p is reduced, the number of components required to construct the network is increased. Thus, the cluster node switch models do not keep the chip count constant. The second set of switch models compares networks with different values of p, but with approximately the same number of components.

In addition to networks constructed from separate computation and communication components, networks with processor and communications integrated onto the same chip are studied. This is the building block for the "network computer" proposed by Wittie [Witt81]. In this model, the communication links between the computation and communication domains are eliminated. In communication component networks, it will be seen that these links sometimes become bottlenecks which bias the results. The simulations under this latter model eliminate this bias. Multicomputers using the Wittie model do require more circuitry per chip than those using communication components, making direct comparisons unfair. Nevertheless, it is included as an alternative model for multicomputer networks.

Since the digital filtering algorithms (Barnwell, Block State, and Block I/O) involve transmitting the same data to several destinations, a mechanism which efficiently distributes multiple-destination packets (i.e. a "multicast" mechanism) is expected to improve performance.

If a multicast mechanism is *not* used, several "single destination" packets are generated at the source node, one for each destination, and each is routed separately through the network to its particular destination using a shortest path routing algorithm. If one traces the paths followed by these packets through the network, it is seen that packets will follow each other up to a certain point, at which time they part and go their separate ways. The multicast mechanism combines the single destination packets which are "following each other" into a single "multicast packet". A new copy is not generated until one or more of the single destination packets incorporated into the multicast packet need to "go their separate ways". If several packets breaking off like this are all going in the same direction, only one new multicast packet is created. Multicast and broadcast mechanisms are described more fully in [Dala78, Bhar83, McQu78]. Note that since virtual circuits are used, implementation of this does not affect other parameters of the switching network. A longer header might be needed to provide a list of destination nodes. However, in a virtual circuit mechanism, this information need only be carried through the network when the multicast circuit is set up.

The mapping of the application program onto the network topology is identified by labels assigned to tasks and processors. As shown in figures 3.3-3.6 (the remaining two programs, Random and LU, use global communications, so the mapping does not influence the results), each task of each application program is characterized by a unique integer called its "task id". Similarly, each node, i.e. processor, of a topology is characterized by a unique node number. In the discussions which follow, task i always executes on processor i. Thus, the simulation results assume a specific mapping of tasks to processors. Care must be taken to ensure that this mapping does not bias the results. More will be said about this later.

3.6. Simulation Results on Cluster Node Networks

As discussed earlier, one can implement a node of a topology requiring b branches per node as a cluster of p-port communication components. The various application programs described above were run on Simon using switch models for several different cluster node networks. The results of these simulation experiments are reported in this section.

For this study, four topologies are examined which vary b over a wide range of values. All topologies are assumed to use full duplex, bidirectional links. These topologies are:

(1) Fully connected network.

(2) Full-ring binary tree [Desp78].

(3) Butterfly network.

(4) Ring network.

The topology within each cluster node is a balanced tree, with the processor attached to the communication component at the root.

In all of the graphs which follow, speedup is plotted as a function of B, the total I/O bandwidth of the communication chip. The only exception is the artificial traffic load program in which average message delay is plotted as a function of traffic load. It is assumed that the bandwidth B is equally divided among the existing communication links. Thus, a Y-component with B equal to 300 Mbit/second has three 100 Mbit/second communication links. For comparison, the speedup on a multicomputer with an infinite-bandwidth, zero-delay interconnection system (i.e. a "perfect switch") is also shown. The perfect switch assumes that messages arrive at their destination at the instant at which they are sent. It thus gives an upper bound on performance for any communication network.

The analytical results presented earlier indicated that cluster node networks constructed from communication components with a small number of ports yielded the most bandwidth and least delay. Thus, one would expect networks constructed from Y-components to yield the best performance. It will be seen that the simulation results confirm this conclusion.

3.8.1. Fully Connected Networks

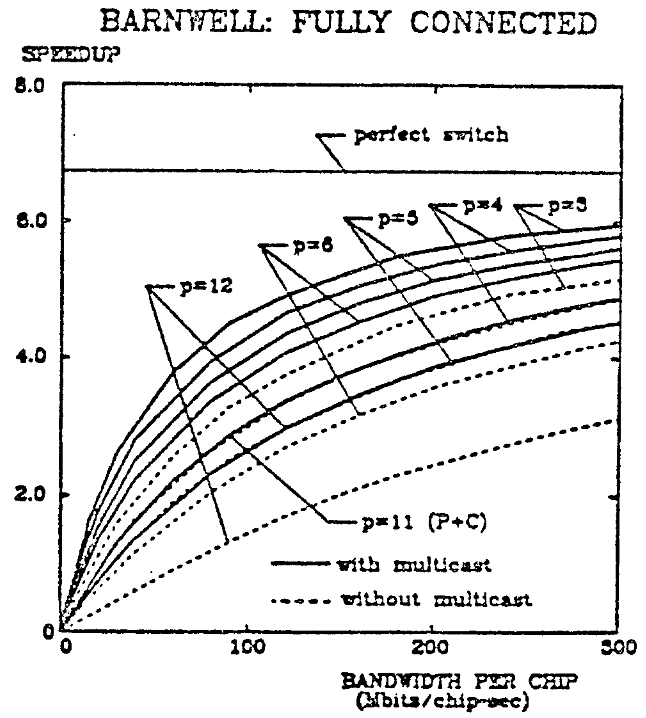
The fully connected network is formed by placing a single link between every pair of nodes. Here, the number of nodes is equal to the number of tasks required by the parallel program, and it thus varies from application to application. This topology minimizes the number of hops between every pair of nodes, but at the expense of a larger number of branches on each node.

Three of the application programs were run on Simon with switch models for fully connected networks. Performance curves are shown in figures 3.8a-c. Due to limited amounts of computing resources, cluster node simulations for the FFT, Block I/O, and Block State programs are not available. Figures 3.8a-c indicate that performance improves as the number of ports is reduced, in agreement with the analytical results. Curves labelled "P+C" indicate that processor and communications circuitry are incorporated onto the same chip.

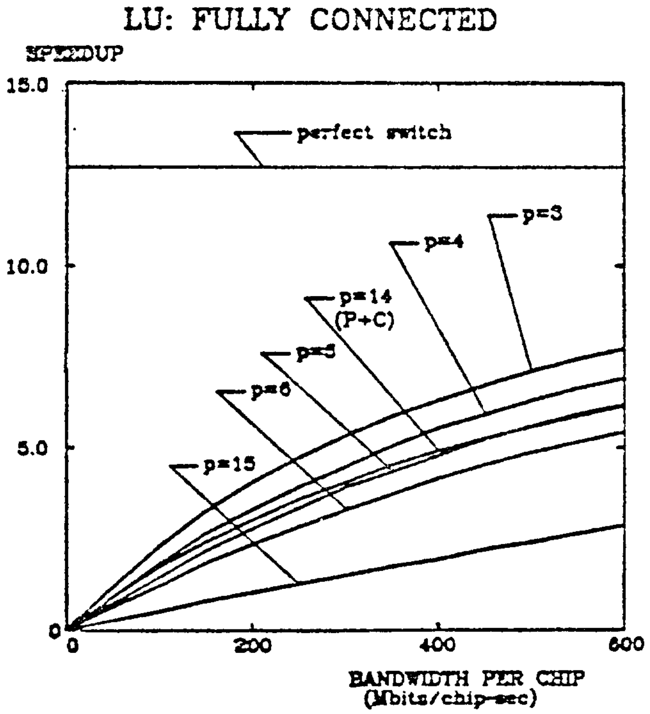
The curves resulting from the artificial traffic load program indicate that reducing the number of ports reduces the average delay in a lightly loaded network, and increases total network bandwidth. The bandwidth result however, is somewhat misleading because the total network bandwidth shown in figure 3.8c is limited by the link between the processor and its communication component. This is demonstrated by the curve in which communication circuitry is included in the same chip as the processor. Network bandwidth is increased significantly when this bottleneck is removed.

Figure 3.8d shows the curves for the artificial traffic load program with this bottleneck link removed. Here, it is assumed that the root component of each cluster node has both computing and switching capabilities. Other components only perform switching functions. As expected, delay in lightly loaded networks improves as the number of ports is reduced. The curves also indicate, however, that networks with a large number of ports provide as much bandwidth as those using Y-components. This unexpected result is a consequence of the lack of store-and-forward communications in the fully connected network, and the particular traffic distribution created by the artificial traffic load generator.

The bandwidths indicated by figure 3.8d represent the minimum of two quantities:



(a)



(b)

Figure 3.8. *Fully connected network (a) Barnwell. (b) LU.*

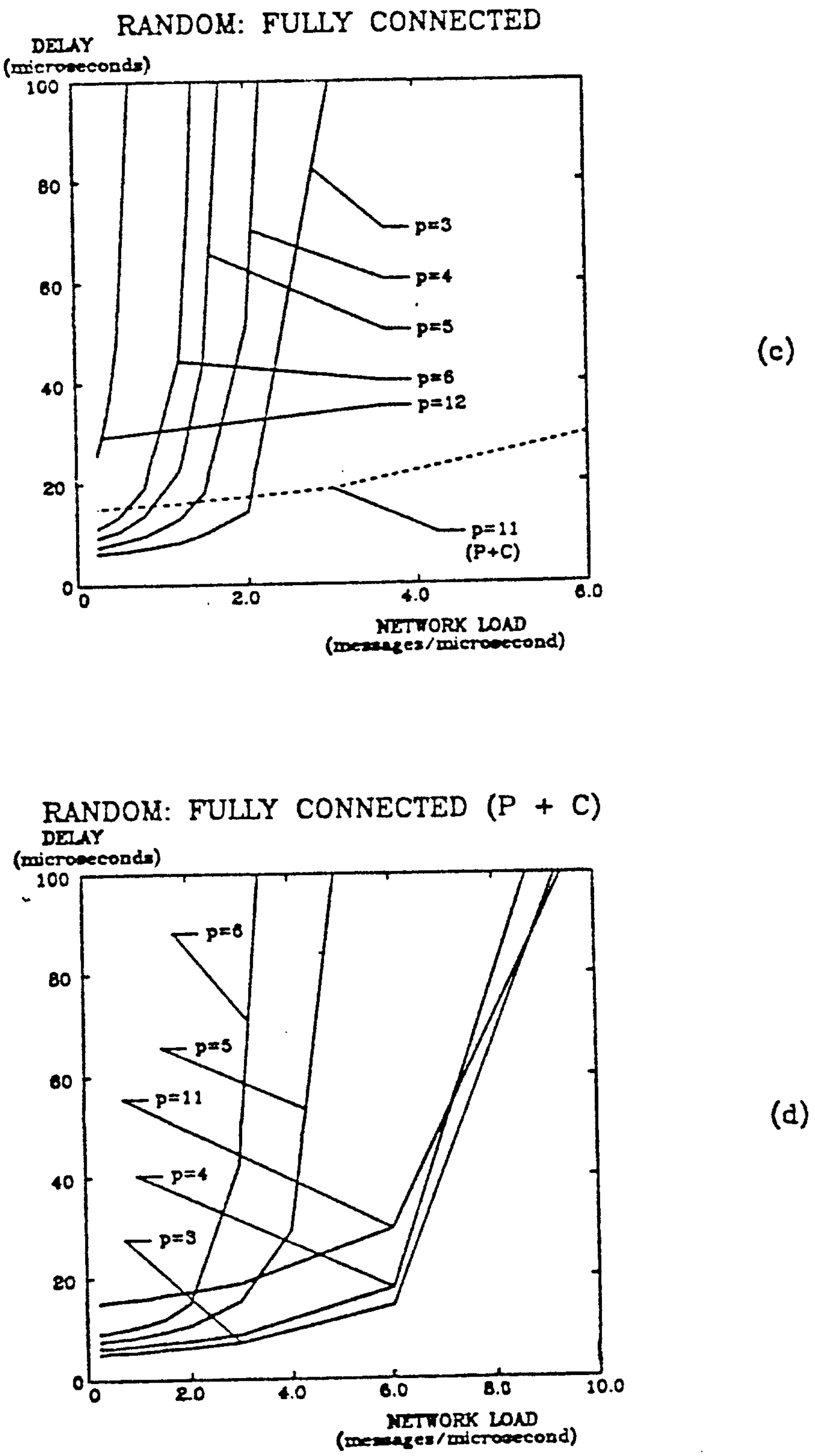


Figure 3.8. *Fully connected network (c) Random. (d) Processor with Communications.*

(1) The maximum bandwidth provided by the network.

(2) The maximum rate at which traffic can be generated by the processors.

If the first quantity is the limiting factor, then the tradeoff between hop count and link bandwidth discussed earlier determines the optimal number of ports. If the second quantity limits performance, then the utilization of the links around the processors generating messages determines performance. The more efficiently these links are used, the greater the amount of traffic sent into the network, and the higher the overall bandwidth. This quantity is maximized if the traffic generated by each processor is evenly distributed across that processor's output links, since this implies that on the average, all of the processor's links will be busy all of the time. An uneven distribution causes some links to be overloaded while others become idle, reducing the total traffic flow into the network.

In the artificial traffic load program, messages from each task are uniformly distributed among all other tasks. Thus, this program is a "perfect match" with the fully connected network with processor and communications integrated onto the same chip, since a direct link exists between each pair of communicating tasks. Because of the uniform traffic distribution, all links are equally utilized, and the amount of traffic generated by the processors is maximized. This rate determines the bandwidths shown in figure 3.8d.

Creating a new network by adding switching components increases the amount of traffic the network can carry, but the amount of traffic which can be generated is *not* increased. Thus, this additional network bandwidth cannot be utilized. In fact, performance will actually be degraded if the new network does not preserve the equal utilization of processor links described above. This phenomena explains the poor performance of some of the networks in figure 3.8d.

The behavior described above is atypical because the global/uniform traffic pattern of the artificial traffic load generator is not always appropriate. It will be seen that other topologies and different traffic patterns yield results favoring a small number of ports. Indeed, performance curves for the other application programs (figures 3.8a-b) indicate that networks using communication components with a small number of ports achieve better performance than networks with a large number of ports, even if the latter have the added advantage of including communication circuitry on the same chip as the processor. Networks with a small number of ports and processor and communications on the same chip will perform even better, widening this gap.

The curves for Barnwell's algorithm indicate that a significant performance improvement results from incorporating a multicast mechanism in the communication hardware. If no multicast mechanism is provided, the processor sending the message must send a separate copy to each destination. A queue appears instantly in the processor sending the message, leading to long delays and poor performance.

No multicast curve is shown when processor and communications are incorporated onto the same chip. This is because networks with and without a multicast mechanism behave identically under these circumstances. Since each processor has a direct link to every other processor, all "splitting apart" of the multicast packet is done at the source node. A network without a multicast mechanism behaves in exactly the same way for this topology.

3.6.2. Full-Ring Tree Networks

The second topology is the full-ring binary tree [Desp78]. This topology is constructed from a binary tree by adding links between siblings and cousins, as shown in figure 3.9. The average hop count grows logarithmically with the number of nodes, while the number of branches per node remains fixed at 5.

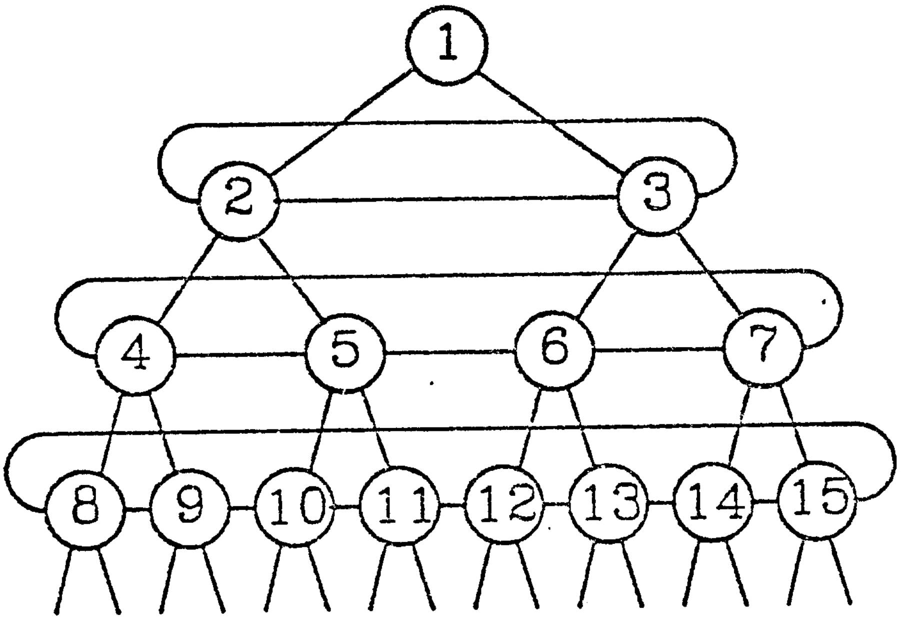


Figure 3.9. *Full ring binary tree.*

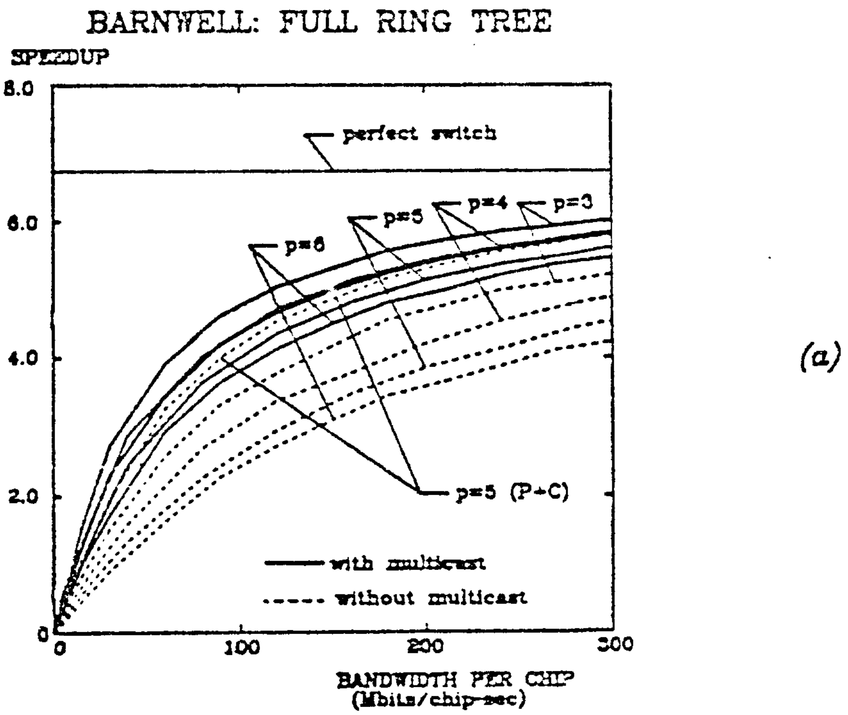


Figure 3.10. *Full ring tree (a) Barnwell.*

Performance curves for full-ring tree networks are shown in figures 3.10a-f. Qualitatively, these curves agree with those presented for the fully connected networks. Again, components with a small number of high-bandwidth links achieve the best performance.

The performance curves for Barnwell's algorithm (figure 3.10a) indicate that as the I/O bandwidth of the communication components is increased, speedup increases quickly at first, but becomes more gradual at higher link bandwidths. Other curves however, such as some of those for the FFT program (figure 3.10d), indicate a linear increase in speedup. These differences arise from the nature of the communication patterns for the different programs. The linear behavior arises when one virtual circuit remains the critical path for the program as chip bandwidth is varied. As bandwidth is increased, delay, and thus execution time, decrease in proportion. The pipelined programs often demonstrate this behavior, with the longest path from the first stage of the pipeline to the last forming the critical path. Many of the SISO programs also demonstrate this behavior. Here, the bottleneck is in distributing the initial data samples to the computations processors. The problem is aggravated if multiple-destination messages are required to distribute the samples, as is the case in the Block I/O (figure 3.10b) and Block State (figure 3.10c) programs, particularly if the network does not include a multicast mechanism. The speed of the links around the input processor becomes the primary factor which determines the execution time of the program. Nonlinear behavior results when no single virtual circuit dominates performance across all chip bandwidths. Instead, delays on a number of virtual circuits determine the overall execution time. Both types of behavior will be seen in the performance curves which follow.

The curves for the artificial traffic load program (figure 3.10f) again indicate that delay and bandwidth are both improved as the number of ports is

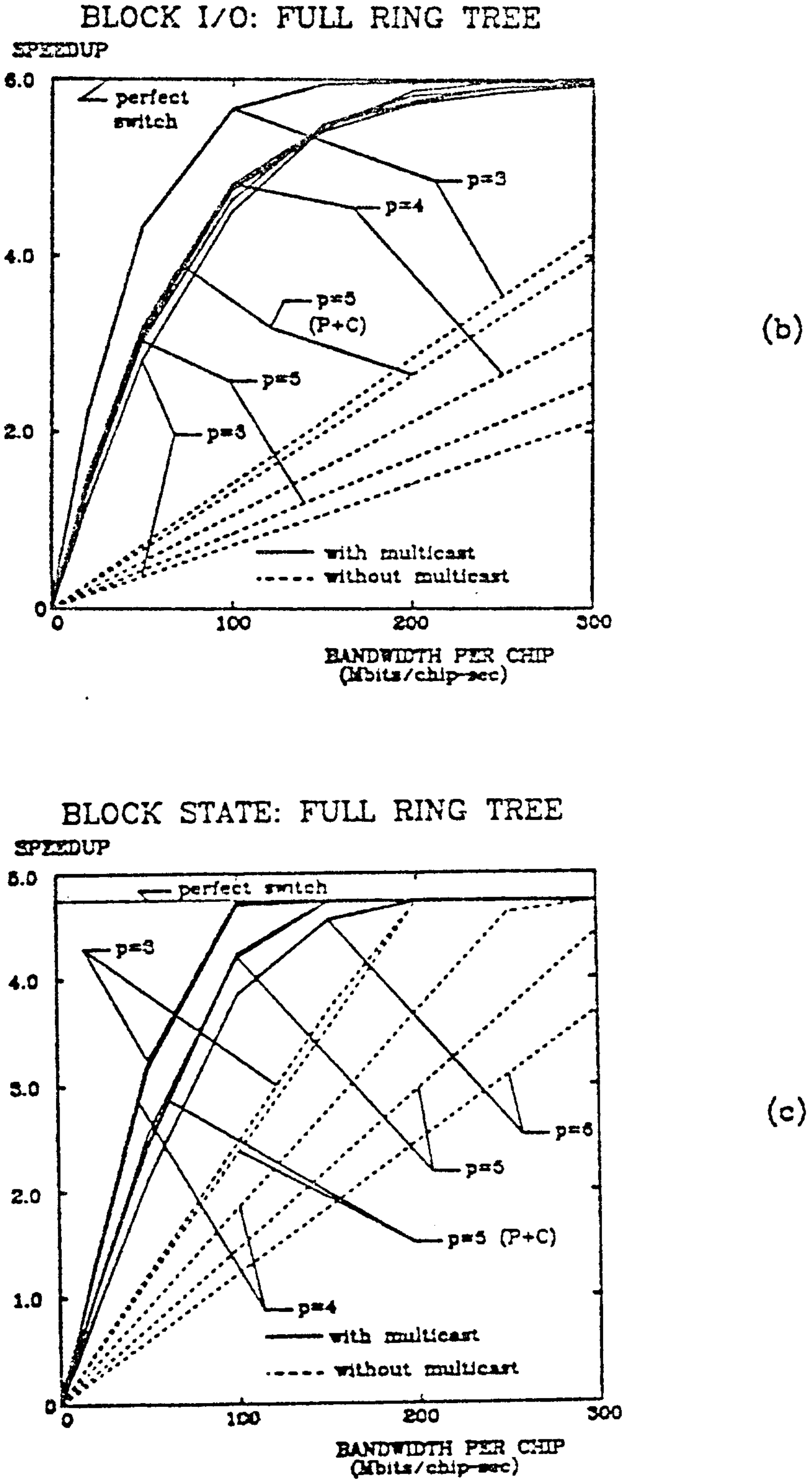


Figure 3.10. *Full ring tree (b) Block I/O. (c) Block State.*

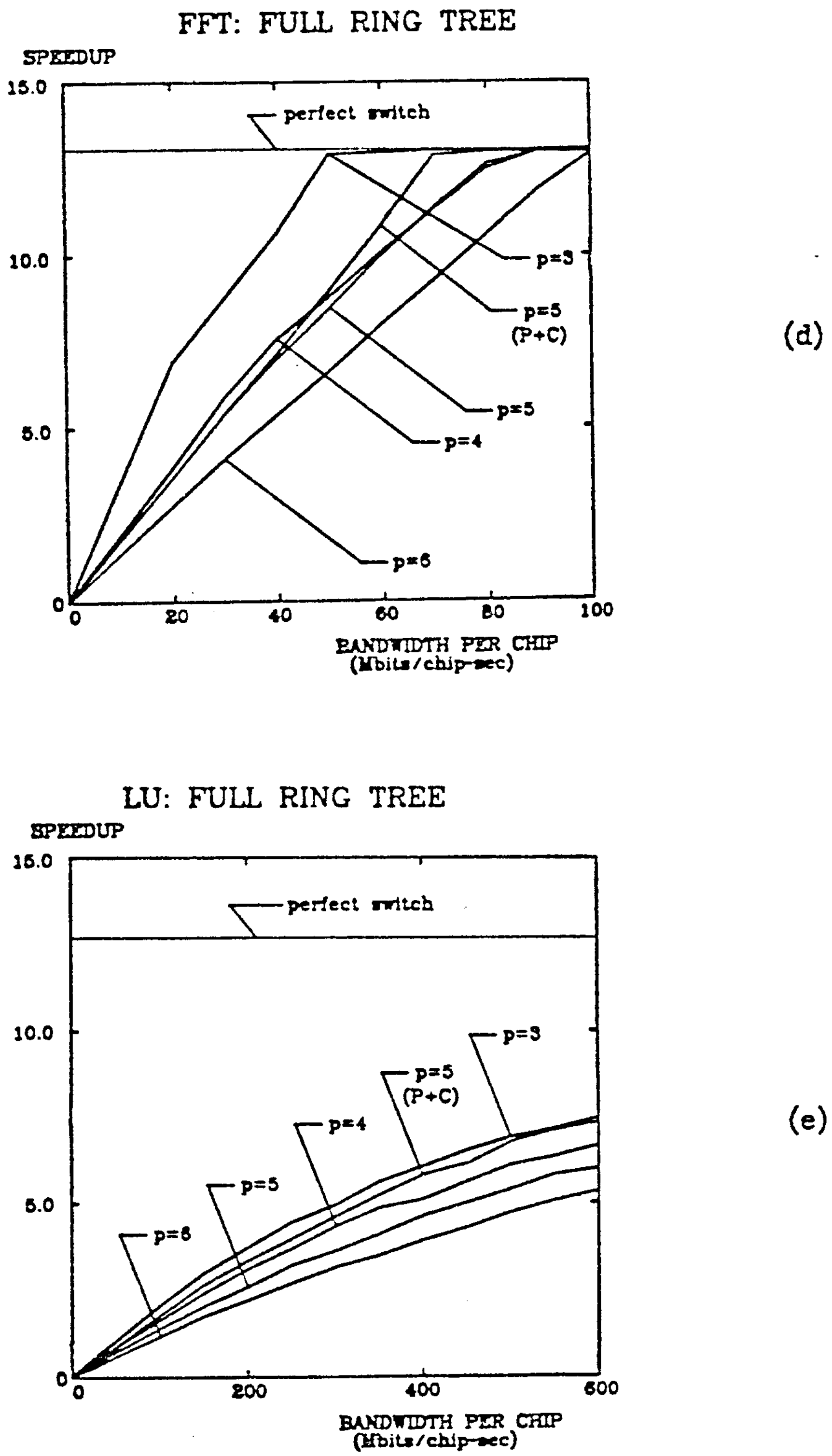
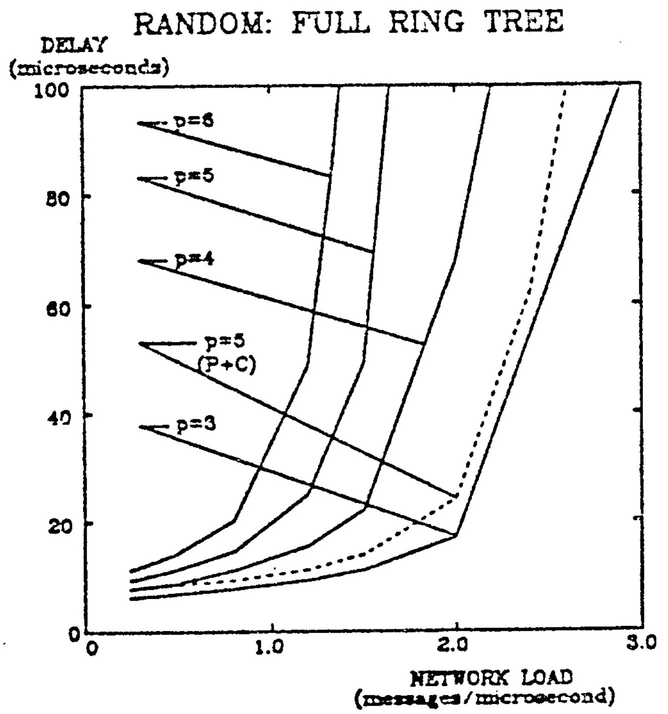
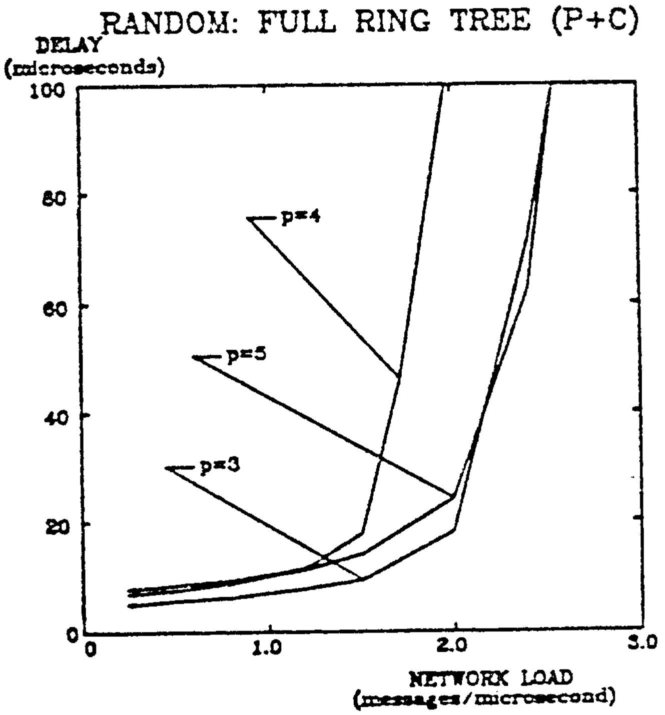


Figure 3.10. *Full ring tree (d) FFT. (e) LU.*



(f)



(g)

Figure 3.10. *Full ring tree (f) Random. (g) Random (P+C).*

reduced. The curve with processor and communications integrated onto the same chip indicates that the link between the processor and communication component is still a bottleneck, since the bandwidth provided is significantly better than that provided by networks using components with 4, 5, or 6 ports per node. This bandwidth is still somewhat less than that of the Y-component network however, even though the latter is handicapped by this bottleneck link. This adds further support to components with a small number of ports.

3.6.3. Butterfly Networks

The 32 node butterfly network shown in figure 3.11 is the third topology studied. The butterfly is similar to the tree to the extent that the average hop count grows logarithmically with the number of nodes. Four branches are required for each node. This topology is more symmetric than the tree however, and thus is less susceptible to bottlenecks for applications exhibiting global traffic patterns. The butterfly is ideally suited for the FFT application program.

Performance curves for the butterfly network are shown in figures 3.12a-e. Due to the excessive amount of computing resources required, a curve for the Block I/O program could not be produced. Networks constructed with components using a small number of ports again achieve the best performance. The FFT program (figure 3.12c) performs unusually well at low chip bandwidths, demonstrating the reduction in bandwidth requirements when a good mapping is found between the application program and hardware. The curve with processor and communications on the same chip in the artificial traffic load program (figure 3.12e) indicates that the link between the processor and communication component is not a serious bottleneck. The bandwidth provided is equal to that of the network using communication components with the same number of ports.

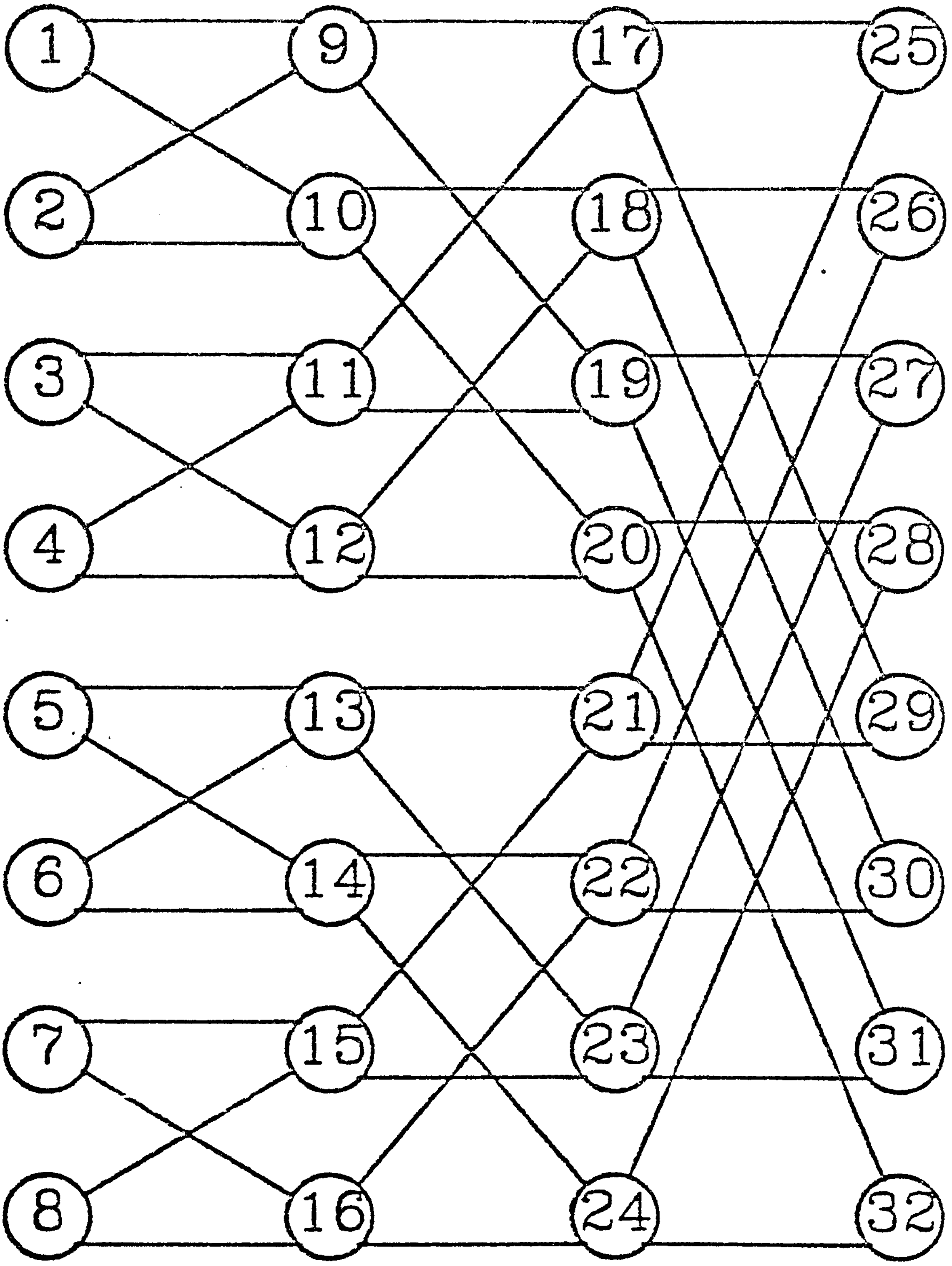


Figure 3.11. *Butterfly topology.*

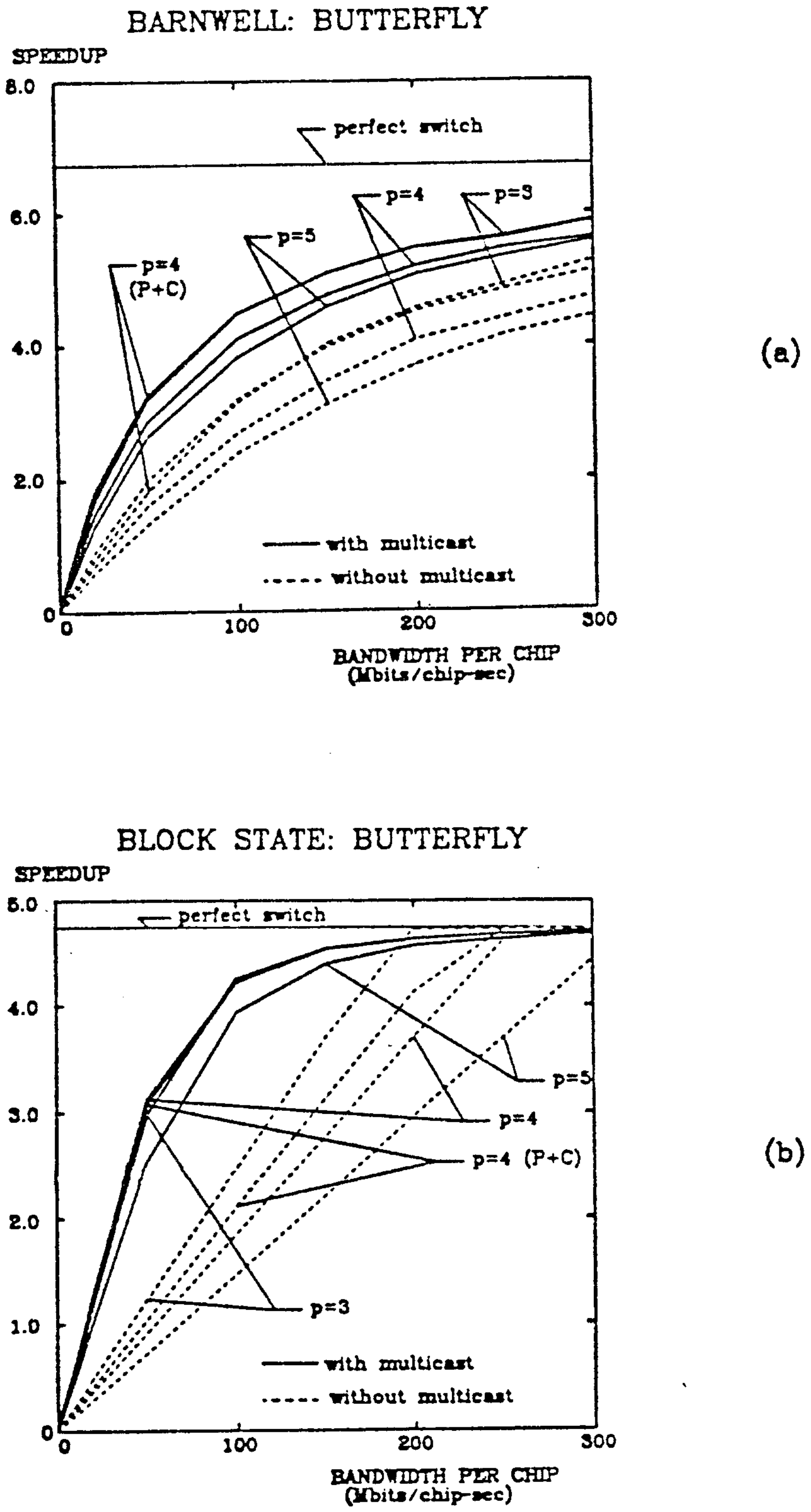


Figure 3.12. *Butterfly (a) Barnwell. (b) Block State.*

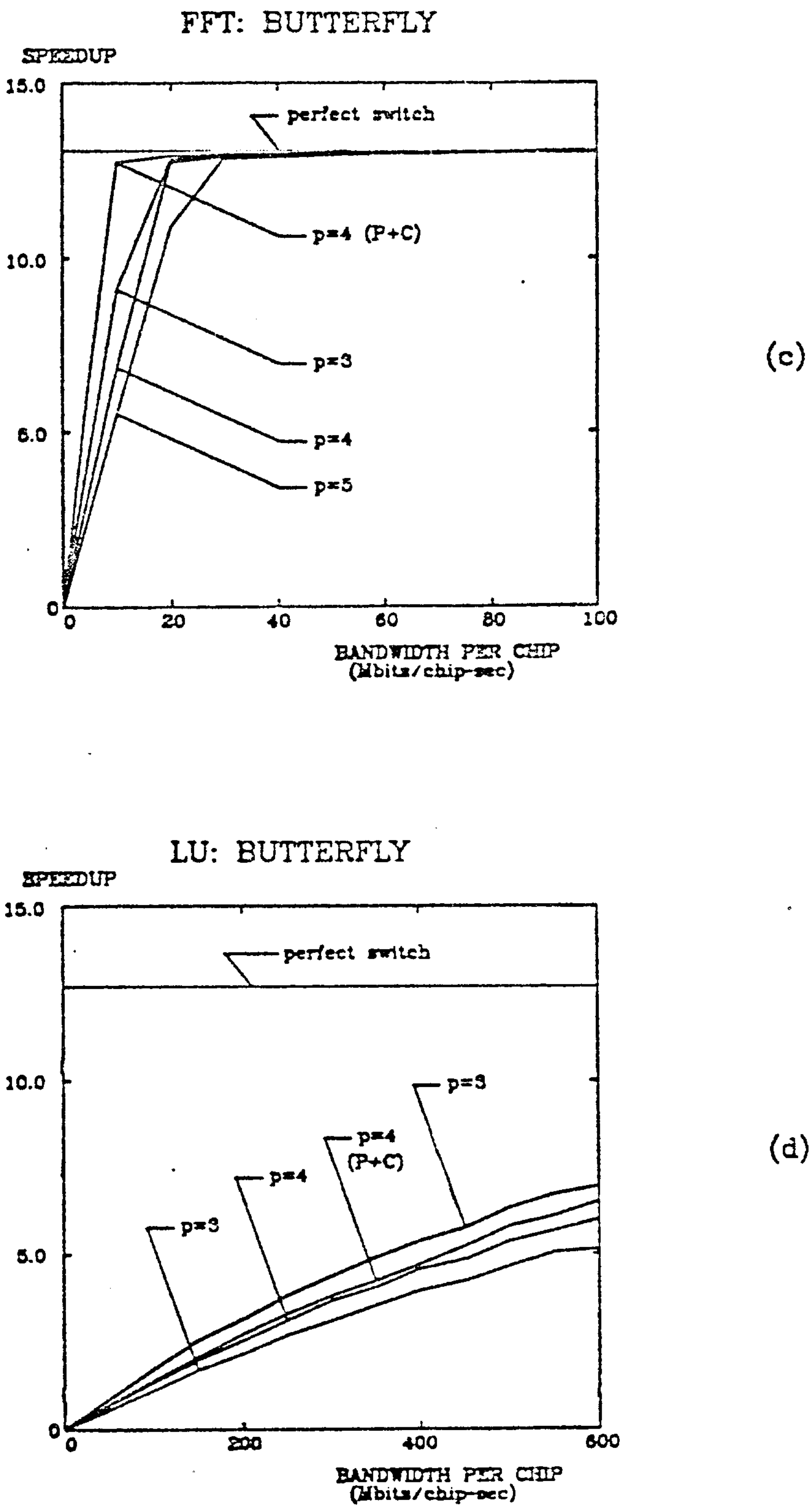


Figure 3.12. *Butterfly (c) FFT. (d) LU.*

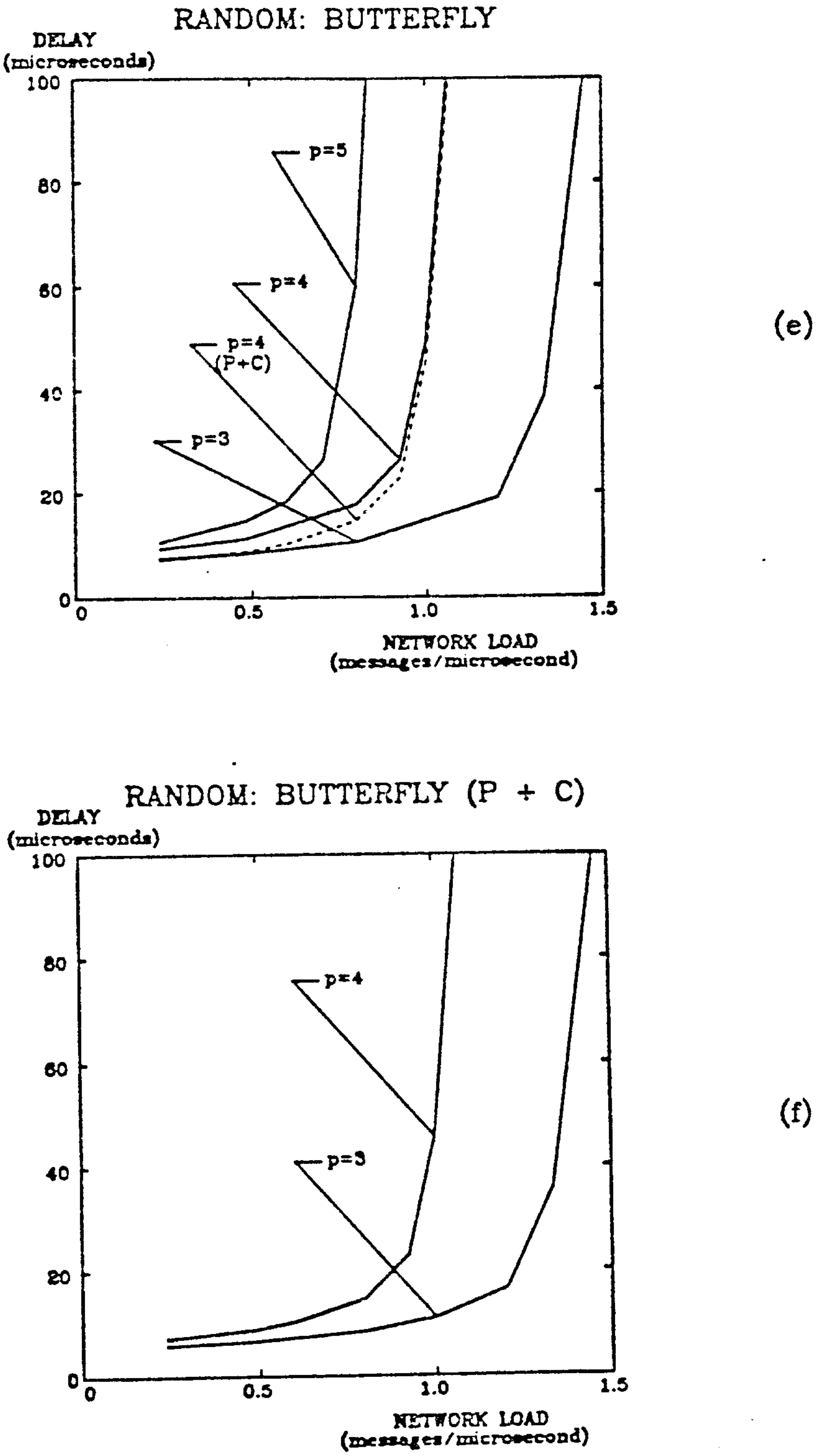


Figure 3.12. *Butterfly (e) Random. (f) Processor with Communications.*

3.6.4. Ring Networks

The fourth topology, a bidirectional ring, minimizes the number of branches per node, but maximizes the average hop count. Like the fully connected topology, the number of nodes is equal to the number of tasks in the application program.

Performance curves for the ring network are shown in figures 3.13a-f. In ring topologies, the use of communication components also implies the use of Y-components, since only three ports per chip are required. Thus, only two distinct networks need to be compared. The network with communication circuitry on the processor chip yields better performance since it has higher bandwidth links (only 2 ports are needed) and smaller hop counts. Thus, networks constructed with components with a small number of ports again achieve the best performance.

The only exception occurs for the Block I/O program. Here, the communication component networks achieve better performance at high chip bandwidths. This behavior is a consequence of the SISO behavior of the program. When execution begins, the "input" processor broadcasts data values to a number of computation processors. In ring networks without communication components, this causes the links around this processor to become saturated, blocking traffic produced by the other processors since messages are serviced at each node by a strict FIFO ordering. As a result, the signal processing calculations cannot proceed until this initial backlog of traffic is cleared up, slowing down the computation. If communication components are used, the link between the input processor and its communication component becomes saturated; however, this link does not block traffic among other processors. The arrival of the input data messages at the communication component is spread out over time. Thus, these messages do not completely block other traffic,

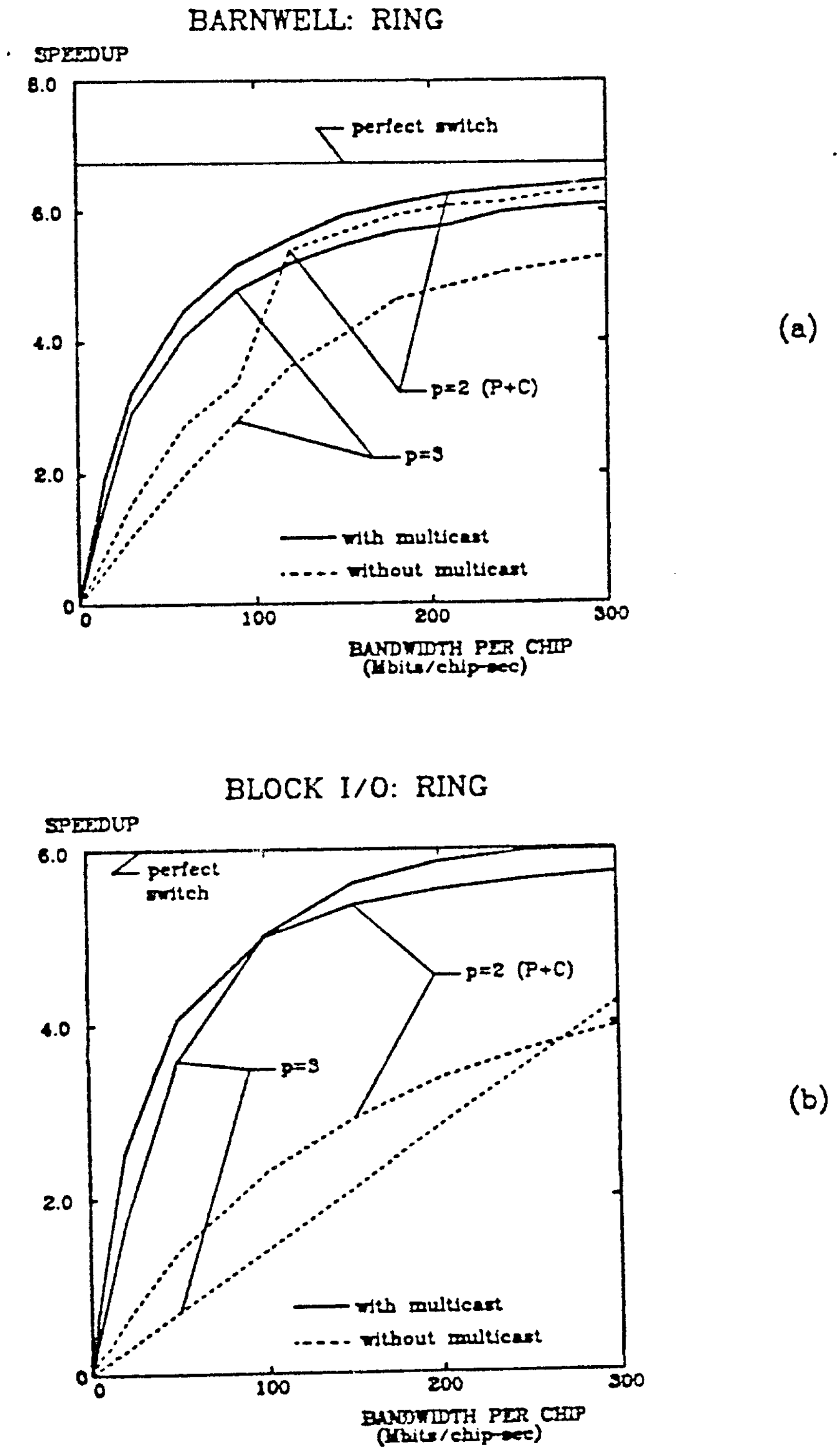
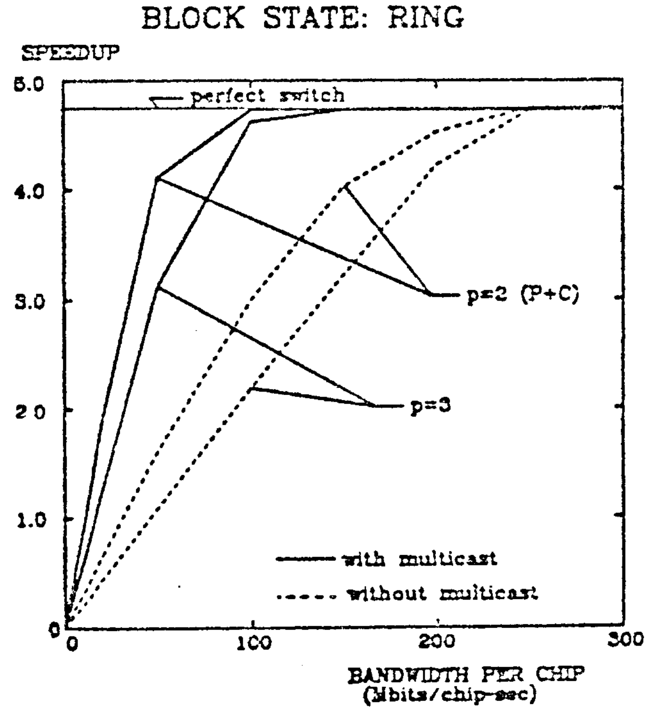
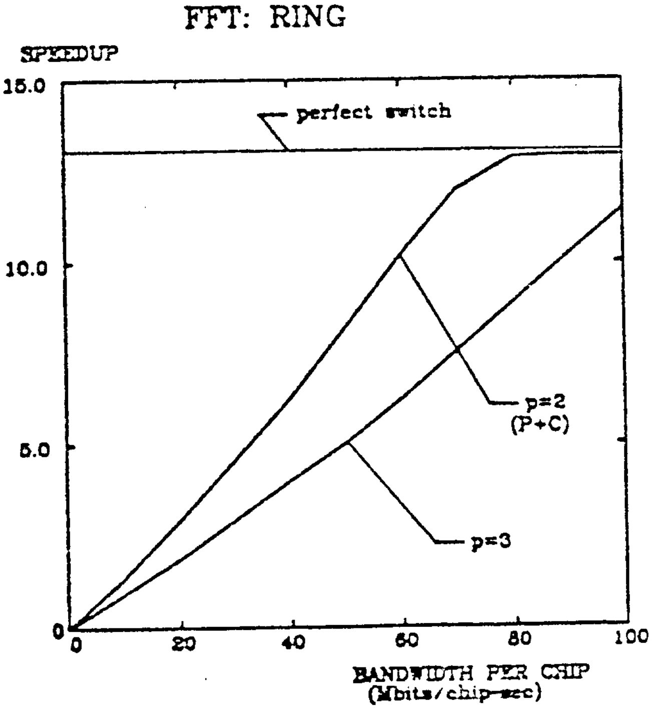


Figure 3.13. *Ring (a) Barnwell. (b) Block I/O.*



(c)



(d)

Figure 3.13. *Ring (c) Block State. (d) FFT.*

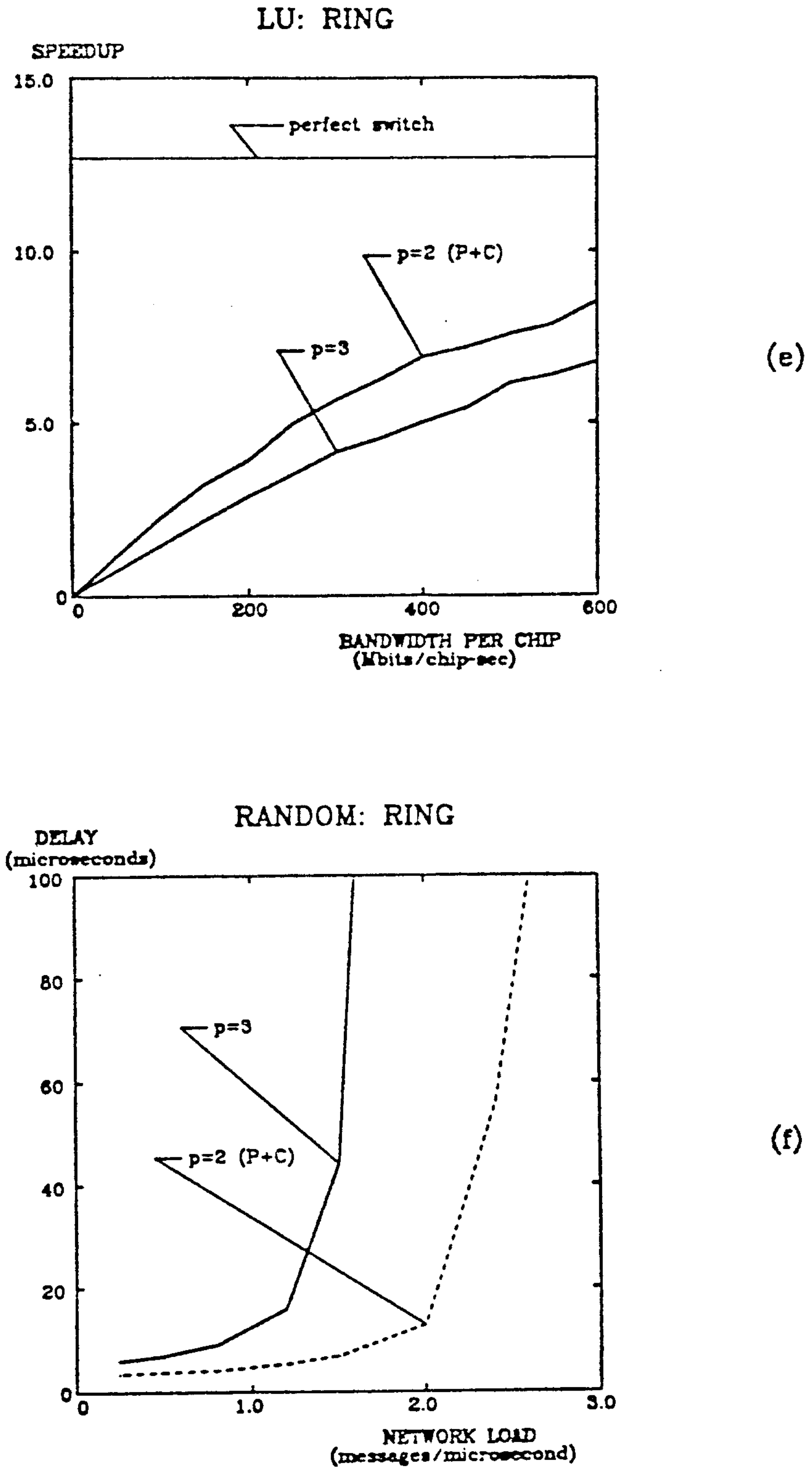


Figure 3.13. *Ring (e) LU. (f) Random.*

although they do increase congestion. In general, a priority mechanism could be used to avoid this anomaly: traffic generated by the computation processors can be assigned a higher priority than the input messages. At low bandwidths, the performance of the communication component networks is limited by the bandwidth of the link from the input processor to its communication component, allowing networks with processor and communications on the same chip to yield better speedup.

The "blocked traffic" behavior described above is not as prominent in the other SISO programs, Barnwell and Block State. In Block State, the traffic within the pipeline exhibits enough locality that it can avoid the congested area around the input processor. In Barnwell, the input message traffic is single destination, in contrast to Block State where the input traffic is multiple destination, and thus does not create as much congestion. The "jump" in performance around 110 Mbit/chip in one of the Barnwell curves is caused by a fortuitous shift in the traffic pattern which causes an unusual reduction in queueing delays along one of the links; it does not reflect any general principles of behavior.

3.6.5. Conclusions for Cluster Node Networks

The simulation results for cluster node networks are in agreement with the analytical results presented earlier. Networks constructed from components using a small number of ports yield less delay than networks using components with many ports. Bandwidth can be increased by adding more components to the communication domain. Eventually, as network bandwidth is increased, the rate at which processors can generate traffic limits performance, rather than the bandwidth of the network. Also, the programs using multiple-destination communications show a significant performance improvement if the communication circuitry includes a multicast mechanism.

3.7. Simulation Results on Networks with a Fixed Number of Components

Cluster nodes constructed from components with a large number of ports require fewer components than those constructed with a small number of ports. Thus, the studies presented above do not consider chip count. In this section, networks using the same number of components are considered. The application programs are executed on lattice and tree topology networks like those analyzed earlier. It will be seen that bottlenecks form around the root of the tree networks, biasing the results to favor components with a small number of high bandwidth links. De Bruijn networks are examined as an example of a class of network topologies with logarithmic average hop count, but without this inherent bottleneck.

The analytical results indicated that networks constructed from components with a small number of ports yielded lower delay, but less bandwidth than networks using components with a large number of ports. Based on these results, one would expect networks using a large number of ports to yield better performance when the network is bandwidth limited. Intuitively, as we move toward networks with a larger number of (slower) links, the average hop count is reduced, and additional paths are created in the network. These trends combine to reduce traffic on congested links. If the reduction in congestion is significant, it will more than offset the disadvantage of using slower links, and overall performance improves. Of course, if the network provides adequate bandwidth for the traffic load presented to it, then the queueing delays will be small, and networks using a larger number of ports can only achieve poorer performance since link speed is reduced. Thus, networks with a large number of ports can be expected to provide better performance when the traffic load is heavy relative to total network bandwidth, but networks with a small number of ports can be expected to perform better otherwise.

3.7.1. Lattice Topologies

The application programs were run with switch models for the lattice topologies shown in figure 3.14a-c. Performance curves are shown in figures 3.15a-f. The FFT program exhibits better performance with a large number of ports, as would be expected in bandwidth-limited networks. The remaining programs. however, indicate little performance variation as the number of ports is varied, or better performance with a small number of ports. One reason for this is that most of the programs encounter bottlenecks which are not alleviated when the number of ports, and thus the number of paths through the network, is increased. In the SISO programs for example, the bottleneck is around the input processor, and performance is determined to a large extent by the speed of the communication links around this congested area. Since components with a small number of ports use faster links, they achieve better performance.

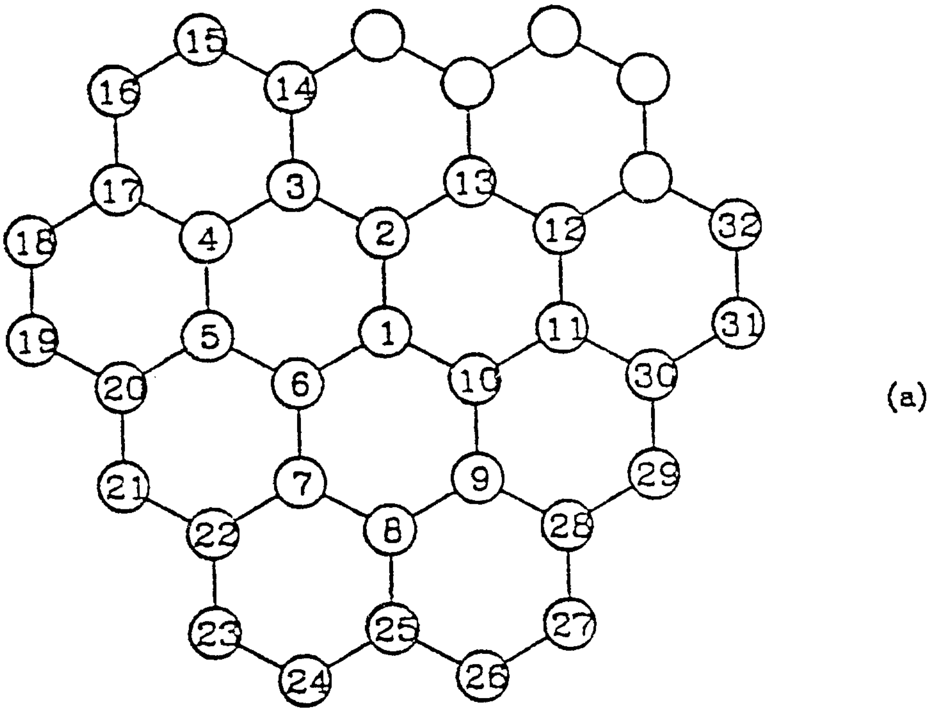


Figure 3.14. *Lattices (a) 3 ports.*

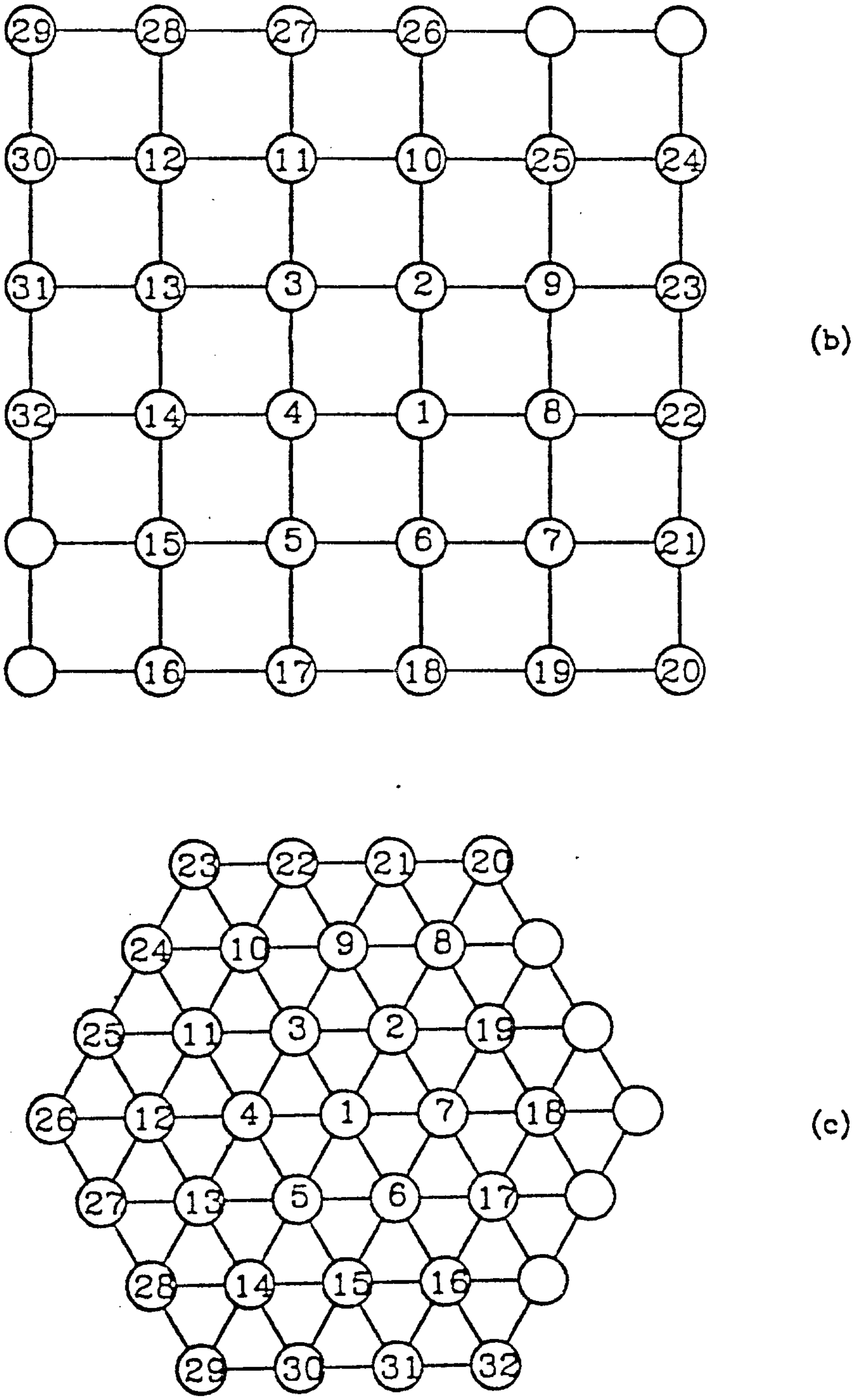


Figure 3.14. *Lattices (b) 4 ports. (c) 6 ports.*

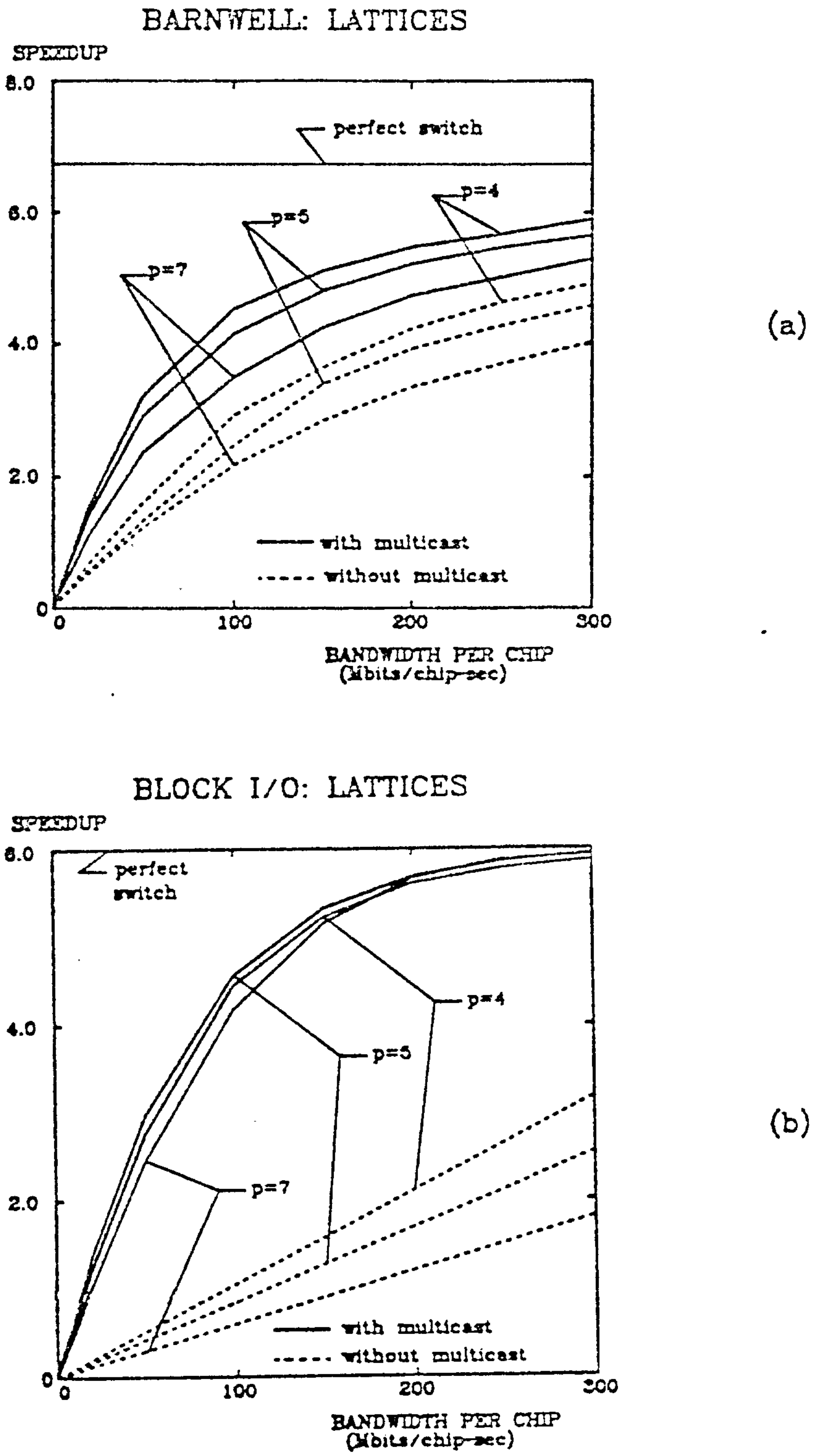


Figure 3.15. *Lattices (a) Barnwell, (b) Block I/O.*

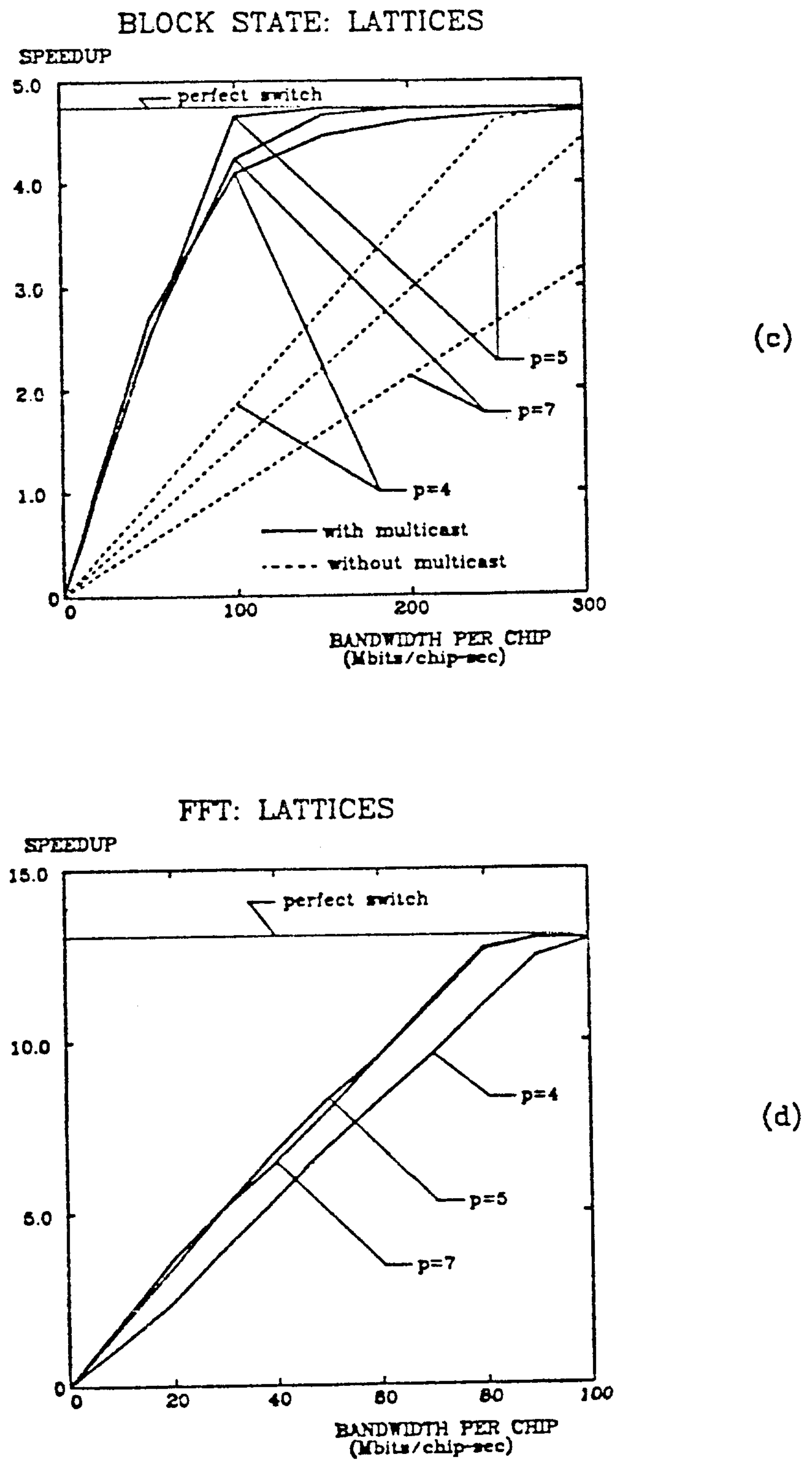


Figure 3.15. *Lattices (c) Block State. (d) FFT.*

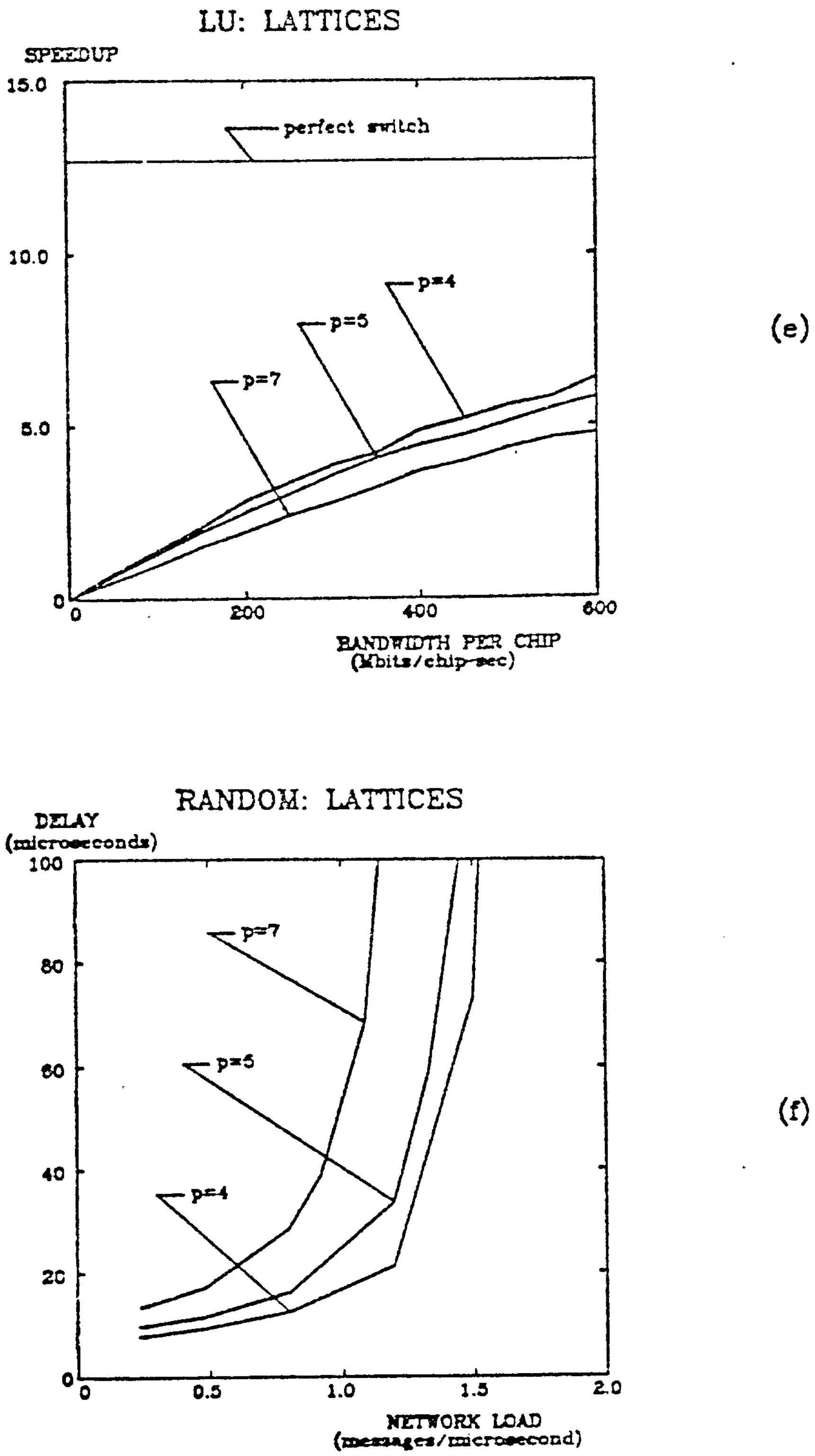


Figure 3.15. *Lattices (e) LU. (f) Random.*

The delay/bandwidth curves for the artificial traffic load program indicate that networks with a small number of ports achieve better delay and bandwidth. The bandwidth result disagrees with the analytical results presented earlier, which suggested that reduced hop counts would allow networks using components with a large number of ports to achieve higher throughput. The reason for the disagreement is that for high traffic loads, the processor/communication component link becomes a bottleneck. Performance is thus determined by the speed of this bottleneck link.

In figures 3.16a-f, this bottleneck is removed by assuming that communication circuitry is integrated onto the same chip as the processor. The curves for the artificial traffic load program are in closer agreement with the analytic results presented earlier, however, the results for the other application programs are qualitatively the same. It is interesting to note that some of the SISO programs, Block State and Block I/O in particular, experience lower performance when this latter model is used. This anomalous effect can be attributed to the "blocking problem" described earlier in the ring topology discussion. Messages that carry the input samples block traffic generated by the computation processors.

Finally, the convergence of some of the multicast/non-multicast curves in the Block State program is one other point of interest. Recall that Block State uses multicast to distribute the initial data samples. The convergence of the curves indicates that beyond a certain bandwidth, here approximately 100M bit/chip, the network can provide computation processors with data samples as quickly as they can be processed, so improved performance along these virtual circuits results in no improvement in overall execution time. In addition, the network provides enough bandwidth that the additional traffic caused by the absence of a multicast mechanism does not degrade performance.

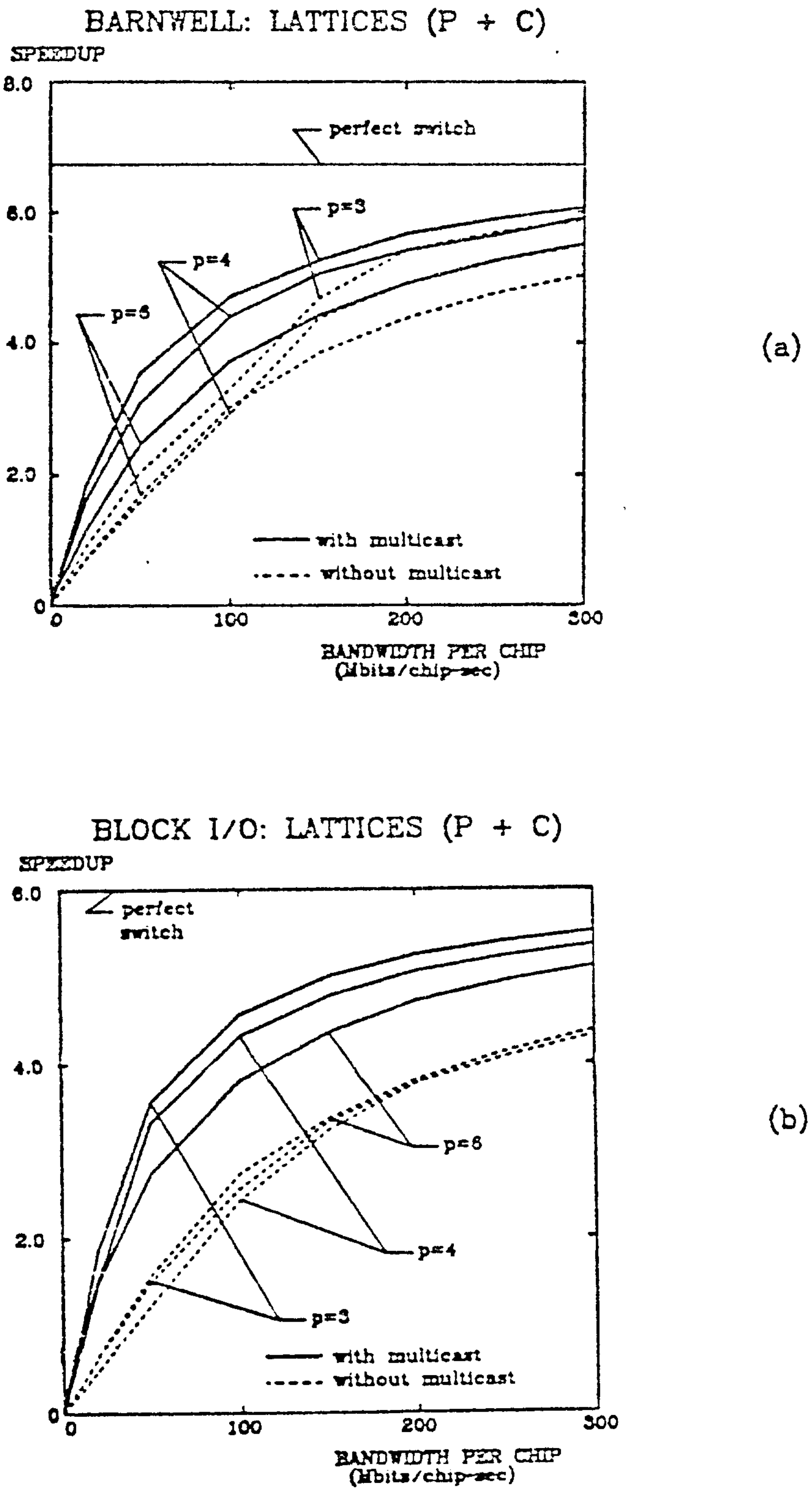


Figure 3.16. *Lattices (P+C) (a) Barnwell. (b) Block I/O.*

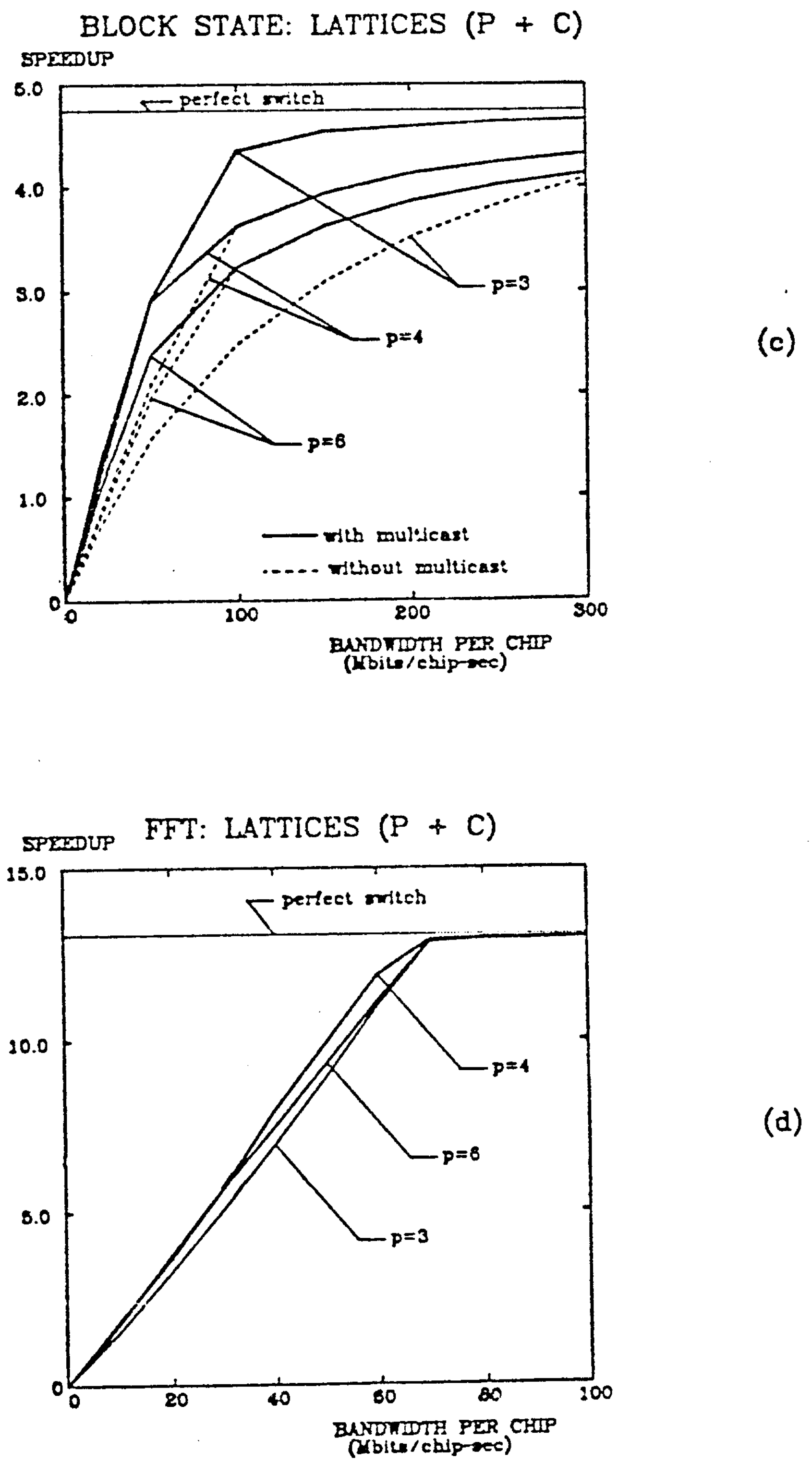


Figure 3.16. *Lattices (P+C) (c) Block State. (d) FFT.*

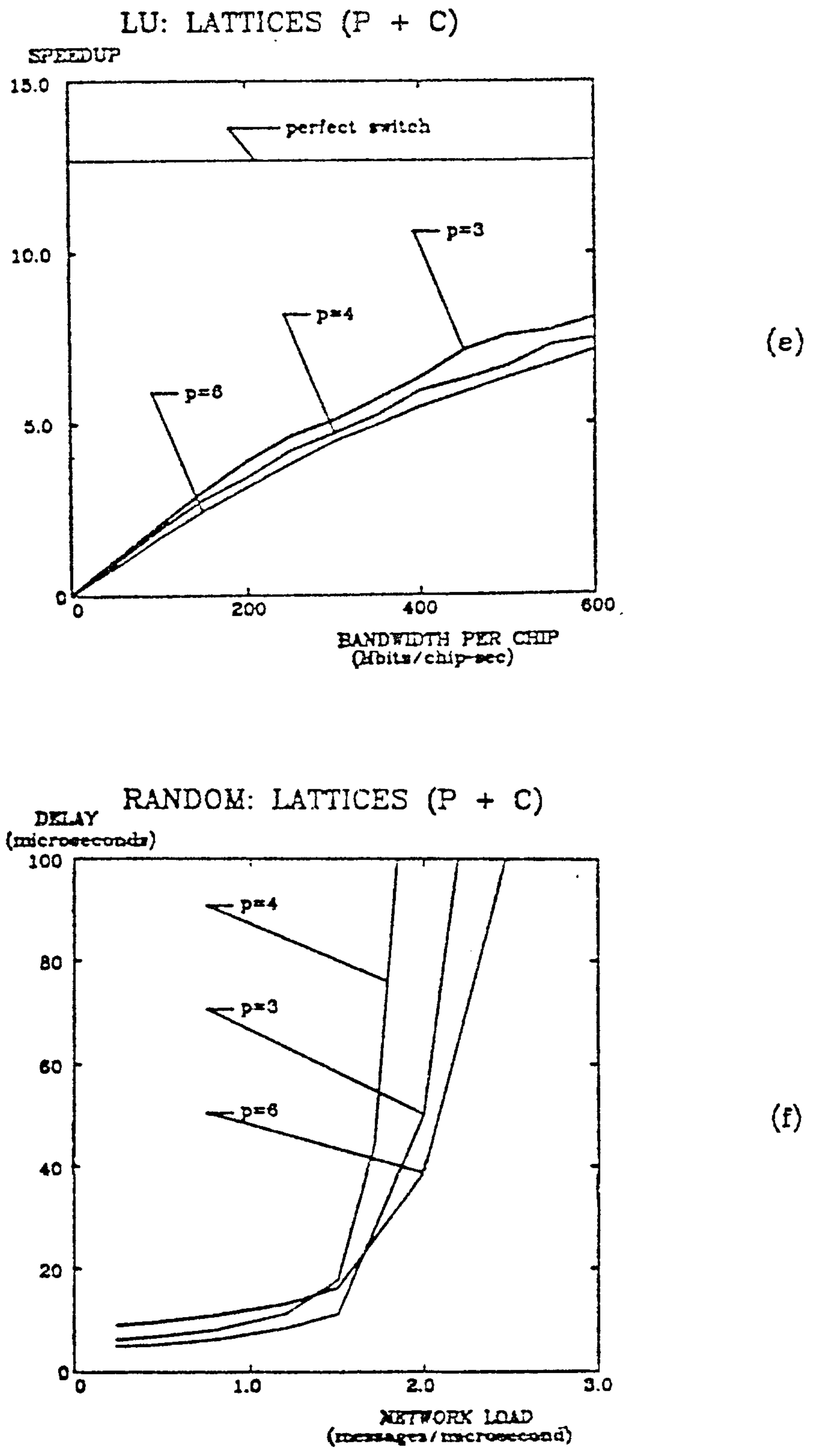


Figure 3.16. *Lattices (P+C) (e) LU. (f) Random.*

3.7.2. Tree Topologies

Performance curves for tree networks (figure 3.17 shows one such network) are shown in figures 3.18a-g. For all application programs, it is seen that networks built from components with a small number of ports yield better performance than those using a larger number of ports, even when processor and communications are incorporated onto the same chip (see figure 3.18g). These results however, are a consequence of congestion around the root node rather than from the hop count/link bandwidth tradeoffs discussed earlier. In trees, a disproportionate amount of traffic must flow through the root, leading to congestion in this portion of the network. Increasing the number of links does not improve the amount of bandwidth allocated to this congested area. As a result, performance is determined to a large extent by the speed of communication links near the root. Since components with a small number of ports have faster links, they yield higher performance.

3.7.3. De Bruijn Networks

The results for tree topologies were biased because of the inherent bottleneck around the root. To provide a true test of the analytical results, a class of topologies is required which does not have this inherent bottleneck, but which also has an average hop count which grows logarithmically with the number of nodes. The class of topologies must be general to the extent that networks with approximately the same number of nodes can be constructed as the number of ports is increased.

One class of topologies which satisfy these requirements are De Bruijn networks [Brui46]. De Bruijn networks, which are only defined for even degree (i.e. an even number of links per node), are the densest known infinite family of undirected graphs of even degree greater than 4. A dense graph of degree p is one with a small diameter. Diameter, which is specified as a function of p and

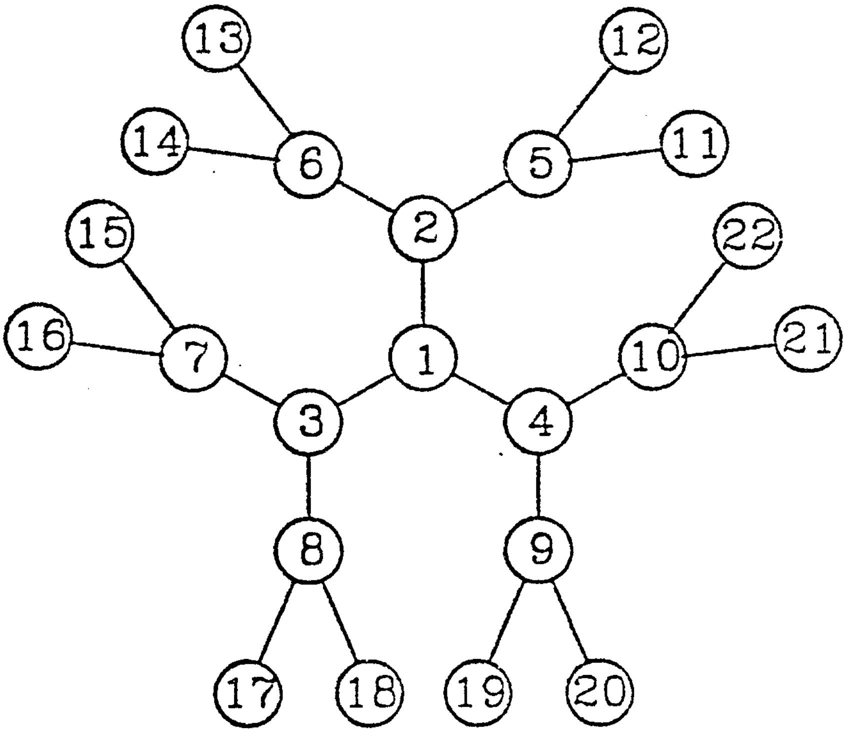


Figure 3.17. *Tree topology.*

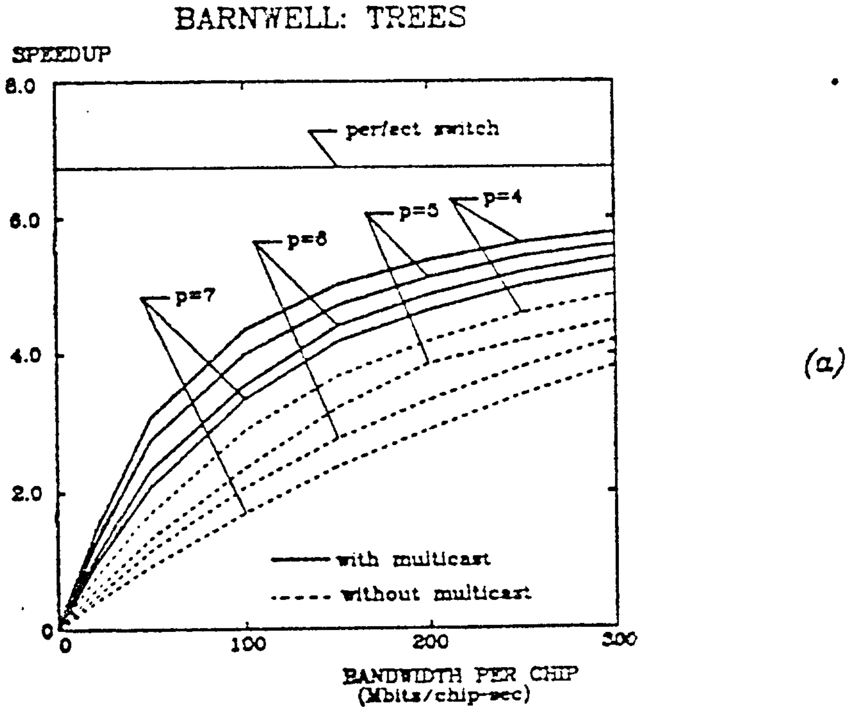


Figure 3.18. *Trees (a) Barnwell.*

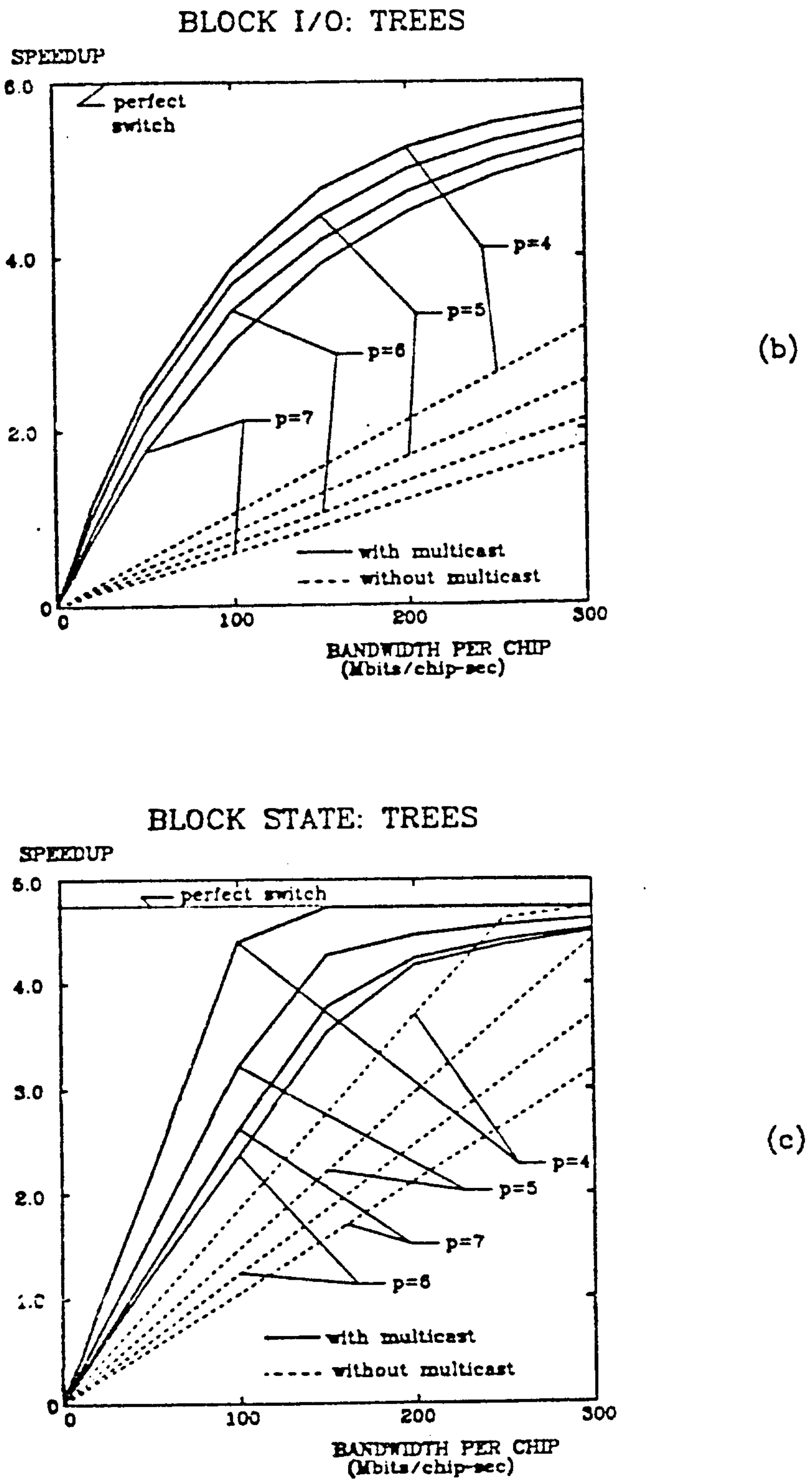


Figure 3.18. *Trees (b) Block I/O. (c) Block State.*

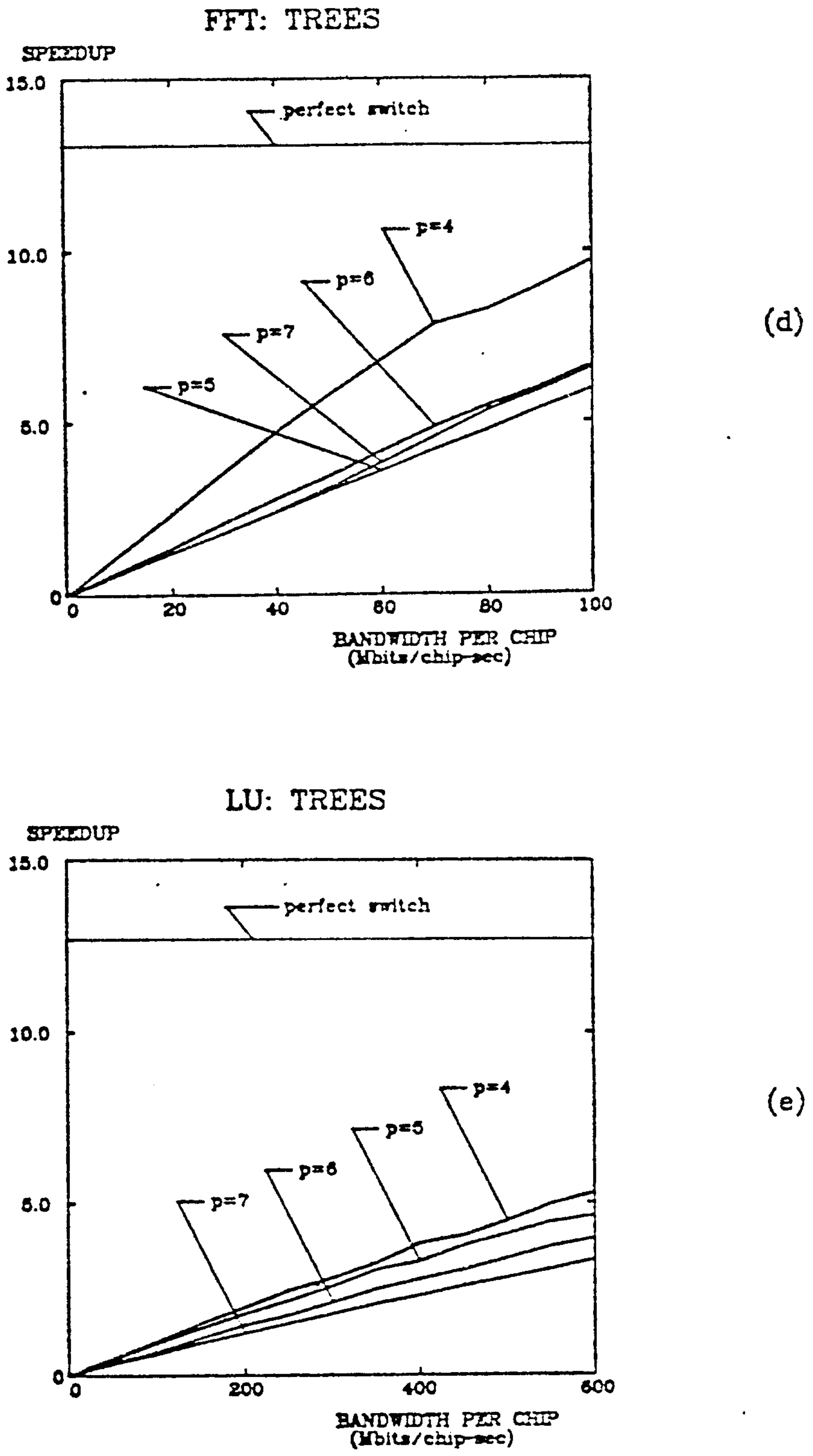
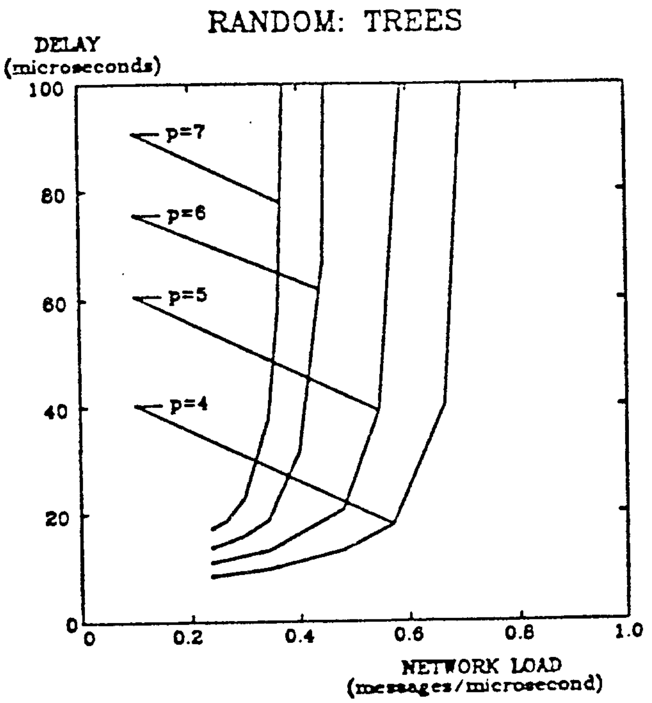
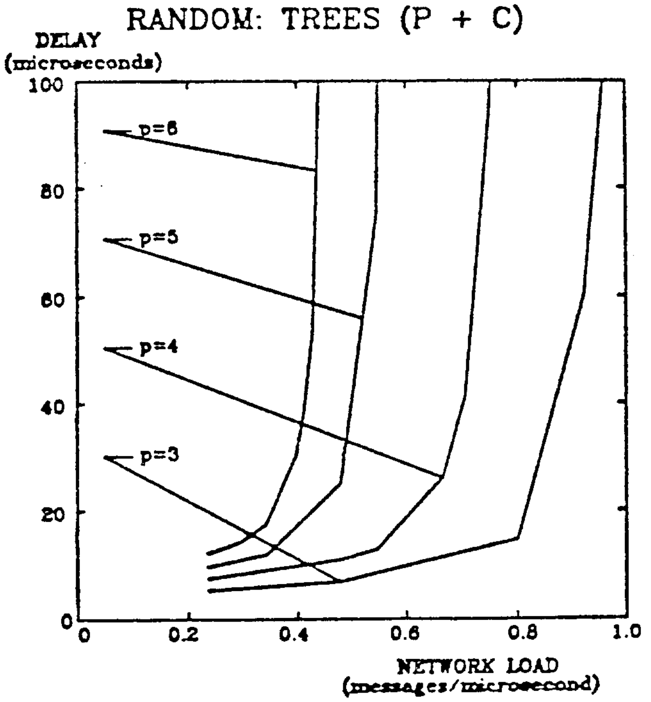


Figure 3.18. *Trees (d) FFT. (e) LU.*



(f)



(g)

Figure 3.18. *Trees (f) Random. (g) Processor with communications.*

the number of nodes in the graph, is defined as the largest distance between any pair of nodes, where distance refers to the length of the shortest path between the two nodes. Until recently, De Bruijn graphs were not only the densest family of graphs of degree greater than 3, but De Bruijn graphs of degree p were also denser than any other family of graphs of degree $p+1$. Recently however, denser graphs have been discovered for degrees 3, 4, 5 [Lela82a, Lela82b]. Also, C_S' graphs, which are only defined for odd degrees greater than 3, yield smaller diameter than De Bruijn graphs with one fewer port per node [Farh81]. Still, the De Bruijn networks represent a set of graphs with logarithmic hop count without the "root bottleneck" inherent in trees, and thus represent an attractive topology for analyzing the optimum number of ports.

A De Bruijn graph is characterized by two parameters, a base b and an integer n. The graph consists of b^n nodes. The address of each node is defined by a string of digits, $x_0x_1 \cdots x_{n-1}$, where $0 \leq x_i < b$. The addresses of nodes which are directly connected to X are derived by shifting X's address left or right 1 digit, and shifting in a new digit k, $0 \leq k < b$. Thus, node X has links to nodes $yx_0x_1 \cdots x_{n-2}$ and nodes $x_1 \cdots x_{n-1}y$, where $y = 0, 1, \cdots b-1$. Each node has up to $2 \times b$ links to other nodes. From this definition, it is clear that node X can reach any other node in at most n hops, since an arbitrary address can be generated by shifting the X address n times. The topology does contain some degenerate cases. For example, with $b=2$, nodes $00 \cdots 0$ and $11 \cdots 1$ have links to themselves, and nodes 0101..., and 1010... have more than one link between them. These are the only special cases however. The edges of the De Bruijn graph yield exactly the same interconnection as the permutation network sometimes called the single-stage shuffle-exchange [Ston71, Ston72]. A base 2, 8 node network is shown in figure 3.19.

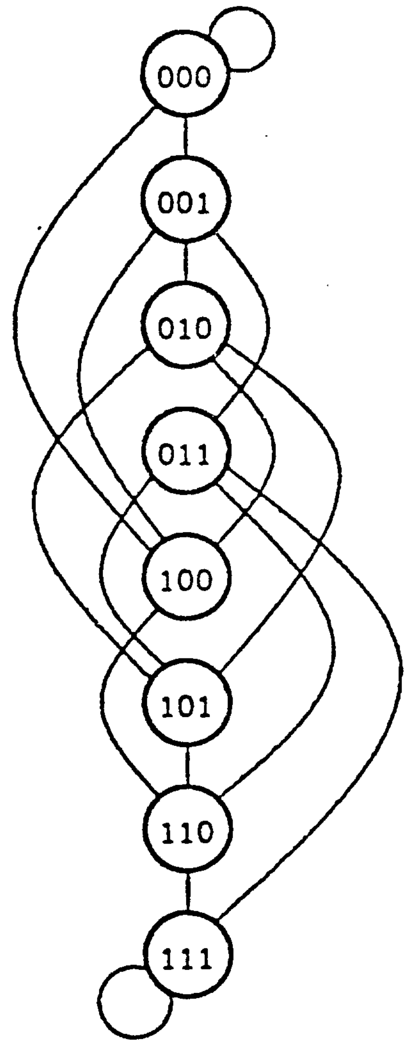


Figure 3.19. *Base 2 De Bruijn network.*

For this study, three De Bruijn graphs were examined:

(1) $b=2$, $n=5$ (32 nodes)

(2) $b=3$, $n=3$ (27 nodes)

(3) $b=5$, $n=2$ (25 nodes).

These graphs were selected since they have roughly the same number of nodes, and also provide enough processors to execute most of the application programs (the FFT is the only one requiring more than 25 processors). Communication components for these graphs require 5, 7, and 11 ports for each node, respectively, including one port to attach to the node's computation processor, providing a wide range in values for p.

The performance curves for the De Bruijn networks described above are shown in figures 3.20a-e. Performance with processor and communication circuitry integrated onto the same chip are shown in figures 3.21a-e. The results are qualitatively similar to those of the lattice topologies. The curves indicate that better performance is achieved when components with a small number of ports are used.

3.7.4. Conclusions for Networks with a Fixed Number of Components

The primary result of these simulation studies is that networks constructed with components using a small number of ports achieve better performance than those using a large number of ports. In some cases, this is in disagreement with the results of analytical studies. This is normally due to bottlenecks that prevent much of the bandwidth provided by the network to be utilized. These bottlenecks can be alleviated by using components with a small number of ports, since this provides maximum bandwidth for the concerned links. The bottlenecks may arise from the application program (e.g. the SISO programs here), or from the network topology (e.g. trees). The limited I/O bandwidth of the processors generating messages may be the source of another bottleneck. Finally, the simulations also demonstrate that significant performance improvements can be achieved if mechanisms are included for efficient handling of multiple-destination messages.

3.8. Influence of the Mapping of Tasks to Processors

The results described above assumed a specific algorithm, to be discussed below, for mapping application programs onto the network topologies. Care

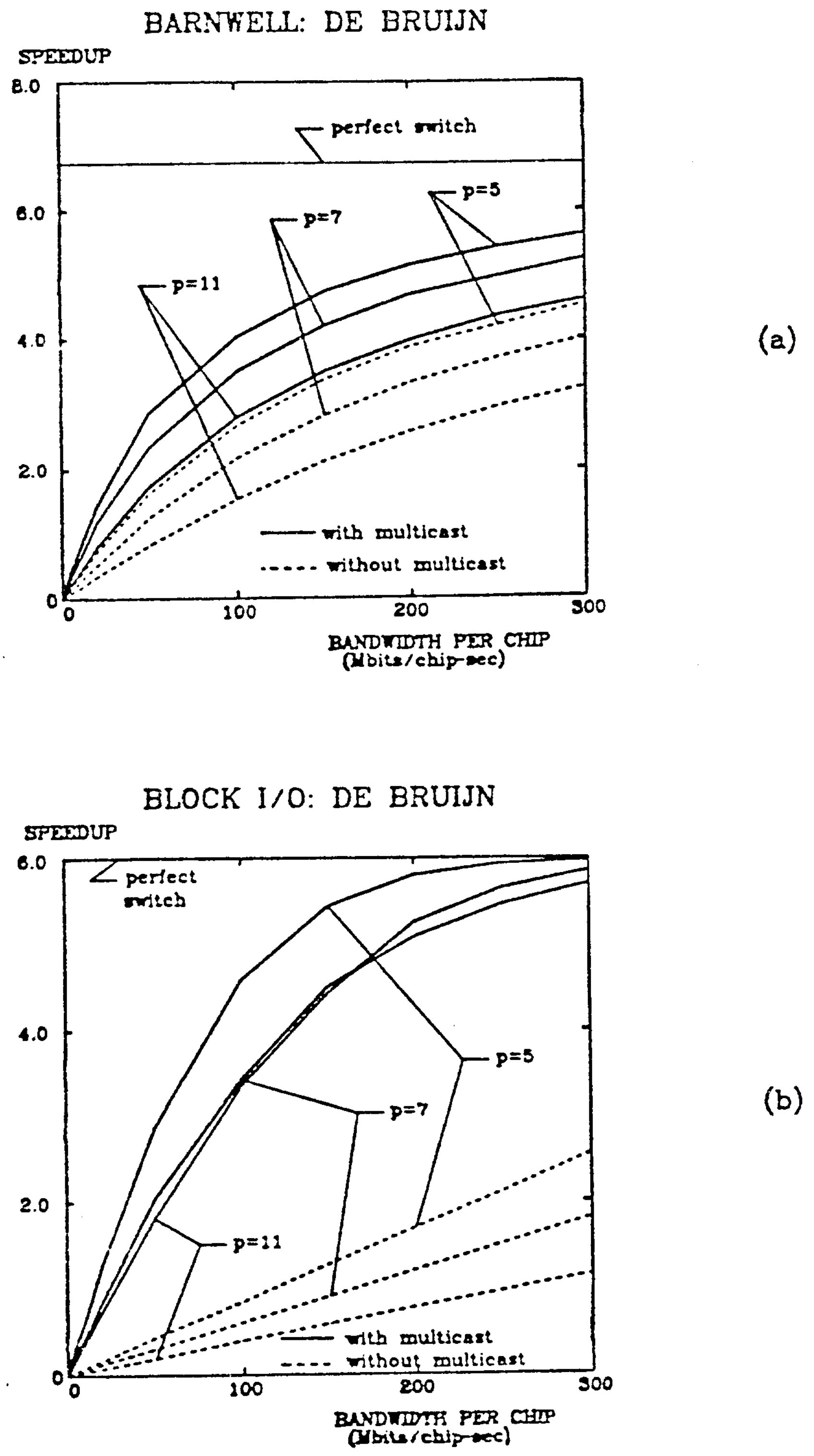


Figure 3.20. *De Bruijn network (a) Barnwell. (b) Block I/O.*

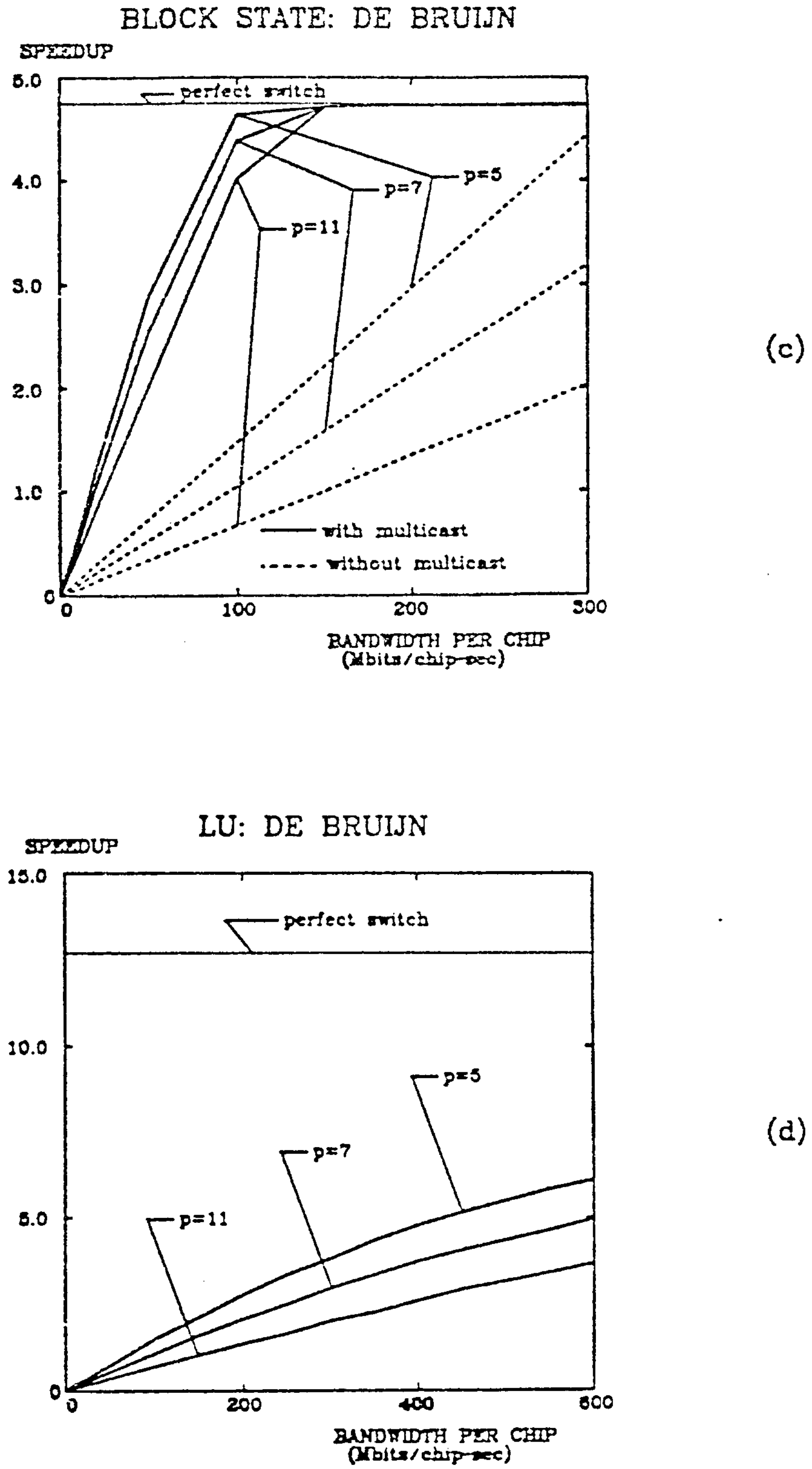
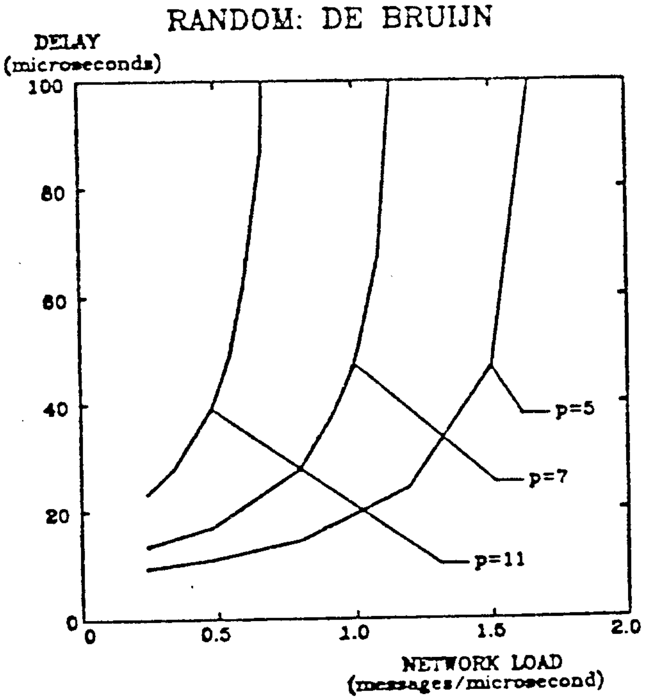
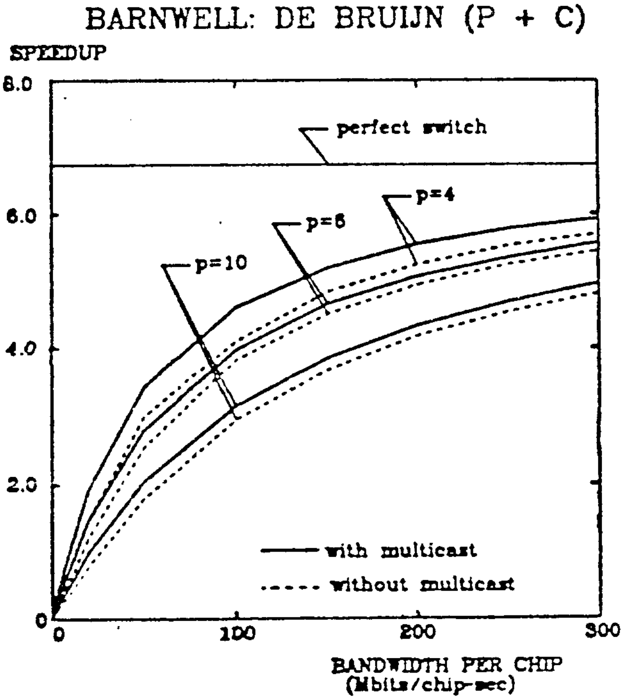


Figure 3.20. *De Bruijn network (c) Block State. (d) LU.*



(e)

Figure 3.20. *De Bruijn network (e) Random.*



(a)

Figure 3.21. *De Bruijn network (P+C) (a) Barnwell.*

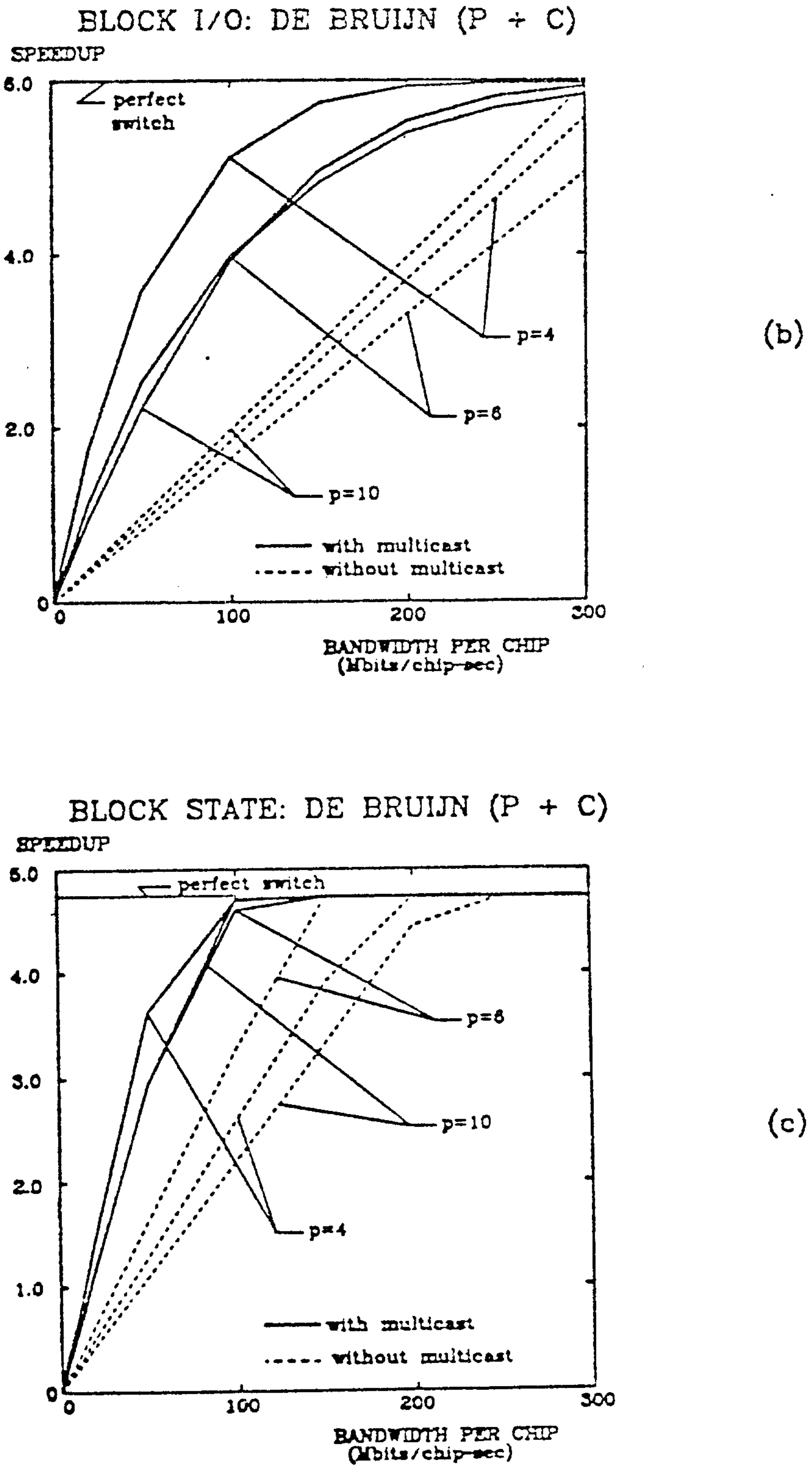


Figure 3.21. *De Bruijn network (P+C) (b) Block I/O. (c) Block State.*

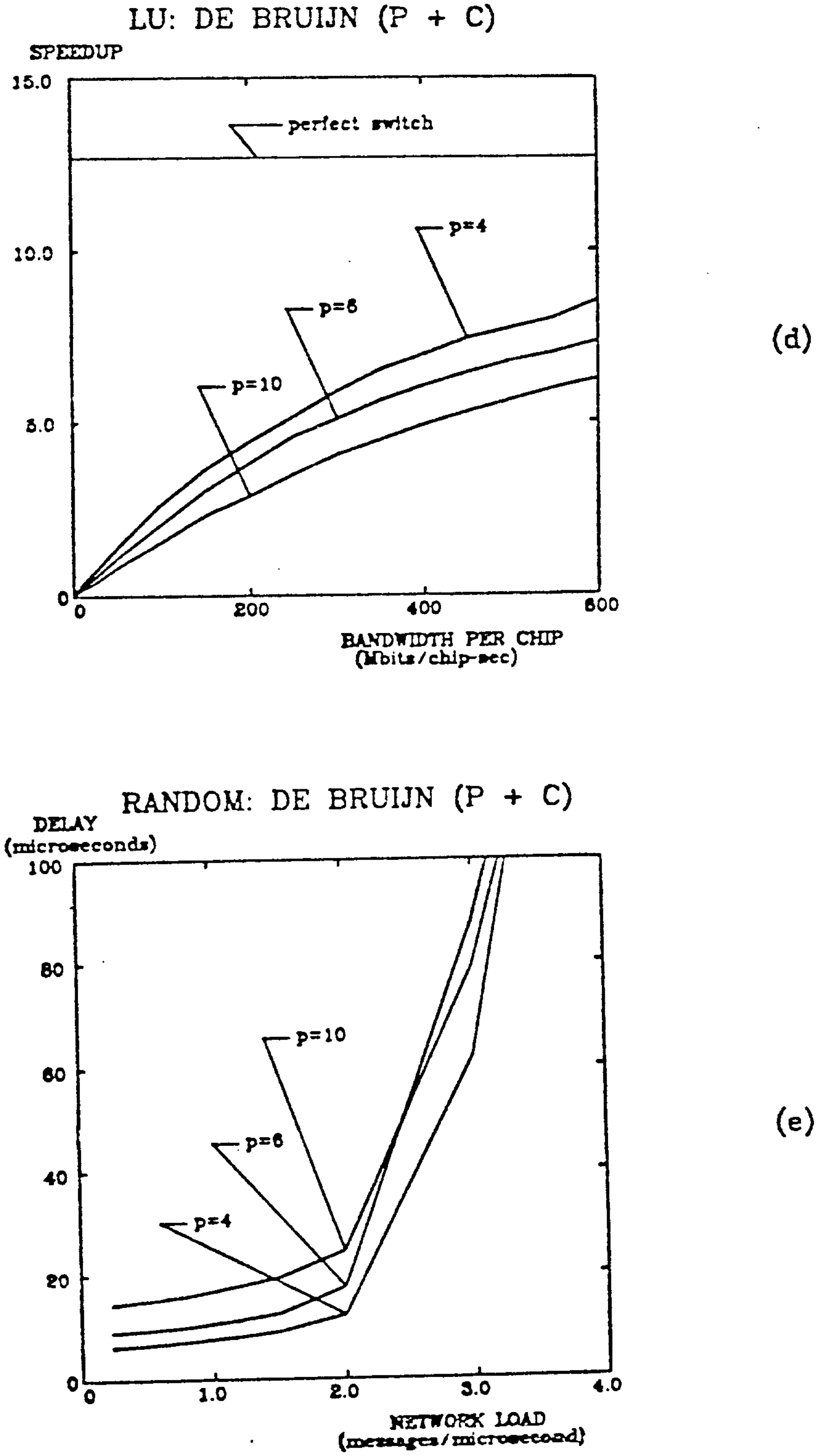


Figure 3.21. *De Bruijn network (P+C) (d) LU. (e) Random.*

must be taken to ensure that this mapping algorithm is "equally good" for the networks being compared, or else the differences between the curves may just be a result of using a better mapping in one network relative to another. This section addresses the question of how the quality of the mapping algorithm affects the results presented above.

The simulation results used two types of switch models. The first is based on the cluster node. However, cluster node networks are only an implementation of a given topology. Thus, within each topology, identical mappings are used, and no cluster node network is favored over another.

The second type of switch model compares networks with different topologies. Performance curvés for different types of lattices, trees, and De Bruijn networks are compared. In order to characterize the "goodness" of a mapping, a quality measure must be established. Changing the number of ports affects link speed and average hop count. Since the mapping algorithm has no impact on link speed, but does affect average hop count, the latter is an appropriate measure. In particular, as the number of ports is increased, the average hop count should decrease in a manner similar to that observed in the analytical studies. If it can be shown for each application program that the average hop count decreases "as it should", then it can be concluded that the mapping algorithm does not bias the results to favor (say) lattices with a small number of ports. On the other hand, if increasing the number of ports creates an unexpectedly small (large) improvement in average hop count, then a better mapping was done on the network with a small (large) number of ports, weakening (strengthening) the conclusion that a small number of ports is better.

Average hop count is defined for an application program a as:

$$\overline{H}_a = \frac{1}{\gamma}\sum_{i,j}\gamma_{ij}\,d_{ij}$$

where d_{ij} is the number of links traversed in the shortest path from i to j, and

γ is the total number of messages sent into the network. γ_{ij} is the total number of messages sent from i to j. This definition differs from that presented earlier because the earlier definition assumed a uniform traffic distribution, implying all paths have equal weight. Values for $\overline{H}_a$ are given in table 3.2 for the different application programs.

Table 3.2.
Average Hop Count

		Lattices			De Bruijn		
		p=3	p=4	p=6	p=4	p=6	p=10
BARNWELL	ant. (g)	3.442	3.079 10.55%	2.437 29.20%	2.605	2.026 22.23%	1.632 37.35%
	map 1	2.850	2.575 9.65%	1.962 31.16%	2.387	1.875 21.45%	1.587 33.51%
BLOCK I/O	l.b. (l)	1.870	1.652 11.66%	1.392 25.56%	1.739	1.565 10.01%	1.392 19.95%
	map 1	2.620	2.471 5.69%	2.092 20.15%	2.944	2.081 29.31%	1.627 44.74%
	map 2	3.827	3.827 0.00%	3.479 9.09%			
BLOCK STATE	l.b. (l)	1.333	1.222 8.33%	1.111 16.65%	1.278	1.222 4.38%	1.111 13.07%
	map 1	3.259	3.000 7.95%	2.314 29.00%	2.630	2.000 23.95%	1.370 47.91%
	map 2	2.649	2.482 6.30%	2.316 12.57%			
FFT	map 1	5.542	4.792 13.53%	3.625 34.59%			
	map 2	2.667	2.167 18.75%	1.917 28.12%			
LU	ant. (g)	3.442	3.079 10.55%	2.437 29.20%	2.605	2.026 22.23%	1.632 37.35%
	map 1	2.979	2.585 13.23%	2.044 31.39%	2.515	1.983 21.15%	1.638 34.87%
RANDOM	ant. (g)	3.442	3.079 10.55%	2.437 29.20%	2.605	2.026 22.23%	1.632 37.35%
	map 1	2.692	2.483 7.76%	1.911 29.01%	2.459	1.925 21.72%	1.556 36.72%

l.b. (l)	= lower bound from weighted average hop count, optimal packing
ant. (g)	= anticipated from 20 node network, global communications

Two mapping schemes were used in the application programs exhibiting local communications (Block I/O, Block State, and FFT). The first is the original mapping whose results were described in section 3.7. This mapping was designed to minimize the average hop count on virtual circuits carrying the initial data samples in the SISO programs. If the processors of the network are envisioned as being uniformly distributed across the surface of a disk, the processor distributing the data samples resides in the center, and the processors receiving these samples are packed around it.

The second mapping attempts to optimize the virtual circuits carrying the local, i.e. pipelined, communication traffic among the computation tasks. This mapping was performed for lattice topologies only. Figures 3.22a-e indicate which tasks are assigned to which processors for this latter mapping. The FFT program uses the same task number assignments that are shown in figure 3.6. Figures 3.23a-f are the performance curves for these mappings for lattices with communication components and lattices with processor and communications integrated onto the same chip. Except for the FFT program, which will be discussed later, the results are qualitatively the same as those found in the original mapping.

Also included in table 3.2 are "anticipated" values for $\overline{H}_a$. For programs using global communications (Barnwell, LU, and Random), this anticipated value is computed by examing the average hop count in a 20 node network, assuming a uniform traffic distribution. For programs exhibiting local communications however (Block I/O, Block State, and FFT), this is clearly an unrealistic measure. Here, a lower bound value for anticipated hop count is used. Suppose the application program requires that processor X must send messages to k other processors. The minimum hop count from X to these processors is obtained by assuming that the k processors communicated with are those which are closest

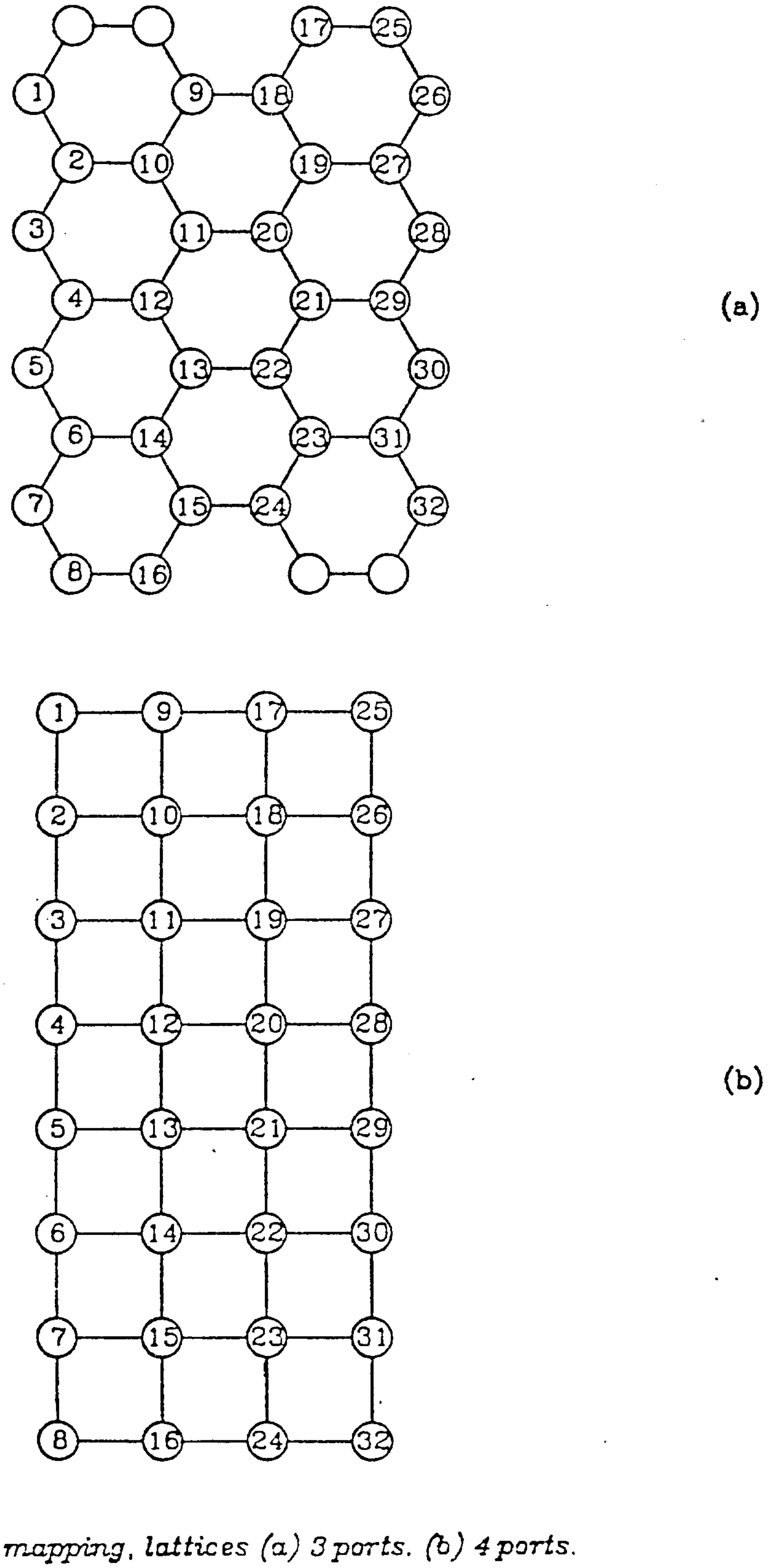
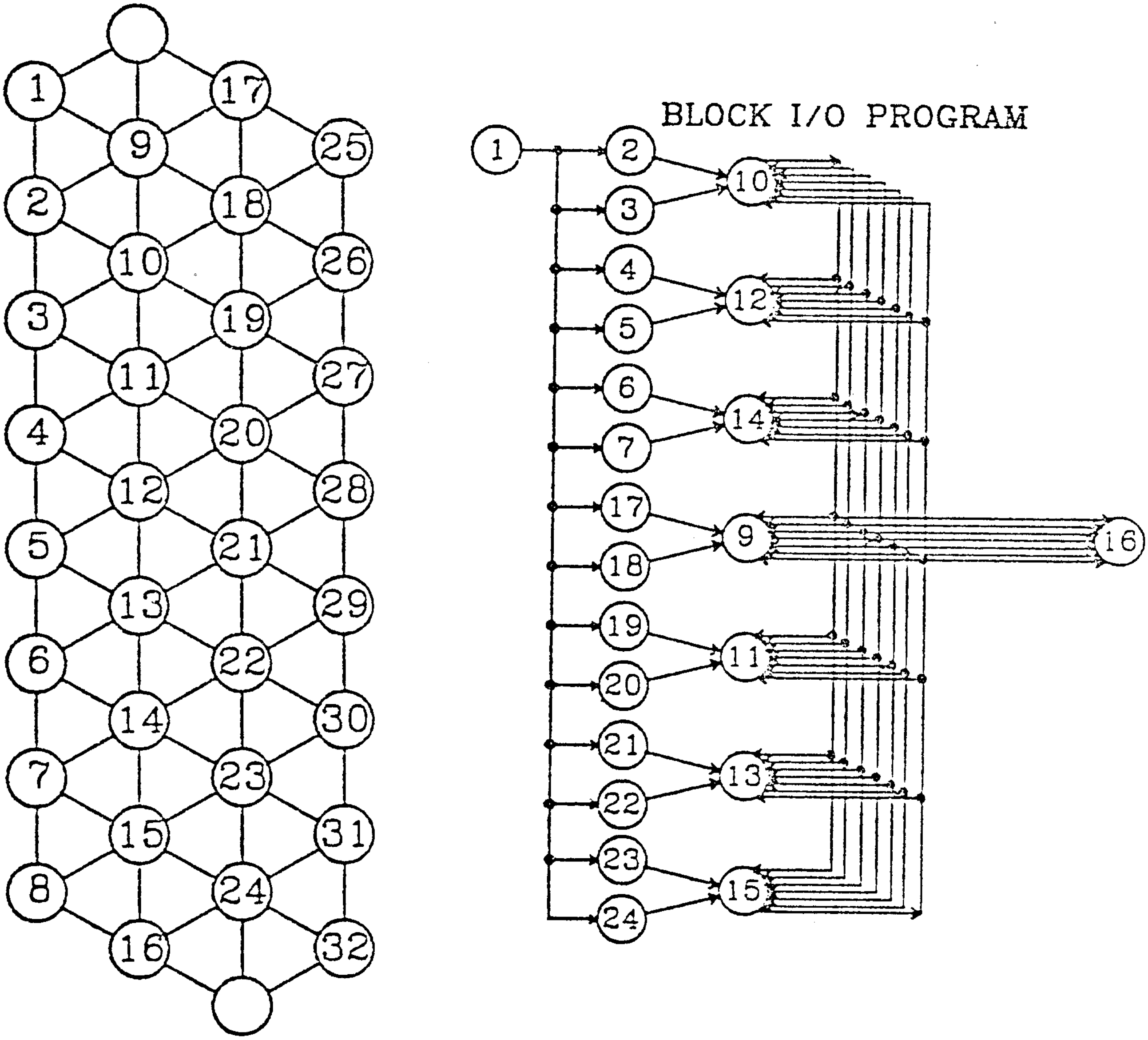


Figure 3.22. *Second mapping, lattices (a) 3 ports. (b) 4 ports.*



(c)

(d)

Figure 3.22. *Second Mapping (c) 6 ports. (d) Block I/O program.*

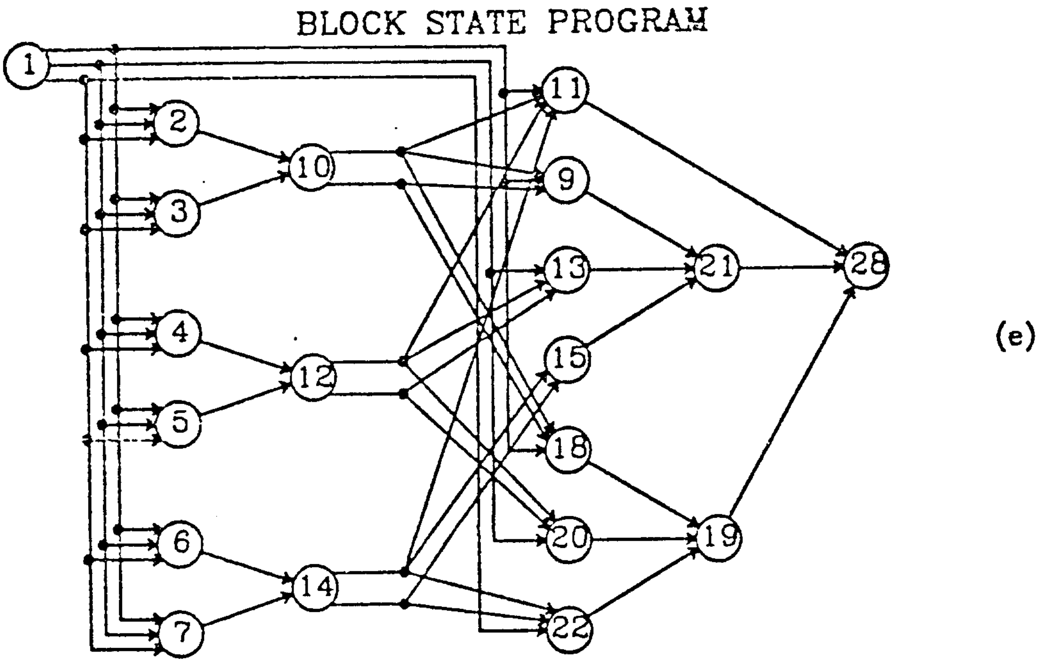


Figure 3.22. *Second Mapping (e) Block state program.*

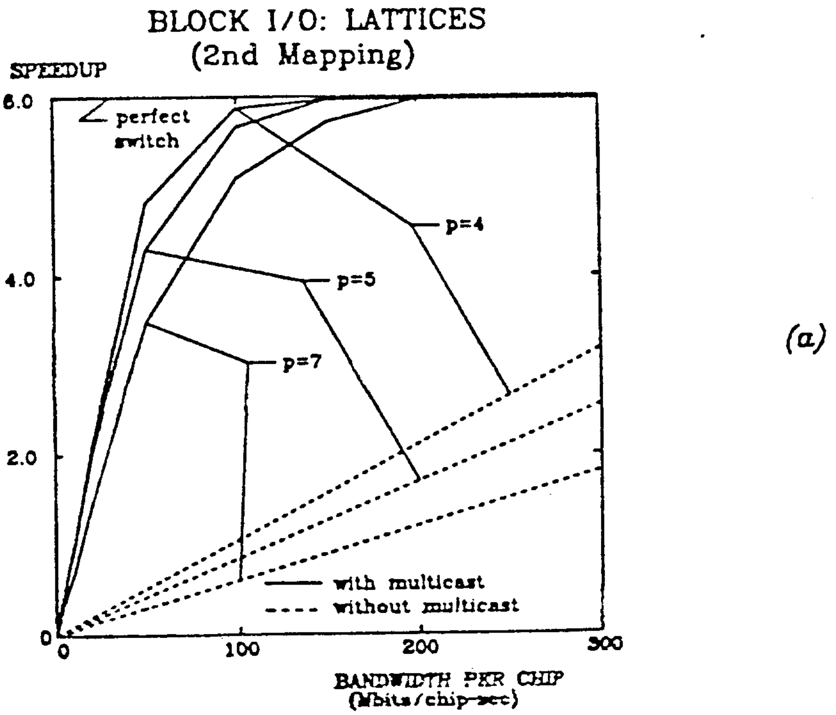


Figure 3.23. *Lattices (2nd mapping) (a) Block I/O.*

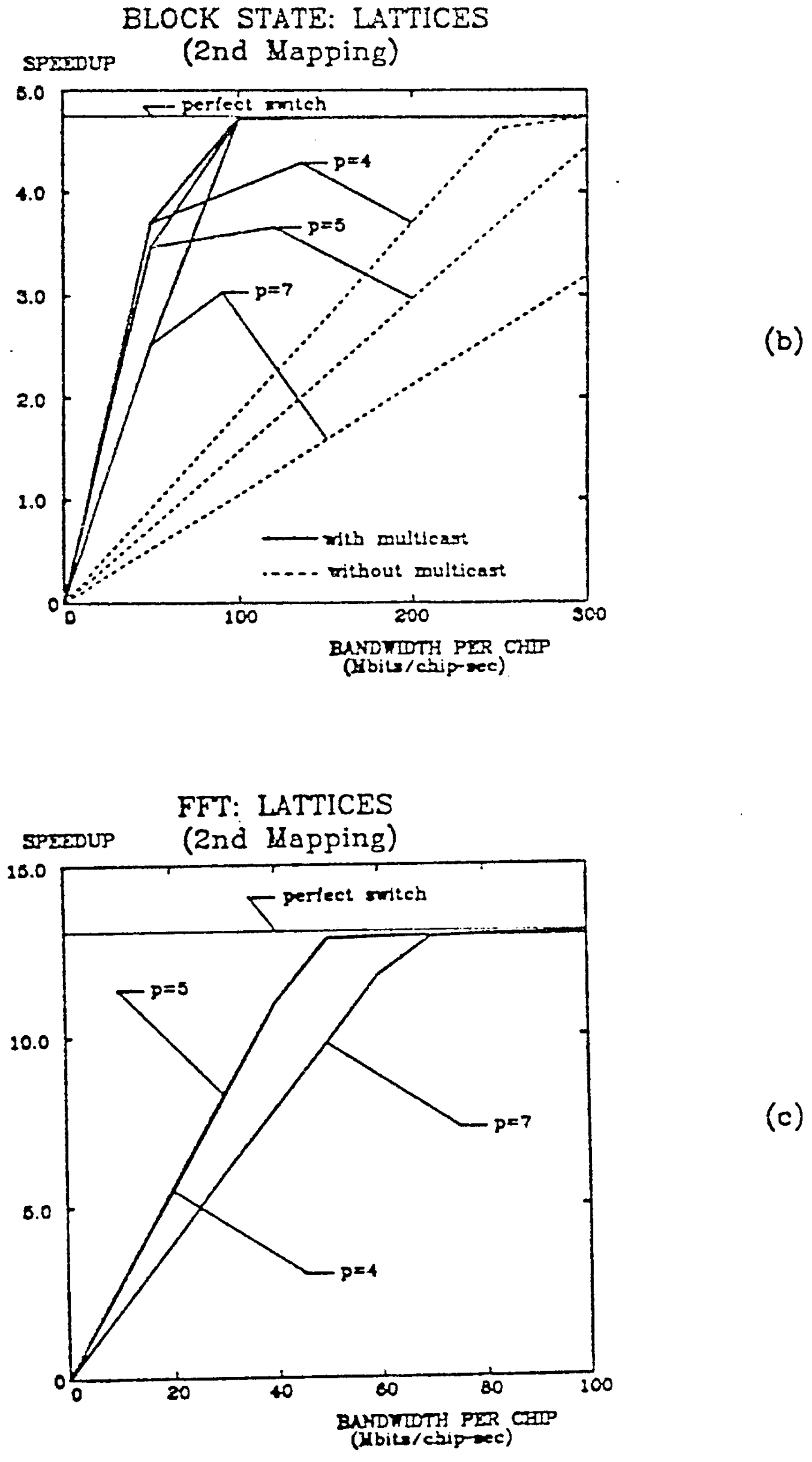
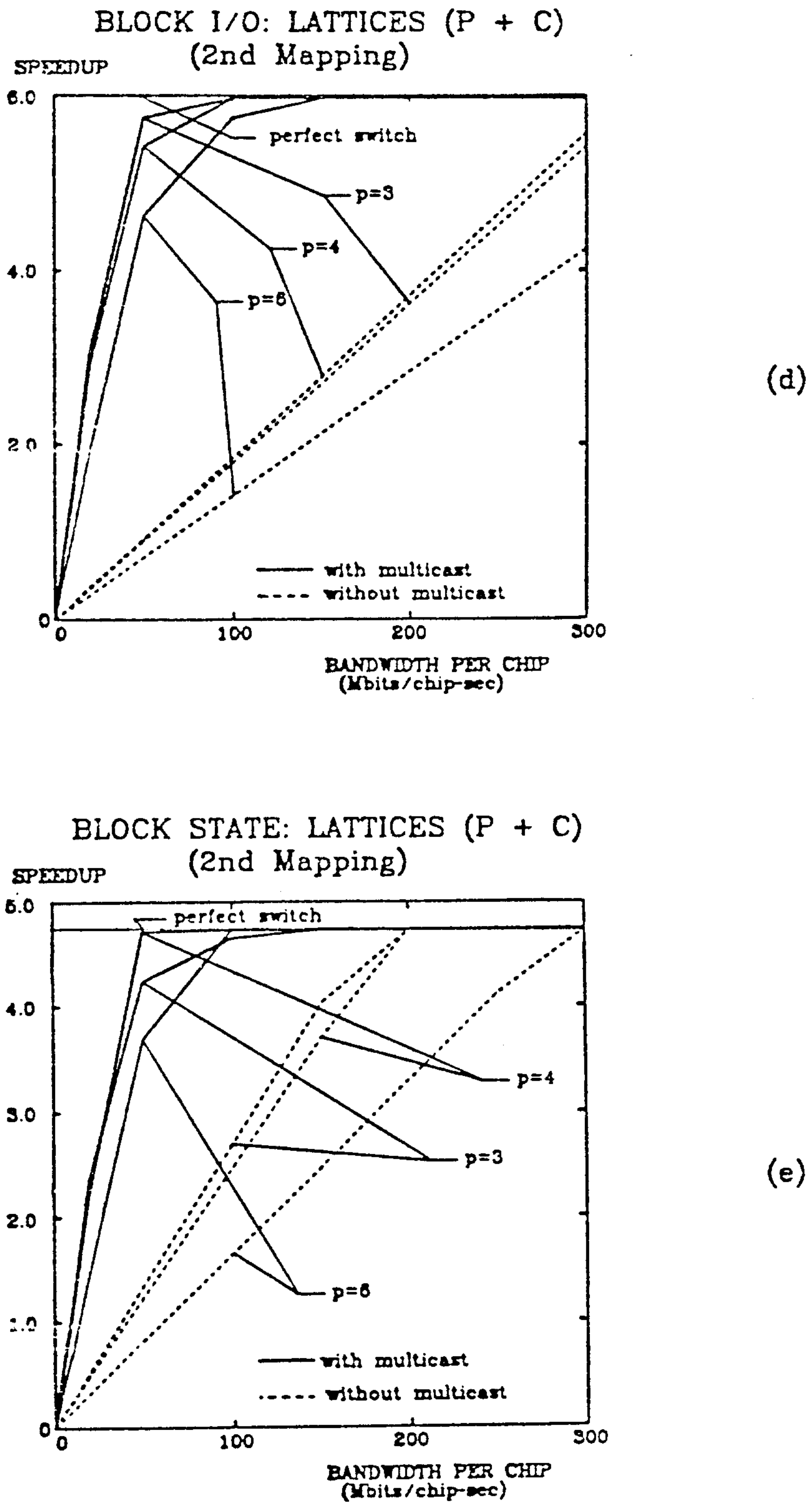


Figure 3.23. *Lattices (2nd mapping) (b) Block State. (c) FFT.*



BLOCK I/O: LATTICES (P + C) (2nd Mapping)

BLOCK STATE: LATTICES (P + C) (2nd Mapping)

Figure 3.23. *Lattices (2nd mapping, P+C) (d) Block I/O. (e) Block State.*

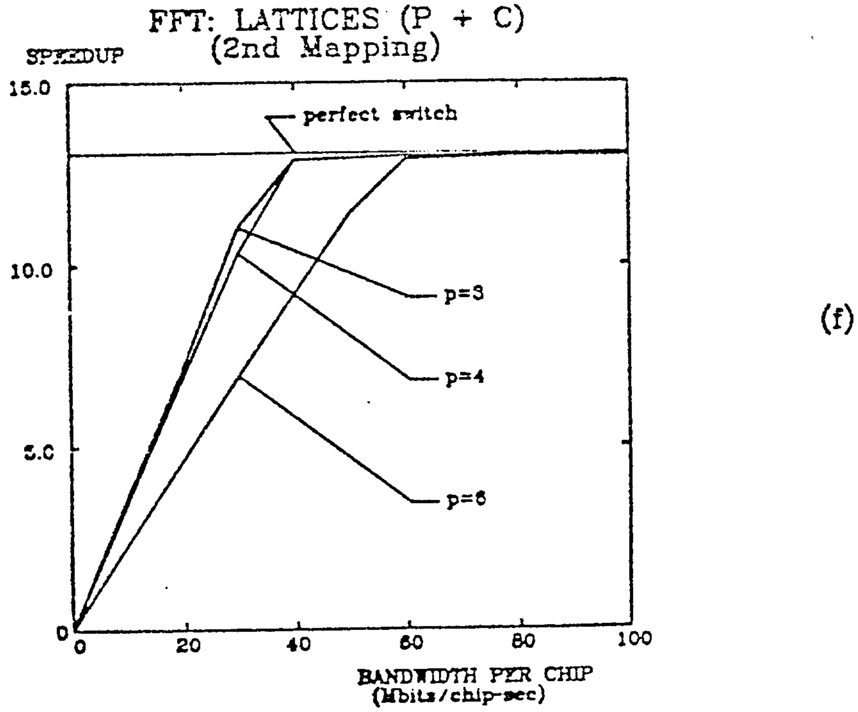


Figure 3.23. *Lattices (2nd mapping, P+C) (f) FFT.*

to X. By repeating this computation for each processor, a lower bound on $\overline{H}_a$ is obtained. This is the traffic distribution assumed in the analytical model presented earlier (networks with a fixed number of components), except the uniform traffic distribution assumption has been relaxed. In table 3.2, no anticipated value is listed for the FFT program since each task communicates with only 2 other tasks, leading to $\overline{H}_a$ = 1 for all topologies.

Table 3.2 also includes fractional improvements in average hop count relative to the network using the minimum number of ports, (p =3 for lattices, p =4 for De Bruijn) as the number of ports per chip is increased. In comparing the experimentally measured results with anticipated and lower-bound hop counts, it is seen that in most cases the experimental results are comparable to or surpass the anticipated improvement. This implies that any bias introduced by differences in the mapping algorithms makes a large number of ports appear in a better light than they should, thus strengthening the conclusion that a small number of ports is better.

Finally, let us consider the impact of using a better mapping algorithm on the results derived so far. As a closer match is found between the communication structure of the application program and the network topology, communications become more localized. Processors communicate less with processors far away, so the reduction in average path length, which occurs when the number of ports is increased, become less effective. The strength of the argument for a large number of ports relies on a reduction in network congestion. However as the mapping is improved, congestion becomes less significant. Increasing the number of ports only decreases the bandwidth available to each virtual circuit, and thus degrades performance. Thus, the fact that the simulation results may not use an "optimal" mapping of tasks to processors can only bias the results to favor a large number of ports, strengthening the conclusion that a small number of ports is better. Some support for this conclusion is seen in figure 3.23f, where an optimized mapping for the FFT program yields better performance with a small number of ports, while the original mapping yielded better performance with a large number of ports. Similarly, as shown in figure 3.12c, the execution of the FFT on the butterfly topology, an optimal mapping, yields better performance when a small number of ports is used.

3.9. Precision of the Simulations Results

A certain amount of uncertainty exists in all of the simulations presented thus far. The arrival times of messages at each node is a function of the queueing delays encountered in previous nodes, which in turn depends on the arrival times of other messages. These complex interactions lead to message delays which vary according to the times at which messages are generated. In general, the application programs executed on the multicomputer are not known a priori, so some uncertainty exists in the times at which messages are generated by the application program, leading to uncertainty in message delays. If this uncertainty is large, the results presented thus far could be based on chance behavior rather than the tradeoffs described earlier. The amount of this uncertainty will now be examined.

The fact that the programs generated a relatively large number of messages (typically, thousands) combined with the large quantity of curves producing the same qualitative results leads one to suspect that the conclusions presented thus far are *not* simply the result of random fluctuations. Furthermore, the fact that the curves yielded results which were consistent with those predicted by the analytical models (or in disagreement in explainable ways) strengthens this belief. Nevertheless, in order to obtain a quantitative measure of the uncertainty described above, simulations were repeated with fluctuations introduced in the times at which messages are generated.

The artificial traffic generator program was executed on the hexagonal lattice network of figure 3.14a for the case of processor and communication circuitry integrated onto the same chip. This network was chosen because it does not exhibit any of the bottlenecks described earlier, e.g. the root bottleneck in the tree network, which might mask the effect of the random fluctuations. Interarrival times are again selected from an exponential distribution. The results of these experiments are shown in figure 3.24. Each curve represents performance for a different set of message arrival times. Fluctuations are introduced by using different seeds in the random number generator which determines interarrival times. As shown in figure 3.24, the delay in the lightly loaded network varies by up to 1.7%, while network throughput varies by up to 8.8%. Uncertainty in delay does increase significantly as the network approaches saturation, however this does not affect the conclusions derived above since they were based on the previous two performance measures.

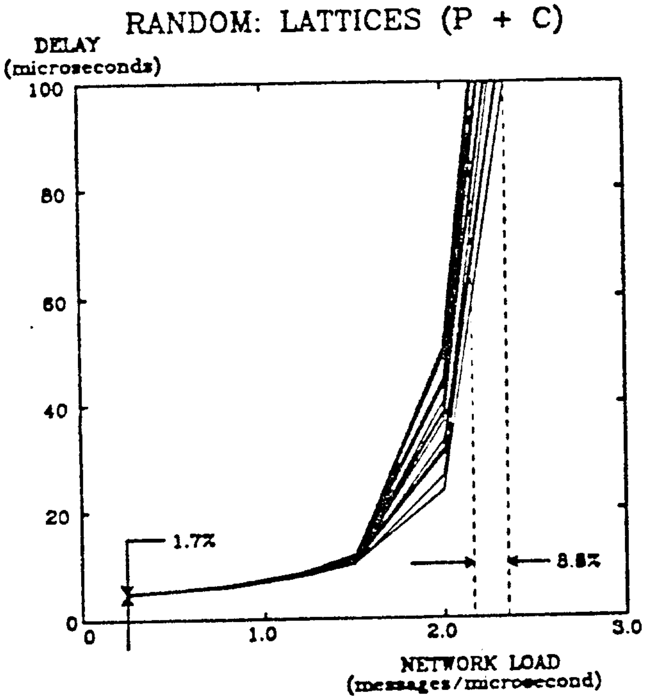


Figure 3.24. *Precision of simulations.*

Message delay uncertainty leads to uncertainty in the execution time of the programs. This latter quantity is the performance measure used in the bulk of the simulations presented here. The percentage of uncertainty in execution time will be *less* than that corresponding to message delay however, because message delay only affects one component of the overall execution time. Execution time is composed of two components, the time spent executing instructions and the time spent waiting for data. Message delays only affect the latter component. Although the absolute magnitude of fluctuations may be the same for both message delay and execution time, the percentage of the uncertainty will be smaller in the latter, since it is always larger. Thus, the uncertainty in message delay described above will lead to an even smaller uncertainty in execution time.

3.10. Summary of Simulation Studies

In most cases, the simulation results support the analytical results discussed earlier. When discrepancies do occur, they favor networks using a small number of ports. It was seen that bottlenecks are the source of these discrepancies. Thus, the simulation results support the conclusion found earlier that components with a small number of ports are better. These results also demonstrate the utility of incorporating efficient mechanisms for handling multiple destination messages. Such a mechanism can yield significant performance improvement in algorithms relying heavily on global information.

CHAPTER FOUR

DESIGN AND IMPLEMENTATION OF COMMUNICATION COMPONENTS

This chapter examines the amount of circuitry required to implement a VLSI communication component. Alternative mechanisms for transporting data through communication networks are first compared, and a *virtual-circuit* transport mechanism is argued to be the most attractive alternative for the networks discussed here. Details of such a transport mechanism are then described. Next, alternative schemes for providing hardware support for three key communication functions: routing, buffer management, and flow control, are described. Practical figures for the number of channels and buffers within each component are derived and used as the basis for estimates of the complexity of such a component. It will be seen that the functional capabilities of VLSI chips are now sufficient to allow the construction of communication components with enough buffer space and virtual channels to provide high-bandwidth communications in multicomputer networks.

4.1. Transport Mechanisms

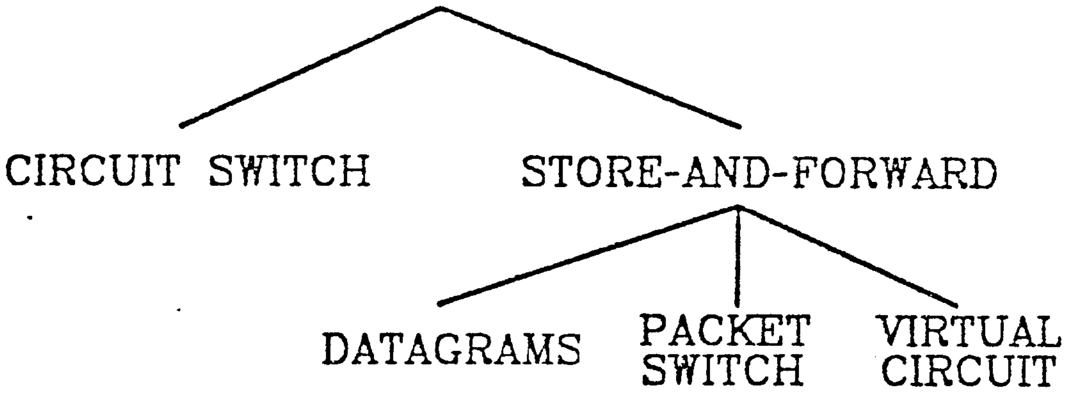
Since the primary function of the communication network is to move data, a transport mechanism, i.e. the means by which data is transmitted through the network, must be selected. A classification tree which includes the various transport mechanisms in use today is shown in figure 4.1. A number of characteristics which distinguish these transport mechanisms are also shown. Briefly, these characteristics are:

(1) *Data Unit*: The unit of data transported through the network is either a variable-length message or a fixed-length packet.

(2) *Routing Overhead:* The overhead associated with message routing is incurred either on a hop-by-hop basis at each node in the network or only in the initial set-up of a circuit.

(3) *Bandwidth Allocation:* Bandwidth is allocated by the network either statically, e.g. when a circuit is set up, or dynamically as messages enter the network.

(4) *Buffering Complexity:* The complexity of the buffering hardware varies with the sophistication of the chosen transport mechanism.



	CIRCUIT SWITCH	DATAGRAMS	PACKET SWITCH	VIRTUAL CIRCUIT
Data Unit	arbitrary	messages	packets	packets
Routing Overhead	set-up only	per message	per packet + (reassembly)	set-up
Bandwidth Allocation	static	dynamic	dynamic	dynamic
Buffering Complexity	low	high	moderate	moderate

Figure 4.1. *Transport Mechanisms.*

According to the tree in figure 4.1, the first alternative to consider in selecting a transport mechanism is between a circuit-switched and a store-and-forward approach. The circuit-switched approach is best exemplified (at least conceptually) by the telephone system: When someone picks up a telephone and dials a phone number, a circuit is established between the caller and the party being called. Once this circuit is established, it remains intact until either party hangs up. Examples of circuit-switched networks are described in [Joel79, Mass79]. The most distinguishing characteristic of this approach is the fact that communicating parties are guaranteed a certain bandwidth and maximum latency when the call is established. Since the communication network cannot know when data will be transmitted, bandwidth must be allocated statically when the call is set up. Otherwise, the bandwidth may not be available when it is needed. Users are allocated a certain amount of bandwidth regardless of whether or not they actually use it. If communications are bursty, as is often the case in computer networks, much of the network's bandwidth will be wasted. This is the primary disadvantage of the circuit-switched approach.

However, the circuit-switched approach also offers a number of advantages. Routing overhead is usually paid only when the circuit is set up, so subsequent messages can flow through the network with little delay. This reduces the average delay on circuits carrying more than one message. Also, buffering strategies are simpler than those required for store and forward networks because bandwidth allocation is performed statically. If circuit-switching is used, the network can be designed to ensure that the rate of traffic flow into each node of the network never exceeds the rate of flow out, alleviating buffer overflow problems. In fact, if each circuit is implemented as a physical electrical connection between the communicating parties (e.g. a series of relays), the network need not provide any buffering at all!

Store-and-forward networks avoid the wasted bandwidth problem described above by allocating bandwidth dynamically to messages as they enter the network. Three types of store-and-forward networks have been implemented in the past:

(1) Datagram networks.

(2) Packet switched networks.

(3) Virtual circuit networks.

Datagram networks are characterized by the unit of data sent through the network — variable length messages. Since each message can be relatively large, communication components would have to provide a relatively large amount of buffer space to hold arriving messages. This implies that a large amount of circuitry in each component must be devoted to messages buffers. In addition, since messages may vary in length, variable size buffers must be used. This increases the complexity of the buffer management circuitry significantly, since the buffer selected for a particular message must be at least as large as the message. This problem is identical to the difference between virtual memory systems based on segments, whose complexity usually requires a software implementation, and those based on pages, which are usually implemented, at least to a large extent, in hardware. Finally, routing overhead in the datagram approach is worse than that of the circuit-switched approach because routing decisions must be made on a hop-by-hop basis with each message sent into the network.

The packet switched transport mechanism alleviates many of the buffering problems described above. Here, each message is divided into a number of (usually) fixed-sized packets which are routed separately through the network. An end-to-end scheme is required to reassemble the message from its constituent packets. Since packets can be relatively small, buffering requirements in each component are reduced. The use of fixed-sized packets also simplifies the buffer management circuitry. This approach does incur a significant amount of overhead on an end-to-end level to reassemble messages however. Since packets are routed separately through the network, they may follow different routes to the destination node, and therefore may arrive in an order different from that at which they were sent. The other disadvantage of the packet switched approach is that the routing overhead problem is worse than that of the datagram scheme, since this overhead now occurs on every *packet* rather than on every *message*.

If we examine the transport mechanisms described thus far, we see that the circuit-switched approach suffers from static bandwidth allocation, while the packet switch approach suffers from reassembly and routing overhead. One might hope that a hybrid which combines these two approaches can achieve the best of both mechanisms without their respective disadvantages. This is the motivation behind the *virtual circuit transport mechanism*, which is a mixture of packet switched and circuit-switched techniques. Here, a *virtual circuit* is established between processors which wish to communicate. A virtual circuit is a fixed, unidirectional path through the network from one processor to another. All messages sent on this circuit travel along this path to reach their destination.

Let us consider the characteristics of the virtual circuit transport mechanism (listed in figure 4.1). Like the packet switched mechanism, the data unit is a fixed sized packet (simplifying the buffering problems of the datagram mechanism). Routing is similar to the circuit-switched approach to the extent that the routing algorithm need only be applied when the circuit is set up, and not with subsequent packets. It will be seen however, that some overhead is still required to route messages, so the routing overhead is intermediate between the circuit switched and the datagram/packet switched approaches. Since a store-and-forward mechanism is used, network bandwidth is allocated dynamically, although allocation is not as adaptive as it is in packet switched networks because packets are constrained to follow a fixed path from source to destination. The fixed path restriction in the virtual circuit mechanism is necessary to reduce routing overhead and to avoid reassembly overhead. Thus, while packet switched networks may be able to achieve higher bandwidth along an end-to-end connection by utilizing multiple paths between the two processors, the virtual circuit scheme will yield lower latency on individual messages since they spend less time in each node waiting for routing decisions to be made. In addition, the virtual circuit mechanism can utilize multiple paths between two nodes by establishing several circuits between the two processors.

Thus, a virtual circuit transport mechanism appears to be the most attractive for the networks described here. Details of the operation of this mechanism are described in the next section. Hardware implementations are described in the sections which follow.

4.2. A Virtual Circuit Based Communication System

The communication domain studied here is a packet-based network using a virtual circuit transport mechanism. Mechanisms for establishing, maintaining, and tearing down circuits are described in this section.

4.2.1. Virtual Circuits

Each processor has a fixed number of input and output circuits for receiving and sending data respectively. Sending a message is a three step process. First, a virtual circuit (i.e. a path of time-multiplexed links) to the destination processor is established by sending a message header with routing information through the network. Once a circuit is set up, an arbitrary amount of data, which may consist of several logical messages, can be sent along this circuit. Data can follow the message header immediately without an end-to-end handshake and need not be transmitted continuously for the circuit to remain intact. This approach reduces the routing overhead on all packets except the message header. When the circuit is no longer needed, it is torn down by sending a tagged message trailer.

The communications system provides only a data transport facility. Except for the header and trailer information, all data passes uninterpreted through intermediate nodes. Error checking and retransmission are left to an end-to-end protocol. This allows the forwarding of data packets in each node to begin before the entire packet has arrived if the proper outgoing link is idle (virtual cut-through [Kerm79]). If error checking and retransmissions were performed within the network on a hop-by-hop basis, forwarding could not begin until the entire packet has arrived and was checked for errors, since otherwise an erroneous packet would have been forwarded by the time the error was detected. This end-to-end approach is justified by the low error rates observed in local computer networks [Shoc80]. Since the networks discussed here cover an even smaller geographic area, and thus are less susceptible to environmental noise, this assumption is even more appropriate.

4.2.2. Virtual Channels

The communications domain can be viewed as a simple, connected graph. Nodes and edges represent communications components and links, respectively. A circuit from one processor to another corresponds to a path in this graph. Two distinct paths (say from node A to B and from C to D in figure 4.2) may use a common edge (from X to Y). Thus, the link associated with that edge must be multiplexed between the two paths, and provisions must be made to ensure that data from A is sent to B, and not to D.

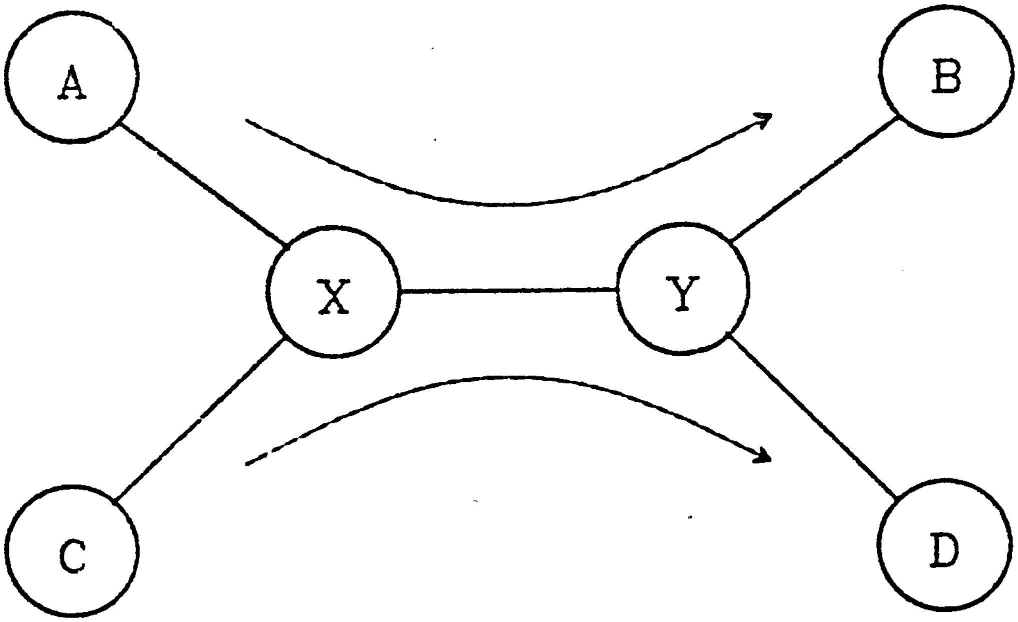


Figure 4.2. *Two Paths Multiplexed through the Same Link.*

Each physical link is divided into some fixed number of unidirectional *virtual channels*. Each channel can carry data for one virtual circuit (i.e. one path). Thus, a circuit from one node to another consists of a sequence of channels on the links in the path between the two nodes. The circuit from A to B in figure 4.2, for example, might use channel #3 to get to X, then #5 to get to Y, and finally #7 to get to B.

When node X sends data to node Y, the latter must determine which circuit this data belongs to. Two commonly used techniques for providing this information are, among others:

(1) Divide the link into a fixed number of *time slots* and statically assign each time slot to a channel (e.g. the first time slot might be assigned to channel #0, the second to channel #1, etc.). The time slot on which the data arrives identifies the channel that sent it.

(2) Precede the data with a tag that identifies the channel it is being sent on. In this scheme, the available bandwidth on the link is allocated to the various channels by some demand-driven scheduling algorithm.

In the first scheme, the link is effectively divided into a number of lower bandwidth links, with the sum of these bandwidths equal to that of the physical link. If a channel does not send any data, its allocated bandwidth is wasted. In addition, latency is increased since each channel must wait for its turn to send a unit of data. In the second scheme, the entire bandwidth of the link can be allocated to channels upon demand, i.e. when they have data to send, so the inefficiencies associated with the previous approach are avoided. However, some bandwidth is required to carry the channel tag. Demand-driven time-multiplexing is superior if the degree of multiplexing on each link is high, and many channels do not always have data to send. This is often the case in computer-to-computer communications, so the dynamic approach is more suitable for the networks described here.

4.2.3. Routing Hardware

In order to route messages through each node of the network, channels entering a node (input channels) must be "linked" to channels leaving the node (output channels). Each node maintains a set of *translation tables* to perform this function. There is one translation table for each input port of a node. Each entry of the translation table contains two fields: an output port, and the number of a channel on that port. When data arrives on an input channel, say channel #3, entry 3 of the translation table for that port is read to yield the output port and the number of the channel the data is to be forwarded on.

The translation tables logically link incoming and outgoing channels, and thus establish the various virtual circuits through the node. Setting up these circuits involves allocating channels and updating translation tables along each path from source to destination. This task is performed by a *routing controller* residing in each communications node.

Initially, all translation tables specify that data is to be sent to the local routing controller. When a message header setting up a new circuit arrives at a node, the routing controller analyzes the destination address in the header, determines the proper output port with the use of some routing algorithm, allocates a free output channel, and updates the translation table at the input port. Measurements on a TTL prototype of such a routing controller [Fuji80] show that this entire operation can be done in 4-5 μsec if a free channel is available on the selected link. Subsequent data is then forwarded without intervention by the routing controller. Similarly, when the circuit is torn down, the channel is released, and the corresponding translation table entry is reset to point to the routing controller.

4.2.4. Packet Types and Formats

Three types of packets have been discussed thus far: a "set-up packet" which establishes virtual circuits, a "trailer packet" which tears them down, and a "data packet" which carries data. In addition, it is useful to provide a "clear packet" which flows through a virtual circuit, removing any data packets it encounters along the way. Such a mechanism is useful in error recovery protocols to reset virtual circuits to a "known" state.

Packets must be tagged to distinguish the various types. Each packet is preceded by a header which indicates the packet's type, as well as a channel number indicating which virtual circuit the packet belongs to. Set-up packets also carry routing information (e.g. a destination address) which the routing controller uses to set up the circuit. Assuming two bits to indicate type, one bit for parity, and a one byte header on each packet (excluding routing information on set-up packets), 5 bits remain for a virtual channel number. This implies a maximum of 32 channels can be supported on each link. Later, it is argued that under current technology, a larger number of channels should be supported, say 64 or 128. Since it is convenient to restrict the header information to an integral number of bytes, a two-byte header could be used to support this many channels.

An alternative approach to the fixed length header scheme described above is to use variable length packet headers. For example, assuming that most of the packets flowing through the network are data packets, we could confine the overhead in these packets to a single byte, while forcing other packet types to use several bytes. Under this scheme, the header of each data packet consists of a 7-bit channel number and a single bit for parity. One channel number, say #0, is declared to be "undefined". When a packet header specifies this channel number, it indicates that the packet is not a data packet, but rather some other type. Subsequent bytes indicate the type of packet, and any type-specific information. This approach reduces the overhead required on data packets, and thus provides better performance in transmitting these packets than the fixed-length header scheme described above. Although the amount of time required to process the other packet types, the set-up packet in particular, is slightly increased, delays on these other packet types are less crucial. The assumption that data packets use a single byte for header information was used in much of the analysis presented earlier.

4.3. Key Functions of the Communication Component

Any communication component must provide mechanisms for routing messages to their proper destination, managing the limited amount of buffer space, and controlling the rate at which packets flow from one node to another. Hardware implementation of these mechanisms is required to achieve high-performance. Mechanisms to perform these functions are outlined in this section. Hardware implementations are described in the section that follows. Implementation of other portions of the communication component, i.e. the I/O ports and routing controller, are only briefly summarized since they are described elsewhere [Laur79, Wong81, Fuji80].

A block diagram indicating the functions that must be provided by each component is shown in figure 4.3. The component contains three or more ports, each accommodating a link to a neighboring node. It also contains a certain amount of buffer memory, bookkeeping tables, and control logic. Translation tables logically link incoming and outgoing channels, and thus establish the various virtual circuits through the component. Finally, a microcoded engine called the routing control is responsible for setting up virtual circuits and implementing less frequently used network functions such as failure recovery protocols.

4.3.1. Routing

All communication networks require some routing algorithm to build the paths, i.e. the virtual circuits, between nodes sending and receiving messages. A great deal of research has been done in the area of routing in loosely coupled computer networks, and much of this work is applicable here [Gerl81, Tane81]. In the context of the proposed communication domain, we will only consider totally distributed routing that does not rely on a centralized authority. For this discussion it is also appropriate to distinguish between regular networks with a predefined topology, such as arrays or binary trees, and irregular networks of arbitrary connectivity.

In regular networks, routing can be performed in each node by a state machine which performs a fixed algorithm based on the local and destination addresses. In square lattices, for example, the routing controller could forward the message header in a direction that would reduce the difference between the x- or y- coordinates of the current and the destination nodes. Routing

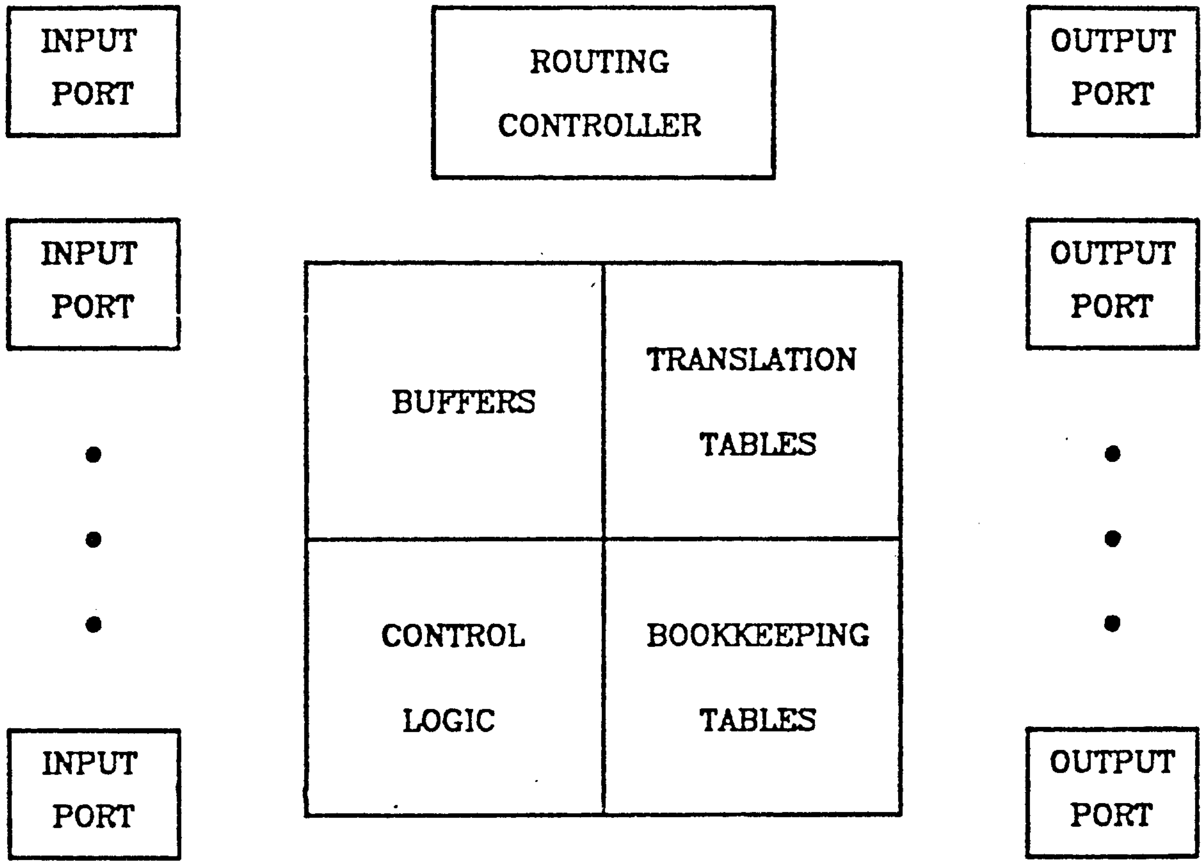


Figure 4.3. *Functions provided by communication component.*

algorithms for binary half-ring and full-ring trees have been discussed elsewhere [Sequ78].

For a general-purpose communication component, the routing algorithm must not be frozen in hardware. A routing controller with a writable program memory is more appropriate and guarantees that the same component can serve many different network topologies. A routing algorithm suitable for the particular network structure could be broadcast at system initialization.

For irregular networks, routing may be based on suitable lookup tables. In a decentralized system each node i has entries of the form:

$$NN = R_i(DN),$$

implying that messages destined for node *DN* are forwarded by node *i* to neighbor node *NN*. This lookup table, commonly called a *routing table*, can be defined statically, or it can be maintained dynamically using information exchanged between neighboring nodes. The latter approach also allows the network to automatically reconfigure itself should the topology change due to node failure or network expansion [Taji77]. Techniques to initialize and maintain the routing tables are discussed in [Gerl81].

If the network has many nodes, the routing table will be excessively large since a separate entry is required for each destination node. A common technique which reduces the size of this table is to employ hierarchical names and multiple routing tables per node [Kamo76]. An example of such a mechanism is seen in the telephone system in which names (telephone numbers) consist of an area code and a seven digit number. When a call to a number with a different area code is made, the area code is first used to route the call to the correct area, and then the phone number is used to locate the final destination. Conceptually, routing could thus be performed as follows:

(1) If the area code of the destination matches that of the router, then the seven digit number is used to locate the next node via a "neighborhood" routing table.

(2) If the area code does not match that of the node doing the routing, then the area code is used to look up the next node via an "area code" routing table. The remaining seven digits in the phone number are ignored.

Thus, a two-level naming hierarchy is used along with a routing table for each level. Such a scheme reduces the table size by grouping nodes which are far away into a single entry in the "area code" routing table.

One can easily extend this principle to an arbitrary number of naming levels. To determine the number of levels required to minimize the storage space required for routing tables, let there be l levels, with g_i entries in the level i table. The object is to minimize $g_1 + g_2 + \cdots + g_l$ subject to constant $N = g_1 \times g_2 \times \cdots \times g_l$, the number of nodes in the network. It is easy to show that this sum is minimized for

$$g_1 = g_2 = \cdots = g_l = e \quad \text{and} \quad l = \ln N,$$

where e is approximately 2.718. Thus, to minimize the table size in each node, there should be many levels with few entries in each level [McQu74].

The reduction in table size resulting from a multi-level routing scheme can be substantial. A 16-bit destination address partitioned into eight 2-bit fields requires eight 4-entry routing tables, or a total of 32 entries. The single-level routing table would require 65,536 entries. The routing controller described in [Fuji80] uses a single-level lookup table with 256 entries. A hardware implementation of a hierarchical routing scheme will be presented later.

4.3.2. Buffer Management

Each message passed into the communication domain must be subdivided by the sender into some number of fixed-length packets. As discussed earlier, allowing variable length packets adds a considerable amount of complexity to the component. These packets form the unit of data transmitted across the links of the communication domain. Due to conflicts that arise when several packets simultaneously require the use of the same link, buffering is required in each node. The communication component must have some strategy for managing these buffers.

A scheme is necessary to allocate a node's buffers among the virtual circuits using the node. A simple solution is to give each channel on each link a separate buffer. This is inefficient however, since much of the buffer space will be unused most of the time. By allowing several channels to share buffers, fluctuations in the need for buffer space can be averaged over a large number of communication paths, and fewer buffers are required to achieve the same performance. A mapping is then required to link each channel to the buffers holding packets for that channel so that they can be found when it is time to forward them. Furthermore, when a new packet arrives, an empty buffer must be found. From this perspective, buffer management is similar to the management of a cache memory: a program (here, a channel) must fit blocks from main memory (packets) into cache pages (buffers).

As in cache memory design, there are three well known schemes for performing this mapping:

(1) direct mapping

(2) set-associative mapping

(3) fully associative mapping

In turn, these three schemes offer an increased degree of buffer sharing, and thus improved memory utilization, but at the cost of increased complexity in the control circuitry. They are distinguished by restrictions on where a channel's packets can be placed. In the direct mapping scheme (minimal sharing), each channel has a set of buffers dedicated to it, i.e. its own fifo buffer queue. The set-associative scheme (moderate sharing) allows each channel to use a larger set of buffers, but it is no longer given sole access to them. This scheme might be implemented by letting all channels of a single port share a pool of buffers dedicated to this port. In the fully associative scheme (maximal sharing), each node has a centralized pool of buffers which all channels share. Implementations of the set-associative and fully associative schemes will be discussed in later sections. An implementation of the direct mapping scheme has been described previously [Laur79, Sequ78].

4.3.3. Flow Control

Flow control refers to the mechanism which regulates the transmission of data packets along virtual circuits. The network must be able to "throttle" traffic on virtual circuits to prevent buffer overflow (such mechanisms are sometimes referred to as congestion control in the literature [Tane81]), and to handle situations in which a processor is sent more messages than it can immediately receive. In addition to providing a *mechanism* which allows components to throttle traffic, a *policy* is also required to determine which virtual circuits must be throttled, and when. Such a policy will be discussed next, followed by a discussion of different throttling mechanisms.

Since one of the purposes of flow control is to avoid buffer overflow, a natural policy is to begin throttling traffic when the pool of free (i.e. empty) buffers becomes depleted. If a node is inundated with data, packets will "back up" along the virtual circuits leading up to it, much like the way cars back up on a congested freeway. This type of flow control, called "back pressure flow control", is analogous to water (packets) flowing through a pipe (buffers). If the pipe becomes blocked or constricted, water backs up to its source. Such a mechanism has been used successfully in TYMNET, a loosely coupled, commercial communication network [Tyme81].

The flow control policy described above can lead to a problem called "buffer hogging". Here, one virtual circuit uses more than its share of the buffers in a node. If a virtual circuit becomes blocked, e.g. due to a congested output link, packets may continue to arrive on that virtual circuit and occupy most, or all of the buffers in the node. Without some mechanism to restrict buffer sharing, buffer hogging will impede other traffic using the node and lead to deadlock situations. This situation can be avoided by controlling the maximum number of buffers each channel can use. It might be noted that the direct mapping scheme, and to a lesser extent the set-associative scheme, automatically provide some protection against buffer hogging, since they inherently restrict buffer sharing. All three schemes however, need some mechanism to ensure that data is not lost if no free buffers are available.

Thus, in order to prevent buffer hogging, each output channel may not hold more than some "channel limit" of buffers at once. Even with this restriction however, another form of buffer hogging may still arise. A congested output link could use all of the node's buffers and block traffic on other links. To prevent this, each output port is restricted to using no more than some maximum number of buffers, determined by a higher level protocol. This maximum number, called the "port limit", can be changed dynamically to shift additional buffers to highly utilized ports, while still providing some space for traffic on lightly loaded ports. Studies indicate that by restricting the number of buffers an output port can use "output port buffer hogging" is prevented, and a significant improvement in the bandwidth provided by the node is obtained [Irla78]. These studies also indicate that as a general rule, each port should not be allowed to use more than $b/\sqrt{p}$ buffers in a p-port node with b buffers.

Assuming a buffer allocation policy is used to control the rate of packet forwarding, let us now examine the flow control mechanism itself, i.e. the mechanism which performs the actual throttling. Two mechanisms, sender-controlled and receiver-controlled throttling, will be discussed. They are characterized by whether the sending or the receiving node implements the policy described above. The receiver-controlled mechanism is the simpler mechanism, and will be described first.

The receiver-controlled flow control mechanism can be implemented by a send/acknowledge protocol to transmit data over the link. In this scheme, each node sends a packet, and waits for the receiver to return a control signal indicating whether it accepted or rejected (i.e. discarded) the packet. An "ack" signal denotes an accepted packet while a "nack" denotes a rejected packet. If a nack is returned, the packet must be retransmitted at a later time.

A receiver may choose to reject a packet because of buffer space limitations or transmissions errors. Here, it is assumed that communication components only check header information for transmission errors, since the virtual cut-through mechanism prevents retransmission if errors in the data are detected. With virtual cut-through, the first bytes of the packet may have been forwarded to the next node before an error in later bytes is detected, making immediate recovery difficult, if not impossible. Errors in data bytes must be handled by an end-to-end protocol which detects and retransmits damaged packets.

It is also assumed here that each link has a separate control line to carry the ack/nack signal back to the sender. Alternatively, the control signal could be piggy-backed onto a packet going in the opposite direction, however, this leads to a "looser coupling" between sender and receiver, forcing the sender to either deal with multiple unacknowledged packets pending over the link [Pouz78] (adding a considerable amount of complexity to the circuitry in the port), or to stop using the link until the acknowledgement arrives (wasting bandwidth). Since the receiver can generate an acknowledgement after only the header is received, and since a direct connection to the sender is available for transmitting this signal, this scheme offers the unusual feature that the sender will receive the acknowledgement before it has finished sending the packet! This allows a virtual circuit to "pipeline" a stream of packets through an otherwise idle node without incurring the delays associated with waiting for acknowledgements or the complexity of multiple unacknowledged packets.

An alternative approach to flow control is to implement the buffer allocation policy for a node in its neighboring nodes, i.e. control the flow of information from the sender rather than the receiver end of each link. For example, each output port could maintain a table remembering how many buffers in the neighboring node are allocated to each channel of the link connecting the two. With this information, the sender can decide which channel to serve next, and packets can be forwarded without the risk of overflowing the buffer space in the receiver. Maintaining this remote status information requires some overhead: The fact that the receiver has freed up a buffer must be reported back to the transmitter. Finally, since packets cannot be retransmitted, transmission errors in packet headers result in lost packets. An end-to-end mechanism is required to retransmit these packets.

As in the send/acknowledge flow control scheme, buffer hogging is prevented by controlling the number of buffers used by each channel. It might be noted however, that output port buffer hogging is much more difficult to prevent. This is because the size of the queue on an output link depends on the packets received from the node's neighbors. When these neighbors send packets, they do not know which output port in the receiving node the packet will use, since routing decisions are made inside the receiver. Thus the neighbors cannot control the queue size on a specific link, and nothing prevents a single port from monopolizing the entire buffer pool.

The send/acknowledge protocol leads to a simple implementation, while the remote buffer management approach prevents rejected packets, and thus avoids retransmissions and waste of bandwidth. Both schemes require some overhead to provide the feedback signals necessary for flow control. In the send/acknowledge scheme, dedicated pins are used, while in the remote buffer management scheme, piggy-backed control signals are required. Implementation and comparisons of these two mechanisms will be described in the sections which follow.

4.4. Implementation of VLSI Communication Components

Hardware implementations of the communication functions described above are outlined in this section, and two designs are presented which integrate these functions into a single chip. The first is a Y-component design using a set-associative buffer management scheme and remote buffer allocation for flow control. It will be seen that there are a number of severe deficiencies in this design. The second design, which corrects these deficiencies, uses a fully associative buffer management scheme. Implementations of both sender-controlled and receiver-controlled flow control mechanisms are also discussed. Common to both designs is the routing controller with hardware support for hierarchical routing. This is the subject of the next section. The two designs are described in subsequent sections.

4.4.1. Routing Hardware

This section describes a hardware implementation of the hierarchical routing table mechanism described earlier. This hardware is part of the routing controller which is responsible for setting up virtual circuits through the node. The remainder of the routing controller is described in [Fuji80].

When a virtual circuit is being constructed, the routing hardware is given a hierarchical destination address, and must determine which output port the virtual circuit is to use. This is accomplished by a set of routing tables, one for each level of the hierarchy, as discussed earlier. The routing controller, part of which is implemented as a microprogrammed engine, is responsible for loading and maintaining the routing tables, e.g. by a shortest path routing algorithm.

Two implementations are discussed. The first assumes that routing tables at all levels are the same size, some power of 2. The second design relaxes this assumption, but at the cost of added complexity.

An l-level hierarchical node address consists of a string of digits, $A_{l-1}A_{l-2} \cdots A_1A_0$. Digit A_i is used to index the routing table at level i. A routing table entry contains either an output port number indicating which port to use, or a "NULL" flag indicating that the table on the next level must be searched. Let RT_i denote the routing table at level i, with $0 \le i < l$. The algorithm to determine the appropriate output port is as follows:

```
level := 0;     /* current level, 0, 1, ... $l$-1 */

while (($RT_{level}[A_{level}]$ = NULL) and (level < $l$))
        level := level + 1;

if (level < $l$)
        return ($RT_{level}[A_{level}]$);       /* return output port */
else
        return (NULL);        /* destination node reached */
```

If this routine returns NULL, then the message has reached it's final destination, i.e. the destination address matches the local address. Otherwise, the number of the output port selected by the routing algorithm is returned.

One hardware implementation of this table lookup mechanism is shown in figure 4.4a. It is assumed that each table contains 2^k entries. The bus widths in figure 4.4a assume that there are 8 levels, and 4 routing table entries in each level (i.e. k=2). The "address register" holds the destination address. The rightmost k bits of this register hold A_0, the next k bits hold A_1, etc. A single RAM holds all of the routing tables. The upper bits of the address lines of this RAM specify a routing table (i.e. a level), and the lower bits specify an offset into this table. The lower k bits of the address register (i.e. A_{level} in the program above) are concatenated with the output of the level counter (the current level) to form this address. The shifter aligns the destination address bits by shifting out A_i and moving A_{i+1} into the rightmost position each clock cycle. Since each bit is shifted exactly k bits on each clock, the shifter can be implemented by an edge triggered register and a simple permutation of wires. Finally, not shown is the control logic which sequences through the various routing tables. Design of this finite state machine is straight-forward, using the level counter and circuitry to detect NULL routing table entries, and generating signals to shift the address bits and increment the level counter.

A second implementation, shown in figure 4.4b, relaxes the "fixed routing table size" restriction. The bus widths shown in this figure support up to 8 levels and a total of 256 routing table entries. The number of levels and sizes of the various routing tables is programmable at system initialization. The "address RAM" holds the base addresses of the various routing tables. Entry i contains the base address of the routing table at level i. The routing tables are again stored in a single RAM. The routing table offset, A_i, is generated by masking appropriate bits of the address register. This offset is added to the base address to generate an address for the routing table RAM. A barrel shifter aligns data in the address register for the next iteration. The mask bits and the number of address bits to be shifted are stored in the "mask RAM" and "shift RAM" respectively. These RAMs are loaded by the routing controller at initialization. Together, their contents describe the format of the address register. The control logic for this second implementation is virtually the same as that of the previous design.

4.4.2. A Y-Component Design

The design of a Y-component has been studied [Wong81]. Details of this design will be repeated here as an example of one implementation of the

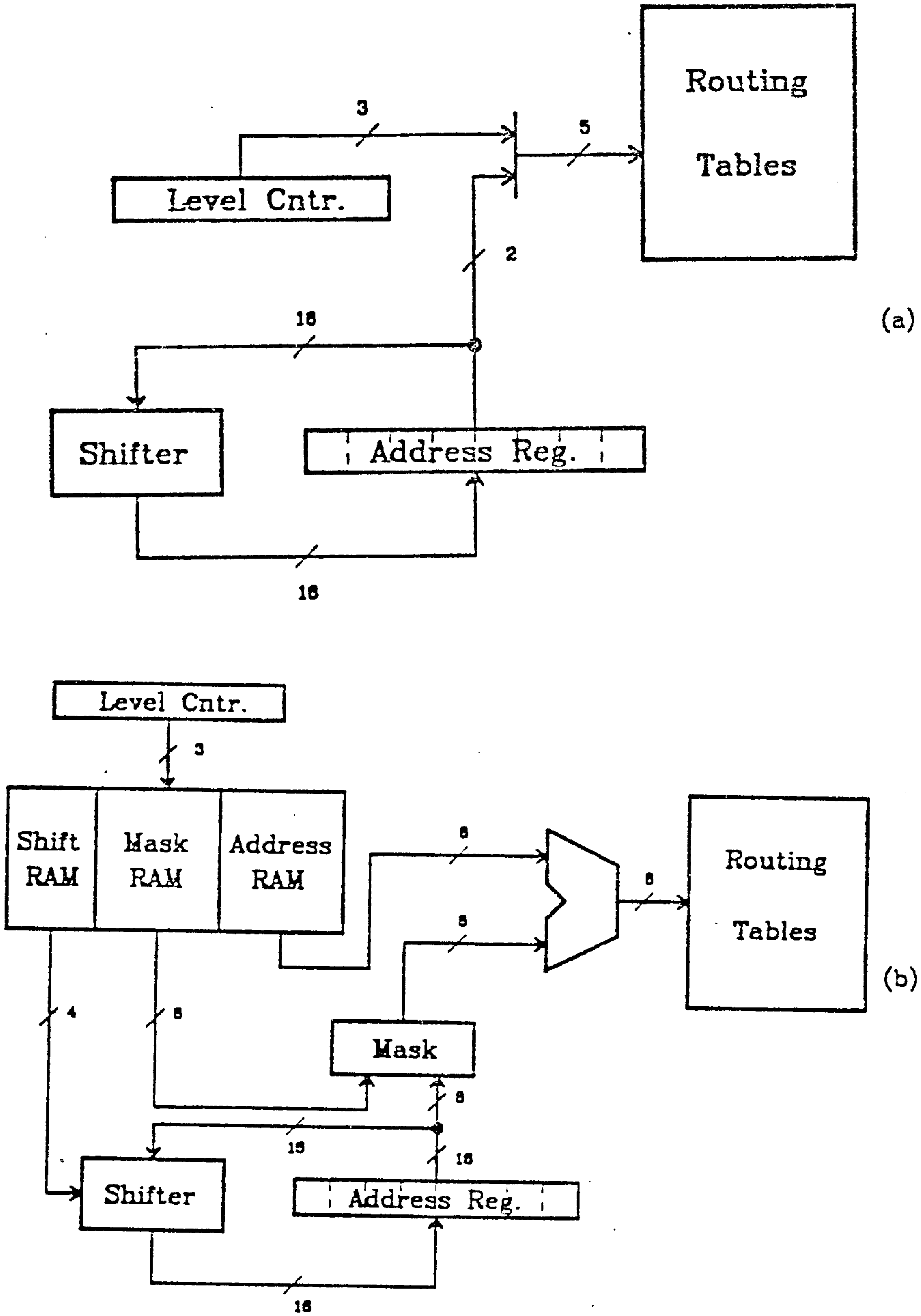


Figure 4.4. *Hierarchical routing circuitry (a) simple. (b) complex.*

functions described above. A block diagram for this design is shown in figure 4.5. The component consists of a routing controller (R), three input ports, three output ports, and three buffer modules (B), one associated with each input port. It will be assumed that there are *c* input and *c* output channels on each port, and that each buffer module consists of *b* data buffers.

When a packet arrives at an input port, it is placed in one of that port's buffers. The routing controller (which can, for the moment, be considered a fourth output port) and the other two output ports actively search this input port's translation table and buffers to locate packets destined for it. When such a packet is found, it is forwarded and the buffer is marked empty. Some addi-

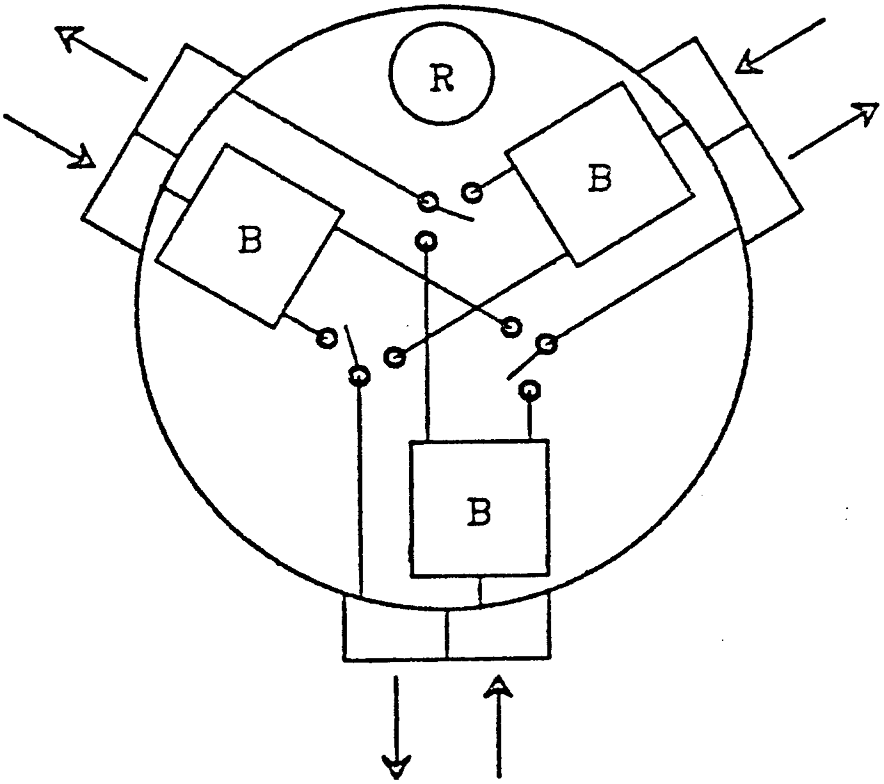


Figure 4.5. *Block diagram of Y-component.*

tional control logic ensures that packets on each channel are forwarded in the order in which they arrived.

Although the translation table and the buffer memory of a single input port can both be read at the same time, it is not possible to simultaneous perform two reads of the *same* translation table or buffer memory. To avoid conflicts, each "major clock cycle" (the time interval to transmit or receive a single word of data over the link) is subdivided into 4 "minor clock cycles", and these minor cycles are statically assigned to output ports to time multiplex access to the buffers and translation tables without contention. It is assumed that each translation table and buffer memory can be accessed during a single minor clock cycle. This assignment ensures that each output port has an opportunity to read the translation table and buffer memory of each of the other two ports (or in the case of the routing controller, the other three ports) during each major clock cycle.

4.4.2.1. Buffer Management Hardware

A set-associative buffer management scheme is used in this design. Of the b buffers assigned to each input port, each channel is statically assigned (say) 4 buffers by means of some algorithm for mapping channel numbers to buffer addresses, e.g. channel i might be able to access buffers i, $i+1$, $i+2$, and $i+3$, where all sums are taken modulo b. Thus, several channels share the use of each buffer.

Each input channel has four status bits which indicate which buffers actually hold a packet for that channel. These bits, as well as the input port's translation tables are scanned by the other two output ports and routing controller to locate packets which must be forwarded.

4.4.2.2. Flow Control Hardware

A remote buffer management scheme is used for flow control. The buffers of each input port are managed by the neighbor on the sender side of the link. In other words, the output port of each node is responsible for allocating buffer space in the neighboring node to the packets it sends.

In addition to the input port status bits described earlier, each output port maintains a bit map indicating which of the buffers of the input port on the other side of the link are free, and which are in use. When a component sends a packet, it not only specifies the number of the channel the packet is being sent on, but also the buffer that the receiving component is to use. It also must set the appropriate bit of the bit map to signify that the remote buffer is now in use. The input port receiving the packet then loads it into the designated buffer, and sets the appropriate input port status bit for the channel the packet arrived on, indicating that it has a packet waiting to be forwarded. When the appropriate output port sees that this bit has been set, it forwards the packet. The neighbor which originally sent it must be notified that this buffer is now free. A control byte piggy-backed onto a packet going to this neighbor accomplishes this task (a dummy packet is created if there is no traffic in this direction). Since the sender does not send a packet unless there is an empty buffer on the neighboring node to receive it, buffer overflow cannot occur.

4.4.3. Deficiencies in the Y-Component Design

The design presented above suffers from a number of deficiencies. The most severe problem arises from the polling scheme used to determine which channels hold packets waiting to be forwarded: each output port polls the translation tables and the status bits of the other input ports. If there are c channels per port, then each port requires c major clock cycles to poll all of the input channels of the other two ports (two channels, one from each port, can be polled in one clock cycle). If a packet arrives on an arbitrary channel, then an average of $c/2$ clock cycles expire before that channel is polled. Later, it will be seen that c should be relatively large, say 128 or 256, so long delays result from this polling scheme. In addition, it will be seen that the number of buffers in each component need not be very large, say 16 or 32, so most channels do not have packets waiting to be forwarded. Many idle channels will have to be polled before a channel with data is found. Thus, channel polling is an unreasonably slow and inefficient mechanism to locate waiting packets.

The remote buffer management scheme described above wastes link bandwidth, since it requires more overhead than is actually necessary. In the previously described scheme, a buffer number precedes every packet sent over the link. This is required because the sender allocates buffers in the receiving node. The allocation function could be controlled by the receiver however, since the sender only needs to be sure that a remote buffer exists to hold each packet it sends, and does not need to know the address of the remote buffer. Thus, buffer numbers need not be transmitted over the link. A single counter indicating the number of free buffers in the remote node's input port could be used to provide the necessary information without incurring additional overhead on the link. Sending a packet decrements the counter, while receiving a signal indicating that the remote node forwarded the packet will increment it. The receiver is left the responsibility of determining which buffer each arriving packet should use. This approach eliminates the need to send buffer numbers over the link, and thus achieves more efficient use of the link's bandwidth. Details of such an approach will be described later.

The design described above requires several memory references to the same memory on each major clock cycle. For example, the translation table polling mechanism requires four memory references per clock. In addition, if two output ports want to simultaneously forward packets from the same input port, two buffer memory reads per clock are required. The time required by these memory references could slow the clock rate, reducing the communication bandwidth of the entire network.

Finally, the studies which follow indicate that a high degree of buffer sharing is desirable, since there are many more channels than buffers. This increases the desirability of a fully associative buffer management scheme. One implementation of such a scheme is described next.

4.4.4. An Alternative Design

In order to remedy the deficiencies described above, an alternative design for a communication component has been studied. Unlike the previous design, this design has been structured in such a manner that the number of I/O ports can be increased without adding unduly to it's complexity. A fully associative buffer management scheme is explored, as well as two types of flow control mechanisms.

A block diagram of the communication component design is shown in figure 4.6. The most distinguishing feature of this design is a single pool of buffers shared by all channels of the component. Since all packets traveling through a node must use this pool, it must provide enough bandwidth to avoid becoming a bottleneck. This is achieved by interleaving the memory 16 ways, assuming packets consist of 16 bytes. Byte i of each packet ($i \leq 0 < 16$) is always stored in memory module i (MM_i). Each of the p ports can simultaneously load a packet into a buffer, provided no two use the same memory module at the same time. In the worst case, p packets simultaneously arrive at a node. Since only one port can be granted access to MM_0, additional registers are required to temporarily buffer the arriving data bytes until they can be stored in MM_0. On the next clock cycle, when the second byte of each packet arrives, one of these newly arriving bytes will be loaded into MM_1, and one of the temporarily buffered bytes can now be written into MM_0. Similarly, three accesses to the buffer pool will occur on the third clock, and so on. Eventually, each port will be able to access a different memory module on each clock cycle.

If the links can transmit one data byte per clock cycle, then the communication component must be able to transport p bytes from the input ports to the memory modules in each clock. A high-speed, time-multiplexed bus performs this function. Since this bus remains entirely within the chip, it can run approx-

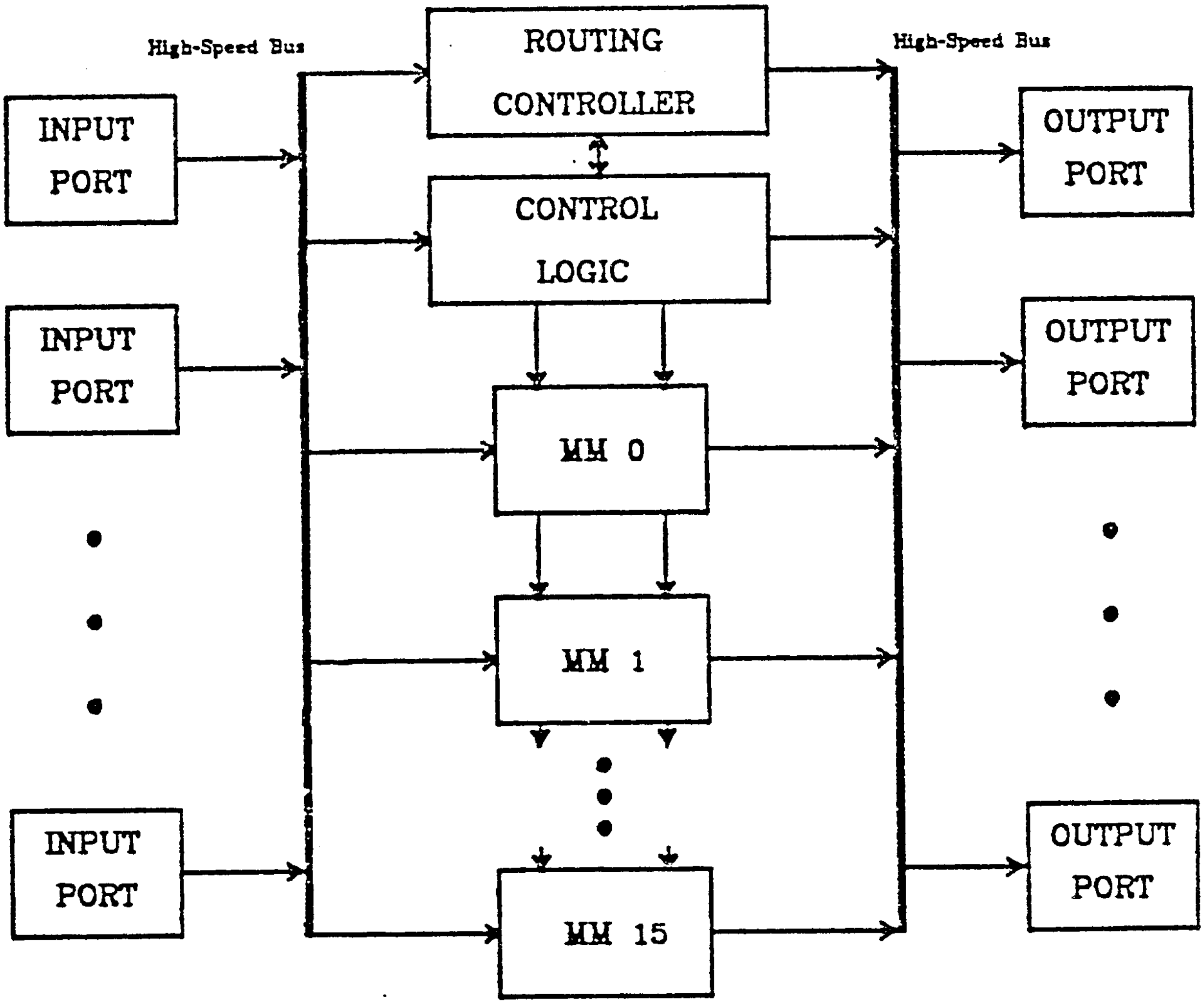


Figure 4.6. *Block diagram of alternative design.*

imately an order of magnitude faster than the I/O links, which require off-chip communications [Sequ78]. A second high speed bus carries bytes from the memory modules to the output ports. Single-port memories can be used in the memory modules provided the control logic only initiates one operation - forward a packet or receive a packet - per clock cycle. The designs which follow assume that this is the case.

A block diagram of the control logic module is shown in figure 4.7. Let us consider the events which occur when a packet arrives at the node. First, the header, i.e. the input channel number, arrives. The translation table is read to determine which output port and channel will be forwarding the packet. The output of the translation table is sent to both the buffer management and flow control modules. The buffer management module allocates an empty buffer to hold the newly arriving packet, and notes the location of this buffer as well as the output port/channel specified by the translation table. This information will be needed when it is time to forward the packet. The buffer module then sends the address of the buffer into MM_0, and the packet is stored, byte-by-byte, into successive memory modules on subsequent clock cycles. The flow control module notes that this output channel now has a packet waiting to be forwarded. When the output link specified by the translation table is free, the flow control logic sends a signal to the buffer management module indicating that the latter should forward the next packet waiting on this output channel. The buffer management module finds the address of the buffer holding this packet and sends it to MM_0. The packet is read from the buffer byte-by-byte, and forwarded over the output link. In both reading and writing a packet, the same memory address is pipelined from memory module to memory module on successive clock cycles. Since the pipelined structure of the memory modules allows the reading, i.e. forwarding, of a packet to begin before all of it arrives, virtual cut-through is easily implemented.

The sections which follow give more detailed explanations of possible hardware implementations of these mechanisms. The next section describes two implementations of a fully associative buffer management scheme which differ in the number of buffers each virtual circuit can hold at one time. It is seen that a significant reduction in complexity is possible if this number is restricted to one. Following this, two possible implementations of flow control mechanisms are presented. The first uses a send/acknowledge protocol, while the second uses a remote buffer management scheme.

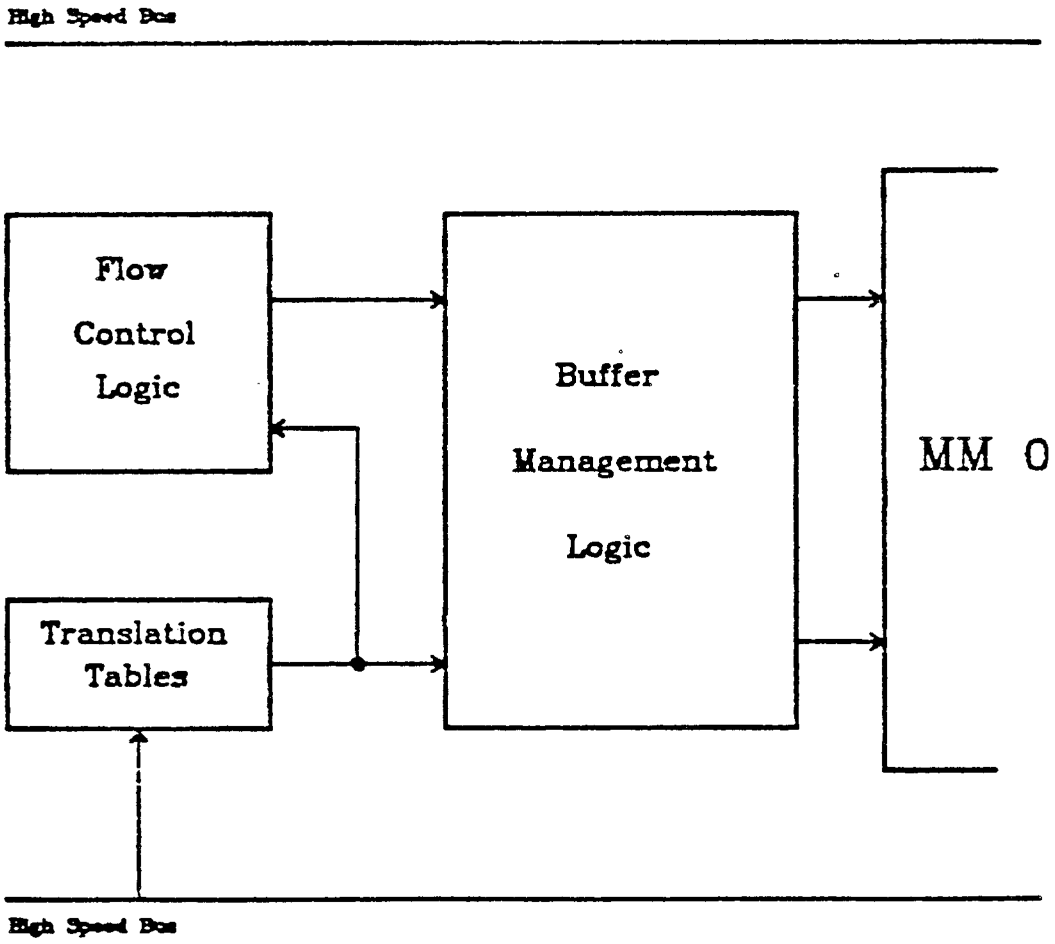


Figure 4.7. *Control logic.*

In the circuit diagrams which follow, the widths of data paths are based on a component with 4 I/O ports, 32 data buffers, and 128 channels per link. Thus, port numbers, buffer numbers, and channel numbers are 2, 5, and 7 bits in length respectively. These choices will be discussed later in this chapter.

4.4.4.1. Buffer Management Hardware

The buffer management module must perform two functions:

(1) Locate a free buffer to hold a newly arriving packet.

(2) Locate the next packet waiting to be forwarded on a particular output channeL

Two implementations will be described for performing these functions. The first assumes that the number of packets waiting to use a given output channel can be larger than one. The second restricts this number to be at most one.

Since buffers are dynamically assigned to virtual channels on demand, a mechanism is required to keep track of which buffers are assigned to which channels at any given time. In the presented solution, this task is accomplished by "chaining" the buffers waiting to be forwarded on an output channel into a linked list for that channel. When a packet arrives, it is placed at the end of the linked list corresponding to the channel the packet is to be forwarded on (read from the translation table). It is removed from the list after it has been successfully transmitted to the next node. The linked lists are managed as a FIFO queue to ensure that packets are forwarded in the same order in which they arrivėd The mechanisms for managing the linked lists are implemented in hardware so that packet forwarding can proceed as quickly as possible. A block diagram of one implementation is shown in figure 4.8a. In the discussion which follows, it is assumed that each p-port component has b buffers and c input (or output) channels per port, i.e. $c \times p$ channels per node.

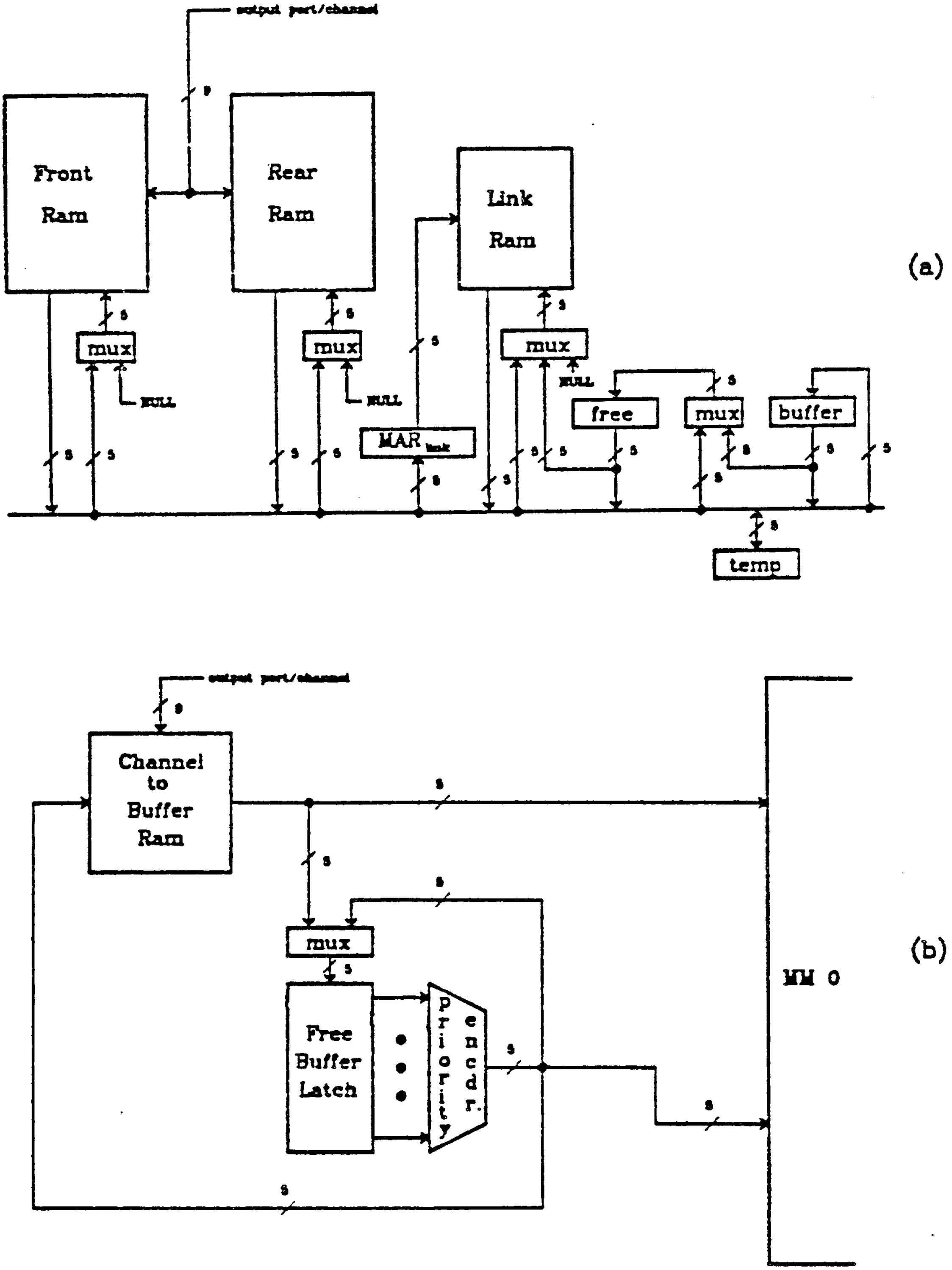


Figure 4.8. *Buffer management circuitry (a) >1 buffers/channel. (b) 1 buffer/channel.*

A buffer consists of a 16 byte data portion, which is physically distributed across the memory modules in figure 4.6, and a pointer word. The pointer word indicates the address of the next buffer in this buffer's linked list. The b-word "link" RAM in figure 4.8a holds these pointers. Each output channel has pointers to the buffers at the front and end of its linked list. The $c \times p$-word "front" and "rear" RAMs in figure 4.8a perform this function. Adding a new buffer to an output channel implies reading the rear RAM (to find the last buffer in the list), and writing the address of the new buffer into this address of the link RAM (to set the new link) as well as the rear RAM (to set the pointer to the new rear element). Deleting an entry implies reading the front RAM (to get the address of the buffer being deleted), reading the link RAM (to get the new front element), and writing this latter address into the front RAM.

Buffers not linked to any channel list are empty, and are linked together in a separate "free list". A register, called the "free" register, points to the beginning of the free list. The arrival of a new packet implies removing an element, i.e. the address of a free buffer, from the free list, and adding this address to an output channel's linked list. Forwarding a packet implies removing the front element from the channel list, and adding it to the free list. Allowing simultaneous access to different memories, a buffer can be added to or deleted from a linked list in four and three clock cycles respectively (where each memory reference requires one clock cycle). The operations necessary to process an arriving/departing packet are shown in figure 4.9 below.

The complexity of the design described above can be reduced significantly if the restriction is made that any output channel can use at most *one* buffer at a time. The impact of this restriction on system performance will be discussed later. Most of the hardware for managing the linked lists can be eliminated, since the lists are at most one element long. This allows the three RAMs in figure

```
Packet arrives on channel "ich":
 clock cycle   action                         comments
   1)          buffer ← free;                 address of free buffer
               $MAR_{link}$ ← free;           get ready to read new free list head
               if (free = NULL) abort;        no more free buffers
   2)          free ← Link[$MAR_{link}$];     read new free list head
   3)          Link[$MAR_{link}$] ← NULL;     mark pointer for new buffer
               temp ← Rear[ich];              locate end of linked list
               $MAR_{link}$ ← Rear[ich];      get ready to add to end of list
   4)          Rear[ich] ← buffer;            update pointer to end of list
               if (temp = NULL)               if channel list now empty
                 Front[ich] ← buffer;         then update front pointer
               else
                 Link[$MAR_{link}$] ← buffer; else update previous last element

Packet forwarded on channel "och":
 clock cycle   action                         comments
   1)          buffer ← Front[och];           get address of first buffer in list
               $MAR_{link}$ ← Front[och];
   2)          if (buffer = NULL) abort;      abort if list empty
               temp ← Link[$MAR_{link}$];     address of new front element
   3)          Front[och] ← temp;             update front pointer
               Link[$MAR_{link}$] ← free;     add buffer to free list
               free ← buffer;                 new front of free list
               if (temp = NULL)               check if list now empty
                 Rear[och] ← NULL;
```

Figure 4.9. *Operations to send and receive packets.*

4.8a to be combined into one RAM, the "channel-to-buffer" RAM shown in figure 4.8b. This $c \times p$-word RAM maps output channels to buffer addresses. Word i holds the address of the buffer currently holding a packet for channel i. The list of free buffers is replaced by a b-bit latch, called the "free buffer latch". The free buffer latch is implemented as a bit-addressable latch, i.e. a memory device which is written as a RAM (one bit at a time), but read as a latch (all bits in parallel). Each bit indicates the status of a buffer: free (1) or in use (0).

When a new packet arrives, the buffer management circuitry must perform two operations, assuming the flow control circuitry has first established that the packet can be accepted (discussed later):

(1) Find and allocate a free buffer.

(2) Note the location of the buffer so that the packet can be found when it is time to forward it.

The address of a free buffer is determined by a priority encoder attached to the free buffer latch. The resulting address is sent to MM_0. This address is also used to clear the corresponding bit in the free buffer latch, effectively allocating the buffer, and completing the first operation. The second operation is accomplished by writing the address of the selected buffer number into the channel-to-buffer RAM at the memory location corresponding to the output channel responsible for forwarding the packet (read from the translation table).

Forwarding a packet on output channel i also requires two operations:

(1) Locate the buffer holding the packet for channel i.

(2) Release the buffer.

The first task is accomplished by reading address i of the channel-to-buffer RAM. The resulting address is used to set the corresponding bit in the free buffer latch, marking the buffer free to be used by other packets, thus accomplishing the second task.

Forwarding a packet requires the time of two memory operations since the channel-to-buffer RAM read must be completed before the latch write can be begun. These two steps are easily pipelined however, allowing a "send packet" operation to be initiated every clock cycle. The operations for receiving a packet can be performed in a single clock cycle since both can be executed concurrently. This is in contrast to the four clock cycles required in the previous buffer management scheme which used linked lists.

4.4.4.2. Flow Control Hardware: Send / Acknowledge Protocol

In the send/acknowledge flow control scheme, each node sends a packet, and receives an acknowledgement signal indicating whether the receiver accepted or rejected (i.e. discarded) the packet. Packets may be rejected because of buffer space limitations or transmissions errors in the header and must be retransmitted at some later time. As discussed earlier, it is assumed that each link uses a separate control line to carry the acknowledgement signal back to the sender with minimal delay.

In order to prevent buffer hogging, each output channel may not use more than some "channel limit" of buffers at any one time. In addition, each output port may not use more than some "port limit" of buffers. Note that the port and channel limits only restrict the number of buffers the port and channel can use, and do not represent an a priori allocation of buffer space.

A block diagram of the flow control circuitry for one port is shown in figure 4.10a. The circuitry performs two functions:

(1) It selects a channel which is waiting to use the link and initiates a request (to the buffer manager) to forward the next packet on this channel.

(2) It accepts or rejects arriving packets.

Note that the flow control circuitry does not deal with buffer numbers. The buffer manager keeps track of which buffers are assigned to which channels.

The first function is accomplished by the "channel FIFO" shown in figure 4.10a: This memory lists channels with packets waiting to be forwarded. When the link is ready to forward a packet, the first element of the channel FIFO is removed. The resulting channel number is sent to the buffer management circuitry indicating that the next packet on this output channel is to be sent over the link. If this packet is accepted by the neighboring node, the FIFO element is

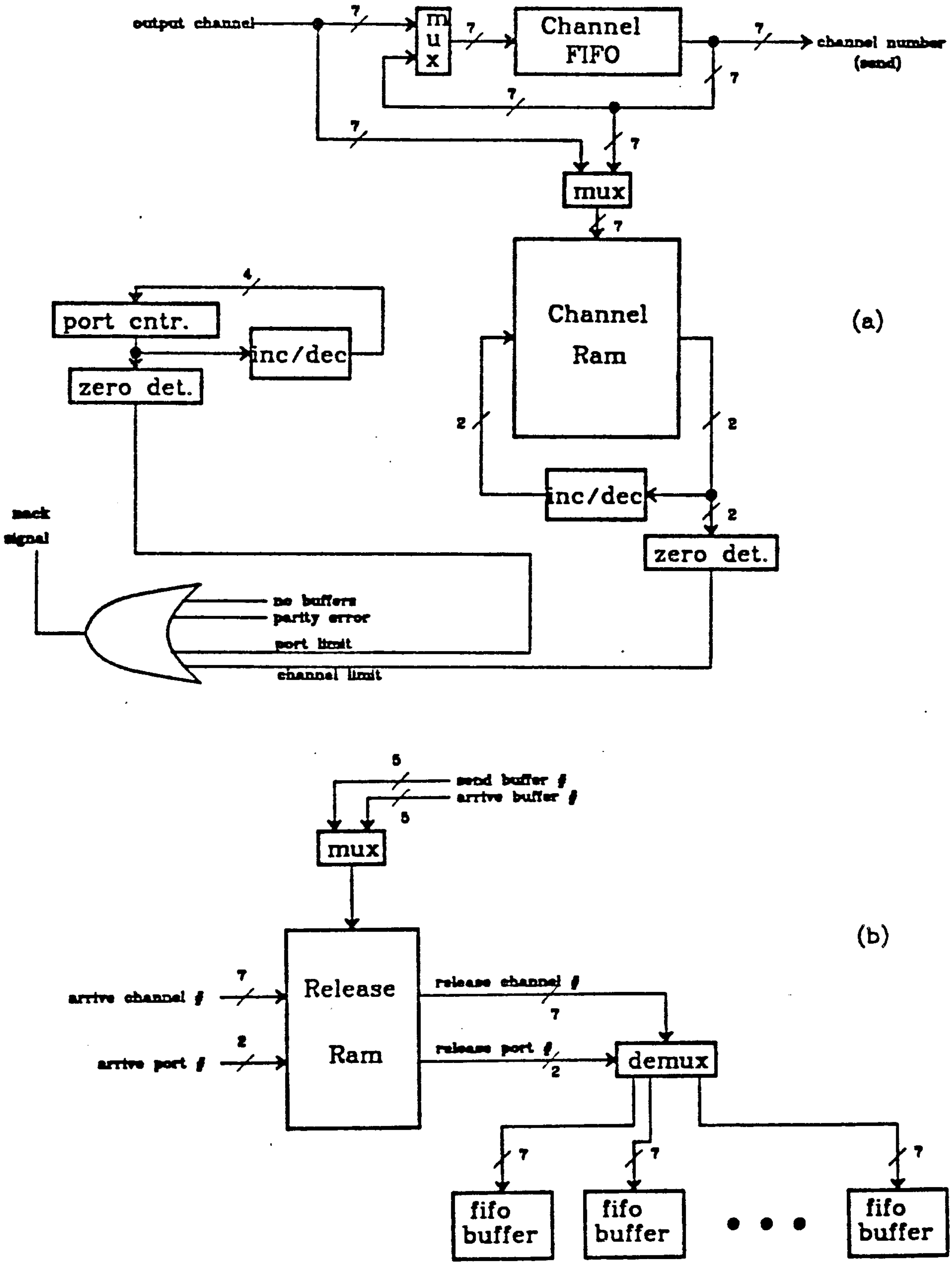


Figure 4.10. *Flow control circuitry (a) send/acknowledge.*
(b) additional circuitry for remote buffer management.

discarded. Otherwise, the channel number is reentered at the end of the FIFO.

The remaining circuitry in figure 4.10a performs the second function: determine whether an arriving packet should be accepted or rejected. A packet may be rejected for any of four reasons:

(1) No buffers remain in the node to hold the packet.

(2) The parity check on the header indicates a transmission error.

(3) The output port is already using the maximum number allowed.

(4) The output channel is already using the maximum number allowed.

The flow control hardware must detect each of these cases, and generate a negative acknowledgement should any of them arise. If none of the conditions arise, the packet is accepted and a positive acknowledgement returned.

The first condition is detected by a control line from the buffer management module indicating whether the free buffer pool has been exhausted. A NULL pointer in the free register in figure 4.8a, or a lack of '1' bits in the free buffer latch in figure 4.8b indicates this condition. Similarly, a parity checker in the input port detects the second condition. A "port counter" is used to detect the third. This counter indicates the number of additional buffers that port can use before the port limit is reached. The counter is initially set to the port limit, decremented each time a packet is accepted, and incremented each time the port successfully forwards a packet. If the output of the counter is zero, the port cannot accommodate another packet. The zero-detection circuitry, implemented by a single nor gate, identifies this situation.

In order to detect the final condition, a channel using its limit of buffers, circuitry similar to the port counter is required for each output channel. The c-word "channel RAM" indicates the number of buffers each channel can use before reaching its channel limit. Entry i is initially set to the channel limit for channel i, is decremented each time a packet is accepted by that output channel, and is incremented when the channel successfully forwards a packet. If a packet arrives and the corresponding channel RAM entry is zero, the packet must be rejected and a negative acknowledgement returned.

If the restriction is made that each channel can use at most one buffer at a time, then the circuitry in figure 4.10a can be simplified. The channel RAM is now one bit wide, and indicates whether or not the channel has a packet waiting to be forwarded. The increment/decrement circuit attached to the channel RAM is no longer needed, since accepting a packet implies setting a bit in the RAM, and forwarding a packet implies reseting a bit. Similarly, the channel RAM's zero-detection circuitry is not required, since only a single bit is output from the RAM.

4.4.4.3. Flow Control Hardware: Remote Buffer Management

In this scheme, each output port maintains enough information about buffer allocation in its neighboring nodes to determine which channels may send packets, and which must wait. A channel must wait if it is using "too many" of its neighbor's buffers, or if there is insufficient free buffer space to hold a new packet. Control decisions are made by the sender, and receivers must accept all packets sent over the link. Eventually the receiver will forward the packet to a third node. When this happens, the receiver reports back to the sender by sending a "release channel number" indicating that the buffer is again available for another packet. This indicates the number of the channel which originally sent the packet.

In addition to restricting the number of buffers each channel can use at one time, a "port limit" restricts the number of buffers each output port can use. Both the port and channel limits are initialized by the local routing controller. If the port limit is pl, then pl buffers are reserved in the neighboring node for use by this output port. This is in contrast to the send/acknowledge design in which the port limit only restricts the number the port can use, but does not actually reserve buffer space. Because no retransmissions are used, the remote buffer management scheme must reserve buffers to avoid overflows, since this will result in lost packets.

The flow control circuitry must perform four functions:

(1) It selects a channel which is waiting to use the link, and initiates a request (to the buffer manager) to forward the next packet on this channel.

(2) It generates release channel numbers to previous nodes.

(3) It processes incoming release channel numbers.

(4) It maintains information of buffer allocation in receiving nodes.

The physical circuitry for performing the first function is virtually the same as that shown in figure 4.10a for the send/acknowledge scheme, however the logical meaning of the information kept in the channel RAM is different. Instead of indicating the number of local buffers below the channel's limit in the local node, the RAM indicates the number of *remote* buffers below the limit in the neighboring node. As long as entry i is not zero, channel i may send another packet, assuming the port limit has not been reached. Similarly, the port counter also refers to buffers in the neighboring node used by this output port.

When a packet arrives, the number of the output channel responsible for forwarding the packet is added to the end of the channel FIFO. Since all packets are accepted, no further processing is required. To forward a packet, the next entry in the channel FIFO is removed. The corresponding channel number is used to address the channel RAM. If the corresponding entry of the channel RAM is not zero, and if the port counter is not zero, then the channel number is sent to the buffer module and the next packet on this channel is sent over the link.

The port counter and channel RAM entry are then decremented. If either of the counters was zero, the channel cannot forward the packet so the channel number is reentered at the end of the channel FIFO.

Each time a packet is forwarded, a release channel number must be sent back to the neighboring node which sent the packet. The circuitry in figure 4.10b performs this function. In order to generate release channel numbers, information must be kept with each packet that indicates which input port and channel it arrived on. A b-word "release RAM" accomplishes this task. Element i indicates the input port and channel number the packet in buffer i arrived on. This information is loaded into the release RAM when a packet arrives and read when it is forwarded. A small fifo buffer in each output port holds the channel number portion of the word until it can be forwarded to the neighbor which sent the packet.

Finally, when a release channel number is received from a neighboring node, indicating that a certain channel is using one fewer buffer, the count of buffers the channel is allowed to use must be incremented. The appropriate entry of the channel RAM is read, incremented, and written back into the RAM, completing the processing of the release.

The simplifications resulting from constraining each output channel to using at most one remote buffer at a time are similar to those described in the send/acknowledge scheme. The channel RAM is again one bit wide. Forwarding a packet resets a bit in the RAM, effectively disabling the channel. Receiving a release channel number causes the bit to be set, reenabling the channel.

4.5. Evaluation of Communication Component Parameters

In order to evaluate the amount of circuitry required to implement the communication component described above, estimates are required of:

(1) the number of I/O Ports

(2) the number of virtual channels

(3) the number of buffers.

Chapters 2 and 3 examined the first question in detail and concluded that from 3 to 5 I/O ports should be used. The remaining two questions will be discussed next.

4.5.1. Number of Virtual Channels

Because each virtual channel requires a certain amount of overhead circuitry, the number of channels must be limited. In addition, it is desirable to limit the number of channels on each link to prevent "overbooking" the link's bandwidth, since this will lead to long queueing delays on the link and to poor performance. On the other band, providing too few channels per link will lead to a high failure rate in establishing virtual circuits, deadlock situations, and underutilization of the link's bandwidth. Thus the number of virtual channels per link must be chosen to achieve good link utilization without incurring an excessive amount of overhead circuitry.

First, link utilization will be used to determine the proper number of channels per link. The overhead issue will be ignored for now. Deadlocks can be broken with an end-to-end timeout mechanism, as will be disussed later.

Each virtual circuit using a link requires a certain amount of bandwidth. Since the bandwidth provided by each link is fixed, each link can support a large number of circuits with low bandwidth requirements, or a small number of circuits with high bandwidth requirements. If a large number of channels are provided to accommodate the former case, a number of high bandwidth circuits may use the link and overbook the available bandwidth. If a small number of channels are provided, much of the link's bandwidth will be wasted when many low bandwidth circuits monopolize the available channels.

One approach to resolving this dilemma is to provide enough channels to accommodate a large number of low bandwidth circuits, but to also provide a separate mechanism which prevents overbooking the link's bandwidth. New circuits may not be established on the link if the link's bandwidth has been fully booked, regardless of the number of unallocated channels remaining. Later, when existing circuits using the link are torn down, new circuits could again be established.

In order to implement this mechanism, the bandwidth requirements of each circuit must be estimated. This could be accomplished statically when the circuit is established (e.g. the operating system may be able to provide this information based on the type of traffic expected over the circuit), or dynamically, "on the fly", by measuring traffic on the circuit. Of course, the latter scheme has the disadvantage that link bandwidth may still be overbooked since the amount of bandwidth required by the circuit is not known until after it is established, i.e. a high bandwidth circuit may be established over the link before it is known that its bandwidth requirements overbook the link.

Both the static and the dynamic schemes could be implemented by associating a "link bandwidth indicator" with each output link which indicates the anticipated bandwidth requirements of circuits using the link. When this bandwidth indicator exceeds some threshold, no more traffic is routed over that link. In the first scheme, using "hints" from the operating system, a field in the packet which sets up the virtual circuit could indicate the anticipated bandwidth requirements of that circuit. The link indicator is increased by the value of this field when the circuit is set up, and decreased when the circuit is torn down. The value of this field must be included in the packet tearing down the circuit as well as the header, since the component does not keep track of the bandwidth requirements of each channel.

In the dynamic scheme, the bandwidth indicator could be incremented each time a packet is sent on that link. Periodically, the routing controller examines the indicator to determine if the link is overbooked, and then clears it. Finally, a third alternative is to measure the average queue length on each link, and declare the link overbooked if this average exceeds a certain threshold. Again however, these two schemes do not prevent overbooking the link's bandwidth, but rather attempt to prevent a bad situation from becoming worse.

If a separate mechanism is used to prevent overloading the link, each component should ideally provide an unlimited number of channels since this guarantees that it will never needlessly block circuits trying to use a link with excess capacity. This number can be reduced however, if the minimum bandwidth requirements of any virtual circuit can be established. The maximum number of channels the link will ever require can be calculated by dividing the total link bandwidth by this minimum channel bandwidth, as will be derived below.

In this context, two questions must be considered:

(1) How many minimum bandwidth circuits can be maintained on a fixed bandwidth link?

(2) How much traffic corresponds to a minimum traffic load?

The first question lends itself to a precise mathematical analysis, and will be discussed next. The second is more difficult to resolve since it is application program dependent. It will be addressed later.

Given an expected traffic load on each circuit, the proper number of channels can be estimated via a queueing model. The model for the traffic load on each communication link is shown in figure 4.11. The n virtual channels using

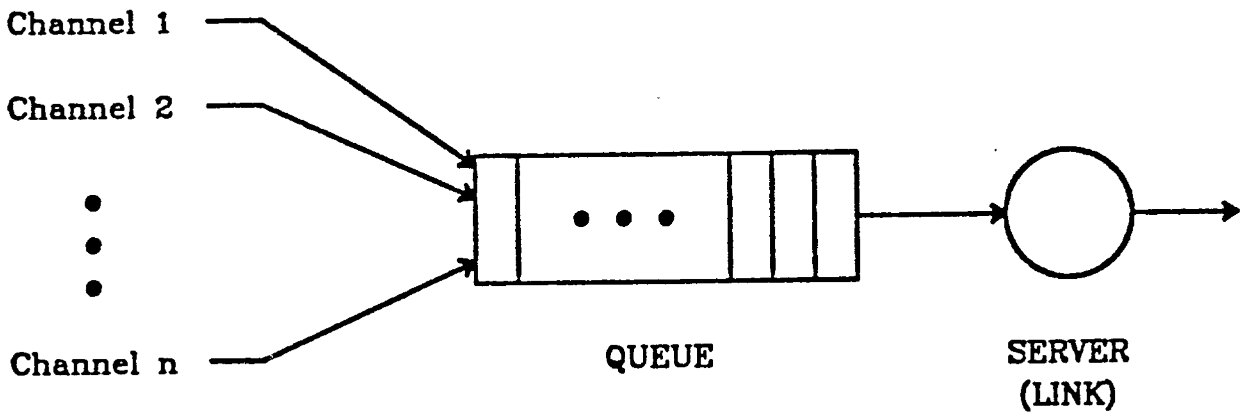


Figure 4.11. *Queueing model for analyzing number of channels.*

the link are modeled as a single server queue with traffic arriving from n sources. It will be assumed that packet arrival times on each virtual circuit follow a Poisson distribution. Since fixed-length packets are used, service times are deterministic, resulting in an M/G/1 queueing model. From the Pollaczek-Khinchin mean value formula [Klei75], the average time $\bar{W}$ each packet spends in the queue waiting for the link is

$$\bar{W} = \frac{\bar{x}\rho}{2\,(1-\rho)}$$

where $\bar{x}$ is the service time, i.e. the time required to transmit a packet, and ρ is the link utilization. Assuming the average arrival rate on each of the n virtual circuits is λ messages per second, or λm_l bits per second, where m_l is the packet length in bits, the utilization of a link with a bandwidth of b bits per second is

$$\rho = \frac{n\lambda m_l}{b}\,.$$

Since the service time is $\bar{x} = m_l / b$, we find

$$\bar{W} = \frac{n\lambda m_i^2}{2b\,(b - n\lambda m_i)} .$$

To determine numerical estimates of the number of channels, consider the number of virtual circuits required to drive the link utilization ρ to 1, which in turn drives the average waiting time to infinity:

$$n = \frac{b}{\lambda m_i}$$

Figure 4.12 shows a plot of this quantity for an 80 Mbit/second link (a byte wide link running at 10 Mhz) as a function of $1/\lambda$, the mean time between packets on each circuit. This plot also assumes that packets are 17 bytes in length.

The critical parameter in evaluating the number of channels is the expected traffic load on each virtual circuit. Unfortunately, the bandwidth requirements of each circuit may be arbitrarily small, implying an arbitrarily large number of channels should be supported. In reality however, seldomly

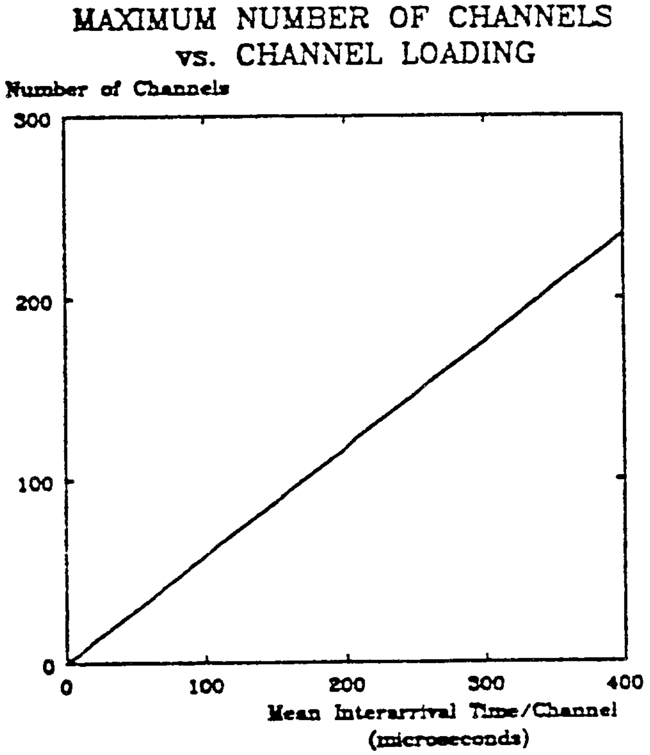


Figure 4.12. *Number of channels vs. load per channel.*

used virtual circuits may be torn down and reestablished as necessary to reduce the number of channels required. Since reestablishing a circuit incurs considerably more delay than sending a message on an existing circuit, these lightly loaded circuits must not have low latency requirements.

In order to determine numerical estimates of virtual circuit loading, the average time between messages on virtual circuits was measured for the five application programs described in chapter 3 (the artificial traffic load program is excluded) assuming negligible communication delays. These arrival rates are shown in table 4.1 below, and represent rates averaged over all virtual circuits in the application program weighted according to the number of messages sent on each circuit. If n_i and λ_i are respectively the number of messages sent and the average arrival rate on virtual circuit i, then the overall average arrival rate is computed as:

$$\lambda = \frac{1}{\sum_i n_i} \sum_i n_i \lambda_i .$$

Standard deviations for the interarrival times are also shown in table 4.1. The zero value in the FFT program is due to the regularity of its structure: All tasks iteratively perform the same computation, so the time between messages is always the same. The other values reflect the fact that the programs typically use two types of circuits - those with little or no computation between messages, e.g. the circuits distributing the initial data samples in the signal processing

Table 4.1
AVERAGE ARRIVAL RATES PER VIRTUAL CIRCUIT

PROGRAM	ARRIVAL RATE (msgs./sec.)	INTERARRIVAL TIME (microseconds)	STANDARD DEVIATION (microseconds)	NUMBER OF CHANNELS (80 Mbit link)
BARNWELL	9940	100.6	40.4	59
FFT	17200	58.0	0.0	34
BLOCK	29200	34.2	43.8	20
JORDAN	45200	22.1	23.2	13
LU	85500	11.7	9.2	7

programs, and those with more significant computations between messages. These latter circuits have an average interarrival time which is much larger than the former.

Figure 4.13 shows the average waiting time to use an 80 Mbit/second link loaded with virtual circuits which each carry an artificial traffic load corresponding to the average values listed in table 4.1. The curves show that up to 59 channels should be allowed for the program exhibiting the lightest traffic load —the Barnwell signal processing program. This value is also listed in table 4.1 along with the corresponding values for the other application programs. Thus, for workloads similar to those discussed here, it would be reasonable to provide up to 64 channels on each link. If one considers the standard deviation on the Barnwell program, one could argue that this figure should be raised to 128, since it is possible that most of the circuits using a link could by chance fall below the average arrival rate.

As discussed earlier, the tasks in the application programs listed in table 4.1 communicate relatively frequently. Programs requiring less frequent communication use circuits that are more lightly loaded, implying each link could support even more channels. Thus, the figure derived above should be considered a lower bound rather than an absolute estimate of the number of channels. Since there may be any number of circuits which communicate infrequently, but which require low latency (making it unreasonable to tear down and reestablish the circuit each time a message is sent), more than 64 channels should be provided. However, increasing the number of channels increases the size of the channel number that must precede each packet, reducing the amount of bandwidth available for transmitting data. A compromise between these conflicting considerations is to provide 128 or 256 channels per link, since this allows more than 64 channels, but still confines the channel number over-

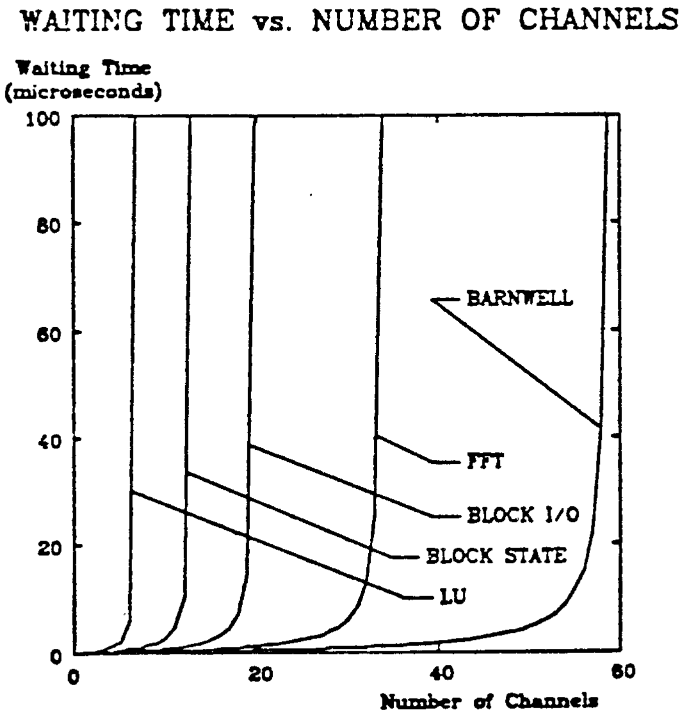


Figure 4.13. *Waiting time vs. number of channels for various programs.*

head to a single byte.

Finally, let us consider the impact future improvements in technology will have on the number of channels. Both computing speeds and link bandwidths can be expected to improve. Higher bandwidths imply that each link could support more channels. On the other hand, higher computation rates lead to higher traffic loads, implying fewer channels are necessary. If switching speeds increase at a faster rate than communication rates, then we can expect virtual circuit loading to be the more dominant factor, implying fewer channels per link. On the other hand, if communication rates progress at a higher pace, then more channels may be provided. While switching speeds can be expected to improve by an order of magnitude over the next 20 years [Keye79], fiber optic links may lead to much larger improvements, implying more channels may be supported.

4.5.2. Amount of Buffer Space

Technological capabilities limit the amount of buffer space that can be provided by each communication component. On the other hand, insufficient buffer space will lead to performance degradations, since communication bandwidth is wasted and delays increased if buffers are not available to hold arriving packets..

In extreme cases, buffer deadlock will result. Buffer deadlock is a situation in which message traffic comes to a complete halt because a set of nodes have exhausted all of their available buffer space. Each node cannot forward a packet because no buffer is available to receive it, and each node cannot free a buffer because no packets can be forwarded. An example of such a deadlock situation is shown in figure 4.14, where each node has a single buffer holding a packet waiting to be forwarded. The network will remain deadlocked until a packet is discarded, releasing a buffer.

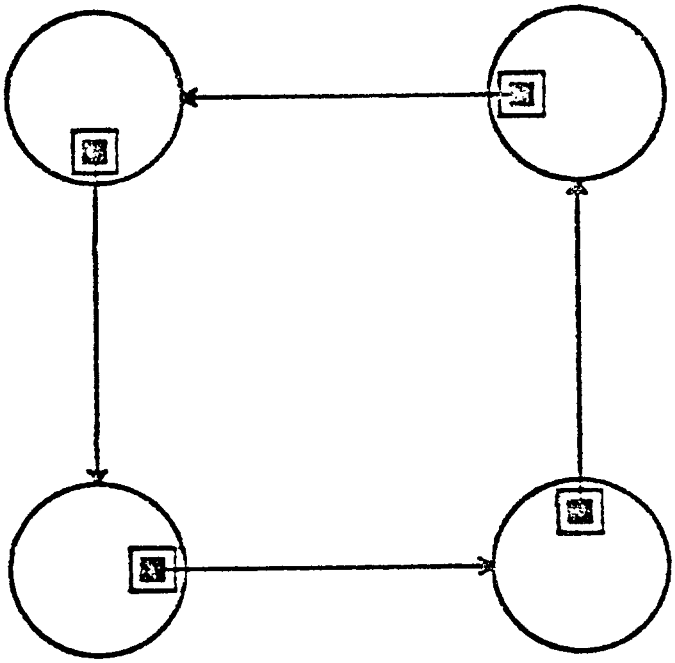


Figure 4.14. *Example of buffer deadlock.*

Thus, sufficient buffer space must be provided to:

(1) avoid buffer deadlock.

(2) ensure good performance.

Each of these issues will now be discussed in turn, followed by results from simulation studies that help to determine the buffering requirements of each communication component.

4.5.2.1. Buffer Space: Deadlock Considerations

Buffer deadlock can be prevented if enough buffer space is provided in each communication component. A brute force solution is to provide each virtual channel with it's own buffer. Since each circuit is allocated a buffer in each node it passes through, traffic on a circuit cannot be blocked by traffic on other circuits, so buffer deadlock cannot occur. Providing a separate buffer on each channel is wasteful however, since each component will have to provide as many buffers as there are channels. It will be seen that virtually the same performance can be achieved if many channels share a much smaller pool of buffers.

An alternative approach to avoiding buffer deadlock is outlined in [Merl80a, Merl80b]. Here, each node must have at least $\overline{H}_{max}$ buffers, where $\overline{H}_{max}$ is the maximum number of hops traversed by any virtual circuit. The buffer pool in each node is partitioned into $\overline{H}_{max}$ disjoint pools or levels, say $1,2, \cdots \overline{H}_{max}$. Each node maintains a "hop count" for each circuit passing through the node indicating the number of hops the circuit has traversed from the source node to the current node. The hop count is set when the virtual circuit is first established. A packet arriving at a node on a circuit with hop count i may only be placed in one of the buffers in level i. It can be shown that buffer deadlock will never occur in this scheme.

The central disadvantage of this scheme is that large networks require more buffers than smaller ones, so the communication component must provide enough buffers to accommodate the largest network it will ever become a part of. This may require an excessively large amount of buffer space. In addition, if traffic is highly localized, many buffers are wasted because those reserved for higher hop counts are never utilized.

Partitioning the buffer pool also adds a certain amount of complexity to the component. The partitioning can be accomplished by limiting the total number of buffers used by circuits at the same level, rather than physically partitioning the buffer space. For example, if $\overline{H}_{max}$ is 10 and there are 20 buffers in the buffer pool, it is sufficient to use the buffer management schemes described above, and ensure that the circuits at any given level collectively use no more than 2 buffers at one time. This could be implemented with a counter at each level indicating the number of free buffers currently available at that level. A packet is rejected if no more buffers are available at it's particular level.

Implementation of this scheme with a remote buffer management policy for flow control is more difficult, since different nodes compete for the buffers in each level. The buffers in each level must be further partitioned among the neighboring nodes to avoid overflow within each level.

A third approach to resolving the deadlock issue is to allow deadlock to occur, but to incorporate a mechanism that ensures that deadlocks are broken. Since detecting and breaking a deadlock may be a time consuming operation, enough buffer space should be provided to ensure that deadlocks occur infrequently. The deadlock breaking mechanism could be implemented as a side effect of an end-to-end protocol using timeout counters to retransmit lost messages. In such a scheme, each message sent over a virtual circuit must be acknowledged by the receiver. If an acknowledgement is not returned after a certain amount of time, the sender assumes that the message was lost and must be retransmitted. In order to avoid duplicate packets, the sender must first "clear" the virtual circuit by sending a special packet which flows through the circuit and destroys all packets it encounters. It then resends the lost message. If deadlock occurs, timeouts will result and circuits will be cleared. This releases buffer space and breaks the deadlock. Since such a timeout mechanism is already required to retransmit *lost* packets, the deadlock breaking mechanism incurs virtually no additional cost. In this scheme, communication components are not required to ensure that buffer deadlock never occurs, so they need not provide excessive amounts of buffer space. Thus this mechanism appears to be an attractive one for solving the buffer deadlock problem.

It might be noted that a similar mechanism could be used to break deadlocks arising when links use all of their virtual channels. Expiration of an end-to-end timeout during the set-up of a virtual circuit could trigger the release of a special packet which destroys the partially completed circuit, releasing channels, and breaking the deadlock. This protocol also requires an end-to-end acknowledgement to mark the establishment of each virtual circuit.

Simulation experiments similar to those discussed in chapter 3 were carried out to evaluate the number of buffers each communication component should provide to avoid buffer deadlock. The six application programs discussed in chapter 3 were executed on Simon with a switch model for a hexagonal lattice network built from 4-port communication components (figure 3.14a).

The first set of experiments assumed that each component provides *b* buffers, and that no restrictions are made on buffer sharing, i.e. virtual circuits may use as many buffers as are available. A large number of buffers, over 100 for some of the programs, was required in each component to avoid buffer deadlock.

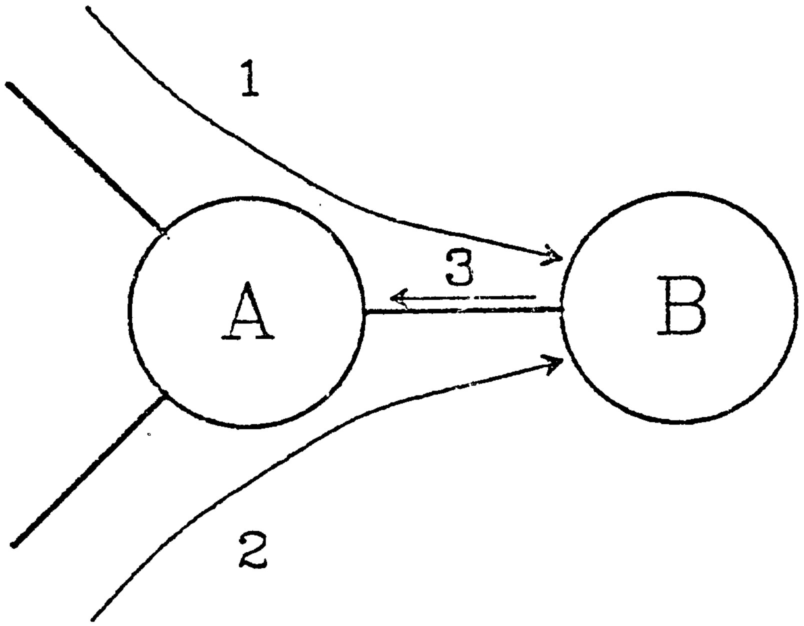


Figure 4.15. *Example of congestion leading to deadlock.*

Buffer hogging was at the root of these deadlock problems. Consider the situation in figure 4.15. Virtual circuits 1 and 2 join at node A, sharing the link from A to B, and virtual circuit 3 uses the link from B to A. Suppose all three circuits carry a steady stream of packets, or equivalently, suppose a burst of packets simultaneously arrives on each circuit. Since the flow of packets into node A on circuits 1 and 2 exceeds the flow from A to B (the latter is limited by the capacity of the link from A to B) a queue begins to form at node A. The queue will grow until the free buffer pool in node A is exhausted. When this happens traffic on circuit 3 is blocked, and a queue of packets begins to grow in node B. Eventually, B's free buffer pool will also be exhausted, blocking traffic on circuits 1 and 2. The network is now deadlocked, and will remain in this state until packets are discarded.

The scenario described above can be avoided if precautions are taken to avoid buffer hogging, e.g. by restricting the number of buffers each circuit can use. The simulation experiments were repeated assuming each circuit could not use more than one buffer in each node at one time. It was found that 12 buffers were sufficient to avoid buffer deadlock in all six application programs.

The simulation experiments thus demonstrate the need to provide a flow control mechanism which prevents buffer hogging. The studies also indicate that, it is reasonable to provide each component with a few tens of buffers, say 32, to reduce the probability of buffer deadlock.

4.5.2.2. Buffer Space: Performance Considerations

Each communication component must provide enough buffer space to maintain a steady flow of traffic through the node. Otherwise, communication bandwidth will be wasted: In the send/acknowledge flow control scheme, retransmissions are required to resend rejected packets, while in the remote buffer management scheme, links simply become idle. How many buffers are required to maintain this flow? Studies of multistage permutation networks, called delta networks, indicate that virtually no performance improvement arises beyond three buffers per node [Dias81a, Dias81b]. Three buffers are not sufficient however, to avoid many deadlock situations. Thus, based on these studies, deadlock rather than performance optimization should be used to determine the amount of buffer space required.

Simulation experiments were carried out to evaluate the following questions:

(1) How many buffers should each component provide to achieve good performance?

(2) How many buffers should each virtual circuit be allowed to use at one time?

(3) How well does the simple send/acknowledge flow control mechanism perform?

Before discussing the results of these simulation experiments, let us anticipate the answers to these questions by deriving an intuitive understanding of the impact of buffering and flow control on network performance.

Buffering and flow control questions are of little consequence when the network is lightly loaded, since buffering requirements are low (buffers start to empty while they are being filled) and throttling mechanisms are not necessary. Therefore, we will only consider the case in which links are congested. An example of a congested link is shown in figure 4.16a. Two circuits, 1 and 2, arrive at a node on links A and B respectively, and share link C. Assume that each circuit carries a continuous stream of packets. Since the combined bandwidths of links A and B is twice that of C, the latter becomes the bottleneck which limits the performance (i.e. bandwidth) of each circuit.

First, let us consider the simplest communication component design in which a send/acknowledge protocol is used for flow control, and where each channel is allowed to use only a single buffer at one time. Figure 4.16b indicates the utilization of the communication links over time. The contents of the buffer held by each circuit is also shown. Successive packets on virtual circuits 1 and 2 are labelled 1a, 1b, 1c ..., and 2a, 2b, 2c, ... respectively. As shown in figure 4.16b, the congested link, C, is fully utilized, and carries packets from both virtual circuits. The end-to-end bandwidth of the two virtual circuits will be equal to half the bandwidth of the C link, assuming traffic in other nodes does not limit performance. Note that most of the packets sent over links A and B must be retransmitted, since the first attempt fails because of the "one buffer per channel" restriction. Yet, these retransmissions do not waste bandwidth on the bottleneck link, C, so the end-to-end bandwidth is not affected. However, the negatively acknowledged packets do reduce the effective bandwidth of other circuits which do not use link C, but which are instead limited by the bandwidth of

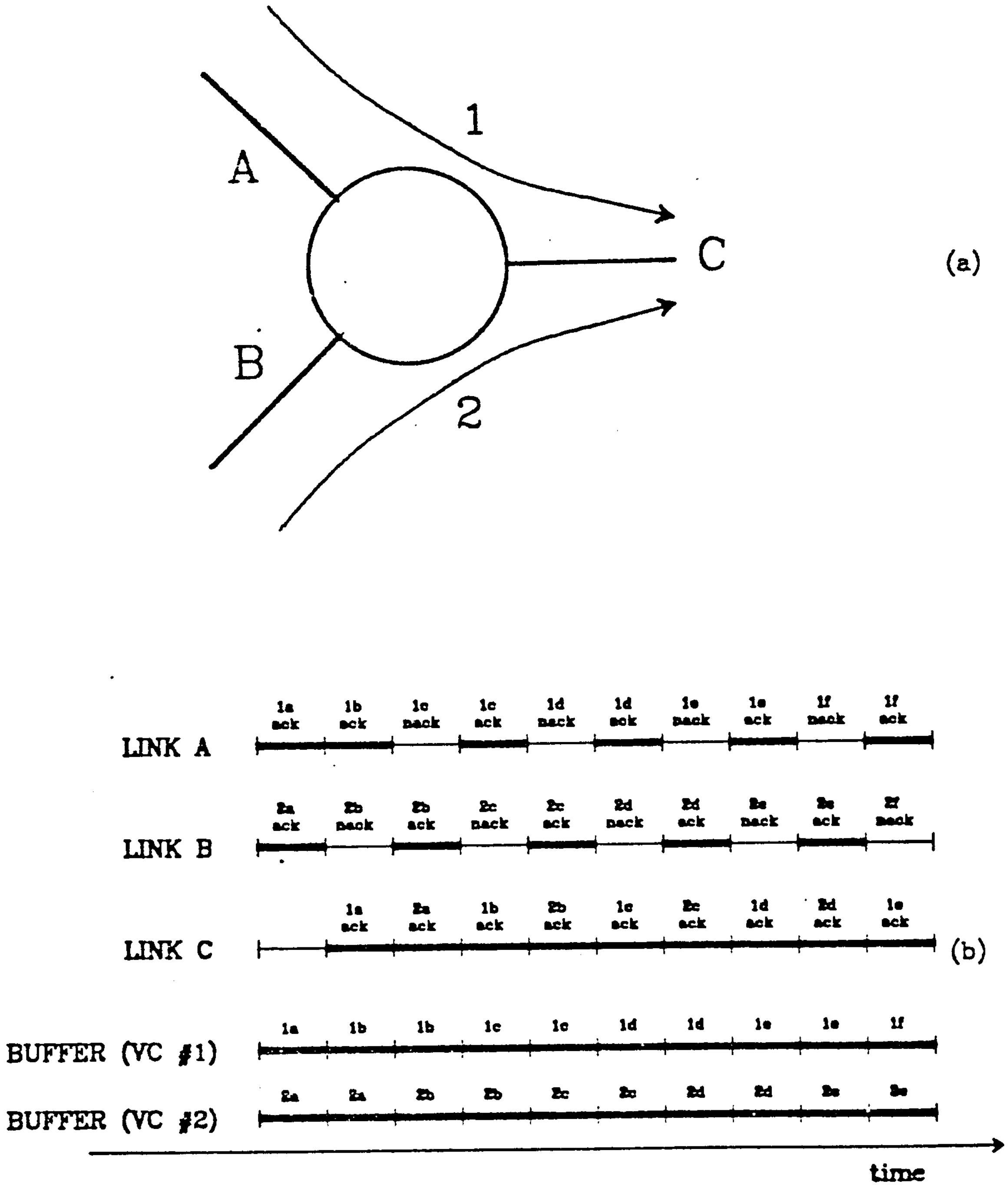


Figure 4.16. *Link usage scenarios (a) Congested link.*
(b) Send/acknowledge protocol, 1 buffer/channel.

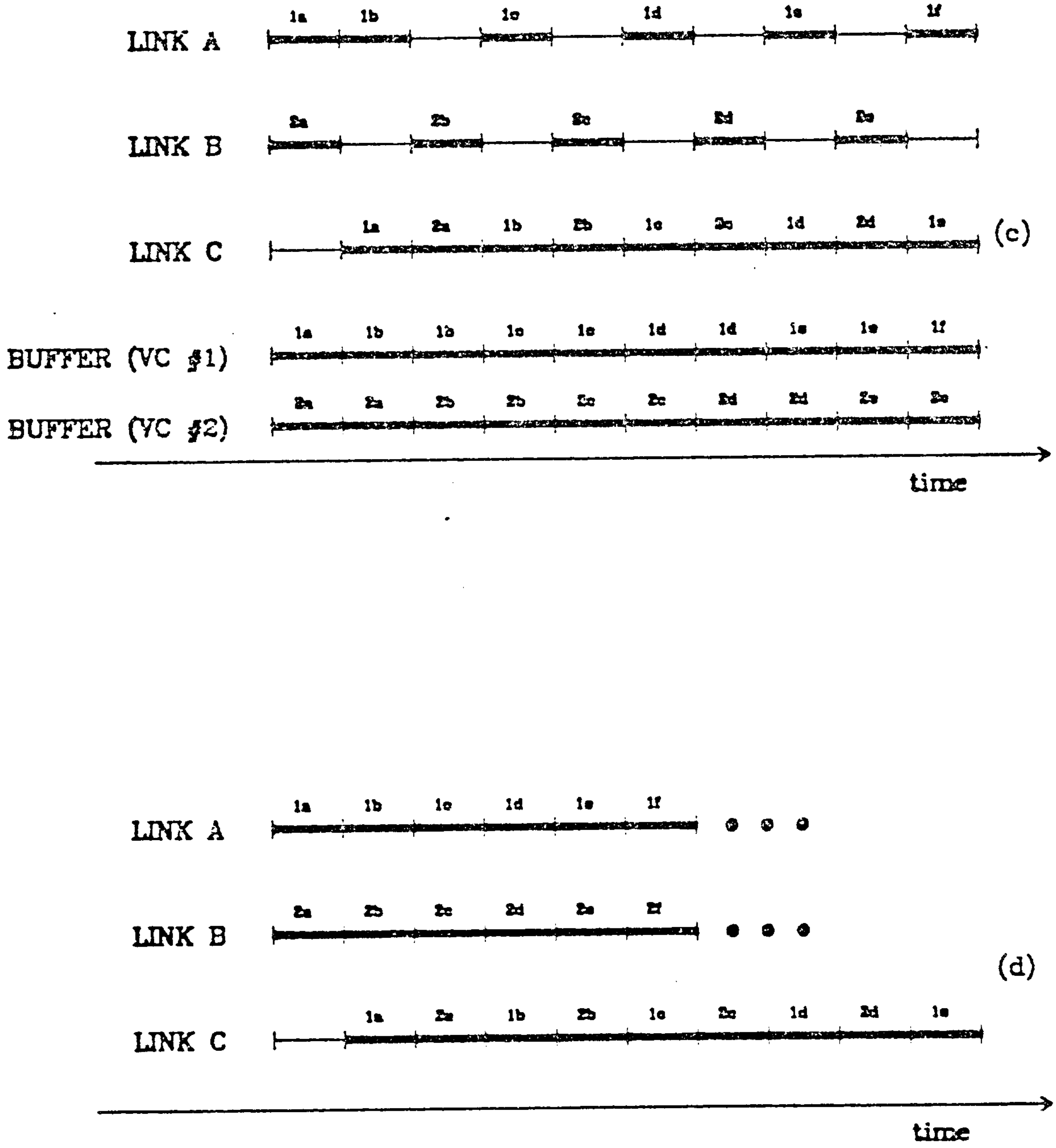


Figure 4.16. *Scenarios (c) Remote buffer management, 1 buffer/channel. (d) Unlimited buffer space.*

links A or B. This will be discussed later.

In figure 4.16c, the send/acknowledge protocol is replaced by a remote buffer management scheme, and circuits are still restricted to using a single buffer at one time. The optimistic assumption is made that each node has "perfect information" about buffer utilization in its neighboring nodes, i.e. the time required to transmit the feedback signal indicating a packet has been forwarded (and consequently, a buffer has been release) is assumed to be negligible. The flow of traffic over the bottleneck link, C, is exactly the same as that when the simpler send/acknowledge protocol is used, so the end-to-end bandwidth of the two virtual circuits remains the same. In comparing figures 4.16b and 4.16c, it is seen that the negatively acknowledged packets on links A and B in figure 4.16b are replaced by idle periods in figure 4.16c.

Finally, in figure 4.16d, an unlimited amount of buffer space is provided in each node. No flow control mechanism is required since there are no buffer overflows, and therefore no reason to throttle traffic. Utilization of the bottleneck link is the same as that of the previous two cases.

Intuitively, buffering is provided in each node to achieve higher network bandwidth by maintaining a large enough "backlog" of traffic in the node so that its output links remain busy under heavy traffic loads. If a node provides enough buffers to keep its links busy, then additional buffers do not improve performance. As demonstrated in figures 4.16b-d, a relatively small amount of buffer space per node is required to perform this function, explaining the results observed in [Dias81a, Dias81b]. Protection against buffer hogging must be provided however, e.g. by limiting the number of buffers each port can use, to ensure that one link does not monopolize the buffer pool and cause other link(s) to become idle.

Figures 4.16b-d indicate that the one buffer per channel restriction does not have a significant impact on network performance. Performance is limited by the bandwidth of bottleneck links, rather than a lack of buffer space. If links are underutilized, then each channel need only provide a single buffer to maintain a steady flow of traffic, as discussed earlier. If links are congested, then there is enough traffic on other circuits to keep the links busy, so no bandwidth is wasted. Performance is not improved by allowing channels to use additional buffers.

These studies also indicate that the send/acknowledge protocol does not adversely affect performance on bottleneck links (link C in figure 4.16b). This flow control mechanism does however, require retransmissions on the links leading up to the bottleneck, implying some wasted bandwidth. Often however, it is the bottleneck link which limits performance, and not the links leading up to the bottleneck, so this wasted bandwidth is of secondary importance. In addition, this waste is only of consequence if there is other traffic waiting to use the link. If other traffic exists, then the amount of wasted bandwidth is reduced, since many of the "negative acknowledgment slots" in figure 4.16b will be replaced by traffic on other virtual circuits, assuming the blocked circuit is not allowed to dominate use of the link. Thus, a more sophisticated flow control mechanism which avoids negatively acknowledged packets, e.g. the remote buffer management scheme described earlier, leads to an even smaller improvement in performance. Since network bandwidth can be improved at a higher level by using more switching chips, the additional complexity of the remote buffer management scheme is difficult to justify. In addition, output port buffer hogging is more difficult to prevent with this more sophisticated scheme (section 4.1.3), so performance may actually be reduced if this scheme is used.

Let us now examine the simulation results to see if they are consistent with the discussion presented above. Figures 4.17a-f show the performance resulting from executing the six application programs discussed in chapter 3 on a hexagonal lattice network built from 4-port communication components. All curves use a send/acknowledge protocol for flow control. If several packets are queued, waiting to use the same link, a round-robin algorithm is used to select the next packet to be sent over the link. This prevents a blocked virtual circuit from dominating use of the link by continuously retransmitting negatively acknowledged packets.

The curves in figure 4.17 are distinguished by the amount of buffer space provided in each communication component, and the degree to which buffer usage is restricted. Four combinations of these two parameters result:

(1) Unlimited buffer space and no restrictions on buffer usage.

(2) Unlimited buffer space but with restrictions on buffer usage.

(3) Limited buffer space and no restrictions on buffer usage.

(4) Limited buffer space but with restrictions on buffer usage.

As discussed earlier, the third class of networks, those with a limited amount of buffer space and no restrictions on buffer usage, resulted in deadlock situations for many of the application programs. Thus, performance of the application programs on these networks is not shown in figure 4.17.

All networks with limited amounts of buffer space provide 16 buffers per communication component. These networks also assume that each output port may not use more than 8 buffers at one time, or $16/\sqrt{4}$ as suggested in [Irla78].

The curves indicate that networks with 16 buffers per component yield virtually the same performance as networks with an infinite amount of buffer space, in agreement with the discussion presented earlier. Restricting virtual

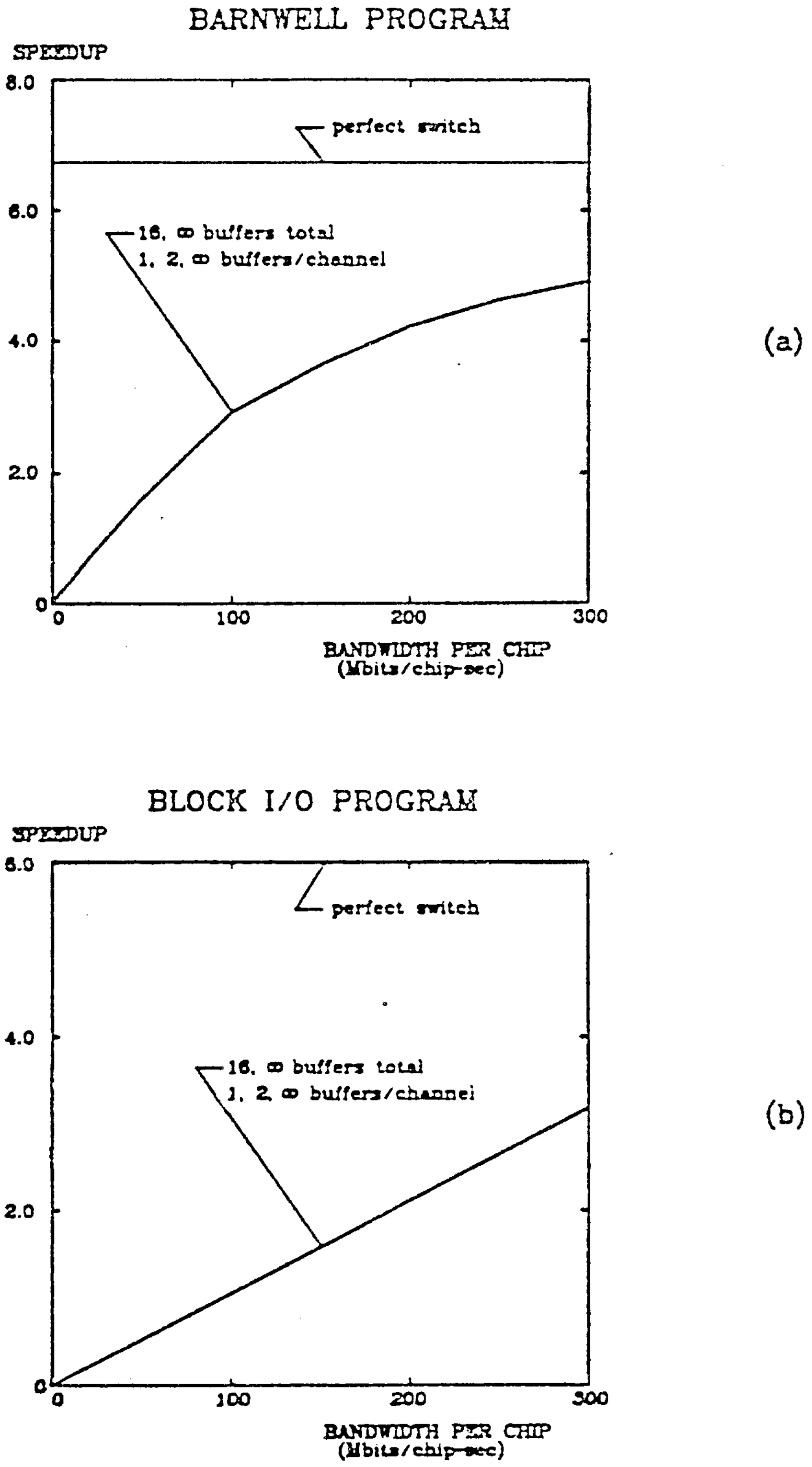


Figure 4.17. *Limited buffer space (a) Barnwell. (b) Block I/O.*

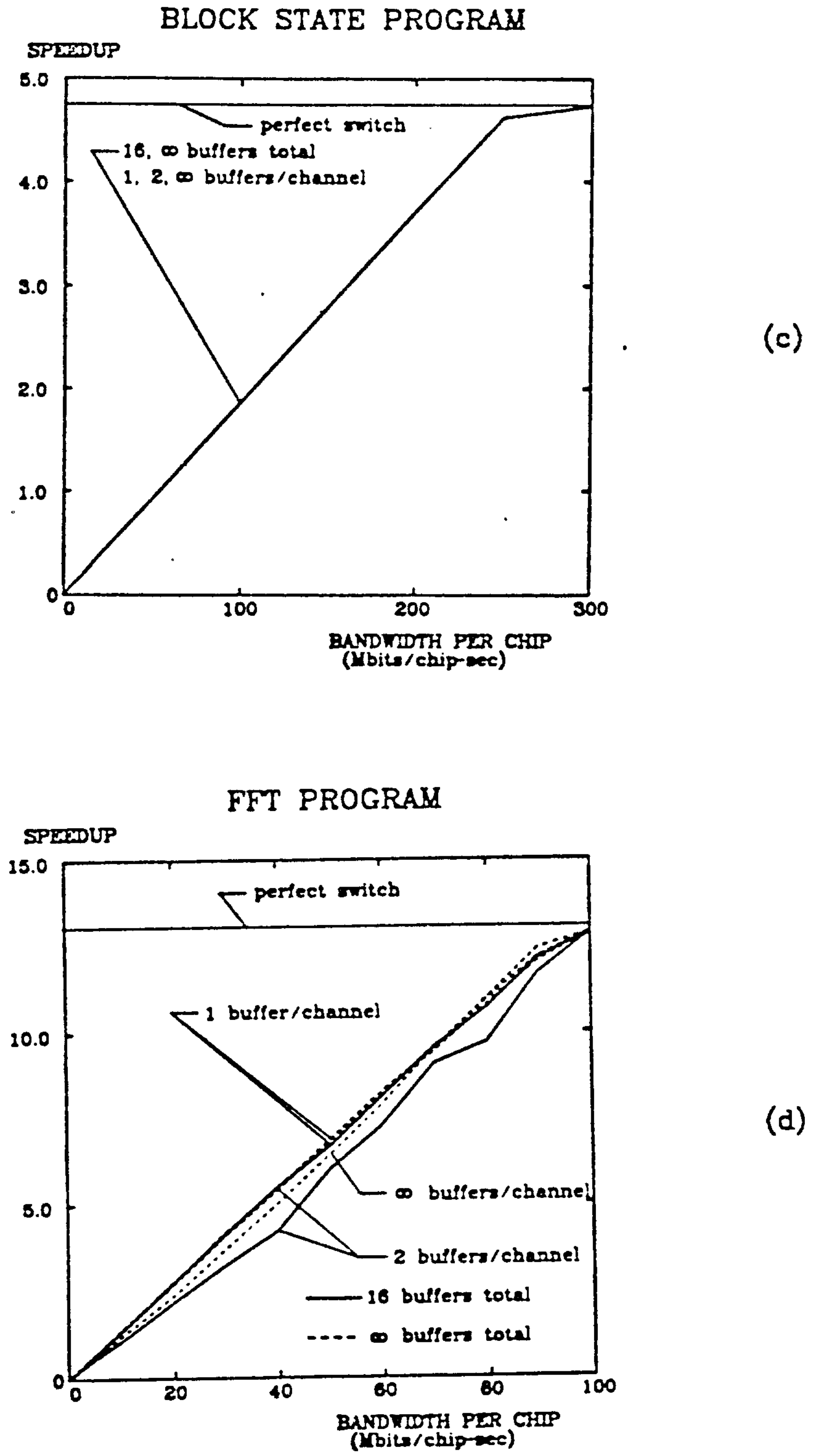


Figure 4.17. *Limited buffer space (c) Block State. (d) FFT.*

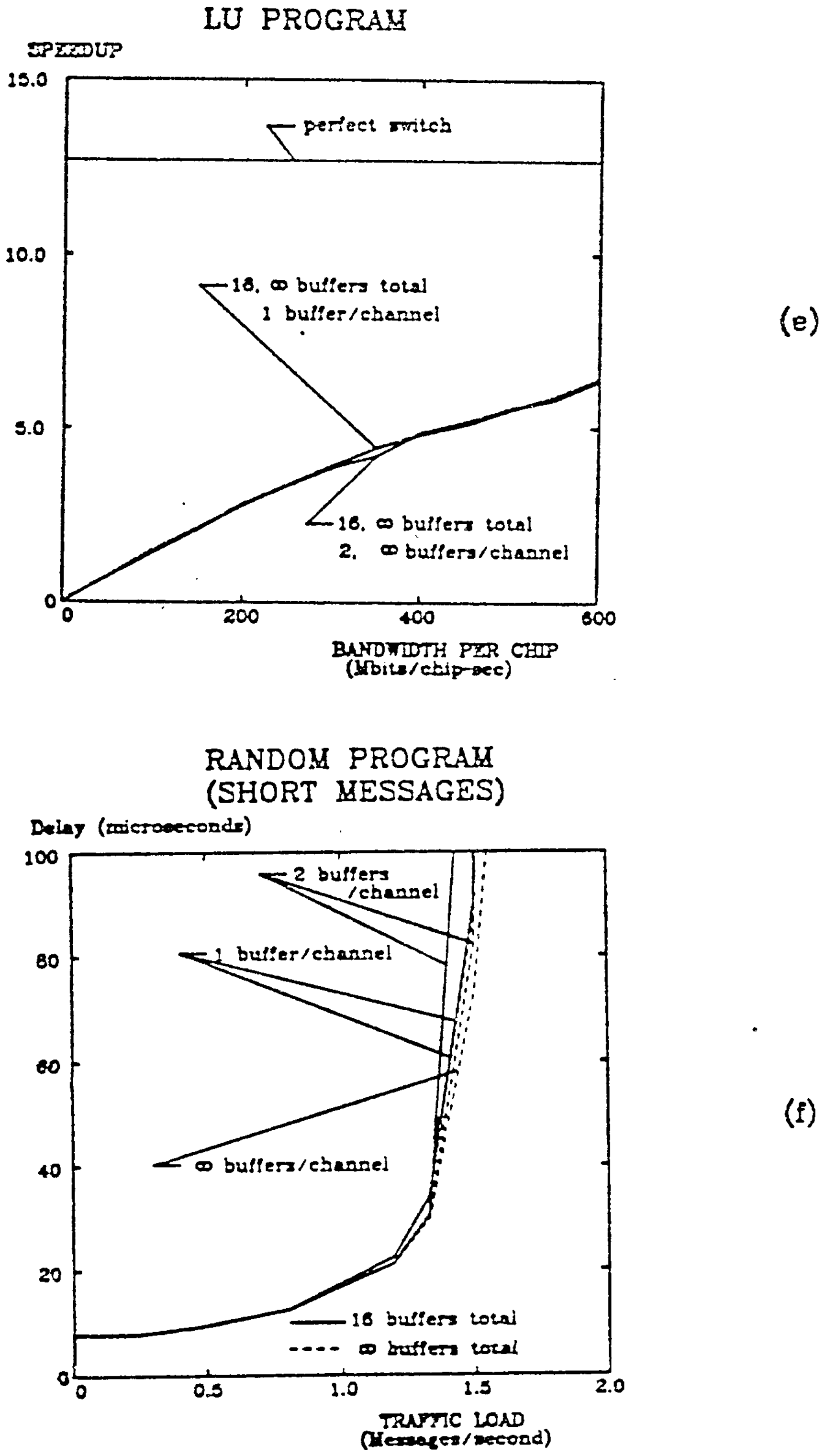
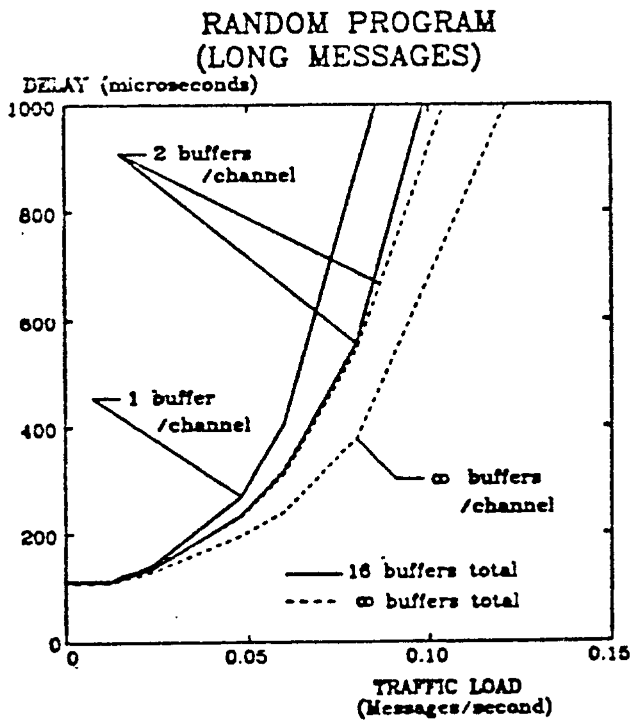


Figure 4.17. *Limited buffer space (e) LU. (f) Random (short messages).*



(g)

Figure 4.17. *Limited buffer space (g) Random (long messages).*

circuits to using at most one buffer at a time results in no significant degradation in performance. Networks using a send/acknowledge protocol for flow control yield virtually the same performance as networks with an infinite amount of buffer space and no restrictions on buffer usage. This indicates that the bandwidth wasted by negatively acknowledged packets does not have a significant effect on performance. This is due in part to the round-robin algorithm used for scheduling usage of the communication links. Blocked virtual circuits relinquish use of the link when a packet is rejected, allowing other traffic to use the link. Thus, the simulation results agree with the intuitive arguments presented earlier.

It might be noted that many of the programs achieve identical speedups regardless of the amount of buffering provided, or the buffer restrictions enforced. In these programs, a single, or a few isolated bottleneck links limit performance, e.g. the SISO programs are limited by the links carrying the initial data samples. This is in agreement with the scenarios outlined in figures 4.16b-d, where identical utilizations are achieved on bottleneck links regardless of the buffering scheme used.

All of the application programs send small, single-packet messages. To examine buffering requirements when large messages are used, the artificial traffic load program was modified to send longer messages consisting of 256 bytes, or 16 packets each. Figure 4.17g shows the performance of this program under different buffering schemes. The curves indicate that message delays are the same under light traffic loads, demonstrating that one buffer per virtual circuit is adequate to maintain a steady stream of packets if there is no interfering traffic. The curves also indicate however, that networks achieve somewhat higher bandwidth if multiple buffers per virtual circuit are allowed. As buffering is increased, the number of negatively acknowledged packets on circuits leading up to congested links is reduced, and bandwidth improves. This additional performance must be weighed against the added complexity of allowing multiple buffers per virtual circuit. In light of the fact that network bandwidth can be improved by increasing the number of communication chips, it is doubtful that this additional improvement is justifiable. In addition, the additional complexity of allowing a circuit to use more than one buffer (a fifo queue on each channel is required) may lead to longer circuit delays, and slower clock rates, reducing performance.

4.6. Complexity of the Communication Component

Using the results presented above, the complexity of the VLSI communication components described here can be estimated. It is assumed that each component provides from 64 to 256 channels per link, 32 buffers, each large enough to accommodate 16 bytes of data, and 4 I/O ports. Transistor counts for different versions of communication components providing different levels of functionality are presented.

The communication component design described here consists of 6 modules:

(1) I/O ports,

(2) routing controller,

(3) translation tables,

(4) packet buffers,

(5) buffer management circuitry,

(6) flow control circuitry.

Each of these will be discussed in turn. Estimates of the amount of circuitry required for these modules are summarized in table 4.2.

Approximate gate counts are derived in part from the designs described in the TTL Databook [Inst76]. For example, the five 2 line to 1 line multiplexers shown in figure 4.10b are assumed to require 18 gates, based on an extension of the corresponding TTL part, the SN74157 [Inst76]. Similarly, registers are assumed to use 5 gates per bit. Data paths for the high speed buses and communication links are 8 bits wide. It is assumed that each finite state machine requires 200 gates. This is based on the average number of gates required for the finite state machines described in [Fuji80], which are of roughly the same complexity as those described here. Finally, the transistor counts assume four transistors are required for each gate. The estimates in table 4.2 are rounded to the nearest 1000 transistors.

Estimates for the I/O ports are based on the circuitry described in [Laur79]. The output port estimate includes circuitry for removing data from the high speed bus and driving the communication links. The input port estimate includes circuitry for driving the high speed bus, as well as three temporary registers to handle conflicts in accessing the shared memory modules, as

Table 4.2
Transistor Counts

module	logic (transistors)	memory (kbits)		
	all designs	64 channels	128 channels	256 channels
I/O Ports (4 ports)	8000	-	-	-
Routing controller				
simple	9000	0.6	1.1	2.1
complex	10000	1.4	1.9	2.9
Routing Processor	20000	32.0	32.0	32.0
Translation Table	-	2.2	5.0	11.0
Buffers (16 modules)	8000	4.0	4.0	4.0
Buffer Management				
≤1 buffer/vc	2000	1.3	2.5	5.0
≤4 buffers/vc	2000	2.7	5.2	10.2
Flow Control				
Send/Acknowledge				
≤1 buffer/vc	11000	0.6	0.9	1.5
≤4 buffers/vc	12000	0.9	1.4	2.5
Remote Buffer				
≤1 buffer/vc	13000	1.0	1.3	2.0
≤4 buffers/vc	14000	1.2	1.8	3.0
totals				
simplest	58000	40.7	45.5	55.6
most complex	62000	43.5	49.9	63.1

discussed earlier. Based on this, each I/O port requires roughly 2000 transistors, or 8000 transistors for 4 ports.

The routing controller estimates are based on the design described in [Fuji80], modified to include circuitry for a 256 entry hierarchical routing table supporting up to 8 levels of lookup tables (see figures 4.4a and 4.4b). The numbers in table 4.2 only include the hardware support provided by the routing controller for setting the translation tables, and do not include the processor or microcode memory portions of the routing controller. These are listed separately in the table under "routing processor".

The translation table requires $p \times c$ entries. Each entry specifies an output port or the routing controller (3 bits) and a channel number (6, 7, or 8 bits for 64, 128, or 256 channels respectively), so 9, 10, or 11 bits are required. The buffer memory estimate includes circuitry to interface each memory module to the two buses, registers to hold the pipelined buffer addresses, and a 32 byte RAM to hold data. The figure in table 4.2 includes circuitry for sixteen memory modules.

Two estimates of the buffer management circuitry are shown in table 4.2. The first assumes each virtual circuit can only use at most one buffer at a time, and is based on the design shown in figure 4.8b. The second assumes four buffers can be used, and is based on the design in figure 4.8a.

Four estimates of the flow control logic are shown. The first two use the send/acknowledge protocol, and the latter two use a remote buffer management scheme. In addition, each of these designs allows virtual circuits to use either one, or up to four buffers at a time. The estimate for the send/acknowledge protocol with up to four buffers per channel is based on the design shown in figure 4.10a. As discussed in section 4.2.4.3, the remote buffer management scheme uses the same circuitry, as well as the additional logic shown in figure 4.10b. Modifications corresponding to the one buffer per channel restriction are outlined in sections 4.2.4.2 and 4.2.4.3.

The figures in table 4.2 indicate that a minimal complexity communication component with 4 I/O ports, 32 16-byte buffers, 64 channels per link, one buffer per virtual circuit, and a send/acknowledge flow control scheme, can be constructed with approximately 58,000 transistors for logic, and 40.7 kbits of RAM. Assuming single transistor ROM cells for microcode memory and single transistor dynamic RAM cells, approximately 100,000 transistors are required. Except for the number of channels, this design is in accordance with the design recommendations derived throughout this chapter. A similar design using 256 channels per link and the more complex routing scheme requires 59,000 transistors of logic, and 56.4 kbits of RAM. Assuming a static RAM implementation using 5 transistors per RAM cell, this more complex design requires on the order of

350,000 transistors, most of which is taken up by memory. Chips are currently available using 450,000 transistors, so the communication components described here can be implemented with current integrated circuit technology.

Figure 4.18 shows a possible floorplan for the 64 channel communication component described above. Sizes of various sections of the chip are based on the approximate transistor counts listed in table 4.2, based on single transistor RAM and ROM cells. Data paths correspond to those shown in figures 4.6 and 4.7.

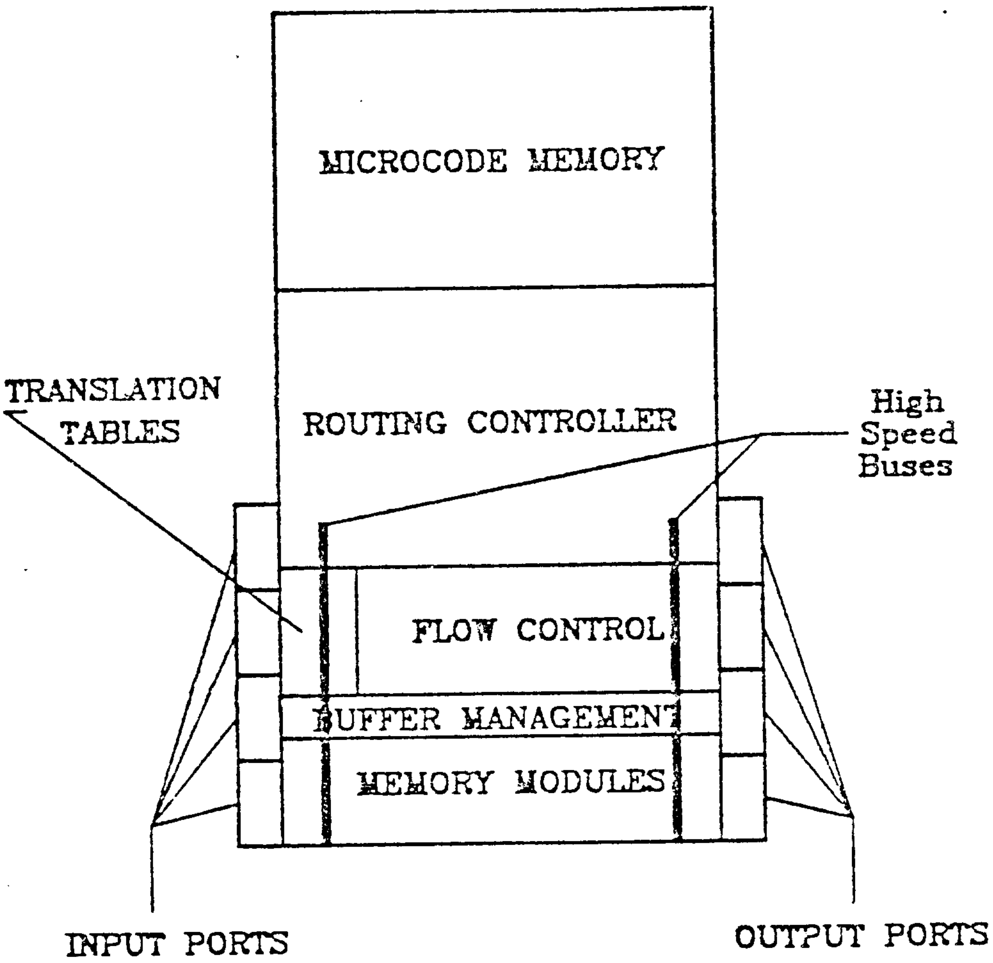


Figure 4.18. *Floorplan for communication component.*

CHAPTER FIVE

CONCLUSIONS

VLSI technology can provide us with a novel set of building blocks for the construction of high-performance point-to-point networks for closely coupled multicomputer systems. "Plug-compatible" VLSI communication components with 3 to 5 ports make particularly attractive building blocks. Their modularity permits the incremental growth of a multicomputer system with a corresponding growth of the total bandwidth of the communication domain. To be useful for the construction of systems with hundreds or thousands of processors, the complexity of these components must be above a certain threshold. The functionality of MOS VLSI chips now exceeds this threshold.

Technological considerations in the design of communication components have been examined. The described approach based on dedicated links between individual switching nodes is well matched to the evolving VLSI MOS technology. The overall performance of the network depends critically on the total chip bandwidth of these components, which is determined to a large degree by packaging technology. Performance is also influenced by the buffering and forwarding policies employed, which depend themselves on the amount of buffer space and the complexity of the control logic in the switching components. Analytic and simulation models have been used to investigate the impact of these considerations on overall network performance. Based on these studies, a number of conclusions can be drawn regarding the design of VLSI communication components. These include:

(1) A small number of ports should be used, say from 3 to 5.

(2) Under current technology, the degree of multiplexing on each communication link should be relatively large. Each link should provide a large number of channels, say 128 or 256.

(3) Only a relatively small number of buffers, say 16 or 32, need to be provided. Further, restricting virtual circuits to using at most one buffer per node at one time causes little performance degradation.

(4) Negatively acknowledged packets in the send/acknowledge flow control mechanism do not lead to a significant performance degradation, implying that more sophisticated schemes, such as the sender-controlled remote buffer management scheme, are not justified.

(5) A multicast mechanism has a significant impact on performance in applications which send the same data to many different destination processors.

An important issue that has *not* been addressed by this thesis concerns fault tolerance. If a communication component fails, routing tables need to be updated, and broken message paths must to be restored. The rerouting must be done in a manner that ensures that loops are not introduced. Much of the work in rerouting strategies in computer networks is directly applicable here [Taji77, Merl79, Sega81]. One must also safeguard the network against messages addressed to non-existent or unreachable nodes. These issues cannot be ignored in any communication component design.

For the near future, the limited number of devices that can be fabricated economically on a single chip will encourage the development of separate switching components. However, towards the end of this decade, the preferred building block may well consist of a powerful processor, a substantial amount of on-chip memory, and the switching circuitry that is needed so that these components can be readily plugged together into a working multicomputer system.

REFERENCES

[Acke65] S. B. Ackers, "On the Construction of (d,k) Graphs," *IEEE Transactions on Electronic Computers* EC-14 p. 488 (June 1965).

[Adam82] G. Adams III and H. Siegel, "The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems," *IEEE Transactions on Computers* C-31(5) pp. 443-454 (May 1982).

[Ahuj82] V. Ahuja, *Design and Analysis of Computer Communication Networks*, McGraw-Hill, New York (1982).

[Amar83] D. Amar, "On the Connectivity of some Telecommunication Networks," *IEEE Transactions on Computers* C-32(5) pp. 512-519 (May 1983).

[Arde81] B. W. Arden and H. Lee, "Analysis of Chordal Ring Network," *IEEE Transactions on Computers* C-30(4) pp. 291-295 (April 1981).

[Arde82] B. W. Arden and H. Lee, "A Regular Network for Multicomputer Systems," *IEEE Transactions on Computers* C-31(1) pp. 60-69 (Jan. 1982).

[Baas78] S. Baase, *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley, Reading, Massachusetts (1978).

[Barn80a] C. Barnes and S. Shinnaka, "Block Shift Invariance and Block Implementation of Discrete-Time Filters," *IEEE Transactions on Circuits and Systems* CAS-27 pp. 667-672 (August 1980).

[Barn80b] C. Barnes and S. Shinnaka, "Finite Word Effects in Block-State Realizations of Fixed-Point Digital Filters," *IEEE Transactions on Circuits and Systems* CAS-27 pp. 345-349 (May 1980).

[Barn78] T. P. Barnwell III, S. Gaglio, and R. M. Price, "A Multi-Microprocessor Architecture for Digital Signal Processing," *Proc. of the 1978 Intl. Conf. on Parallel Processing*, pp. 115-121 (August 1978).

[Barn79] T. P. Barnwell III, C. J. M. Hodges, and S. Gaglio, "Efficient Implementations of One And Two Dimensional Digital Signal Processing Algorithms on a Multiprocessor Architecture," *1979 Intl. Conf. on ASSP, Washington, D. C.*, pp. 698-701 (Apr. 1979).

[Barn82] T. P. Barnwell III and C. J. M. Hodges, "Optimal Implementation of Signal Flow Graphs on Synchronous Multiprocessors," *Proc. of the 1982 Intl. Conf. on Parallel Processing*, pp. 90-95 (August 1982).

[Batc76] K. E. Batcher, "The FLIP Network in STARAN," *Proceedings of the 1976 International Conference on Parallel Processing*, pp. 65-71 (August 1976).

[Bhar83] K. Bharath-Kumar and J. M. Jaffe, "Routing to Multiple Destinations in Computer Networks," *IEEE Transactions on Communications* COM-31(3) pp. 343-351 (March 1983).

[Brow80] S. Browning, *Communications in a Tree Machine*, Bell Laboratories internal memorandum (Jan. 1980).

[Brui46] D. G. de Bruijn, "A Combinatorial Problem," *Koninklijke Nederlandsche Academie van Wetenschappen te Amsterdam, Proc. Section of Sciences* 49(7) pp. 758-764 (1946).

[Burr71] C. Burrus, "Block Implementation of Digital Filters," *IEEE Transactions on Circuit Theory* CT-18 pp. 697-701 (Nov. 1971).

[Burr72] C. Burrus, "Block Realization of Digital Filters," *IEEE Transactions on Audio and Electroacoustics* AU-20 pp. 230-235 (Oct. 1972).

[Cant74] D. G. Cantor and M. Gerla, "Optimal Routing in a Packet-Switched Computer Network," *IEEE Transactions on Computers* C-23(10) pp. 1062-1069 (Oct. 1974).

[Carr72] W. Carr and J. Mize, *MOS/LSI Design and Applications*, Texas Instruments Inc. (1972).

[Chen81] P. Chen, D. Lawrie, D. Padua, and P. Yew, "Interconnection Networks Using Shuffles," *Computer* 14(12) pp. 55-64 (Dec. 1981).

[Chou81] W. Chou, A. W. Bragg, and A. A. Nilsson, "The Need for Adaptive Routing in the Chaotic and Unbalanced Traffic Environment," *IEEE Transactions on Communications* COM-29(4) pp. 481-490 (April 1981).

[Chua75] L. Chua and P. Lin, *Computer-Aided Analysis of Electronic Circuits: Algorithms and Computational Techniques*, Prentice-Hall, Englewood Cliffs, New Jersey (1975).

[Chu77] W. W. Chu and M. Y. Shen, *A Hierarchical Routing and Flow Control Policy (HRFC) for Packet Switched Networks*, North Holland Publishing Co., Amsterdam (1977).

[Clos53] C. Clos, "A Study of Nonblocking Switching Networks," *Bell System Technical Journal* 32 pp. 406-424 (1953).

[Dahl74] G. Dahlquist, A. Bjorck, and N. Anderson, *Numerical Methods*, Prentice-Hall, Englewood Cliffs, New Jersey (1974).

[Dala78] Y. K. Dalal and R. M. Metcalfe, "Reverse Path Forwarding of Broadcast Packets," *Communications of the ACM* 21(12) pp. 1040-1048 (December 1978).

[Davi73] D. W. Davies and D. L. A. Barber, *Communication Networks for Computers*, John Wiley & Sons, National Physical Laboratory, Teddington, England (1973).

[Deke83] E. Dekel and S. Sahni, "Binary Trees and Parallel Scheduling Algorithms," *IEEE Transactions on Computers* C-32(3) pp. 307-315 (March 1983).

[Desp78] A. Despain and D. Patterson, "X-Tree: A Tree Structured Multiprocessor Computer Architecture," *Proceedings of the 5th Annual Symposium on Computer Architecture, Palo Alto, Ca.* 6(7) pp. 144-151 (April 1978).

[Dias81a] D. Dias and J. Jump, "Packet switching Interconnection Networks for Modular Systems," *Computer* 14(12) pp. 43-53 (Dec. 1981).

[Dias81b] D. M. Dias and J. R. Jump, "Analysis and Simulation of Buffered Delta Networks," *IEEE Transactions on Computers* C-30(4) pp. 273-282 (April 1981).

[Ensl74a] P.H. Enslow, "Appendices I & J: IBM System 360 & 370," pp. 238-256 in *Multiprocessors and Parallel Processing*, John Wiley & Sons (1974).

[Ensl74b] P.H. Enslow, "Appendices N, O, & P: UNIVAC 1108, 1110, and AN/UYK-7," pp. 290-327 in *Multiprocessors and Parallel Processing*, John Wiley & Sons (1974).

[Farh81] G. Farhi, "Diametres dans les graphes et numerotations gracieuses," Ph. D. Thesis, l'Universite de Paris Sud (April 1981).

[Feng81] T. Feng, "A Survey of Interconnection Networks," *Computer* 14(12) pp. 12-27 (Dec. 1981).

[Fink80] R. A. Finkel and M. H. Solomon, "Processor Interconnection Strategies," *IEEE Transactions on Computers* C-29(5) pp. 360-371 (May 1980).

[Fink81] R. A. Finkel and M. H. Solomon, "The Lens Interconnection Strategy," *IEEE Transactions on Computers* C-30(12) pp. 960-965 (Dec. 1981).

[Floy62] R. W. Floyd, "Algorithm 97: Shortest Paths," *Communications of the ACM* 5(6) p. 345 (June 1962).

[Fran71] H. Frank and W. Chou, "Routing in Computer Networks," *Networks* 1(2) pp. 99-112 (1971).

[Fran81] M. A. Franklin, "VLSI Performance Comparison of Banyan and Crossbar Communication Networks," *IEEE Transactions on Computers* C-30(4) pp. 283-291 (April 1981).

[Fran82] M. A. Franklin, D. F. Wann, and W. J. Thomas, "Pin Limitations and Partitioning of VLSI Interconnection Networks," *IEEE Transactions on Computers* C-31(11) pp. 1109-1116 (Nov. 1982).

[Frie66] H. D. Friedman, "A Design for (d,k) Graphs," *IEEE Transactions on Electronic Computers* EC-15 pp. 253-254 (April 1966).

[Fuji80] R. M. Fujimoto, "Routing Controller for X-Tree," Master's Report, UC Berkeley (Dec. 1980).

[Fuji83] R. M. Fujimoto, "Simon: A Simulator of Multicomputer Networks," ERL Report, in preparation (1983).

[Gall77] R. G. Gallager, "A Minimum Delay Routing Algorithm using Distributed Computation," *IEEE Transactions on Communications* COM-25(1) pp. 73-85 (Jan. 1977).

[Gerl81] M. Gerla, "Routing and Flow Control," pp. 122-174 in *Protocols and Techniques for Data Communication Networks*, ed. F. F. Kuo,Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1981).

[Ghee82] T. Gheewala, "The Josephson Technology," *Proc. IEEE* 70(1) pp. 26-34 (Jan. 1982).

[Glas78] A. Glaser and G. Subak-Sharpe, "Failure, Reliability and Yield of Integrated Circuits," pp. 746-799 in *Integrated Circuit Engineering*, Addison-Wesley, Reading, MA (1978).

[Goke73] L. R. Goke and G. J. Lipovski, "Banyan Networks for Partitioning Multiprocessor Systems," *Proceedings of the 1st Annual Symposium on Computer Architecture, Gainesville, Florida* 2(4) pp. 21-28 (Dec. 1973).

[Good81] J. R. Goodman and C. H. Séquin, "Hypertree: A Multiprocessor Interconnection Topology," *IEEE Transactions on Computers* C-30(12) pp. 923-933 (Dec. 1981).

[Gott82] A. Gottlieb and J. T. Schwartz, "Networks and Algorithms for Very-Large-Scale Parallel Computation," *Computer, Special Issue on Highly Parallel Computing* 15(1) pp. 27-36 (Jan. 1982).

[Grif79] M. Griffin, "X-Tree Communication Buses," Master's Report, UC Berkeley (August 1979).

[Hear70] F. Heart, R. Kahn, S. Ornstein, W. Crother, and D. Walden, "The Interface Message Processor for the ARPA Computer Network," *Proceedings AFIPS Spring Joint Computer Conference* 36 pp. 551-567 (May 1970).

[Hodg80] C. J. M. Hodges, T. P. Barnwell III, and D. McWhorter, "The Implementation of an all Digital Speech Synthesizer Using a Multimicroprocessor Architecture," *1980 Intl. Conf. on ASSP, Denver, Colorado*, pp. 855-858 (April 1980).

[Hopp79] A. Hopper and D. Wheeler, "Binary Routing Networks," *IEEE Transactions on Computers* C-23(10) pp. 699-703 (Oct. 1979).

[Horo81] E. Horowitz and A. Zorat, "The Binary Tree as an Interconnection Network: Applications to Multiprocessor Systems and VLSI," *IEEE Transactions on Computers* C-30(4) pp. 247-253 (April 1981).

[Hosh83] T. Hoshino, T. Kawai, T. Shirakawa, J. Higashino, A. Yamaoka, H. Ito, T. Sato, and K. Sawada, "PACS: A Parallel Microprocessor Array for Scientific Calculations," *ACM Transactions on Computer Systems* 1(3) pp. 195-221 (August 1983).

[Imas81] M. Imase and M. Itoh, "Design to Minimize Diameter in Building Block Networks," *IEEE Transactions on Computers* C-30(6) pp. 439-442 (June 1981).

[Inst76] Engineering Staff of Texas Instruments, *The TTL Data Book for Design Engineers - Second Edition*, Texas Instruments Inc., Dallas, Texas (1976).

[Irla78] M. I. Irland, "Buffer Management in a Packet Switch," *IEEE Transactions on Communications* COM-26(3) pp. 328-337 (March 1978).

[Jack57] J. Jackson, "Networks of Waiting Lines," *Operations Research* 5(4) pp. 518-521 (August 1957).

[Jans80] P. Jansen and J. Kessels, "The DIMOND: A Component for the Modular Construction of Switching Networks," *IEEE Transactions on Computers* C-29(10) pp. 884-889 (Oct. 1980).

[Joel79] A. E. Joel Jr., "Circuit Switching: Unique Architecture and Applications," *Computer* 12(6) pp. 10-22 (June 1979).

[Juen76] R. R. Jueneman and G. S. Kerr, "Explicit Path Routing in Communications Networks," *Proceedings from International Conference on Computer Communications*, pp. 340-342 (August 1976). Toronto

[Kahn72] R. E. Kahn and W. R. Crowther, "Flow Control in a Resource-Sharing Network," *IEEE Transactions on Communications* COM-20(3) pp. 539-546 (June 1972).

[Kamo76] F. Kamoun, "Design Considerations for Large Computer Communications Networks," Ph. D. Dissertation, Engineering Report 7642, UCLA, Los Angeles, California (April 1976).

[Kerm79] P. Kermani and L. Kleinrock, "Virtual Cut Through: A New Computer Communication Switching Technique," *Computer Networks* 3(4) pp. 267-286 (Sept. 1979).

[Keye79] R. Keyes, "The Evolution of Digital Electronics Towards VLSI," *IEEE Journal of Solid-State Circuits* SC-14(4) pp. 193-201 (April 1979).

[Klei75] L. Kleinrock, *Queueing Systems, Volume I: Theory*, John Wiley & Sons (1975).

[Klei76] L. Kleinrock, *Queueing Systems, Volume II: Computer Applications*, John Wiley & Sons (1976).

[Knut73] D. E. Knuth, *The Art of Computer Programming, Sorting and Searching (vol. 3)*, Addison Wesley, Reading, Massachusetts (1973).

[Korn67] I. Korn, "On (d,k) Graphs," *IEEE Transactions on Electronic Computers* EC-16 p. 90 (Feb. 1967).

[Kung80] H.T. Kung and C.E. Leiserson, "Algorithms for VLSI Processor Arrays," pp. 271-292 in *Introduction to VLSI Systems*, ed. C.A. Mead and L.A. Conway, Addison-Wesley, Reading, MA (1980).

[Kung82] H. T. Kung, "Why Systolic Architectures," *Computer, Special Issue on Highly Parallel Computing* 15(1) pp. 37-46 (Jan. 1982).

[Kuo81] F. F. Kuo, *Protocols and Techniques for Data Communication Networks*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1981).

[Kush82] T. Kushner, A. Y. Wu, and A. Rosenfeld, "Image Processing on ZMOB," *IEEE Transactions on Computers* C-31(10) pp. 943-951 (Oct. 1982).

[Laur79] M. Laurent, "Input-Output Ports for the X-Tree Nodes," Master's Report, UC Berkeley (Nov. 1979).

[Lawr75] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Transactions on Computers* C-24(12) pp. 1145-1155 (Dec. 1975).

[Lela82a] W. E. Leland and M. H. Solomon, "Dense Trivalent Graphs for Processor Interconnection," *IEEE Transactions on Computers* C-31(3) pp. 219-222 (March 1982).

[Lela82b] W. E. Leland, "Density and Reliability of Interconnection Topologies for Multicomputers," Ph. D. Thesis, Computer Sciences Technical Report #478, University of Wisconsin, Madison (July 1982).

[Lev81] G. F. Lev, N. Pippenger, and L. G. Valiant, "A Fast Parallel Algorithm for Routing in Permutation Networks," *IEEE Transactions on Computers* C-30(2) pp. 93-100 (Feb. 1981).

[Long80] S.I. Long, F.S. Lee, R. Zucca, B.M. Welch, and R.C. Eden, "MSI Highspeed Low-power GaAs Integrated Circuits using Schottky Diode FET Logic," *IEEE Trans. Microwave Theory Tech.* MTT-28(5) pp. 466-472 (May 1980).

[Lu83] H. Lu, "High Speed IIR Digital Filters," Ph. D. Dissertation, in preparation (1983).

[Mass79] G. M. Masson, G. C. Gingher, and S. Nakamura, "A Sampler of Circuit Switched Networks," *Computer* 12(6) pp. 32-48 (June 1979).

[McQu74] J. McQuillan, "Adaptive Routing Algorithms for Distributed Computer Networks," NTIS Report (AD-781 467), U. S. Department of Commerce (May 1974).

[McQu78] J. M McQuillan, "Enhanced Message Addressing Capabilities for Computer Networks," *Proceedings of the IEEE* 66(11) pp. 1517-1527 (Nov. 1978).

[McQu80] J. M. McQuillan, I. Richer, and E. C. Rosen, "The New Routing Algorithm for the Arpanet," *IEEE Transactions on Communications* COM-28(5) pp. 711-719 (May 1980).

[Mead80] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass. (1980). Gov't. ordering no. AD-781 467

[Memm82] G. Memmi and Y. Raillard, "Some New Results About the (d,k) Graph Problem," *IEEE Transactions on Computers* C-31(8) pp. 784-791 (August 1982).

[Merl79] P. M. Merlin and A. Segall, "A Failsafe Distributed Routing Protocol," *IEEE Transactions on Communications* COM-27(9) pp. 1280-1288 (Sept. 1979).

[Merl80a] P. M. Merlin and P. J. Schweitzer, "Deadlock Avoidance - Store and Forward Deadlock," *IEEE Transactions on Communications* COM-28(3) pp. 345-354 (March 1980).

[Merl80b] P. M. Merlin and P. J. Schweitzer, "Deadlock Avoidance in Store and Forward Networks II - Other Deadlock Types," *IEEE Transactions on Communications* COM-28(3) pp. 355-360 (March 1980).

[Mitr78] S. Mitra and R. Gnanasekaran, "Block Implementation of Recursive Digital Filters -New Structures and Properties," *IEEE Transactions on Circuits and Systems* CAS-25 pp. 200-207 (April 1978).

[Nage75] L. W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," Ph. D. Thesis (ERL-M520), University of California, Berkeley (1975).

[Nass79] D. Nassimi and S. Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," *IEEE Transactions on Computers* C-27(1) pp. 2-7 (Jan. 1979).

[Nass81] D. Nassimi and S. Sahni, "A Self-Routing Benes Network and Parallel Permutation Algorithms," *IEEE Transactions on Computers* C-30(5) pp. 332-340 (May 1981).

[Nass82] D. Nassimi and S. Sahni, "Parallel Algorithms to Set Up the Benes Permutation Network," *IEEE Transactions on Computers* C-31(2) pp. 148-154 (Feb. 1982).

[Nath83] D. Nath, S. N. Maheshwari, and P. C. P. Bhatt, "Efficient VLSI Networks for Parallel Processing Based on Orthogonal Trees," *IEEE Transactions on Computers* C-32(6) pp. 569-581 (June 1983).

[Park80] D. S. Parker, "Notes on Shuffle/Exchange Type Networks," *IEEE Transactions on Computers* C-29(3) pp. 213-222 (March 1980).

[Pate81] J. H. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors," *IEEE Transactions on Computers* C-30(10) pp. 771-780 (Oct. 1981).

[Patt80] D. A. Patterson and C. H. Séquin, "Design Considerations for Single-Chip Computers of the Future," *IEEE Transactions on Computers* C-29(2) pp. 108-116 (February 1980).

[Patt82] D. A. Patterson and C. H. Séquin, "A VLSI RISC," *Computer* 15(9) pp. 8-21 (Sept. 1982).

[Peas77] M. C. Pease, "The Indirect Binary n-Cube Microprocessor Array," *IEEE Transactions on Computers* C-26(5) pp. 548-573 (May 1977).

[Pouz76] L. Pouzin, "Flow Control in Data Networks - Methods and Tools," *Proceedings of the Third International Conference on Computer Communications, Toronto, Canada,* pp. 467-474 (August 1976).

[Pouz78] L. Pouzin and H. Zimmerman, "A Tutorial on Protocols," *Proceedings of the IEEE* 66(11) pp. 1346-1370 (Nov. 1978).

[Pouz81] L. Pouzin, "Methods, Tools, and Observations on Flow Control in Packet-Switched Data Networks," *IEEE Transactions on Communications* COM-29(4) pp. 413-426 (April 1981).

[Prad80] D. K. Pradhan and K. L. Kodandapani, "A Uniform Representation of Single- and Multistage Interconnection Networks Used in SIMD

Machines," *IEEE Transactions on Computers* C-29(9) pp. 777-791 (Sept. 1980).

[Prad82] D.K. Pradhan and S.M. Reddy, "A Fault-Tolerant Communication Architecture for Distributed Systems," *IEEE Trans. on Computers* C-31(9) pp. 863-870 (Sept. 1982).

[Prep81] F. P. Preparata and J. Vuillemin, "The Cube Connected Cycles: A Versatile Network for Parallel Computation," *Communications of the ACM* 24(5) pp. 300-309 (May 1981).

[Prep83] F. P. Preparata, "A Mesh-Connected Area-Time Optimal VLSI Multiplier of Large Integers," *IEEE Transactions on Computers* C-32(2) pp. 194-198 (Feb. 1983).

[Reed83] D. A. Reed and H. D. Schwetman, "Cost-Performance Bounds for Multicomputer Networks," *IEEE Transactions on Computers* C-32(1) pp. 83-95 (Jan. 1983).

[Rile82] D. D. Riley and R. J. Baron, "Design and Evaluation of a Synchronous Triangular Interconnection Scheme for Interprocessor Communications," *IEEE Transactions on Computers* C-31(2) pp. 110-118 (Feb. 1982).

[Rind77] J. Rinde, "Routing and Control in a Centrally Directed Network," *AFIPS Conference Proceedings, National Computer Conference* 46 pp. 603-608 (1977).

[Sega77] A. Segall, "The Modelling of Adaptive Routing in Data-Communication Networks," *IEEE Transactions on Communications* COM-25(1) pp. 85-95 (Jan. 1977).

[Sega81] A. Segall, "Advances in Verifiable Fail-Safe Routing Procedures," *IEEE Transactions on Communications* COM-29(4) pp. 491-497 (April 1981).

[Sequ78] C. H. Séquin, A. Despain, and D. Patterson, "Communications in X-Tree, A modular Multiprocessor System," *Conference Proceedings, ACM,* (Dec. 1978).

[Sequ82] C. H. Séquin and R. M. Fujimoto, "X-Tree and Y-Components," in *Proc. Advanced Course on VLSI Architecture*, Univ. of Bristol, England, ed. P. Treleaven, Prentice Hall, Englewood Cliffs, New Jersey (1982).

[Shoc80] J. Shoch and J. Hupp, "Measured Performance of an Ethernet Local Network," *Communications of the ACM* 23(10) pp. 711-721 (Dec. 1980).

[Sieg79a] H. Siegel, "Interconnection Networks for SIMD Machines," *Computer* 12(6) pp. 57-65 (June 1979).

[Sieg81] H. Siegel and R. McMillen, "The Multistage Cube: A Versatile Interconnection Network," *Computer* 14(12) pp. 65-76 (Dec. 1981).

[Sieg79b] H. J. Siegel, "A Model of SIMD Machines and a Comparison of Various Interconnection Networks," *IEEE Transactions on Computers* C-28(12) pp. 907-917 (Dec. 1979).

[Spro81] D. Sproule and F. Mellor, "Routing, Flow and Congestion Control in the Datapac Network," *IEEE Transactions on Communications* COM-29(4) pp. 386-391 (April 1981).

[Stei83] D. Steinberg, "Invariant Properties of the Shuffle-Exchange and a Simplified Cost-Effective Version of the Omega Network," *IEEE Transactions on Computers* C-32(5) pp. 444-450 (May 1983).

[Ston71] H. S. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Transactions on Computers* C-20(2) pp. 153-161 (Feb. 1971).

[Ston72] H. S. Stone, "Dynamic Memories with Enhanced Data Access," *IEEE Transactions on Computers* C-21(4) pp. 359-366 (April 1972).

[Stor70] R. M Storwick, "Improved Construction Techniques for (d,k) Graphs," *IEEE Transactions on Computers* C-19 pp. 1214-1216 (Dec. 1970).

[Stri83] L. Stringa, "EMMA: An Industrial Experience on Large Multiprocessing Architectures," *Proceedings of the 10th Annual Symposium on Computer Architecture, Stockholm, Sweden* 11(3) pp. 326-333 (June 1983).

[Swan77a] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "CM*—A Modular, Multi-microprocessor," *Proceedings of the National Computer Conference, Dallas, Texas*, pp. 637-643 (June 1977).

[Swan77b] R. J. Swan, A. Bechtolsheim, K. Lai, and J. K. Ousterhout, "The Implementation of the CM* Multi-microprocessor," *Proceedings of the National Computer Conference, Dallas, Texas*, pp. 645-655 (June 1977).

[Swar82] E. Swartzlander Jr. and B. Gilbert, "Supersystems: Technology and Architecture," *IEEE Transactions on Computers* C-31(5) pp. 399-409 (May 1982).

[Taji77] W. Tajibnapis, "A Correctness Proof of a Topology Maintenance Protocol for a Distributed Computer Network," *Communications of the ACM* 20(7) pp. 477-485 (July 1977).

[Tane81] A. S. Tanenbaum, *Computer Networks*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1981).

[Than81] S. Thanawstien and V. P. Nelson, "Interference Analysis of Shuffle/Exchange Networks," *IEEE Transactions on Computers* C-30(8) pp. 545-556 (August 1981).

[Thom80] C. D. Thompson, "A Complexity Theory for VLSI," Ph. D. Dissertation, Computer Science Dept., Carnegie-Mellon University, Pittsbugrh, Pa. (August 1980).

[Toue79] S. Toueg and K. Steiglitz, "The Design of Small-Diameter Networks by Local Search," *IEEE Transactions on Computers* C-28(7) pp. 537-542 (July 1979).

[Tyme81] L. Tymes, "Routing and Flow Control in TYMNET," *IEEE Transactions on Communications* COM-29(4) pp. 392-398 (April 1981).

[Wagn83] R. A. Wagner, "The Boolean Vector Machine (BVM)," *Proceedings of the 10th Annual Symposium on Computer Architecture, Stockholm, Sweden* 11(3) pp. 59-66 (June 1983).

[Widd80] L.C. Widdoes, "The S-1 Project: Developing High-Performance Digital Computers," *Proc. COMPCON,* pp. 282-291 (Feb. 1980).

[Wing80] O. Wing and J. W. Huang, "A Computational Model of Parallel Solution of Linear Equations," *IEEE Transactions on Computers* C-29(7) pp. 632-638 (July 1980).

[Witt81] L. D. Wittie, "Communications Structures for Large Networks of Microcomputers," *IEEE Transactions on Computers* C-30(4) pp. 264-273 (April 1981).

[Wong81] R. Wong, "Serial Communications in X-Tree," Master's Report, UC Berkeley (June 1981).

[Wu80a] C. Wu and T. Feng, "On a Class of Multistage Interconnection Networks," *IEEE Transactions on Computers* C-29(8) pp. 694-702 (August 1980).

[Wu80b] C. Wu and T. Feng, "The Reverse-Exchange Interconnection Network," *IEEE Transactions on Computers* C-29(9) pp. 801-811 (Sept. 1980).

[Wu81a] C. Wu and T. Feng, "The Universality of the Shuffle-Exchange Network," *IEEE Transactions on Computers* C-30(5) p. 324 (May 1981).

[Wulf72] W. A. Wulf and C. G. Bell, "C.MMP--A Multi Mini Processor," *Proceedings of the AFIPS Fall Joint Computer Conference, Montvale, N. J.* 41 pp. 765-777 (1972).

[Wu81b] S. B. Wu and M. T. Liu, "A Cluster Structure as an Interconnection Network for Large Multimicrocomputer Systems," *IEEE*

Transactions on Computers C-30(4) pp. 254-264 (April 1981).

[Yew81] P. Yew and D. H. Lawrie, "An Easily Controlled Network for Frequently Used Permutations," *IEEE Transactions on Computers* C-30(4) p. 296 (April 1981).

[Yu84] W. Yu, "LU Decomposition on a Multiprocessing System with Communication Delay," Ph. D. Dissertation, in preparation (1984).

[Zema81] J. Zeman and A. Lindgren, "Fast Digital Filters with Low Round-Off Noise," *IEEE Transactions on Circuit and Systems* CAS-28 pp. 716-723 (July 1981).

[Zimm80] H. Zimmermann, "OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications* COM-28(4) pp. 425-432 (April 1980).