Systolic Combining Switch Designs

Susan R. Dickey[1]
Courant Institute of Mathematical Sciences
New York University

May 17, 1994

[1]Supported by U.S. Department of Energy grant number DE-FG02-88ER25052.

For my father
John Wilson Dickey
1916–1991

Acknowledgments

I have been fortunate to carry out the research for this dissertation in the stimulating and supportive environment of the Ultracomputer Research Laboratory. I would like to thank my advisor, Allan Gottlieb, Ultra's director, for his role in creating that environment as well as for his careful reading of several drafts of my dissertation. As part of my work for Ultra, I have enjoyed fruitful research collaborations with Richard Kenner, my long-term partner in VLSI design, Ora Percus, my mentor in stochastic analysis and queueing theory, Yue-sheng Liu, who helped me formalize my thinking about switch types as well as simulating network behavior, Jan Edler, who was always willing to add another feature to his network simulator, and Ron Bianchini, who makes hardware work.

In the computer science department as a whole, I would like to thank Alan Siegel, for reading my dissertation and making helpful comments, Ernie Davis and Richard Wallace, for serving on my committee, Elaine Weyuker, whose literature review seminar was very helpful in building both my confidence and competence to do computer science research, and Anina Karmen-Meade, who often helped me navigate NYU's bureaucratic labyrinths.

Most of all, my gratitude goes to my husband, Tom Duff, for his faith and support at all times, and to my children, Rachael Evans, Keelan Evans and Timothy Duff. No matter how the research is going, my wonderful family always fills my life with joy.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Communication between hundreds or thousands of cooperating processors is the key problem in building a massively parallel processor. This thesis is concerned with the best way to design a fast VLSI switch to be used in the interconnection network of such a parallel processor. Such a switch should handle the "hot spot" problem as well as provide good performance for uniform traffic. The switch designs we consider alleviate the "hot spot" problem by adding extra logic to the switches to combine conventional loads and stores as well as fetch-and-ϕ operations destined for the same memory location, according to the methods described in [57].

The goal of this work has been to analyze and design a switching component that is inexpensive compared to the cost of a processing node, yet provides the functionality necessary for high-bandwidth, low-latency network performance. The theoretical peak performance of a highly-parallel shared-memory multiprocessor may be less than that of a message-passing multicomputer of equal component count, in which all nodes contain a processing element as well as switching hardware. However, the actual performance achieved per processor on a large class of applications should be much higher in the shared-memory multiprocessor because the dedicated hardware of the network switches provides greater bandwidth per processing element and handles communication in a more efficient way.

The first section of this introductory chapter outlines the contributions of this thesis. The second section discusses related research.

1.1 Contributions

The analyses and simulations reported in this thesis were carried out in support of the design and implementation of a switching component for the NYU Ultracomputer architecture. The results are generally concerned with the trade-off between overall performance and implementation cost. The different areas in which results have been obtained are described in the following subsections.

1.1.1 Performance analysis of different switch types

Chapter 2 analyzes the effect that the arrangement and arbitration of buffers and the degree of the crossbar may have on switch performance and cost.

Switches in interconnection networks for highly parallel shared memory computer systems may be implemented with different internal buffer structures. For a 2×2 synchronous switch, previous studies [78, 116] have often assumed a switch composed of two queues, one at each output, each of which has unbounded size and may accept two inputs every clock cycle. We call this type of switch *Type A*; a $k \times k$ Type A switch has k queues, one at each output, each of which may accept k inputs per cycle.

Hardware implementations may actually use simpler queue designs and will have bounded size. Two additional types of switch are analyzed, both using queues that may accept only one input at a time: for $k \times k$ switches, a Type B switch uses k^2 queues, one for each input/output pair; a Type C switch uses only k queues, one at each input. In both cases, a multiplexer blocks all but one queue if more than one queue desires the same output, making these models more difficult to analyze than the previous Type A

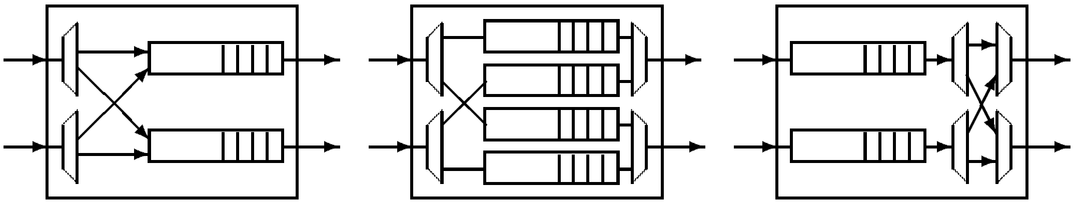


Figure 1.1: Three basic switch types

model. We have found maximum bandwidth, expected queue length, expected waiting time, and queue length distribution for the Type B and Type C 2×2 switches, with unbounded queue size and with queue size equal to 1. For 2×2 switches we have proved that the bandwidth per port of a Type C switch is limited to 75 percent. While the Type C switch is less expensive, Type A and B have considerably better performance.

1.1.2 An efficient CMOS implementation of systolic queues

Chapter 3 describes an efficient CMOS implementation for systolic queue designs; the basic queue design is useful for buffered non-combining as well as combining switches.

The timing constraints on blocking and unblocking for the systolic queue design originally developed by Snir and Solworth [130] are formalized. A proof is given that an implementation of this design using only two global control signals operates correctly under these timing constraints.

A non-combining switch using this systolic queue design was fabricated by MOSIS in 3 micron CMOS and used in a 2 processor prototype for over a year. This implementation employed the NORA (no race) clocking methodology, using qualified clocks as the mechanism for distributing global control. NORA allows the use of compact CMOS circuits with high tolerance for clock skew. Qualified clocks provide a natural way to implement local data movement in a systolic design, but their use with NORA involves certain complications. A circuit to produce a qualified clock for use in the NORA methodology was developed. The circuit's maintenance of NORA assumptions, as well charge-sharing and noise problems that can arise, are described.

1.1.3 Cost and performance of an implemented combining switch

Chapter 4 describes the combining switch that we have implemented for use in the 16×16 processor/memory interconnection network of the NYU Ultracomputer prototype. A 12-processor configuation using these switches is currently operational. Packaging, message types and message formats are described. Details are given about the internal logic of the two component types used in the network. The forward path component includes a systolic combining queue and an ALU for combining; the return path component includes non-combining systolic queues, an associative wait buffer, and an ALU for decombining. A design usable in networks of size up to 256×256 has also been prepared for fabrication at a smaller feature size in a higher pincount package; differences in the logic partitioning of the two designs are described.

Simulation results were used to compare the performance of the specific switch architecture and flow control method actually implemented with performance predicted by analytical models and by simpler simulation models of queue behavior. The effective queue size of our systolic queues, compared to standard linear FIFOs, was determined through simulation. Performance differences between Type A and Type B combining switches were explored.

Hardware combining is not without cost, but our experience in implementing a combining switch indicates that the cost is much less than is widely believed. We describe the design choices made in implementing the switch to keep network bandwidth high and latency low. We compare the cost of a combining switch to that of a non-combining switch and discuss the scalability of the design we have implemented to large numbers of processors.

1.1.4 Methods for providing greater combining capability

Chapter 5 compares the performance of this design with somewhat costlier switches that provide greater combining capability.

Small differences in the capabilities of combining switch architectures can make a significant difference in their performance. We have studied several such architectures using both analytical techniques and simulation.

Recurrence relations were constructed for the queue length probability distribution of several different 2×2 combining switch architectures. These recurrence relations were used to compute average queue length and output rates; results were validated with simulations.

We describe the implementation of one of "two-and-a-half-way" combining, which promises to avoid network saturation at later stages at only slightly greater cost than two-way combining. Two-and-a-half-way combining switches were simulated and their performance was compared to that of Type A and Type B switches with two- and three-way combining. Implementation alternatives for some different types and arrangements of buffers in a 4×4 combining switch were also simulated to compare their relative performance.

1.2 Related research

This section includes definitions of terms used to describe and analyze interconnection networks for parallel systems, descriptions of the features of the NYU Ultracomputer architecture, and a survey of the literature on hot spots in interconnection networks, including different schemes that have been proposed to alleviate the hot spot problem. We are interested in possible architectural differences in the broad class of MIMD (multiple instruction, multiple data stream) architectures, according to Flynn's taxonomy[46]. Although some of the results about network structure originally arose in the context of SIMD architectures or telephone switching, that literature will be reviewed only in the context of implementing highly parallel MIMD architectures.

Network architectures may differ in topology, routing protocol, or switching strategy. Such architectures may be evaluated according to many different metrics, such as ease of programming, direct mappings between the architecture and important data structures, fault tolerance, scalability or cost, which may be measured in wires, pins, physical space or amount of logic. Ease of programming and match between algorithm and architecture are complicated and somewhat qualitative measures of network performance. Such measures are beyond the scope of this discussion, though it should be noted that systems that depend on exploiting data locality and nearest neighbor connections to get good performance are generally more difficult to program.

The NYU Ultracomputer project began by investigating parallel algorithms on message-passing, staticly connected shuffle-exchange machines [56, 122] and evolved in the direction of implementing an approximation to the parallel random access (PRAM) model of computation first described in [47], because of the greater ease and generality of this model for the programmer and implementation of software systems. This evolution included the construction of bus-based prototypes as well as the design of an interconnection network [16, 54].

The NYU Ultracomputer network has the topology of an Omega network [85] with a buffered VLSI switch at each node (see Figure 1.2). As discussed in the following sections, such a network has

- Bandwidth linear in N, the number of PEs.
- Latency, i.e. memory access time, logarithmic in N, ignoring the effects of contention.
- $O(N \log N)$ identical components.
- A fixed number of input and output ports for each node.
- Routing decisions local to each switch.

The addition of combining to each node allows concurrent access by multiple PE's to the same memory cell with no performance penalty.

We evaluate network performance primarily in terms of the quantitative measures of *bandwidth* and *latency*, considered in comparison to the cost of the network, and the way cost and performance scale with the size of the system.

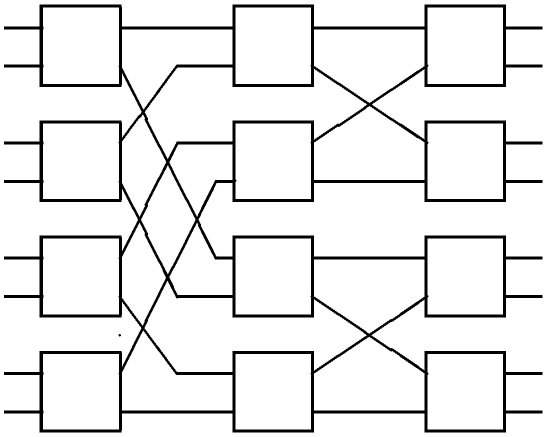


Figure 1.2: An 8×8 Ω-network.

In [14], *bandwidth* or *throughput* in a shared memory multiprocessor is defined as the mean number of memory requests delivered by the network to the memory per cycle. We use the term *bandwidth per processor* to refer to the steady-state message generation rate that can be accepted from each processor by the interconnection network and memory system. Performance is said to be *bandwidth-limited* if the processors themselves would be capable of generating a higher rate. In this case, messages are blocked from entering the network or dropped because of conflicts for resources in the interconnection network or at memory.

By *latency* we mean the total time from the cycle a processor issues a request to the network until the network delivers the response from memory. As a network becomes loaded close to its maximum bandwidth, latencies increase. Performance may be *latency-limited* in two ways:

1. The processor may be unable to generate new messages, due to an instruction or data interdependency, until the response to a previously sent message has been received. How frequently this occurs will depend on the processor, on the application and on the sophistication of the compiler and cache technology and of the operating system.

2. The hardware in the processor-network interface may allow only a fixed maximum number of messages outstanding in the network at a time.

In either case, the result will be that the message generation rate may drop as the latency increases, not because messages are blocked from entering the network, but because the PE must wait for a response to some previous message before a new message can be generated.

1.2.1 Interconnection network topology

Topology refers to the pattern of interconnections between the processors and other system elements. The distinction between a *static* (or *direct*) topology and a *dynamic* (or *indirect*) topology is a fundamental one, often associated in practice with differences in computational model. A direct network may be used to provide low-latency high-bandwidth direct connections to neighboring nodes, for computations that can be structured to have such spatial locality, but when destinations are randomly distributed, bandwidth is limited compared to a multistage indirect network.

The cost of a network depends not only on the total number of wires and nodes in a network, but on this pattern of interconnections or topology. The number of direct connections or *links* to a node is the *degree* of the node. In practice, the pin-out of the chips or boards containing the processors and switches limit the number of links of a given width, so that the degree of the node is a major cost factor. The *bisection width*, the number of wires which must be cut to divide the network into two equal size pieces, is a measure of wire density: the denser the wires, the larger the area (or the more layers of wiring) required for layout. Dally

[26] used bisection width as the major cost factor, comparing networks of equal bisection width. Under this metric, networks with low degree nodes could have wider data paths, thus giving lower latency performance under low traffic loads than networks with higher degree nodes. The product of degree and bisection width has also been used to characterize a network's cost [19].

Using the definitions in [145], a static topology has each switching point connected to a full processor, including memory, while in a dynamic network processing elements (PEs) and memory modules (MMs) are connected only at inputs and outputs of the switching network. This perhaps unintuitive terminology evolved from the usage in [45], where *static* means that the links between processors are dedicated, passive buses and *dynamic* indicates that the links can be reconfigured by setting active switching elements. Since current processor-at-a-node "static" architectures may include sophisticated switching hardware at each node (see, e.g.,[22, 28]) and since architectures that do not have a processor at every node but do not have well-defined network inputs and outputs have been developed [6, 27], the terms *direct* and *indirect* network, as used in [124] and elsewhere may be less misleading.

In a direct network, each switch is directly connected to a single processor; a processor and the switch it is connected to form a single node. (In a message-passing system like the Cosmic Cube[125], the "switch" may exist only as software running on the processor.) The most frequently considered direct networks are the family of k-ary n-cubes [124], in which k^n nodes are each labeled with an n-bit radix k number and connected to nodes that differ in only one radix k digit. Each node has degree $2n$, twice the number of the dimension. A k-ary n-cube may also be described as an n-dimensional array with k elements in each direction and end-around connections. Two and three-dimensional *mesh* topologies are examples of k-ary n-cubes without the end-around connections.

For direct networks with N nodes, following the notation in [3], if the degree of each node is d and the data path width has the same number of bits as a single message, the overall message generation rate per processors at steady state, (when message generation is equal to message arrival) is

$$m_g = \frac{dm}{D} \tag{1.1}$$

where m is the average utilization of each incoming link in steady state and D is the average number of hops to deliver a message (thus $1/D$ is the probability that an incoming message terminates). D can never be less than $O(\log_d N)$ for any topology [9] if messages are sent from one node to all other nodes with equal probability. Since m must of course be less than 1, the steady stage message generation rate of direct networks, assuming uniform distribution of destinations, has an upper bound of $O(d/\log_d N)$; the corresponding upper bound on the usable bandwidth per node is $d/\log_d N$ times the capacity of a link.

Indirect networks contain switches that are not directly connected to a processor. Another way to describe this is to say that the network contains nodes without a processor at the node. Thus, for the same maximum node degree, these networks are richer in links per processor and can maintain a higher bandwidth per node. To find the maximum message generation rate per PE for such a network, consider that m_g is constrained by

$$NDm_g \leq L \tag{1.2}$$

where L is the total number of links in the system, since at steady state the total amount of traffic which can be generated each cycle on average can be no greater than the link capacity. The message generation rate must also be less than the degree of the connection of the processor node to the network. Clearly, depending on the specific topology and traffic pattern, not all of the L links may act to add usable bandwidth, but this constraint provides an upper bound.

The most commonly discussed and analyzed indirect networks are the multistage networks with a logarithmic number of stages, such as the Omega network [85], the rectangular SW-Banyan [51], square delta networks [110], the baseline network [146] and the indirect binary n-cube [113], all of which can be shown to be essentially equivalent [146] when the number of inputs and outputs to the network and the degree of the switches is the same. By analogy with the k-ary n-cube direct networks and the butterfly network [90], such networks are sometimes called k-ary n-fly networks [25], where k is the number of inputs and outputs for each switch and n is the number of stages in the network, and there are k^n inputs to and k^n outputs from the network. Networks in this class have an upper bound on the message generation rate at inputs to the network equal to the full capacity of the link.

Topology	No. of nodes	Bandwidth per PE	Average distance	No. of links	Bisection width
k-ary n-fly	k^n	1	$2(n+1)$	$2k^n(n+1)$	$2k^n$
2×2 Omega	N	1	$2(\log N + 1)$	$2N(\log N + 1)$	2N
k-ary n-cube	k^n	$\frac{4}{k-1}$	$\frac{k-1}{2}n$	$2nk^n$	$2k^{n-1}$
hypercube	N	4	$\frac{\log N}{2}$	$2N \log N$	N
2 D torus	N	$\frac{4}{\sqrt{N}-1}$	$\sqrt{N} - 1$	$4N$	$2\sqrt{N}$
3 D torus	N	$\frac{4}{\sqrt[3]{N}-1}$	$\frac{3(\sqrt[3]{N}-1)}{2}$	$6N$	$2N^{\frac{2}{3}}$
fat tree	k^n	1	$2(n - \frac{1}{k-1})^\dagger$	$2nk^n$	$2k^n$

Table 1.1: Performance factors for interconnection network topologies

To illlustrate the bandwidth difference between direct and indirect networks Table 1.1 shows maximum bandwidth for a few interconnection networks of interest, assuming a message can be transmitted across a link in one cycle, and that all destinations for a message are equally likely (thus exploiting locality may allow some improvement over this "maximum," at a possible cost in software effort.) The number of links is the number of unidirectional links, or twice the number of bidirectional links.

The 2×2 Omega network is a special case of k-ary n-fly. For these two networks a "dance hall" model is assumed, with processors on one side of the network and memories on the other. Even if processors and memory are co-located and allow local memory access, these networks are generally used in a shared memory context where the unit of network communication is a memory operation and its acknowledgment, rather than a processor-to-processor communication. The average distance is the number of links traversed in a round trip from processor to memory and back to the processor.

The hypercube (the binary n-cube, also called the boolean n-cube [131]), 2D and 3D torus are frequently encountered special cases of the k-ary n-cube. The formulas for the k-ary n-cube networks are based on the analysis in [26] and assume a one-way message from processor to processor, with a processor at every node. The average distance may be an overestimate for practical situations because it assumes random destinations and does not take locality into account, but may also be an underestimate if adaptive routing is used and messages do not always follow a shortest path to their destination.

Fat trees are described in [92]. Each of the k^n leaves is assumed to contain a processor. Processors communicate by traversal of the k-ary tree up to the lowest common ancestor and then descending to the destination; average distance is based on a one-way message, as for k-ary n-cubes. The bandwidth per processor of the fat tree is limited by the single link out a leaf node, rather than by the constraint of Equation 1.2. This constraint would give a bandwidth of $\frac{(k-1)n}{(k-1)n-1}$, which does approach 1 from above as the tree gets large. A fat tree topology, though having the bandwidth advantage of an indirect network (as well as its cost), may also take advantage of locality to reduce latency, like the direct networks.

Two interesting metrics for the cost of network are the ratio of the number of links to the total system bandwidth, which we will call the *wire cost ratio,* and the ratio of the bisection width to the total system bandwidth, which we will call the *bisection cost ratio.* The higher these ratios, the higher the system cost for comparable bandwidth. For all networks in Table 1.1, the bisection cost ratio is $O(1)$. This corresponds to the intuition that, for reasonably designed networks, the bandwidth available under uniform traffic is closely related to the bisection width. The wire cost ratio, however, varies considerably among these network topologies.

As long as the constraint from Equation 1.2 is the limiting factor, the wire cost ratio is the same as the average number of hops a message takes. For the k-ary n-fly, hypercube and fat tree networks, the wire cost ratio is $O(\log N)$. However, for low-dimensional k-ary n-cubes, the wire cost ratio is $O(N^{\frac{1}{n}})$, implying that these networks becomes progressively more costly compared to the indirect networks as the system size grows. Note, also, that when k-ary n-cube networks of equal bisection width are compared, as is done in [26], the lower-dimensional networks with wider data paths between nodes have better performance but also have a greater wire cost ratio.

†Plus an additional $O(\frac{1}{k^{n+1}})$ term

Topology	No. of nodes	Bandwidth per PE	Average distance	No. of wires	Bisection width
CM-5	256	8	7.3	16,384	2,048
J-machine	512	5.14	10.5	27,648	1,152
NYU Ultra	256	32	18	147,456	8,192
Tera	256	91.02	45	1,048,576	16,384

Table 1.2: Performance factors for sample parallel computers

Table 1.2 shows the same performance factors as Table 1.1 for the interconnection networks of particular parallel computer architectures, measuring cost in wires instead of links, and maximum bandwidth in bits per cycle rather than messages per cycle. Number of nodes was chosen to be comparable with the 256 processor nodes in the Tera architecture [6], which has an arrangement of nodes that is not easy to define for other sizes. Some of these architectures may be implemented by allowing bidirectional data transfer on a single wire in the same processor cycle [28]; such bidirectional wires are counted as two wires in the total number of wires. In the descriptions below the degree of the node is the number of bidirectional links, and the link width is the number of bits that may be transmitted in one direction in one cycle.

The Tera network has 4096 nodes arranged in a 3D torus with sparse links. Each node is connected in only two of the three possible X, Y or Z directions, so nodes have degree 4 rather than 6. The links each transmit 64 data bits per cycle, plus an unspecified number of bits for control; only the wires for the data bits are counted here. Only 256 nodes are populated with processors; the rest are populated with memories and I/O processors or are available only to add bandwidth. The average distance reported in Table 1.2 is that for the 3D torus, doubled to allow a round-trip to memory, and is pessimistic in the sense that there may be considerable opportunity for locality.

For the other three networks, the wires transmit both control and data. The NYU Ultracomputer actually has 39 bit data paths in the forward path network for a 256 PE machine [17], but only the 32 bits used for data packets are counted in the link width. Since 2×2 switches are used, nodes in the network have degree 4.

The J-Machine [108] is a 3D torus architecture with a processor at every node. For ease in computing the average number of hops, we wish the number of processors to be k^3 for some k, so a system with 8^3 processors was chosen. (The router hardware would allow an ensemble of up to 64K nodes.) Nodes have degree 6, with a link width of 9 bits.

Only the data network of the Connection Machine CM-5 [93] is shown. This processor-to-processor network has the topology of a 4-ary fat tree, implemented using nodes of degree 8, with a link width of 8 bits. In theory, the link bandwidth at each node in a fat tree increases at each level approaching the root, while the number of nodes at each level decreases proportionately; in practice, the number of nodes at each level in the tree remains the same, as does the link width, so that the "root" of the tree actually consists of the same number of nodes as all the leaves. The values reported in Table 1.2 were computed under the assumption that all parent connections are used at every level of the tree. To decrease wiring cost, not all the possible connections in the CM-5 network are actually used.

The variety of networks that have been chosen for parallel architectures illustrates that cost assessments in practice are not continuous, based on total quantity of wires or silicon area, but very much dependent on fitting into constraints. Design costs and the market for components used in the system may be more important in the short run than the quantity and utilization of hardware. However, in our research, we are interested in network architectures that provide scalable bandwidth as system size grows, and thus concentrate our attention on multistage buffered networks.

1.2.2 Routing protocol

In SIMD systems, methods for routing permutations of data among the processors may be important for performance. For MIMD systems, we are primarily interested in routing in a renewal context, i.e., each PE repeatedly and independently generates messages to be routed to other PEs with some probability. In such a context, distributed routing, with only local decisions, is more appropriate than global control of routing decisions.

The k-ary n-fly networks have the property that exactly one path connects a given network input to a given network output, and that routing can be done locally by a simple scheme in which the output at the ith stage in the network is selected by the ith of n k-ary digits in a routing address. For the round-trip messages used in a shared memory system, local control on both the forward and return path would seem to require that both the destination and return addresses must be transmitted with each message. However, for k-ary n-fly networks, only a single path descriptor field which contains an amalgam of the origin and destination addresses is required [65].

For 2×2 switches, this scheme works as follows: initially, the path descriptor field is set to the destination address. At each switch, the high-order address bit selects the port to which the message is to be routed. Each switch replaces this bit with the number of the input port on which the message arrived and rotates the address one bit so that the routing bit for the next switch will be the new high-order bit. When leaving stage j of an n-stage network, the low-order j bits will be the high-order j bits of the origin address and the high-order $n - j$ bits will be the low-order $n - j$ bits of the destination address. Thus, when the message reaches its destination, the path descriptor field may be reversed and used in the same way as the return address to the origin of the message.

Following the definition in [76], in a *delta* network the path descriptors associated with different paths leading to the same output node are identical, so that, if the inputs are processors and the outputs are memory modules, each processor uses the same routing tag for a given memory module. *Bidelta* networks have this property in the reverse direction as well; that is, each input also has a unique numeric identifier than can be used to route in the reverse direction from any output to any input. It was shown in [35] that any two n-stage bidelta networks composed of $k \times k$ switches are isomorphic. The bidelta property, besides providing a functional description of a unique network of a given size, is of practical value in a shared memory system since it provides a unique PE or MM number for each module that can be used for interrupts or other purposes from any location in the system.

However, networks without the delta or bidelta properties can still use digit-controlled routing and create the return address as the message is routed to its destination, as long as the switches are connected in such a way that every output can be reached from every input. In such networks, each input may use a different address for each output of the network. In a shared memory system, a functional mapping or table lookup to get the correct memory module address would be required at each processor as a part of memory address translation. Figure 1.3 shows such a "non-delta" network that was considered for use in a 16-PE NYU Ultracomputer prototype because it allowed short wires on a backplane connecting 4×4 switch boards that were all wired identically [16].

For networks with more than one path from source to destination, adaptive routing may improve performance [48], at some cost in complicating logic at the switching nodes. We will not consider such networks here.

1.2.3 Switching strategy

Maximum bandwidth and minimum latency are determined by the number of wires in the network, the way the wires are connected, and the number of hops a message must take, as discussed in section 1.2.1. The bandwidth and latency actually experienced are, however, strongly affected by the switching strategy adopted. *Circuit switching* provides a guaranteed low latency transmission for a message once a circuit has been established, at the cost of tying up resources (links) that could be used by other messages and thus limiting bandwidth. *Store-and-forward switching* (often called *message* or *packet switching*) increases overall network throughput by releasing a link as soon as a message passes through it, at the expense of increasing latency for individual long messages by only using one link at a time. *Cut-through* techniques, which include the recently popular "wormhole routing," can combine some of the advantages of both techniques.

Circuit switching, the standard technique used for telephone switching, requires a set-up period during which each link used for the transmission must be visited. After the circuit has been set up, all links can be active simultaneously on a pipelined transmission. Circuit switching is profitably used when transmission times are typically much larger than the set-up time and has seen relatively little use in computer networks or interconnection networks for multiprocessors, where the set-up time is often substantial compared to the

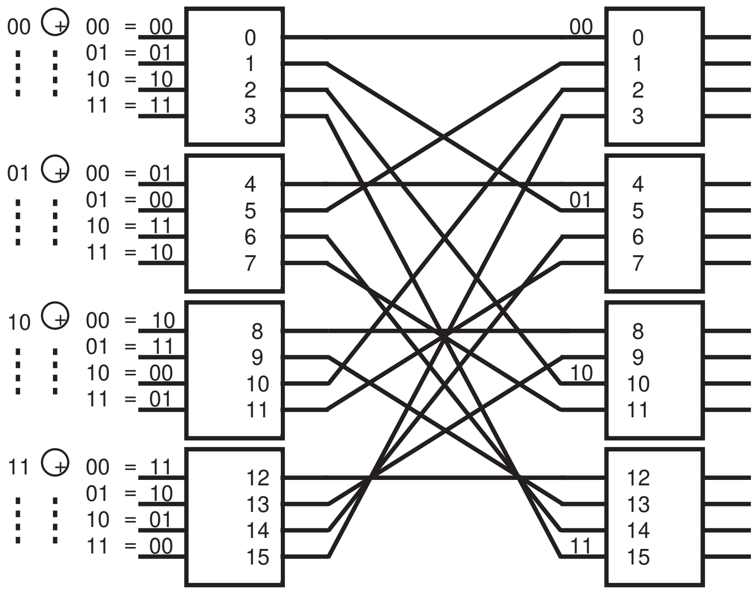


Figure 1.3: A non-delta network, from [16].

length of the message.

Store-and-forward switching has been the standard technique in long-haul computer communications. In Chapter 5 of [73], Kleinrock describes both message and packet switching as store-and-forward techniques in which the unit of transmission (message or packet) is transmitted completely on one link and buffered, if necessary, until the entire unit has been transmitted across the link, before any transmission begins on the next link. The distinction between the two methods is that in packet switching, a higher level protocol at the sending node has broken the message into smaller packets, each of which contains a header with all the addressing information needed to deliver the packet to its destination. Though the duplicated header information incurs some cost, the latency for message transmission can be greatly reduced because the packets of a message are pipelined on multiple links. Tanenbaum [134] states that pure message switching is never used in computer networks because of the overhead in storage and delay at each node.

In virtual cut-through switching (the name "virtual cut-through" is due to Kermani and Kleinrock [69] and is often abbreviated to "cut-through"), messages are broken into packets that are transmitted in a pipelined fashion. Unlike store-and-forward message switching, not all the packets in a message need be buffered in a node before the first packet may leave; only enough packets to give the routing information must arrive before the first packet may exit. The pipelining advantage of packet switching is obtained without duplicating the header in each packet. Only the first packet(s) of a message contain(s) routing information; logic in the switches ensures that the remaining packets in a message follow the path of the header. Cut-through switching avoids the assembly and disassembly of packets into messages by a higher-level protocol needed in packet switching, while still providing lower latency. Like circuit switching, all the links on a path from source to destination can be active on a transmission simultaneously, but, unlike circuit-switching, there is no set-up time and the settings in each switch are not held after the last packet of the message passes through. As noted in [107], the "wormhole routing" version of cut-through switching has been adopted for several parallel computer architectures, including the Ametek 2010, the Intel Touchstone Delta, and the Intel Paragon.

Our designs for switches in the NYU Ultracomputer employ cut-through, which is particularly easy to implement with systolic switches (see Chapter 3). The switching technique and flow control protocol we use was first described by Snir and Solworth in [130]. A message is composed of packets; the size of a packet is the same as the number of bits that can be transmitted on a link in a single cycle. The address packet contains the header information; a message may also contain one or more data packets. Most of the implementations discussed in detail in Chapters 4 and 5 have data paths wide enough to contain the network routing information in a single packet, though conceptually header information could be spread across more than one packet.

A variety of terms have been used to describe units of transmission in the context of cut-through switching. Our usage of the terms "message" and "packet," with packet as the sub-unit of message that can be transferred in a single network cycle, follows that of Kruskal and Snir in their analysis of what they called "pipelined message switching" in [77]. In the original paper [69] in which Kermani and Kleinrock define and analyze *virtual cut-through,* "message" and "packet" are used interchangeably to describe the unit of network transmission that contains routing information. In their paper, whenever the head of a message arrives at a node and the outgoing link is free, the message is said to "cut through" and begin transmission to the next node, without being buffered. If the outgoing link is busy at the time the head arrives, the message must be buffered. A *partial cut* is said to occur if a message that has been blocked and partially buffered is allowed to continue transmission as soon as the link is free, without waiting for the message to be completely buffered. No terminology is developed for sub-units of a message; instead the analysis is phrased in terms of the time it takes for the message to be transmitted across a link.

Dally and Seitz, in [30] and elsewhere, use the term *wormhole routing* to describe a flow control strategy used with cut-through switching that provides only minimal buffering. Instead of either providing enough buffering for a complete message or dropping messages when they are blocked (as, e. g., the BBN Butterfly [24]), only enough buffering is provided at each node to make routing decisions; blocked messages are held in place and tie up links leading back to the tail of the message. Dally and Seitz use "packet" for what we call "message", the set of bits being routed together to a destination, and have two terms for the sub-units of a message. *Flit* (flow control unit) refers to the smallest sub-unit of a message that a queue can accept or refuse, while *phit* (physical transfer unit) refers to the part that can be transmitted across a link in one cycle, corresponding to our use of the term "packet."

The switching designs we consider in later chapters have a flow control protocol that allows partial cuts in the sense that a blocked address packet may start up transmission before the final packet in the message has arrived at that node. Our protocol does not, however, allow the data packets of a message to be blocked once the address packet has been transmitted to the next node. So there is no sub-unit of our messages that exactly corresponds to a flit in the sense that it is used in describing wormhole routing, as a part of a message that can be both accepted and refused independently.

Abraham and Padmanabhan [3] analyze both "full cut-through," in which cuts are allowed only if the link is free when the head of the message arrives, and "partial cut-through," in which partial cuts are allowed as well.[1] They use "message" in the same sense that we do, but they use the term "nibble" to refer to what we call a "packet."

Analyzing the relative performance of cut-through switching and store-and-forward switching is a difficult problem, and a number of different approaches have been tried. Kermani and Kleinrock analyzed systems with both single and multiple channels per link and with noisy and noiseless channels, using assumptions of independence, Markovian distribution and balanced traffic to make the analysis tractable. In Kermani and Kleinrock's analysis, the message must be completely buffered (partial cuts are not allowed). This assumption both makes the analysis simpler and is sensible for the data communications application which they were considering, where it is advantageous in the presence of noisy channels to buffer a blocked message in order to perform error checking. For single channel links with noiseless channels, the difference between T_m, the average time to transmit a message under store-and-forwards message switching, and T_c, the average time to transmit a message using virtual cut-through, is

$$T_m - T_c = (\overline{n_h} - 1)(1-\rho)(1-\alpha)t_m \tag{1.3}$$

where $\overline{n_h}$ is the average number of hops the message must travel, ρ is the average utilization of a link, α is the proportion of the message taken up by header, and t_m is the average time required to transmit a message over a single link. When messages are long compared to the capacity of the link, t_m will be large and virtual cut-through can significantly improve performance compared to store-and-forward. Furthermore, virtual cut-through also provides a substantial savings in storage when traffic is light. On the other hand, if the utilization ρ is high or the average number of hops is small, cut-through will not provide a large performance gain, even for long messages.

[1] Since partial cut-through requires greater hardware capability and gives better performance than full cut-through, the terminology is somewhat unintuitive, since people normally expect "full" to be better than "part."

In [77], Kruskal and Snir give the following approximate formula for network delay using cut-through switching with m packet messages in a delta network with $k \times k$ switches:

$$T = \log_k N(t + mt\frac{m(1-1/k)p}{2(1-mp)}) + (m-1)t \tag{1.4}$$

where N is the number of network inputs and outputs, p is the number of messages per cycle, and t is the cycle time of a switch.[2] The first t in the expression being multiplied by the number of stages in the network represents the transition time for the head of a message from input to output, assumed to be a single cycle, and the second term represents queuing delay. The last term accounts for the "pipe-setting" delay. In [79], this formula was modified by a factor obtained from simulations to account for the changes in queueing delay at later stages. No analysis was done for the store-and-forward case.

In [3], a delay formula for cut-through was calculated by subtracting the benefit due to cut-through at each stage from the store-and-forward delay formula. For the full cut-through case, the benefit due to less waiting for all messages that arrive at the same time as a message that cuts through was given as

$$\frac{1-p}{1-a_0}pmt \tag{1.5}$$

and an additional benefit for all messages that arrive at a queue during the time the output link is busy after a message that cuts through arrives is given as

$$\frac{1-a_0-a_1}{1-a_0}p^2mt \tag{1.6}$$

where a_i is the probability that i messages arrive at a queue on a given cycle, and the rest of the notation is as in Equation 1.4. Their analysis showed a 35 percent improvement for virtual cut-through over store-and-forward, with about an additional 20 percent additional improvement when partial cuts are allowed, for a total improvement of 55 percent at moderate traffic levels, but the figures given are for long messages issued at low rates.

Dally's work [25, 31] analyzed the performance of wormhole routing compared to virtual cut-through with queueing under light traffic conditions for both indirect k-ary n-fly networks and direct k-ary n-cubes. His analysis for virtual cut-through with infinite queues in a k-ary n-fly network, gave the following approximate formula for network delay

$$T = \log_k N(t + mt\frac{m(1-1/k)p}{2}) + (m-1)t \tag{1.7}$$

using the same notation as in Equation 1.4. Omitting the factor $(1 - mp)$ from the denominator in the expression for queueing delay makes this approximation less accurate as traffic increases.

Wormhole routing has become very popular in systems where the offered load is low, because of its latency advantages over store-and-forward and its storage advantage over buffered cut-through techniques [107]. However, because messages retain channels when blocked, this technique is particularly susceptible to problems with deadlock in many network configurations. Sophisticated switching hardware may be required to solve this problem. In a recent study by Adve and Vernon [4], a closed queueing network model was developed for wormhole routing in k-ary n-cube networks using the non-adaptive deadlock-free routing scheme of Dally and Seitz [30]. Adve and Vernon found that, when processors are allowed to have multiple outstanding requests, system performance is bandwidth-limited rather than latency-limited and thus, since the bandwidth in k-ary n-cubes scales as $N^{\frac{1}{k}}$ (see 1.1), this configuration does not scale well with increasing system size under uniform access patterns. With four outstanding requests per processor, at least 70-80 percent of each processor's traffic must be directed to its nearest neighbors for system performance to scale well. They also showed that the deadlock avoidance algorithm places asymmetric loads on the virtual channels that create differences in efficiencies for processors at different points in the network.

[2] Kruskal and Snir distinguish between the transmission time and cycle time, but we assume a system in which the cycle time is the maximum of these two values.

1.2.4 Non-uniform traffic patterns

As discussed in section 1.2.1, multistage interconnection networks can be constructed to have bandwidth linear in the number of processors under uniformly distributed memory traffic. However, message traffic may depart from uniformity in a variety of ways. Some of these departures from uniformity may actually increase the available bandwidth, by preferentially accessing closer processors or by creating access patterns that in which different paths share fewer links than they would be expected to share under uniform traffic. Others degrade performance, due to contention at the memory or at links in the network.

A variety of traffic patterns can cause contention within the network or memory system even if no two PEs are accessing exactly the same memory addresses. For example, the typical spatial locality in most programs will cause a memory access bottleneck if consecutive virtual addresses are stored in the same memory module. Using low-order address bits to interleave memory across the modules exploits spatial locality to distribute the references, thus avoiding this potential bottleneck. Since shared interleaved memory makes it more difficult to keep processor local data in the memory module physically adjacent to the processor, caches at the processor are normally required in such an architecture to allow the exploitation of temporal locality.

Patterns that do not favor any particular memory module can also cause problems. Mitra and Cieslack [106] studied the behavior of Omega networks under traffic distributions that satisfy the full λ relations: each processor issues traffic at a rate λ and each memory module receives traffic at a rate λ. They showed that there are traffic patterns that satisfy the λ relations but induce a traffic intensity of $\lambda 2^{\min(i,n-i)}$ on some links in stage i of an n-stage Omega network. They proposed inserting a *scattering* network of r stages to randomize the paths followed through the network, and showed that such a network had a maximum traffic intensity, for traffic distributions that satisfy the full λ relations, of $\lambda 2^{\min(i,n-r-i)}$ on any link in stage i.

Kim and Garcia [70] developed an analytical model of buffered banyan networks composed of Type C switches (see section 2.1) of sizes 1 and 2 and applied the model to non-uniform traffic patterns, including a pattern with single-source-to-single-destination with a background of uniform traffic and a *maximum conflict* pattern similar to those studied by Mitra and Cieslack. They found little improvement due to increasing the buffer size, but a more substantial improvement if two parallel buffered networks are used.

Lang and Kurasaki [84] evaluated degradation of performance due to non-hot-spot, non-uniform traffic patterns that included the bit-reversal permutation pattern, other patterns in which each source favors a single destination but with source-destination pairs chosen at random, and the EFOS (even first, odd second) pattern, which sends messages from even numbered sources to the first half of memory, and from odd numbered sources to the second half. They found that, for these patterns, the use of *diverting switches* and augmenting the network with additional links were successful at reducing performance degradation, but that randomization and discarding messages were not successful. Diverting switches never block messages but instead send a message to the wrong output buffer if the intended output buffer is full. Messages may be routed to the wrong destination and must be retransmitted from there to the original destination. They augmented the of the network with additional links connecting switches in the same stage into rings, thereby providing additional paths around points of congestion within the network.

Lee, Cheung and Peir [89] considered the *consecutive requests* traffic pattern, which can be formed when each processor issues a sequence of requests to successive memory locations. To avoid the conflicts in the early stages of the network that can occur if consecutive memory locations are placed in consecutive memory modules, they suggest a bit reverse mapping. For example, in an 8×8 system composed of 2×2 switches, consecutive requests 0,1,2,3,4,5,6,7 would be mapped to MMs 0,4,2,6,1,3,5,7, and thus (assuming routing at the first stage is done on the high order bits of the MM address) successive requests would go to alternate outputs. Since even with the bit-reverse mapping, progressively longer sequences requesting the same output may occur as a result of merging of request streams, they also suggest a dynamic priority scheme, in which once a message has been sent on output port i from a input port j, messages for output port i arriving on input port j will have priority over messages for output port i from any other input port, for as long as input port j continues to have messages for output port i. This dynamic priority scheme has the effect, when both input ports have clusters of messages to be output on port 0, followed by clusters of messages to be output on port 1, of improving the utilization of the output ports, since one port begins sending a stream on output 0 only after the other has finished with output 0 and begun to send messages to output 1. Their simulations showed these techniques to reduce delay by about 20 to 30 percent on a 64 PE system, when all PEs are performing accesses to vectors of length 256; simulations in which all PEs began access simultaneously to the same memory module naturally showed the most improvement.

Contention patterns like those just described, which arise when *different* locations are being accessed by multiple processors, can thus be resolved by interleaving, hashing, or other methods of randomizing the addresses, or by diversion techniques or other methods of providing alternative paths around congestion in the network. In this thesis, we are concerned with the kind of non-uniformity in which a disproportionate number of memory requests are concentrated at a particular memory location, called a *hot spot* [119]. If these requests are serviced serially, a bottleneck arises, which cannot be avoided by the techniques suitable for other forms of non-uniform traffic. Such hot spots are particularly likely to arise as part of synchronization or other interprocessor communication, or when many processors working on the same problem are accessing shared data structures or loading the same code segments into cache.

1.2.5 The hot spot problem

As an example of the hot spot problem, consider a system with a $\log N$ stage interconnection network running a parallel application that uses a single shared queue to implement a workpile. The most straightforward parallel implementation uses a critical section to protect the insertions and deletions. This critical section produces a bottleneck, limiting the speed-up of the parallel application as well as creating hot spots at the variable guarding the critical section.

More sophisticated queue algorithms, described in [59], use a synchronization operation called fetch-and-add to eliminate the critical section. When these operations are combined in the network, as described in section 1.2.6, the bottleneck is eliminated. Without combining, these algorithms move the bottleneck from software to hardware, since the fetch-and-adds to the coordination variable are serialized at the memory module. If references to a particular hot spot are issued by each processor more often than once every N cycles, where N is the number of PEs, servicing these references will be the limiting factor on system performance.

The obvious problem with hot spots is the serial bottleneck at memory. The not-so-obvious problem is the effect this serial bottleneck has on other traffic. In [119], Pfister and Norton studied these effects with simulations using a simplified single hot spot model in which messages are independently generated at each node by a steady state renewal process. Messages are generated at an overall rate of p, with some fraction h of the messages from all processors destined for the hot spot. In simulations of networks with Type A switches, they showed that hot spot requests not only received slow service themselves because of serialization at memory, but also cut overall network bandwidth and increased delays for requests destined for other memory locations through the phenomenon that they called *tree saturation.* The paths followed by messages destined for a single hot spot form a tree in any unique path multistage network. Output queues on these paths become longer than those not on the tree because the traffic intensity to those outputs becomes progressively greater: at stage i, with stages numbered from 0, the input rate directed to the hot output will be

$$p_i(1-h) + 2^{i+1}p_i h. \tag{1.8}$$

If the queues are of unbounded size, the queue length will become infinite as this value approaches 1; if the queues are of bounded size, blocking will occur and the bandwidth entering the network will be limited.

Messages that enter queues at hot outputs experience increased waiting time, even if not destined for the hot spot. If the buffers holding the queues are limited in size, buffers filling at later stages will block outputs at earlier stages and can cause all queues on the tree from the hot MM to the PEs to become full, so that blocking occurs at all inputs to the network.

The time required from the time a hot spot begins to be active until tree saturation occurs was examined by Kumar and Pfister in [81]. Their results, obtained from an approximate analysis supplemented by simulation, show that a hot spot traffic pattern does not need to be sustained for long before tree saturation occurs. They also studied the recovery time: the time after message generation patterns return to a uniform distribution when throughput is still depressed, and delay still elevated because of the congestion in the network caused by the hot spot pattern. Their simulations indicate that the recovery time is much longer than the time required for onset of tree saturation.

Pfister and Norton claimed that performance degradation due to hot spot traffic would occur regardless of topology or queue size. Subsequent studies have supported this, on the whole, although some topologies and buffer structures show less degradation to non-hot spot traffic than others.

Hot spots in multistage interconnection networks

Networks containing unbuffered switches that discard contending messages do not exhibit full tree saturation, since messages to the hot spot do not accumulate in the network, and thus inputs with non-hot spot messages are never completely blocked. In this sense, such switches have been called *non-blocking* switches. Experiments on the BBN Butterfly [136] showed that the access time for a memory that contained a hot spot was degraded, but showed little effect on the performance of programs that avoided the hot memory. However, the processors of the BBN Butterfly were slow compared to the speed of the network, so that program bandwidth requirements were low. Liu has shown [99] that even in unbuffered networks the bandwidth to memories reached by paths that overlap the hot spot tree is lowered by contention with hot spot requests. Furthermore, if all processors are jointly cooperating on the same application, it may not be possible to arrange for them to avoid the hot memory. In addition, Liu's model is optimistic in that the probability of generating a hot spot request was assumed to be independent of previous requests, whereas in a real system a rejected hot spot request is likely to be resubmitted the next cycle. Simulations by Lesher and Thazhuthaveetil [95] confirm that unbuffered networks suffer a drastic bandwidth reduction due to hot spots.

Patel and Harrison have studied tree saturation in circuit-switched delta networks [112]. Using an iterative method in which sub-networks are replaced by flow equivalent servers, they were able to compute throughput as a function of the population of the system and the routing probability to the hot memory module. They showed that, for a 16-way, 4-stage delta network, the throughput is a maximum for uniform traffic, and the throughput curves tend toward $1/\rho$, where ρ is the total amount of traffic directed toward the hot memory module.

Lin and Kleinrock [96] present a method for analyzing finite-buffered multistage interconnection networks under a general traffic pattern, allowing the specification of a different distribution of request destination to each processor, including the specification of a hot spot pattern. Lin and Tantawi [97] use this method to determine the improvement in the probability of acceptance by adding buffers (from unbuffered to buffer size 8) and found an upper bound on the tree build-up time (the time until the saturated tree has formed).

The effect of hot spots on buffered networks containing Type B and Type C switches (see section 2.1) has also been studied. Tamir and Frazier [133] simulated the performance of four switch architectures, one of Type C and three representing different implementations of Type B architectures, in a 64×64 Omega network composed of 4×4 switches with buffers of size 4, and found that all structures saturated at a throughput of 24 percent with a 5 percent hot spot rate.

In [10], Atiquzzaman and Akhtar present a method for analyzing the performance of an Omega network composed of Type C switches with a queue of size one at each input. The iterative Markov chain analysis they present assumes that a buffer in a switching element may be in one of four stages: empty, containing a newly arrived message, blocked for the upper output link, and blocked for the lower output link, and does not generalize easily to larger queue sizes. As in [66, 135, 150], the stage cycle is split into two phases for modeling purposes. In the first phase, availability of buffer space at the next stage is determined. In the second phase, messages may move forward one stage if the next stage buffers are ready to accept them. Under hot spot traffic, routing probabilities at a switch are determined recursively for each of the i classes of switches at stage i (see also [99]). Their graph of results for a network of 8 inputs and outputs shows a maximum bandwidth of around 3.1 (average active memory modules) with the percentage of requests for the hot module at 30 percent. This compares with a value of approximately 3.3 from Liu's bound on steady state bandwidth in [99] for the same hot spot rate and network size but with unbounded queue size.

Although assumptions about switch architecture are not given, the 8×8 and 16×16 networks studied by Sivaram in [128] are apparently composed of Type C switches, since a throughput reduction is observed under uniform traffic even with unbounded buffer size. The throughput is much worse and the delays much greater when hot spot traffic is introduced, but it is difficult to compare the results of this study with other reported results because the message generation rate is given in terms of messages per microsecond, rather than messages per switch cycle, and the assumed service rate of the switches is not given in the paper.

Hot spots in direct networks

Full crossbars show no degradation in acceptance probability of other messages due to hot spot traffic, since there is no conflict for resources within the network, but the hot spot traffic itself is still serialized. Pinsky and Stirpe [120] analyzed a circuit-switched optical crossbar with asynchronous arrivals, taking into account conflicts for inputs as well as outputs, and found no significant effect on uniform traffic due to small hot spot percentages. However, their model did not take into account the overall system in which lost hot spot messages must be regenerated until satisified, effectively forming a queue at the inputs. Kurian and Thazhuthaveetil [82], considering this effect, found that the *relative* bandwidth degradation due to hot spots is actually higher in crossbars than in multistage networks.

Studies of other direct networks have also shown degradation in performance similar to that in multistage networks. In simulations on a binary n-cube network done by Abraham and Padmanabhan [2], in a 1024 node system with hot spots, buffer sizes of less than five always resulted in a deadlock at 100 percent offered load when using dimension-order routing. Even with deadlock-free LR routing, there is a disastrous performance degradation when a hot spot occurs. Buffer sizes of 10 were needed to avoid this problem. Their results for a single synchronization cycle show little degradation of background traffic due to hot spot traffic, but hot spot messages themselves are necessarily serviced quite slowly.

Badouel et. al. [11], in a paper primarily devoted to demonstrating improvement in performance of a routing algorithm based on a line drawing technique over dimension-order routing in k-ary n-cubes, simulated hot spot traffic within a buffered network where blocked messages are discarded and showed a decrease in saturation throughput from 100 percent of capacity to 65 percent with the introduction of a 1 percent hot spot using dimension-order routing for a 6-ary 3-cube. However, it is not clear how their production and reabsorption models of node behavior correspond to the more usual steady-state renewal model.

Dandamundi and Eager [32] simulated binary hypercube networks under a model which considers the effect of both global and local traffic. For a 256 PE network, with a hot spot rate of 1 percent and local non-hot spot traffic at 75 percent of the total traffic, the saturation throughput was only 76 percent of capacity using non-adaptive routing, despite the large percentage of local traffic. With an adaptive routing algorithm using unbounded buffers and allowing messages to choose the shortest queue on a shortest path, the effect of hot spot traffic on regular traffic was negligible, but when buffers were bounded, there is significant degradation at hot spot rates greater than 4 percent.

Adve and Vernon's closed queueing network analysis of bidirectional and unidirectional tori and bidirectional meshes, validated by simulations with one and four outstanding requests per processors, showed no tree saturation due to hot spot traffic on 64 PE systems, and no significant degradation of response time from the hot node for hot spot rates of less than 10 percent [4]. They ascribe the difference between their results and Pfister and Norton's results for 64 PE systems to the difference between closed and open queueing models, that in a closed system the round-trip delays prevent enough messages being issued to created the tree saturation effect. However, some of the difference they observe is due to the difference in topology between direct and indirect networks. For a bidirectional torus, with uniform traffic and four outstanding messages, the total effective request rate per processor was bandwidth-limited to less than .15 packet per cycle, according to their results. Thus, even at a hot spot percentage of 20 percent, the highest they examined, the total rate from all sources to the hot spot would still be less than $64 \times .15 \times .20 = 1.92$, and no individual link would average more than half that traffic, because of multiple links leading into the hot node. In a 64 PE multistage indirect network, in contrast, because of both the greater uniform bandwidth and the limited bandwidth at the single link into a hot node, lower hot spot percentages saturate the link, even if the number of outstanding messages is limited. Our own simulations show, for example, that with four outstanding requests per processor and 100 percent offered load, effective throughput falls from 38 percent for uniform traffic to 24 percent for 5 percent hot spot traffic, while average round-trip delay increases from 18 to 31 switch cycles.

1.2.6 Solving the hot spot problem

Assessing the merits of different solutions to the hot spot problem requires complicated tradeoffs between the allocation of hardware and software resources. Many solutions have been proposed, but relatively few have been carried out to the stage of detailed implementation so that relative costs can be assessed. Solutions may be classifed by the kind of improvement made, and by the kinds of resources dedicated to the improvement. Performance may be improved in several ways:

1. Software techniques can be used to cut down the number of accesses made to any single memory location.

2. Extra links and buffers in the network and at processors and memories may be provided, improving performance for all types of congestion, including hot spot.

3. Tree saturation may be avoided and the effect on background traffic may be mitigated, by preferentially dedicating resources to regular rather than hot spot traffic.

4. Extra service may be provided to hot spot requests, to remove them rapidly from the network, by combining the requests in hardware as they pass through the network.

Hardware combining provides the most complete solution, since it eliminates the bottleneck at memory for the hot spot requests and thus avoids tree saturation altogether, but has often been considered to be too expensive.

In this section we first describe hardware combining, and then describe other techniques that have been proposed to handle the hot spot problem. Some of these techniques can be understood as approximations to hardware combining in the network using software or hardware local to the processor; others do not eliminate the bottleneck at memory, but attempt to alleviate the detrimental effect of hotspots on overall bandwidth and on processors not involved in access to hot spot locations. In section 1.2.7 we review studies of the effectiveness of hardware combining and in section 1.2.8 we describe a variety of methods for implementing hardware combining that have been proposed.

Combining memory requests

Combining of accesses to read-only locations was proposed as part of the CHoPP (Columbia Homogeneous Parallel Processor) project [131]. To achieve a "stochastically conflict-free memory/interconnection system" [72, 71], the CHoPP design included the deliberate random allocation of memory addresses and the inclusion of cache-like "repetition filter memories" (RFMs) in the switches of the interconnection network. A read access in such a network would create an entry in each RFM on the path to the memory module; subsequent accesses could be satisfied by the closest RFM. The design assumed that all accesses to shared read/write data would be made using buffers that could be written by only one process and read by only one other process, and thus no shared read/write item would ever have more than two messages that referred to it traversing the network at the same time.

The NYU Ultracomputer project generalized this idea to a less restricted model of interprocessor communications and proposed combining fetch-and-ϕ operations (which include loads and stores) to shared memory at switches in the interconnection network [58, 57]. Fetch-and-$\phi(X, e)$, where X is an integer variable and e is an integer expression, returns the (old) value of a memory location X and replaces it with $\phi(X, e)$. In early work by the Ultracomputer project, "add" was the only ϕ considered, and the idea of hardware combining at network switches was developed in the context of devising an efficient implementation for fetch-and-add [54, 55]. Gottlieb and Kruskal [57] showed that this method of combining could be used for any associative operative ϕ. Snir and Solworth [130] outlined an implementation of a combining switch that was the basis of the work described in Chapters 3 and 4. Kruskal, Rudolph and Snir [75] extended the class of combinable read-modify-write operations to include some operations that are not associative.

With read-only shared data, the order of access of different processors never matters. But with read/write shared data, the result of concurrent fetch-and-ϕ operations to the same variable must satisfy the *serialization principle* [83]. Fetch-and-ϕ operations simultaneously directed at X must cause the final value of X to be the same as the result of executing the operations in some serial order, and each operation must return a value corresponding to an intermediate value of X in a serialized execution.

We illustrate how combining is performed, using the operation of addition as ϕ: When two fetch-and-adds referencing the same shared variable, say fetch-and-add(X, e) and fetch-and-add(X, f), meet at a switch, the switch forms the sum $e + f$, transmits the combined request fetch-and-add(X, $e + f$), and stores the value e in its local memory. When the value Y is returned to the switch in response to fetch-and-add(X, $e+f$), the switch returns Y to satisfy one request, fetch-and-add(X, e), and $Y + e$ to satisfy the other, fetch-and-add(X, f). Assuming that the combined request was not further combined with yet another request, memory location X becomes $X + e + f$. If other fetch-and-add operations updating X are encountered, the combined requests are themselves combined.

The associativity of addition guarantees that this procedure gives a result consistent with the serialization principle. Other associative operations can be combined in a similar manner. Since combined requests can

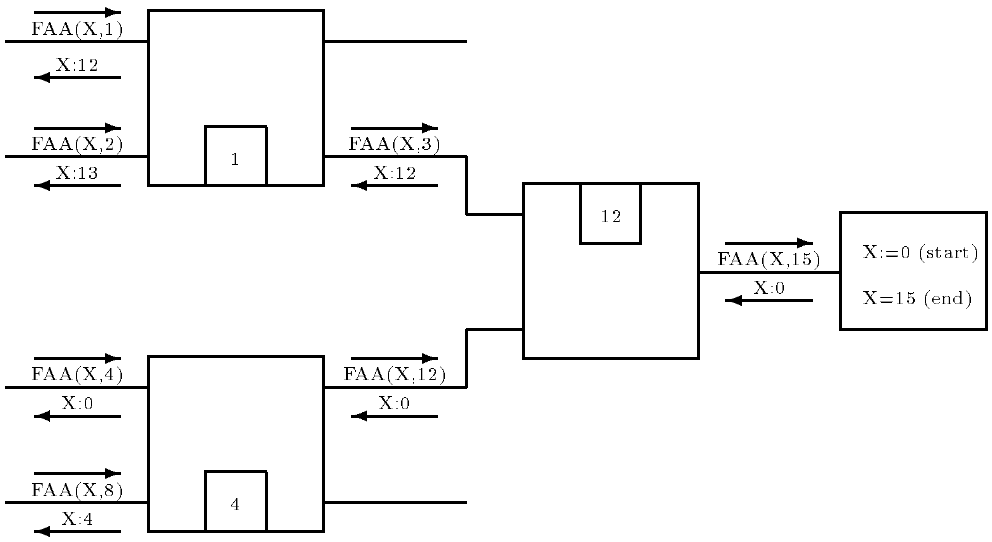


Figure 1.4: Example of combining fetch-and-add operations.

themselves be combined, any number of concurrent memory references to the same location can be satisfied in the time required for one shared memory access from a single PE.

Figure 1.4 shows a particular example of four fetch-and-add messages combining in two stages of switches before the destination. The results returned are as if the fetch-and-add (FAA) operations were executed sequentially in the following order, on a memory location X that was initially zero:

FAA(X,4)
FAA(X,8)
FAA(X,1)
FAA(X,2)

Note that the value saved for decombining may come from either input port, depending on which message arrived "first."

A formal proof of the correctness of combining, in the sense that the observable behavior of a combining memory system is a behavior that could be observed in a correct non-combiningsystem, is given by Kruskal, Rudolph and Snir in [75]. They consider a memory system to be "correct" provided that each memory location receives a sequential stream of requests from processors, obtained by merging the serial streams of requests directed to that location by individual processors, and that these requests are processed in the order in which they appear. However, even with such a "correct" memory system, it may be necessary to *fence* or delay a processor until some previous memory access has completed, in order to maintain sequential consistency and prevent hazards due to interdependencies when processors have multiple requests outstanding to different locations. Analysis of when such fences are needed can be found in [126].

As noted in [75], this method of combining can be used in a variety of interconnection networks, as long as requests return on the path on which they were sent. Extra logic is required at nodes in the network, but no additional wires are needed between nodes (though, depending on packaging, additional pins may be required for intranode wiring).

Avoiding hot spots in applications

As an alternative to hardware combining, a variety of software techniques for avoiding hot spots have been proposed. Some of these techniques require taking into account both the algorithm to be implemented and detailed features of the network in the target architecture; see, for example, the discussion of global summation on mesh architectures in [13]. The problem of mapping algorithms to architectures in such a way as to avoid communication conflicts and bottlenecks has been the focus of a great deal of research in the context of particular applications.

The prevalence and difficulty of removing hot spots in real applications is still an open question. In [33], Darema-Rogers *et. al.* studied the memory access patterns of three scientific programs using traces obtained with the PSIMUL simulator running under the VM/EPEX parallel environment. They found that the fraction of fetch-and-add instructions executed to shared variables ranged only from 0.0003 to 0.008 percent of instructions. However, even with such low average figures, tree saturation still occurred when traces were used to drive a network model.

In [50], Glenn *et. al.* found three types of hot spots in the applications they studied using the Horizon simulator:

1. Hot spots due to simultaneous access of shared read-only data.
2. Hot spots due to stride access.
3. Hot spots due to heavy access of test and set variables.

The first type of hot spot can be avoided by distributing copies of the data, but, as Glenn and Pryor describe in [49], identifying the variables responsible for the hot spot is not trivial. The variables that turn out to be hot spots may be surprising to the programmer and may change as the program is scaled to larger machines. Like Glenn and Pryor, Bianchini et. al. [15] have also observed that increasing the number of processors may increase both the number of hot spots and the degree of contention for each hot spot. Using traces from the Tango simulator and then simulating the contention effects of those traces when accessing a distributed memory, they found that eliminating hot spots on an individual basis can cause other hot spots to worsen.

Glenn *et. al.* found the second type of hot spot, due to stride access, to be difficult to eliminate with software techniques. However, stride access tends to create hot memory modules, rather than hot locations, and thus can be alleviated with randomized memory bank addresses.

The third type of hot spots, which involve repeated access to synchronization variables, has been the target of software combining techniques which mimic the effect of hardware combining using a tree of locations, each of which will receive only a constant number of references, to replace each hot spot location [149, 148]. These algorithms utilize a software tree with each processor assigned to a leaf; $\log N$ locations in different memory modules are used to combine data from a fixed number of processors. Although most processors only ascend a few levels, all must wait for the tree to be fully ascended and subsequently descended for each access to the tree. Since traversing each level of the tree requires $\log N$ transit time through the interconnection network, asymptotically each access to the combining tree will require $\Theta(\log^2 N)$ time with software combining, as compared to $\Theta(\log N)$ time for accessing a memory location if all accesses to the hot spot are combined in hardware and require only one network traversal.

Software combining techniques, assisted by hardware local to the processor, have been used to devise synchronization techniques that avoid generating hot spots and cut down on traffic in the network due to busy waiting. Synchronization primitives that make use of hardware supported *syncbits* are described in [53]. Reader-writer synchronization that rely on fetch-and-ϕ operations and on spin-waiting, without network access, on local shared memory are described in [103]. Such methods are significantly more complicated than synchronization techniques that rely on hardware combining, such as those described in [59, 75], and show the same difference in performance as that between hardware and software combining.

Adding resources at the memory and network

Making the network faster, in the presence of hot spots, will only deliver messages more quickly to the bottleneck at memory. However, alternative paths in the network may delay the onset of tree saturation and allow PEs not involved in the hot spot to make progress. If all PEs are accessing the hot spot in addition to other locations, they must be able to have multiple outstanding requests in order to take advantage of alternate paths around the hot spot tree.

Tzeng [138, 139] proposed using Type B switches and multiple queues at inputs to the network to lessen the impact of tree saturation due to hot spots. The multiple queues at the input are selected based on high order address bits, and the protocol for sending messages to the switches allows a message to be sent as long as one of the queues at the input has a message that follows a path through a non-full first stage output buffer. His simulations of a 256×256 network composed of 2×2 switches, with 2 queues at each input, show a maximum throughput under uniform traffic of about 75 percent, and a maximum throughput under 5 percent single hot spot traffic of close to 40 percent, a considerable improvement over the results for the Type C switches simulated in [36]. However, this scheme is still subject to input blocking from tree saturation if there is more than one hot spot in the system and each of the queues at the input has a message at its head destined for a hot spot. Furthermore, this scheme, especially in the variant with dynamic allocation of storage to longer queues, requires somewhat complicated flow control between the stages, since acceptance of a message depends on its destination in the next stage.

In a 256 PE wormhole-routed k-ary n-cube, under the low traffic assumption that the injection rate into the network is much less than the bandwidth of the physical channel, providing multiple consumption channels at each processor improved hot spot performance considerably with adaptive routing, though the routing bottleneck dominated when dimension order routing was used [12].

Hardware resources in the network may also be used to support synchronization in a way that cuts down on hot spot accesses. In [8], Andrews, Beckmann and Poulsen described hardware designs that cut down on hot spot accesses by allowing notification and multicast to be used instead of polling or spin-waiting on a synchronization variable. Two hardware designs for implementing notification and multicast in packet-switched multistage interconnection networks were presented. In each, multicast and notification are controlled by the memory module using a directory, in a way similar to directory-based cache coherence schemes, but without implementing all the requirements of a full cache coherence scheme.

The Switch Table Multicast Network (STMN) has an implicit, network-based directory, distributed over the reverse network switches. The Hybrid Multicast Network (HMN) has an explicit, memory-based directory, distributed over the memory modules. In both STMN and HMN, the number of simultaneous multicast trees that can be supported is limited by the size of the directory: for STMN, by the routing tables in the reverse path switches, for HMN, by the multicast tables at memory. Switches to implement the reverse path in an STMN must have an additional input to and output from the switch crossbar. The hardware costs for either scheme are non-negligible, comparable to or greater than that required to implement a combining network.

Simulation experiments compared the average time to synchronize for a tree barrier with broadcast exit using HMN and STMN with a symmetric tree barrier using polling. These were the fastest barrier algorithms for each scheme. The notification barriers outperformed polling by almost a factor of two for large numbers of synchronizing processors. However, a comparison of the effect of synchronization on the latency of background traffic, using symmetric tree barriers with HMN and STMN as well as with polling, showed no improvement over polling in the effect on background traffic, and it is possible that using the broadcast exit, as in the best algorithm for HMN and STMN, would have a detrimental effect on background traffic.

Preferential allocation of resources

If messages destined for a hot spot are prevented from tying up resources within the network, tree saturation can be avoided, and "cold" processors, which do not require service from the hot memory module, will see improved network performance. However, such methods will do nothing to improve service for "hot" processors, and may actually make it worse.

Rotating messages to the back of the buffer and comparing memory addresses to ensure that no buffer acquires more than one message destined to the same memory module are techniques suggested by Dias and Kumar to avoid degradation to uniform traffic due to hot spots [36]. Simulations of this method, using Type C 2×2 switches with buffers of size 4 in a 256×256 network, showed a maximum throughput for uniform traffic of around 43 percent, compared to 53 percent for Type C switches that do not reject or rotate messages, and a maximum throughput of 37 percent for 5 percent hot spot traffic, compared to 8 percent with the simple switches. Several extensions of their scheme that improved throughput for uniform traffic were also studied, including comparing memory addresses only when the queue length is greater than one, and giving lower priority to messages once they have been rejected. No figures were given for the increase in delay experienced by hot spot messages. Their model seems to have assumed no queueing of blocked hotspot messages at network inputs, and no limit on the number of outstanding requests within the network; thus, no detrimental effect of long delays for satisfying hot spot requests would be seen in the throughput figures from their simulations.

Ho and Eager [61] proposed simply discarding one of any two messages destined for the same memory location that contend at a switch. In simulations of a 1024 PE network, they limited the total number of PEs that may have hot spot messages outstanding to 512 (i.e, when 512 PEs had an outstanding message to the hot spot, no more hot spot messages would be generated until one of these outstanding requests was satisfied.) A PE was allowed to have at most one memory request outstanding at a time. Switch-initiated retransmission, in which a switch sends a message back to a PE when it discards a message, was used. Under such conditions, the maximum achieved throughput for all traffic using the discarding strategy was almost 50 percent, compared to 30 percent when all messages are queued. Simulations of a combining network had a throughput of around 90 percent. In order to determine when to discard messages, Ho and Eager's scheme would require the same associative matching as a combining network, but would not require an adder or decombining logic. It would, however, require logic to send a negative acknowledgement to the processor; this could have an effect on network packaging, since it would require communication between the forward and return paths of the network, as does combining.

Feedback from the memory modules can be used to block messages destined for a hot memory module from entering the network, whenever the length of the queue at the memory module is above a certain threshold. In [123], Scott and Sohi showed that this method can improve bandwidth and decrease latency for messages destined to cold memory modules, as long as only a fraction of the processors are "hot", that is, are accessing the hot memory module at a rate greater than that of uniform traffic. In their simulations of a 256×256 network of Type C switches with queue sizes of 4 (or, effectively, 5, because of additional input latches, see section 2.1.4), the most improvement was shown when 50 percent of the processors were hot. In this case, using a feedback scheme with a threshold of 4 and adaptive backoff when a memory module changes from hot to cold, bandwidth increased from about 10 percent to about 35 percent for an 8 percent hot spot rate, compared to a bandwidth under uniform traffic of 75 percent. However, no improvement is shown over a regular Omega network when all processors are accessing the hot spot, and improvements for hot spot rates lower than 8 percent are much more modest.

In a refinement of Scott and Sohi's ideas, Farrens, Wetmore and Woodruff in [44] explored the use of large queues at the memory modules combined with a feedback damping scheme that they call *bleeding* to alleviate tree saturation and improve effective bandwidth. Although their methods show slight improvements over the results of Scott and Sohi, even in the situation most favorable to feedback schemes, with 50 percent of the processors accessing the hot spot, the effective bandwidth increases by less than 50 percent over the low bandwidth available from an unmodified Omega network, when the hot spot rate at the hot processors is 2 percent. The cost of their scheme includes queues at memory, a bus to allow the memory network interface to communicate with all processor network interfaces (PNI), possibly a buffer at the PNI, and logic at the PNI to do the damping.

1.2.7 Effectiveness of combining

Hardware combining has the potential to execute N operations directed to the same location in one network traversal, but, if PEs are issuing messages asynchronously and combinable messages are mixed with other traffic, it is not immediately clear that enough combining will actually take place to improve performance. A number of studies have shown that asynchronous hardware combining is nevertheless effective in both preventing tree saturation and providing reasonable delays for messages sent to a hot spot. In Pfister and Norton's paper on hot spots [119], simulations showed combining to be effective for networks up to 64 PEs. Wong [144] also simulated 4 and 6 stage networks and explored the sizes of queues and wait buffers needed to get good combining performance. Simulations by Liu [100] showed the effectiveness of a variety of combining schemes for systems of size up to 1024 PEs. Simulation results showing the effectiveness of the combining switch described in this thesis are given in Chapter 4.

Lin and Tantawi [97] extended Lin and Kleinrock's method [96] to analyze the performance of combining with various offered loads and queue sizes for a 9-stage combining network. Merchant [104] also showed combining to be effective using a Markov chain model for unlimited combining in which messages are dropped whenever a queue is blocked. This model makes the approximations that the output of the switches is a Markov chain, that there is no time correlation in the output stream affecting the probability that the output of a switch is combinable, and that if there is a combinable message in a queue, its location in the queue is distributed randomly over the length of the queue.

An important parameter in studies of combining effectiveness is the *multiplicity* of the combining method; that is, the number of messages that may combine in a single stage. Theoretically, a message could combine with an unlimited number of other messages in a single stage, but there are two problems with this method: it is more complicated and expensive to implement, especially for decombining, and the burstiness of multiple decombines may adversely affect performance on the return path from memory. Alternatively, two-way combining, in which a message combines with at most one other message, provides a simpler implementation. The results of Lee, Kruskal and Kuck [87], using a model with infinite queues at the processor (and thus an indefinitely large number of messages in the system) indicated that two-way combining is not adequate to prevent saturation for indefinitely large networks [86, 87], showing degradation for systems with as few as 8 stages. On the other hand, studies using practical assumptions have shown good performance for two-way combining networks of 10 stages or more [68, 67, 100]. The question of the need for and implementation of more than two-way combining is considered at length in Chapter 5.

If combinable messages entering a network are synchronized so that no combines are missed, two-way combining in a network composed of 2×2 switches is sufficient to combine messages issued from all processors into a single message at memory. To implement a synchronous PRAM, Ranade [121] proposed sorting messages by destination before they enter the combining network, and ensuring that they leave switches in sorted order. This guarantees that messages destined to the same location will meet in a switch and combine, and obviates the need for greater than two-way combining. However, it increases the minimum latency of a message and makes pipelining difficult, as we discuss in section 1.2.8.

Wilson, in [143, 142], studied what he calls *opportunistic combining networks* in message-passing multicomputers with hypercube and torus topologies. In opportunistic combining networks, references to distributed data structures, such as queues and stacks, are done by accessing *index servers* for each such distributed data structure through a spanning tree of processor nodes. Messages performing the same operation on the data structure may be combined at any node in the tree. The responses will be decombined in such a way that the processors originating the messages receive a pointer to different elements of the distributed data structure, which will be stored in many processor nodes. Simulations of opportunistic combining networks show greatly improved throughput, especially under heavier loads in torus networks.

Hsu and Yew [62] simulated the performance of barrier synchronization and parallel queue algorithms, comparing the performance of hardware combining in a multistage network to that of no combining, single-stage combining and, for barrier synchronization, software combining. They maintained that parallel queue accesses are not as well structured as for barriers, and that it is not realistic to use software combining in that case. Hardware combining in a multistage network had the best performance in all cases. Both single-stage and multistage combining showed considerably better performance than no combining or software combining.

1.2.8 Combining implementations

The original combining switch implementation proposed for the NYU Ultracomputer [55] allows combining to be performed on any message routed through a multistage network. In such a scheme, combinable messages do not need to be separated from the regular traffic stream, and use the same wires used for all messages. Instruction fetches, data loads and stores, as well as special fetch-and-ϕ operations, may all be combined without the programmer or compiler specifically indicating that this should be done. Messages may enter the network asynchronously, without coordination between PEs, and a PE may have multiple messages outstanding in the network. This implementation, pictured in Figure 1.5, requires comparators within the ToMM queue to detect messages heading for the same memory address, an ALU to perform the combining operation as messages exit the ToMM queue, and a wait buffer to hold information needed for decombining as well as a ToPE queue. An efficient implementation and partitioning of this high-level scheme is described in Chapter 4.

The concern with this scheme is that it would be too expensive, or ineffective because many combines are missed due to asynchrony. Many alternative implementations have been proposed.

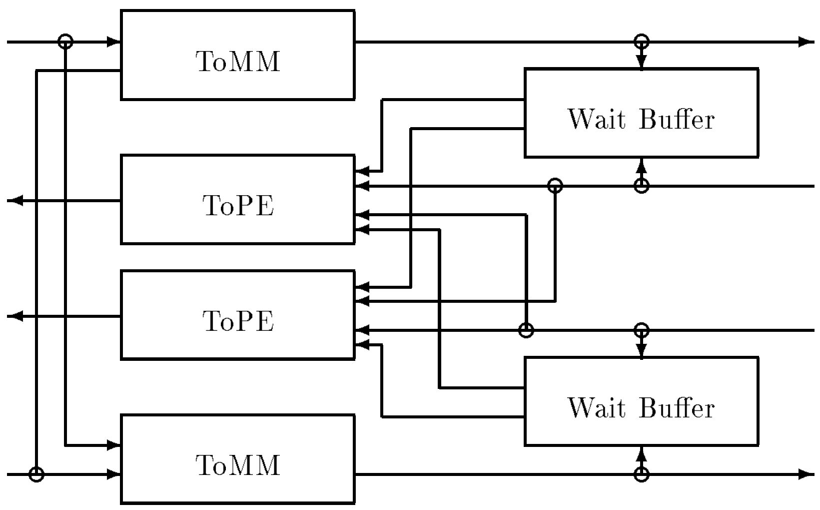


Figure 1.5: NYU Ultracomputer combining switch.

RP3 combining switches

The IBM RP3 (Research Parallel Processing Prototype) [118] was designed to connect 512 microprocessors through a multistage interconnection network. Processors and memory were co-located, and the memory at each node was partitionable into shared and private memory. The RP3 project cooperated closely with the NYU Ultracomputer project, and the two architectures have much in common. The RP3 project, however, planned a separate combining network, with ordinary traffic passing through a high-speed water-cooled bipolar network with 8-bit wide data paths and 4×4 switches. Since the standard microprocessors used as PEs could not tolerate much latency to memory, a network design that mimimized latency was considered to be of paramount importance, and it was not considered possible to implement a combining switch with low enough latency. The design of a combining switch efficient enough to carry a large volume of traffic was not given high priority, as it was assumed that the volume of traffic using it would be small. Furthermore, it was assumed that the percentage of hot spots in the combining network would be quite high.

As part of the RP3 project, Wong [144] simulated a 16 PE network with combining switches, under 50 percent and 90 percent hot spot loads, to determine optimal queue and wait buffer sizes. The simulated design was different from that described in Chapter 4 in the following ways:

1. Separate buffers at input ports are used to hold messages until they can be inserted in the queues. Separate simulations were done under the assumption that a messages from each input port could both be accepted into a queue in a single cycle (equivalent to a Type A switch) or that only one message could be accepted into a queue per cycle (a variation of a Type C switch; see section 2.1.4).

2. Comparison of an incoming item with all items in the queue must be done before insertion of the item in the queue.

3. The combining logic was in series with the off-chip path.

4. Instead of using an associative wait buffer, a *Snir field*, which contains the stage number and location within the wait buffer, was transmitted with each combined message and used for decombining. Multiple combines at different stages were handled by storing the Snir fields from preceding stages in the wait buffer and replacing them as part of the decombining operation. The advantage of this method is that values in the wait buffer can be accessed by location instead of by associative search. The disadvantage of this method is that the message returning from memory must have its Snir field altered before continuing on the return path, and thus cannot be inserted in the queue or sent on to the next stage until wait buffer access and decombining has completed.

No implementation for this design was ever published, though Pfister and Norton [119] at about the same time period claimed that cost and size estimates, using silicon and packaging data, were made of a combining switch as part of the RP3 project and indicated that "message combining increases switch size and/or cost by a factor of between 6 and 32."

In later work by members of the RP3 project, Hsu *et. al.* [141] described an implementation that include the combining of heterogeneous op codes and stores messages in central queues as linked lists. Each linked list is associated with both an output port in this stage and the desired output in the next stage, allowing the multiple clear-to-send protocol described in [100] to be implemented. Such a shared central queue allows better utilization of storage space, at the price of considerable complexity in control and delay in insertion. Like Wong's design, comparison of the incoming message must be done with all elements before insertion. If the queue is empty, a bypass path allows a message to go directly to the output without being inserted in the queue. This design was supposed to have been implemented in one micron CMOS, but no clock speeds for a fabricated chip have been published. The RP3 project was terminated before a combining network was actually used in the system.

Tree-structured combining networks

If messages likely to combine are a small part of the regular traffic stream, and if these messages can be separated from that stream in advance, tree-structured combining networks can be used, since the network does not require much bandwidth. Lipovski and Vaughn [98] describe a bit-serial circuit-switched implementation of fetch-and-ϕ, suitable for SIMD systems or synchronized operation in MIMD systems, that requires only 5 gates per node. In their scheme, PEs with associated memory are at the leaves, connected through a fetch-and-ϕ network very similar to a carry-lookahead adder. All PEs need not be involved in every fetch-and-ϕ operation, but only one memory location at a time may be the object of an operation.

In Tzeng's [137, 140] more complicated scheme, multiple hot spots may be active concurrently and asynchronously. Messages that have been designated as combinable are removed from the regular routing network at the input to stage i, where i is chosen depending on the number of hot spots active in the network, sent through all or part of a combining tree and reinserted at an input or inputs to stage i of the routing network. The combining structure must be reconfigured to suit the expected number of hot spots in an application. In order to have effective combining and keep the configuration unchanged during the course of a computation, the number of concurrently active hot spots must not vary too greatly. The logic for components in this scheme is simple, but extra connections are required at each node in the regular routing network, adding to wiring costs and packaging difficulties.

The tree-structured control network of the CM-5 [93] provides a set of operations useful for synchronous MIMD operations: broadcasting, four types of combining operations, and global single-bit OR operations. Separate FIFOs at the network interface are used for each type of control network function, and the network is pipelined, so that several control messages can be active at once. All PEs are potentially involved in every operation on the control network, though each PE can choose to abstain from certain control network operations. The four types of combining operations supported are reduction, forward scan, backward scan and router done. *Reduction* can be performed on 32-bit messages from all processors using one of five operators: bitwise logical OR, bitwise XOR, signed maximum, signed addition and unsigned addition. Reduction returns the result of combining all values using this operator to each processor. *Scan* operations take a binary associative operator $\otimes$ with identity I and an ordered set $[a_0, a_1, \ldots, a_{n-1}]$ and returns the ordered set $[I, a_0, (a_0 \otimes a_1), \ldots, (a_0 \otimes a_1 \otimes \cdots \otimes a_{n-2}]$ [18]. Such operations could be used to simulate many of the actions of combinable fetch-and-ϕ operations. In the CM-5, forward and reverse scans can be implemented for any of the reduction operations, and scans may be segmented.

Combining networks on sorted traffic streams

If all references to memory in a single time step are sorted in order of lowest memory address before they pass through a combining network, and if they leave each switch in the network in sorted order, waiting until they are sure no lower message will arrive on the other port before they leave, then any reference will be sure to meet any message with which it might combine at the first switch that they both traverse. Thus no more than k-way combining is ever necessary, for $k \times k$ switches. By always ordering the inputs of the switch in the same way when combining is done, a fixed serial order of execution can be given to any set of processors for concurrent operations in a time step. Ranade [121] used these ideas, together with hashing of memory addresses, to develop a scheme for implementing a synchronous CRCW PRAM in which the probability, assuming a perfect random address map, that any memory reference takes more than $15 \log N$ steps is less than N^{-20}, where N is the number of nodes in the butterfly.

As originally described, this scheme involves six passes through a butterfly network, with a processor at every node. For a multistage interconnection network, multiple passes could be eliminated in favor of sorting hardware at the inputs to the network. Since messages are kept sorted, replies arrive at a switch in the same order as the messages were sent out, so decombining can be done in the same order that combining occurred and does not require an associative memory. However, a high cost is paid both in the minimum latency of a message, because of the delay induced by waiting to leave in sorted order, and in low utilization of network links, which remain inactive whenever sending a message might result in messages being sent out of order. In addition, the destination address of a message must be compared to that of the last message from the other input port before it can be sent out, even if there are no other messages present in the switch; this puts a limitation on the switch cycle time, since the comparison cannot be done in parallel with sending the data off-chip, as it can for Ultracomputer-style combining (see Chapter 4). Implementations developed by a group at the University of Saar and described in [1, 42] use pipelining, simplified sorting hardware and overlapping networks to address some of these problems.

An end-to-end synchronous method using combining in a Batcher double Omega network that is non-blocking for permutations was developed by Amano and Kalidou [7]. In this circuit-switched network, the head of a message, as it is routed through the network, creates a virtual circuit for the following data and also for the return transmission. Combining is implemented by forwarding only one request and holding both return path circuits. The overall network design consists of a Batcher sorting network followed by a double-sized Ω network used as a "distributor" and another double-sized Ω network used as a "concentrator"; adders at the output of the Batcher network are used to create routing addresses that allow conflict-free transmission through the double Ω network if all messages are headed for different memory modules. Combining is implemented for read-read, write-read and test-and-set operations destined for the same location in memory. Conflicts can occur only if two messages are destined for the same memory module but not for the same location; in that case, a negative acknowledgment is sent on the return path..

The design for the Batcher double Ω network uses the lower-bandwidth but simpler virtual circuit switching strategy to get switches that are simpler and thus may have a higher frequency clock. To avoid severe loss of bandwidth due to conflicts, extra switches and stages are added to the network. The Batcher network requires $(\log_2 N(\log_2 N + 1))/2$ stages, with $N/2$ switches each stage; the double Ω network requires $2(\log_2 N - 1)$ stages, with N switches each stage. Thus the minimum delay scales as $O(\log^2 N)$ rather than $O(\log N)$ for a simple Ω network, and the cost in wires and components scales as $O(N \log^2 N)$ rather than $O(\log N)$.

Combining as part of cache-coherency support

Recent work proposes hardware support for combining in certain cases as part of hardware-supported cache coherency in shared memory multiprocessors, like that in the Stanford Dash [94] and the MIT Alewife [5]. Bianchini and others [15] propose distributing hot data to multiple memory modules as part of the the cache coherency protocol, using a combination of eager sharing and combining trees called *eager combining* to provide hardware-supported replication. Assuming that certain physical address ranges are marked as hot, a fixed number of server nodes are designated for each hot physical page. The protocol distributes data to servers using eager sharing; the servers then satisfy requests from multiple client nodes, combining multiple requests that cannot be satisifed immediately. On the transition of a hot data block from modified to read-shared, the block's home node multicasts the data to the block's servers; on the transition from read-shared back to modified, the clients and servers must have their copies of the block invalidated. While cutting down on tree saturation at the home block, more messages may be sent on transitions than in a cache-coherency protocol without eager combining, if data is multicast to locations where it is not actually used. Other difficulties in practice include methods for designating hot data and for determining the appropriate number of servers.

Other combining schemes

Hsu and Yew [62] propose and evaluate a design for a recirculating single-stage shuffle-exchange combining network, which would be used just for handling hot spot traffic, while regular traffic passes through a high-bandwidth non-combining network. In an RP3-like environment, with separate networks, their simulation data makes a good case that a single-stage combining network is more cost-effective than a multistage combining network. However, when the cost of the single-stage combining network is added to the cost of the network for regular traffic and compared to a network in which combining is integrated into routing for all traffic, the hardware cost advantage is not clear, depending on the relative weights given to wiring and logic costs. Furthermore, the integrated combining network may provide advantages in combining read-only instruction fetches and shared data that their simulations of barrier and parallel queue algorithms did not measure.

Merchant [104] proposed and analyzed a modified combining scheme based on low priorities for hot spot requests. This scheme is similar to that of Dias and Kumar (see section 1.2.6), except that instead of just recirculating hot spot messages, they may combine multiple times while waiting in the queue. Though this scheme was shown to improve throughput, hot spot requests must be identifiable within the network, and the modified scheme increases delays for the hot spot requests at the same time that it improves throughput for background traffic, which may not be desirable for overall code execution. This scheme does not seems to have any cost advantage over other proposals for combining in multistage networks.

1.2.9 Summary

A number of complicated issues are involved in the design of an interconnection network for a parallel computer, and a great deal of research has been carried out in this area. Many architectures are reasonable, and a great deal of implementation experience will be required before realistic cost and performance figures can be used to compare different architectures over a range of applications. To acquire this implementation experience, one needs to pick a promising candidate architecture and construct an efficient implementation.

To implement a network for the NYU Ultracomputer, we are first concerned with providing scalable bandwidth. As system size grows, multistage networks provide the greatest system bandwidth at the least cost in wires, for uniformly distributed messages. To maintain bandwidth and limit latency when pinout considerations will not allow an entire message to be transmitted in a single packet, we use a cut-through switching strategy.

Hardware combining is provided for all traffic in the multistage network to avoid tree saturation due to hot spots and support fetch-and-ϕ operations for synchronization. While software techniques or a separate combining network could be used for the same functions, they are not likely to be as effective and represent significant recurring software costs.

The combining network must provide good service to all traffic, not just combinable messages, at acceptable cost. In the following chapters we analyze and describe the design of a switch for use in such a combining network.

Chapter 2

Performance of Switch Architectures Under Uniform Traffic

Our goal has been to design a combining switch that performs well under the expected preponderance of uniformly distributed traffic as well as hot spot traffic, so we have carried out analytical and simulations studies of the performance of switch architectures under uniform traffic. The first two sections in this chapter present a taxonomy of switch architectures and review the performance implications of the arrangement of buffers within each switch, summarizing joint work with Yue-sheng Liu [39, 101] and Ora Percus [115]. The third section explores the effect of changing the queue size and the number of outstanding messages on the performance of the preferred Type A and Type B buffered switches. The fourth section compares simulation results with multiple packet messages and finite buffers to the predicted performance of cut-through switching with infinite buffers. The fifth section looks at the effect of increasing the degree of the switch (from 2×2 to 4×4 or 8×8) with the constraint of a fixed number of pins per node.

2.1 Basic switch architectures

Consider a $k \times k$ crossbar switching component in a multistage interconnection network. Its basic function is to forward messages from any of its k inputs to any of its k outputs. It may include buffers to hold messages in case of conflicts for the output ports or blocking from later stages. These buffers may be associated with either input or output ports and may perform extra functions such as combining messages destined for the same memory location or sorting messages according to destination in a later stage.

Switches can be classified according to the presence or absence of buffers, and according to the location of the buffers [39]. These variations cause differences in performance, for the same cycle time and data path width. The illustrations below show 2×2 switches, for simplicity.

The interarrival time of requests at the first stage of the network is assumed to be geometrically distributed. The network is assumed to be an $N \times N$ square delta network, as defined in [111], composed of $k \times k$ switches, with $\log_k(N) = n$ stages. All performance figures in this chapter are derived from a traffic model in which messages from a given input are uniformly distributed among the outputs of a switch. Such a model is appropriate for switches in a multistage Ω network, if all the memory modules are accessed randomly and uniformly. However, such a model may not be appropriate for mesh-connected multiprocessors using dimension order routing, which results in most messages being routed "straight-through" either horizontally or vertically.

2.1.1 Unbuffered

The simplest switch design (see Figure 2.1), has no buffers at either input or output; such a node was used in the interconnection network of the BBN Butterfly Parallel Processor [24]. A protocol was used to kill messages that conflicted for an output port; these messages were retransmitted.

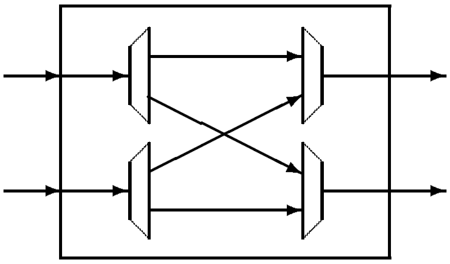


Figure 2.1: 2×2 switch with no buffers

The limited bandwidth of networks built from unbuffered switches is well known. Under the optimistic assumption, made for analytical tractability, that messages are generated independently to uniformly distributed random addresses on each cycle, the probability of an output at a switch in the i^{th} stage is

$$p_i = 1 - (1 - p_{i-1}/k)^k, \tag{2.1}$$

where $p_0 = p$ is the offered load on an input port to the network [111]. This can be approximated by

$$p_n = \frac{2k}{(k-1)n + 2k/p} \tag{2.2}$$

(see [77]). Thus for a square delta network with N PEs and N MMs, the throughput at each output port of the network is limited to $\mathrm{O}(1/\log N)$, holding k and p constant. The overall bandwidth of the network is then limited to $\mathrm{O}(N/\log N)$. Since retransmissions are likely to cause persistence of conflicts at higher loads, this is probably not an achievable upper bound.

The latency of a message, measured from the time a processor makes a request until it is satisfied, is more difficult to estimate because of transmission. Suppose, following the assumptions in [100, 101], that the processor can generate new requests at a rate b, independent of any responses it receives. The offered load on the network p will be b plus the rate r of rejected messages. Over time, if b is less than the maximum bandwidth of the network the network, retransmissions will accumulate until some offered load p minus the rate r of rejected messages produced by that p gives the desired b. At this point, the offered load should stabilize at $p = b + r$, and the output of the final stage $p_n = b$. We make the simplifying assumption that the traffic pattern of rejected requests is still uniformly and randomly distribed over the memory modules, and thus Equation 2.2 may still be applied. A similar traffic model for crossbars is described in [147]).

If a steady state throughput $p_n = b$ can be sustained, the offered load p must satisfy:

$$p = \frac{b}{1 - nb(k-1)/2k}. \tag{2.3}$$

Note that the term $1 - nb(k-1)/2k$ above must be greater than zero. This condition is automatically enforced by the condition that the request rate $p \leq 1$. Manipulating equation 2.3, $p \leq 1$ is equivalent to the following condition:

$$b \leq \frac{2k}{n(k-1) + 2k}. \tag{2.4}$$

Equation 2.4 gives the an upper bound for the maximum bandwidth achievable in an n stage unbuffered network.

Since to transmit b messages per cycle we need to make p requests per cycle (including re-transmissions), each message is expected to be transmitted p/b times. Let r be the average retry number ($r = 1$ for no retries). Then we have, from equation 2.3,

$$r = p/b = \frac{1}{1 - nb(k-1)/2k}. \tag{2.5}$$

In unbuffered networks there are no conflicts on the return path, as long as all messages use the same links on the return path and have equal delays at memory. When a message gets through the network without being killed during the conflicts on the forward path, the round trip takes $d_0 = 2n + 1$ network cycles, plus additional time for memory access that we will ignore for simplicity. Hence, if a request does not return after d_0 cycles, it must have been killed due to the conflicts, and must now be re-transmitted. The expected latency of a message, measured from the time a processor makes a request until it is satisfied, depends on the length of the timeout before re-transmission of a lost message; in the BBN Butterfly, the lost message was transmitted after a random period [23]. Since each message needs to be transmitted on average r times and each retransmission follows the previous transmission of the same message by at least d_0 cycles, the expected average delay $D \geq rd_0$. With Equation 2.5 we have the expected average delay

$$D \geq \frac{2n+1}{1 - nb(k-1)/2k}. \tag{2.6}$$

At the maximum bandwidth from Equation 2.4, the delay is thus

$$D \geq \frac{(2n+1)(n(k-1)+2k)}{2k}. \tag{2.7}$$

This $O(\log^2 N)$ growth in delay at the maximum bandwidth may in practice prevent the request rate from ever reaching the maximum bandwidth, and may thus be a more severe constraint than the $O(N/\log N)$ bandwidth limitation. For an extreme case of latency limitation, suppose that a processor can only generate a request after it has received a reply, and that it will always generate a request as soon as the reply is received. Assuming that receiving a reply or a notice of rejection each takes $2 \log N$ cycles (neglecting all delay at memory), p, the total traffic per input port cannot be greater than $1/(2 \log N) = 1/2n$. From Equation 2.2, the traffic per output is limited to

$$p_n = \frac{2k}{(k-1)n + 4kn} = (\frac{4k}{5k-1})(\frac{1}{2n}), \tag{2.8}$$

less than one-fourth that of Equation 2.4 for $n > 8$.

The latency and bandwidth values for unbuffered networks of up to 1024 PEs with 2×2 and 4×4 switches are shown in Figures 2.2 and 2.3. As can be seen by comparing these figures to those for buffered switches in section 2.3, the bandwidth available at the outputs is less than half that available with moderate-sized buffers. Comparing the values for unbuffered 2×2 and 4×4 switches, it can be seen that although increasing the degree of the switch makes only a small difference in the maximum bandwidth at the output of the network, it makes a very significant difference in the latency.

To increase bandwidth using an unbuffered network, the network may be duplicated or *dilated* [77]. Dilation of a network refers to increasing the number of links at each input and output of a switch. Multipath multistage networks, in which the connections are not only dilated, but wired in patterns for *maximal fan-out*, can show good performance with unbuffered switches [21]. Such designs pay for the increased performance by doubling the number of wires in the system; the utilization of the wires at later stages in the network remains low. Their great advantage is the relative simplicity of the basic switch design. This design simplicity may allow a network to be designed more quickly (and thus more inexpensively), or it may make it easier for the designers to pay attention to other issues, such as the fault tolerance and error correction features of the MIT Transit Network [34, 105].

2.1.2 k-input buffers, one per output port (Type A)

For this switch structure, a hardware buffer capable of accepting k inputs in one cycle is needed (see Figure 2.4). This is the type of switch that has received the most thorough analysis in the literature. Analyzing Type A switches with queues allowed to grow without bound ("infinite buffers"), Kruskal, Snir and Weiss [79] first found the generating function for the steady state distribution of waiting times in the first stage of the network. Using the same assumptions, Percus and Percus [116] determined the difference equations for the exact probability distribution of the number of waiting messages at the end of the n^{th} input cycle for both finite and infinite capacity output buffers, and solved the single queue or one dimensional distribution. The

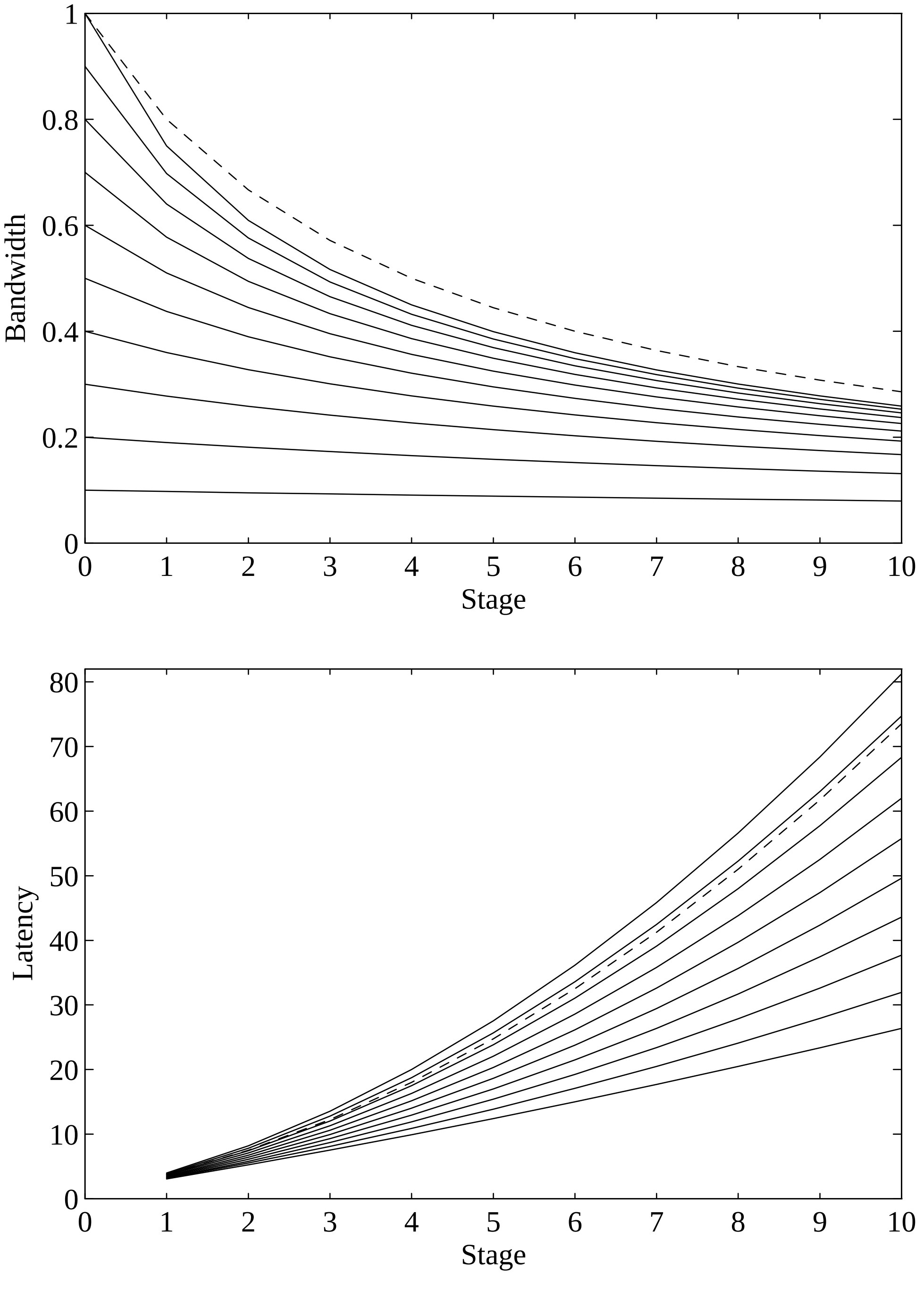


Figure 2.2: Unbuffered 2×2 switches, bandwidth from Equation 2.1, dashed line is upper bound on bandwidth from Equation 2.2. Latency computed as $(p/p_i)(2n+1)$, dashed line is lower bound from Equation 2.7.

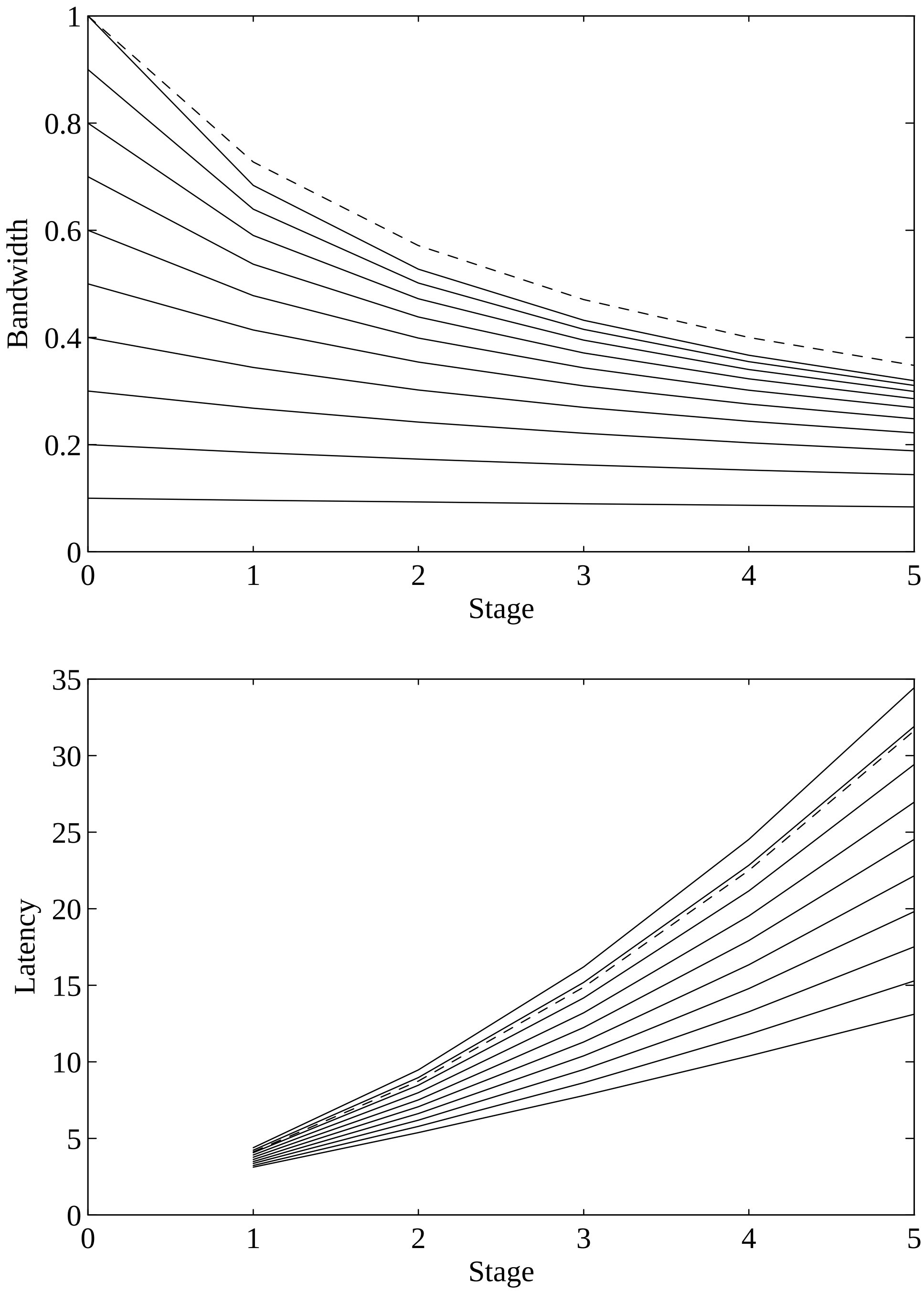


Figure 2.3: Unbuffered 4×4 switches, bandwidth from Equation 2.1, dashed line is upper bound on bandwidth from Equation 2.4. Latency computed as $(p/p_i)(2n+1)$, dashed line is upper bound from Equation 2.7.

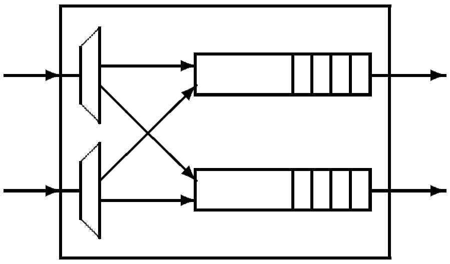


Figure 2.4: 2-input buffers, one per output port (Type A)

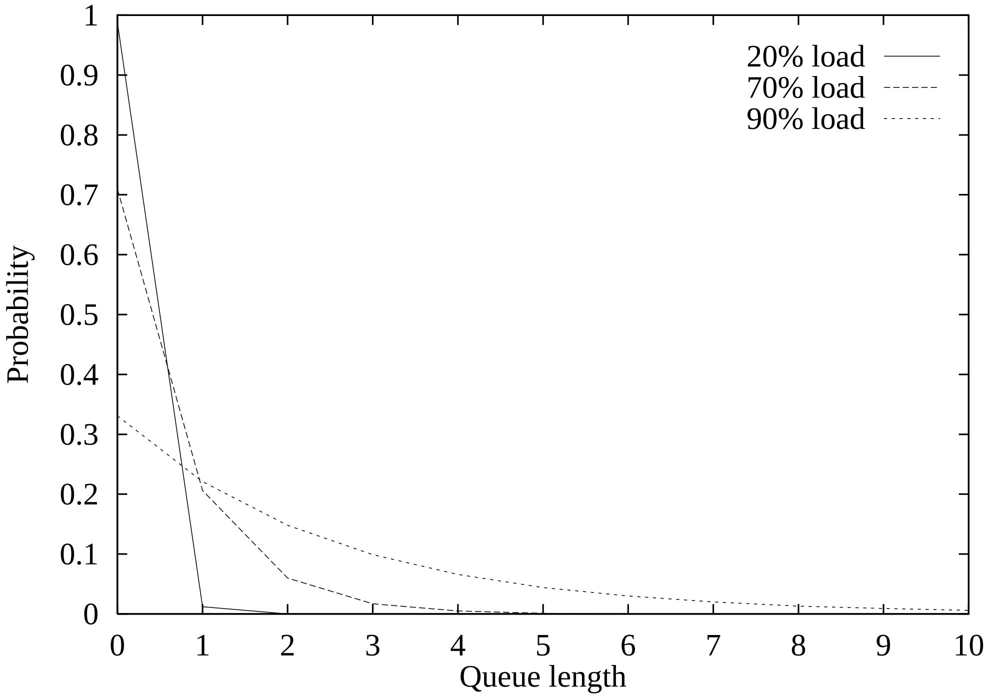


Figure 2.5: Distribution of queue lengths, Type A switch, unbounded buffer size.

latter paper also includes results on the output distribution, showing that it does not consist of independent identically distributed random variables.

In the Type A switch, messages are queued at each output port. If two messages destined for the same output port arrive in the same cycle, they are assumed to be queued in random order. The assumptions made in [79, 116] for analyzing buffers of infinite capacity are:

1. At the (first stage) input, arriving messages are treated as independent identically distributed Bernoulli variables. At each cycle, a message arrives with probability p and enters one of the two output queues with probability 1/2. Hence, the probability that j messages arrive in any cycle at a given queue is

$$f_j = \binom{2}{j} (\frac{p}{2})^j (1 - \frac{p}{2})^2 - j \text{ for } j = 0, 1, 2. \tag{2.9}$$

2. Every message requires one time unit (cycle) to be served.

3. If S_n is the number of waiting messages at the end of the n^{th} cycle, and a_n is the number of messages arriving at the beginning of the n^{th} cycle, then

$$\begin{aligned} S_0 &= 0 \text{ and} \\ S_n &= \max[0, S_n - 1 + a_n - 1] \text{ for } n \geq 1, \end{aligned}$$

i.e. a single message leaves during the n^{th} cycle if either at least one arrived or the queue was non-empty.

Let $p_n(j)$ represent $P(S_n = j)$, the probability of having j messages in the queue at the n^{th} cycle. Under the above assumptions

$$\begin{aligned} p_0(j) &= \delta_j, 0, \\ p_n(0) &= f_0 p_{n-1}(1) + (f_0 + f_1) p_{n-1}(0) \text{ for } n > 0, \text{ and} \\ p_n(j) &= f_0 p_{n-1}(j+1) + f_1 p_{n-1}(j) + f_2 p_{n-1}(j-1) \text{ for } n > 0, j \geq 1. \end{aligned}$$

Percus and Percus [116] used generating functions to find the steady state limiting distribution $p(j) \equiv \lim_{n\to\infty} p_n(j)$:

$$p(j) = \frac{(1-p)}{(1-p/2)^2} (\frac{p/2}{1-p/2})^{2j}. \tag{2.10}$$

The probability of a queue being empty is then

$$p(0) = (1-p)(1 - \frac{p}{2})^{-2}. \tag{2.11}$$

The mean queue length in steady state is

$$\lim_{n\to\infty} E(S_n) = p^2/4(1-p) \tag{2.12}$$

and the variance is

$$\lim_{n\to\infty} Var(S_n) = \frac{p^4}{16(1-p)^2} + \frac{p^2}{4(1-p)}. \tag{2.13}$$

The latter two results were first obtained by Kruskal, Snir and Weiss [79], who also developed approximate formulas for the mean and variance of Type A switches of degree greater than 2 (see section 2.4) and for messages containing multiple packets.

Figure 2.5 shows the probability distribution of queue lengths for input rates of 20 percent, 70 percent and 90 percent, assuming that the output is never blocked, from equation 2.10. At 20 percent input rate, the queue is empty most of the time; even at 70 percent it rarely has more than one item.

The difficulties of implementing Type A switches, which require k-input buffers, as opposed to Type B and C switches, which can be built with buffers having only one input, are discussed in Chapter 5. The performance of minimum-sized Type A switches is analyzed, along with minimum-sized Type B and Type C switches, in section 2.2. The performance of Type A switches is compared under uniform traffic to that of Type B switches in section 2.3; Chapter 4 contains simulation data for Type A and B switches under hot spot traffic.

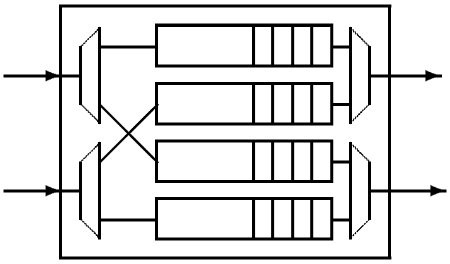


Figure 2.6: One-input buffers, k buffers per output port

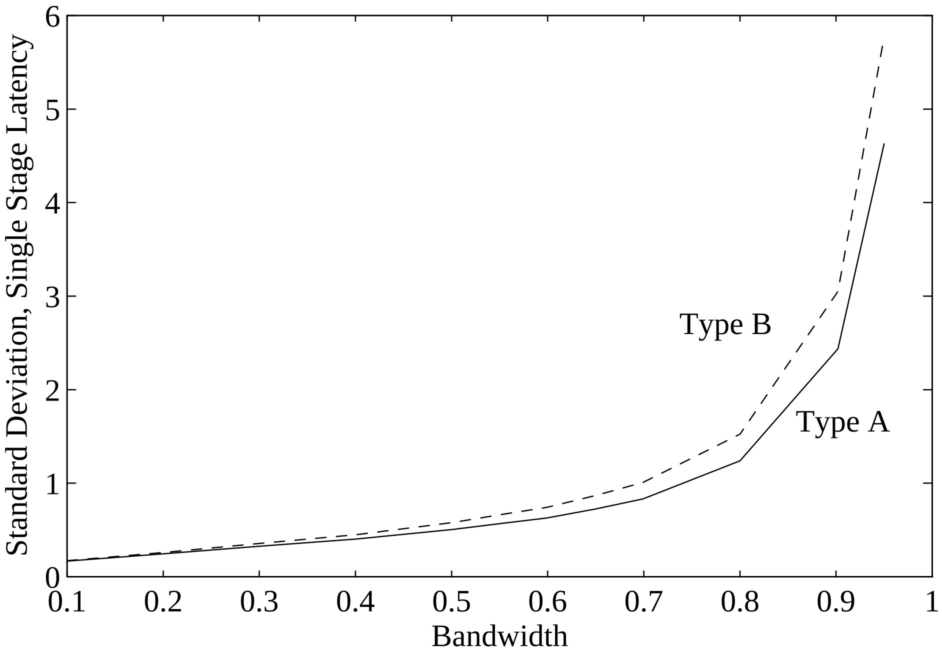


Figure 2.7: Standard deviation of the waiting time, single stage, Type A and Type B switches, unbounded buffer size. From simulation.

2.1.3 One-input buffers, k buffers per output port (Type B)

This configuration (see Figure 2.6) use multiple instances of a simpler type of buffer to approximate the performance of the k-input buffers discussed above. A packet leaves an output port whenever any of the k associated buffers has data; if more than one has data, arbitration must occur.

Kumar and Jump [80] called called this configuration "buffers within the switches." Their simulations of Type B switches showed better performance, especially for high loads, than "buffers between the switches," which correspond to the Type C switches of the next section. In [41], interconnection networks with Type B switches were called "Split Buffered MINS." Tamir and Frazier [132] simulated different implementations of Type B switches, which they call statically and dynamically allocated multiqueue buffers.

If we look at the k buffers associated with an output port as a single queueing system with a service rate of 1 per cycle, then the expected waiting time in the first stage is the same as that of the k-input buffer. In [114, 115], Percus and Dickey obtained the generating function of the $p_n(k, j)$, the probability of k messages in $q1$ and j messages in $q2$ at time n, where $q1$ and $q2$ are the two queues paired at an output. Using this generating function, it was shown that in steady state $p(0, 0)$ for Type B switches is the same as $p(0)$ for Type A switches, and the probability that a total of l messages are waiting in the 2 queues ($q1$ and $q2$),

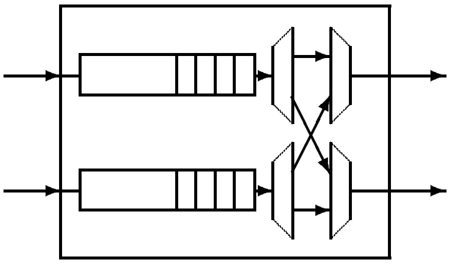


Figure 2.8: One-input buffers, one per input port (Type C).

was shown to be

$$\sum_{k=0}^{l} p(k, l-k) = (1-p)(1-\frac{p}{2})^{-2}\left(\frac{p^2}{4}(1-\frac{p}{2})^{-2}\right)^{l}. \tag{2.14}$$

Let $E(S)$ be the expected queue length in the steady state. Then

$$2E(S) = (1-p)(1-\frac{p}{2})^{-2}\sum_{l=0}^{\infty} l(\left(\frac{p^2}{4}(1-\frac{p}{2})^{-2}\right)^{l} = \frac{p^2}{4(1-p)}. \tag{2.15}$$

Notice that the expected queue length for a queue in the Type B switch is half that for a queue in a Type A switch, (i.e., the expected total number of queued items is the same for switches of Types A and B) and that the probability that both queues paired at an output port in a Type B switch are empty is the same as that for the single queue in a Type A switch. This corresponds to the intuitive perception that a Type B switch will output a message from a given port whenever a Type A switch does, though messages may be output in a different order, since the service discipline is no longer first come, first serve. Simulations of a single stage switch show that a Type B switch has a somewhat greater variance in the waiting time (see Figure 2.7).

2.1.4 One-input buffers, one per input port (Type C)

In this arrangement (see Figure 2.8), outputs of the buffers are multiplexed, and a buffer may be blocked on output by another buffer. Though the simplest type of buffered switch from the point of view of the hardware designer, this is the most difficult to analyze with conventional methods of queueing theory, due to the blocking which occurs even if "infinite buffers" are assumed.

A simple argument gives an upper bound on the bandwidth. Suppose that α is the probability that a buffer is not empty; then, under the assumption that the queue lengths of the different buffers in a switch are independent, and the destinations of items at the head of the queue are uncorrelated,

$$b = 1 - (1-\alpha/k)^k \tag{2.16}$$

is the average number of items exiting from each output port each cycle. Since α cannot be greater than one, the maximum bandwidth per port is bounded by $1-(1-1/k)^k$ At $k=2$, this is .75; as k gets large, this approaches $1-1/e \approx 0.63$.

The above analysis is marred by the assumptions that the queue lengths are independent, when in fact service at one buffer depends on the queue length at the others, and that α is independent of the state of the system on the previous cycle. Yen, Patel and Davidson [147], in the context of a complete crossbar network between PEs and MMs, with queues at the PE inputs to the network, surveyed possible improvements to this simple analysis. The conditional probability α for a given load p and system state defined by lengths of queues and output port destinations is difficult to determine, particularly when finite buffer effects are taken into account. Our own simulations show that bandwidth per port out of a $k \times k$ crossbar with unbounded buffers is actually less than $1-(1-1/k)^k$ for k greater than 2 (see Figure 2.9). This is consistent with results from the improved models in [147].

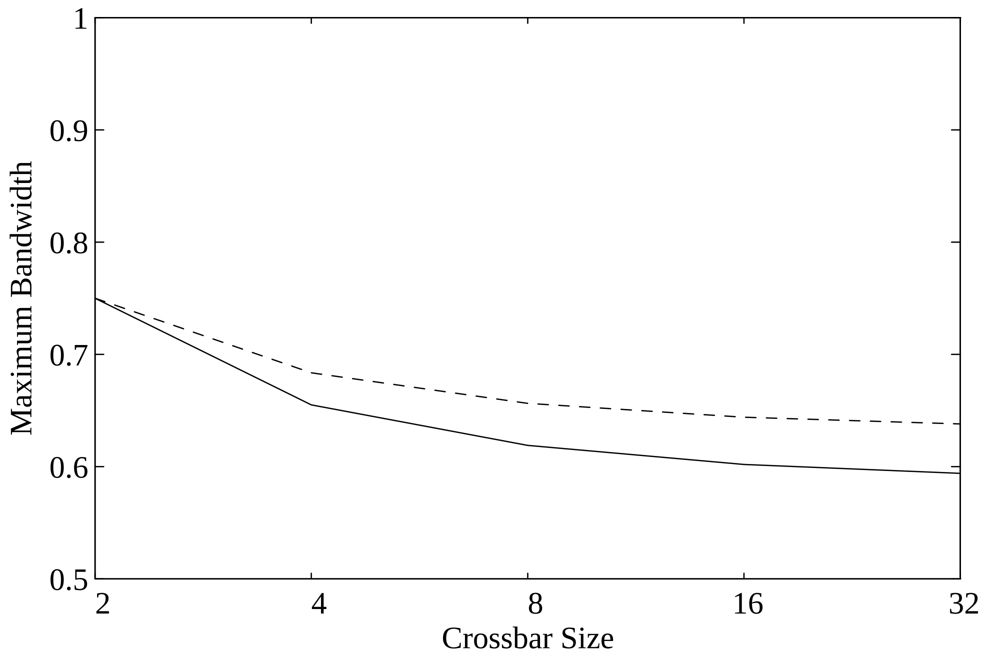


Figure 2.9: Maximum bandwidth of different size crossbars. Solid line from simulations, dashed line is $1-(1-1/k)^k$).

	$q2$		
$q1$	red	blue	0
red	$\frac{p^2}{4}$	$\frac{p^2}{4}$	$\frac{p}{2}(1-p)$
blue	$\frac{p^2}{4}$	$\frac{p^2}{4}$	$\frac{p}{2}(1-p)$
0	$\frac{p}{2}(1-p)$	$\frac{p}{2}(1-p)$	$(1-p)^2$

Table 2.1: Arrival probabilities for different output ports at the inputs to a 2×2 switch.

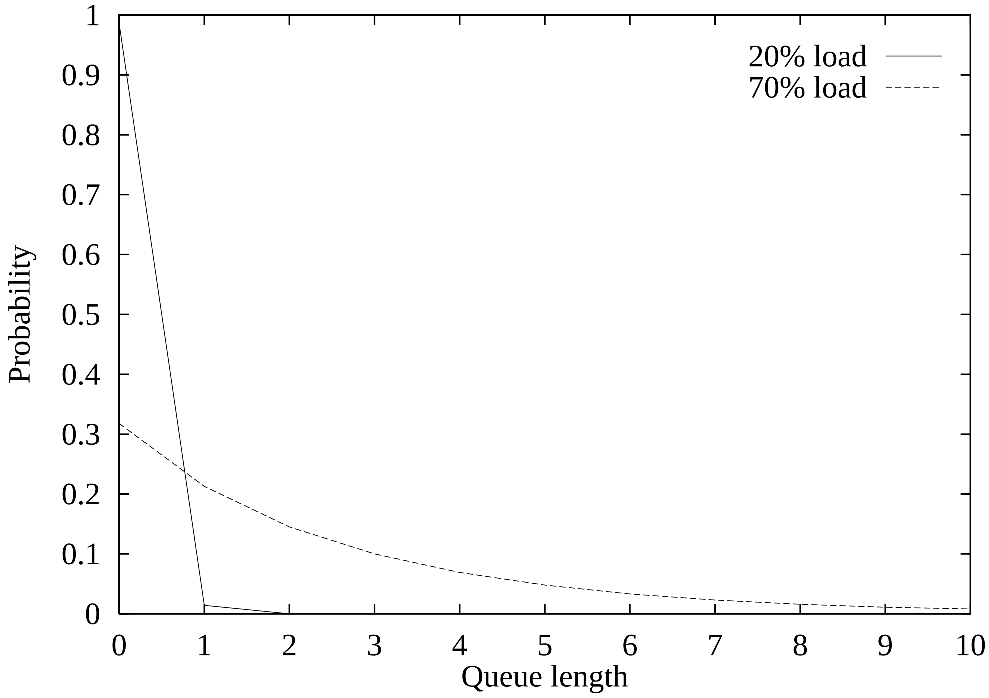


Figure 2.10: Distribution of queue lengths, Type C switch, unbounded buffer size.

For 2×2 switches, we have shown in [114, 115] that bandwidth is in fact limited to 75 percent, and that for loads under 75 percent, Type C switches have a steady state limiting distribution. That is, for 2×2 switches, the $1-(1-1/2)^2$ limitation is a tight upper bound. In the analysis of 2×2 Type C switches given in [114, 115] and summarized below, $q1$ and $q2$ refer to the two queues in the switch, each with its own input port and each connected to both output ports. Since each message is predestined for a given port by its address, one may think of them as colored red and $blue$, with one output port serving red messages and the other blue messages. Intuitively, if the queues were never empty and all four possible patterns of the two colors were equally probable at each cycle, one would expect to output two messages half of the time, and only one message the other half, for a maximum expected output rate per port of 75 percent.

The proof of the 75 percent bandwidth limitation follows from a case by case analysis. Assume that the messages arrive independently at the 2 queues with probability p at each cycle (i.e. $P(red) = P(blue) = p/2$). The service discipline is first come, first served, but when two identically colored items appear in front of the 2 queues at the same cycle one item is blocked (i.e, a message is selected for service from one of the two queues with equal probability).

The nine possibilities for arrivals to the inputs at the first cycle (or any cycle when both queues are empty) are shown in Table 2.1, where 0 indicates no item arrives. The number of items served at the first cycle will be 2 for the cases (red, blue) and (blue,red), 0 for the case (0,0) and 1 otherwise. The number of items left in the system after the first cycle will be 1 for the cases (red, red) and (blue, blue) and 0 otherwise.

In a similar way, the effect of the nine different arrival possibilities on the state of the system can be analyzed for the case when only one queue is empty and the case when both queues are not empty. If difference equations that depend only on the number of items in each queue, and not on the colors of all items, can be constructed, the output probabilities for a given cycle must be independent of the exact color pattern in the preceding cycle. For the 2×2 switch, it is necessary to show that if items from both input streams appear at the output on the same cycle, they are equally likely to be of the same color or of different colors. This can be shown by analyzing the different cases.

Consider the case when both queues are non-empty and exactly one queue, $q1$, was non-empty on the

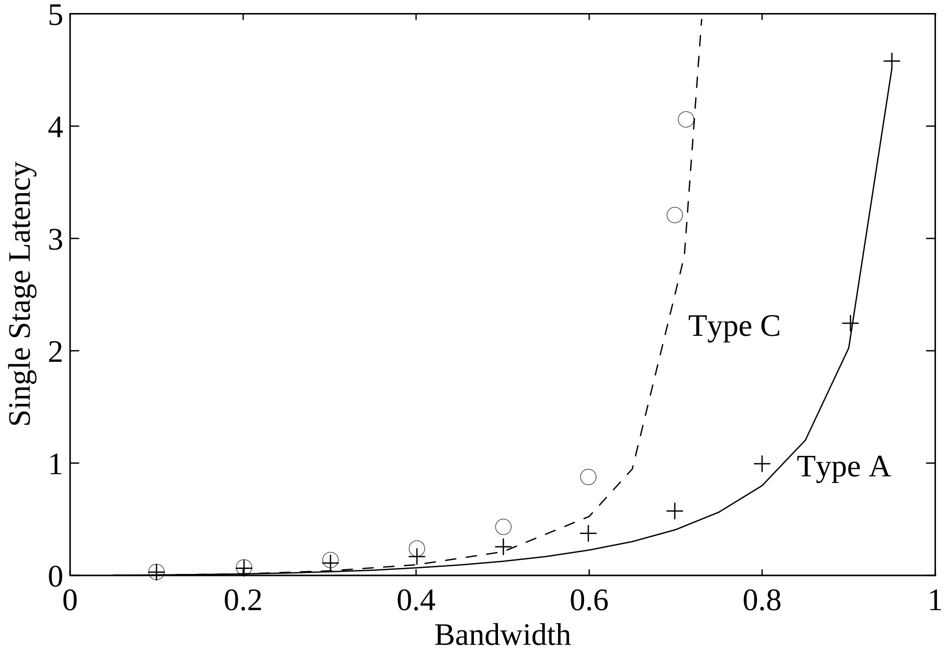


Figure 2.11: Latency comparison for Type A and C switches, unbounded buffer size, single stage. Type B values are identical to Type A.

preceding cycle. Suppose for definiteness that the item at the front of $q1$ was *red*. Then both queues will be non-empty only if a *red* item arriving at $q2$ was blocked by the item leaving from the front of $q1$ and another item either existed in or arrived at $q1$. The items remaining at the front of the queues will both be the same color if the next item in $q1$ was *red*, and they will be different colors if the next item in $q1$ was *blue*; the item is equally likely to be of either color because the input streams to the two queues are memoryless and independent. The other cases can be analyzed similarly.

Note that this analysis does not extend easily to switches of size greater than 2, where contention by more than two input queues for the same output on a particular cycle implies that some contention will continue to exist on the next cycle, and thus a description of system state must include not just the number of messages at each output, but their colors as well.

In [114, 115] we also used the case by case analysis to construct the difference equations for $p_n(k, j)$, the probability that there are k messages in the first queue and j messages in the second. The queue length distribution values in Figure 2.10 were found be evaluating these equations until convergence; the values were also checked by simulation.

From these queue length distribution values, we also computed the latency, which is compared in Figure 2.11, for a single stage with unbounded buffers, to the latency of Type A and Type B switches.

Figure 2.12 shows a design similar to the Type C switch, but with the two single-input queues preceded by instead of followed by a crossbar. Such a design, often with a buffer at the input, has frequently been proposed (see, e.g., [29, 123, 144]. If data movement from chip to chip takes only a single cycle, this design shows the same behavior as Type C switches, as can be seen by redrawing the network so that the queues are part of the same switch as the crossbar (see the dotted box in Figure 2.12. The buffer at the input increases the effective size of the queues. However, if the multiplexing, routing and queueing logic is fast enough to input two messages to a queue in series in the time it takes for one message to pass from chip to chip, the switch will function as in implementation of a Type A switch.

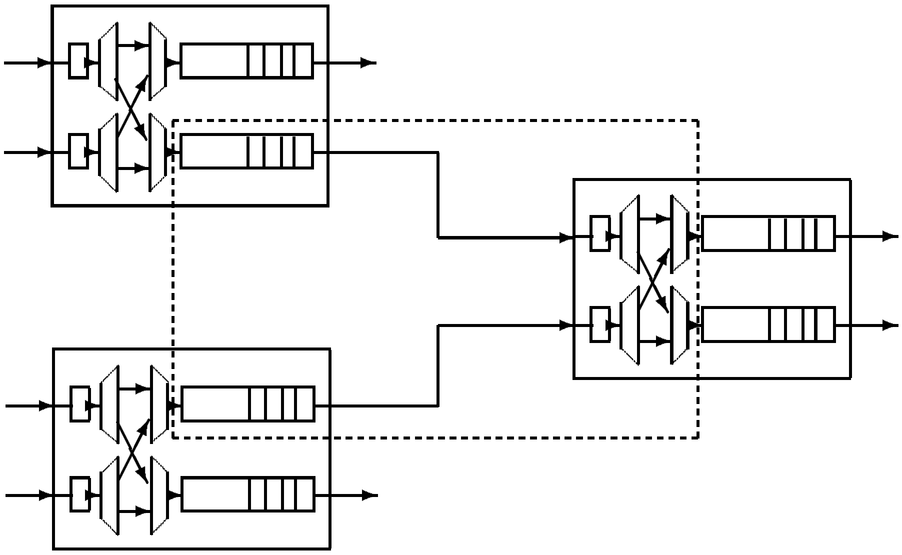


Figure 2.12: Variant switch architecture, may perform as Type A or Type C.

2.2 Switches with finite buffers

Though networks composed of switches with unbounded buffer size may be easier to analyze, since one need not be concerned with blocking by the next stage, when actually building a switch one wishes to make its buffer size as small as compatible with good performance. For Type A switches, it is possible to carry out a complete analysis for a single stage switch with buffers of any finite size, as was done by Percus and Percus in [116]. For Type B and C switches, Percus and Dickey [114, 115] analyzed single stage switches with the smallest possible buffers, finding the maximum output bandwidth as well as the queue length distribution.

When analyzing Type A switches for queues with finite capacity N, the third assumption in section 2.1.2 must be changed. Percus and Percus [116] investigated two models:

$$\begin{aligned}\text{Model I} : S_n &= \min[N, \max(0, S_{n-1} + a_n - 1)] \\ \text{Model II} : S_n &= \begin{cases} \max[0, S_n - 1 + a_n - 1] & \text{if } S_{n-1} <= N - 1 \\ N - 1 & \text{if } S_{n-1} = N \end{cases}\end{aligned}$$

The first model makes the assumption that messages can continue to be offered at the input to a queue on every cycle, even after it is full, because departures may allow the arriving message to be accepted. This corresponds to a handshaking protocol, in which the source of the message finds out after it has been sent whether or not it was accepted. The second model corresponds more closely to the flow control protocol described in section 4.1.5, in which the receiver must guarantee that a message can be accepted before it is sent.

For Model I, the steady state distribution was found to be

$$p(j) = \frac{1-\beta}{1-\beta^{N+1}}\beta^j, \text{ for } j = 0, 1, \ldots N, \text{ where } \beta = \left(\frac{p/2}{1-p/2}\right)^2, \tag{2.17}$$

and the average queue length

$$E[\lim_{n\to\infty} S_n] = \sum_{j=0}^{N} jp(j) = \frac{\beta}{1-\beta}[1 - \frac{(N+1)(1-\beta)\beta^N}{1-\beta^{N+1}}], \tag{2.18}$$

with a more complicated expression for $Var[\lim_{n\to\infty} S_n]$. The steady state distribution and mean for Model II are close in value to these, but yet more complicated and are also found in [116].

For comparison with minimum size hardware implementations of the other two types of switches, we are interested in a Type A switch with N = 2. This is the smallest value for N which allows both messages arriving at a cycle to be accepted if the queue is not empty on the preceding cycle, and is comparable in hardware resources to a Type B switch with 4 queues each of size one. It is assumed that a message is lost if two arrive at a full queue on a single cycle.

Percus and Dickey [115] found the expected output per cycle by introducing a tagging variable z in the difference equations of [116] for every term in the equation that results in an output in that cycle. For $N = 2$ the equations take the form:

$$\begin{aligned}
p_0(0) &= 1, \\
p_n(0) &= f_0 z p_{n-1}(1) + (f_0 + f_1 z) p_{n-1}(0), \\
p_n(1) &= f_0 z p_{n-1}(2) + f_1 z p_{n-1}(1) + f_2 z p_{n-1}(0), \\
p_n(2) &= f_2 z p_{n-1}(2) + f_1 z p_{n-1}(2) + f_2 z p_{n-1}(1),
\end{aligned}$$

since the only state which does not produce an output is when the queue is empty and no messages arrive. Using the generating functions

$$F_n(x, z) = \sum_{k=0}^{2} p_n(k) x^k, \tag{2.19}$$

and

$$F(\lambda, x, z) = \sum_{n=0}^{\infty} \lambda^n F_n(x, z). \tag{2.20}$$

Then the expected output per cycle, $E_p(O)$, in steady state will be

$$\begin{aligned}
E_p(O) &= \lim_{\lambda \to 1} (1-\lambda)^2 \frac{\partial}{\partial z} F(\lambda, x, z) \Big|_{\substack{z=1 \\ x=1}} \\
&= p - \frac{p^6}{64 - 128p + 112p^2 - 48p^3 + 12p^4}
\end{aligned}$$

with $\max_p E_p(O) = 11/12$.

Note that the expected output per cycle is equal to $1 - f_0 p(0)$, that is, the probability that either a message arrives or the queue was not empty at the start of the cycle.

Although we have not found a general formula for the queue length probability of queues in Type B switches with finite buffers, the result for queue size 1 was found in [115] to be

$$\begin{aligned}
p(0,0) &= \frac{32 - 48p + 24p^2 - 4p^3}{32 - 48p + 32p^2 - 8p^3 + p^4}, \\
p(0,1) = p(1,0) &= \frac{4p^2 - 2p^3}{32 - 48p + 32p^2 - 8p^3 + p^4}, \\
p(1,1) &= \frac{p^4}{32 - 48p + 32p^2 - 8p^3 + p^4}.
\end{aligned}$$

Notice that for $p = 1, p(1,1) = 1/9$; $p(0,1) = p(1,0) = 2/9$; $p(0,0) = 4/9$ and for $p = 0, p(0,0) = 1$. Also, the one-dimensional probability

$$p(0) = p(0,0) + p(0,1) = \frac{32 - 48p + 28p^2 - 6p^3}{32 - 48p + 32p^2 - 8p^3 + p^4} \tag{2.21}$$

is the probability that $q1(q2)$ is empty, and

$$p(1) = p(1,0) + p(1,1) = \frac{4p^2 - 2p^3 + p^4}{32 - 48p + 32p^2 - 8p^3 + p^4} \tag{2.22}$$

is the probability that $q1(q2)$ is not empty, which, since the queue capacity is 1, is also the mean queue length $E(S)$.

Notice that for $p = 1$, $p(1) = 1/3$, so that the average length of $q1$ is less than 1/3 if $p < 1$.

The expected output per cycle of the Type B switch with queues of size 1 was also found in [115] to be

$$E_p(O) = p\left(1 - \frac{p^3}{p^4 - 8p^3 + 32p^2 - 48p + 32}\right) \tag{2.23}$$

Note that

$$E_p(O|p = 1) = 8/9 \tag{2.24}$$

gives the maximum bandwidth that can be obtained per output port for this switch structure with queue size 1. This is lower than the maximum bandwidth of 11/12 which can be obtained from the Type A switch with queues of size 2.

For Type C switches we were not able to solve the recurrence relations for switches with infinite buffers or for finite buffers in general, but we were able in [115] to obtain a solution for queues of capacity one. The conditions under which a message is lost depend for Type C switches on the color of the items in the queue, as well as their number. For example, suppose $q1$ has a *red* item and $q2$ is empty. If a *red* item arrives at $q2$, it has a 50 percent chance of blocking the item in $q1$, causing the *red* item in $q1$ to remain for another cycle and any message of either color arriving at $q1$ to be lost. Considering each such case for all nine input possibilities and all nine initial queue configurations, and consolidating symmetric equations, we get the following difference equations:

$$p_0(0, 0) = 1. \tag{2.25}$$

For $n \geq 1$,

$$\begin{aligned}
p_n(0,0) &= (1 - \frac{p^2}{2})p_{n-1}(0,0) + (1-p)(1-\frac{p}{2})[p_{n-1}(0,1) + p_{n-1}(1,0)] \\
&\quad + \frac{(1-p)^2}{2}p_{n-1}(1,1), \\
p_n(0,1) &= \frac{p^2}{4}p_{n-1}(0,0) + (\frac{5p}{4} - \frac{p^2}{2})p_{n-1}(0,1) \\
&\quad + \frac{p}{4}(1-p)p_{n-1}(1,0) + \frac{(1+p-2p^2)}{4}p_{n-1}(1,1), \\
p_n(1,0) &= \frac{p^2}{4}p_{n-1}(0,0) + \frac{p}{4}(1-p)p_{n-1}(0,1) \\
&\quad + (\frac{5p}{4} - \frac{p^2}{2})p_{n-1}(1,0) + \frac{(1+p-2p^2)}{4}p_{n-1}(1,1), \\
p_n(1,1) &= \frac{p^2}{4}p_{n-1}(0,1) + \frac{p^2}{4}p_{n-1}(1,0) + \frac{p}{2}(1+p)p_{n-1}(1,1).
\end{aligned}$$

The generating function

$$\begin{aligned}
F_0(x,y) &= 1, \\
F_n(x,y) &= \sum_{k=0}^{1}\sum_{j=0}^{1} p_n(k,j)x^k y^j, \text{for } n \geq 1,
\end{aligned}$$

was then used to define

$$\begin{aligned}
F(\lambda, x, y) &= \sum_{n=0}^{\infty} \lambda^n F_n(x,y) \\
&= (1, x, y, xy)(I - \lambda Q)^{-1} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix},
\end{aligned}$$

where
$$Q = \begin{pmatrix} 1-p^2/2 & (1-p)(1-p/2) & (1-p)(1-p/2) & (1-p)^2/2 \\ p^2/4 & 5p/4-p^2/2 & p/4(1-p) & \frac{(1+p-2p^2)}{4} \\ p^2/4 & p/4(1-p) & 5p/4-p^2/2 & \frac{(1+p-2p^2)}{4} \\ 0 & p^2/4 & p^2/4 & \frac{p(1+p)}{2} \end{pmatrix}.$$

Then $F(x,y)$, the steady state generating function of the limiting distribution, is

$$F(x,y) = \lim_{\lambda\to 1}(1-\lambda)F(\lambda,x,y) = \sum_{i=0}^{1}\sum_{j=0}^{1} p(i,j)x^i y^j \tag{2.26}$$

and we get, after considerable work,

$$\begin{aligned} p(0,0) &= \frac{8-16p+7p^2+2p^3-p^4}{8-16p+11p^2-2p^4}, \\ p(0,1) &= p(1,0) = \frac{2p^2-p^3-p^4}{8-16p+11p^2-2p^4}, \text{ and} \\ p(1,1) &= \frac{p^4}{8-16p+11p^2-2p^4}. \end{aligned}$$

Since the expected value of the sum of the queue lengths of the two queues paired at an output is the same as the sum of the expected values,

$$2E(S) = p(0,1)+p(1,0)+2p(1,1) \tag{2.27}$$

hence the average queue length is

$$E(S) = \frac{2p^2-p^3}{8-16p+11p^2-2p^4} \tag{2.28}$$

and the probability that one queue is empty is $p(0,0)+p(0,1) = 1-E(S)$.

To find the expected output per cycle, $E_p(O)$, from both queues in steady state, we considered the three cases: both queues empty, one queue empty and neither queue empty. By considering the nine different input possibilities, we found that

$$E_p(O|(0,0)) = \frac{2p-p^2}{2} \tag{2.29}$$

and

$$E_p(O|(0,1)) = E(O|(1,0)) = 1+p/2. \tag{2.30}$$

When both queues are non-empty, one item is output when the items at the front of each queue are the same color, and two are output when they are different colors. Since each case is equally likely, as discussed in the section 2.1.4,

$$E_p(O|(1,1)) = 3/2. \tag{2.31}$$

Then

$$\begin{aligned} E_p(O) &= (\frac{2p-p^2}{2})p(0,0) + (1+\frac{p}{2})(p(0,1)+p(1,0)) + \frac{3}{2}p(1,1) \\ &= \frac{3p}{2} + \frac{p}{2}(1-p)\frac{1-p+\frac{3}{8}p^2+\frac{1}{8}p^3-\frac{1}{8}p^4}{1-2p+\frac{11}{8}p^2-\frac{1}{4}p^4}, \end{aligned}$$

with

$$\max_p E_p(O) = \frac{3}{2}, \tag{2.32}$$

so that the average output per queue (or output port) is less than 3/4. It is interesting that the bandwidth limitation for a single stage is the same for buffers of size 1 as for infinite buffers.

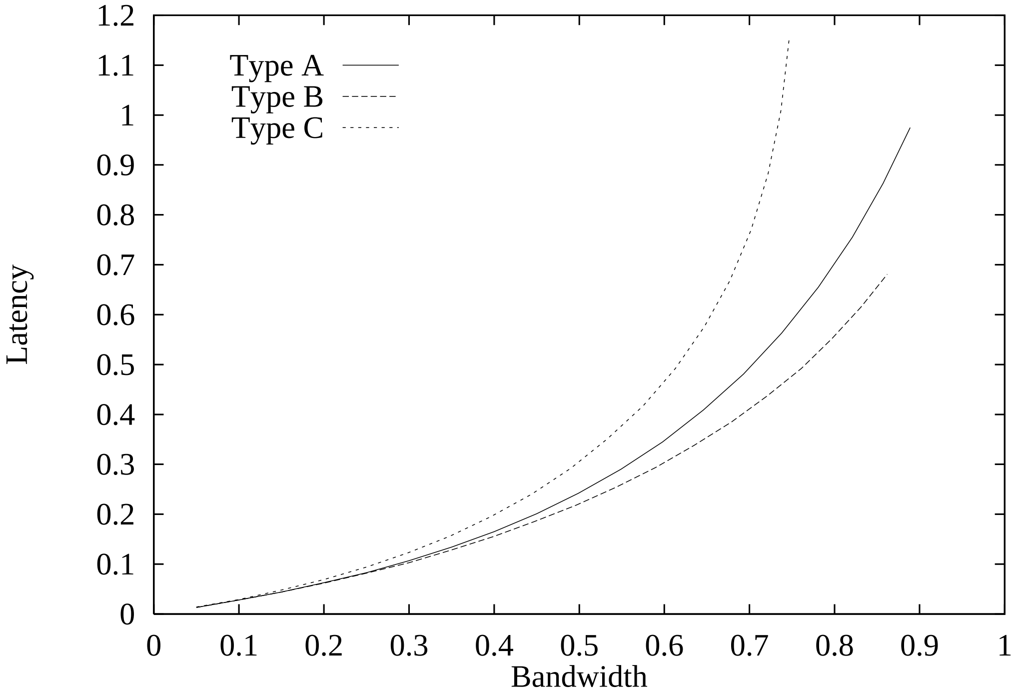


Figure 2.13: Latency for queues of size 2 in Type A switches, and queues of size 1 in Type C and Type B switches, single stage.

Figure 2.13 shows the single stage latency for minimum size Type A, Type B and Type C switches, as a function of the actual bandwidth achieved. The rightmost point on a curve represents the bandwidth and latency achieved at maximum offered load (100 percent). The most interesting and unexpected result is that, while Type A switches achieve a higher bandwidth than Type B switches, the latency of a message in a minimum size Type B switch is less than that in a Type A switch, for the same achieved bandwidth. This reduction in latency is somewhat misleading, since we are not accounting for any difference in waiting time at the input before a message is accepted by a switch, but does indicate that a better overall performance within the network may be sometimes be achieved by more frequent blocking of the inputs.

In [114, 115], only a single stage of a network was analyzed. Theimer, Rathgeb and Huber [135] and Yoon, Lee and Liu [150] have both done approximate iterative Markov chain analyses of finite-buffered multistage Type C networks. Both analyses use the optimistic value of b from equation 2.16 as the average number of messages passing through the output crossbar per cycle. Neither considers possible correlations within the output stream, as was done for Type A switches in Percus and Percus [116].

In Yoon, Lee and Liu's paper [150], the computation of arrival, departure and queue length probabilities from $k \times k$ switches at each stage in the network are much simplfied and assume that the state of the buffers within a switch as well as the state of buffers in different stages are all independent. When they checked their results by simulation (for a six-stage network with 2×2 switches), they found that their results were optimistic for heavy loads and for small queue sizes. For buffers of size 1 in a six-stage netowrk, their analysis predicted a throughput of 50 percent, but simulations showed between 35 and 40 percent. No comparison was made of the latency from simulation with the predicted latency. Even though the results are approximate, their analytical comparison of crossbar sizes for a 12-stage network is interesting: at a queue size of 4 or greater, 2×2 switches have higher throughput than any other option, including a full crossbar.

Theimer, Rathgeb and Huber [135] first presented a simplified analysis for $k \times k$ switches similar to that in [150] and then described a "refined analysis" (for 2×2 switches with buffers of size 1) that considers the mutual dependence of queue length probabilities within a switch and takes some of the dependence between successive stages into account. These refinements improved the match between simulation and analysis, though throughput was still overestimated and latency underestimated compared to simulations. For a six-stage network, their simulations showed a maximum bandwidth of between 35 and 40 percent, and their analytical prediciton was only a little over 40 percent. At 60 percent offered load (which from their graphs was approximately the lowest load to give the maximum bandwidth), the latency per stage was simulated at around 9 cycles, while the analytical predictions were around 8.5. The throughput predictions are similar to that in [150], but the latency values are much higher.

2.3 Type A and Type B multistage networks

To summarize the preceding section, unbuffered switches cannot be used to construct a network with bandwidth linear in the number of PEs. Type C switches have scalable bandwidth, but it is limited to 75 percent of the wire bandwidth available at the output of a switch. Accordingly, we have directed our efforts to the analysis and simulation of Type A and Type B switches, especially the easier-to-implement Type B.

Results in this section are from simulations of entire networks with single packet messages. For these simulations we have used a simulator called *molasses*,[1] written by Jan Edler, that has been designed to be used for investigating architectural variations of the NYU Ultracomputer. The network simulator is part of an overall simulation environment that is planned to include an accurate but efficient PE simulator that will allow program segments to be simulated.

Currently molasses simulates $k \times k$ switches composed of output queues. The number of inputs per queue may be specified, and this determines the number of queues per switch. Type A, Type B and hybrid schemes for greater than 2×2 switches may be simulated, but a blocking crossbar at the output for simulating Type C switches is not supported. A variety of input patterns and combining options are shown in Chapters 4 and 5. Executable versions of molasses may be customized for a set of options, so that the simulation executes more efficiently. Results for some of these combining switch variations are shown in Chapters 4 and 5.

Unlike some network simulators (see, e.g [87, 100, 119]), which model the processors as an infinite source of message requests and allow an indefinite number of outstanding requests, molasses has specified limits both for the number of pending requests which are waiting to enter the network and for the number of outstanding requests which have entered the network but have not yet received a response from memory. In most of the simulations reported here the number of pending requests has been set to 0, corresponding to a model of a processor which does not queue requests, but instead issues them with some probability whenever possible.

By *offered load* we mean the probability that a message will be presented to the network on a cycle at which the network interface is able to begin transmission of a message. For a particular offered load, the processors actually achieve an average rate of messages issued, after the system reaches a steady state in which the rate of messages issued by the processor is equal to the rate being serviced at the memory; we refer to this average rate as the *bandwidth* per PE. Round trip *latency* represents only waiting and transmission time in the network and at the memory. Delay at memory can be set to be more than one switch cycle, but was simulated as a single switch cycle in this chapter, unless specifically noted otherwise. Thus the minimum latency for a system of size N with single packet messages is $2\log(N) + 1$. If processors are blocked, no new messages are generated, and no waiting time is added to account for the time the processor remains blocked. The effect of blocking is reflected in the reduction of bandwidth per PE.

The simulation results in Figure 2.14 show the bandwidth and latency values at 100 percent offered load for Type A and Type B queues of comparable sizes. Single packet messages were used, and 128 outstanding requests were allowed at each PE, to minimize the reduction in bandwidth due to waiting for requests to return. Queue sizes of 1, 2, 4, 6, 8, 10 were simulated for Type B, with double those values for Type A, representing switches with equal storage. Type A switches are shown with solid lines, Type B dashed, as is done throughout this chapter whenever Type A and Type B results are shown on the same graph. The results are similar to the results of the analysis for single stages, but there are some differences. In Figure 2.14, the bandwidth is higher for minimum-sized Type B switches than for Type A switches. As discussed by Liu in [100], this is a result of the Model II protocol used in the simulation. In the analysis of section

[1] A reference to its blinding speed. Actually, it is very fast, but the result of having a faster simulator is that larger simulations are run, which do not finish quickly.

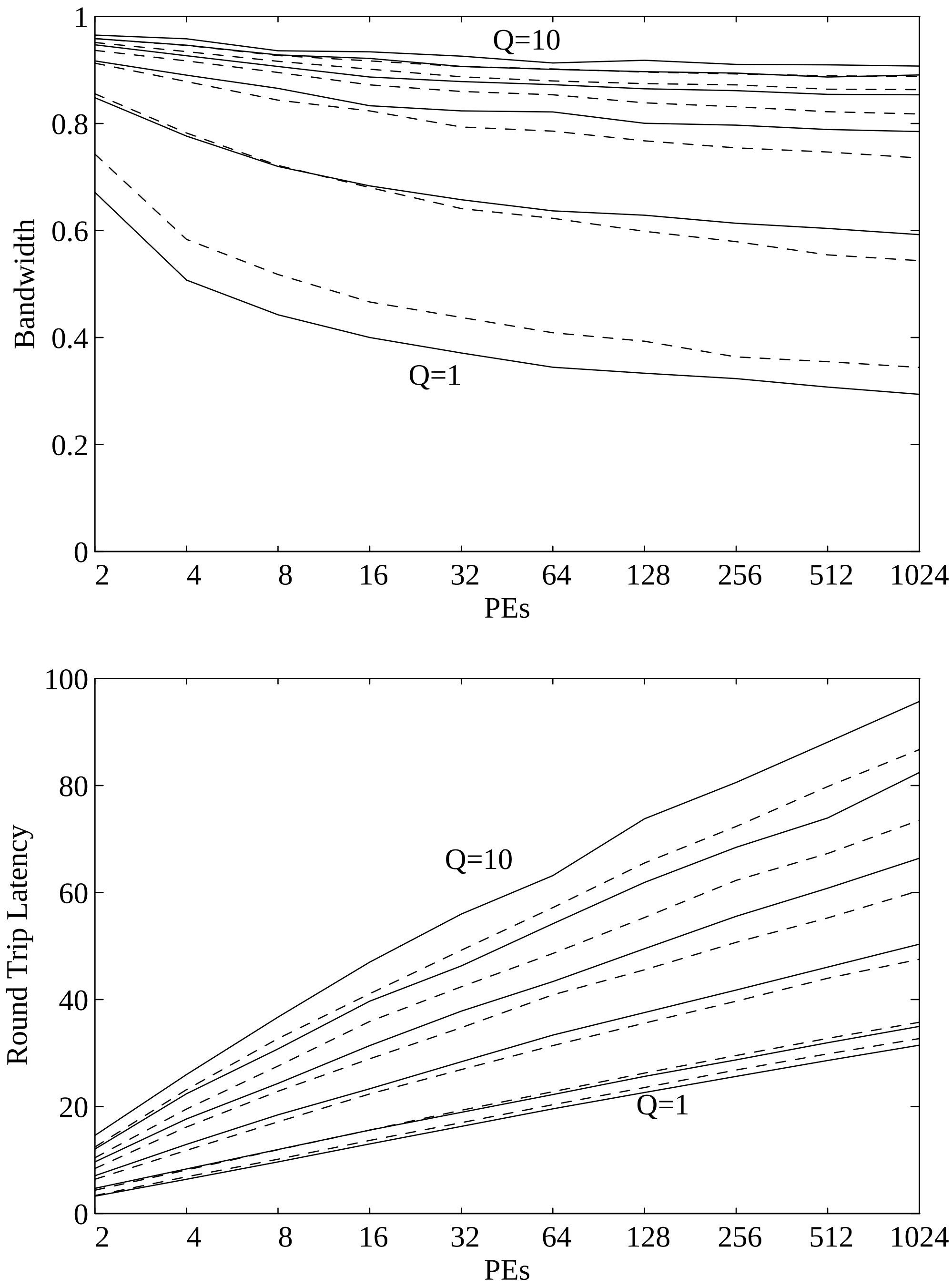


Figure 2.14: Bandwidth and latency values for Type A (solid) and Type B (dashed) networks with queues of comparable sizes. From simulation.

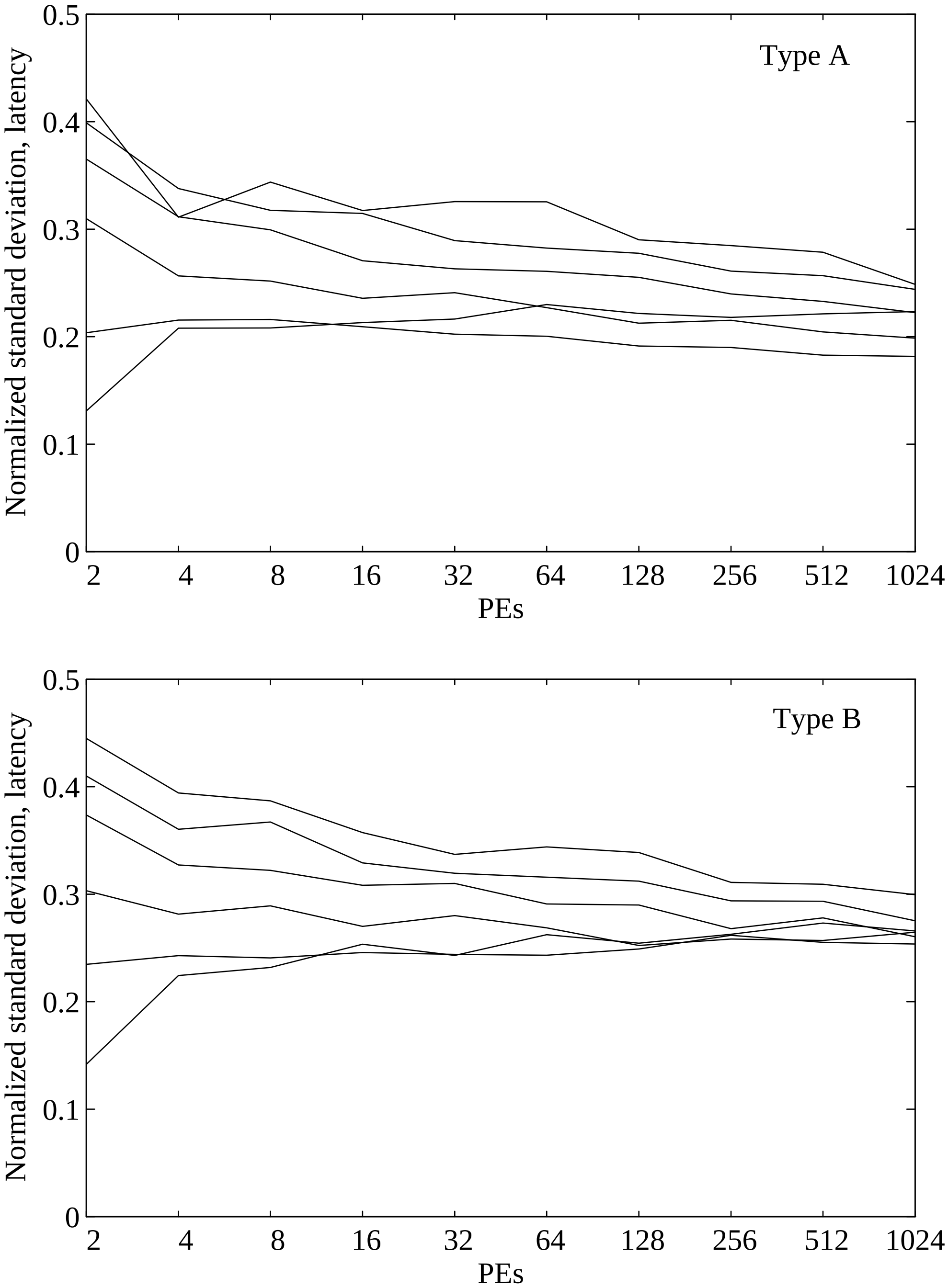


Figure 2.15: Normalized standard deviation of network latency for systems from 2 to 1024 PEs. From simulation.

2.13, switches accepted as many messages as they had room for. In the simulation, a switch will not accept any messages from a port on a cycle unless it has room for the maximum number of messages which may arrive. So a Type A switch of size 2 will block both input ports if either queue has room for only one more message, while a Type B switch may have one port blocked, but the other free.

By comparison, the optimistic analysis of Type C switches in [150] for queues of size 6, with a ten-stage network, showed throughput around 70 percent, and one-way latency between 35 and 40 cycles. An equivalent throughput is achieved by Type B switches of size 4 (for a total storage of 8 items per output), with a one-way latency of only around 20 cycles.

Figure 2.15 shows the normalized standard deviation (standard deviation divided by the mean) of the network latency for Type A and Type B switches, for the same set of comparable queue sizes. As in the single stage simulations, the variance of Type B switches is somewhat larger.

Figure 2.16 shows the blocking probability stage by stage through the network for a system with 1024 PEs, with large queues (10 packets for Type B, 20 for Type A), and and minimum size queues (1 packet for Type B, 2 for Type A), at 100 percent offered load. For minimum size queues, though some additional blocking occurs at later stages in the network and on the return path from memory, almost all of the bandwidth reduction occurs in the first half of the forward path network. For large queues, only the first few stages show any appreciable blocking, indicating the size of queues needed for good performance is relatively independent of network size.

Figure 2.17 shows latency as a function of the achieved bandwidth, for large queues and miminum size queues, for a system of 1024 PEs, as the offered load is varied from 10 percent to 90 percent. For minimum size queues, Type B switches not only achieve a higher bandwidth at maximum offered load, but also show less latency for the same achieved bandwidth. For large queues, Type A switches show a slight advantage at the heaviest loads.

Figure 2.18 shows the effect on bandwidth and latency of varying the number of outstanding requests each PE may have, with the queue sizes kept large (10 packets for Type B queues, 20 packets for Type A queues). The number of outstanding requests ranges from 1 to 128; some are marked on the graphs. For a small numbers of outstanding messages, the possible bandwidth is greatly reduced, and latency remains close to the minimum value. With small numbers of outstanding messages, a much smaller queue size could be used without affecting performance. The behavior of Type A and Type B switches is virtually identical, except for the highest number of outstanding requests, which allow a higher rate of messages to enter the network.

Figures 2.19 and 2.20 show the relative effects of the two ways that messages can be prevented from entering the network. The solid line shows the probability that a PE is blocked from issuing a message by the first stage of a network. The dashed line shows the probability that a PE cannot issue a message because the maximum number of outstanding requests has been reached. If the delay through the network and memory were some constant δ, and the maximum number of outstanding messages were X, the maximum latency-limited bandwidth at maximum offered load would be X/δ. The dotted line shows this predicted blocking. Allowing more outstanding requests increases the rate at which messages can be entered into the network, which in turn increases the round trip latency due to blocking within the network. The difference between the dashed line and the dotted line represents the effect of the extra blocking.

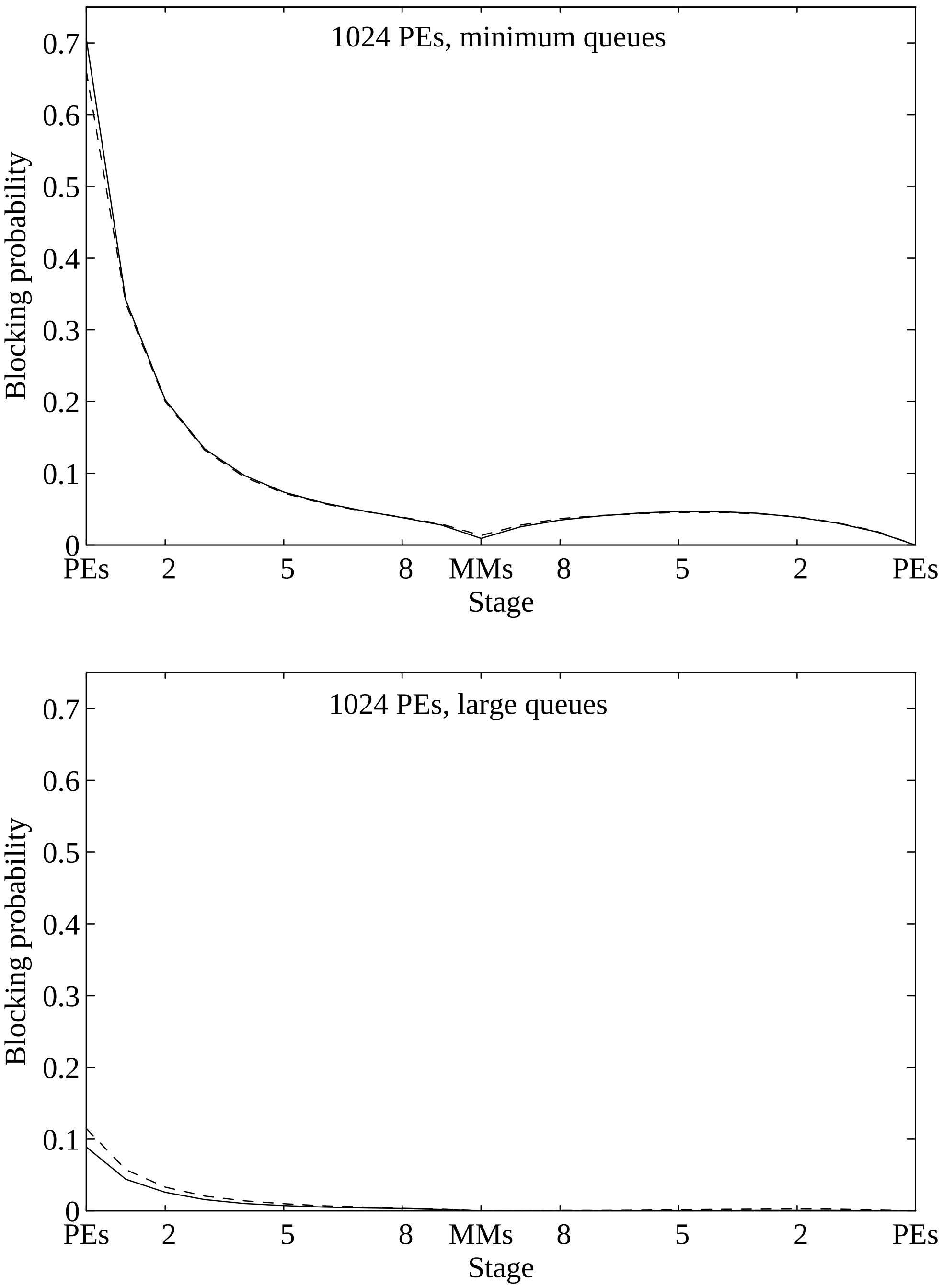


Figure 2.16: Blocking probability, 1024 PEs, small (Type A 2; Type B 1) and large (Type A 20, Type B 10) queues. Type A solid, Type B dashed. From simulation.

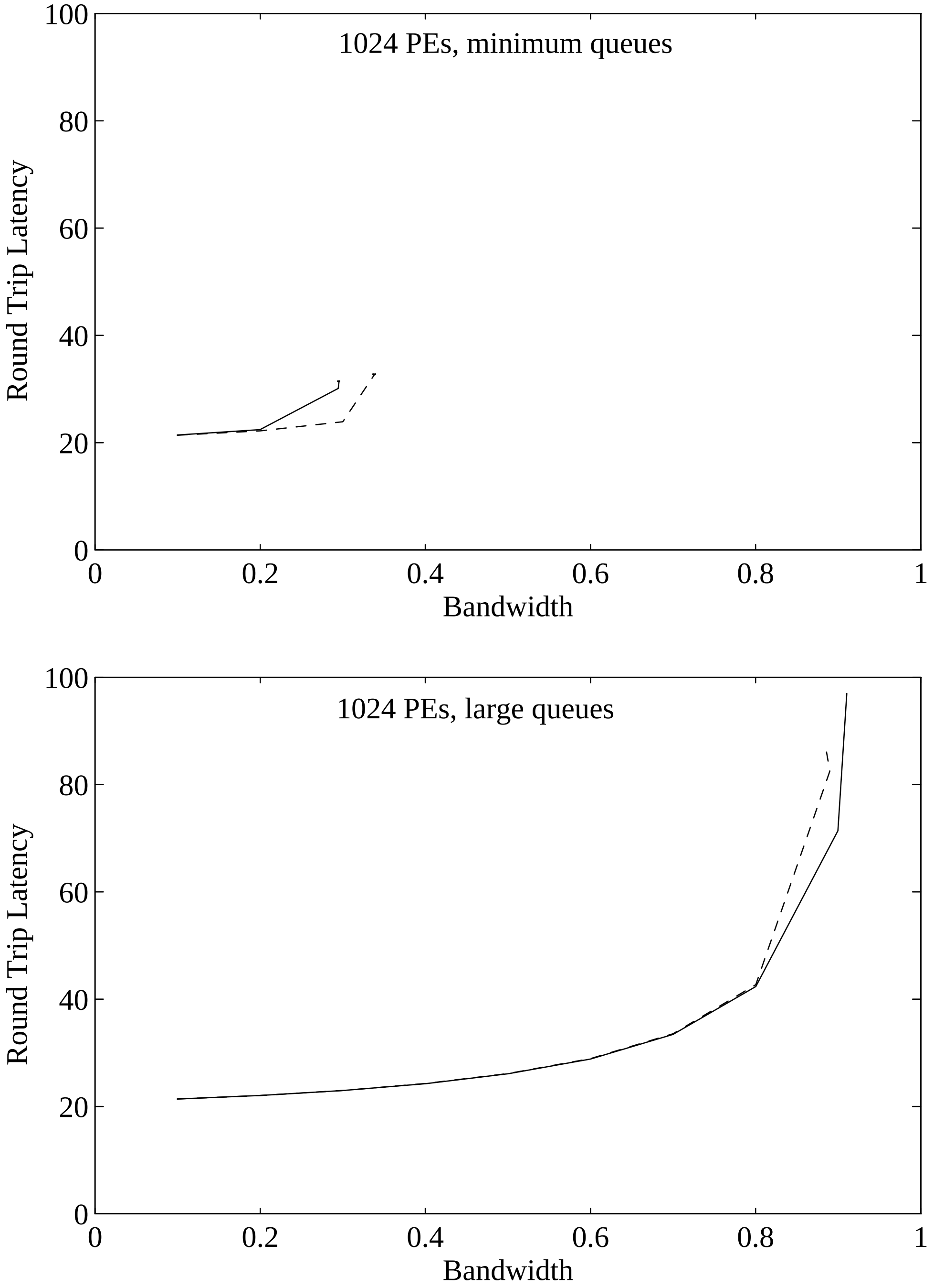


Figure 2.17: Latency as a function of bandwidth, 1024 PEs, small (Type A 2; Type B 1) and large (Type A 20, Type B 10) queues. Type A solid, Type B dashed. From simulation.

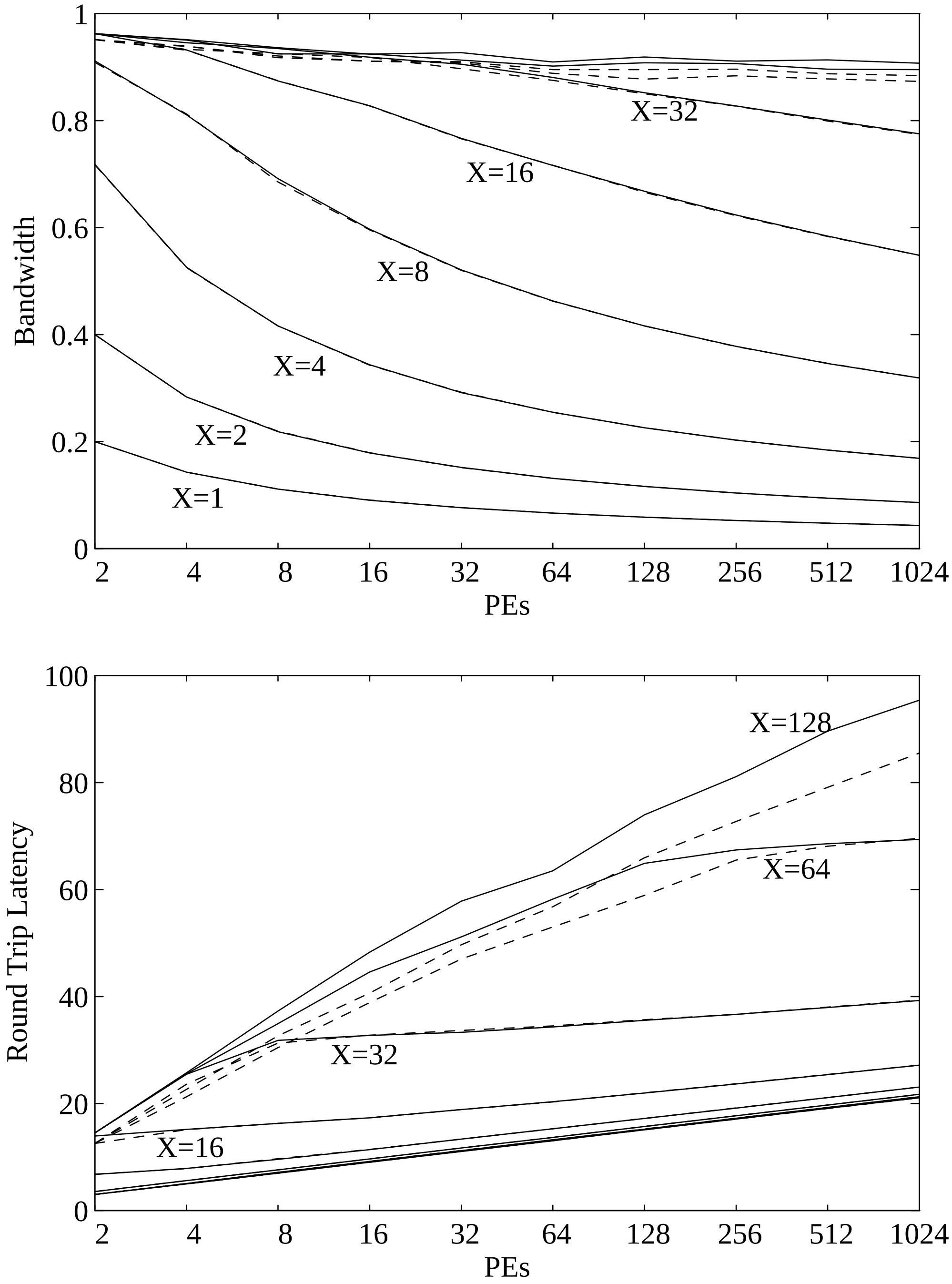


Figure 2.18: Bandwidth and latency values for Type A (solid) and Type B (dashed) networks as a function of the number of outstanding requests. From simulation.

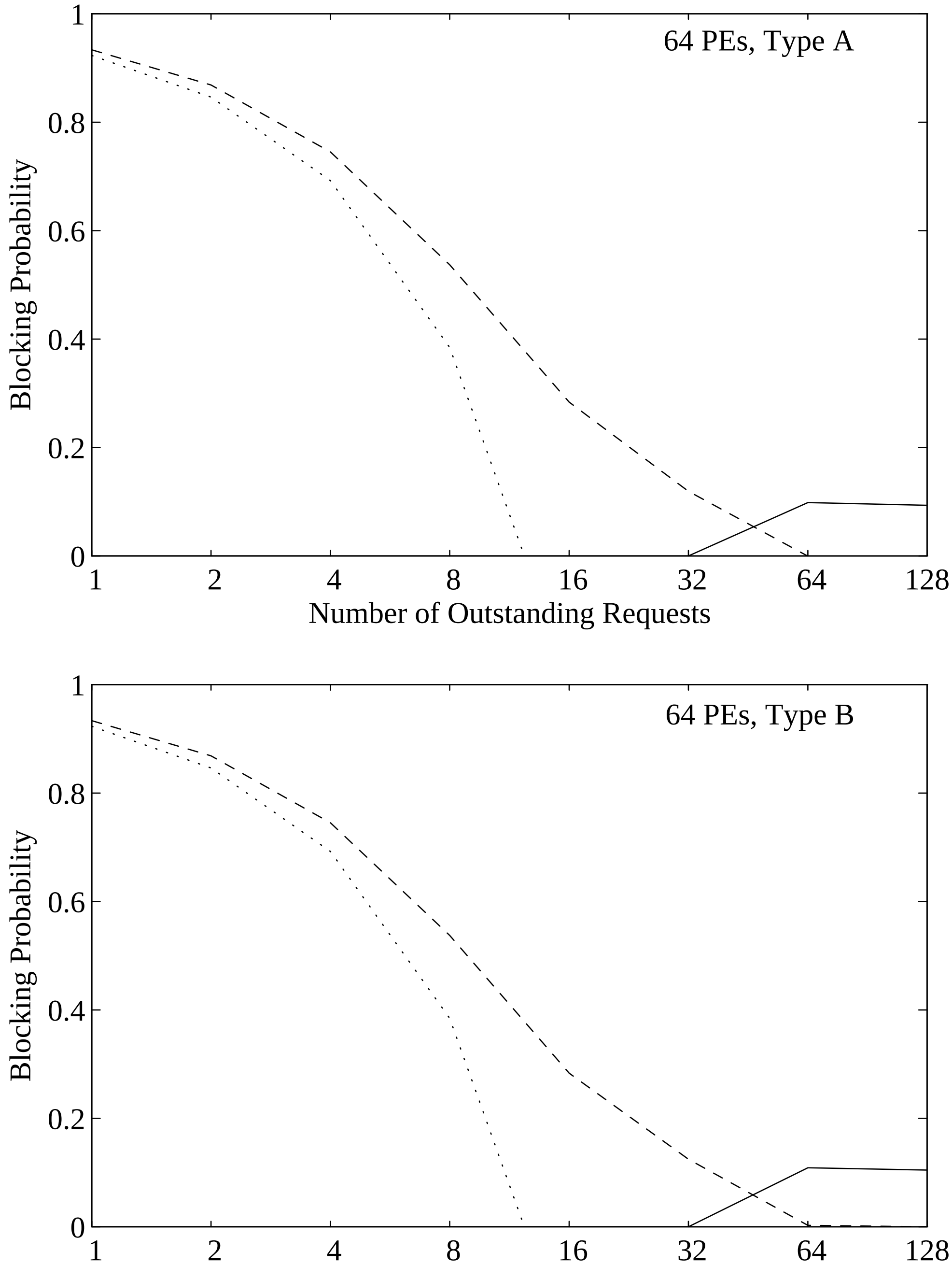


Figure 2.19: Blocking probabilities as a function of the number of outstanding requests, 64 PEs. Type A solid, Type B dashed, from simulation. Dotted line shows predicted blocking with constant delay

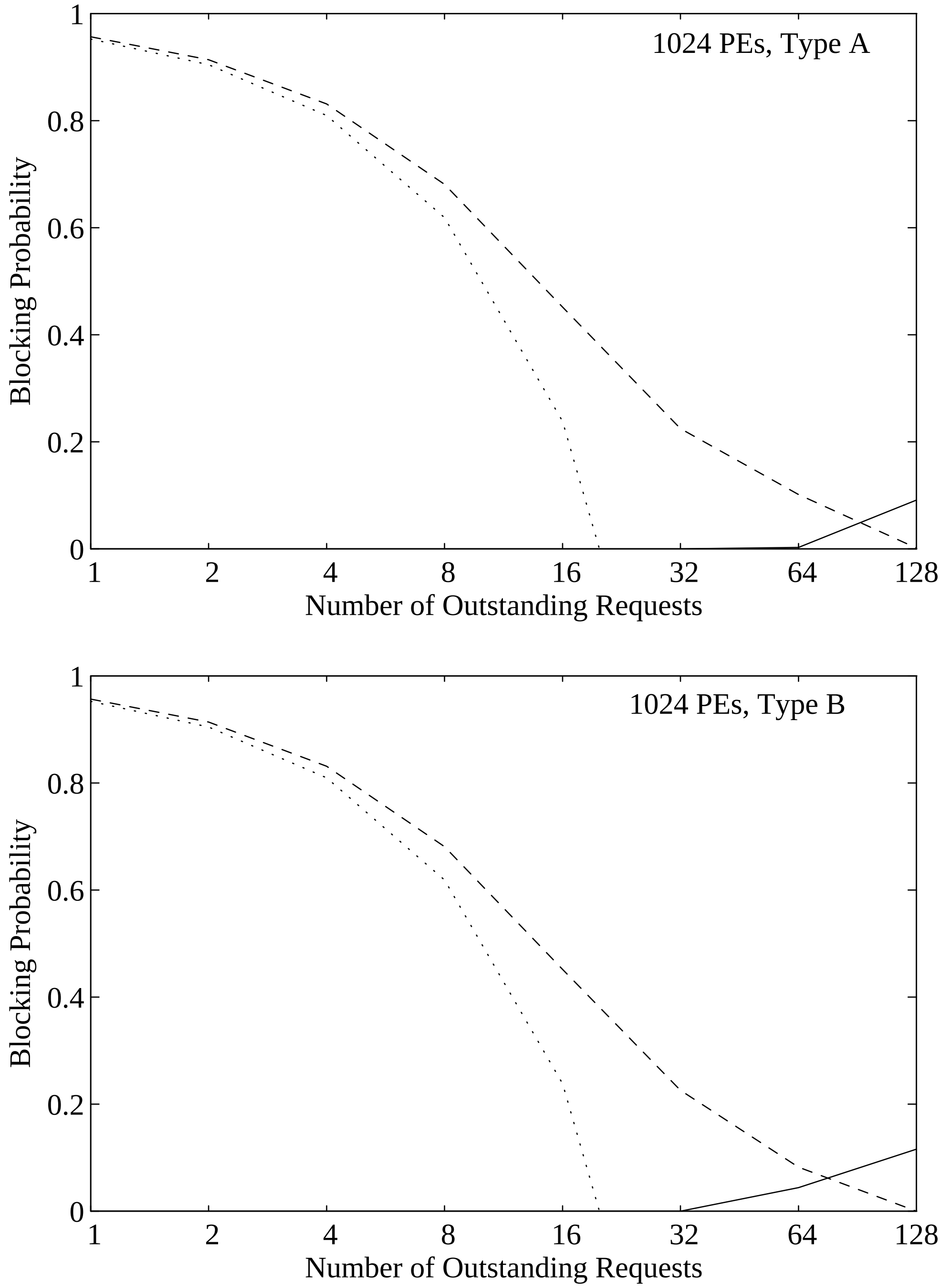


Figure 2.20: Blocking probabilities as a function of the number of outstanding request, 1024 PEs. Type A solid, Type B dashed, from simulation. Dotted line shows predicted blocking with constant delay.

2.4 Multiple packet messages

The results in section 2.3 assumed the entire message could be transmitted in a single packet. In practice, the pinout of chips and boards is limited, and even messages as short as requests to memory must typically be transmitted as multiple packets. For pure store and forward switching, latency for multipacket messages can be approximated by multiplying that for single packet messages by m^2, where m is the number of packets in a message, considering that the "cycle" for analysis is m times as long as before, and the probability of a message arriving in the new 'cycle' is m times as great. For cut-through switching, according to the analysis for switches with unbounded buffer size by Kruskal, Snir and Weiss [79], the average switch delay at the first stage is

$$1 + (\frac{m^2 p(1 - 1/km)}{2(1 - mp)}) \tag{2.33}$$

and the average switch delay at later stages is approximately

$$1 + (1 + \frac{4mp}{5k})(\frac{m^2 p(1 - 1/k)}{2(1 - mp)}), \tag{2.34}$$

where the additional factor of $4mp/5k$ was included in the formula to match simulation results. The average network traversal time T (in one direction) is the sum of the individual stage delays plus the setup time for the pipe, i.e. $(m - 1)$.

$$T = 1 + (\frac{m^2 p(1 - 1/km)}{2(1 - mp)}) + n(1 + (1 + \frac{4mp}{5k})(\frac{m^2 p(1 - 1/k)}{2(1 - mp)}) + m - 1 \tag{2.35}$$

For light loads, the extra delay for increasing the number of packets in the message is the pipeline delay $m - 1$, but for heavy loads, the increase in delay is dominated by the m^2 factor, as for store-and-forward.

Figure 2.21 compares the results of our simulations, which simulated finite queues, to the delay calculated from Equation 2.35. The queue sizes used allowed the storage of 10 messages per queue for Type A, 5 messages for Type B. Thus the storage per node was kept constant, though the storage per queue measured in packets increased in proportion to the number of packets per message. The number of outstanding requests was set at 128, to minimize the limitation on bandwidth due to waiting for a message to return. We show the latencies for the actual bandwidth, not for the offered load; the rightmost point of a curve corresponds to the maximum bandwith achieved. Solid lines shows results for Type A switches, dashed lines are for Type B switches, and dotted lines are from Equation 2.35.

Note that, with finite queues, the network has a capacity of under the theoretical maximum of $1/m$ messages per switch cycle per PE. For the queue sizes illustrated, the limitation is about 0.8 messages per switch cycle for single packet messages, and about 0.4 messages per switch cycle for two packet messages. This bandwidth limitation does not appear to be very sensitive to network size, as long as sufficiently many outstanding messages may be generated, as discussed in the previous section. The latency results are quite close to those predicted by the Kruskal, Snir and Weiss formula, providing both an independent test of the formula and a validation of our simulator.

The latencies shown in Figure 2.21 indicate the benefit of making the data path width as large as feasible, given other architectural constraints. A latency penalty in addition to the $m - 1$ pipeline delay (which may already be sizable compared to the network latency) can be seen at bandwidths well under the $1/m$ limit.

2.5 Increasing degree with constant pinout

As discussed in section 2.1.1, increasing the degree of a switch from 2×2 to 4×4 can be very advantageous for unbuffered switches. For buffered networks, increasing the degree of a switch is also attractive, if it can be done without increasing the number of packets in a message. However, the limited number of connections per node is usually a critical constraint on system design, doubling the degree is likely to require doubling the number of packets per message.

Figure 2.22 shows the results of simulations in which the number of packets increases as the degree of the node, for systems of size 64 and 256. Like the results in section 2.4, these simulations verify previous analysis,

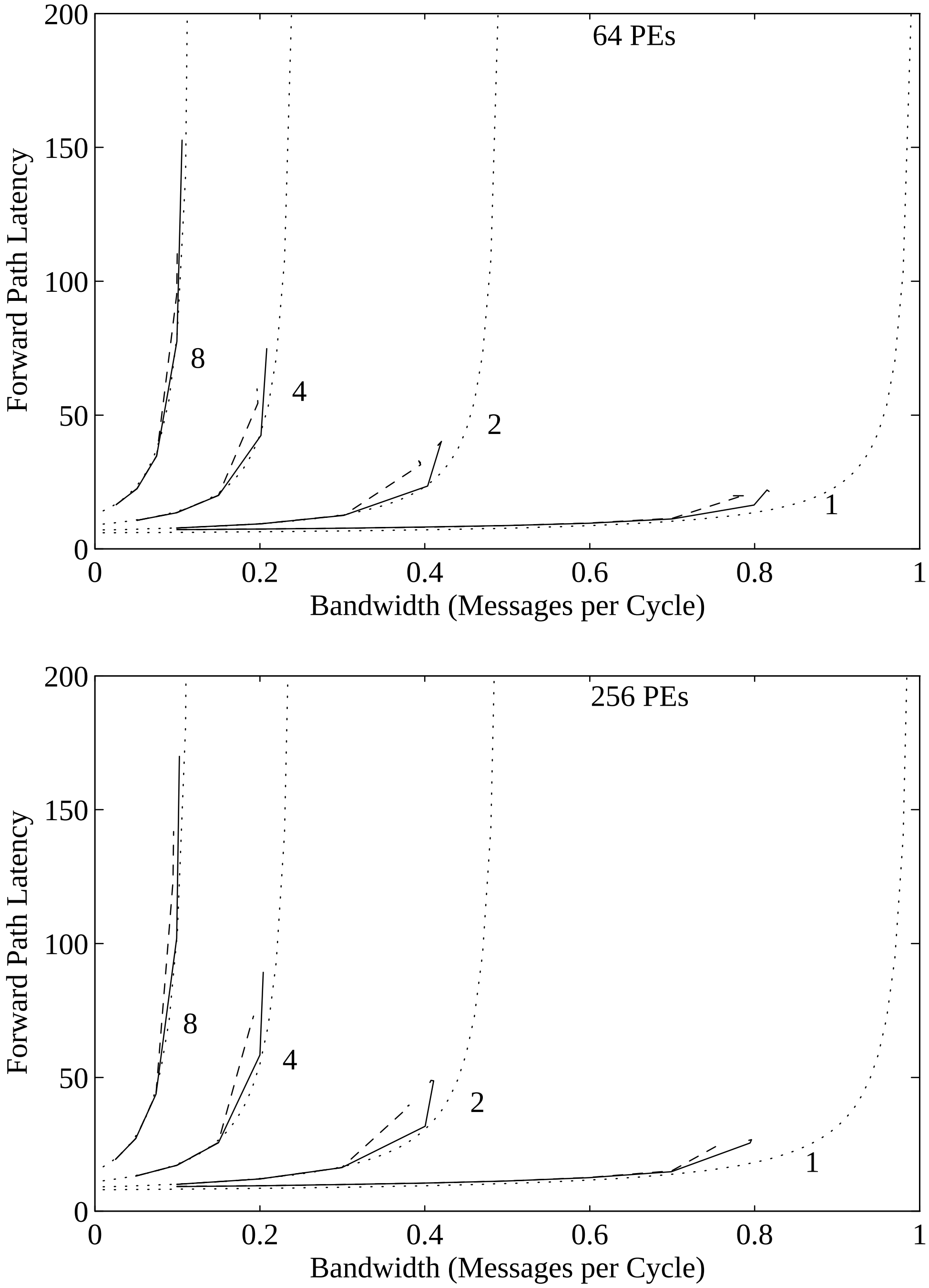
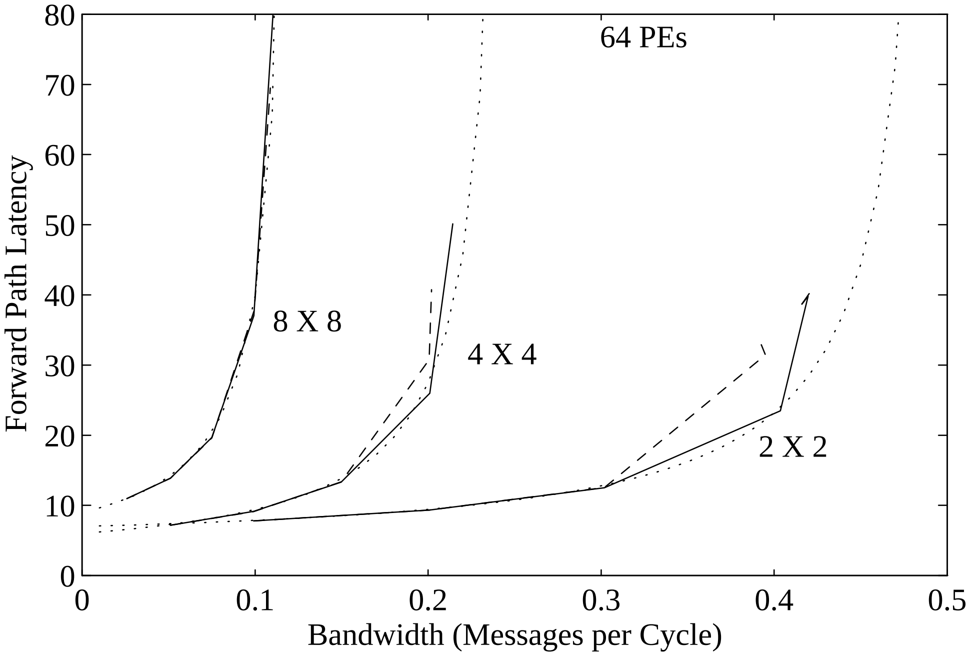


Figure 2.21: Multipacket messages, Type A (solid) and Type B (dashed) 2×2 switches, 64 PEs and 256 PEs, from simulation. Dotted line shows latency computed from Equation 2.35



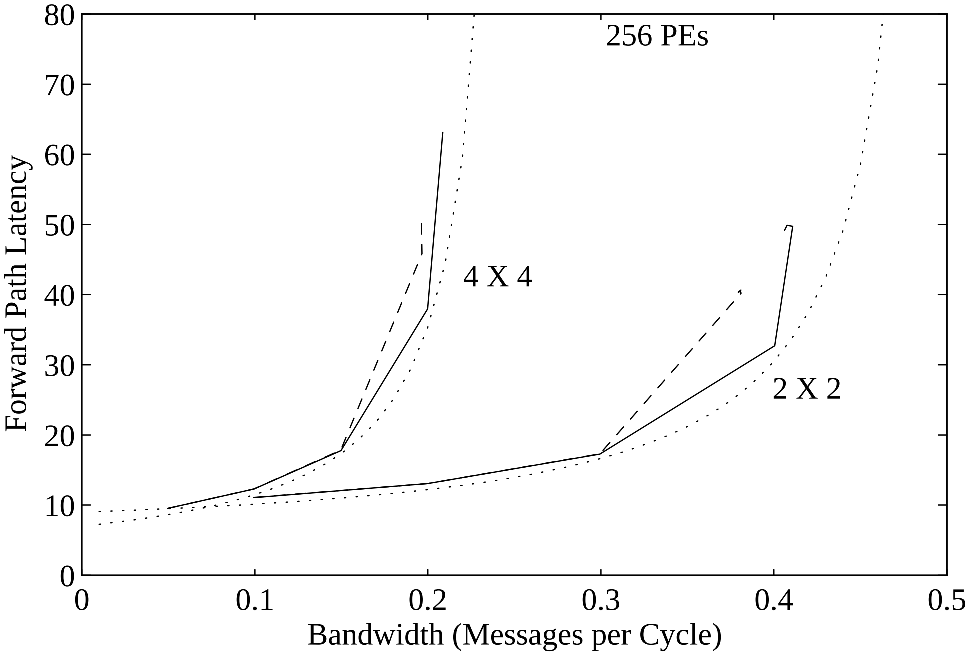


Figure 2.22: Different switch degrees, constant pinout, Type A (solid) and Type B (dashed) switches, 64 and 256 PEs, from simulation. Dotted line shows latency computed from Equation 2.35

in this case that of Kruskal and Snir [77] comparing the performance of systems with constant pinout per node. In these simulations, the queue size in messages was kept constant at 10 messages for Type A and 5 messages for Type B. The number of packets per message was taken to be equal to the degree of the switch. Constant storage per queue means that the storage per node increases as k for Type A $k \times k$ switches and as k^2 for Type B switches, and assumes that the nodes are pin-limited rather than storage limited.

As in section 2.4, the number of outstanding messages was 128 for the simulations shown in Figure 2.22, solid lines represent Type A switches and dashed lines represent Type B.

At these (relatively small) system sizes, 4×4 switches show a latency advantage over 2 switches only at bandwidths under 5 percent. In a 64 PE system, 8×8 switches show no latency advantage at all, since the pipeline delay is greater than the minimum network latency. However, as is pointed out in [77], the systems composed of 4×4 and 8×8 switches switches have fewer components and fewer wires, and thus represent less expensive networks.

Figure 2.22 does not show 8×8 switches for 256 PEs, since 256 is not a power of 8; the next power of 8 which is also a power of 4 is 4096, which is rather expensive to simulate. For interconnection networks with multistage delta network topologies, switches of small degree k have the additional advantage that, for a given maximum system size M, more subsystems with exactly $k^n < M$ processors exist. Such subsystems do not require redundant switches, as do subsystems that are not a power of k.

Chapter 3

Systolic Queue Designs

This chapter describes a non-combining systolic queue design and the implementation of this design in nMOS and in CMOS. The circuits used in this design were the basis for the design of the combining switch. The CMOS implementation used the NORA clocking methodology with modifications we developed for employing qualified clocks. We continued to use this clocking methodology in designing the combining and decombining components described in Chapter 4.

3.1 Advantages of systolic designs for VLSI

Systolic queue designs, as described in [60], have advantages even for non-combining switches. Memory-based FIFO designs require that input and output buses be connected to all storage elements; the capacitance on these buses must be charged and discharged for each insertion and deletion. Systolic designs require external connections only to the first slot in the queue. The semi-systolic design we have implemented requires only two global control signals distinguishing the four possible states of the queue: normal (both IN and OUT rows are moving), full, blocked and emptying (see [37]). These global control signals can be pre-computed, at the expense of an extra message slot to accept messages after the switch tells the preceding stage that it is full, and can be implemented efficiently as qualified clocks [38]. Furthermore, systolic queue designs have the advantages of regular layout and limited connections per cell which are characteristic of systolic structures in general [91]. An additional benefit is their suitability for cut-through switching, compared to alternatives such as that mentioned in [3] in which each cell of a shift register is connected to the output to allow partial cut-throughs.

The systolic combining queue design has the further advantage of distributing the comparison logic that finds matching messages so that it does not add significantly to the cycle time of the switch. A memory-based design, like those described in [141, 144], requires a comparator connecting the input bus to every message in the queue; such a comparison is likely to be quite slow and must be done in series with insertion, since the destination of the message on the input bus will be different depending on the result of the comparison. In a systolic combining queue, matching can be done in parallel with insertion. Comparisons are done within the systolic queue structure; each comparison requires interconnections between only two adjacent items, though many comparisons may be performed simultaneously.

3.2 Systolic queue design of Guibas and Liang

The systolic queue design we have implemented is essentially the folded shift register shown in Figure 3.1. Items enter the IN row at the left and move right until the slot underneath, in the OUT row, is empty. As items are deleted from the front of the queue, items in the OUT row move to the left.

Guibas and Liang's design [60] is fully systolic, according to Leiserson's definition [91], requiring no global signals except a clock. In their design, as items are inserted into the IN row, they move right whenever the slot to the right is empty and the slot below in the OUT row is occupied. As an item is deleted from the front of the OUT row, the cell the item has vacated is left "indisposed" instead of empty. Any item immediately

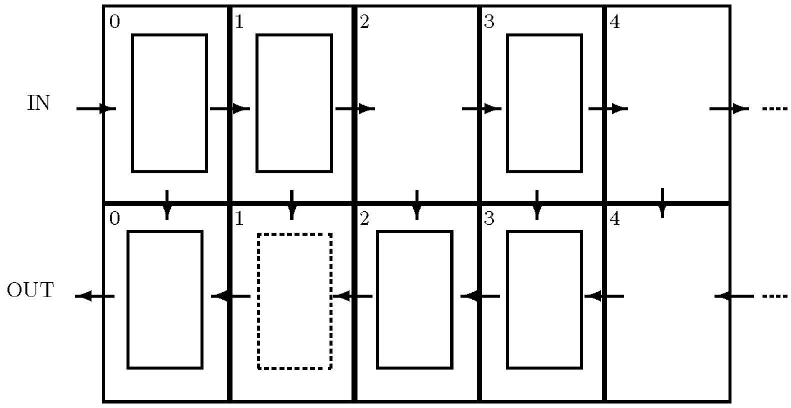


Figure 3.1: A systolic queue design.

to the right of an indisposed cell in the OUT row senses that it may move left on the next cycle, but an item in the IN row directly above will not move down. In general the OUT row will consist of a string of occupied and indisposed cells, terminated by a string of empty cells to the right. An item can be deleted or inserted every second cycle.

There are practical difficulties with using this scheme for the queues in an interconnection network switch.

1. To keep bandwidth up and latency down, the insertion rate should be limited by the off-chip delays, not by the characteristics of the internal switch logic. Not allowing insertions and deletions on consecutive clock cycles is a serious practical defect of this design.

2. The rules governing the movement of items, which are based on the assumption of unlimited size, are different for the rightmost cells in the IN and OUT rows.

3. Since the OUT row has three states, the representations of its state transitions can be somewhat complicated.

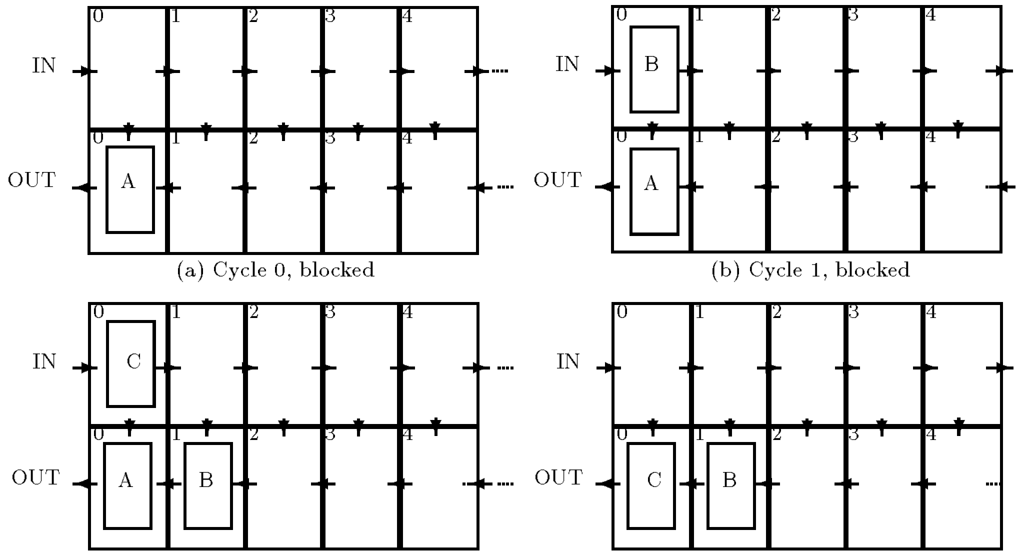
3.3 Snir and Solworth's basic queue design

Snir and Solworth [130] developed a semi-systolic design that distributes global information about whether the queue is blocked to each cell in order to be able to delete an item every cycle. If the queue is not blocked, a deletion will always occur and all items in the OUT row simultaneously move left. The global information allows all the items to move in lock-step into the neighboring location, even though it was not empty on the preceding cycle, thus providing an unbroken stream of items to be deleted. To handle the problem of finite queue size, this design also distributes global information about whether the queue is full. This information determines whether the items in the IN row move right or retain their position, allowing them also to move in lock-step as long as the queue is not full.

To describe the queue's operation more precisely, using a standard two-phase clocking scheme like that in Mead and Conway[102], let $\mathrm{in1}(j)$ and $\mathrm{in2}(j)$ correspond to the values of the item in slot j of the IN row at the end of phase 1 and at the end of phase 2. Similarly, $\mathrm{out1}(j)$ and $\mathrm{out2}(j)$ are the values for slot j of the OUT row. The Boolean variables $\mathrm{valid1}(x, j)$ and $\mathrm{valid2}(x, j)$ are true if there is an item in slot j of the x row at the end of the corresponding phase, false if the slot is empty.

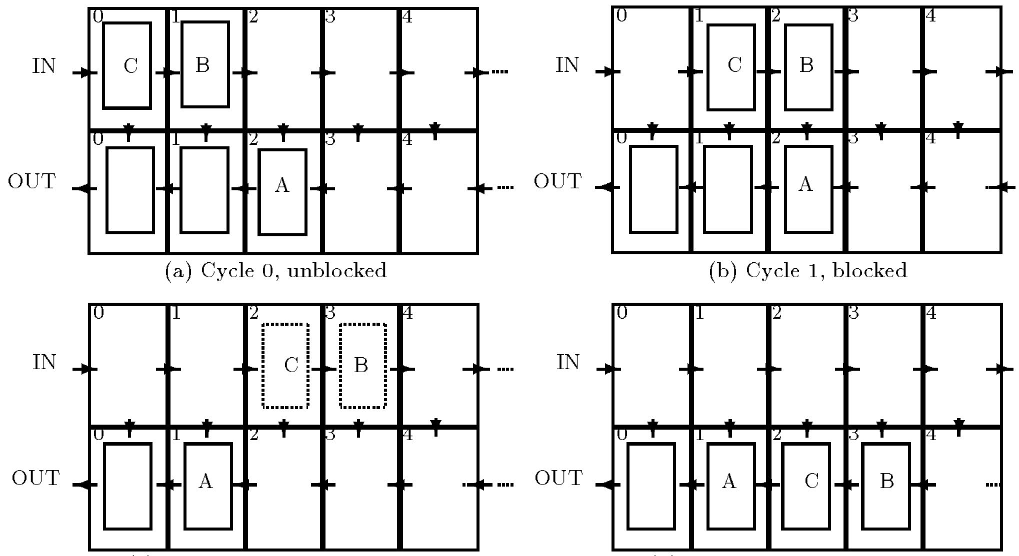
Using this notation, the phase 1 transitions of the Snir and Solworth design when the queue is neither blocked nor full are

$$\begin{aligned} \mathrm{in1}(j) &:= \mathrm{in2}(j-1), \\ \mathrm{out1}(j) &:= \mathrm{out2}(j+1), \\ \mathrm{valid1}(\mathrm{IN}, j) &:= \mathrm{valid2}(\mathrm{IN}, j-1), \end{aligned}$$



(a) Cycle 0, blocked
(b) Cycle 1, blocked
(c) Cycle 2, blocked
(d)Cycle 2, unblocked

Figure 3.2: A queue must unblock an odd number of cycles after the first item was inserted.



(a) Cycle 0, unblocked
(b) Cycle 1, blocked
(c) Cycle 2, unblocked, phase 1
(d) Cycle 2, unblocked, phase 2

Figure 3.3: Out of order items due to blocking a non-empty queue for an odd number of cycles.

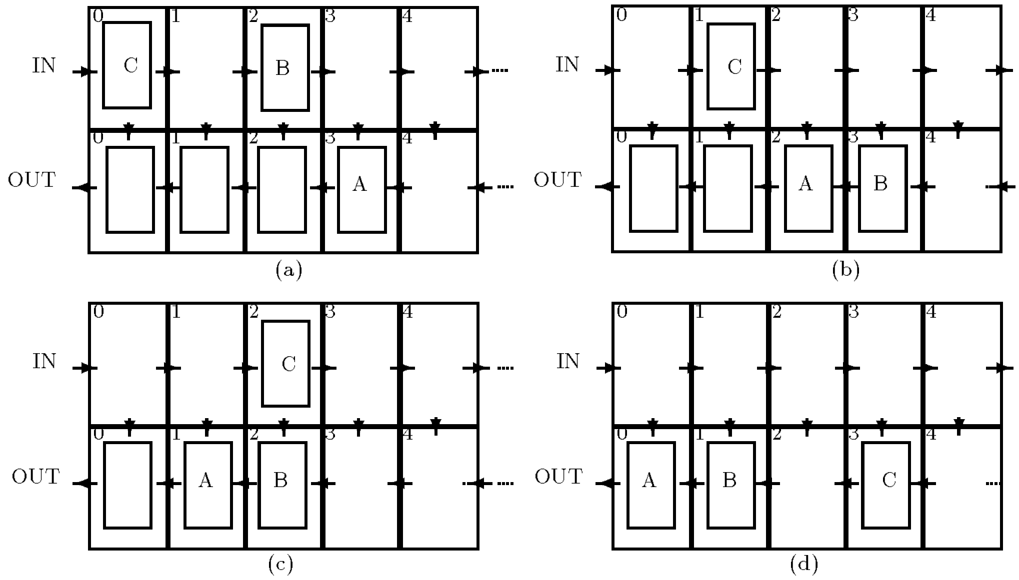


Figure 3.4: A hole in the OUT row due to an odd number of cycles between insertions.

$$\text{valid1}(\text{OUT}, j) \quad := \quad \text{valid2}(\text{OUT}, j+1),$$

and the phase 2 transitions are

$$\begin{aligned}
\text{in2}(j) \quad &:= \quad \text{in1}(j), \\
\text{out2}(j) \quad &:= \quad \text{if valid1}(\text{OUT}, j) \\
&\qquad\ \text{then out1}(j) \\
&\qquad\ \text{else in1}(j), \\
\text{valid2}(\text{IN}, j) \quad &:= \quad \text{valid1}(\text{IN}, j) \wedge \text{valid1}(\text{OUT}, j), \\
\text{valid2}(\text{OUT}, j) \quad &:= \quad \text{valid1}(\text{IN}, j) \vee \text{valid1}(\text{OUT}, j).
\end{aligned}$$

Snir and Solworth claim that the following invariants ensure correct operation of the queue:

1. The full slots in the OUT row occupy an initial segment of consecutive locations (i.e., there are no "holes" in the OUT row).

2. When the queue is neither blocked nor full, the difference between the largest index of a valid location in the OUT row and the largest index of a valid location in the IN row is non-negative and odd. [1]

They list the following conditions which must be satisfied by the control logic to maintain these invariants:

1. The number of cycles with no insertions between two consecutive insertions must be even.

2. When the queue is blocked, it must remain blocked for an even number of cycles.

These constraints are necessary to prevent items from getting out of order. Consider what happens when the queue is blocked for the first time. As long as the queue is not blocked, at the end of phase 2 a single item will be available for output as out2(0). When the queue is blocked, this item will remain in place and the IN row will keep moving. Let cycle 0 be the cycle that the item at the front of a queue that has blocked for the first time was inserted. As illustrated in Figure 3.3, if a stream of consecutive insertions is fed to the

[1] The numbering convention here is different than that in [130], so this difference is odd, rather than even.

queue while it remains blocked, the rightmost items will be lined up vertically at the end of each odd cycle. If the queue is then unblocked during the next even cycle, the items in the OUT row will move left during phase 1 while the items in the IN row move right, and during phase 2 two items will slide down into the OUT row in the wrong order. Thus correct operation requires that a queue stay blocked for an odd number of cycles after the first insertion.

Figure 3.3 shows the effect of blocking a non-empty queue. At cycle 0, the queue is in a legal state, with both rows moving and an odd difference between the highest indices of the IN and OUT rows. At cycle 1, the queue is blocked for one cycle; the IN row keeps moving but the OUT row is stationary. At cycle 2, both columns start moving, resulting in items B and C being placed out of order in the OUT row. Note that if C had not been inserted at cycle 0 there would have been a hole in the OUT row at cycle 2 instead, which might have taken some later item out of order.

The input stream need not be a continuous stream of consecutive insertions for correct operation, but, as illustrated in Figure 3.4, there cannot be an odd number of cycles between insertions. In the figure, there is only one cycle between B and C; this results in a hole between B and C in the OUT row.

In the Snir and Solworth description of their design, a queue is considered to be in one of three states: normal (both inserting and deleting messages allowed), blocked (no messages can be deleted, but messages can be inserted) and full (no messages can be inserted). They assume that a full queue has been blocked, so that no messages can be deleted. In implementing such a queue, the full signal can be derived from valid bit of the rightmost slot in the OUT row which can be occupied and still leave room for the rest of the message. There is a fourth state, which we will call emptying, after the output of the queue has unblocked but before the IN row can begin moving again. The IN row can begin moving when there is enough space to allow a complete message to be accepted even if the queue becomes blocked again.

3.4 A semi-systolic queue with two global control signals

As should be clear from the discussion and drawings in the previous sections, holes or out of order messages can appear in the OUT row only if at some cycle n both rows are "lined up" at the rightmost edge and at the following cycle $n+1$ both rows are moving. For correct operation of the queue, one must ensure that this situation can never arise.

In the switch, each item in the queue will be a packet of a message. In this section we show that a two-row semi-systolic queue design based on messages with an even number of packets, two global control signals and simple restrictions on when these global signals can change satisfies the invariants of the previous section, ensuring correct operation.

The two global control signals are in_moving and out_moving. The phase 1 transitions of the queue are:

$$\begin{aligned}
\mathrm{in1}(j) &:= \text{if in_moving} \\
&\quad\ \text{then } \mathrm{in2}(j-1), \\
&\quad\ \text{else } \mathrm{in2}(j), \\
\mathrm{out1}(j) &:= \text{if out_moving} \\
&\quad\ \text{then } \mathrm{out2}(j+1), \\
&\quad\ \text{else } \mathrm{out2}(j), \\
\mathrm{valid1}(\mathrm{IN}, j) &:= \text{if in_moving} \\
&\quad\ \text{then } \mathrm{valid2}(\mathrm{IN}, j-1), \\
&\quad\ \text{else } \mathrm{valid2}(\mathrm{IN}, j) \text{ and} \\
\mathrm{valid1}(\mathrm{OUT}, j) &:= \text{if out_moving} \\
&\quad\ \text{then } \mathrm{valid2}(\mathrm{OUT}, j+1), \\
&\quad\ \text{else } \mathrm{valid2}(\mathrm{OUT}, j),
\end{aligned} \tag{3.1}$$

and the phase 2 transitions are

$$
\begin{aligned}
\text{in2}(j) &:= \text{in1}(j), \\
\text{out2}(j) &:= \text{ if valid1}(\text{OUT}, j) \\
&\quad\ \text{then out1}(j) \\
&\quad\ \text{else in1}(j), \\
\text{valid2}(\text{IN}, j) &:= \text{valid1}(\text{IN}, j) \wedge \text{valid1}(\text{OUT}, j), \\
\text{valid2}(\text{OUT}, j) &:= \text{valid1}(\text{IN}, j) \vee \text{valid1}(\text{OUT}, j).
\end{aligned}
\tag{3.2}
$$

Consider that the index $j = -1$ corresponds to the values of the input port.
The following constraints must be enforced:

1. The first packet of a message is accepted only at an even cycle.
2. All messages are of even length.
3. The control signal in_moving can change its value only before an even cycle.
4. The control signal out_moving can change its value only before an odd cycle.

A cell j in the x row is said to be occupied if valid2(x, j) is true at the end of the cycle. Under the above constraints, the following can be proved:

Theorem 3.4.1 *If the queue is not empty and, at the end of cycle n, the highest index of an occupied cell in the IN row is equal to the highest index of an occupied cell in the OUT row, then in_moving and out_moving cannot both be true during cycle $n + 1$.*

Proof:
If the queue is empty, the theorem is trivially satisfied. Let cycle 0 be the first cycle that a message enters an empty queue; in_moving must be true to allow a message to enter, and the first packet may leave the queue at cycle 1 if out_moving is true. At the end of the first cycle the theorem is satisfied, since the premise is false, as OUT(0) is the only occupied cell. Assume inductively that the theorem is true for every cycle through $n - 1$, and thus at cycle n there are no holes in the OUT row. Assume without loss of generality that the queue is not empty at any cycle from 0 to $n - 1$.

At cycle n, for n odd, consider separately the cases when out_moving is true and when it is false.

1. out_moving is false: Since out_moving can only change at the beginning of odd cycles, it will still be false the next cycle, which is even.
2. out_moving is true: Some even number $2j$ of packets must have entered the queue since cycle 0, because there have been an even number of cycles and messages have an even number of packets and can be initiated only at even cycles. Some odd number $2k + 1$ of packets must have exited, because out_moving, since it is true now, was false for an even number of cycles from the time packets could have begun to exit at cycle 1. So $2j - 2k - 1$ packets (an odd number) must remain in the queue. Let m be the highest index of an occupied cell in the OUT row; there are m packets in the OUT row, since it has no holes. Suppose m were also the highest index of an occupied cell in the IN row. The number of unoccupied cells to the left of cell m must be $2l$ for some l, because messages are of even length and initiated only at even cycles. So there would be $m + m - 2l$ or an even number of packets in the queue. But the number of packets must be odd, so m is not the highest index of an occupied cell in the IN row.

At cycle n for n even, consider separately the cases when in_moving is true and when it is false.

1. in_moving is false. Since in_moving can only change at the beginning of even cycles, it will still be false the next cycle, which is odd.

2. in_moving is true. An even number $2k$ of packets has exited the queue since the the first packet could exit on cycle 1, as the queue has been blocked for an even number of cycles in that time. At the end of cycle n, in_moving will have been true for an odd number of cycles, so either an even number $2j$ of packets has entered and cell 0 of the IN row is not occupied, or an odd number $2j+1$ has entered and cell 0 of the IN row is occupied (because the queue is in the middle of receiving a message). This creates two subcases :

   (a) There are an even number of packets in the queue, since an even number have entered and an even number have exited, and cell 0 of the IN row is not occupied. Let m be the number of packets in the OUT row. There must be some even number $2l$, $l < m$ of unoccupied cells in the IN row with index greater than 0 and less than the highest occupied index in the IN row, so there would be $2m - 2l - 1$ packets in the queue if the highest occupied index were m, which contradicts the assumption that the number of packets in the queue is even.

   (b) There are an odd number of packets in the queue, since an odd number have entered and an even number have exited, and cell 0 in the queue is occupied. Again, there must be some even number $2l$, $l < m$ of unoccupied cells in the IN row with index greater than 0 and less than the highest occupied index in the IN row, so there would be $2m - 2l$ packets in the queue if the highest occupied index were m, which contradicts the assumption that the number of packets in the queue is odd.

Corollary 3.4.2 *The OUT row in a systolic queue operating according to the constraints given on cycle parity of message initiation, message length, in_moving and out_moving will not have holes or out of order packets.*

We implement the parity constraints by tying changes in in_moving and out_moving and the related input and output DA (data accept) signals (see section 4.1.5) to the end of a message on the input and output ports. The input DA must be lowered in time to prevent the queue from filling, but will not be noticed by the preceding stage until the end of the message being sent. This means that in the worst case there must be room for the maximum number of packets in a message in the OUT row when the input DA is lowered.

3.4.1 Queue blocked and queue full

A queue may be blocked either because a queue in the next stage is full or because of contention for an output port. To satisfy the constraints on in_moving and out_moving described in section 3.4, an external protocol, like that described in section 4.1.5, must make sure that a sending queue does not output a message unless a receiving queue can take it. Because a queue outputs a message after one cycle, the cycle parity of a receiving queue is the reverse of that of a queue sending to it.

A queue signals full when it can no longer take any more messages. If there are $2m$ packets in a message, then the queue full condition is equivalent to having the mth slot from the right in the OUT row occupied (see Figure 3.4.1). Data accept (DA) to the sending queue must be lowered when this slot becomes occupied. When this slot first becomes occupied, the parity restrictions ensure that the IN row cannot be receiving the last packet of a message. In the worst case the queue may be receiving the first packet of a $2m$ packet message, and the IN row must move $2m - 1$ more times. Even if the output is blocked, there will still be space in the queue for all packets. The sending queue must not initiate a new message after DA has been lowered. When that slot becomes unoccupied, DA may be raised, and there will be room for any new message initiated.

In our queue implementations, layout considerations cause us to use the valid1 signal from an occupied slot rather than the valid2 signal to produce the DA signal. Because in the detailed logic of the queue, valid2 becomes true while valid1 is still false, we actually take the valid1 signal from the $n+1$st slot from the right of the queue. If there were never any holes in the IN row and the OUT row were blocked, the valid2 signal of the nth slot would become true one cycle after the valid1 of the $n+1$st slot; so using this more conservative signal rarely makes a difference. Simulations show no difference between the two schemes in the number of messages contained in the queue when DA is lowered.

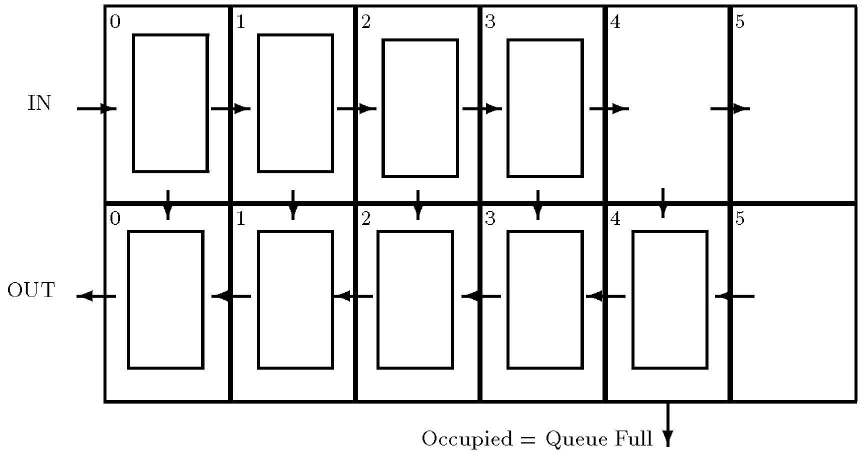


Figure 3.5: Illustration of the queue full condition for a systolic queue like that of section 3.4 which can accept messages of up to 4 packets.

3.4.2 Handling messages of odd length

Two characteristics of our semi-systolic queue design make it impossible for messages to have both even and odd lengths in packets:

- Both the IN and OUT rows can move at the same time.
- Transmission may not be halted in the middle of a message.

Consider the case when the OUT row has moved out the first packet of a message, and IN has accepted the first packet of another. Depending on the parity of length of the message just accepted, the queue may be put into a situation where the two rows line up, but because it is in the middle of a message in both rows, neither row may halt.

The requirement that messages be of even length raises the question of how to handle messages of variable length without undue waste of network resources. In the Ultra III architecture [17] used as the design point for the switch described in Chapter 4, load requests and store acknowledgements require only one packet for addressing and control information, while load responses and store requests require two packets, one address packet and one 32-bit data packet. Fetch-and-ϕ operations require two packets in both directions, and load double operations (used for instruction fetch) require only one packet on the forward path but three on the return.

Methods for making all messages of even length include:

Padding Include one extra packet for message that would otherwise be of odd length.

Packetizing Divide the "natural" length of packets in half so that all messages are of even length.

Padding is very simple, and wastes little bandwidth if messages are composed of many packets, but can be very wasteful if messages are short. Packetizing may complicate the network interface, by requiring disassembly and assembly of packets prior to connections with the processor bus, and limits the maximum network bandwidth in messages (see section 2.4), but avoids waste of network bandwidth. The relative performance of each of these schemes depends on the number of bits in each message and on the frequency with which each message is issued, which is hard to predict without detailed studies of both instruction and data cache behavior on a system. For the Ultra III prototype, we have chosen the simpler-to-implement alternative of padding the messages.

As the ratio of off-chip to internal logic delay increases with the decreasing feature size of VLSI designs, the internal logic of the systolic queue may be able to operate at a clock twice as fast as the rate at which data packets can enter and leave the chip. In that case, a systolic queue design that moved only one of the IN and OUT rows in a cycle could be used without penalty to allow messages of both even and odd lengths.

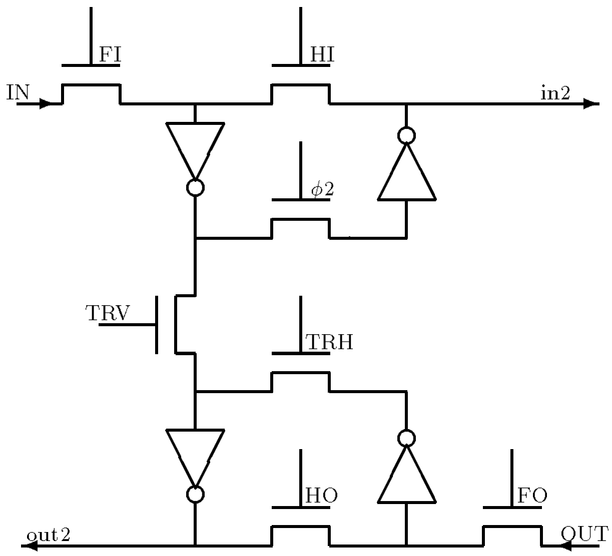


Figure 3.6: nMOS implementation of non-combining queue data cell

3.5 Implementation using nMOS

In our original nMOS implementation, we controlled movement in the single-bit data cell with qualified clocks derived from two-phase non-overlapping clock signals, as is done in Mead and Conway's stack design [102]. Data movement within a row (IN or OUT) occurs on phase 1 (when clock signal $\phi 1$ is high); data transfer from IN to OUT occurs on phase 2 (when clock signal $\phi 2$ is high). Pass transistors control the horizontal and vertical data movement from cell to cell in the data path (Figure 3.6). During the periods when both clocks are low, all pass transistors are turned off. The decision as to which direction data will move in a phase is computed in the previous phase; thus data movement and control computations are completely overlapped.

In Figure 3.6, IN and OUT are input signals from neighboring cells; in2 and out2 are output signals to neighboring cells. The qualified clock FI ("Flow In") is high only when $\phi 1$ is high and items in the IN row should move to the right; that is, when in_moving is high. HI ("Hold In") is high only when $\phi 1$ is high and in_moving is low. FO and HO are defined similarly for the OUT row; HI, FI, HO and FO control the phase 1 transtions described in equation 3.1. TRV ("Transfer Vertical") and TRH ("Transfer Horizontal") are a pair of $\phi 2$-qualified clocks controlling whether out2 is set from the IN or the OUT row, and are derived from valid1(OUT, j), corresponding to transition equation 3.2. The OUT row receives the value from the IN row whenever it is empty, whether or not the slot in the IN row is valid. The pass transistor with gate $\phi 2$ is used to transfer the phase 1 value to in2.

In preparation for the design of a complete combining switch chip, we designed several chips which were fabricated by DARPA's MOSIS facility. In 1986 we received functional 11-bit wide 2x2 non-combining switch chips containing approximately 7500 transistors and fabricated in 3-micron nMOS. These parts operated at a clock speed of 23MHz with propagation delays from clock to output of approximately 25ns. Power dissipation was approximately 1.5W. A 4x4 test network was constructed using four of these parts and functioned as expected.

A 6-bit wide portion of the FPC (without the adder) for a 2x2 combining switch was also fabricated in 4-micron NMOS. This switch was composed of four 1-input combining queues. These parts also operated as expected and had performance and power dissipation similar to the non-combining switches.

Since the final combining switches were to be at least 32-bits wide and air-cooled, we converted our design effort to the newly available scalable double-metal CMOS process. When we moved to CMOS to obtain a low-power implementation, we wanted to keep the compactness of the nMOS design.

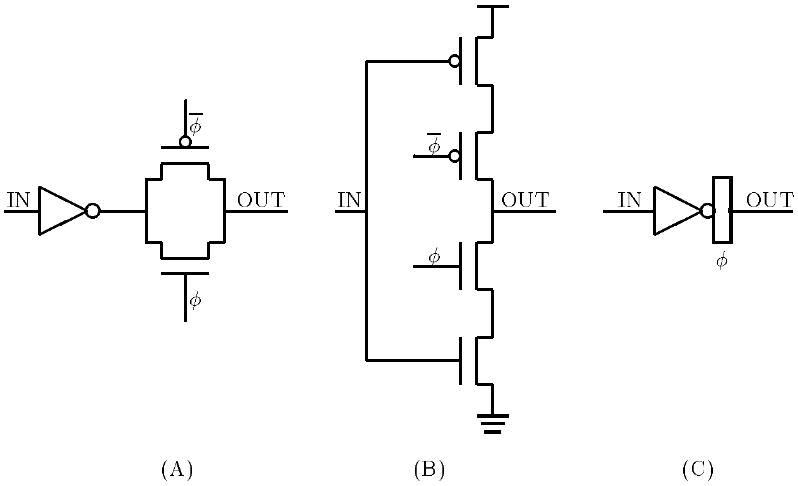


Figure 3.7: (A) Inverter and transmission gate. (B) C^2MOS latch. (C) Notation for C^2MOS latch.

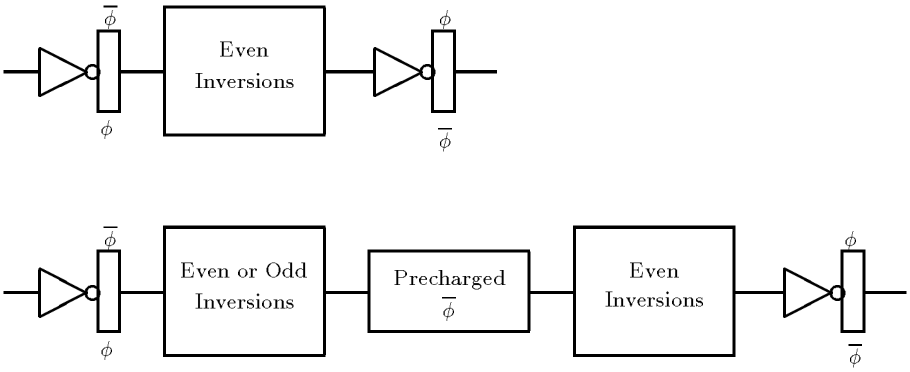


Figure 3.8: Summary of NORA inversion parity restrictions.

3.6 Implementation using NORA CMOS

A direct mapping of this design to CMOS transforms the pass transistors to transmission gates, as in Figure 3.7. However, four basic clocks ($\phi 1$, $\overline{\phi 1}$, $\phi 2$ and $\overline{\phi 2}$) must be provided in order to have non-overlapping clock phases. As in nMOS, the non-overlap time must be guaranteed in the presence of clock skew, so the requirement of non-overlapping clock phases can result in considerable dead time in each clock cycle. In CMOS this problem is compounded by having four clock signals to distribute.

3.6.1 NORA methodology

The C^2MOS latch (see Figure 3.7) can be used to replace an inverter followed by a pass transistor in the nMOS implementation. The key difference between the C^2MOS latch and the inverter followed by a transmission gate is that with the latch only one clock controls each transition of the output node, i.e., in Figure 3.7(B), OUT can be driven low only if ϕ goes high, and it can be driven high only if $\overline{\phi}$ goes low. This is true for every latch: the output of a latch can only be driven low when some clock is driven high and vice versa.

The NORA (NO RAce) methodology described in [52] uses this latch to construct two-phase pipelined circuits that use only two basic clocks, ϕ and $\overline{\phi}$. In this methodology, *phase 1* corresponds to the time when

ϕ is high and $\overline{\phi}$ is low, and *phase 2* corresponds to the time when $\overline{\phi}$ is high and ϕ is low. Therefore, in a standard NORA system, a phase 1 latch has ϕ on its N transistor and $\overline{\phi}$ on its P transistor; a phase 2 latch has the reverse. Restrictions on the parity of inversions between latches prevent data from flowing through a pair of latches during overlap periods when both ϕ and $\overline{\phi}$ have the same value.

The key to a properly-functioning NORA circuit is what happens during this overlap period. Consider the trivial circuit of one latch driving another latch and the case when both clocks are low. In that case, all the N clock transistors are off and all the P clock transistors are on; there is no path from ground to any output. During this overlap, the only transition that the output of the first latch can make is to go from low to high. That transition will not affect the output of the second latch because only an N transistor can be turned on and the path from the output to ground has been turned off by the low clock.

Generalizing, it can be seen that a signal that is an even number of inversions from the output of a latch on one clock phase can be safely connected to the input of a latch on the opposite clock phase. Similarly, a signal that is an even number of inversions from the output of an n-type precharged gate controlled by $\overline{\phi}$ cannot cause a transition on the output of a phase 2 latch. The use of precharged gates allows inverting as well as non-inverting logic between latches.

Goncalves and DeMan [52] developed rules for both n-type and p-type precharged logic blocks, giving added flexibility. As pointed out in Shoji [127] and discussed in section 3.6.4, this kind of circuit is extremely sensitive to noise. We use only n-type precharged logic, connecting successive n-type precharged gates with static inverters, in the style of Domino CMOS [74].

Figure 3.8 summarizes the NORA inversion parity rules developed in [52] that are used in our design style. There must be an even number of inverting gates between two C^2MOS latches on different phases if only static gates are used. If either a single precharged gate or a series of precharged gates connected by Domino-style static inverters is used, there must be an even number of inverting gates after the last precharged gate.

3.6.2 Qualified clocking in NORA

A circuit commonly used for producing qualified clocks is shown in Figure 3.9(A). ϕ and $\overline{\phi}$ are a pair of clocks denoting a particular clock phase. A, $\overline{A}$, A^*, and $\overline{A}^*$ are the qualification inputs, with the latter two logically equivalent to the former two. Q and $\overline{Q}$ are the output qualified clocks, which are active only during the clock phase controlled by ϕ and when A is active. Q will be connected to the gate of an n-transistor in a latch and $\overline{Q}$ will be connected to the gate of a p-transistor.

This circuit is often used in preference to static logic to reduce the propagation delay from the clock to the qualified clock. Such a delay is a component of the clock skew of the system, which should be reduced as much as possible.

The operation of this circuit in an environment with non-overlapping clock phases is straightforward. In such an environment, A and $\overline{A}$ are assumed to be stable during the clock phase that is being qualified. Therefore, Q and $\overline{Q}$ are always deasserted by their respective input clocks being deasserted. Thus, the two transistors N3 and P3 serve no logical function. When this circuit is used with non-overlapping clocks, these transistors are included as bleeders to ensure that Q and $\overline{Q}$ remain quiet, especially if there is a long period between assertions of A.

The same clock qualification circuit can be used in NORA CMOS. In that case, the signals ϕ and $\overline{\phi}$ represent the two clocks used in a NORA design rather than a pair of clocks representing a single clock phase. A correct NORA clock qualification circuit must produce logically correct Q and $\overline{Q}$ signals during the two valid clock phases. During the overlap periods, it must not allow Q or $\overline{Q}$ to change from an inactive to an active state; i.e., it must produce behavior on Q and $\overline{Q}$ that is no worse than that of ϕ and $\overline{\phi}$.

For NORA logic, it is important to distinguish signals that are logically identical, but are a different parity (odd or even) of inversions away from a latch, or are produced from the output of precharged gates. Our convention is to mark signals that are an odd number of inversions from a latch with a "$*$"; signals not so marked are an even number of inversions away from a latch. Although it is possible to use precharged gates between the $\overline{\phi}$ latch and one or more of the qualifying signals, doing so would introduce delays which would defeat much of the advantage of using a clock-qualification scheme in the first place. Therefore, only fully-static logic is used between the latch and the clock qualifier circuit. It is shown in [38] that the signals with parity as illustrated in Figure 3.9(B) are the ones required for correct operation of the clock qualification

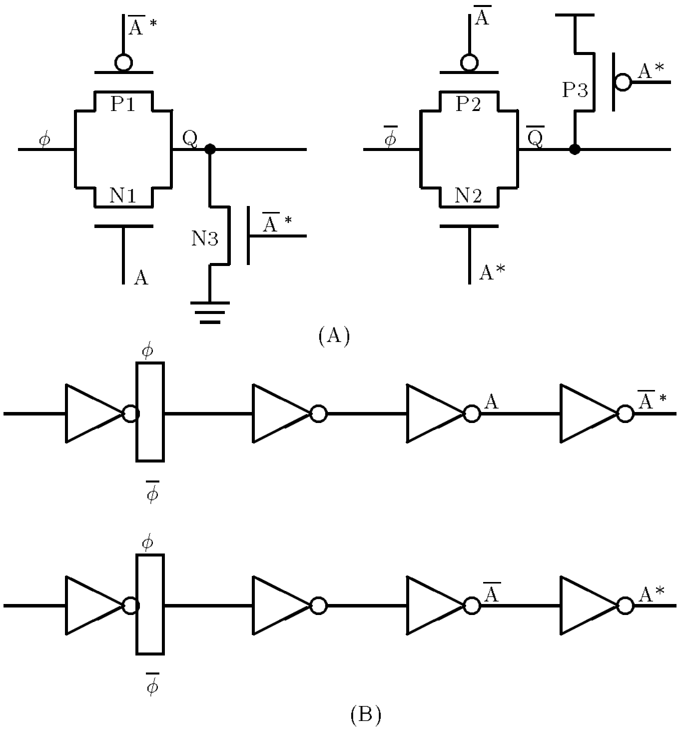


Figure 3.9: (A) Qualified clock circuits. (b) Parity from a latch of clock qualifier signals in NORA.

circuit.

3.6.3 CMOS non-combining switch

Our first NORA semi-systolic queue design was fabricated in 3 micron CMOS by MOSIS in 1987. The non-combining 2×2 switch containing this queue was a Type B switch, like the combining switch described in Chapter 4, but instead of packaging each queue separately, all four queues were placed on a single chip. A full-width data path was implemented internally, to verify the behavior and timing of global signals for a one-chip implementation of a 2×2 switch, but, due to the limitations imposed by the 80-pin maximum MOSIS package size at that time, only 10 outputs per port could actually be brought to the pads, so that 4 chips were required for a bit-sliced implementation.

This non-combining switch worked correctly at 11 MHz in a test environment, and was used in a functioning 2 PE prototype for over a year. The chip's performance compared favorably to a design of similar functionality using memory-based FIFOs fabricated in 1 micron CMOS at IBM for the RP3 project at about the same time. The IBM design operated at 33 MHz but had a three cycle latency between input and output [63].

3.6.4 Noise problems of dynamic logic

Although the chip tolerated 15ns clock overlaps, either high or low, it was highly sensitive to different values of the supply voltage and clock timings. We were unable to get the part to operate reliably in a system at a clock frequency above 4 MHz, although this may have been due to other problems in the system. The schematic of the basic cell of this systolic queue in this design is shown Figure 3.10.

After analysis of the situation as described in [38], we realized that, when used in a NORA circuit, C^2MOS latches have a low noise immunity during the clock overlap period and can operate incorrectly if the rise or fall times of the clocks are comparable to the propagation delay of the logic. We therefore revised our design to include at least one static gate between any two dynamic gates (whether latches or precharged

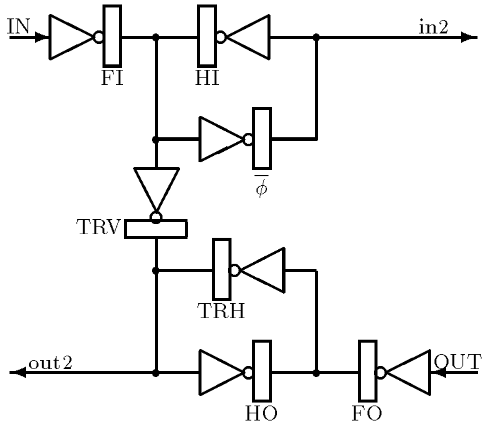


Figure 3.10: Basic cell of a CMOS queue design with noise problems.

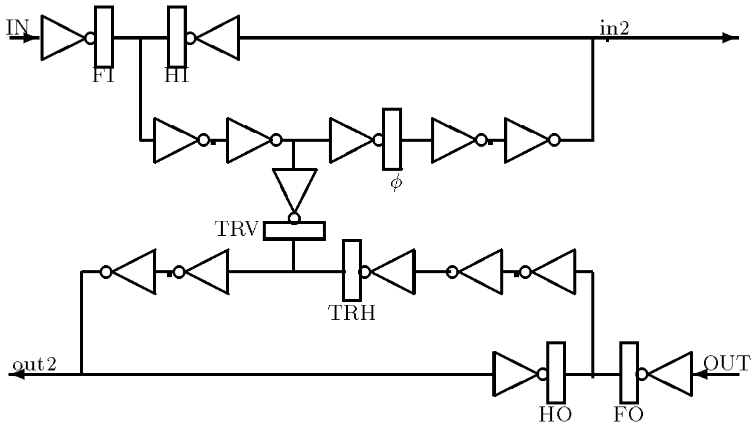


Figure 3.11: Corrected CMOS implementation of basic cell

gates). The revised design is shown in Figure 3.11. This was used as the basis of the combining queue design in Chapter 4.

Our experience in designing with the NORA methodology has been positive overall. Care is required in using dynamic latches and gates, and design of the control logic that produces the clock qualifiers is occasionally tricky. Sections of control logic must often be designed double-rail in order to have the clock qualifying signals the correct number of inversions from a latch. However, the data paths, which take up most of the area of the designs described in later chapters, are compact and efficient.

It should also be pointed out that output signals in NORA logic are not glitch-free. Due to precharging, signals may drop below logic high immediately after a clock transition even when they will return to logic high by the time the signal is valid. Such glitches on many outputs can create noise on the board, and require careful board design.

Chapter 4

Two-way Combining Switch

In implementing our combining switch, we avoided design choices that would hurt performance for non-combining traffic. Our goal was to design a network that provided the highest bandwidth and lowest latency possible, given packaging, processing, and human resource constraints. In particular, we were careful to avoid design decisions that would increase latency for light traffic.

This chapter provides a detailed description of the VLSI design of the combining switch and its packaging, packet format and protocols.

4.1 Combining switch architecture

The combining switch that we have designed and fabricated for use in the 16 PE NYU Ultracomputer prototype is a 2×2 switch node composed of four each of two types of custom VLSI design blocks: *forward path components* (FPC) and *return path components* (RPC) as shown in Figure 4.1. Control for the switch is distributed as tri-state selection logic in each chip. The routing information is included in the message. Flow control avoids the necessity of acknowledgement for messages and allows pre-computation of signals that control data movement from stage to stage. The components accept two and four packet messages and allow the first packet of a message arriving when the queue is empty to exit at the next cycle.

Both the FPC and the RPC have been fabricated in 2μ CMOS with 132 pin packages using the MOSIS service. Both parts run solidly at 10 MHz, the upper limit of performance that can be measured in the test rig. A 4×4 combining switch board has been in use in a 4 PE prototype since November of 1992, and functions correctly at all speeds at which the memory and processors work reliably (up to 15MHz). We are presently assembling a 16 PE, 16 MM prototype using these switch boards. Currently, in spring 1994, a 12 PE, 4 MM configuration is operational.

The combining switch components have also been prepared for fabrication by NCR in a 1.4μ process with 208 pin packages. While the MOSIS parts can be used only in systems with no more than 16 PEs, because only 4 pins are allowed for PE/MM address, the NCR parts can be used in systems of up to 256 PEs. The higher pin count packages also allow the logic from two MOSIS chips that share an output port to be packaged on a single chip. The differences between the two designs are noted as part of the descriptions in the following sections, and help to illuminate the effect of different technologies on design choices.

4.1.1 Packaging

Each of the 132-pin MOSIS parts contains either a single combining queue (FPC) or a return path queue and its associated wait buffer (RPC). Four of each type of component are used, one for each input/output combination. Two components are connected to each input bus; two components share each tri-stated output bus (see Figure 4.1). The two FPCs that share an output bus to the next stage also share a wait buffer bus that sends the information needed to decombine a message to a wait buffer in the correct RPC at the same time that a combined message is sent on the output port. The MOSIS FPC has two tri-state output ports (to the next stage and to the wait buffer) and one input port, for a total of 113 signal pins, excluding power, ground and clocks; 74 of these are output pins. The MOSIS RPC has one tri-state output port and

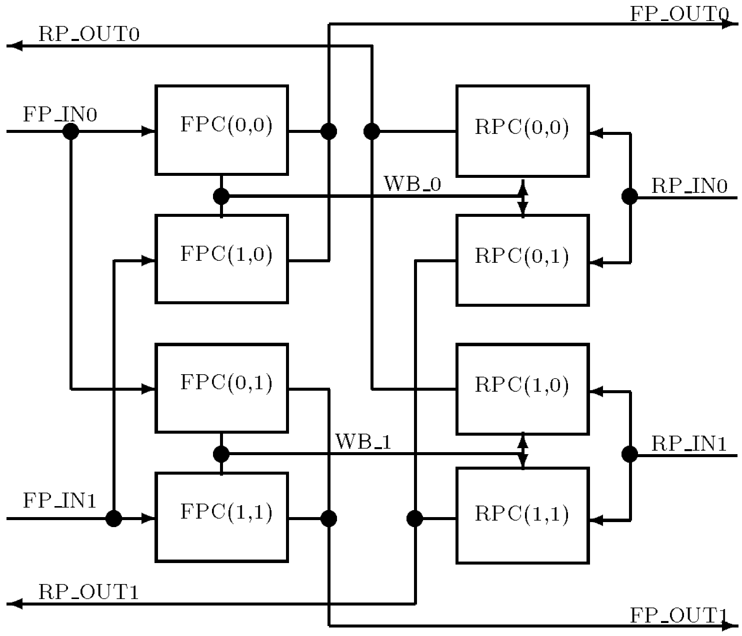


Figure 4.1: Block diagram of combining switch showing connections of forward path components (FPC) and return path components (RPC).

two input ports (from preceding stage and from the forward path), for a total of 108 signal pins; 38 of these are outputs.

The 132-pin packages allowed three 32-bit ports per component, but not four. At the next level of integration, there are two choices:

1. FPC(i, j) could be packaged with RPC(j, i),[1] so that the connection between the combining queue and the wait buffer is on-chip.
2. Components that share an output port could be packaged together, so that no output needs to be tri-stated and communication between queues sharing an output port can be on-chip.

The first choice saves pins, but the second choice allows the parts to operate at a higher clock rate, as discussed in section 4.5.3, so we have chosen the second alternative.

The NCR FPC has 2 output ports (next stage and wait buffer) and two input ports, for a total of 165 signal pins, excluding power, ground and clocks; 83 of these are outputs. The NCR RPC has has one output port and four input ports (two preceding stage and two forward path), for a total of 177 signal pins; 40 of these are outputs. Though the NCR RPC has five ports, while the NCR FPC has four, the ports in the FPC are wider because of the extra bits needed in the address packet (see next section), so the total number of signal pins is only 12 greater and, because there are only 40 outputs on the RPC compared to 83 on the FPC, the power and ground pin requirements are less.

4.1.2 Packet format

Each message consists of an address packet and from zero to two data packets. Data packets are 32 bits wide, to simplify the interfaces at the processor and memory. Messages with zero or two data packets are padded with a null packet to be of even length. Packet formats for the MOSIS parts are shown in Figure 4.2. The numbers in parentheses after the field names are the number of bits in the field.

[1] The first index is the input port number, the second index is the output port number, see Figure 4.1.

Forward path address packet

PE/MM address (4)	op code (4)	memory address (23)	ORI (4)

Forward path data packet

Extra (3)	Data (32)

Wait buffer address packet

PE/MM address0 (4)	ORI0 (4)	PE/MM address1 (4)	ORI1 (4)

Return path address packet

MM/PE address (4)	op code (4)	ORI (4)	SWD (1)

Return path and wait buffer data packet

Data (32)

Figure 4.2: Packet formats for the interconnection network in a 16×16 NYU Ultracomputer prototype.

In the MOSIS parts, which can be used in a network with a maximum of 4 stages, each forward path port has 35 bits, with 4 bits used for the PE/MM address. The 4-bit PE/MM address allows 16 MMs to be addressed by each of the 16 PEs. Since the NCR parts are designed for use in networks of up to 8 stages, each forward path port has 39 bits, and the PE/MM address field is 8 bits wide. On the return path, the address packet needs fewer bits than the 32 bit data packet because the 23 bits of memory address (i.e., the address of the referenced word within the specified memory module) are not returned to the processor. Both the MOSIS and the NCR parts have return path and wait buffer ports that are 32 bits wide.

At each stage in the network the routing decision is made on the basis of the high-order bit of the PE/MM address field. A port accepts a message as valid if this high-order bit matches the port number; otherwise, the message is ignored. Before being sent to the next stage, the high order bit is replaced with the address of the input port at which the message arrived (0 or 1) and the PE/MM address is rotated one bit left. At the memory this field is reversed and used as the MM/PE address field on the return path.

A 4-bit field is reserved for the op code. In addition to its use at the memory, the op code is used by the switches to control combining and decombining of messages and to signal the length of a message. The operations supported are described in section 4.1.4. A single SWD ("store was done") bit is returned to indicate whether or not a store was actually done for the conditional Fetch and Store operations, as explained in section 4.3.

The ORI (outstanding request index) field contains a number assigned by the PE when the message is issued. A PE must have only one message with a given ORI in the network at a time; the combination of PE/MM address and ORI is a unique identifier for a message at a switch. One such identifier must be sent to the wait buffer in the RPC to identify a message waiting to be decombined and another to identify the combine message that has been sent on to memory. The RPC needs no other information from the forward path in order to match and decombine a message. Thus for the MOSIS parts, only 16 bits are needed in the address packet of a message sent to the wait buffer in the RPC; the NCR parts need 24 bits. There are 16 extra bits for the MOSIS parts, 8 extra bits for the NCR parts, since the packet size is determined by the

32-bit data width. The data width determines the packet size on the return path as well; thus there are 15 unused bits in the MOSIS parts, 7 unused bits in the NCR parts.

The correspondence between fields in the address packet and the order of data packet bits remains the same so that both the MOSIS and the NCR parts can be used in the 16 PE Ultra3. The return path address packet format is designed to be compatible with the 16PE machine, at the expense of making the correspondence between address fields and data bit ordering for the 256 PE machine somewhat unintuitive.

For system sizes of less than 16 (for the MOSIS parts) or less than 256 (for the NCR parts), the network interface at the processor must place the PE/MM address at the high order end of the field, since routing will be done by the network on the $\log N$ high order bits of the PE/MM address.

4.1.3 Programming part location and system size

In order to route and match messages correctly, parts must know their location in the system: input number, output number and stage location. Because of pin limitations, the MOSIS parts are programmed using the values on certain pins during initialization; this requires external circuitry to multiplex these pins so that they are sourced from the programmed values during initialization and from their normal source the rest of the time. The stage mask is the value on the three low order PEtoMM address bits during initialization. If the value is high during initialization, that bit is masked out of the match logic and will not prevent a match from occurring if two messages differ only in that bit. (The high order PEtoMM address bit, which is the routing bit, need never be masked.) Thus the values these bits should have during initialization of a 4-stage network are 000 for stage 0, 001 for stage 1, 011 for stage 2 and 111 for stage 3. The In.Rte bit determines the index (0 or 1) of the input port and the Out.DA bit determines the index of the output port.

The NCR parts have four pins dedicated to programming and no external multiplexing. The input ports are programmed on-chip; the output port index is a single pin tied to power or ground. The stage mask is decoded on-chip from three dedicated input pins, whose value in binary is the stage number, counted from low to high from PEs to MMs.

Since the 16 PE Ultra prototype does not have enough pins available in the connectors on the boards to transmit the full width of the NCR FPC ports, both the NCR FPC and the NCR RPC will have an input SYS16 that will be used to treat the MM/PE address bits differently when being used in the 16 PE system with limited connector pins.

In the NCR FPC, bits 0-7 are the PEtoMM address bits in the 256 PE system. In the 16 PE system, bits 4-7 will hold the "real" address, and will map to the byte encoding bits and D0 in the data packet, as the PEtoMM address bits do in the MOSIS part. Bits 0-2 must be tied off-chip to GND; bit 3 must be tied off-chip to the input number, so that it will be shifted into bit 4, which is the low order bit of the 16 PE address. SYS16 set high causes GND to be shifted into bit 0 instead of the input port number. This on-chip change is needed for addresses sent to the wait buffer; addresses sent to the wait buffer will have bits 0-3 of the PEtoMM address set low. Since bits 4-7 hold the "real" PEtoMM address in a 16 PE system, stages 0, 1, 2 and 3 in a 16 PE machine must be programmed as stages 4, 5, 6, 7 on the forward path.

In the NCR RPC, bits 0-3 are the low order four bits of the MMtoPE address and bits 17-20 are the high order bits. Bits 17-20 must be set low at the memory for matches in the wait buffer to work correctly. SYS16 set high will cause bit 3 of the MMtoPE address to be used as the routing bit from the previous stage instead of bit 7 and will cause GND, rather than bit 3, to be shifted into MMtoPE bit 4. Also, when SYS16 is high, the routing bit from the forward path (which lets a wait buffer know whether or not it is supposed to take this message) will be bit 4 of the PEtoMM address instead of bit 0.

4.1.4 Operations supported

We implement a set of combinable memory requests that have been found useful in the development of parallel algorithms, including fetch-and-ϕ operations as well as loads and stores. If two messages with the same combinable op code meet in a queue, they will be combined, as described in [57].

Table 4.1 shows the ALU functions that must be performed on the forward and return paths to implement combining of this set of memory requests. Descriptions of the meaning of the names of the ALU inputs are given in sections 4.2.2 and 4.3.2. Only a four-function ALU, performing the functions ADD, OR, SELECT A and SELECT B on its inputs A and B, is needed for all of these operations. fetch-and-and is performed

Memory Operation	Forward Path	Return Path
Fetch and AndNot	OR	OR
Fetch and Or	OR	OR
Store Double	SELECT Chute	Don't Care
Fetch and Store	SELECT Chute	SELECT D
Fetch and Store if $= 0$	SELECT Out	if SWD SELECT D else SELECT RP_IN
Fetch and Store if ≥ 0	SELECT Out	if SWD SELECT D else SELECT RP_IN
Fetch and Add	ADD	ADD
Load Double	Don't Care	SELECT RP_IN

Table 4.1: ALU operations for the memory requests implemented in the combining switch

without implementing the AND function in the ALU by transmitting the bitwise complement of the data value and using the OR function for combining.

A three-function ALU would suffice for combining in the forward path if the data packet from the OUT row instead of the CHUTE row were output by the ALU for the fetch-and-store operation; see 4.2.2 for the problems with this approach. The proper decombining of conditional fetch-and-store operations is explained in section 4.3.2.

Combinable single word loads are implemented as fetch-and-add of 0, and thus may combine in the network with other fetch-and-adds. Combinable single word stores are simply fetch-and-store ignoring the returned value. In addition, the four-bit op code allows eight non-combinable operations that are forwarded to the MMs without receiving any special processing. The non-combinable operations can include partial word loads and stores, broadcast and reflect (see [43] for the use of reflect in the Ultracomputer architecture). Only broadcast requires any special processing in the network. Broadcast is done from a memory location to all processors, after a broadcast request has been received from a processor. A message is inserted in parallel into both queues connected to an input when the broadcast op code is decoded.

All messages are two packets in length unless the op code is all ones. This op code is used for Store Double in the forward path and Load Double in the return path; the memory must make the op code change. Both Store Double requests and Load Double responses are transmitted as four packet messages, with an empty final packet.

4.1.5 Flow control logic

As described in Section 3.4, the construction of the queues requires that there be an even number of packets per message and that switches distinguish even and odd cycles. A cycle is defined to be "even" for a given switch if the parity of the cycle is the same as the parity of the stage to which the switch belongs. Reception of messages starts only at even cycles while transmission of messages starts only at odd cycles.

At initialization, the parity of the cycle is set to the parity of its stage, so that cycles that are even for one switch are odd for its predecessors and successors. In fact, there is no explicit "even" signal for the queues. End of message (EOM) signals for input and output are generated on-chip in such a way that they always change at cycles of a given parity and are used to enforce the parity restrictions on operation. An explicit "even" signal is generated on-chip only for the wait buffer.

The wait buffer and the interaction between blocks in the RPC have been designed in such a way that the cycle parity of the FPC and RPC in the same switch are identical (see section 4.3).

Each port has two protocol bits: a data valid bit (DV) traveling in the same direction as the data and a data accept bit (DA) traveling in the reverse direction. The two protocol bits, in conjunction with the routing bit, regulate the transmission of messages through the network. A sender asserts DV when it wishes to initiate a message transmission. Independently, a receiver asserts DA when it is able to accept a new message. A message transfer starts only if both DV and DA are asserted and the cycle parity is correct. Signals controlling the blocking and unblocking of the queue are ignored during cycles when a message transfer cannot be started (that is, they are looked at no more often than every other cycle), so they can be set ahead of time to overlap data transfer and flow control operations.

Note that this is not strictly speaking a handshaking protocol: DA is not an answer to DV, nor an acknowledgment, but is issued independently and simultaneously. The sender is transmitting the data on the data lines whenever DV is asserted. If it receives DA, it assumes the data has been accepted and proceeds with the next packet. No provision for retry is necessary.

Since each sender sends to two receiving queues or wait buffers, DA must be asserted only when both can accept data. In the partitioning of Figure 4.1, this requires either an external AND gate for each paired input (the solution we used with our MOSIS-fabricated parts), or the transmission of two DA signals back to the source of the message, where the AND gate can be done on-chip (the solution we used with our NCR-fabricated parts). More complicated protocols could use the DA signals from both receivers to improve performance; see [100].

The MOSIS parts have a single data accept (DA) signal and a single data valid (DV) signal per port. The DA signals from two input ports receiving from a single output port are ANDed together off-chip to produce the DA to the source output port.

The NCR parts have one DV bit per port, and in addition two decombine queue data valid signals will be brought out from the RPC for performance monitoring. The input ports have a single DA per port, but both DAs are sent to the source output port where they are ANDed together on-chip. This requires an extra pin per output port on the chip and on the connections between stages, but eliminates the need for any off-chip logic on boards using the NCR parts.

Our protocol requires that message transmission not be halted once it begins. This requires reserving enough buffer space whenever DA is asserted to accept the rest of the message. In our implementation, this means four packets. (Were these only two-packet messages, the requirement of an even number of cycles between blocking and unblocking of the queue would require reserving enough space for the second packet, in any case.)

With the current protocol, to distinguish between two and four packet messages, queues paired at an input keep track of the length of valid messages sent to the other queue in order to know when to look at the input for the start of a new message. The op code bits are decoded and either the second or fourth cell of a shift register is set to 1; the output of this shift register is then used to signal input EOM (end of message). Whenever input EOM is asserted, and on the next cycle DV is asserted, a new valid message is assumed to begin for the queue with output port number equal to the routing bit.

In order to block in the middle of a message for messages of 4 or more packets, a change to this input logic would be necessary. It would no longer be sufficient to decode the op code in the first packet of a message and wait two or four cycles before looking again. A queue must know whether or not the other queue is blocked in order to know whether a new message may begin. This requires the communication of the DA signals between the paired queues. In both the MOSIS and the NCR packaging, this would be an off-chip communication.

An alternative solution would be to transmit an EOM signal instead of computing the length of the message from the op code. This would require an additional pin per port on stage-to-stage (off-board) communication. To allow pre-computation of chip control signals, it would be best to transmit this signal not in the last packet but in the second-to-last packet of a message.

4.1.6 Arrangement of buffers

We have implemented Type B switches, described in section 2.1.3, for the following reasons:

- Output rate and average queue lengths are equivalent to Type A for uniform traffic [115].
- Implementing the two-input queues for a Type A switch requires either separate locations for the insertion of the two inputs, or the serialization of the two insertions. Two input systolic queue designs, which have 2 IN rows and one OUT row, carry out arbitration for each slot of the OUT column rather than just at the output and avoid the serialization problem [130]. However, the logic would have been more difficult to get right in our initial implementation than that of the Type B switch, which requires only arbitration of the outputs.

- Type B switches, which require only one input, one output, and for combining switches, one port to the wait buffer, have greater packaging flexibility than Type A, which require two input ports, one output port and one wait buffer port per switch. To implement Type A switches at the level of package integration available to us would have required increasing the number of packets per message.

A cost of using Type B switches is fewer combines in the earlier stages of the network, since messages from different PEs that might combine cannot be together in a queue until one stage later than with Type A switches. According to simulation results in [100], this results in a 13 percent reduction in throughput from that achieved by Type A two-way combining switches, for systems with 64 PEs and a hot-spot rate of 3 percent. This is still twice the throughput achieved with a non-combining network under those conditions. For systems of 1024 PEs with a 3 percent hot-spot rate, Type B switches show a 21 percent throughput reduction compared to Type A switches, but still have 28 times the throughput of a non-combining network.

4.1.7 Arbitration of buffers

The arbitration rules for the queues paired at an output in Type B switches must not reduce bandwidth or starve one of the queues. The analysis described in [115] assumes that if one queue is empty and the other is not, the non-empty queue will drive the output, and that if both queues are not empty, the queue driving the output will be selected at random.

In practice, it is not easy to find a simple and reliable digital CMOS circuit that will select each output randomly but with equal probability. Strict alternation of outputs, ignoring whether or not the selected queue has a message, is easy to implement but increases latency under light load. Our first implementation alternated the selection whenever both queues had messages, chose the non-empty queue if only one was empty, and remembered which queue was selected last when both queues were empty, in order to give priority to the queue which had not sent the last message. In pseudo-code, the arbitration logic is

```
if not empty (0) and not empty (1)
        select := not (old-select)
else if not empty (0)
        select := 0
else if not empty (1)
        select := 1
```

for queues connected to inputs 0 and 1, where *select* equal to i corresponds to the control signal for the tri-state pads selecting the queue connected to input i. We implemented this in CMOS as a single AOI (and-or-invert) gate with 4 inputs but no more than two transistors on any path between a power rail and the output, plus the latches necessary to save the old value of *select*. All logic was carried out double-rail, for NORA correctness, since the AOI gate required both the old *select* signal and its inverse.

Unfortunately, as we discovered when simulating the design in a network environment, this logic does not take into account what happens when the queue is blocked. If the next stage blocks for an odd number of arbitrations while both queues are not empty, unblocks for long enough to allow one message to exit and then blocks again for an odd number of arbitrations, the same queue will get repeated service and the queue associated with the other input will be starved. As we discovered while running a version of the switch with this arbitration logic, such starvation is not necessarily an uncommon or self-correcting event. Certain memory behavior can cause it to occur consistently. To prevent this starvation due to arbitration while blocked, the information about whether the output is blocked or not must be included in the arbitration logic. While the output is blocked, the selected queue and the priority must not change. Thus no queue will get repeated service after an interval of blocking if the other queue has a message to send. We implemented this new arbitration logic by multiplexing the output of the AOI gate producing the new *select* value with the old *select* value, choosing the old value whenever the output is blocked. The control signal on this multiplexer is the AND of the DA from the next stage and the DA from the wait buffer.

This logic can be generalized to arbitrate k queues at an output. In that case the input to the logic will be k empty signals. Selection priority will be given to queue i if queue $i-1$ modulo k was the last queue to send a message. If queue i is empty, the next non-empty queue j obtained by incrementing i modulo k will be selected to send a message, and priority will be given to queue $j+1$ at the next arbitration. If all queues are empty, or if the output is blocked, priority will stay at queue i. This generalization was implemented for $k=4$ on the return path, where each RPC has one queue for messages received from the previous stage and one for decombined messages, making four queues altogether at each output.

Implementing this generalization of the logic required replacing the two AOI gates of the two queue implementation with 22 inverters, NAND, NOR and AOI gates, all with four or fewer inputs, resulting in a worst case of four gate delays from the *not empty* and *old-select* signals to the input of the multiplexer controlled by the DA signals.

4.2 Forward path component design

The forward path component in the MOSIS design is essentially a single systolic combining queue, with the ALU embedded in the first packet location of the queue, rather than on the critical off-chip path. Embedding the ALU in the first slot of the queue allows combining to be done in parallel with data movement off-chip. Loss of combining in the first slice does increase memory latency due to queueing delay in the presence of hot spots, as shown in Chapter 5, but the alternative is to increase the latency for all traffic patterns, either by degrading the cycle time or losing the property of having single cycle transmission time when the queue is empty.

Control logic on each component decodes the op code to determine the ALU operation and the length of the message. The output is blocked if a wait buffer data accept signal (WB.DA) or a data accept signal from the next stage (FP_OUT.DA) is low. The QNE (not empty) signal from the paired queue is used to determine which component has priority to drive the output port, as described in section 4.1.7; out_moving is true if the output is not blocked and the component has priority to drive the port. The DA (data accept) signal sent to the previous stage is derived from the queue full signal and latched on the input end of message (EOM) signal. The EOM signal is derived from the op code, which can be used to differentiate two and four packet messages.

4.2.1 Combining queue

The semi-systolic combining queue in the forward path component adds another row to the queue design described in section 3.4 (see Figure 4.3). A non-combining queue has a hardware cost similar to that of a shift register; the extra row costs about 50 percent more, while the comparator to recognize matching messages adds only 8 transistors to each cell (Figure 4.4). The comparator consists of a dynamic XOR gate followed by an inverter whose output can pull down a pre-charged *match* line which is shared by all the cells used for the match. These gates form a short chain of Domino CMOS logic, as described in [74, 127].

The comparator checks for equality between every slot in the IN row and the corresponding slot in the OUT row. As messages move to the right along the IN row they are compared with the messages in the OUT row below. If a match occurs, data from the IN row moves to the CHUTE. Comparison can be done in parallel with data movement because the message in the IN row is copied to the the CHUTE as long as the latter is empty. A valid bit for each row in the slot is set based on the result of the match. The comparator to recognize matching messages is a few transistors distributed in each cell (Figure 4.4).

If the IN and the OUT row move simultaneously, an entering packet will be matched against alternate packets. If every message has two packets, an entering address packet will be matched against the address packet of every message in the queue when it arrives except the first (since we have placed the adder in the first slice and do not allow combines there). The systolic design we use, which allows a packet to enter one cycle and exit the next, already constrains messages to have an even number of cycles between the start of one message and the start of the next (see [130] and section 3.4), so the addition of combining places no further constraint on the number of packets in a message. Since messages can be either 2 or 4 packets long, and the IN and OUT rows need not be moving simultaneously, a match occurs only if both the IN row and the OUT row contain address packets. To ensure that matches take place only between two address packets, an internal start of message (SOM) signal enters the queue with the first packet of a message.

A block diagram of the combining queue is shown in Figure 4.5. The input to the first valid cell is FP_IN.DV; the outputs are FP_OUT.DV and WB.DV. HI, FI, HO and FO have the same meaning as in

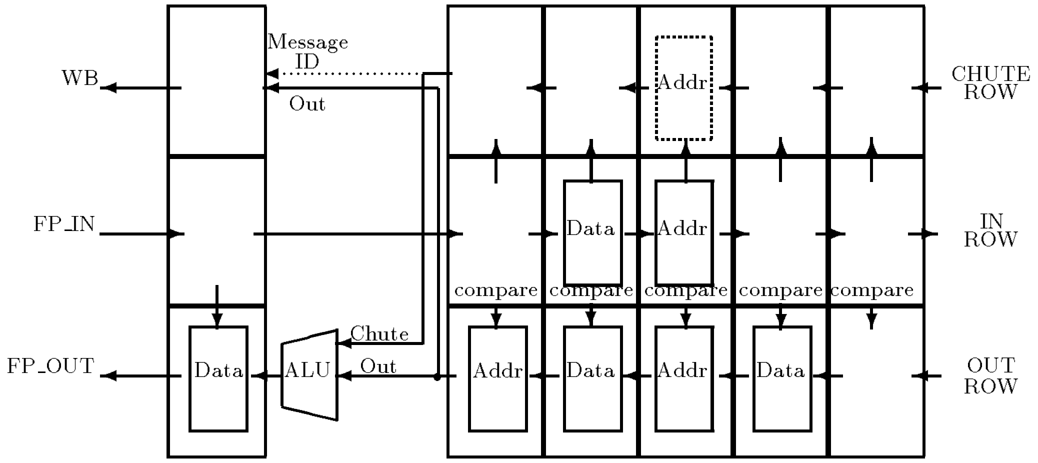


Figure 4.3: Design of systolic combining queue.

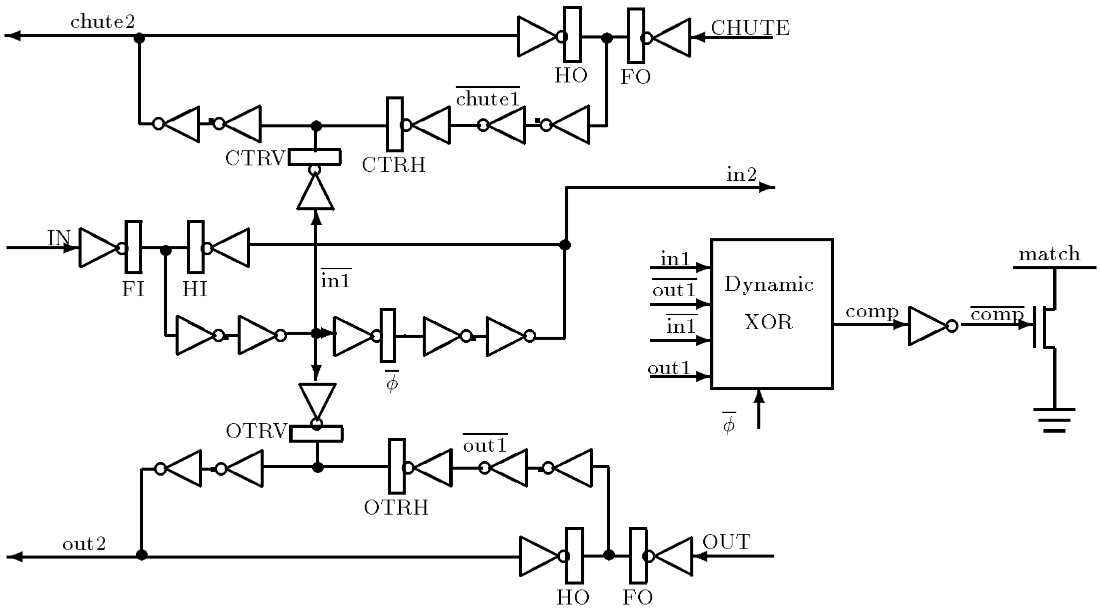


Figure 4.4: Schematic of a single cell of the combining queue in the forward path component.

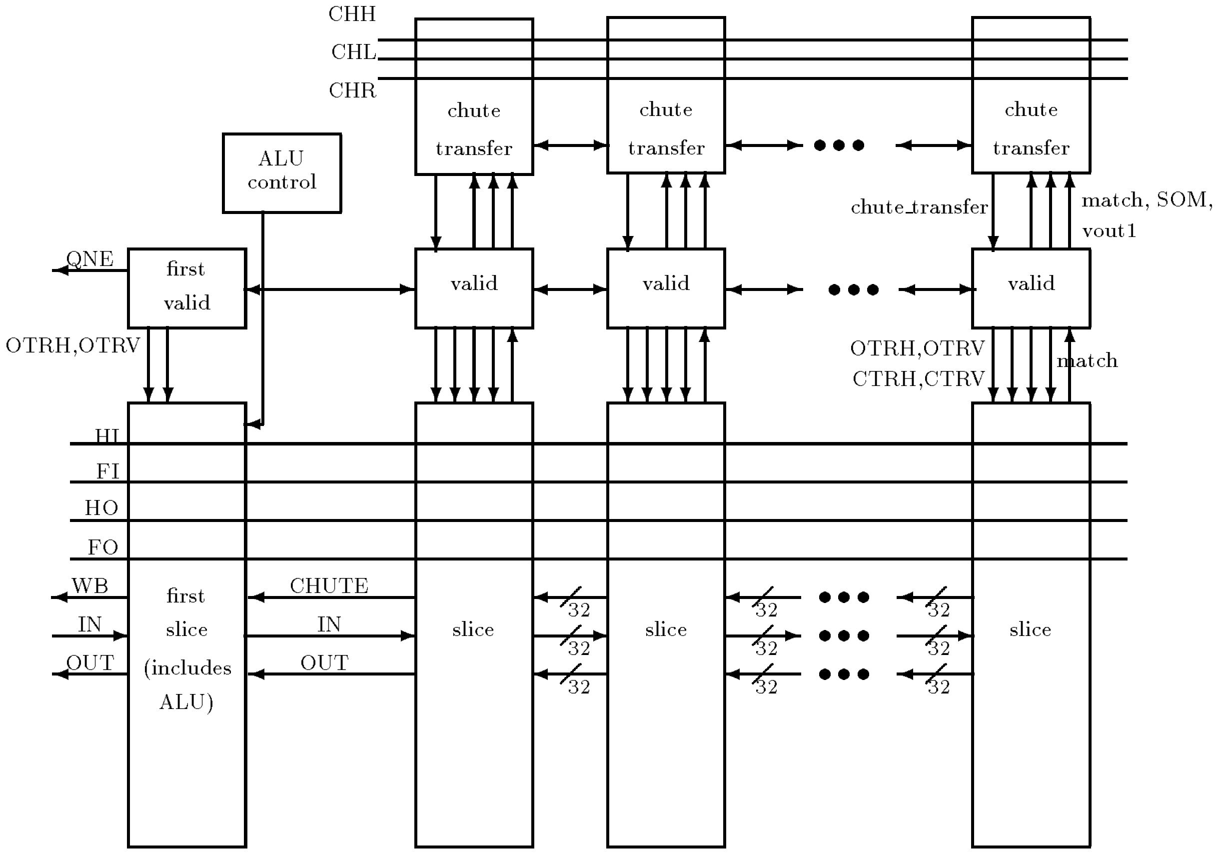


Figure 4.5: Block diagram of a combining queue implementation.

$$
\begin{array}{rcl}
\mathrm{in1}(j) & := & \text{if} \quad \text{in\_moving then in2}(j-1) \text{ else in2}(j), \\
\mathrm{out1}(j) & := & \text{if} \quad \text{out\_moving then out2}(j+1) \text{ else out2}(j), \\
\mathrm{chute1}(j) & := & \text{if} \quad \text{out\_moving then chute2}(j+1) \text{ chute2}(j), \\
\mathrm{valid1}(\mathrm{IN},\ j) & := & \text{if} \quad \text{in\_moving} \\
 & & \qquad \text{then valid2}(\mathrm{IN},\ j-1) \\
 & & \qquad \text{else valid2}(\mathrm{IN},\ j), \\
\mathrm{valid1}(\mathrm{OUT},\ j) & := & \text{if} \quad \text{out\_moving} \\
 & & \qquad \text{then valid2}(\mathrm{OUT},\ j+1) \\
 & & \qquad \text{else valid2}(\mathrm{OUT},\ j), \\
\mathrm{valid1}(\mathrm{CHUTE},\ j) & := & \text{if} \quad \text{out\_moving} \\
 & & \qquad \text{then valid2}(\mathrm{CHUTE},\ j+1) \\
 & & \qquad \text{else valid2}(\mathrm{CHUTE},\ j), \\
\mathrm{in2}(j) & := & \mathrm{in1}(j), \\
\mathrm{out2}(j) & := & \text{if} \quad \text{valid1}(\mathrm{OUT},\ j) \text{ then out1}(j) \text{ else in1}(j), \\
\mathrm{chute2}(j) & := & \text{if} \quad \text{valid1}(\mathrm{CHUTE},\ j) \text{ then chute1}(j) \text{ else in1}(j), \\
\mathrm{valid2}(\mathrm{IN},\ j) & := & \text{valid1}(\mathrm{IN},\ j) \wedge \text{valid1}(\mathrm{OUT},\ j) \wedge \\
 & & \qquad (\ \neg\text{chute\_transfer1}(j) \vee \text{valid1}(\mathrm{CHUTE},\ j)), \\
\mathrm{valid2}(\mathrm{OUT},\ j) & := & \text{valid1}(\mathrm{IN},\ j) \vee \text{valid1}(\mathrm{OUT},\ j), \\
\mathrm{valid2}(\mathrm{CHUTE},\ j) & := & (\text{valid1}(\mathrm{IN},\ j) \wedge \neg\text{chute\_transfer1}(j)) \\
 & & \qquad \vee\ \text{valid1}(\mathrm{CHUTE},\ j).
\end{array}
$$

Figure 4.6: Combining queue transitions for slot j.

section 3.5. OTRH, OTRV and CTRH, CTRV are the analogues of TRH, TRV for OUT and CHUTE, respectively. The CHUTE row, like the OUT row, is set from the IN row whenever the slot in the CHUTE row is empty, so that the setting of the qualified clocks CTRH and CTRV does not depend on the match logic. SOM (start of message) is true if the packet in the slot is an address packet. The combining queue contains eight slices, corresponding to eight slots in each of the three rows. Each slot holds a single packet of a message.

The transitions for slice j of a combining queue are shown in Figure 4.6. The chute_transfer signal flags whether data packets of a message should move into the CHUTE row or not. Chute_transfer is a somewhat complicated function of the match line for that slice and the state of the queue, including in_moving, out_moving and the parity of the cycle. The chute_transfer signal is asserted or deasserted at the start of a valid message, depending on whether or not a match has occurred. The chute_transfer signal must follow the movement of the each remaining packet in the message as it proceeds along the IN row, so that the packet is transfered to the CHUTE row if chute_transfer is asserted, and to the OUT row otherwise. If both rows are moving, the chute_transfer signal stays in place, and the transfer is made at the point where the match was detected. (see Figure 4.7). When the IN row is moving and the OUT row is not moving, the chute_transfer signal must move right every second cycle, since the packets are piling up behind the first packet (see Figure 4.8). When the IN row is not moving and the OUT row is moving the chute_transfer signal must move left every second cycle (see Figure 4.9). If neither row is moving, the chute_transfer signal again stays in place, until the OUT row starts moving. The CHH, CHR and CHL signals shown in Figure 4.5 control the movement of the chute_transfer signal between slices of the queue.

The matching and chute transfer logic can be modified easily for any even number of packets in a message, as long as the address information needed to perform the match is contained in the first packet. If more than one packet must be examined before a combine is determined, "partial match" or "temporary valid" signals must move through both the IN and CHUTE rows following the two possible paths a message might take, until a combine has been confirmed or denied. If only two packets are needed for the address information, this confirmation can be done using only connections to neighboring slices, but if more than two packets are needed, the lead packet will not be in a slice adjacent to the slice where the combine is confirmed, so that non-local connections (anathema in a systolic design) would be required. Instead, it may be necessary, if the combining information does not fit in one or two packets, to assemble packets on entry into the queue into units large enough to hold the combining information, and disassemble these units before sending off-chip. Unfortunately, such assembly and disassembly increases minimum chip latency.

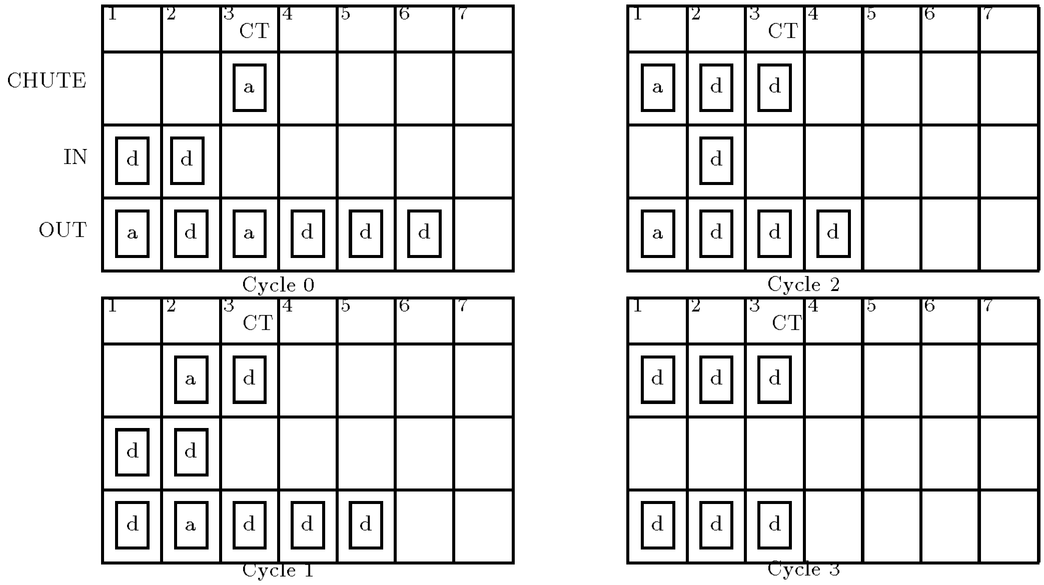


Figure 4.7: Behavior of chute transfer signal with IN and OUT both moving, when combining a 4-packet message .

4.2.2 Combining ALU

Inputs to the ALU come from the second slice of the OUT and CHUTE rows of the combining queue. The next stage output (FP_OUT in Figure 4.3) gets address packets from the OUT row, passing through the ALU unchanged; data packets on FP_OUT are the output of the ALU, which simply selects the OUT row for messages that did not combine. The wait buffer output (WB) gets message identifiers from both rows for the address packet; data packets on WB come from the OUT row without change.

To understand why correct decombining of associative fetch-and-ϕ operations can always be performed when the wait buffer receives the data packet from the OUT row, we show that in our design the effect of combining message M_{OUT} from the OUT row and message M_{CHUTE} from the CHUTE row is as if M_{OUT} arrived at memory first, followed by M_{CHUTE}. Since M_{OUT} arrived at the switch before M_{CHUTE}, this is the "natural" order. Let $data_{OUT}$ be the data value of M_{OUT}, and $data_{CHUTE}$ be the data value of M_{CHUTE}. As described in [57], if ϕ is the associative operation to be performed at memory, then $\phi(data_{OUT}, data_{CHUTE})$ is sent to memory as a result of combining, and the operation

$$\phi(X, \phi(data_{OUT}, data_{CHUTE})) = \phi(\phi(X, data_{OUT}), data_{CHUTE}) \tag{4.1}$$

replaces the value X in the memory location. X is returned as a response to M_{OUT}; in our implementation, since the address packet for the combined message came from M_{OUT}, the response from memory passes through the return path stage where it combined without modification. After decombining, $\phi(X, data_{OUT})$ must be returned as a response to M_{CHUTE}. The presence of $data_{OUT}$ in the wait buffer allows this decombining operation to be done.

The fetch-and-store operation can be considered a fetch-and-ϕ operation where ϕ is the projection of the second operand ($\pi_2(x, y) = y$), since the last store made is the value retained in memory. Thus $\pi_2(data_{OUT}, data_{CHUTE}) = data_{CHUTE}$ is the correct value to send to memory in order for the operations to be serialized as M_{OUT} before M_{CHUTE}. Although it would require only three functions in the ALU on the forward path if $data_{OUT}$ were sent to memory instead of $data_{CHUTE}$, in that case $data_{CHUTE}$ would need to be sent to the wait buffer. Furthermore, the address packet for the combined message would have to come from M_{CHUTE} in order for the response from memory to pass through the return path stage without modification.

Except for the conditional fetch-and-store operations, all operations could be implemented with a three-

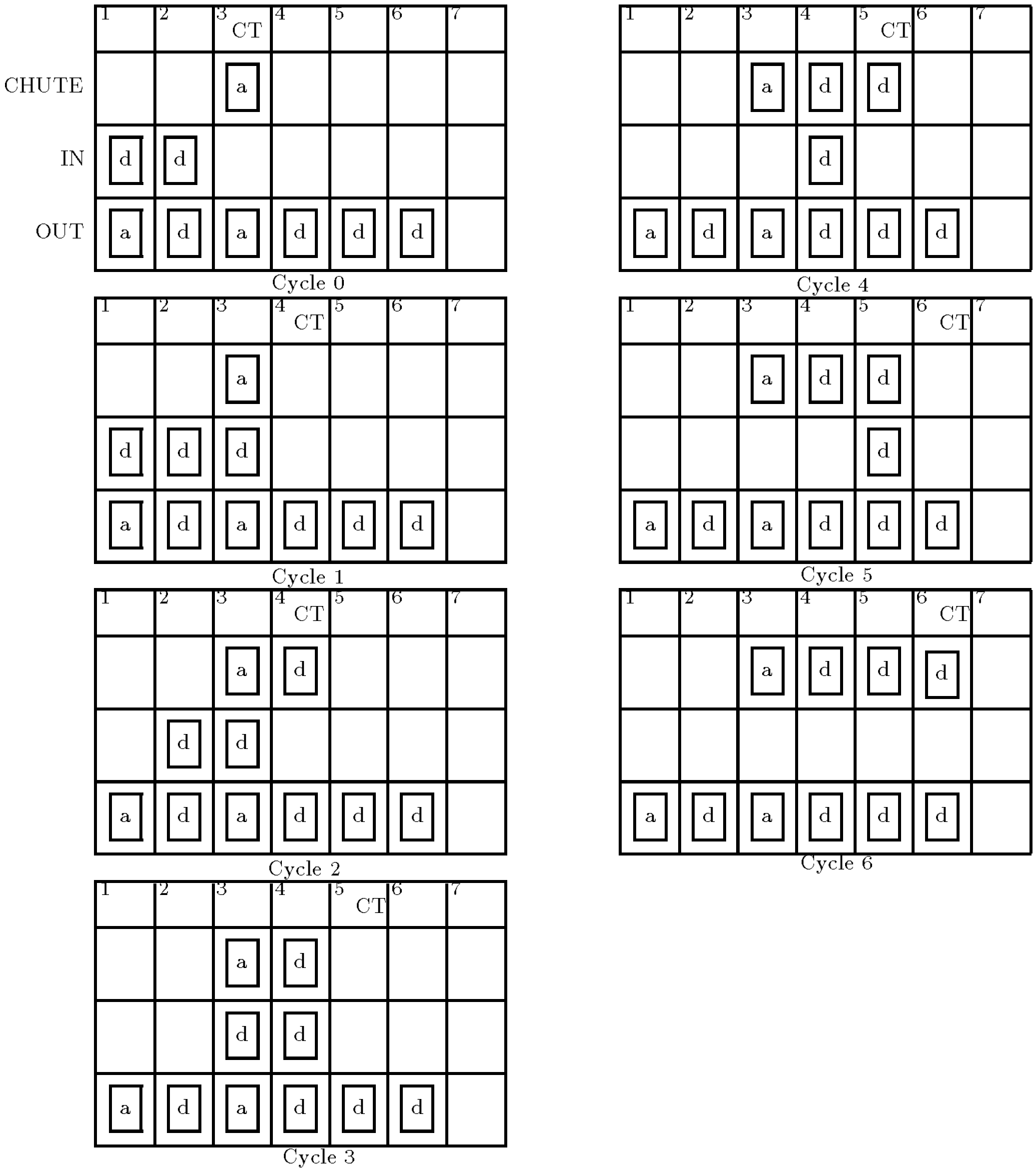


Figure 4.8: Behavior of chute transfer signal with IN moving and OUT not moving, when combining a 4-packet message.

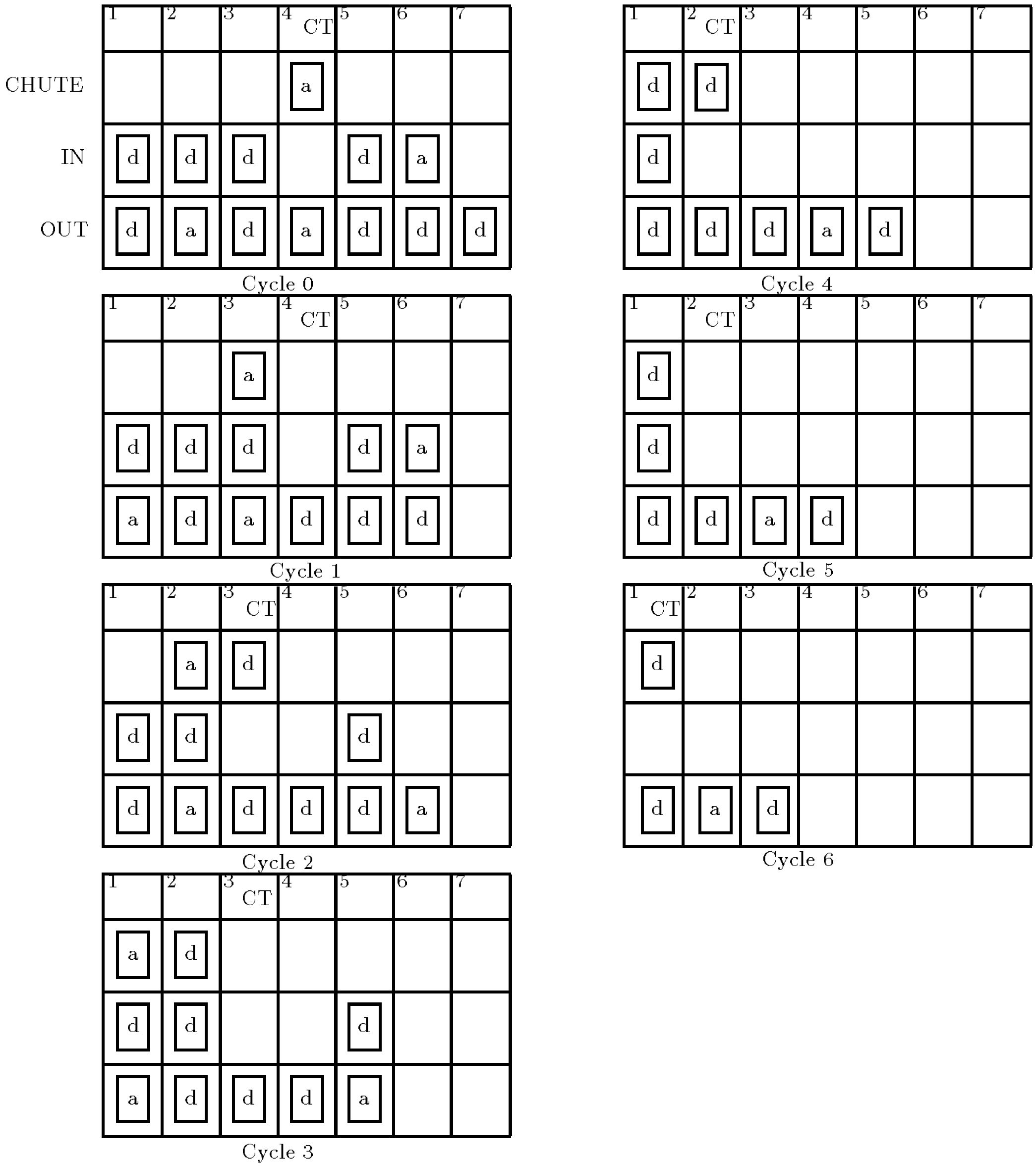


Figure 4.9: Behavior of chute transfer signal with OUT moving and IN not moving, when combining a 4-packet message.

func0	func1	Operation
0	0	ADD
0	1	SELECT Out
1	0	SELECT Chute
1	1	OR

Table 4.2: Control signals for ALU operations.

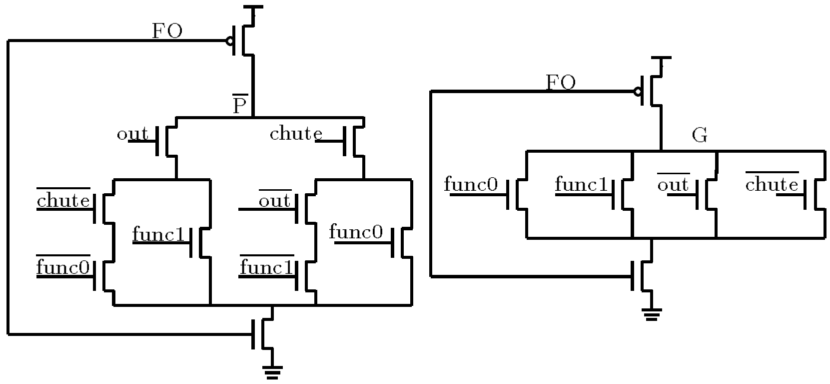


Figure 4.10: Logic to produce propagate and generate signals.

function ALU with the address packet for the combined message always coming from M_{CHUTE} and $data_{CHUTE}$ always going to the wait buffer, at the cost of making the serialization order of the messages the opposite of the order in which they arrived at the switch. We require that the operand of a conditional fetch-and-store *not* satisfy the condition. Thus, only the first conditional fetch-and-store succeeds and receives the return value from memory; all others combined with this store must receive the value of the successful store. This means that, for correct decombining, the same value must be sent to memory and saved in the wait buffer. Since for regular fetch-and-store *different* values must be sent to memory and saved in the wait buffer, no scheme to implement all these operations can have both a three-function ALU and the same data path to the wait buffer, independent of op code. We have chosen to implement the "natural" serial order with a four-function ALU.

Our four-function ALU has binary carry look-ahead implemented with Domino CMOS precharged logic. Pre-charged gates on phase 1 compute propagate and generate signals using the logic shown in Figure 4.10. The signals func0 and func1 represent the op codes as shown in Table 4.2.

During phase 1, the propagate signals, which are inputs to phase 2 dynamic gates at all levels of the carry tree, are set up. At each level, the generate signal to the next level $GG := G_1 \vee (G_0 \wedge P_1)$, and the carry signals $C_0 := CC$ and $C_1 := G_0 \vee (CC \wedge P_0)$ are computed by Domino CMOS gates on phase 2. Multi-output gates are used to shorten the carry chain; see Figure 4.11. The design is similar to that in [64]. All of the logic is done double rail; the final stage is a static exclusive-or of the carry and propagate signals.

4.3 Return path component design

The return path component in the MOSIS packaging contains a wait buffer (associative memory array that contains the data from a pair of combined messages), an ALU, and two non-combining queues (Figure 4.12). Each of these components was fabricated and tested individually before completing the single-chip return path component design.

Decombining messages requires an associative matching. For good performance, this matching must be done in parallel with insertion of arriving messages into the output queue of the RPC. Recall from the

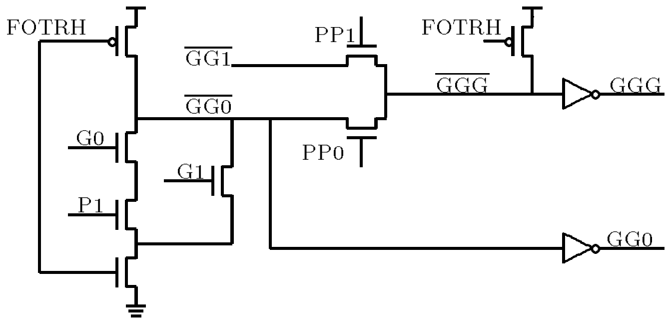


Figure 4.11: Multiple output Domino CMOS gate in carry chain.

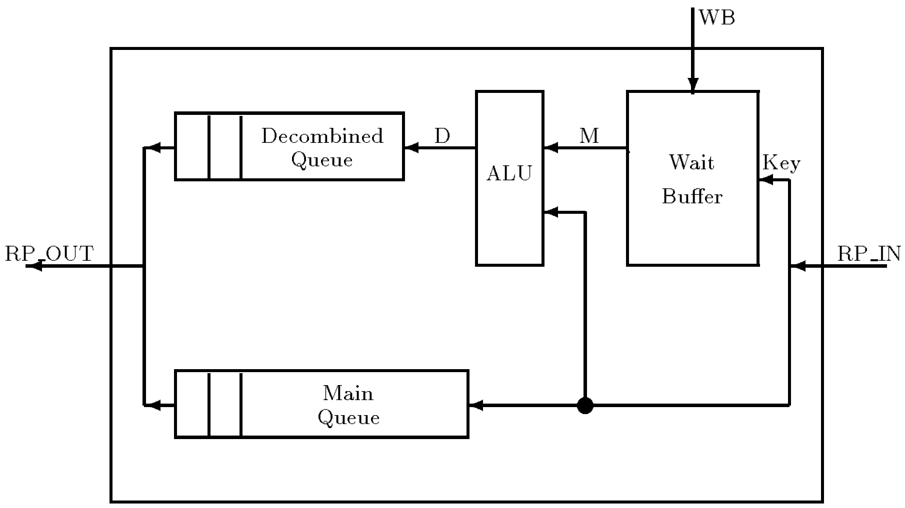


Figure 4.12: Block diagram of a return path component.

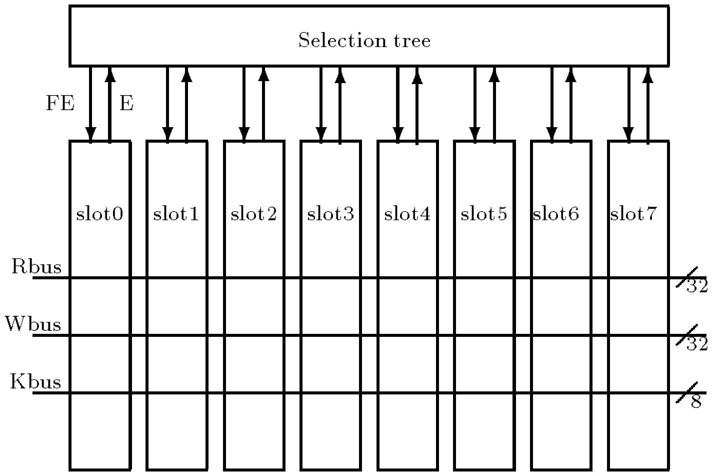


Figure 4.13: Block diagram of a wait buffer.

discussion in section 4.2.2 that data from the OUT row always goes to the wait buffer and the message ID from the OUT row always goes on toward memory. With correct decombining, this has the same effect as if the message from the OUT row arrived at the memory first, with the message from the CHUTE row following immediately. Thus the response from memory does not need to wait to find out whether it had combined before entering the Main Queue in Figure 4.12, since the value from memory is the correct response to the message originally from the OUT row. A delay register at the RP_IN input to the ALU holds a copy of this message until a match can be detected in the wait buffer, in case the message happens to be a response to a combined message.

A wait buffer is associated with each input/output pair, but the wait buffer input buses associated with a forward path output port can be tied together since they are driven only when the associated output port is driven, and thus will never both be driven at the same time. A message received on RP_IN starting at cycle t is sent to the main queue and also to the wait buffer where its address packet is simultaneously compared with all the messages currently in the wait buffer. If a match is found, the wait buffer asserts its *match* line during cycle $t+1$. The output of the ALU is sent to the decombined queue at cycle $t+2$ so that the two queues receive the first packets of their messages at cycles of the same parity.

The WB input to the RPC begins and ends messages on a cycle of the same parity as the FP_OUT output of the paired FPC. The input D to the Decombined Queue must have the same cycle parity as the RP_IN input, since the Main and Decombined Queues share an output. As a result, the detailed design of the RPC constrains the parity of the memory delay. As described in the next section, the internal wait buffer structure determines whether the difference between the cycle a message enters the wait buffer and the cycle it exits is even or odd. In the current implementation, the total number of cycles between the time the first packet of a message begins transmission on WB until it appears on D is even. Thus the number of cycles between the time a message begins transmission on FP_OUT and the time it begins transmission on RP_IN must be even as well. The FPCs and RPCs in the same switch must have the same cycle parity for their queues, and the memory must have a delay which is an even number of switch stages.

4.3.1 Wait buffer

The wait buffer is an associative memory that stores information sent by the FPC when combining two fetch-and-ϕs into a single request. The wait buffer inspects all responses from MMs and searches for a response to a request previously combined by the FPC. When it finds a response to such a request, it generates a second response and deletes the request from its memory. Packets are stored in return path format, with the PE/MM address field from the WB port reversed to be a MM/PE address. The structure of a wait buffer (WB) is shown in Figure 4.13.

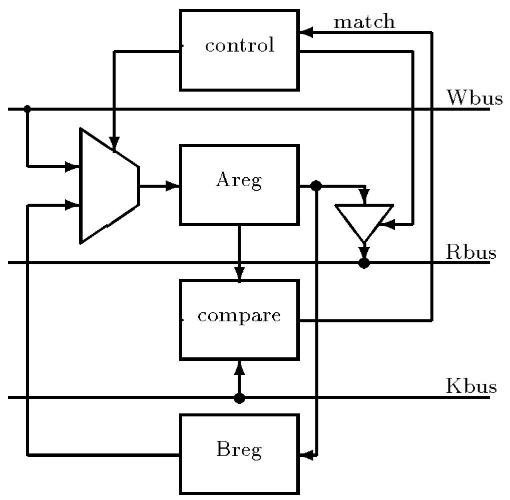


Figure 4.14: Slot of a wait buffer holding a two-packet message.

A typical message slot is shown in Figure 4.14 and consists of two registers (called Areg and Breg), compare logic, and a controller. Each register contains the data bits and a data valid (DV) bit. The registers are connected in a loop of length two, and shift at each cycle. This structure requires each message sent to the wait buffer to consist of a single address packet followed by a single data packet. Similar structures support messages containing a *fixed* even number of packets. In the combinable operations we support, no message requires more than one data packet to be stored in the wait buffer. For the fetch-and-ϕ operations, the data packet from the OUT row in the forward path is stored. For the store double operation, no value is returned, and the data packet may be set to any function of the ALU that is convenient. For the load double operation, the three data packets from the RP_IN input are selected by the ALU and placed in the decombined message; no additional values need to be stored in the wait buffer.

Each slot connects to the following buses:

- The write bus (Wbus) is used to send data to the wait buffer from the FPC and connects to a wait buffer input port.
- The read bus (Rbus) is used by each slot for transmission of its message out of the wait buffer.
- The key bus (Kbus) contains the search key received from RP_IN.

The first-empty (FE) lines are computed in a simple tree structure from the empty lines for each slot. The empty lines are ORed and the result is transmitted up the tree, with an OR performed at every node. The FE value for the parent of each sub-tree is set to true if there are no empty slots in any subtree to its left (ordering the slots from left to right). FE is true not only for the first empty leaf slot, but for all non-empty slots of lower order. This causes no problem in operation: FE causes the slot to receive a new message when it is asserted, but only if the slot is not already valid.

A schematic of a wait buffer cell is shown in Figure 4.15. HP ("Hold Packet") is asserted if the cell already has a message; otherwise AP ("Advance Packet") is asserted. The comparison logic to compute the match signal is not shown, but is similar to the short Domino CMOS chain in the combining queue cell, except that a larger dynamic XOR takes the inputs from two bits to produce the comparison signal for the pull-down transistor. CanRead is computed from the match signal at the cycle of appropriate parity, so that a read always begins with the first packet of the message; the precharged $\overline{\mathrm{R}}$ signal is shared among all slots of a wait buffer.

The wait buffer is designed under the assumption that only one message present in the wait buffer at a given stage can match a given response from memory. For regular messages, the combination of rotated PE/MM address and ORI number used for the match forms a unique key at a given stage. Since our

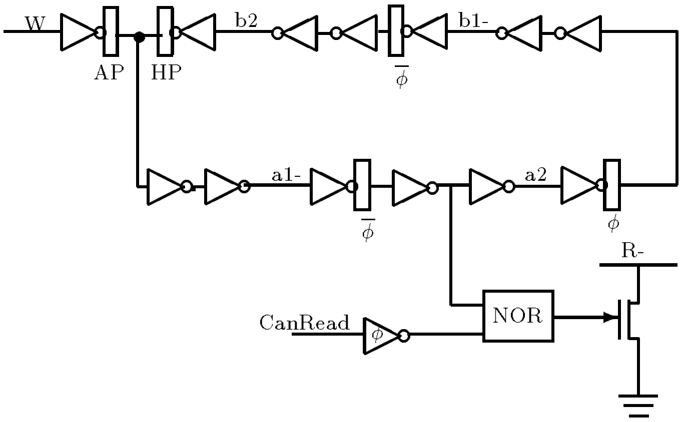


Figure 4.15: Schematic of a wait buffer cell.

processor never issues two messages with the same ORI, no false matches can occur. However, if broadcast messages are propagated through the return path by sending the message out on both ports, the PE/MM address and ORI combination can inadvertently match a message in a wait buffer. To avoid this, the valid bit on the key input to the wait buffer is lowered when a broadcast message appears on RP_IN. Thus the wait buffer never sees a broadcast message.

Since the input to the NOR that pulls down $\overline{\mathrm{R}}$ (see Figure 4.15) comes from the "a2" section of the recirculating loop, the head packet of a message will always be read out of the wait buffer an odd number of cycles after it was read in. An additional cycle is required to pass through the ALU and enter the queue, giving the total of an even number of cycles from WB to D in Figure 4.12, as mentioned previously. The input to the NOR could equally well have come from the "b2", which would have changed the parity of the cycle total and thus changed the parity of the memory delay in switch cycles.

4.3.2 Decombining ALU

A four-function ALU identical in design to the combining ALU described in section 4.2.2 is used to generate the second response to a fetch-and-ϕ operation by operating on the data packet received from a wait buffer slot and the data packet received from RP_IN. It passes address packets unchanged, except for replacing the op code with the one from RP_IN. The packet from RP_IN is delayed one cycle at the input to the ALU. The output of the ALU goes to the decombined queue on the cycle after that.

Unlike the forward path, the control signals for the ALU are a function not just of the op codes but of the SWD (store was done) bit as well. For the conditional fetch-and-store operations, if the store was done at memory, then any messages waiting to decombine should return the value stored from the wait buffer, which was stored in the memory by the combined message. If the store was not done, then the value in the message that arrived on RP_IN (representing the value in memory before this conditional fetch-and-store was attempted) should be returned by the decombined message as well. Furthermore, when a decombine occurs, the SWD bit in the message sent to the decombined queue must always be deasserted, since the decombined message always represents an unsuccessful attempt to store its own value.

4.3.3 Non-combining semi-systolic queues

The decombined queue and the main queue are multiplexed with the queues from the paired RPC for control of the output port. Each queue is a simple non-combining semi-systolic queue, like those described in Chapter 3. The main queue, like the forward path queue, has eight slices. The decombined queue is only six slices. Since our longest message is 4 packets, a queue with fewer than six slices can be full with only one message in it (see section 3.4.1), so six slices is the minimum practical size.

The logic to determine the RP_IN.DA signal is quite complex. It can be asserted only if there is at least one empty slot in each of the two queues. In addition, the decombined queue must leave room for a message coming out of the wait buffer but not yet added to the queue. Simulation of the component in a network environment was required to get it right.

4.4 System simulation and verification

A sophisticated simulation methodology ensured that the custom VLSI components functioned correctly in the system. A three-level simulator, called *susy*,[2] was written in C and used to specify and verify the behavior of components. At the highest level, a behavioral simulation specified how each switch component should perform, using high-level modeling of processor and memory module message inputs to the network and modeling the switches with standard software queues of message. The next level, similar to that of a register-transfer language (RTL), included data structures in the simulation to represent each cell in the VLSI design, so that the behavior of the systolic queues, as implemented, was fully and faithfully simulated. As the simulator executes, outputs from the behavioral simulation are checked cycle by cycle against the output from the RTL simulation. At the lowest level, routines from the *rsim* switch-level simulation library were called to simulate the circuit extracted from the VLSI layout, and all output signals of the component were checked against the signals of the register-transfer level simulation. Networks of sizes from 4 to 256 were simulated at the behavioral and register-transfer level. Switch-level simulations of individual FPC and RPC layouts were checked signal by signal against the RTL simulation. Test vectors were generated during these simulations and used in an IMS tester to test the fabricated parts. The susy simulator, called *susy*, has been jointly validated against *molasses* (described in section 2.3). Molasses is more flexible and much faster than susy, but does not faithfully simulate the internal systolic structure and flow control logic of the VLSI components. One difference is in the blocking behavior of the queues. In particular, susy models the blocking behavior of the systolic queue, lowering DA when the m^{th} slot (where m is the number of packets in the longest possible message) from the right in the OUT row of the queue is occupied (see section 3.4.1.), while molasses has fixed size queues which block when fewer than m places are left in the queue. Another difference is in the timing of insertion of a message into the wait buffers. Susy has a data structure for the chute and causes a message to be sent to the wait buffer on the same cycle that the combined message is sent to the next stage, while molasses checks for combines when a message enters a switch and places messages in the wait buffer immediately. the wait buffer

All the simulations in this section, for both molasses and susy, are done under the architectural assumptions similar to those of the *Ultra III* design[17], but slightly simplified. All messages are assumed to be two packets and the network is assumed to be able to begin transmission of a message every even cycle, as long as DA is being received from the network input port (i.e., the queues in the switches in the first stage are not full). The PE is assumed to allow 16 outstanding requests.

Bandwidth values are given in packets per cycle. The PE may not issue a message more often than every other cycle. For most simulation experiments, the probability of issuing a message on each even cycle that the queue was unblocked was varied between 10 and 100 percent. Since the messages are two packets long, the maximum bandwidth in packets that can be issued is the same as this offered load. The bandwidth in messages is half that shown in the figures.

The cycle time of the memory also places a limitation on the bandwidth. A memory cycle time of 2 places no further constraint on the bandwidth than that imposed by a two-packet message, but a memory cycle time of 4 limits the bandwidth in packets to 50 percent. For simulation runs with a memory cycle time of 4, only offered loads up to 50 percent were issued.

The hot spot percent represents the percentage of total traffic that is directed to the hot spot. For example, a 20 percent offered load with a 5 percent hot spot means that the overall rate of issuing messages to the hot spot is 0.005 messages per cycle (since a message may be issued only on even cycles) and the rate of background uniform traffic is 0.095 messages per cycle.

Round trip latency represents only waiting and transmission time in the network and at the memory. If processors are blocked, no new messages are generated, and no waiting time is added to account for the time the processor remains blocked. The effect of blocking is reflected in the reduction of bandwidth per PE.

[2] "SUper SYstolic" simulator? Or maybe just my namesake?

4.4.1 Comparison of results from the two simulators

Comparing the results of the two simulators, with molasses simulating Type B switches, allows us to determine the effective size of our semi-systolic queues. Our queues, which have storage for a maximum of 16 packets, may have as few as 8 and as many as 14 packets when they lower the DA (data accept) signal. The closest match with the simple array-based queues simulated by molasses occurs for queues with 10 packets of storage. Recall (see sections 4.2.1 and 4.3.3) that the systolic queues we have implemented have eight slices, except for the decombined queue in the return path, which has only six slices. Eight slices corresponds to 16 packets, so we are "wasting" about 6 packets of storage in the interest of the simple control logic provided by systolic queues. With 12 packets, the results from molasses, in the limiting case with 100% offered load, showed higher bandwidth and higher latency than the results from susy; with 8 packets, lower bandwidth and lower latency were shown.

Figures 4.16 to 4.24 compare round trip latency and effective bandwidth results from the two simulators with a memory cycle time of 2 and a queue size of 10 packets for molasses. Each line represents the results of a susy simulation with an offered load from 10 percent to 100 percent, at 10 percent intervals. A plus sign ('+') is used to indicate the results of molasses simulations for the same load. The latency and bandwidth results are quite close, both with and without hot spots.

In our current Ultracomputer prototype, the memory cycle time is four network cycles, not two. Advances in technology are likely to make processors and switches yet faster compared to memory cycle time. The hot spot problem is naturally accentuated by slower memories, but combining is still effective in relieving the congestion and maintaining bandwidth.

Figures 4.25 through 4.33 show round trip latency and bandwidth for offered loads from 10 percent to 50 percent. The effectiveness of combining in reducing latency is even more dramatic than with the faster memory, since the memory is much more easily saturated. For example, at the low hot spot rate of 0.5 percent, 50 percent offered load, the average round trip latency with memory cycle time of 2 increases only to 32 cycles, compared to 27 cycles with uniform traffic, even without combining. With a memory cycle time of 4, the average round trip latency at this load increases from 59 cycles with uniform traffic to 72 cycles without combining, but decreases to the uniform value when combining is done. The bandwidth increases from 26 percent without combining to 38 percent with combining.

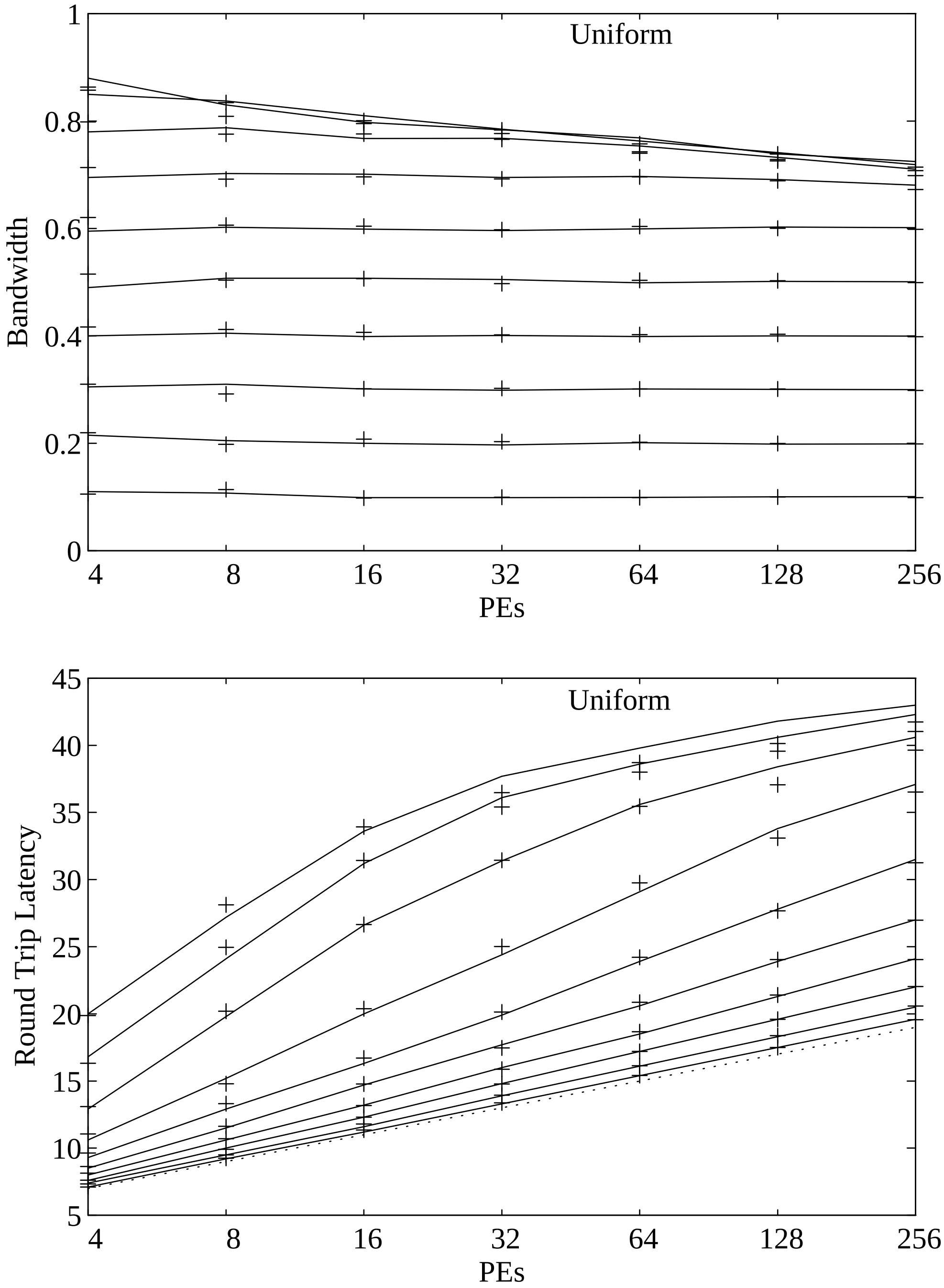


Figure 4.16: Type B switches, molasses and susy simulations, uniform traffic, memory cycle 2

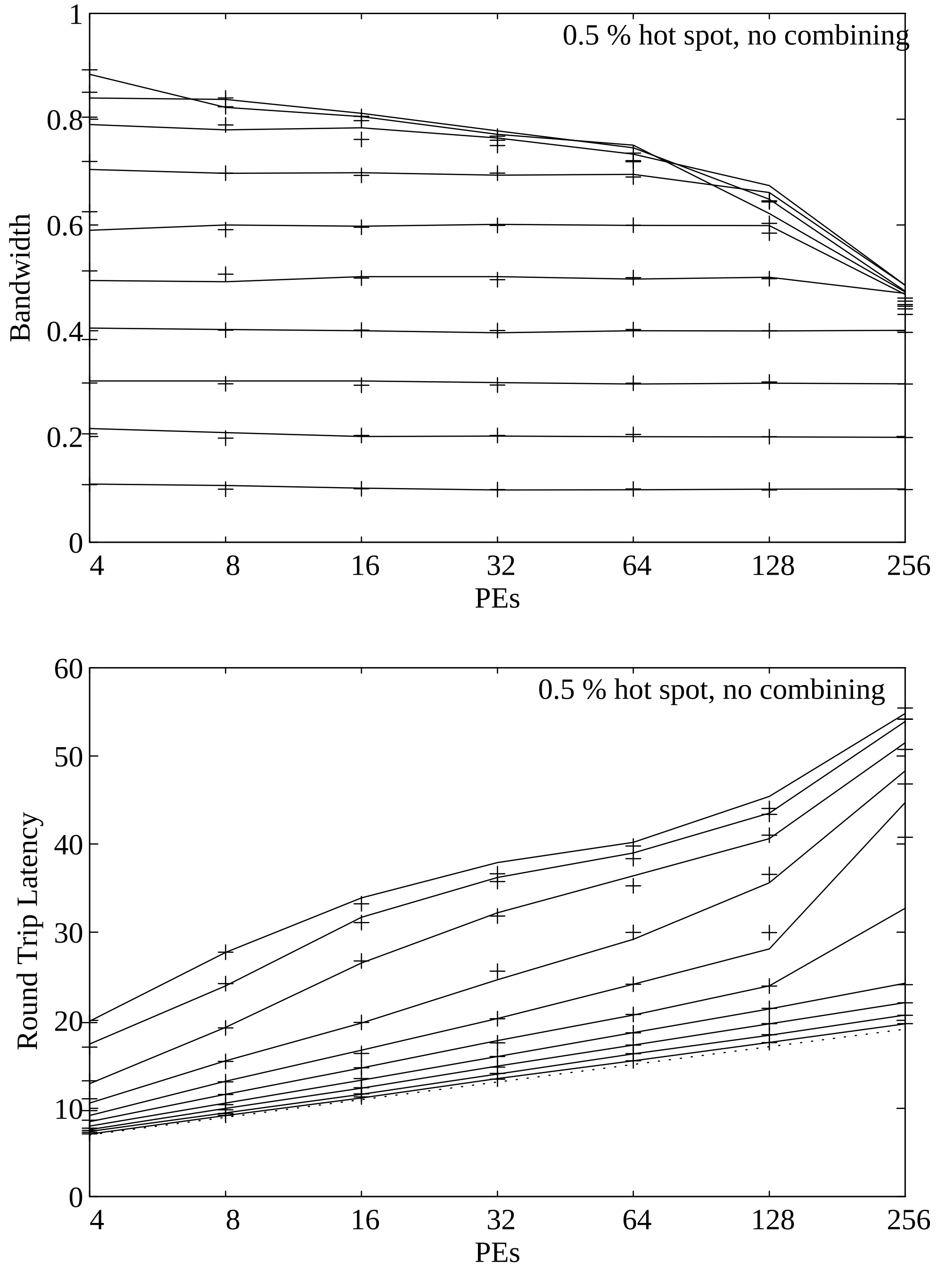


Figure 4.17: Type B switches, molasses and susy simulations, 0.5 percent hot spot, no combining, memory cycle 2.

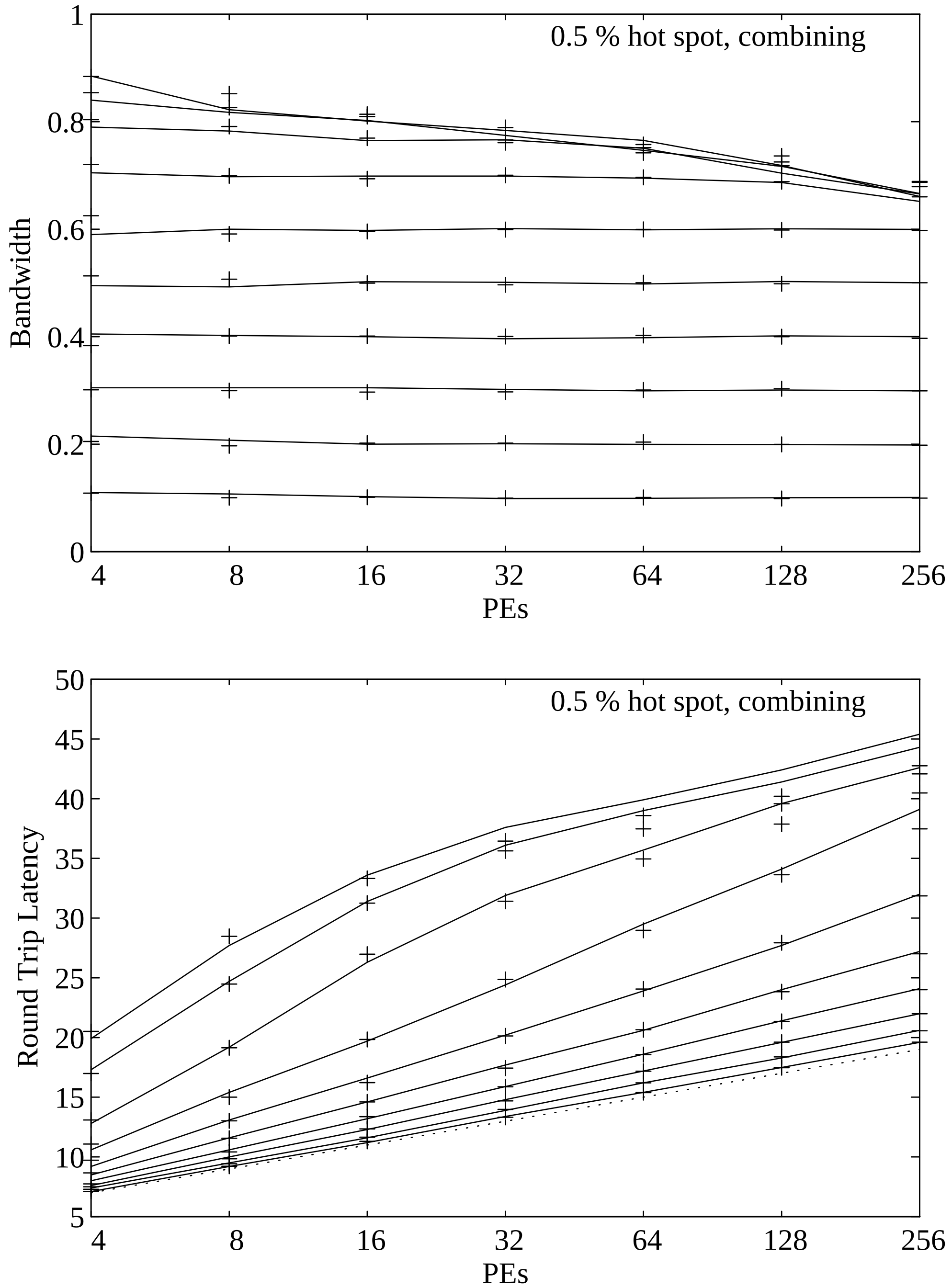
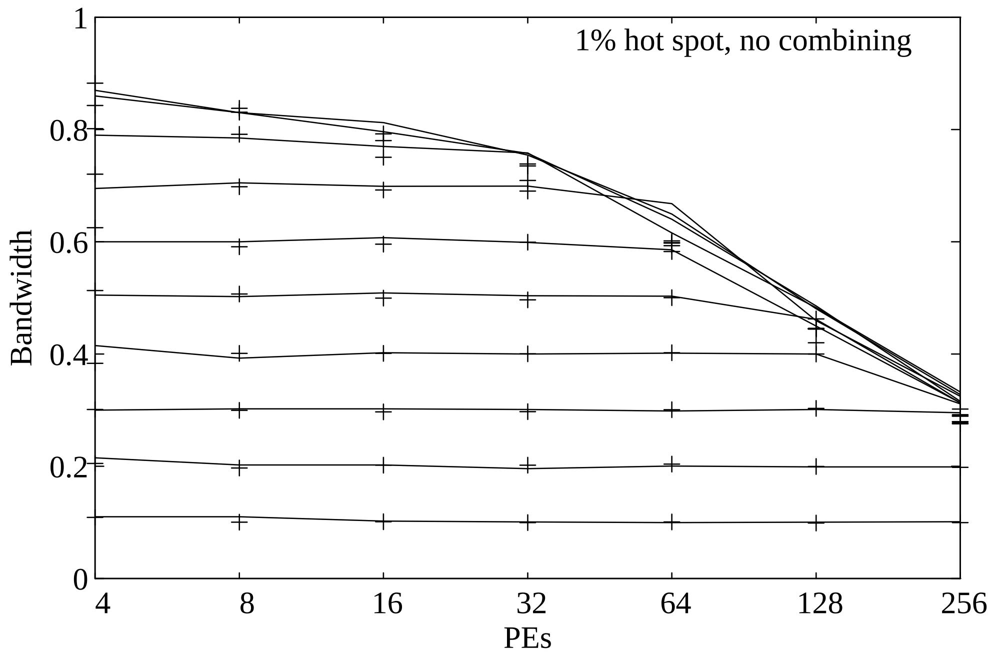


Figure 4.18: Type B switches, molasses and susy simulations, 0.5 percent hot spot, combining, memory cycle 2.



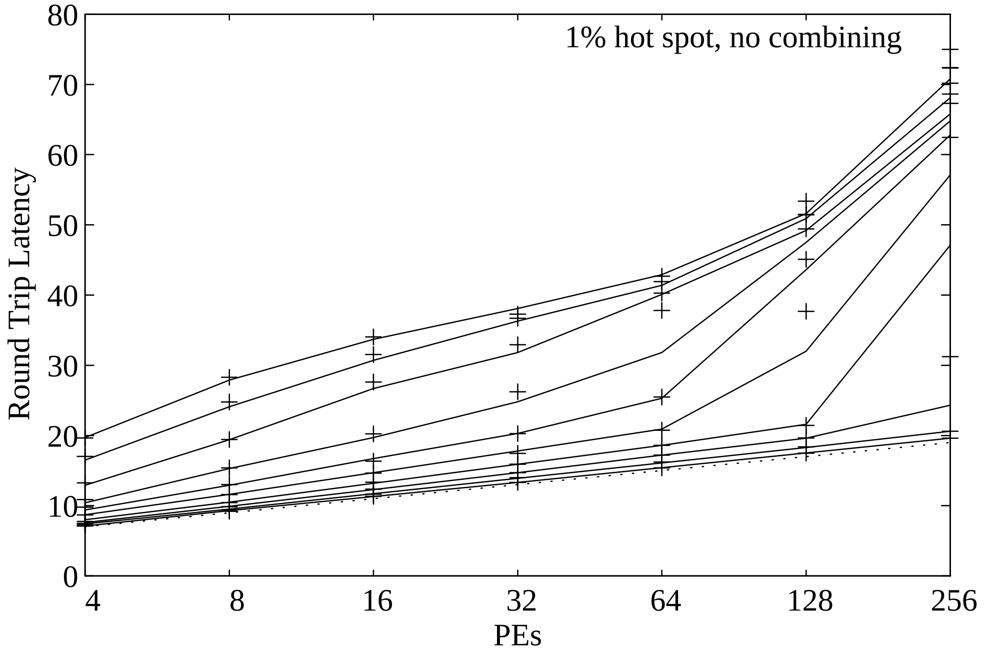


Figure 4.19: Type B switches, molasses and susy simulations, 1 percent hot spot, no combining, memory cycle 2.

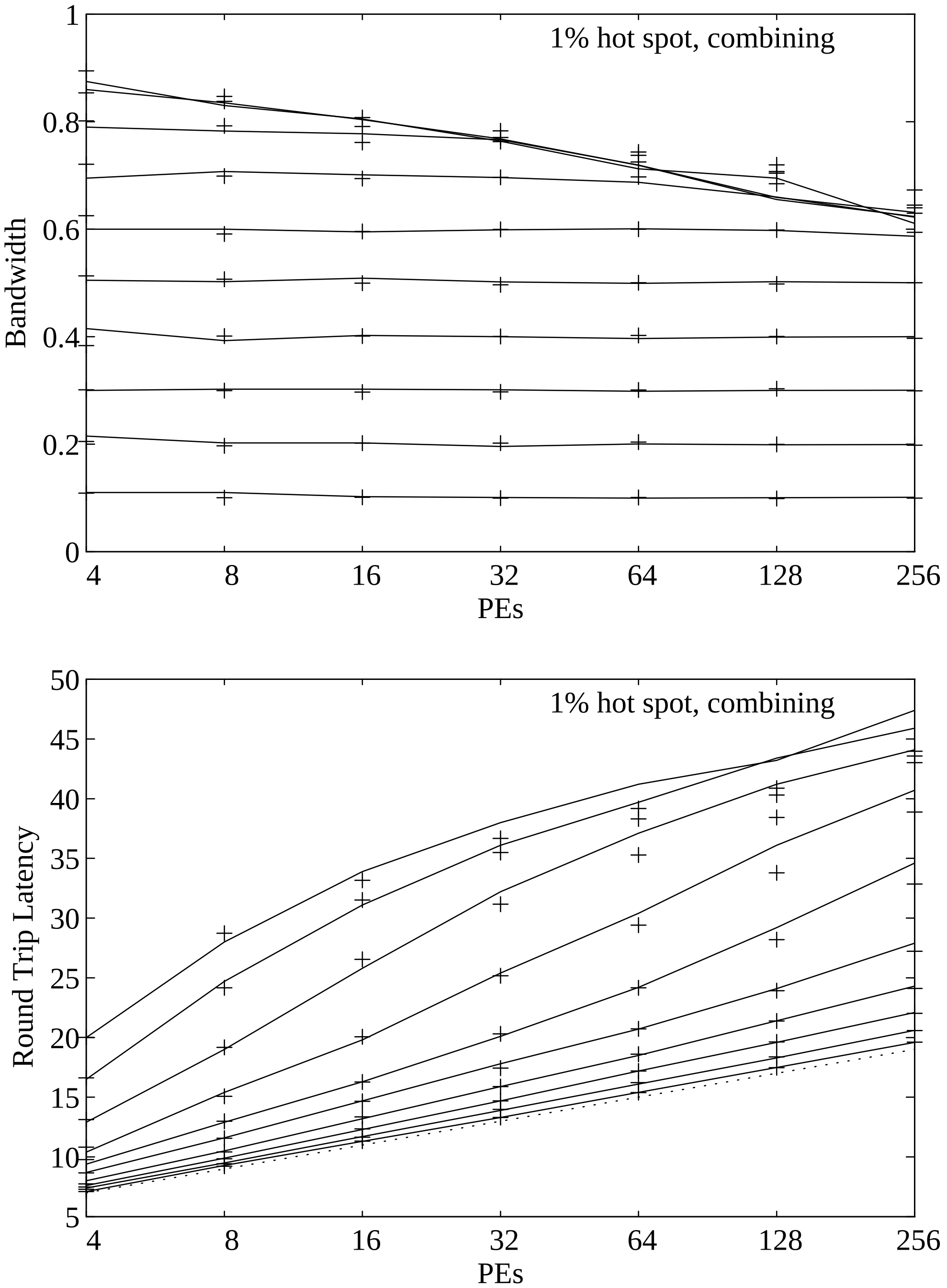


Figure 4.20: Type B switches, molasses and susy simulations, 1 percent hot spot, combining, memory cycle 2.

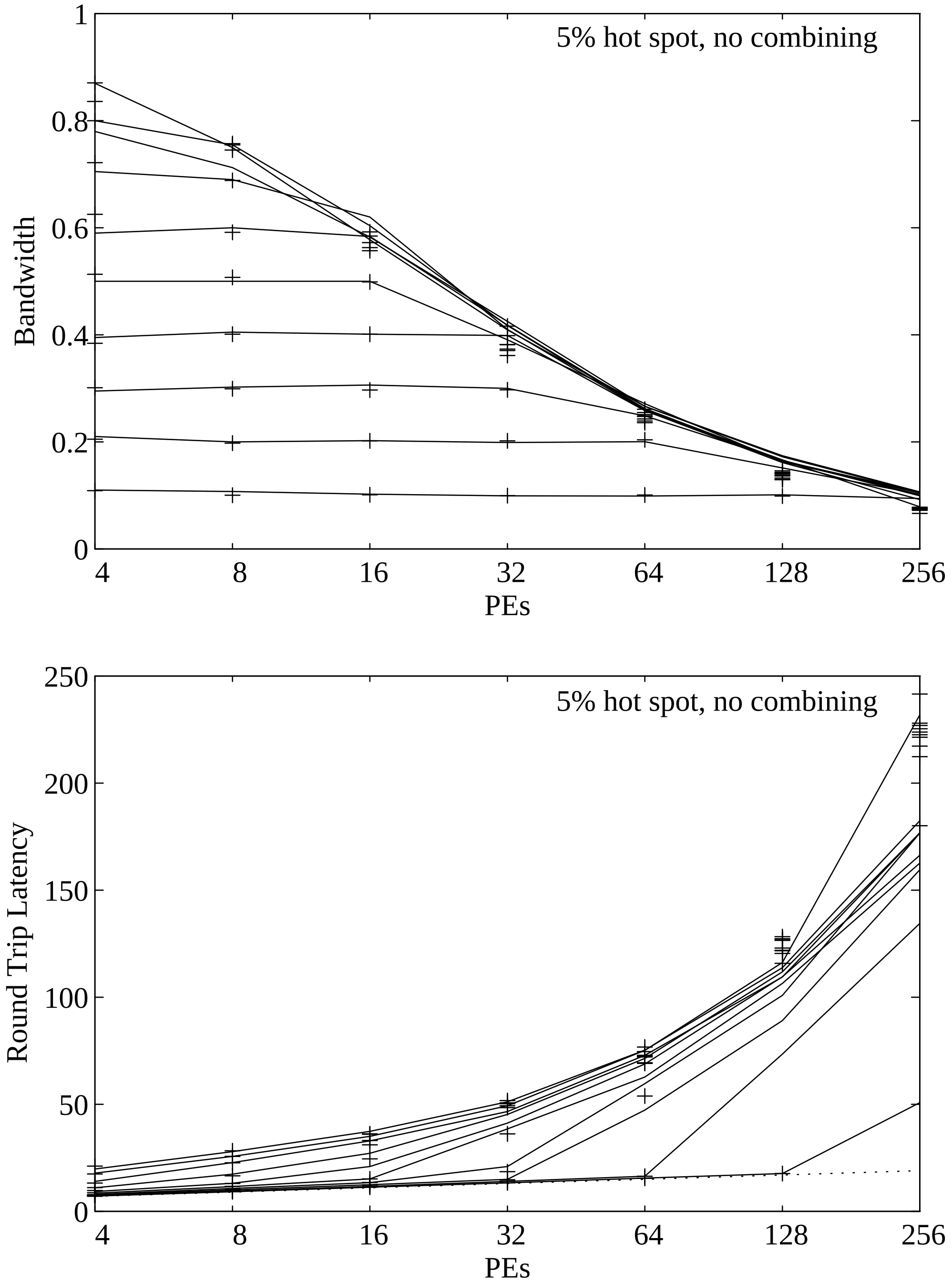


Figure 4.21: Type B switches, molasses and susy simulations, 5 percent hot spot, no combining, memory cycle 2.

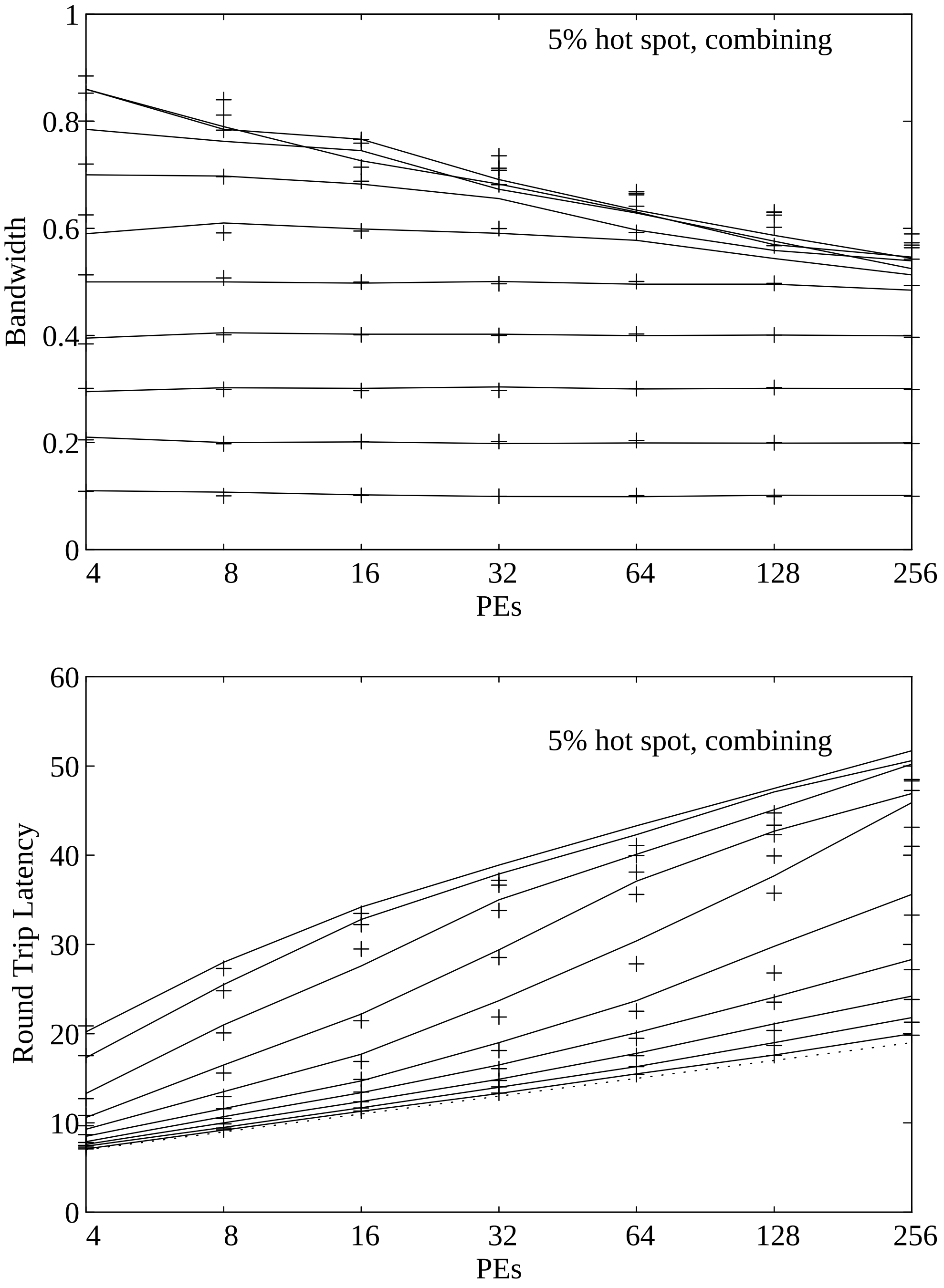


Figure 4.22: Type B switches, molasses and susy simulations, 5 percent hot spot, combining, memory cycle 2.

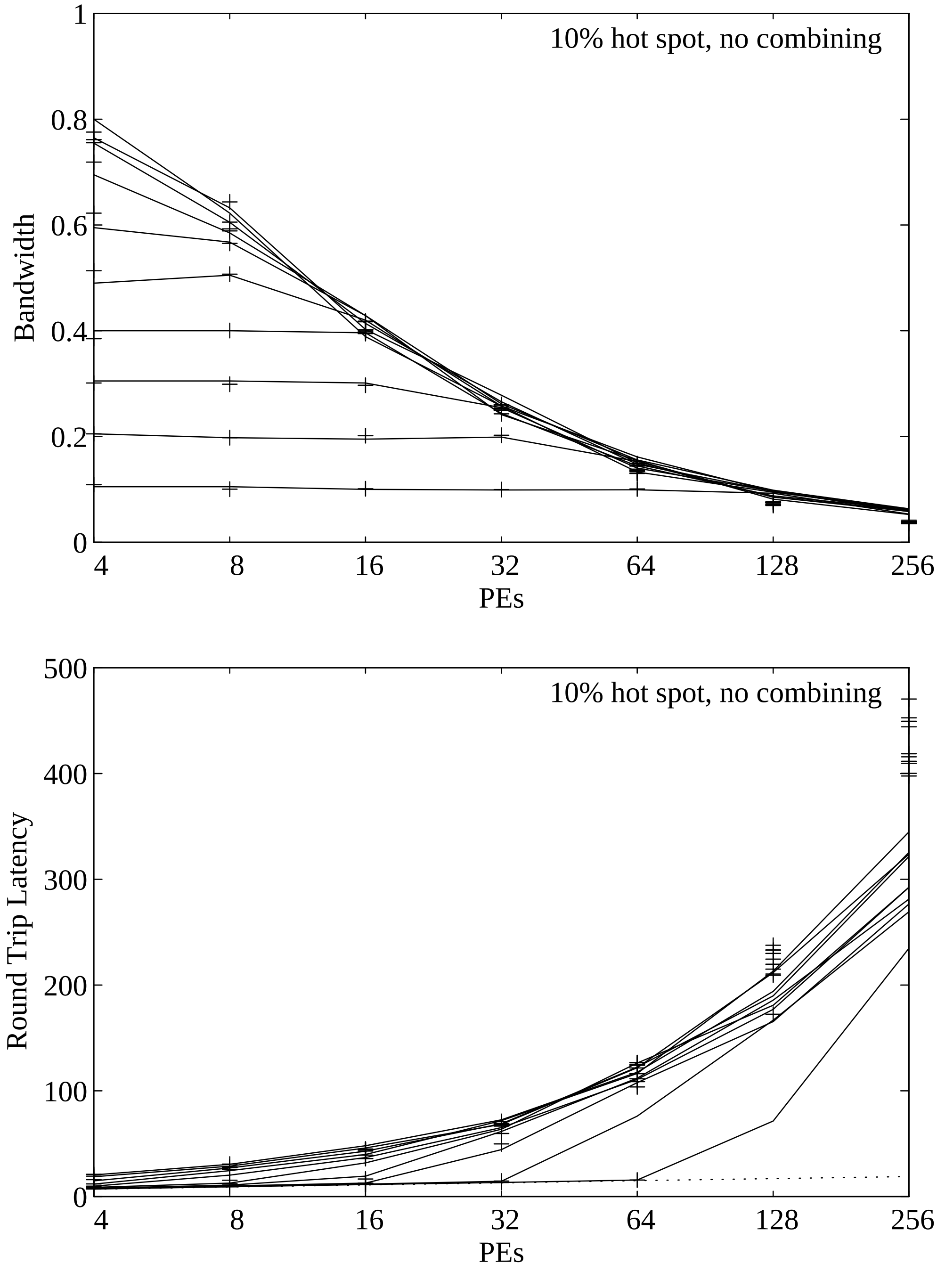


Figure 4.23: Type B switches, molasses and susy simulations, 10 percent hot spot, no combining, memory cycle 2.

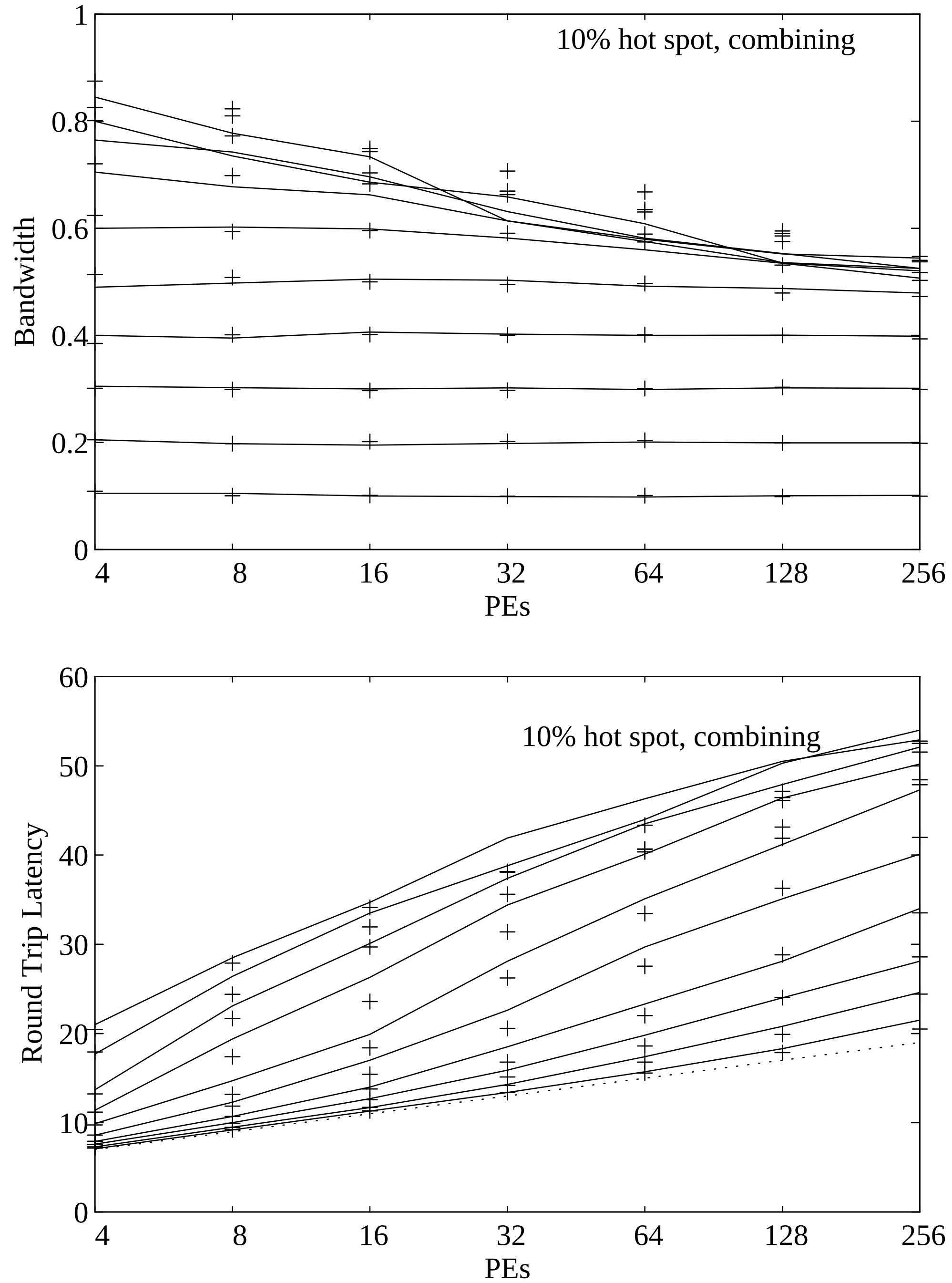
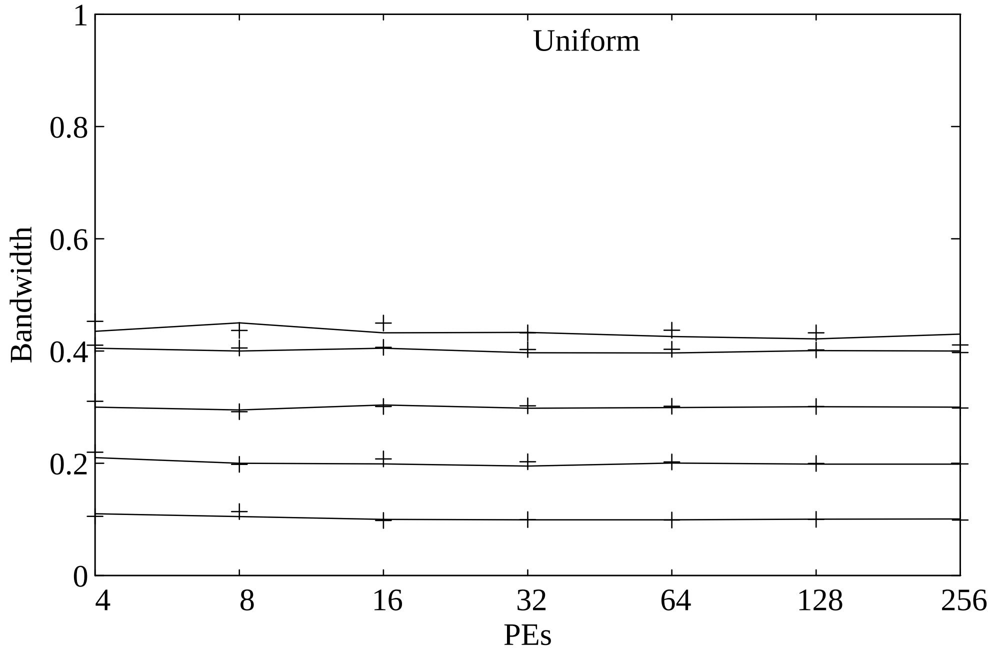


Figure 4.24: Type B switches, molasses and susy simulations, 10 percent hot spot, combining, memory cycle 2.



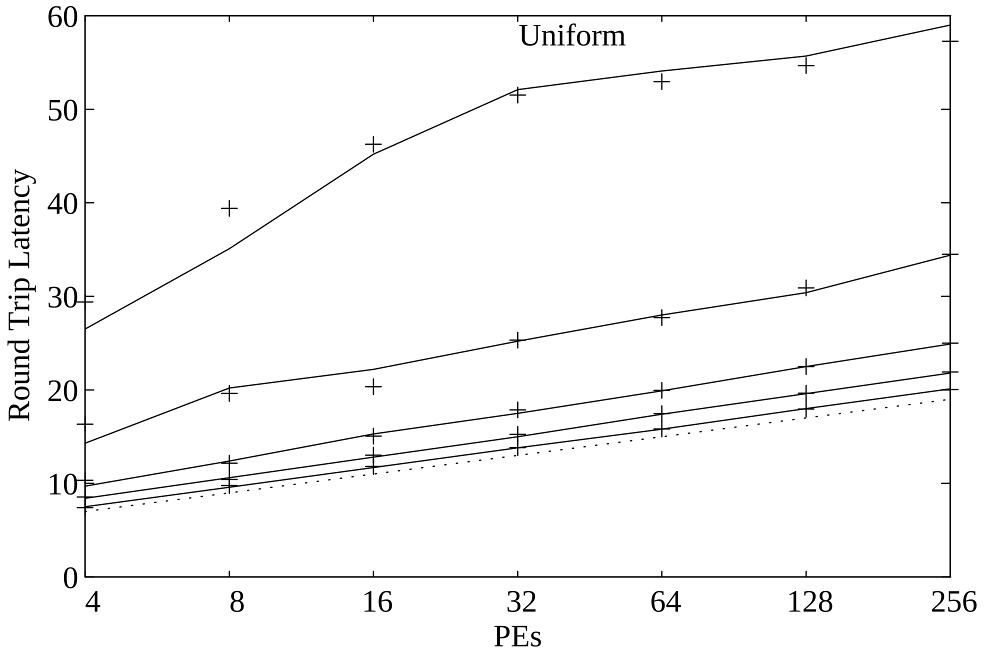
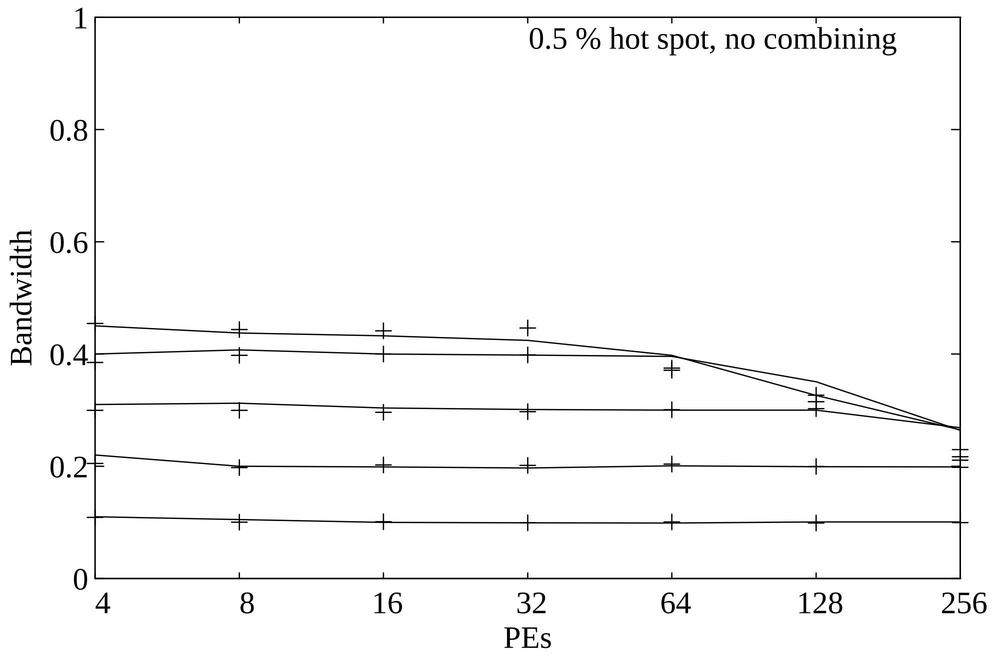


Figure 4.25: Type B switches, molasses and susy simulations, uniform traffic, memory cycle 4



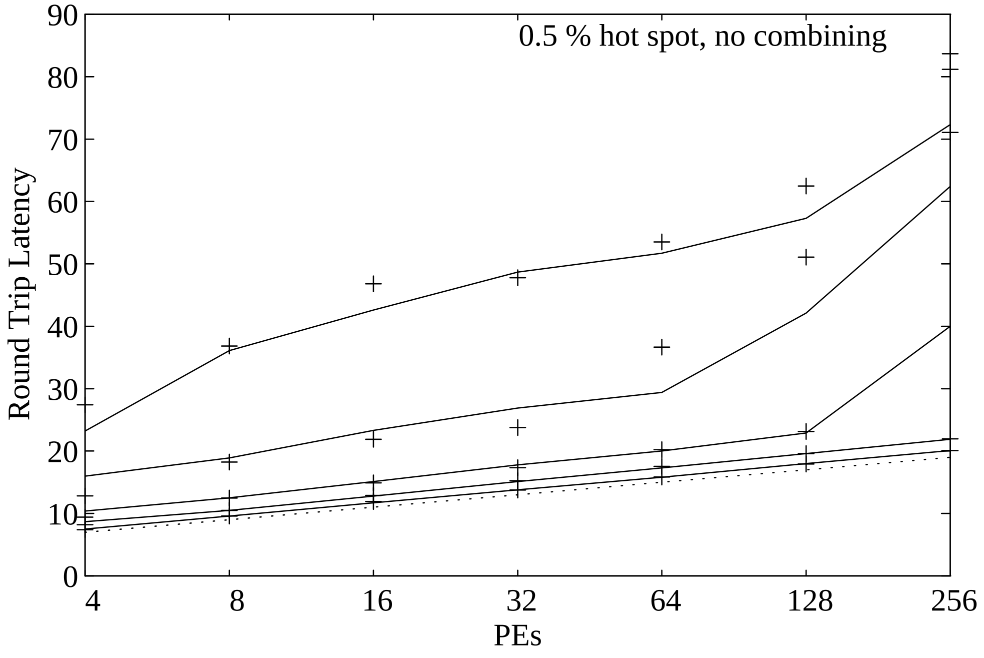


Figure 4.26: Type B switches, molasses and susy simulations, 0.5 percent hot spot, no combining, memory cycle 4.

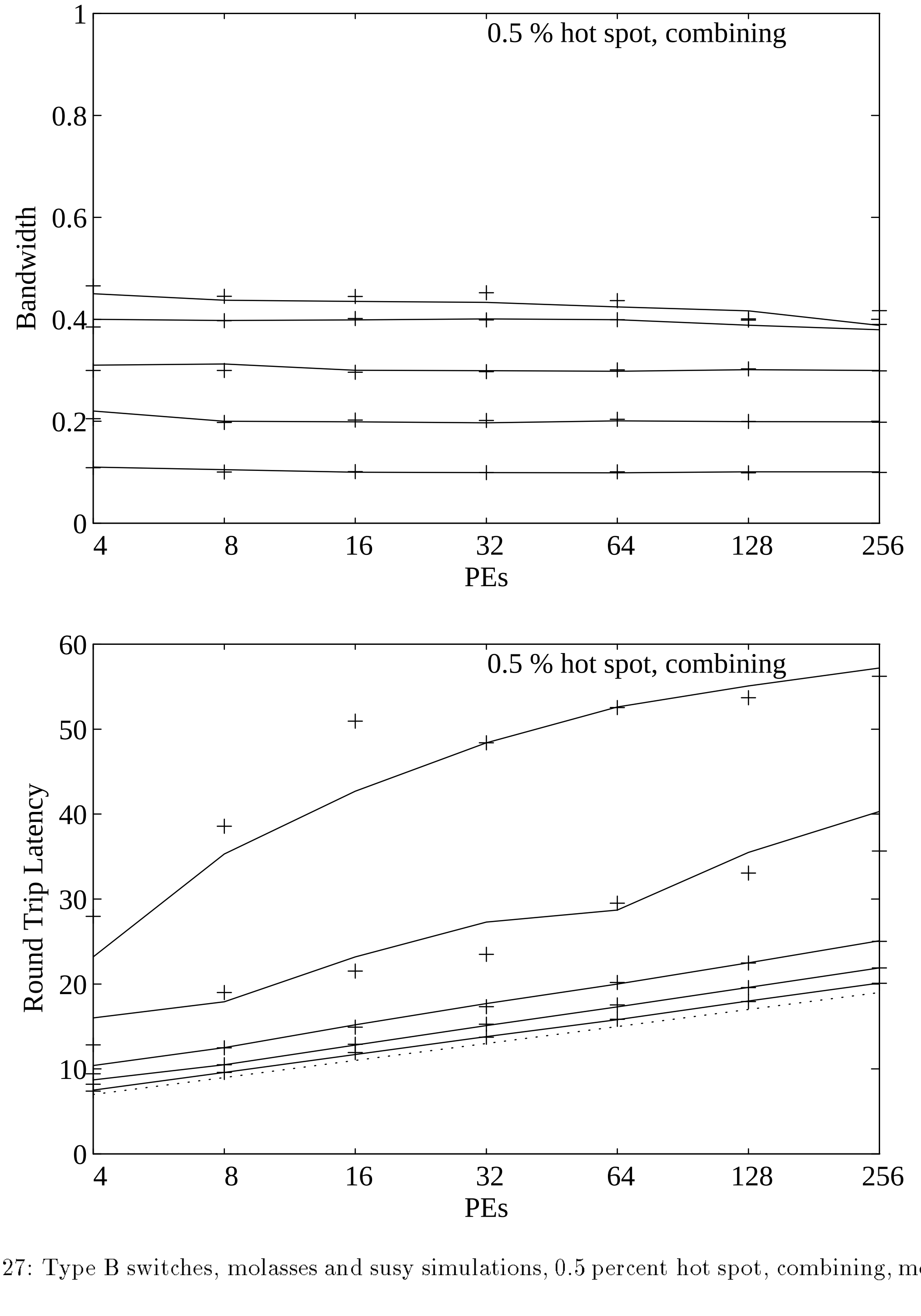
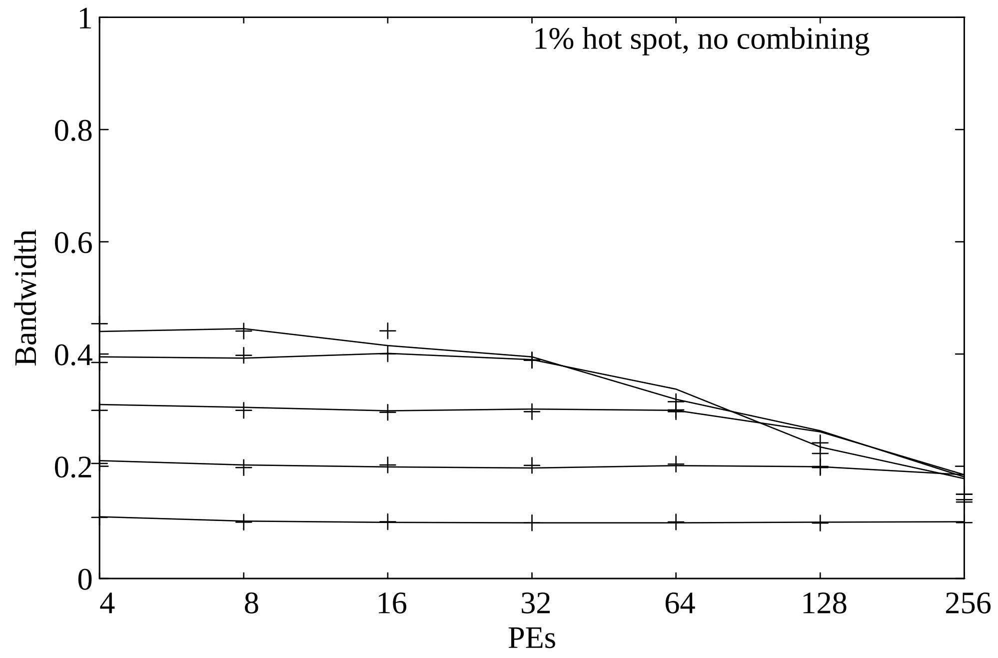


Figure 4.27: Type B switches, molasses and susy simulations, 0.5 percent hot spot, combining, memory cycle 4.



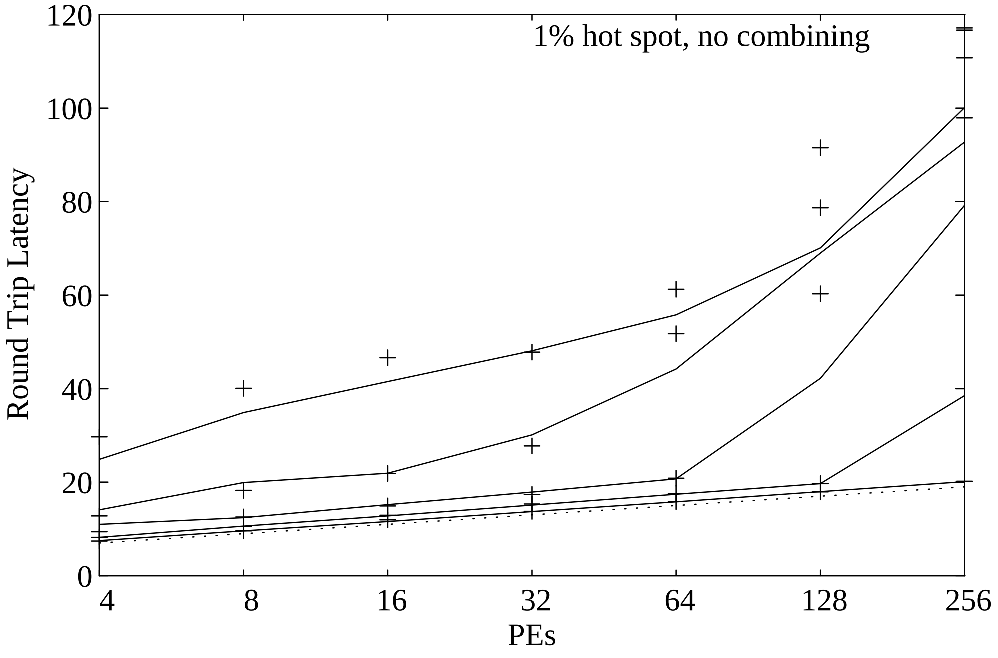


Figure 4.28: Type B switches, molasses and susy simulations, 1 percent hot spot, no combining, memory cycle 4.

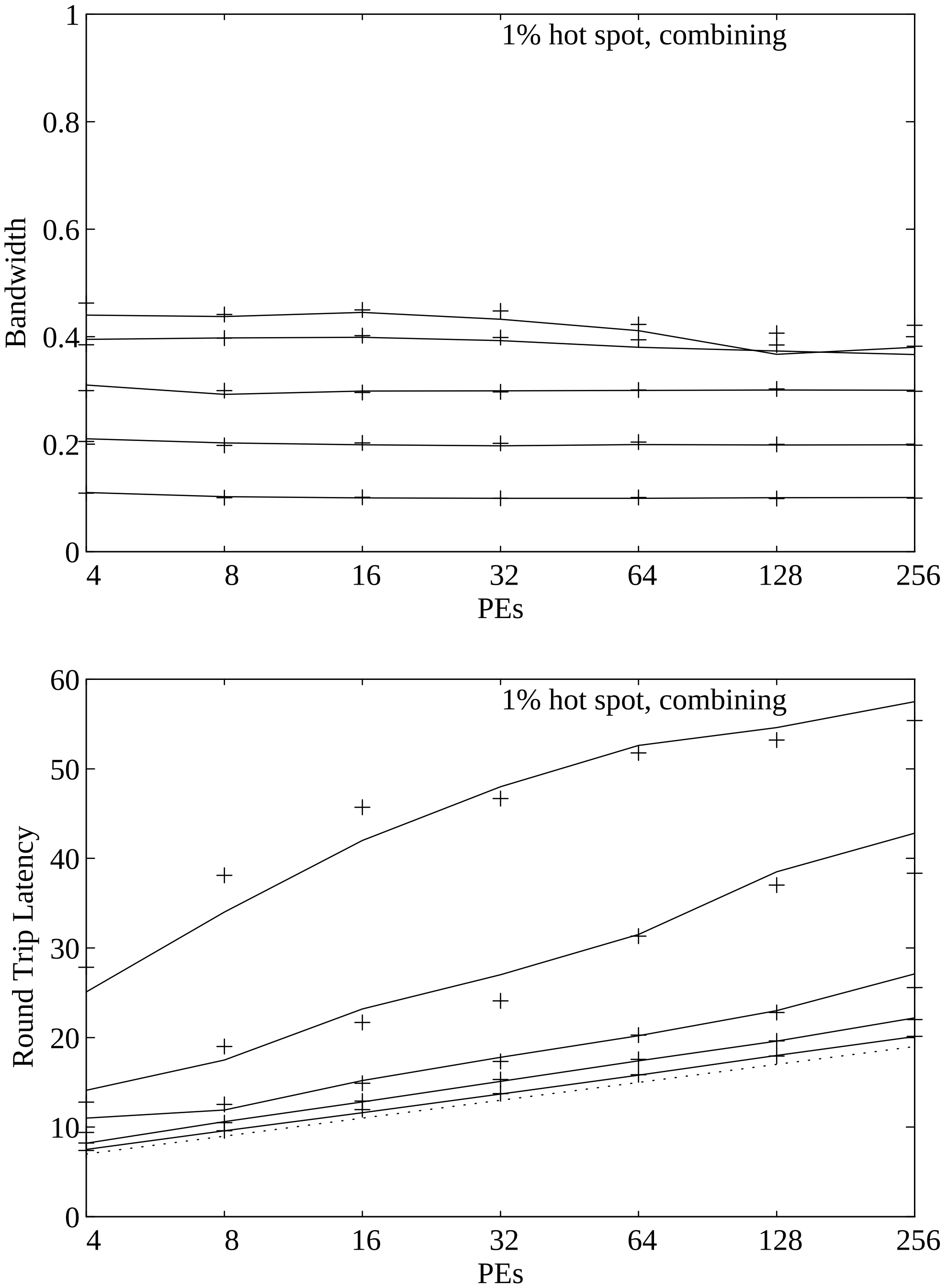


Figure 4.29: Type B switches, molasses and susy simulations, 1 percent hot spot, combining, memory cycle 4.

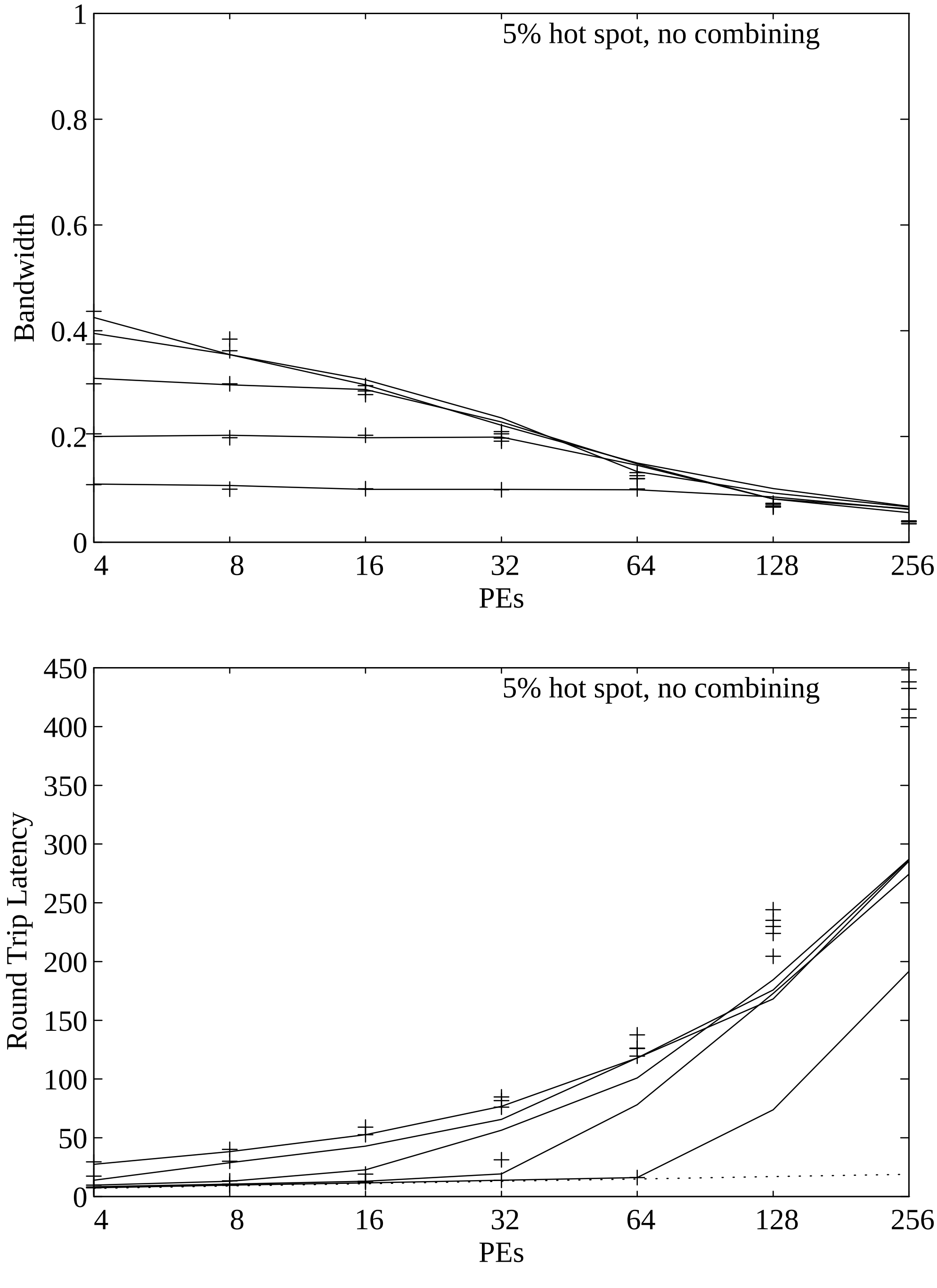


Figure 4.30: Type B switches, molasses and susy simulations, 5 percent hot spot, no combining, memory cycle 4.

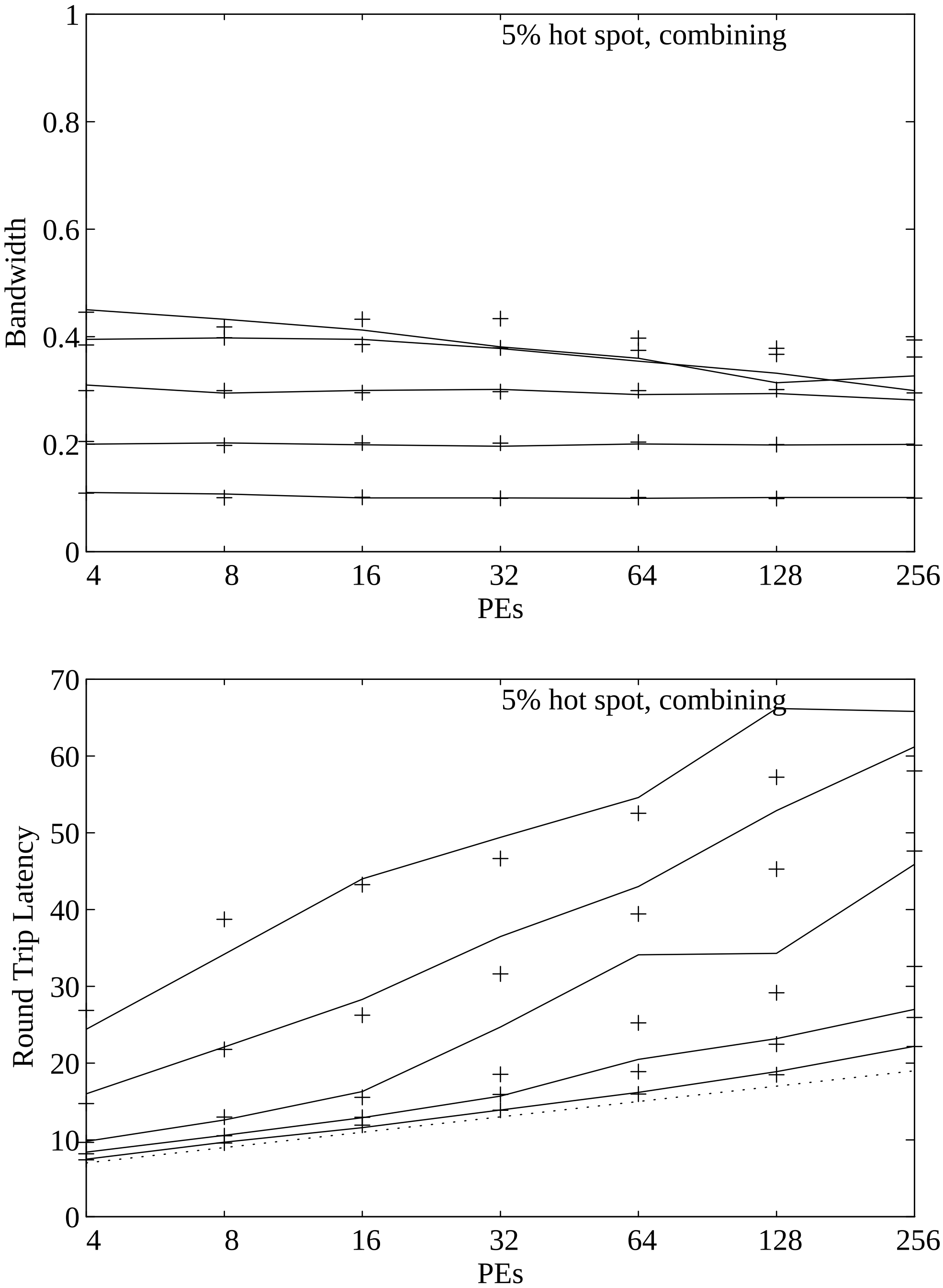


Figure 4.31: Type B switches, molasses and susy simulations, 5 percent hot spot, combining, memory cycle 4.

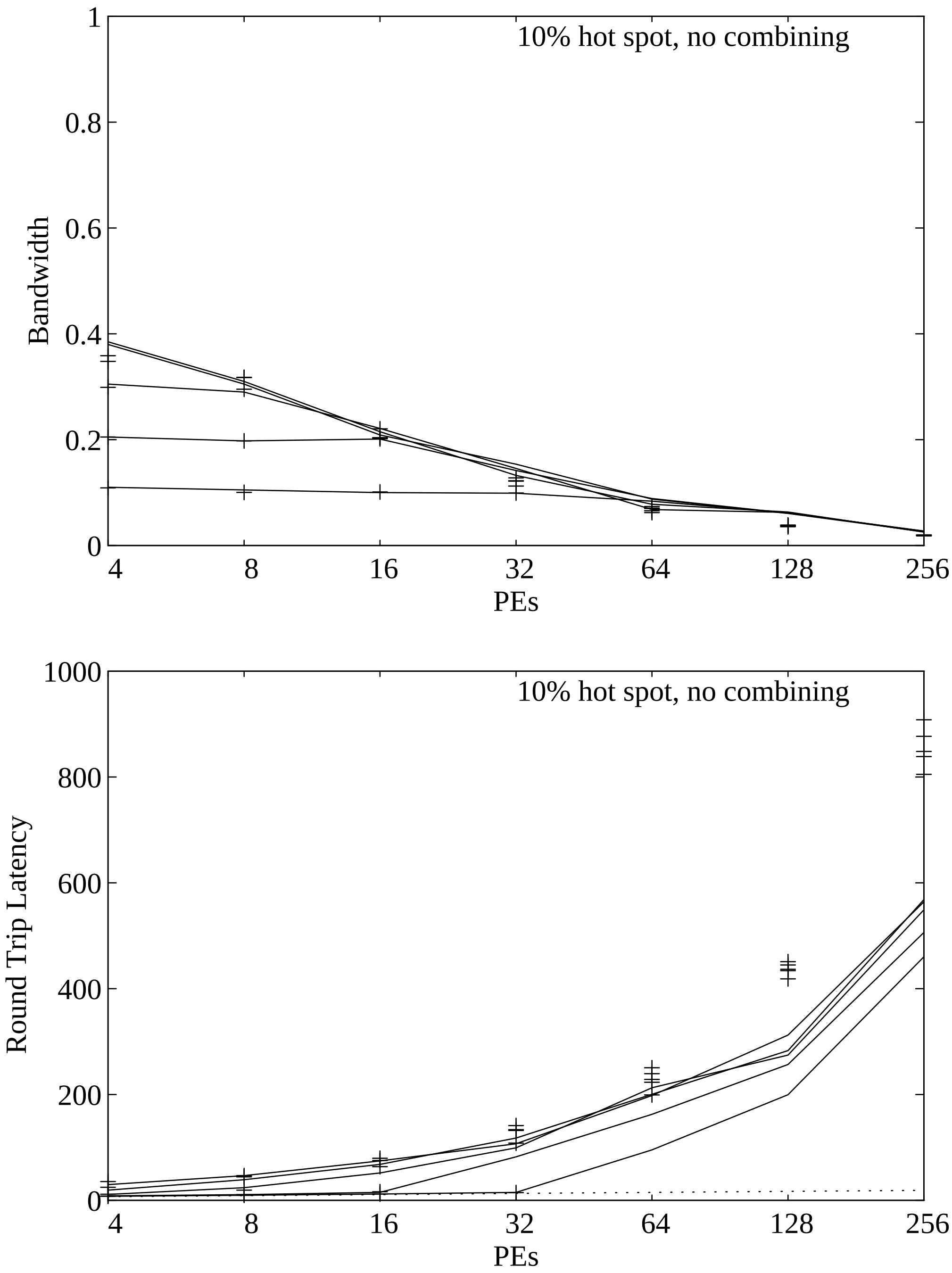


Figure 4.32: Type B switches, molasses and susy simulations, 10 percent hot spot, no combining, memory cycle 4.

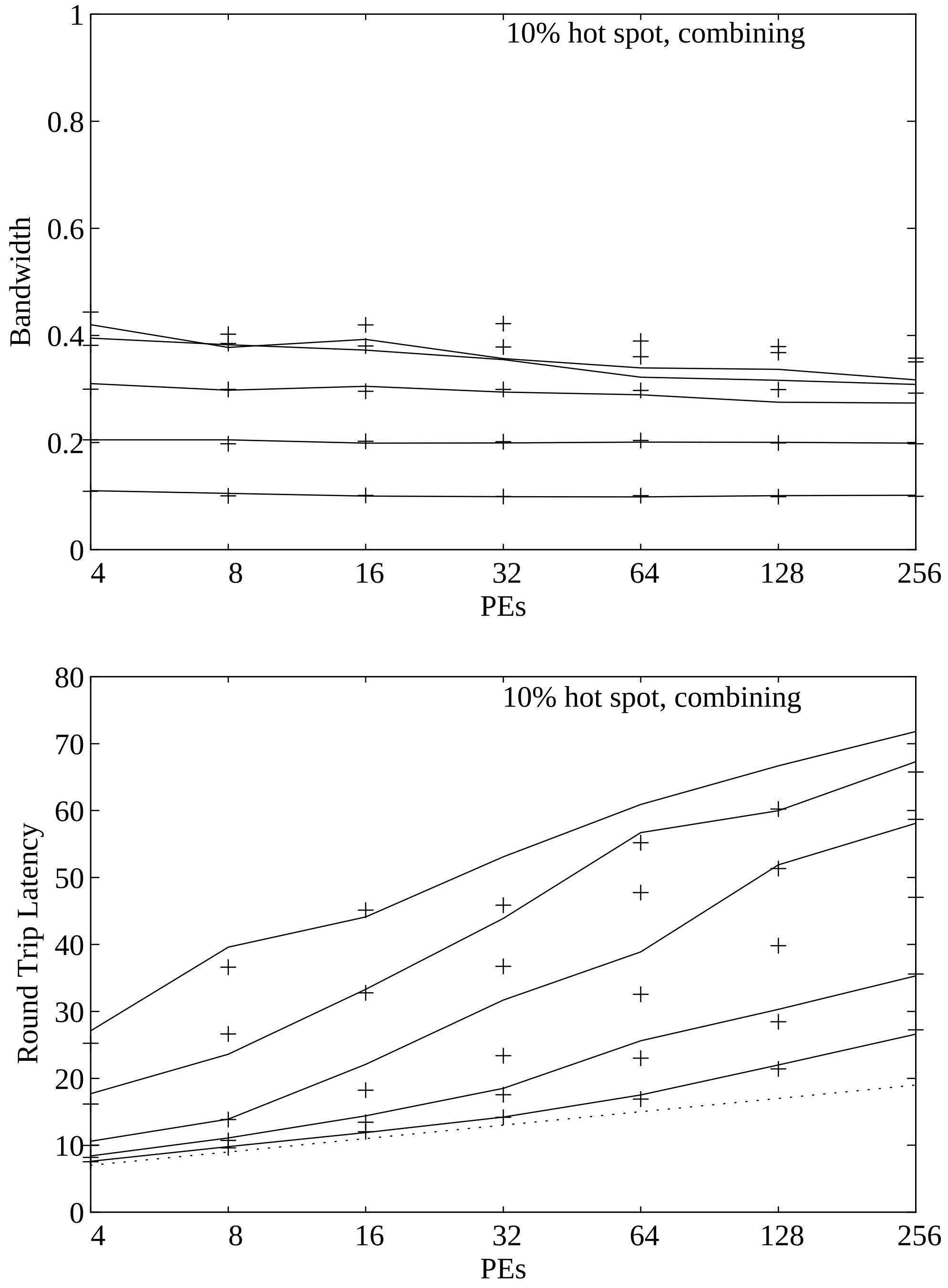


Figure 4.33: Type B switches, molasses and susy simulations, 10 percent hot spot, combining, memory cycle 4.

4.4.2 Type A and Type B combining networks

For uniform traffic, as seen in section 2.3, the difference between the performance of networks with Type A and Type B switches was very minor. For combining networks, however, a significant difference can be seen for larger systems at higher loads and hot spot rates. In Type B switches with combining systolic queues, only messages that entered the switch from the same input port will be in the same queue and thus eligible to combine. Two messages destined for a hot spot and entering a Type B switch from different input ports do not have the opportunity to combine until the following stage. Both messages take up space in the queue, increasing latency, and the output rate leaving the stage where they first meet is not reduced.

Figures 4.34 and 4.35 show the results of simulations at 100 percent offered load for Type A and Type B networks with from 2 to 1024 PEs. These simulations were done using molasses with the same parameters for the Type B simulations as those used in Figures 4.16 through 4.24, and with the equivalent parameters for the Type A simulations, using 20 packets as the size of each queue. With no combining, performance with hot spot traffic shows close to identical bandwidth at the same offered load for Type A and Type B switches, and a latency slightly but consistently lower for Type B switches. With combining, Type A switches show higher bandwidth and lower latency, for consistently better performance at larger system sizes and higher hot spot rates.

Figure 4.36 shows that, with a 1 percent hot spot rate, differences between the performance of Type A and Type B networks become noticable only at an overall bandwidth of greater than 50 percent. However, for a high hot spot rate of 10 percent, the difference in latency is significant even at low loads.

4.5 The cost of combining

The next three sections compare 2×2 two-way combining switches to 2×2 non-combining switches where similar choices have been made about packaging, packetizing and arrangement of buffers. The fourth section discusses the significance of the greater flexibility in packaging options available for non-combining switches.

4.5.1 Pins

At the current level of integration of the design, with a target system of 16 PEs and 16 MMs, the wait buffer port, which is needed only for combining, accounts for 34 signal pins out of 120 on the FPC and 34 out of 108 on the RPC. Since the wait buffer connections are always local to a switch board, they may be considered to have less cost than stage to stage connections, which may sometimes be from board to board. In our current implementation, boards contain a 4×4 network. Switch boards as well as switch chips are pin-limited rather than logic-limited or local wiring limited, so at the board level there is no additional wire cost for combining.

The op code and memory address fields are independent of system size, if the amount of memory per memory module is kept constant. The PE/MM address fields have $\log N$ bits. Since the number of possible outstanding messages must be kept proportional to $\log N$ in order to maintain linear bandwidth, the length of the ORI field must be $O(\log\log N)$. Currently we use a 4-bit ORI, allowing flexibility in the processor logic that assigns ORIs.

At $N = 2^{10}$, the wait buffer port would still require only 20 bits for the two address fields, allowing 6 bits each for ORIs within the 32 bits required for data transmission. Since the wait buffer address packet must contain two PE/MM address fields and two ORIs, for the two combined messages, compared to one address field and one ORI in the stage to stage address packets, the required width of the wait buffer port grows twice as fast as that of the stage to stage ports, for system sizes that force the width of the address packet to be larger than the data width.

4.5.2 Area and transistor count

Table 4.3 shows the cost of combining in on-chip logic as estimated from our current implementation. The percentage of combining cost for the sub-blocks of each component depend on the sub-block's function. In the FPC, the combining queue uses about 30 percent of its transistors for matching and the CHUTE row, which are the additional transistors needed for combining. In the RPC, 50 percent of the control and

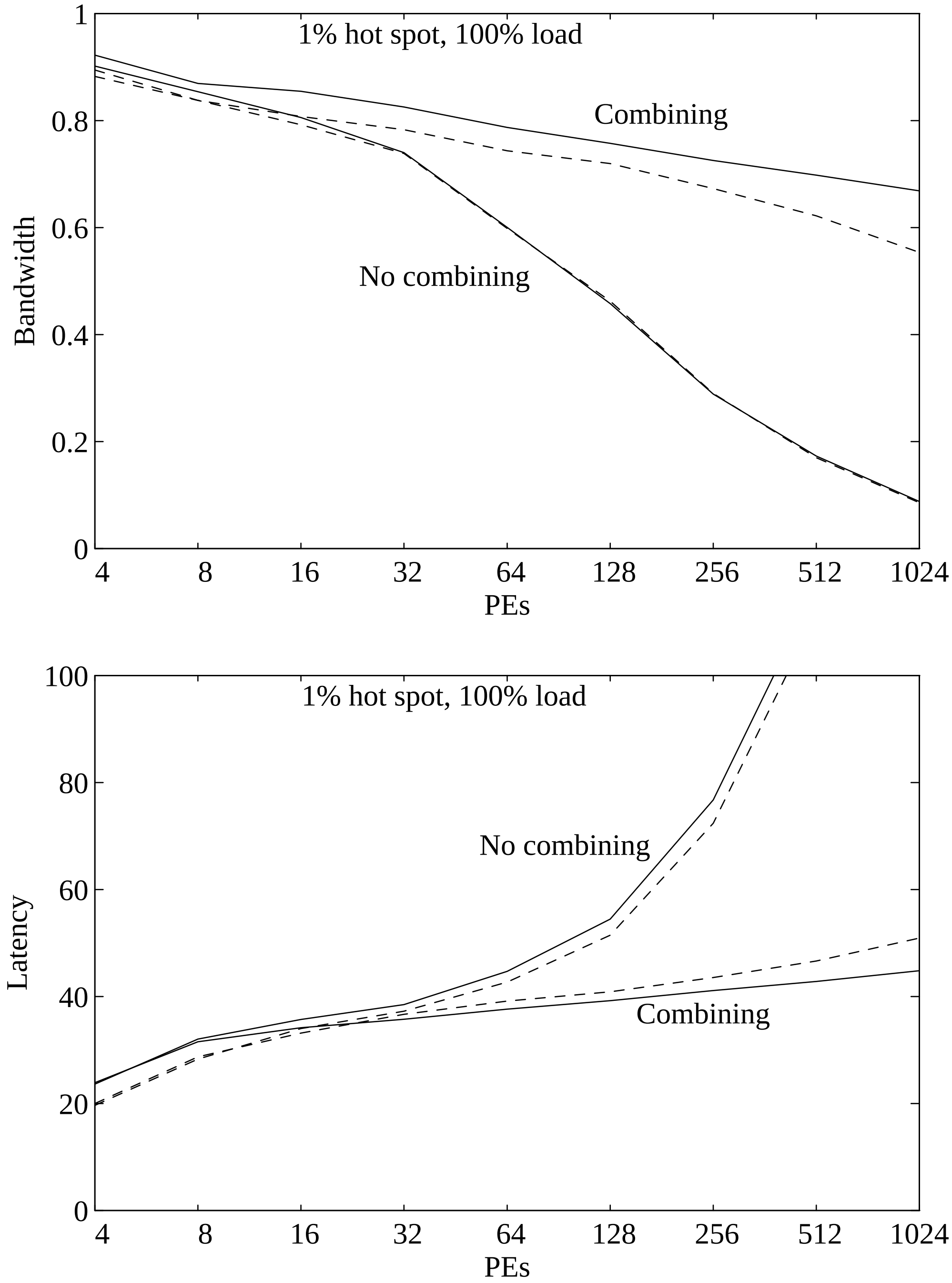
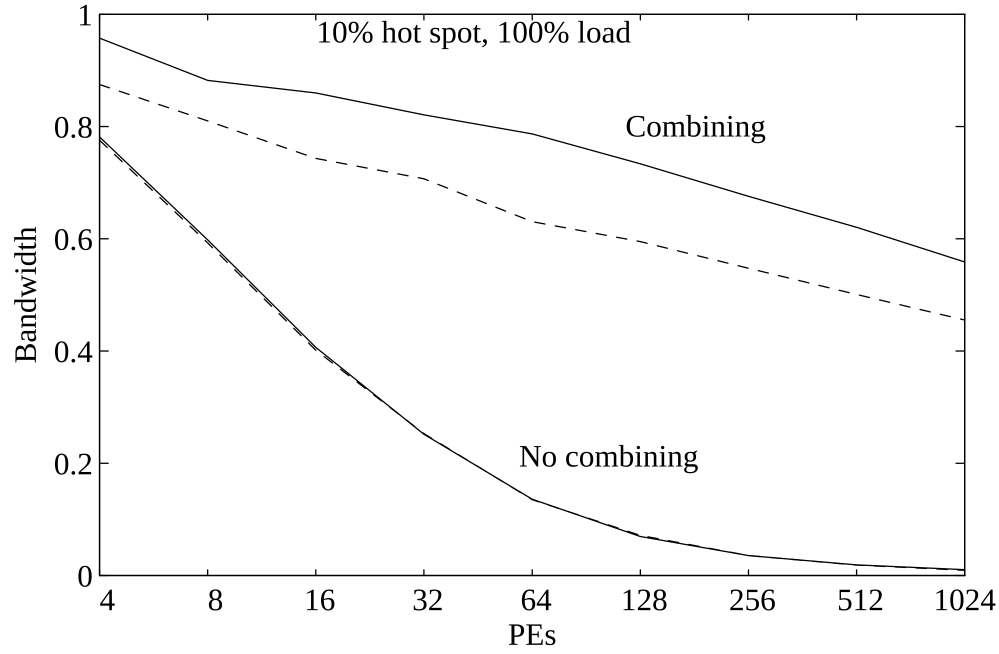


Figure 4.34: Type A and Type B networks, 1 percent hot spot, bandwidth and latency.



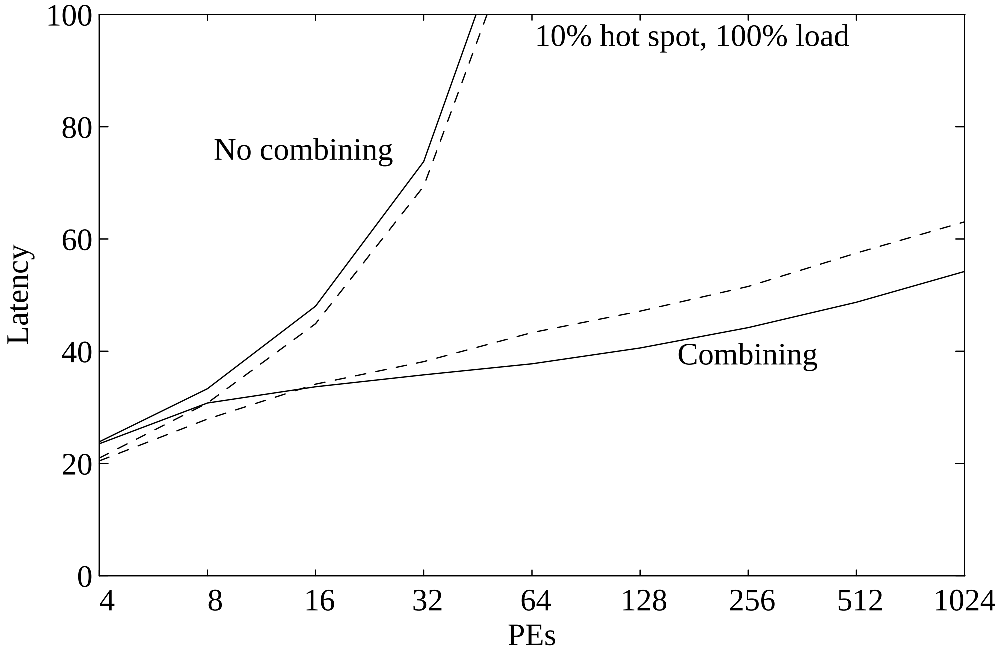


Figure 4.35: Type A and Type B networks, 10 percent hot spot, bandwidth and latency.

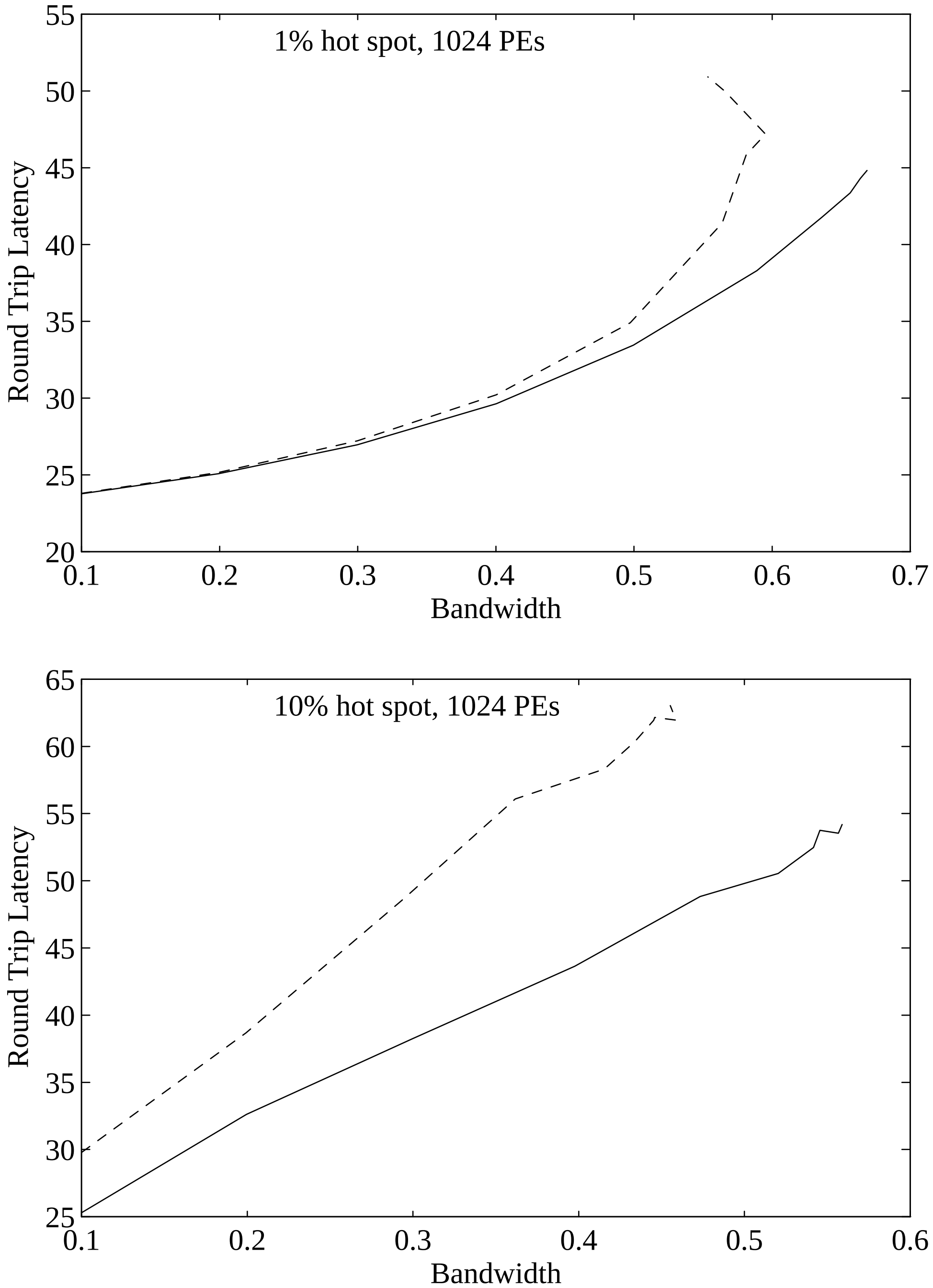


Figure 4.36: Type A and Type B networks, 10 percent hot spot, combining, 1024 PEs, latency as a function of bandwidth.

Component	Area		Transistors	
	$10^6\lambda^2$	Combining cost (percent)	10^3	Combining cost (percent)
Combining queue	9.0	30	22.4	30
ALU	3.7	100	5.8	100
Control and routing	5.0	0	2.5	0
FPC	17.7	36	29.1	43
ALU	3.1	100	3.9	100
Wait buffer	5.7	100	14.6	100
Main queue	4.9	0	15.7	0
Decombined queue	3.8	100	12.6	100
Control and routing	15.2	50	3.5	50
RPC	32.7	62	50.3	65
Total	50.4	53	79.4	57

Table 4.3: Area and transistor cost of combining capability in a switch

routing cost is charged to combining, to account for the cost of the internal buses and of multiplexing the two non-combining queues. The percentage of combining cost per component and for the total switch is then calculated.

As the number of bits in the PE/MM address and ORI fields increases, storage space must increase correspondingly. As in the case of pin cost, for systems where the length of the address packet dominates the length of the data packet, the wait buffer storage measured in bits must increase more rapidly than the regular queue storage. Furthermore, since the number of cycles a message has to wait to be decombined increases with network transit time, asymptotically the number of messages in the wait buffer will increase as $\log N$, like the number of outstanding messages at the PE. In the current design the wait buffer is over-engineered, capable of holding 8 messages in a 16 PE system.

4.5.3 Cycle time

Although we have implemented a non-combining switch and used it in our prototype, direct comparisons with our combining switch are difficult, because improvements in the fabrication process and in our design methodology have made our combining switch more than twice as fast as our non-combining switch! However, we have used a timing simulator to get process-independent estimates of the critical path delay.

The *critical paths* in a VLSI design are the slowest connections from an input on a clock phase to an output on that phase. We have used *crystal* [109], a timing analysis program developed at UC Berkeley and distributed as part of the Magic VLSI design tools, to find and optimize critical paths in our design. Table 4.4 shows delays for the worst paths in our design and for some other signals of interest as determined by crystal runs on parameters extracted for a 2 micron CMOS layout of the MOSIS design.

In the FPC, the most critical path during phase 1 is the path from the data valid signal in the first slice of the queue to the *QNE* (queue not empty signal) sent to the queue paired at an output. Crystal estimates this delay as 22ns at the output pad; the equivalent signal before buffering to the pad, *qne*, had a delay of 13 ns. The *QNE* signal is an input to the arbitration logic that produces the output select signals. The worst path on phase 2 to the *OUT.EOM* (output end of message) includes the output select signal. So the sum of these two delays on connected paths give a minimum clock cycle of 46 ns, for an expected maximum clock frequency of around 20 MHz.

The delay on this path does not show any dependency on the combining logic, but is affected by the partitioning of the node into separate components. If the two FPCs paired at an output were on one chip, the *QNE* input to the output select logic would have the 13 ns delay of *qne*. Furthermore, the phase 2 delay to the select signals of the on-chip multiplexing between the two queues, illustrated by *Addrsel* in the table, would be 14 ns, so that the total delay for both phases, including time driving the output pads, would be around 37 ns, or around 27 MHz. Note that the pin count required to eliminate this off-chip delay is smaller

Component	Phase 1		Phase 2	
	Signal	Delay	Signal	Delay
FPC	PadQne	22ns	OUT.EOM	24ns
	qne	13ns	Addrsel	14ns
	v.fo	13ns	data.out	14ns
	PPPP	10ns	WB.30	25ns
			IN.DA	17ns
RPC	in.qsfirst	19ns	RNQ_EOM	24ns
	PadMqne	19ns	padsel	16ns
	mqne	10ns	IN_DA	14ns
	full03-	13ns	FP_DA	18ns
			v_wb	10ns

Table 4.4: Critical path signal delays.

for non-combining switches than for combining switches.

Looking at the delays in the FPC that depend on the delay of the combining ALU, *data.out* (the output of the ALU that is input to the on-chip multiplexing) has a worst case phase 2 delay of 14 ns, for outputs of the high order bits of the carry lookahead ALU. This results in the worst case phase 2 delay to the output at *WB.30* of 25 ns.(*WB.30*, an output on the port to the wait buffer, is connected to an ALU output because it receives the ORI from the message in the OUT row in its address packet. Thus, when the output of the ALU is actually selected for this signal, the long delay paths in the CLA are not active. However, other output signals from the ALU to the NQ (next queue) port have a delay only one or two nanoseconds shorter.) During phase 1, the worst delays in the ALU chain are to *PPPP*, the propagate bit at the top of the CLA tree, and *v.fo*, the buffered data valid signal used to qualify the clock used in the ALU and CLA logic. So the total delay for both phases is 38 ns. Since this delay is unaffected by chip partitioning, the delay through the ALU becomes the critical path with the NCR packaging.

The RPC behavior is similar, with *PadMQNE* to *RNQ_EOM* forming the critical path over both phases, for a total predicted clock period of 43 ns. The ALU outputs such as *in.qsfirst* are active during phase 1 on the return path and show a delay of 19ns. Delays attributable to wait buffer logic are relatively small. The worst path is through the wait buffer full logic, signal *full03-* on phase 1 with a delay of 13 ns, to the data accept signal to the forward path, *FP_DA*, with a phase 2 delay of 18 ns, for a total of 31 ns. This delay would be shortened by packaging the FPC and RPC components together on one chip.

4.5.4 Packaging options

With higher pin count packages, a switch can be implemented with fewer chips; the next level of integration is achieved by packaging components which share output ports together. Pin counts, area estimates and transistor counts for these packaging alternatives are shown in Table 4.5. The 8 chip per switch alternative corresponds to our MOSIS implementation, with higher pin count due to longer PE/MM addresses; the 4 chip per switch alternative is the NCR implementation. Note that area and total pin count for packaging the entire switch on one chip is well within current technological limits, though such high pin-count packages are not cheap.

The area estimates include routing for internal buses, but do not include pads. The transistor count is slightly overestimated for the higher levels of integration because some of the control logic is duplicated in each part. The switch design is pin-limited rather than area-limited. Even with all the logic on a single chip, the area for internal logic would only be approximately 6 mm $\times$ 6 mm with 0.8 micron feature size.

Including combining in a design does have the additional cost of decreasing the package flexibility available to the designer. Bit-slicing becomes an undesirable option, because of the larger amount of “horizontal” chip-to-chip communication required by matching and the ALU operations. Decreasing packet size by pipelining can also cause difficulties, if the address information must be divided into multiple packets (see section 4.2.1).

Chips per Switch	Transistors 10^3	Area $10^6\,\lambda^2$	Signal Pins per Chip				
			Input	Output	Tri-state	Clocks	Total
8							
FPC	29	18	43	2	74	2	121
RPC	50	32	70	4	35	2	111
4							
FPC	58	36	82	76	0	2	160
RPC	100	64	134	38	0	2	174
2							
FPC	116	72	84	150	0	2	236
RPC	200	128	136	74	0	2	210
1	316	200	152	150	0	2	304

Table 4.5: Signal pin count, area and transistors per chip for 2×2 combining switches to be used in a 256-PE system with a 4 gigabyte address space

Chapter 5

Providing Greater Combining Capability

Increasing the combining capability of a switch naturally increases its hardware cost. This chapter describes the results of investigations into the relative performance of switches of different combining capabilities, and outlines the implementation of some of these alternatives. The first section of this chapter compares the theoretical performance, with unbounded buffer size, of two-way and unlimited combining, describing joint work with Ora Percus that was published in [40]. The second section discusses the implementation and performance of a Type A 2×2 switch with "two-and-a-half-way" combining, which promises better performance at only slightly greater cost than the Type B switch with two-way combining described in Chapter 4. The third section discusses implementation alternatives and performance for a 4×4 combining switch.

5.1 Unlimited and two-way combining

Previous simulation studies have suggested that a two-way combining switch, like that implemented in Chapter 4, may not be adequate to prevent saturation for indefinitely large networks [86, 87]. These studies modeled processors as an infinite source of requests, and indicated that, for some patterns of traffic and queue size, switches at later stages in the network become saturated beyond the capability of two-way combining to improve, and that combining of mulplicity greater than two was needed for two-way switches. On the other hand, studies using practical assumptions, like those in the simulations of Chapter 4 have shown good performance for two-way combining networks of 10 stages or more [68, 67, 100].

In this section we consider two kinds of combining, unlimited and two-way, for Type A switches with unbounded buffer size. They correspond to the most expensive and cheapest hardware implementations of combining. In unlimited combining, any number of hotspot messages which are in a queue at the same time may join together and exit the queue as only one message. In two-way combining, a hotspot message may combine with only one other hotspot message at a given switch. Once these two messages have combined, the resulting message may not combine with any other hotspot message in that switch, though it may combine again at a later stage in the network. We refer to this type of combining as having a *combining multiplicity* of two.

We also study the effect of allowing the message at the front of the queue to combine with a new message before exiting. If such a combine is to occur, the cycle time of the switch must be long enough to allow the two messages to be combined before the result leaves the switch. If a message at the front of the queue is not available for combining, the combining logic can be executed in parallel with sending a message to the next stage, as is done in the switch in Chapter 4. The cycle time of the switch is reduced, but fewer combine operations take place and the hotspot traffic leaving the switch is greater.

In all cases, we analyze combining switch behavior under a simple model that assumes there is only a single active hot spot. While realistic traffic patterns will certainly be much more complicated, this simple pattern is useful for acquiring insight into the relevant factors in switch design. We want to know how much

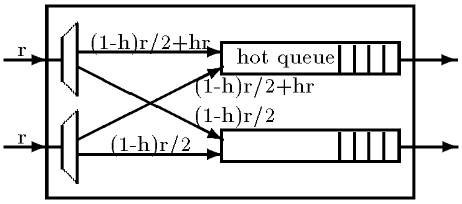


Figure 5.1: Type A switch with hot spot traffic.

each of the different schemes of combining reduces the output rate of hotspot traffic, and how long the queues in the switches grow.

5.1.1 Model of a combining switch

We modifying the model of Type A switch behavior presented in section 2.4 by assuming that there are two classes of message traffic from processors to memory: the regular (*red*) traffic which is distributed uniformly over the memory modules, and the hotspot (*blue*) traffic which is all destined for a single memory location (the "hot spot"). [1] The hotspot traffic will ordinarily be a small percentage h of the total message request rate r from a processor. Messages directed to the hotspot have the property that they may be combined; messages that have been combined continue through the network as a single message. Because of the way routing is done in a delta network, one of these queues will receive all the messages directed to the hotspot.

Looking just at this "hot" queue (see Figure 5.1), the input rate of regular traffic is $(1-h)r$; the input rate of hotspot traffic to the "hot" queue is $2hr$. The expected output rate of regular traffic will be the same as the input rate; the expected output rate of hotspot traffic will be reduced from $2hr$ by the number of combines that have occurred. S_n no longer depends just on S_{n-1} and the number of arrivals, but also on the color of the arrivals and on the color of the messages in the queue.

In [68, 67], the authors construct queue size transition probabilities based on a potential combining probability e which they describe as the probability that the number of combinables in the queue is odd. The accuracy of the equation they use to evaluate e is discussed in [40]. At best, a single value for the potential combining probability of a switch can only be an average over the different queue sizes; the actual combining probability at a given cycle is a function of the length of the queue at that time. The transition probabilities they construct are thus only approximations.

To track the queue length dependent probability of combining, we rely on the insight that there can never be more than one message in a queue that has the potential to combine. With unlimited combining, a combinable message in the queue will absorb all new combinable messages until it departs. With two-way combining, a combinable message loses its ability to combine in a switch as soon as it combines with another message, so, though there may be more than one message in the queue destined for the hot spot, at most one message will still be able to combine. The state of the queue at the end of cycle n can thus be described by the pair (S_n, T_n) where S_n is the number of non-combinable items in the queue and T_n is the location in the queue of the combinable item (T_n is 0 if there is no item in the queue which can combine).

The probability $p_n(k, j)$, defined as the probability that $S = k$ and $T = j$ at cycle n, can be computed from recurrences whose exact form depends on the details of the combining switch architecture being modeled. Although these recurrences were too complicated for us to solve in a closed form, as was done for the non-combining switch, steady state probabilities can be obtained for any values for which the queues are stable, by computing until $p_n(k, j)$ has converged to $p_{n-1}(k, j)$ for all k, j. The details of the recurrence relations for four possible combining switch architectures are given in the next four subsections.

Changes in S and T for all the combining switch designs described below depend not just on the number of arriving messages but on their color. At each cycle, at each input port, a *red* message (from the regular traffic stream) arrives with probability $(1-h)r$ and enters one of the two output queues with probability $1/2$; a *blue* message (from the hotspot traffic stream) arrives with probability hr and enters the hot output

[1] We have been told we chose the wrong colors. Try to remember that a blue flame is hotter than a red one.

queue (see Figure 5.1). Hence, the probability that a given pattern of messages arrive in any cycle at a given queue is

$$\begin{aligned} f_{0,0} &= (1 - \frac{(1+h)r}{2})^2 \\ f_{r,0} = f_{0,r} &= \frac{(1-h)r}{2}(1 - \frac{(1+h)r}{2}) \\ f_{r,r} &= (1-h)^2 \frac{r^2}{4} \\ f_{b,0} = f_{0,b} &= hr(1 - \frac{(1+h)r)}{2}) \\ f_{b,r} = f_{r,b} &= \frac{h(1-h)r^2}{2} \\ f_{b,b} &= h^2 r^2 \end{aligned}$$

where the subscript r corresponds to the arrival of a red input, b corresponds to the arrival of a blue input, and 0 corresponds to no message arriving on that port.

5.1.2 Unlimited combining, front of queue can combine

The recurrence relations for this combining switch are the simplest to formulate. There are essentially three cases to consider:

1. $T_{n-1} = 0$. (At the start of the cycle, there was no blue message in the queue.) Then any single arrivals to the queue will simply be added at the end of the queue. Two red arrivals will likewise just be added to the end of the queue. If two blue messages arrive together, they will combine and one blue message will be added to the end of the queue. If a red message and a blue message arrive together, they will both be added to the end of the queue, with each having an equal likelihood of being added first. At the end of the cycle, the message at the front of the queue will be deleted. This will be a red message, unless the queue was empty at the start of the cycle.

   Thus $S_n = S_{n-1} + 1$, if two red messages arrived, or $S_n = S_{n-1}$, if one red message arrived or none arrived when the queue was empty, or $S_n = S_{n-1} - 1$, if no red messages arrived and the queue was non-empty. T_{n-1} was 0, and will be set to S_n if a red and a blue message arrive, and the blue one is added first, or to $S_n + 1$, if a red and a blue message arrive, and the red one is added first. If one or two blue messages arrive with no red messages, $T_n = S_n + 1$, unless the queue was empty, in which case both will exit, leaving $T_n = S_n = 0$.

2. $T_{n-1} = 1$. (At the start of the cycle, there was a blue message in the queue, at the first place.) Then any blue arrivals to the queue will disappear, being combined with the blue message already in the queue, and will exit that cycle. Any red arrivals will be added to the queue as in case (1), but since no red message will exit, $S_n = S_{n-1} + a_n$, where a_n is the number of red arrivals. T_n must be zero.

3. $T_{n-1} > 1$. (At the start of the cycle there was a blue message in the queue, not at the first place.) Then any blue arrivals to the queue will disappear, being combined with the blue message already in the queue. Any red arrivals will be added to the queue as in case (1), and since a red message exits, $S_n = S_{n-1} + a_n - 1$. The blue message in the queue will move closer to the output, so $T_n = T_{n-1} - 1$.

These transitions are summarized in Table 5.1. If two possible resulting states are listed for a given arrival pattern, either can occur with equal probability.

Using the function

$$\delta_{x,y} = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$$

S_{n-1}, T_{n-1}	S_n, T_n					
	Arrivals					
	$0,0$	$r,0$ $0,r$	r,b b,r	r,r	$b,0$ $0,b$	b,b
$0,0$	$0,0$	$0,0$	$0,1$ $1,0$	$1,0$	$0,0$	$0,0$
$k,0$ $(k > 0)$	$k-1,0$	$k,0$	k,k $k,k+1$	$k+1,0$	$k-1,k$	$k-1,k$
$k,1$ $(k \geq 0)$	$k,0$	$k+1,0$	$k+1,0$	$k+2,0$	$k,0$	$k,0$
k,j $(k > 0,$ $j > 0)$	$k-1,j-1$	$k,j-1$	$k,j-1$	$k+1,j-1$	$k-1,j-1$	$k-1,j-1$

Table 5.1: Transitions for unlimited combining, including combining at the front of the queue.

it is possible to write the recurrences for the probability distribution explicitly as

$$
\begin{aligned}
p_n(0,0) &= (f_{0,0} + 2f_{r,0} + 2f_{b,0} + f_{b,b})p_{n-1}(0,0) \\
&\quad + (f_{0,0} + 2f_{b,0} + f_{b,b})p_{n-1}(0,1) \\
&\quad + f_{0,0}p_{n-1}(1,0), \\
p_n(0,1) &= f_{r,b}p_{n-1}(0,0) + (f_{b,0} + f_{b,b})p_{n-1}(1,0) + (f_{0,0} + 2f_{b,0} + f_{b,b})p_{n-1}(1,2) \\
p_n(1,0) &= (f_{r,r} + f_{r,b})p_{n-1}(0,0) + (2f_{r,b} + 2f_{r,0})p_{n-1}(0,1) \\
&\quad + f_{r,0}p_{n-1}(1,0) + f_{0,0}p_{n-1}(1,1) + f_{0,0}p_{n-1}(2,0), \\
p_n(k,0) &= f_{r,r}p_{n-1}(k-2,1) + (2f_{r,0} + 2f_{r,b})p_{n-1}(k-1,1) \\
&\quad + (f_{0,0} + 2f_{b,0} + f_{b,b})p_{n-1}(k,1) + f_{r,r}p_{n-1}(k-1,0) \\
&\quad + f_{r,0}p_{n-1}(k,0) + f_{0,0}p_{n-1}(k+1,0), \text{for } k \geq 2, \text{ and} \\
p_n(k,j) &= f_{r,b}p_{n-1}(k,0)\delta_{j,k} + f_{r,b}p_{n-1}(k,0)\delta_{j,k+1} \\
&\quad + (2f_{b,0} + f_{b,b})p_{n-1}(k+1,0)\delta_{j,k+1} \\
&\quad + (f_{0,0} + 2f_{b,0} + f_{b,b})p_{n-1}(k+1,j+1) \\
&\quad + (2f_{r,0} + 2f_{r,b})p_{n-1}(k,j+1) + f_{r,r}p_{n-1}(k,j) \text{ for } k \geq 1, j > 1.
\end{aligned}
$$

The program to do the computation, however, can be written directly from the table with less chance of error, by accumulating probabilities for each of the four cases corresponding to the rows of the table.

The probability that the queue has length k at time n is given by

$$p_n(k) = p_n(k,0) + \sum_{j=1}^{n} p_n(k-1,j). \tag{5.1}$$

If the values of the probability distribution converge, the output rate in steady state

$$O_n = 1 - (1 - f_{0,0})p_n(0,0). \tag{5.2}$$

Figure 5.2 graphs the computed values for average queue length and output rate as a function of the hot spot rate for input rates of 0.1, 0.3, 0.5, 0.6, 0.7, 0.8 and 0.9. The match with simulation values is very close.

5.1.3 Unlimited combining, no combining at front of queue

This differs from unlimited combining which allows the message at the front of the queue to combine only in the cases when the queue is empty or when there is a blue message at the front of the queue. We consider the same three cases as before, describing only the behavior which is different from section 5.1.2:

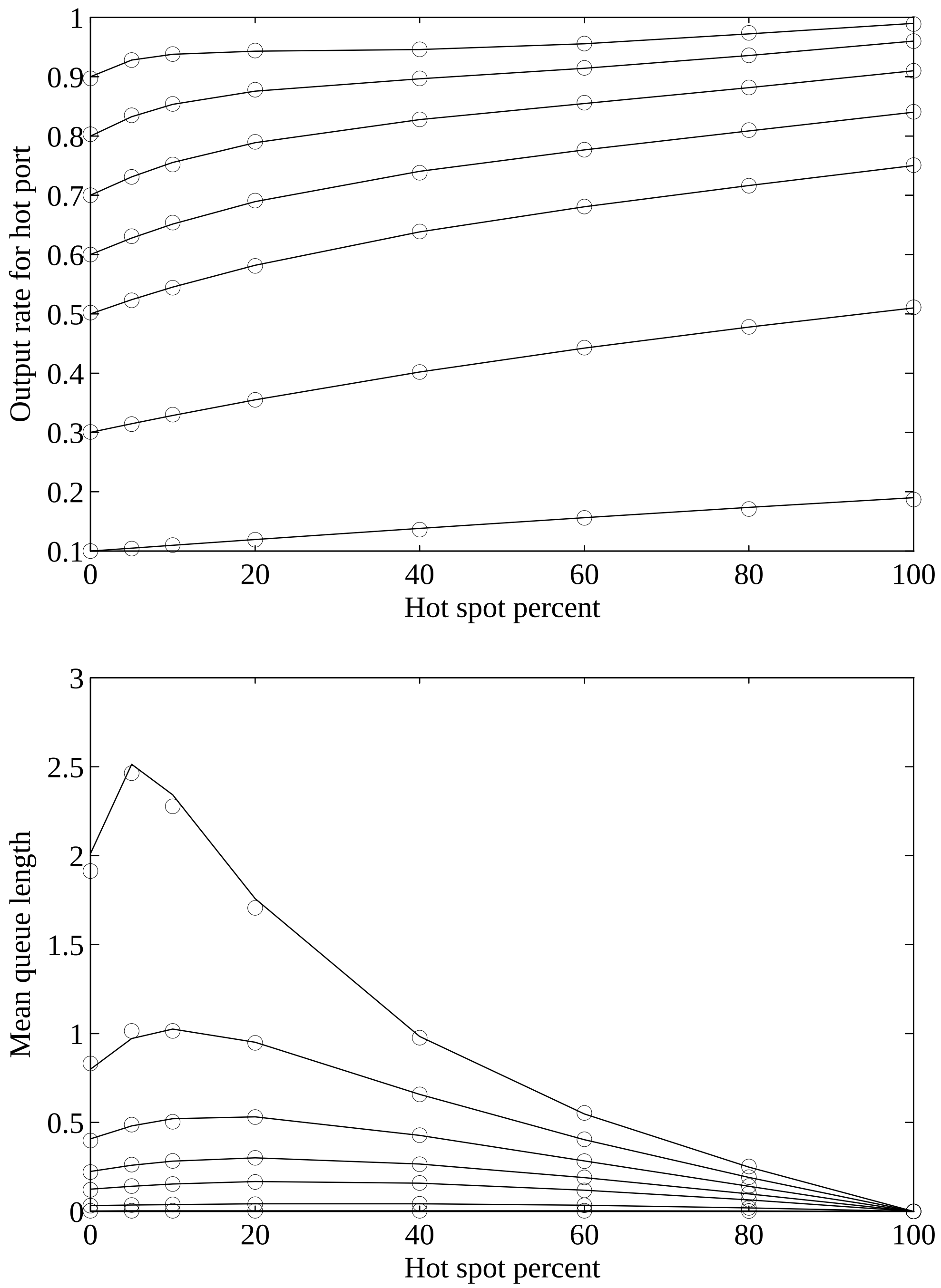


Figure 5.2: Unlimited combining, including combining at the front of the queue.

	S_n, T_n					
	Arrivals					
S_{n-1}, T_{n-1}	$0, 0$	$r, 0$ $0, r$	r, b b, r	r, r	$b, 0$ $0, b$	b, b
$0, 0$	$0, 0$	$0, 0$	$0, 1$ $1, 0$	$1, 0$	$0, 0$	$0, 1$
$k, 0$ $(k > 0)$	$k-1, 0$	$k, 0$	k, k $k, k+1$	$k+1, 0$	$k-1, k$	$k-1, k$
$k, 1$ $(k \geq 0)$	$k, 0$	$k+1, 0$	$k+1, k+1$ $k+1, k+2$	$k+2, 0$	$k, k+1$	$k, k+1$
k, j $(k > 0,$ $j > 0)$	$k-1, j-1$	$k, j-1$	$k, j-1$	$k+1, j-1$	$k-1, j-1$	$k-1, j-1$

Table 5.2: Transitions for unlimited combining, no combining at the front of the queue.

1. $T_{n-1} = 0$. If two blue messages arrive when the queue is empty, one of them will be at the front of the queue, so they will not combine. All other cases are the same as in section 5.1.2.

   Thus S_n will behave as previously for this case, but if S_{n-1} was 0, $T_n = 1$ if two blue messages arrive, instead of remaining 0.

2. $T_{n-1} = 1$. A single blue arrival to the queue will not combine, but will be added to the end of the queue; two blue arrivals will likewise combine with each other and be added at the end of the queue. $S_n = S_{n-1} + a_n$, where a_n is the number of red arrivals, as before. If there are one or two blue arrivals, and no red arrivals, $T_n = S_n + 1$. If both a red and a blue arrive, $T_n = S_n + 1$ or S_n, depending on which is added first.

3. $T_{n-1} > 1$. Since the message at the front of the queue is red and not combinable, behavior is the same as in section 5.1.2.

These transitions are summarized in Table 5.2.

Average queue length and output rate are shown in Figure 5.3. For a single stage, the values show little differences from unlimited combining including the front of the queue for low hot spot percentages. In order for combining to occur with a low hot spot percentage, the overall rate must be high enough that the queue lengths are over 1; thus the combining scheme of this section is not at a disadvantage. For higher hot spot rates, the queue length naturally has a lower bound at 1 instead of at 0.

5.1.4 Two-way combining, combining at front of queue

For two-way combining, a blue message can be allowed to combine only once with another blue message, so we will think of it as having turned red when it combines. There can still never be more than one blue message in the queue, but now blue messages can "disappear" from the queue (T_n becomes 0) when a new blue message arrives, not just when the blue message exits the queue. S_n will increase whenever a blue message turns red, as well as when a red message is added.

Considering the three cases as before:

1. $T_{n-1} = 0$. Single arrivals of either color and double red arrivals will be added to the end of the queue as for unlimited combining. The only difference is in the behavior of a double blue arrival. Instead of adding a blue item to the queue, a red one is added.

   Changes in S_n and T_n are the same as for unlimited combining, except that a double blue arrival behaves like a single red arrival.

2. $T_{n-1} = 1$. Again, the only change from unlimited combining is for double blue arrivals. Instead of both arrivals combining with the message at the front of the queue and exiting, one will combine and

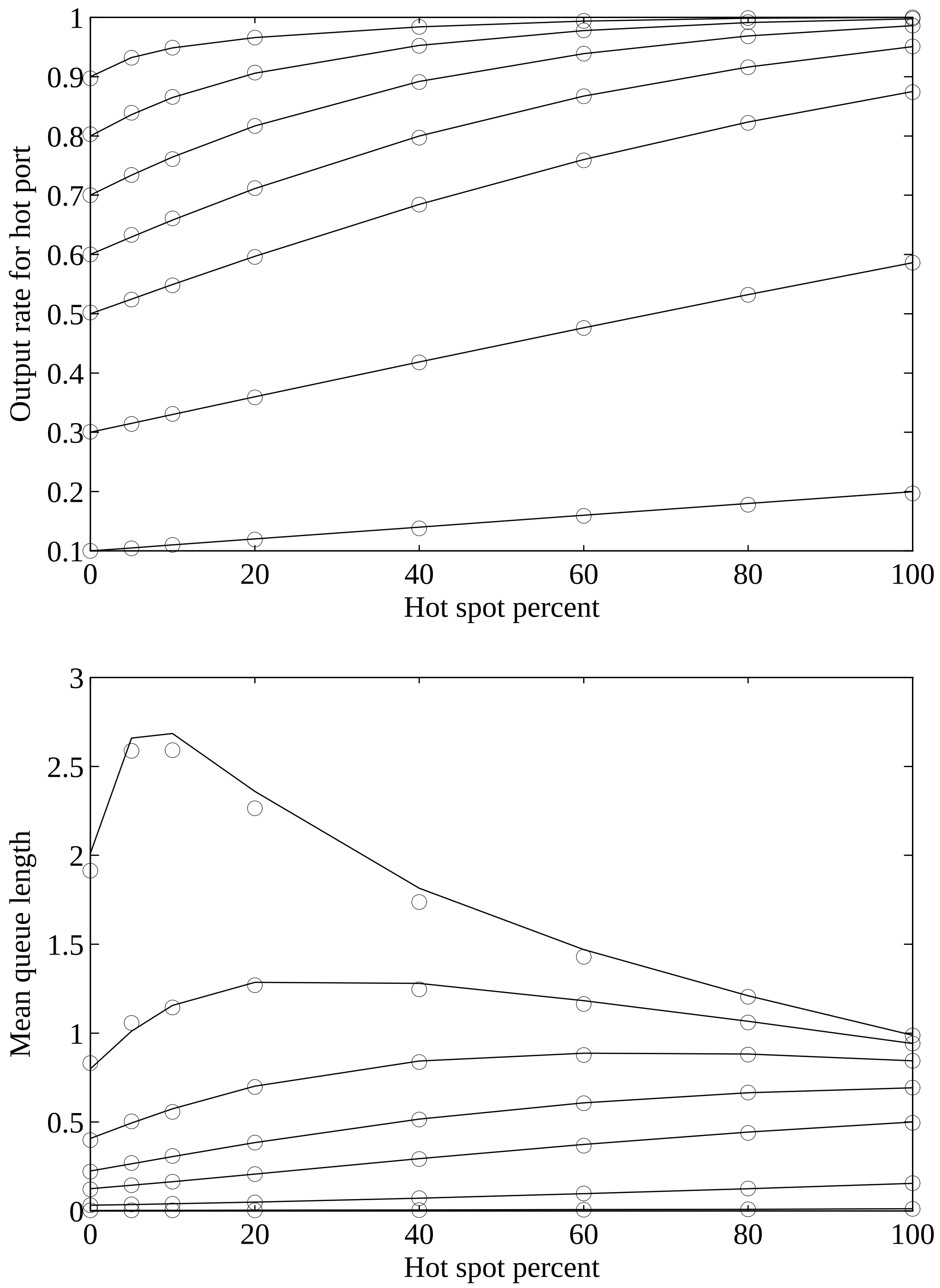


Figure 5.3: Unlimited combining, no combining at the front of the queue.

	S_n, T_n					
	Arrivals					
S_{n-1}, T_{n-1}	$0,0$	$r,0$ $0,r$	r,b b,r	r,r	$b,0$ $0,b$	b,b
$0,0$	$0,0$	$0,0$	$0,1$ $1,0$	$1,0$	$0,0$	$0,0$
$k,0$ $(k>0)$	$k-1,0$	$k,0$	k,k $k,k+1$	$k+1,0$	$k-1,k$	$k,0$
$k,1$ $(k \geq 0)$	$k,0$	$k+1,0$	$k+1,0$	$k+2,0$	$k,0$	$k,k+1$
k,j $(k>0,$ $j>1)$	$k-1,j-1$	$k,j-1$	$k+1,0$	$k+1,j-1$	$k,0$	$k,k+1$

Table 5.3: Transitions for a 2-way combining queue.

exit, and the other will join the end of the queue. So $T_n = S_n + 1$ for double blue arrivals; other cases are as before.

3. $T_{n-1} > 1$. A single blue arrival to the queue will disappear, being combined with the blue message already in the queue, and will cause that message to turn into a red message. A double blue arrival will both cause the blue message in the queue to turn into a red message, and will add a blue item to the end of the queue. Any red arrivals will be added to the queue as before.

   $S_n = S_{n-1}$ in the case of a single red or blue arrival, or a double blue arrival. $S_n = S_{n-1} + 1$ if there is a double red arrival or a red and a blue arriving together. $T_n = 0$ if there is a single blue arrival, or a red or blue arriving together, and $T_n = S_n + 1$ if there is a double blue arrival. Other transitions are as before.

These transitions are summarized in Table 5.3.

Average queue length and output rate are shown in Figure 5.4. For a single stage queue, two-way combining including the front message shows slightly shorter queue sizes and lower output rates than unlimited combining that does not combine the front message for loads under 0.9.

5.1.5 Two-way combining, no combining at front of queue

This case will be described in terms of its differences from section 5.1.4. Considering the three cases as before:

1. $T_{n-1} = 0$. The only difference from section 5.1.4 occurs when two blue messages arrive at an empty queue. Instead of combining and exiting together, one will exit and the other will remain in the queue. So in that case $S_n = 0, T_n = 1$.

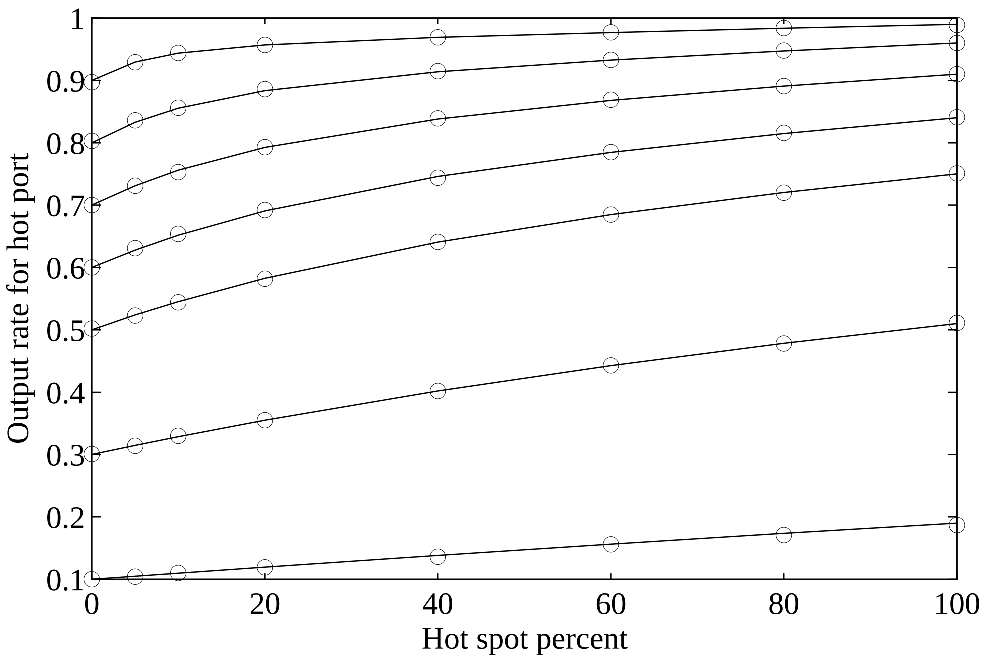
2. $T_{n-1} = 1$. The major differences from section 5.1.4 are in this case. Instead of combining and exiting, a single blue arrival or a blue arriving with a red will be added at the end of the queue. A double blue arrival will combine and add one red message at the end of the queue.

   Thus in the case of a single blue arrival, $S_n = S_{n-1}$ and $T_n = S_n + 1$. If a red arrives with a blue, $S_n = S_{n-1} + 1$ and T_n equals S_n or $S_n + 1$. If two blue messages arrive, $S_n = S_{n-1} + 1$ and $T_n = 0$.

3. $T_{n-1} > 1$. Behavior is exactly as in section 5.1.4.

These transitions are summarized in Table 5.4.

Average queue length and output rate are shown in Figure 5.5. Unsurprisingly, this scheme has the worst performance of the four, though the differences for hot spot percentages under 10% are not great.



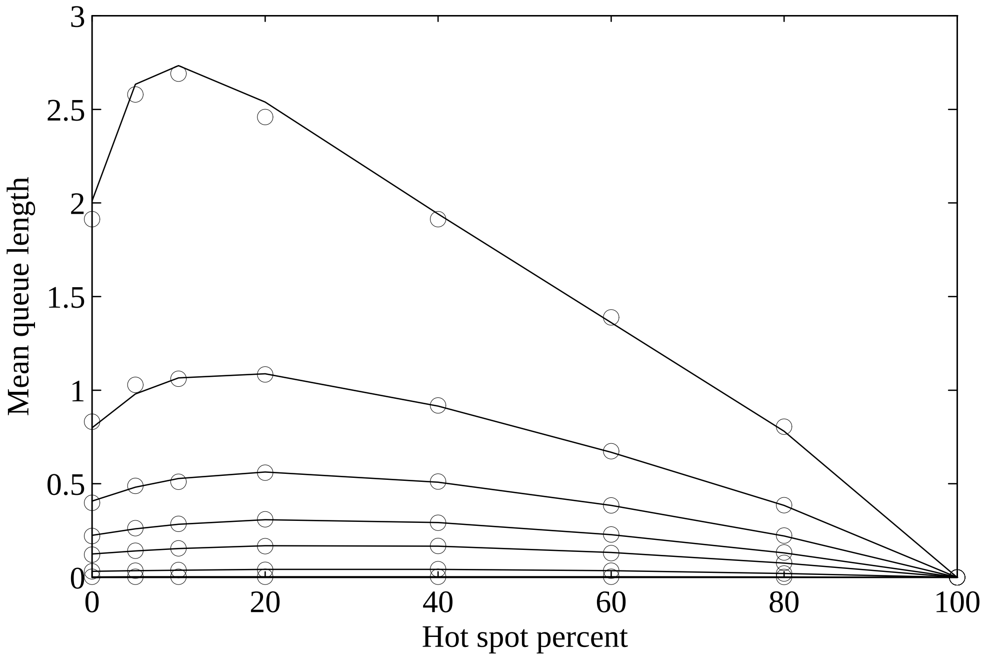


Figure 5.4: Two-way combining, including combining at the front of the queue.

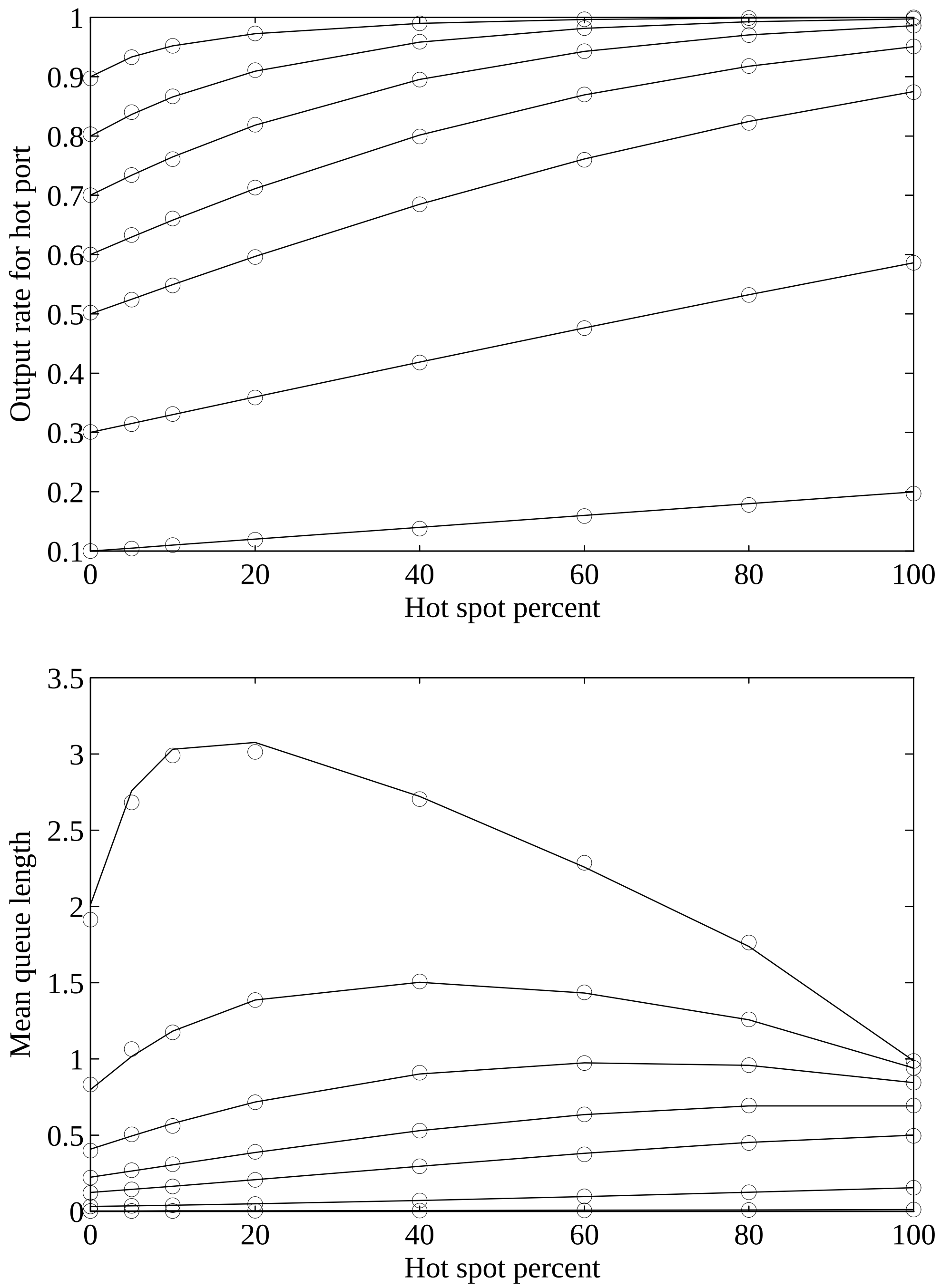


Figure 5.5: Two-way combining, no combining at the front of the queue.

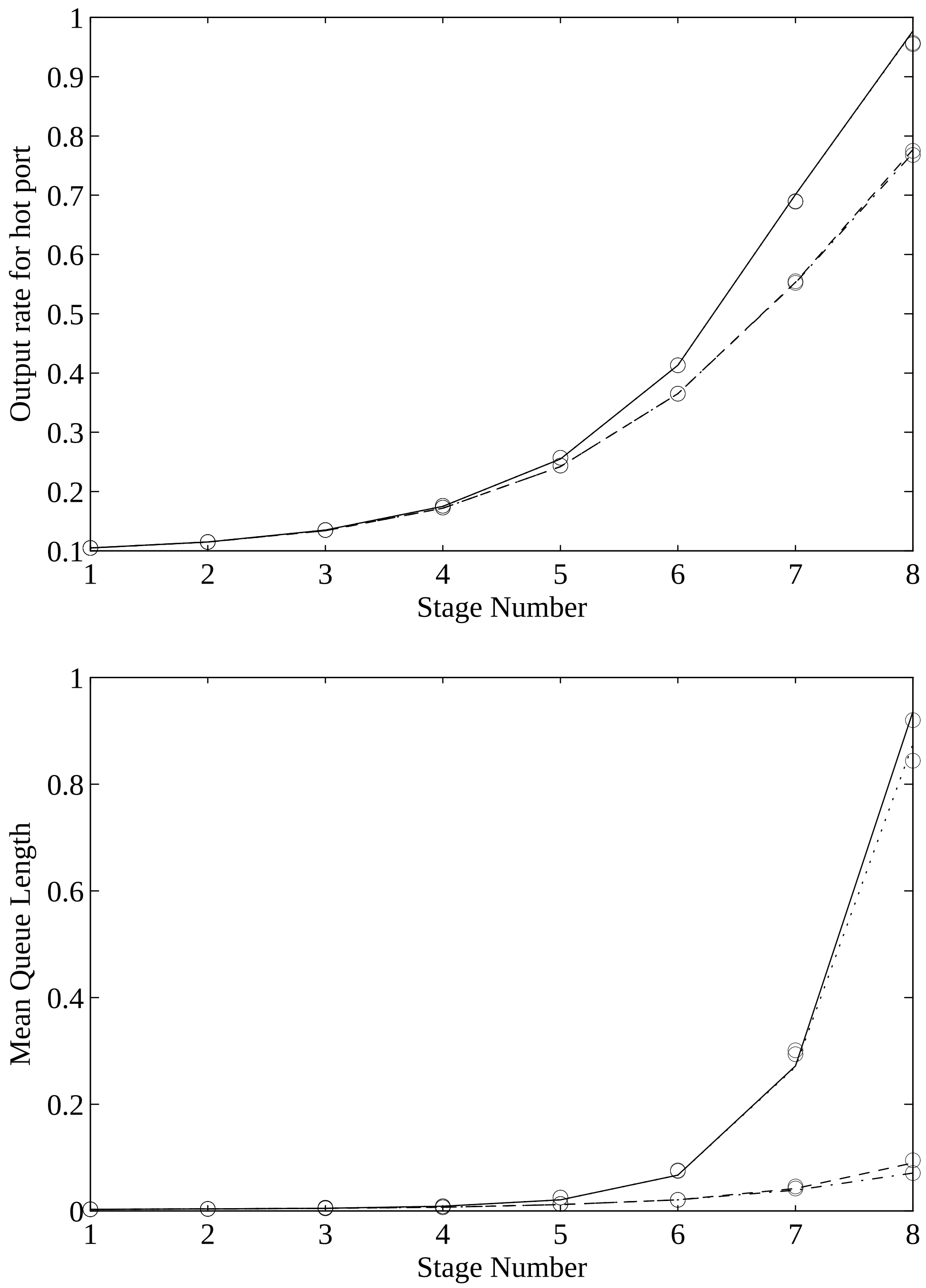


Figure 5.6: Analytical estimates of combining performance, 10 percent load, 5 percent hot spot.

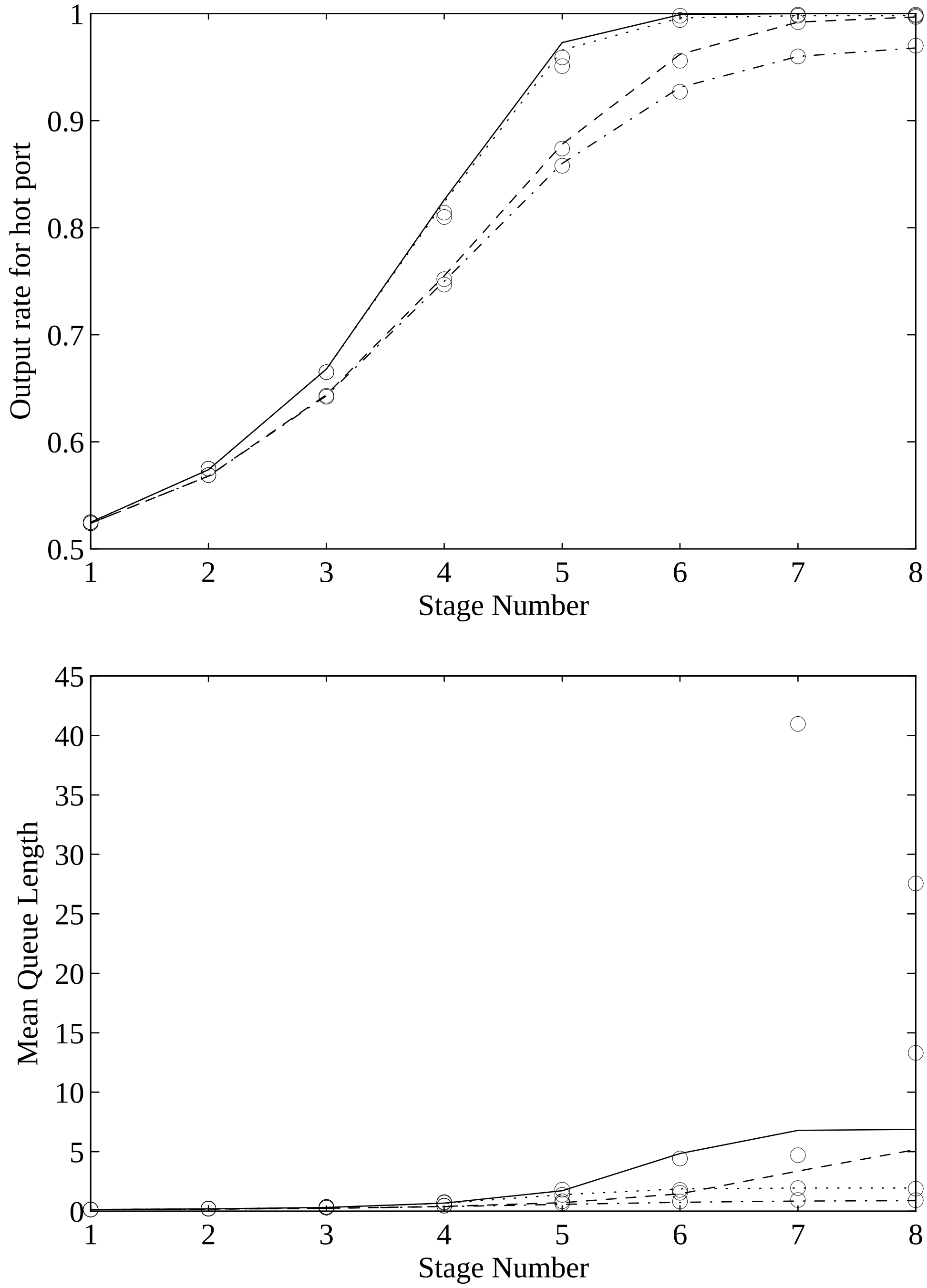


Figure 5.7: Analytical estimates of combining performance, 50 percent load, 5 percent hot spot.

S_{n-1}, T_{n-1}	S_n, T_n Arrivals $0,0$	$r,0$ $0,r$	r,b b,r	r,r	$b,0$ $0,b$	b,b
$0,0$	$0,0$	$0,0$	$0,1$ $1,0$	$1,0$	$0,0$	$0,1$
$k,0$ $(k>0)$	$k-1,0$	$k,0$	k,k $k,k+1$	$k+1,0$	$k-1,k$	$k,0$
$k,1$ $(k \geq 0)$	$k,0$	$k+1,0$	$k+1,k+1$ $k+1,k+2$	$k+2,0$	$k,k+1$	$k+1,0$
k,j $(k>0,$ $j>1)$	$k-1,j-1$	$k,j-1$	$k+1,0$	$k+1,j-1$	$k,0$	$k,k+1$

Table 5.4: Transitions for two-way combining, no combining at the front of the queue.

5.1.6 Network performance of combining options

It is difficult to extend analytical results from our analysis of a single switch to that of a multistage network for several reasons:

- The output stream from the first stage switch no longer conforms to the model of independent trials assumed as input to a switch. As shown in [116] for non-combining queues, the output becomes more clustered than the input as the input rate approaches 1. This results in longer queue length at later stages, as noticed from simulation in [78, 119] and shown analytically in [117].
- In a real system, queues are finite. The analysis we use for a single stage combining switch can be extended to finite queues, but the details are messy and depend on assumptions about how the input process behaves when it is blocked by a full queue (see the description of the two models of a finite non-combining queue in section 2.2).
- Blocking at later stages causes more combining at earlier stages which in turn will lower the input rate to the later stages, as can be seen in the simulations of two-way combining in Chapter 4. The attempt to model this feedback in [68, 67] uses blocking probabilities that assume the queue lengths at a stage are independent of those at the preceding and following stages, a poor approximation for high message rates.

Nevertheless, the output rate from our analysis of a single switch gives the input hotspot rate for the next stage, since the regular traffic rate can be assumed to remain the same, as long as no blocking occurs. These rates can be used to compute $f_{0,0}$, $f_{r,0}$, $f_{r,r}$, $f_{r,b}$, $f_{b,0}$, and $f_{b,b}$, and the recurrences can be computed using these input probabilities to get the output rates from the next stage.

This stage by stage computation provides an approximation of network behavior that is reasonable under certain assumptions: that the input rate and the hot spot rate are not too high, and that the queue lengths are large enough that blocking seldom occurs. These assumptions are reasonable in situations where the input rate from the processor is limited by latency on the round-trip response rather than by blocking.

Figures 5.6 and 5.7 show average queue length and output rate for all four types of switches. Two-way combining switches with no combining at the front of the queue are shown with a solid line, two-way combining switches with combining at the front are shown with a dashed line. Unlimited combining switches with no combining at the front of the queeu are shown with a dotted line, unlimited combining switches with combining at the front are shown with a dash-dot line. With a low input load at the first stage of 10 percent with 5 percent hot spot traffic, for 8 network stages, the analytical results show a good match with simulations. At an input load of 50 percent with 5 percent hot spot, the queue length match with simulations is a serious underestimate at later stages for two-way combining.

The general shape of the results is not surprising. They agree with the results in [87] that show switches with a combining multiplicity of two unable to handle the traffic at later network stages. They also indicate

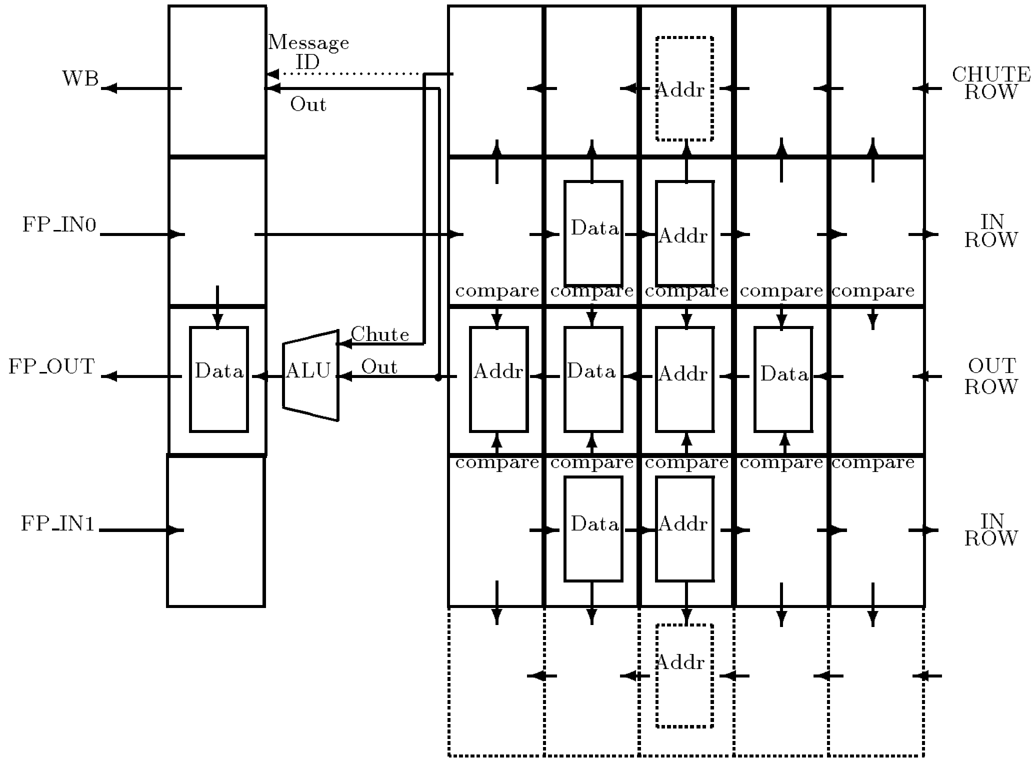


Figure 5.8: A two-input combining queue with one CHUTE

that the loss of combining at the front of the queue is not important for later stage results.

5.2 Implementing queues with greater combining capability

In section 4.4.2, the performance disadvantages of Type B switches compared to Type A switches were illustrated. One disadvantage of the combining switch described in Chapter 4 is that requests in queues paired at an output port cannot be combined; in this section we show how to collapse these two queues into a single systolic combining queue with two inputs and one output. A further limitation of the design in Chapter 4 is the limitation of combining to pairs of requests. We describe how to implement three-way combining, or combining with multiplicity three, in which a given message may combine with two others at a stage, discuss implementation costs, and consider the performance of a limited form of three-way combining, which we call "two-and-a-half-way" combining.

5.2.1 Implementation of a Type A switch

Starting with the combining queue shown in Figure 4.3, we add a second IN row and a second row of comparators to the bottom of the OUT row, to form the two-input queue illustrated in Figure 5.8. This implementation was first suggested by Snir and Solworth in [129].

There is actually only one CHUTE; the dashed row on the bottom is the same as the top row. Since the vertical lines from IN1 to CHUTE are actually wires passing through several rows, the VLSI layout would be poor. The arbitration that is performed between the outputs of two queues in a Type A switch must be performed twice for every column, once for the OUT row and once for the CHUTE row.

Figure 5.9 shows the schematic of the basic cell for this design. The two inputs are labeled INX and INY to avoid confusion with the numerical designations for clock phase. The qualified clocks OTRVX and OTRVY are no longer derived just from the valid1(OUT, j) signal. The valid bits from the IN rows are used

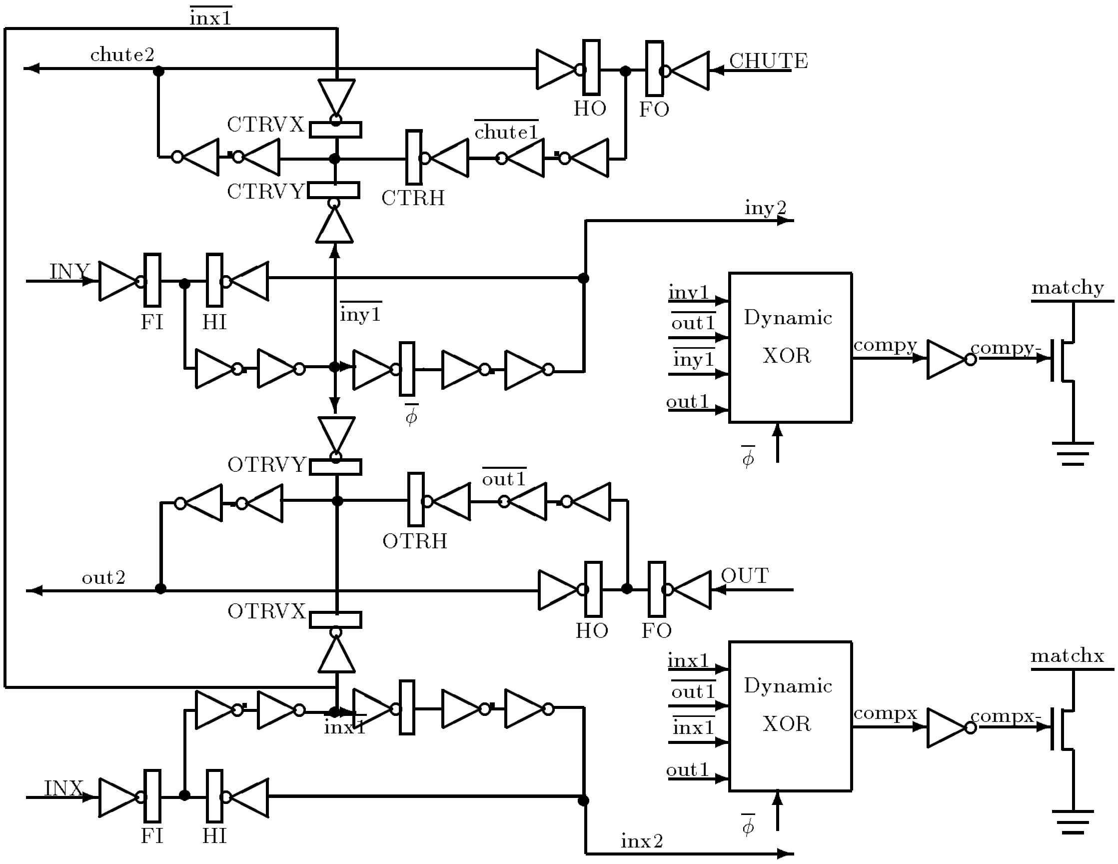


Figure 5.9: Basic cell for a two-input, one-output queue with one CHUTE

in selection logic like that described in section 4.1.7 to compute these OUT transfer signals. The CTRVX and CTRVY signals controlling data movement into the CHUTE are derived from selection logic based on the result of both match computations. The match signals are precharged on $\overline{\phi}$, the base clock of CTRVX and CTRVY, so significant delay will be introduced (see the discussion of qualified clock generation in section 3.6.2). Since the delay for selection logic is already long compared to other delays in the design (see section 4.5.3), this may well be the critical path of the design.

Although the control logic for each slice is more complicated, the number of transistors for this cell is much less than twice that of the basic cell of the single-input combining queue, 117 as compared to 154. Ignoring edge effects due to the exact derivation of the queue full signal (see section 3.4.1), a two-input systolic queue needs $\frac{4}{3}Q$ slices to have the same total storage as two one-input systolic queues of Q slices each, so that the number of transistors used for equal maximum storage, is almost the same, $156Qb$ for the two-input queue (where b is the number of bits in a slice and control logic is ignored) and $154Qb$ for the single-input queue. The storage in the Type A switch is also likely to be more efficiently utilized (see the comparison of dynamic and static multiqueues in [132].)

With only a single CHUTE, three-way combining is impossible (the CHUTE entry fills when a pair combines) but the restriction that both partners come from the same input port has been eliminated. As shown by Liu's simulations [100] as well as by the molasses simulations whose results are given in Figures 5.13 to 5.20, eliminating the same input port restriction and implementing a Type A switch with two-way combining gives a more significant improvement than adding three-way combining to a Type B switch, for the system sizes we have simulated.

The packaging for a Type A FPC with two-way combining and a single CHUTE can be identical to the packaging of the NCR FPC described in section 4.1. The RPC logic need not be changed at all, since the

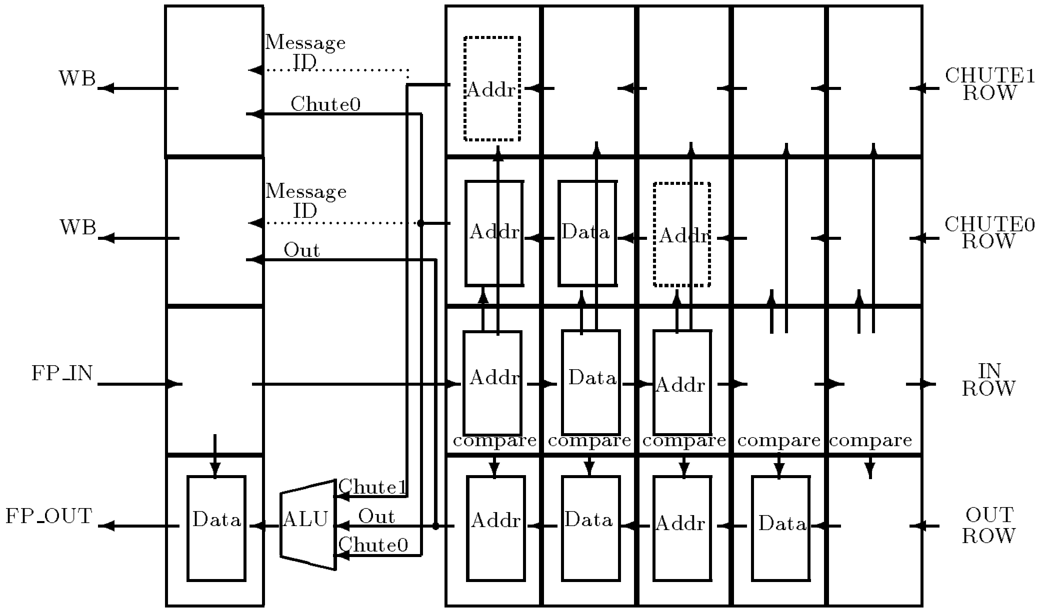


Figure 5.10: A single-input queue capable of combining three requests.

behavior of the Type A FPC is identical to that of the Type B FPC, except for blocking behavior and order of outputs. Non-combining multi-input queues could be used, though a four-input queue would be required to replace each main queue/decombined queue pair in the RPC.

5.2.2 Three-way combining in a Type B switch

The design of Figure 4.3 can be extended to support three-way combining by adding a second CHUTE as shown in Figure 5.10. When an entry in the IN row matches one in OUT, the first entry moves to a CHUTE unless the corresponding entries in both CHUTEs are full, which means that three items have already combined.

This design presents several complications over the two-way combining design for a Type B switch:

- Logic is needed to decide which CHUTE is to obtain a matched entry. Unlike the arbitration logic required for the CHUTE in the two-input queues of section 5.2.1, such logic is relatively simple, depending only on the CHUTE valid bits, and not on the match logic.

- Two messages must be sent to the wait buffer. If they are sent on the same cycle, more pins may be needed if the FPC and RPC are packaged separately. If the two messages are sent serially, consecutive triple combines create a problem for flow control, possibly requiring output from the queue to halt until the messages sent to the wait buffer catch up.

- The wait buffer must be able to output two messages destined for the same return path output port as the result of matching a single message returning from memory. The same flow control problem exists as when sending messages to the wait buffer from the forward path, and, in addition, the internal wait buffer logic may become more complicated.

- The ALU in the FPC must now accept three inputs, to produce the correct result to forward to memory. Furthermore, for correct decombining, either the ALU in the RPC must accept three inputs, so that both the Out data packet and the Chute0 data packet can be used for decombining with the the data packet from memory, or the Out and Chute0 data packet must be added together before being stored in the wait buffer. If this addition is done on the FPC, both a three-input and a two-input ALU are needed. If done on the return path, an additional ALU is needed at the input to the wait buffer.

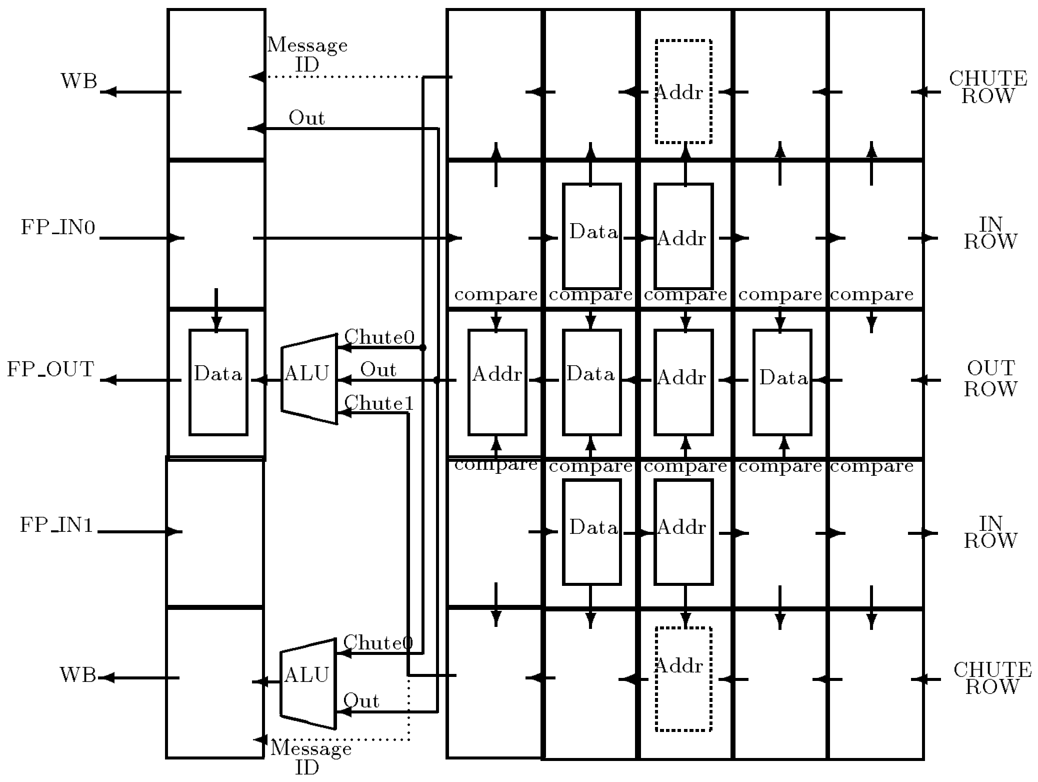


Figure 5.11: Two-and-a-half-way combining queue

- Bursts of three messages may increase delays on the return path[88].

One can use still more CHUTEs to support higher levels of combining but the VLSI layout becomes less and less planar and the proliferation of multi-input adders is expensive. Simulation results in [87, 100] comparing three-way combining with unlimited combining indicate that, for two-way switches, combining multiplicity greater than three is not needed.

A two-input queue capable of combining three requests to the same memory location can be obtained by doubling both IN rows, as in section 5.2.1, as well as both CHUTEs. With two IN rows each able to shift to either of two CHUTEs, the queue can combine three requests regardless of their input ports. This implementation has all the difficulties of the single input queue with three-way combining, and also has the problem, as for the Type A switch design described in section 5.2.1, that the CTRV signals depend on the match logic.

5.2.3 Two-and-a-half-way combining

We propose "two-and-a-half-way combining" as an alternative with better layout and decombining properties than three-way combining that still shows good performance for large networks. As shown in Figure 5.11, two-and-a-half-way combining limits the connections of the IN rows to the neighboring CHUTE rows. Three messages can combine as long as the second and third messages come from different inputs.

The logic to do two-and-a-half way combining, like that for full three-way combining, whether in a Type A or Type B switch, requires additional ALU operations on the Out and two Chute outputs, either with a tree of two-way ALUs or with a three-input ALU. Since the intermediate result combining the data of Out and Chute0 is required anyway, feeding the output of the lower two-input ALU in Figure 5.11 into the upper ALU, so that it requires only two inputs, may be somewhat more efficient in transistor count than implementing both a three-input and a two-input adder, and could be pipelined deeper into the queue to prevent delay through two ALUs from limiting the clock rate. Alternatively, as illustrated, a three-input

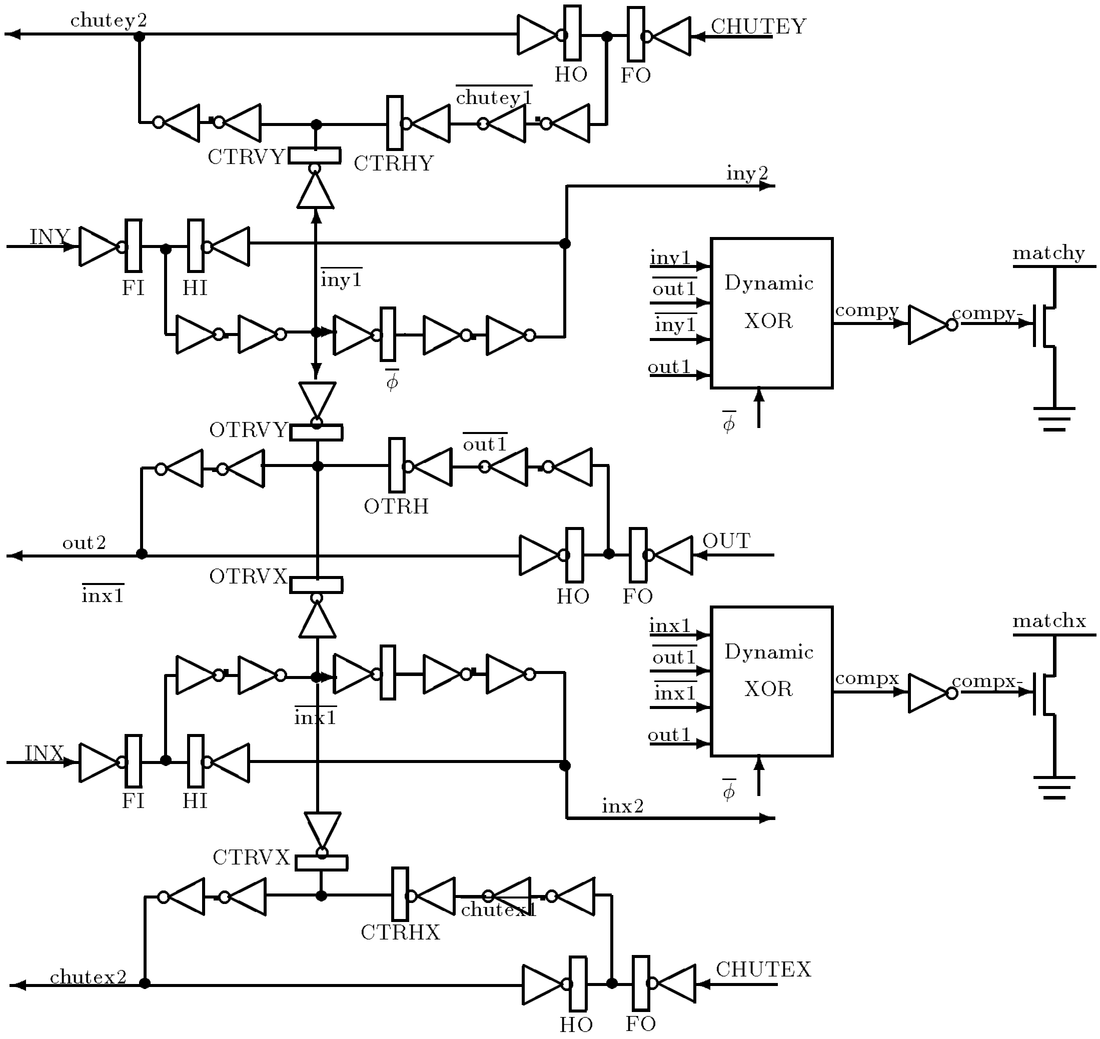


Figure 5.12: Schematic for the basic cell of a two-and-a-half-way combining queue.

carry-save ALU can be implemented in parallel with a two-input ALU. A three-input carry save adder has only slightly increased delay, with no significant complication of the carry lookahead tree that forms the greater part of the ALU logic (see [90] for a textbook description of carry-save addition).

The basic cell of the two-and-a-half-way combining queue is illustrated in Figure 5.12. The additional CHUTE row costs about 20 more transistors than the design of Figure 5.9, but the connection from one IN row to two CHUTE rows is eliminated. More important, the CTRV signals no longer depend on the match logic. Since only one IN row ever sends data to a CHUTE, the CHUTE data values can be set to the IN row values whenever the CHUTE is empty, and only the CHUTE valid bit depends on the match logic.

The same number of wait buffer input ports to the RPC are needed as for the design of Chapter 4. However, the buses cannot be tied together, since they may both be active simultaneously, and thus more pins would be required for the forward path to wait buffer connections in the 4 chip and 2 chip designs of Table 4.5. Since the combined messages came in on different input ports on the forward path, they are guaranteed to exit from different wait buffers on the return path. Therefore, decombining is unlikely to create a problem with bursty traffic. Furthermore, if the combination of the data packets from the OUT row and one of the CHUTE rows is done on the forward path, the wait buffer design can remain simple. With one wait buffer for each input-output pair, the two combined messages will be in different wait buffers, and can be matched and queued in parallel.

5.2.4 Combining options for 2×2 switches

We have used molasses to simulate the performance of two-and-a-half-way combining as well as Type A and Type B switches with two- and three-way combining, for systems of size up to 1024 PEs. Figures 5.13 to 5.20 use the same assumptions of queue sizes of 10 for Type B and 20 for Type A, and 16 outstanding requests at the PE, that were used in the simulations in Chapter 4. In these figures, solid lines show Type A, for both 2 and 3 way combining, dashed lines show Type B, for both 2 and 3 way combining, and dash-dot lines show two-and-a-half way combining. The dotted line showing worst case performance is the result of simulation with Type B switches with no combining; the dotted line showing best case performance is the result of simulation under uniform traffic with Type A switches.

In all cases, the performance of Type A switches, whether with two-way, two-and-a-half way, or full three-way combining, is very close. For the moderate load and low hot spot rate of Figure 5.13, all combining options are virtually identical in performance to the bandwidth and latency of uniform traffic, even though systems with more than 128 PEs show signficant performance degradation without combining. As the load and hot spot rate increase, Type B switches, whether with two-way or three-way combining, begin to show consistently poorer performance than Type A switches. For 1024 PEs, with a 10 percent hot spot rate and 100 percent offered load, Type B switches with three-way combining showed a bandwidth in packets per cycle of only around 45 percent. Twenty-one percent of this loss in bandwidth was due to blocking at the PE inputs to the network, and thus could not be improved by increasing the number of outstanding requests. In contrast, for Type A switches with three-way combining, bandwidth of 60 percent was achieved, with less than 0.5 percent blocking on the part of the PEs, indicating that bandwidth could be improved by increasing the latency tolerance of the PEs and allowing more outstanding requests.

A slight consistent difference is shown between the performance of Type A switches with two- and three-way combining, but it amounts to only a few percent in bandwidth and a few cycles in latency. Figures 5.21 and 5.23 show just Type A switches with two-, two-and-a-half- and three-way combining. Two-way combining is shown with a solid line, two-and-a-half-way with a dash-dot line, and three-way with a dotted line. Two-and-a-half-way combining actually appears to perform better than three-way combining for 1024 PEs in Figures 5.21 and 5.22, but the graph of bandwidth versus latency for 1024 PEs in Figure 5.23, as well as a more detailed examiniation of the simulations shows that this is just random variation.

To test whether or not increasing the queue size and allowing more outstanding requests would significantly change the relative performance of the different combining architectures, we simulated systems with single packet messages and 128 outstanding requests. The queue sizes are still 10 and 20 in packets, but the queue size measured in message units has doubled. Figure 5.24 shows the results of these simulations with round trip latency as a function of bandwidth. The vertical line at around 10 percent shows a Type B switch with no combining. The dashed line represents a Type B switch with two-way combining, the solid line represents a Type A switch with two-way combining, and the dash-dot line represents two-and-a-half-way combining. A dotted line representing Type A switches with three-way combining is almost invisible, due to the close overlap with two- and two-and-a-half-way combining. The larger queue size and increased number of outstanding requests allows a much higher peak bandwidth, but there is still no appreciable difference between the Type A combining switches, as is consistent with the results reported in [100].

In none of these simulations do we see the kind of catastrophic saturation and increase in latency in later stages predicted by the analysis with infinite buffers and unlimited outstanding requests. Instead, blocking in the early stages of the network causes increased combining as messages wait. Blocking at the inputs and increased latency due to conflicts in the early stages cause a gradual decrease in bandwidth, for larger system sizes and heavier loads, but performance is always much better than predicted by a model in which infinite buffers allow messages to proceed through the early stages of the network without combining until they saturate later stages. Our simulations show blocking probabilities at later stages in the forward path and on the return path are uniformly low (under 1 percent), again confirming results in [100].

5.3 4×4 Combining Switches

Networks composed of 4×4 switch nodes have two important advantages over networks with 2×2 switches:

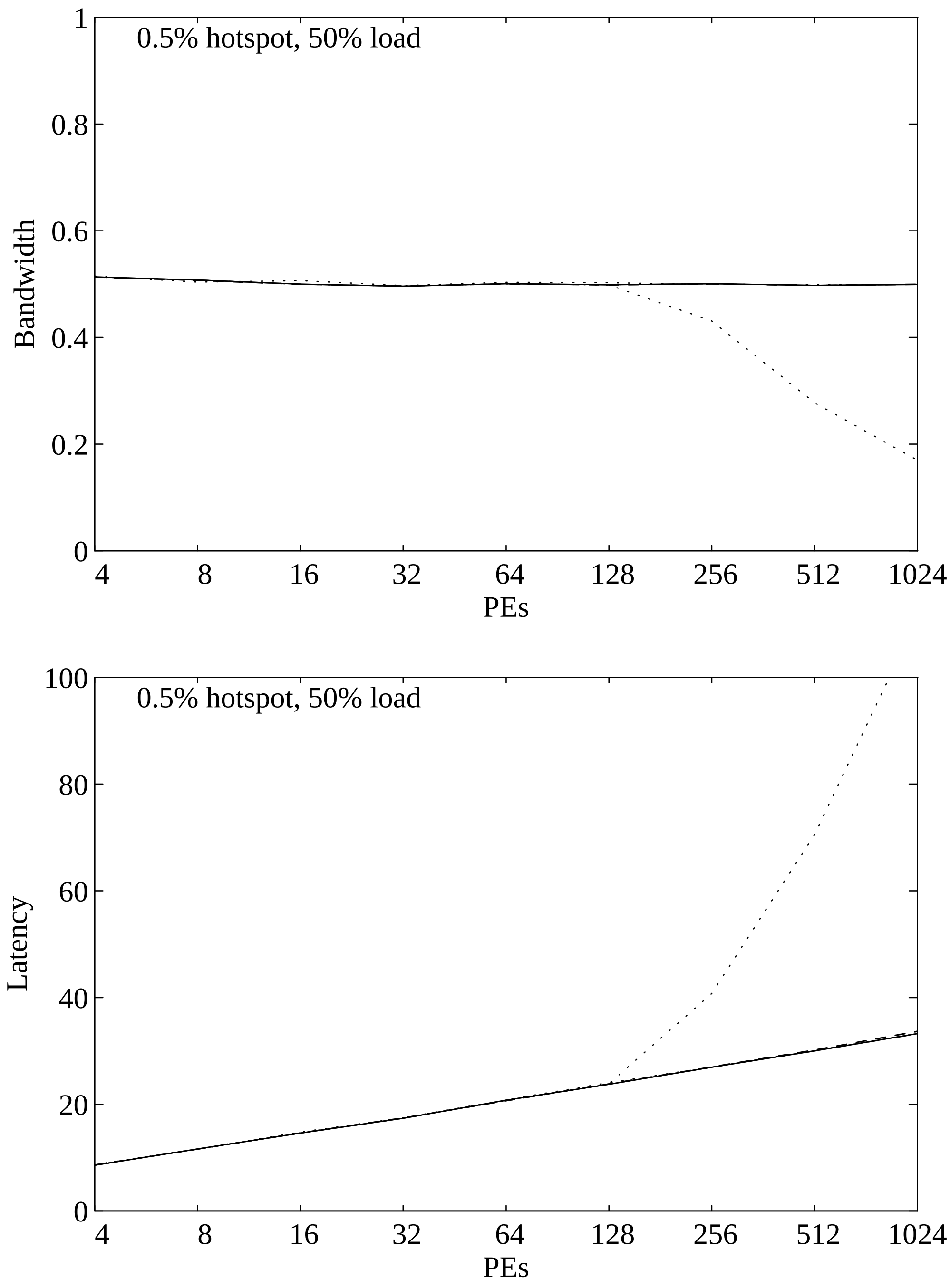


Figure 5.13: Combining options, 2×2 switches, 0.5% hot spot, 50% load.

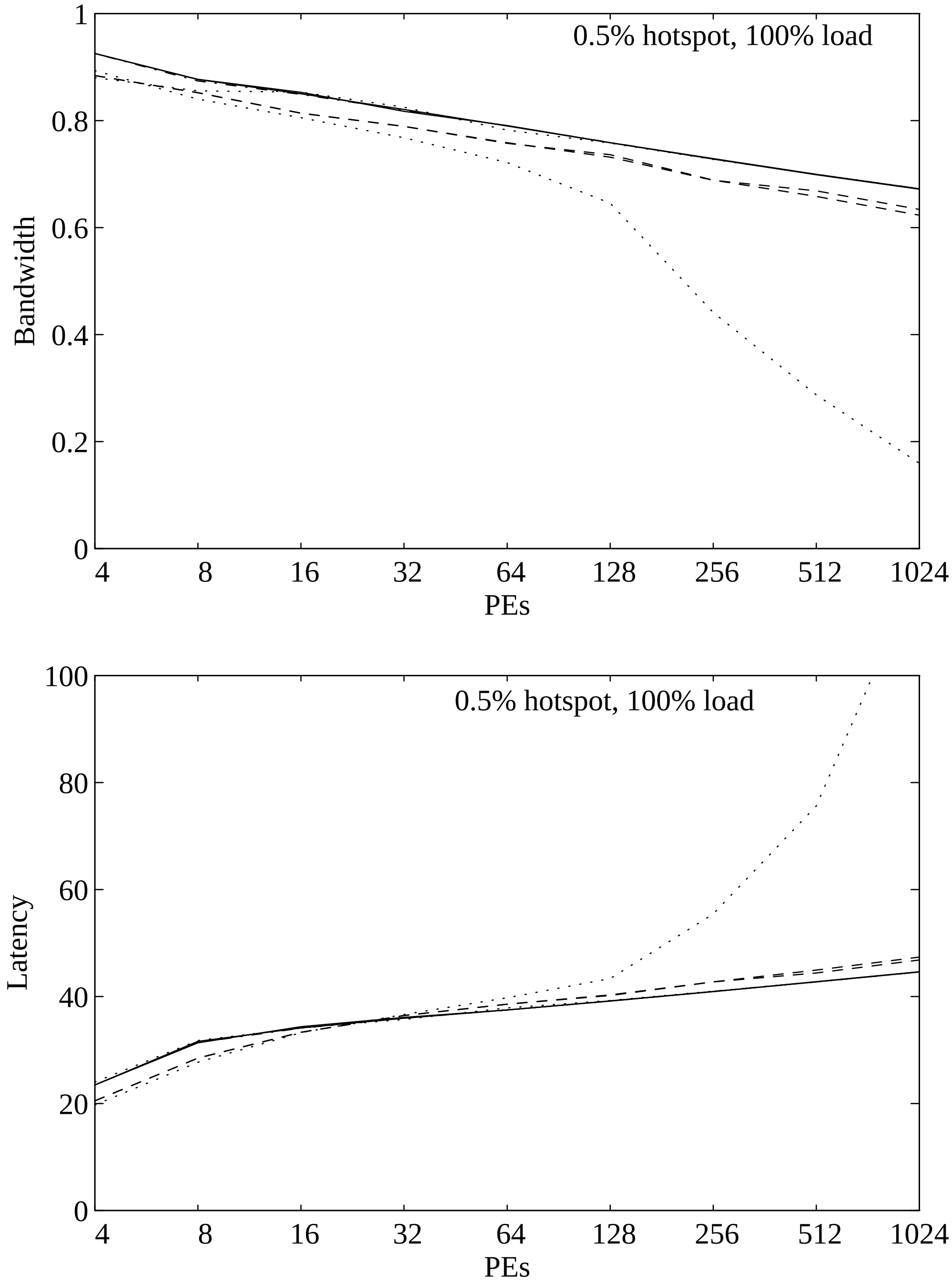


Figure 5.14: Combining options, 2×2 switches, 0.5% hot spot, 100% load.

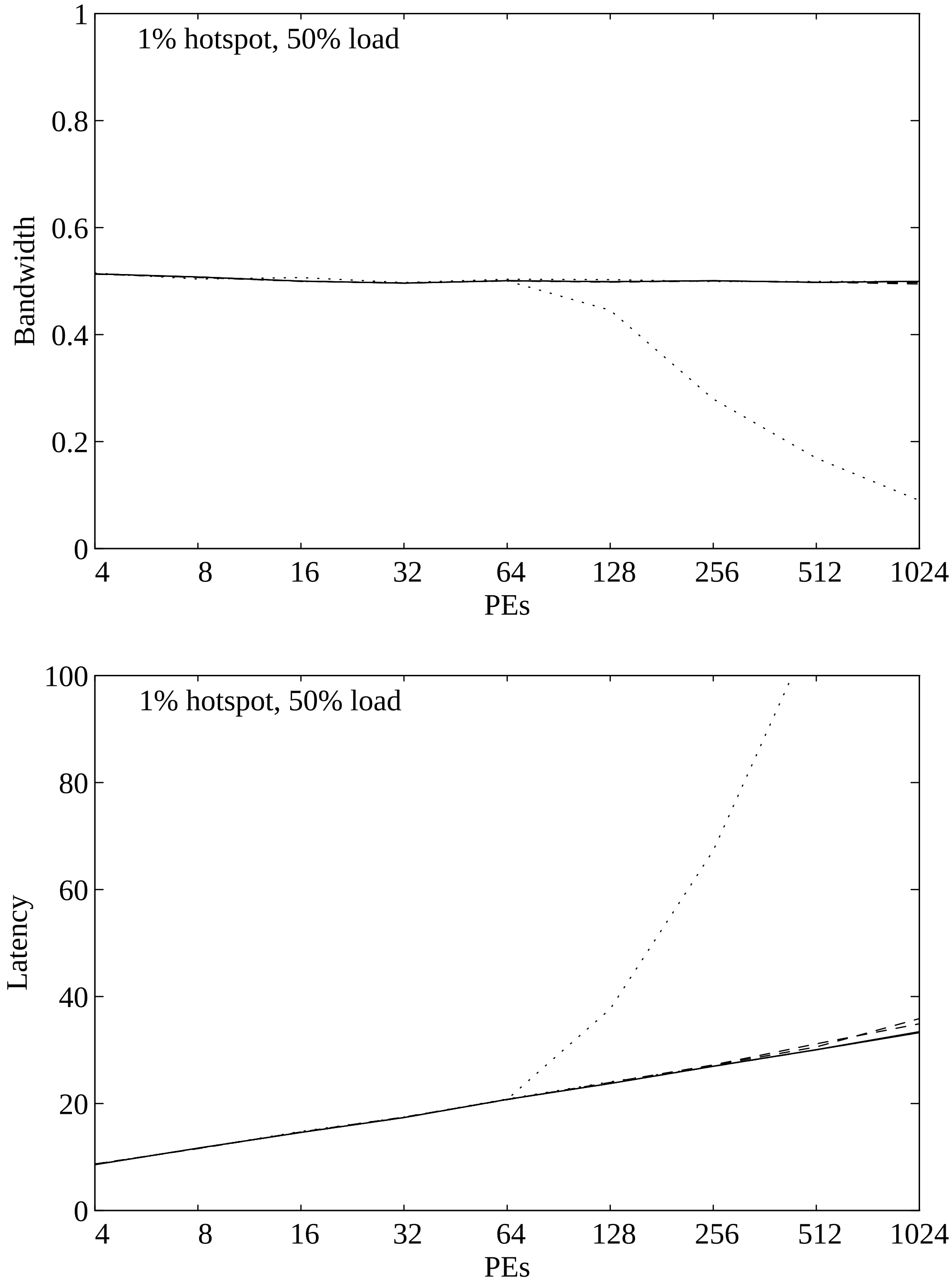


Figure 5.15: Combining options, 2×2 switches, 1% hot spot, 50% load.

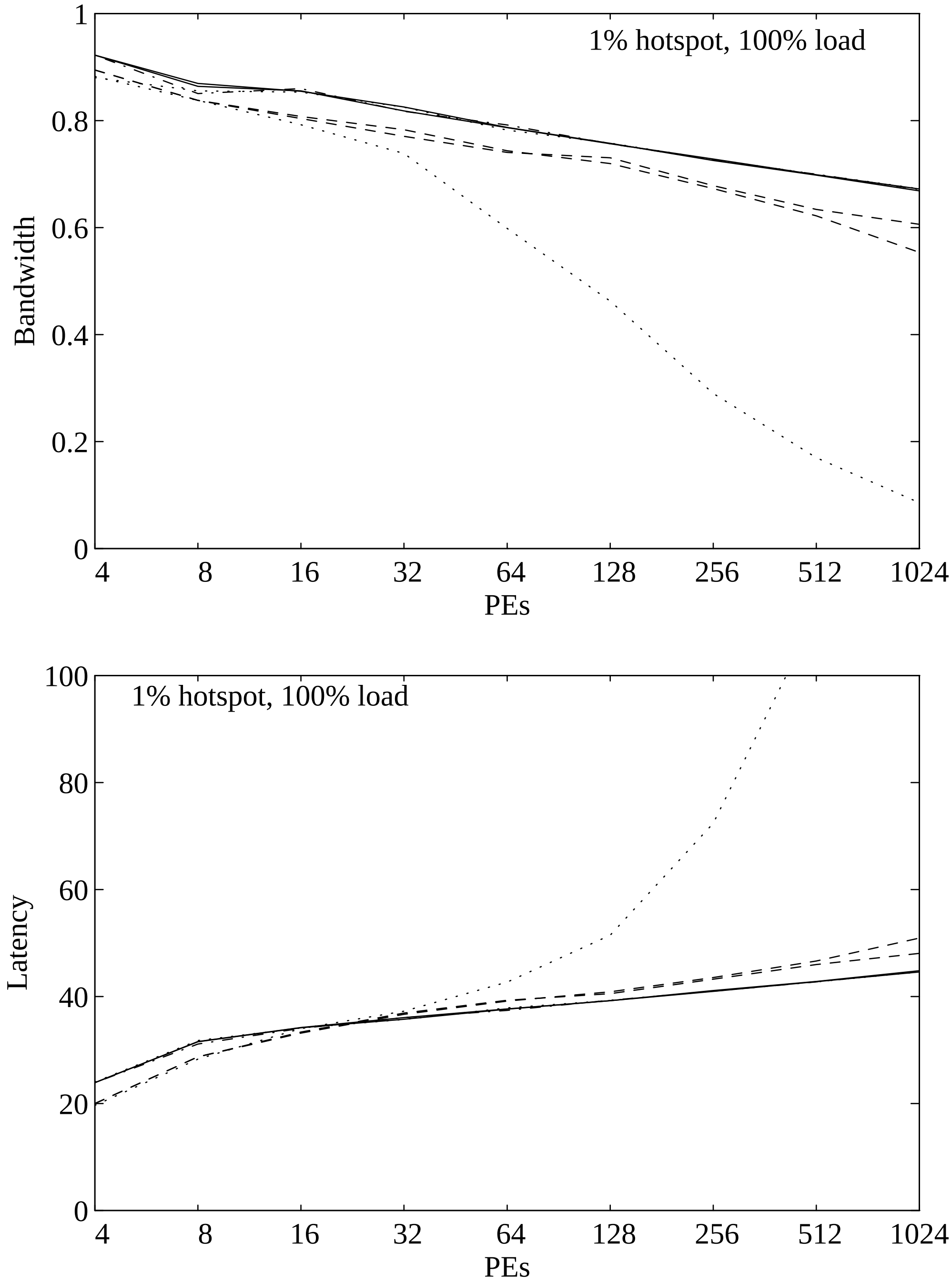


Figure 5.16: Combining options, 2×2 switches, 1% hot spot, 100% load.

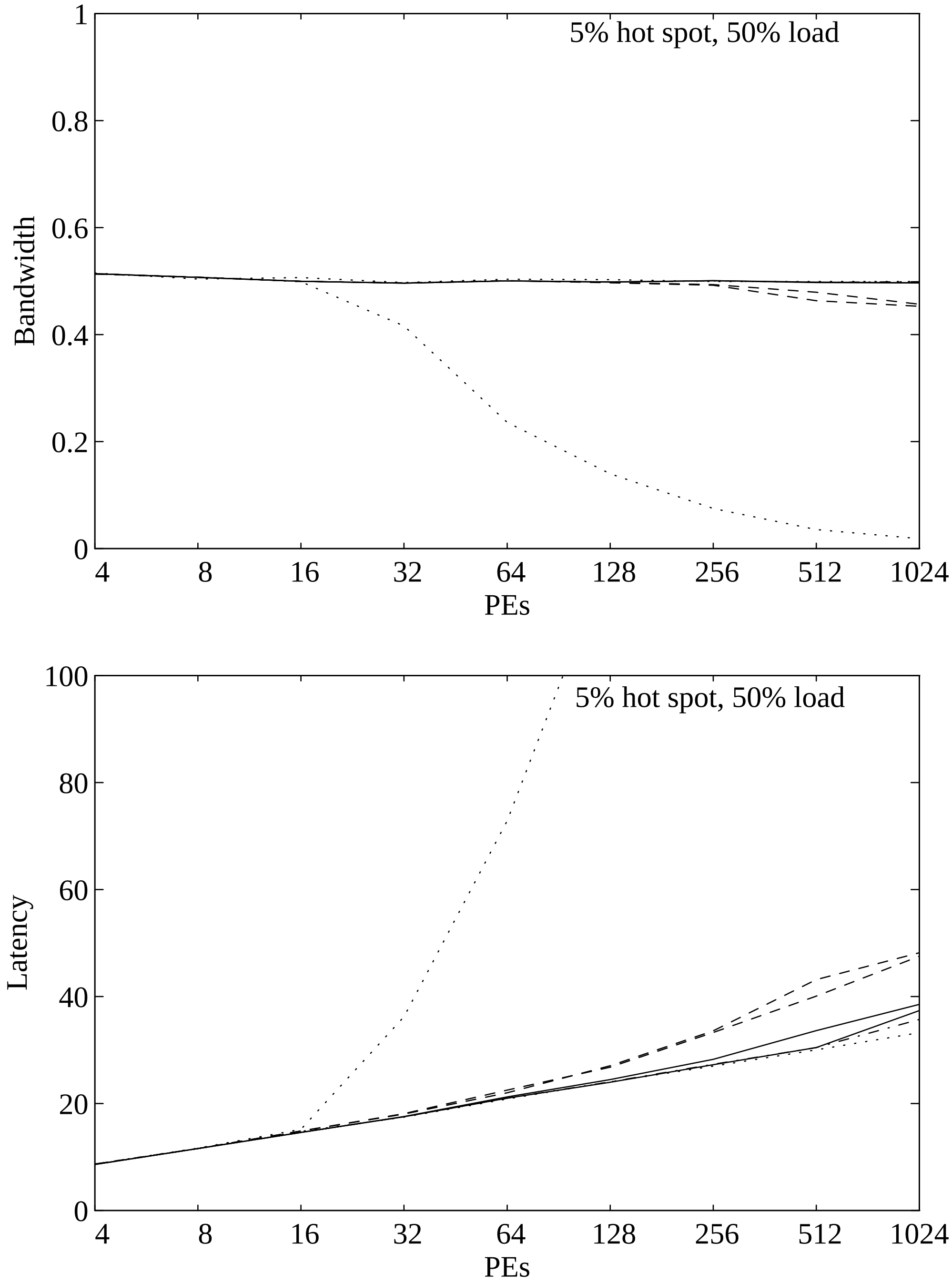


Figure 5.17: Combining options, 2×2 switches, 5% hot spot, 50% load.

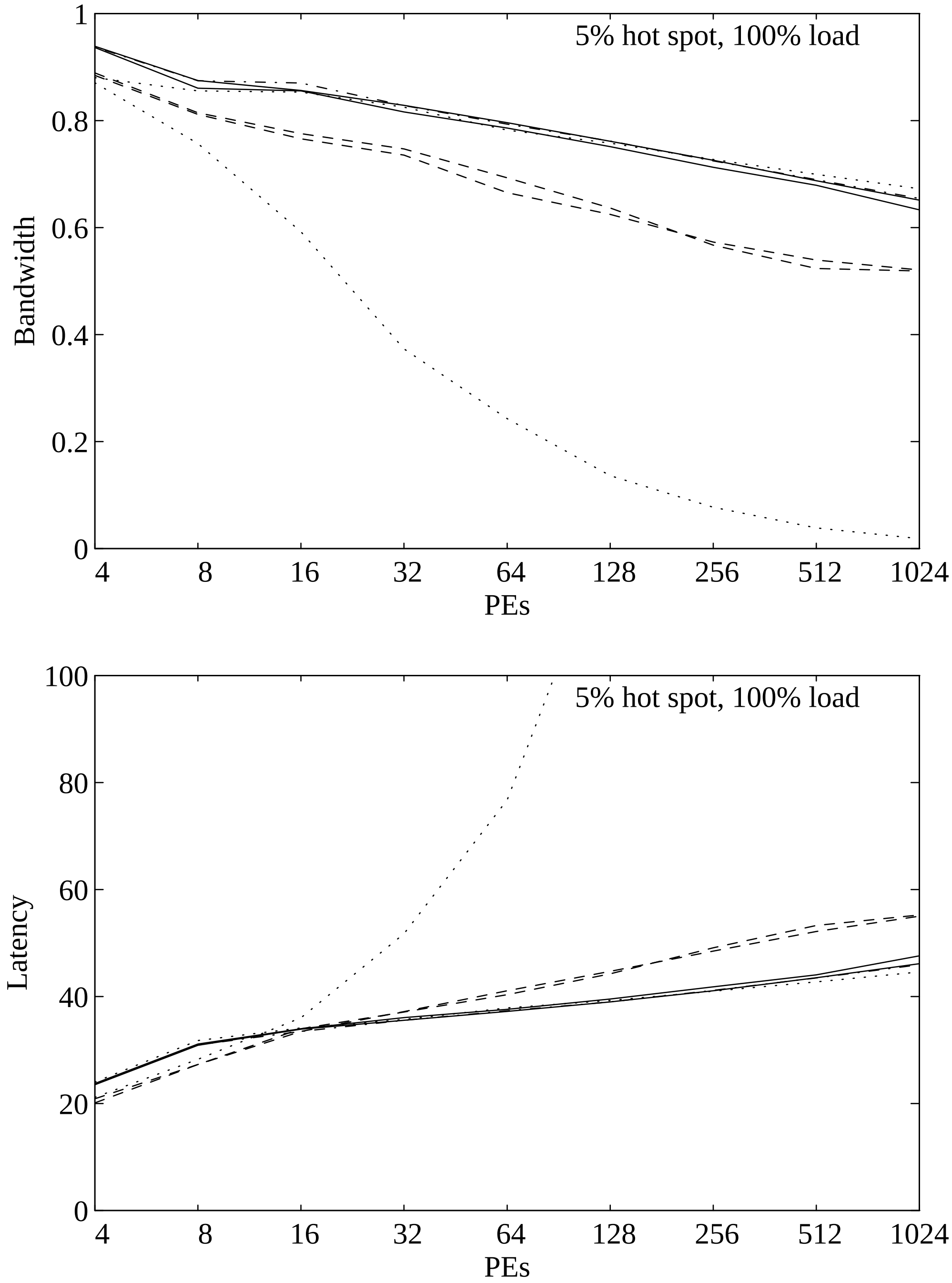


Figure 5.18: Combining options, 2×2 switches, 5% hot spot, 100% load.

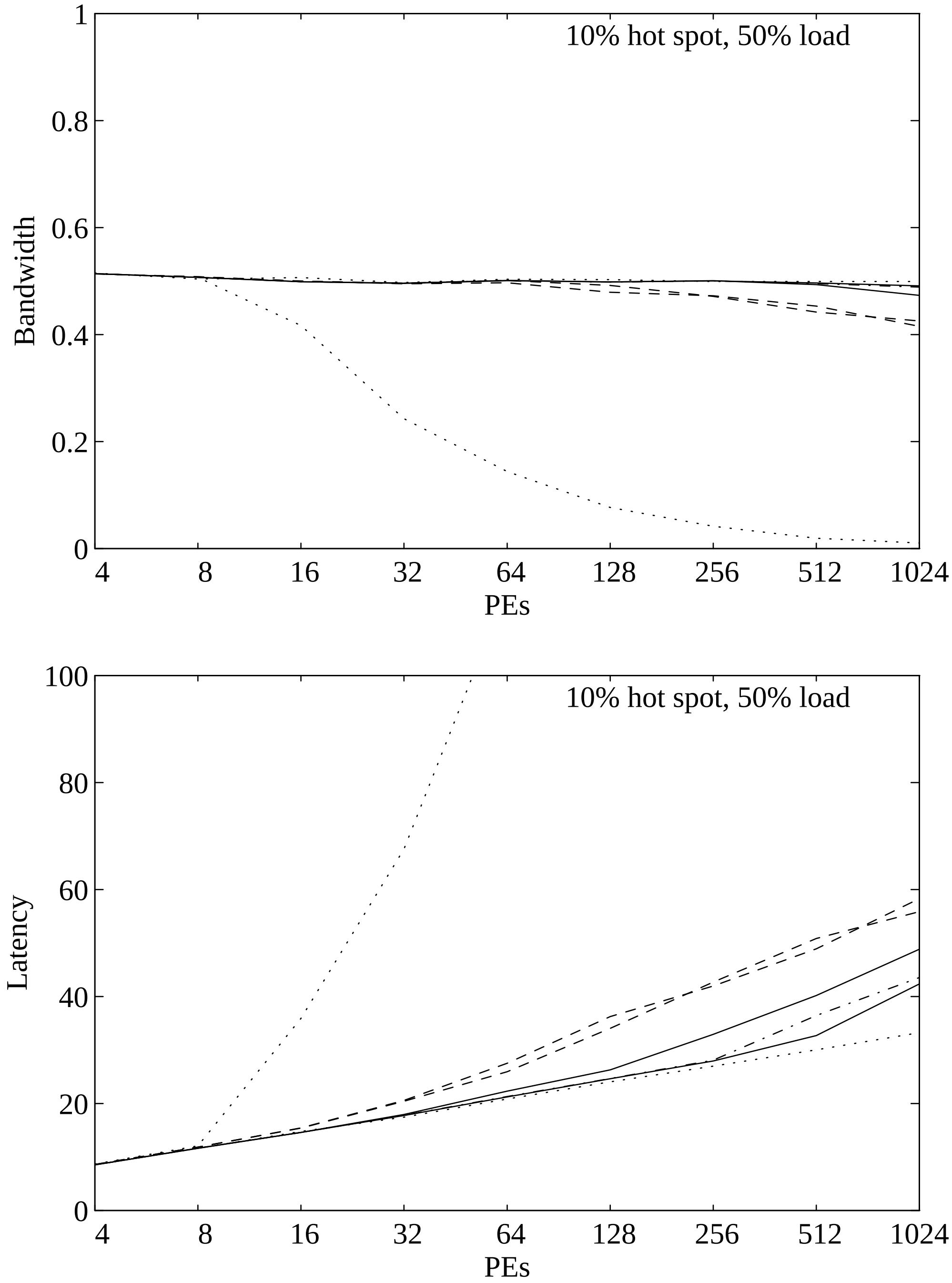


Figure 5.19: Combining options, 2×2 switches, 10% hot spot, 50% load.

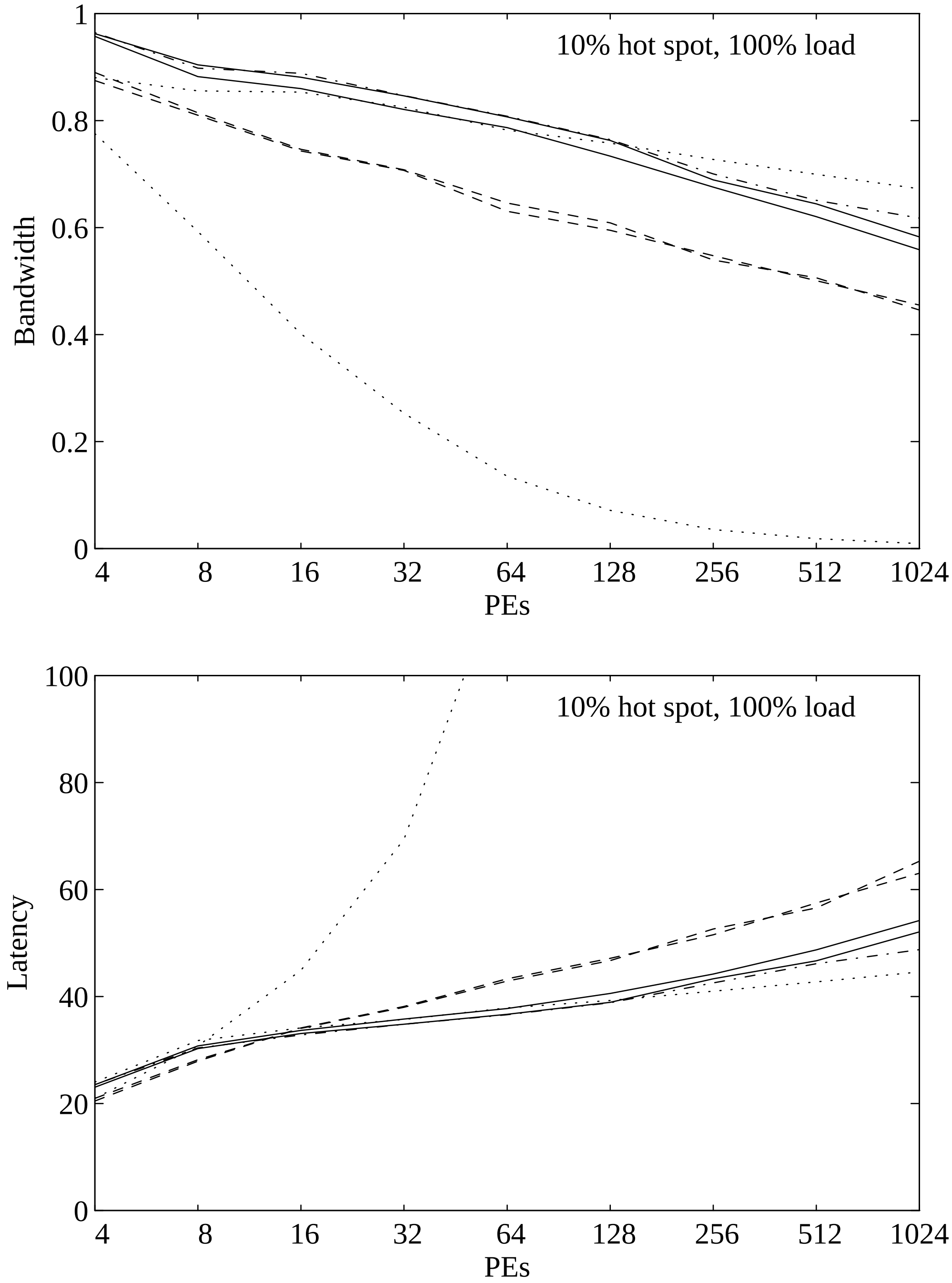
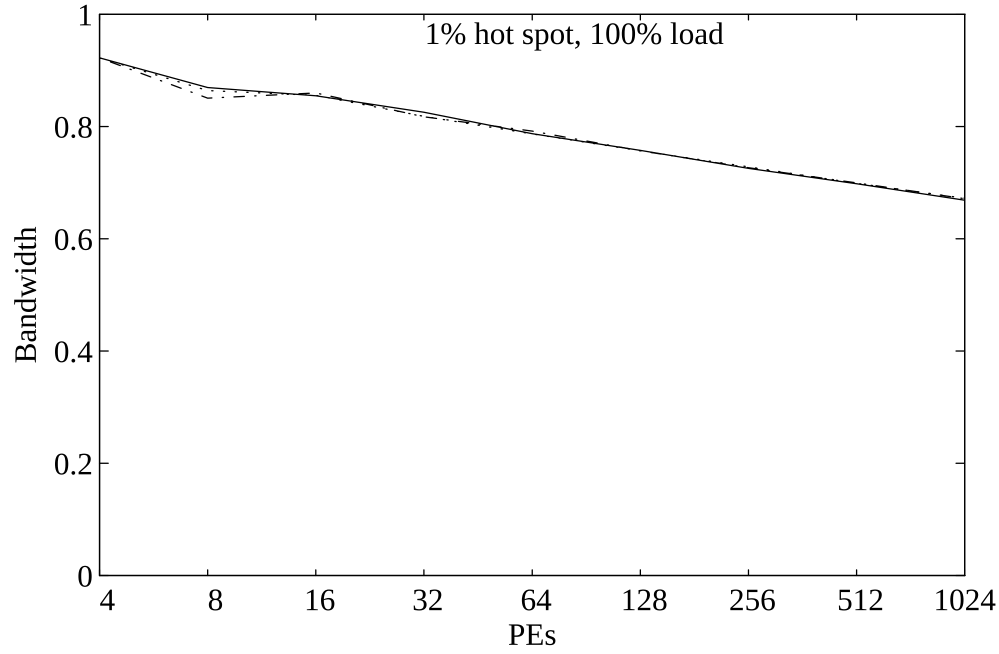


Figure 5.20: Combining options, 2×2 switches, 10% hot spot, 100% load.



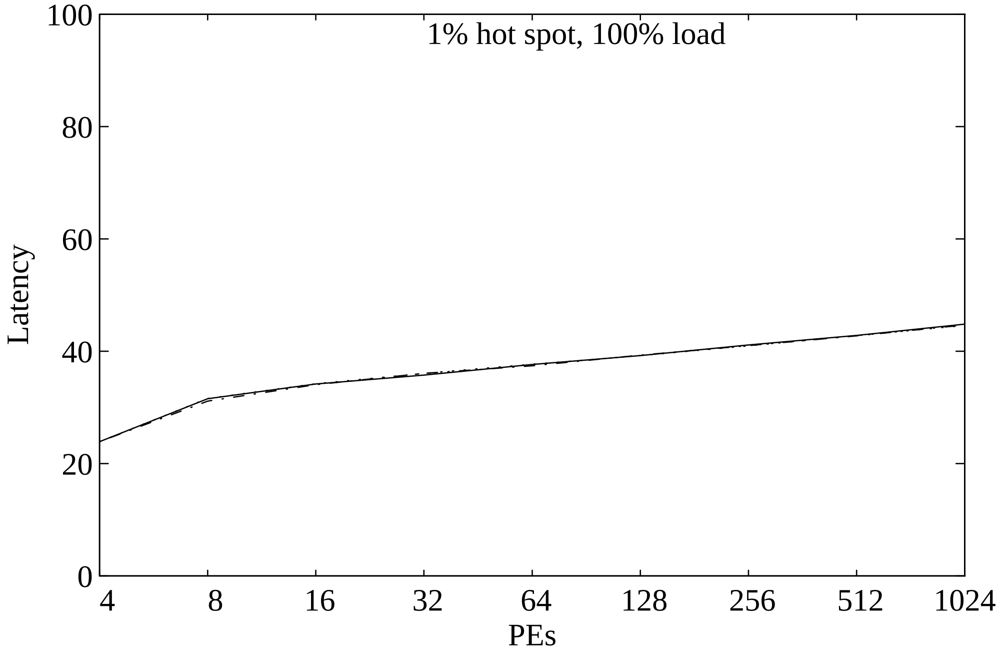


Figure 5.21: Type A 2×2 switches, two, two-and-a-half and three-way combining, 1 percent hot spot.

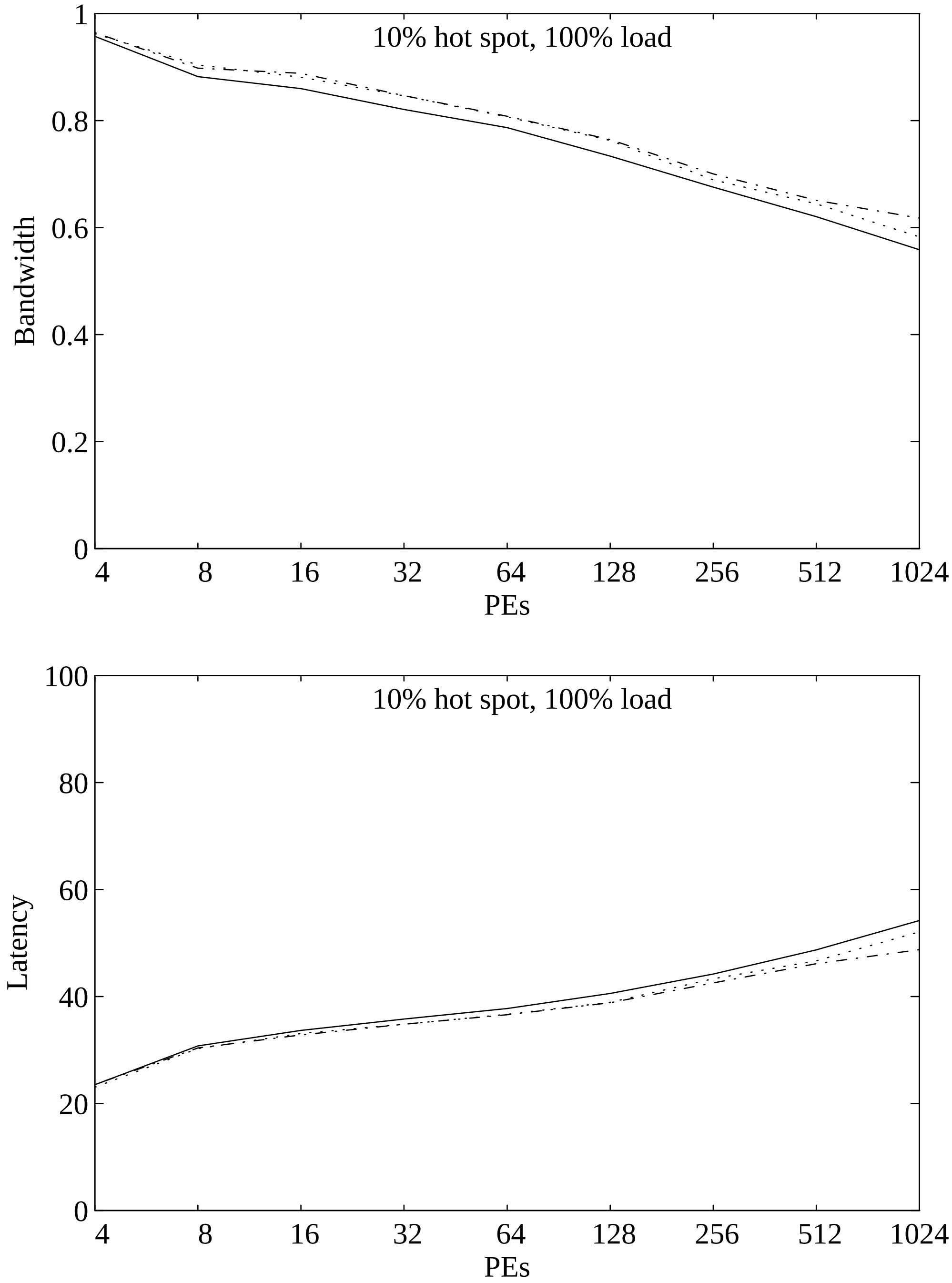


Figure 5.22: Type A 2×2 switches, two, two-and-a-half and three-way combining, 10 percent hot spot.

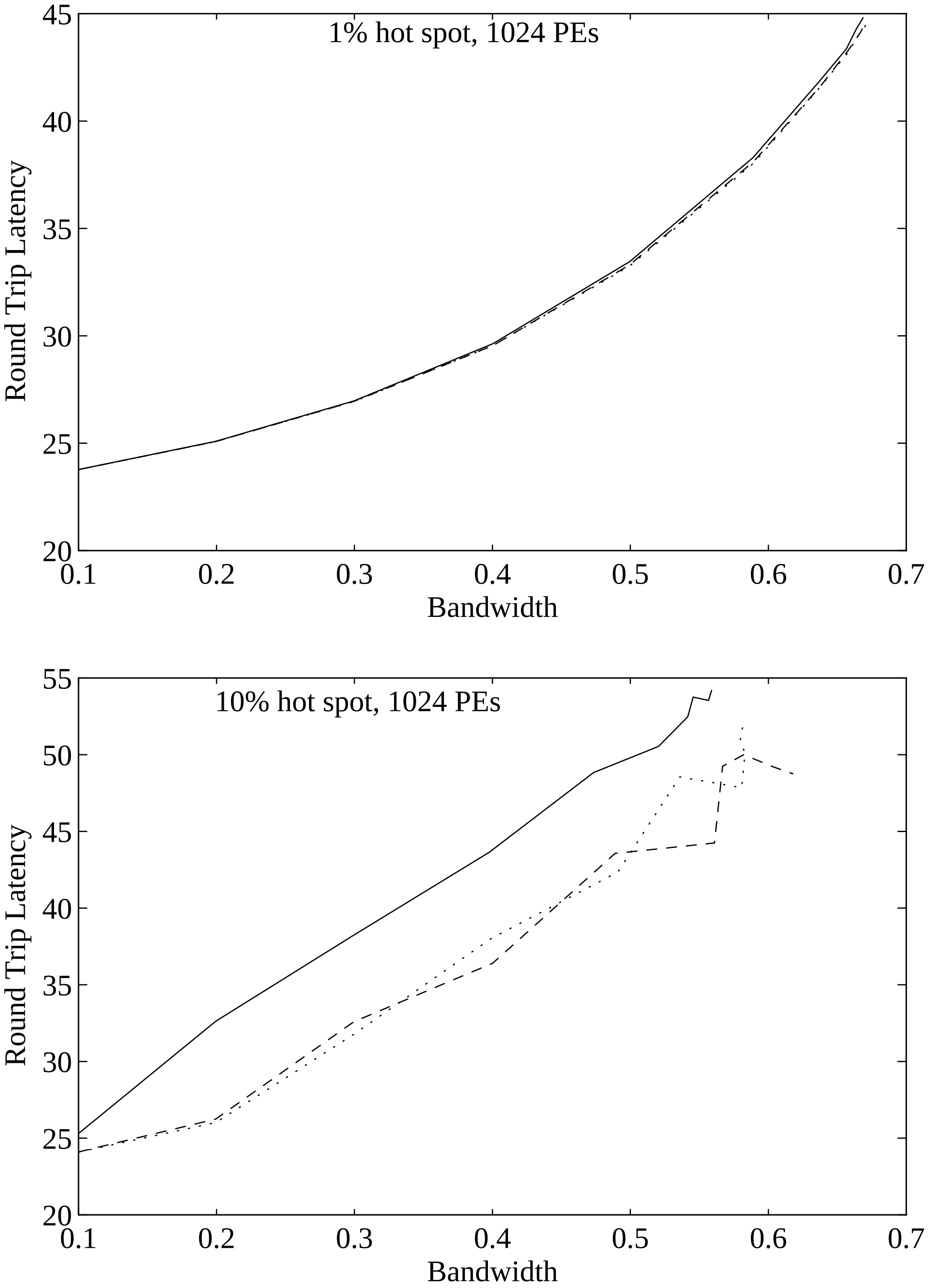


Figure 5.23: Type A 2×2 switches, two, two-and-a-half and three-way combining, latency as a function of bandwidth.

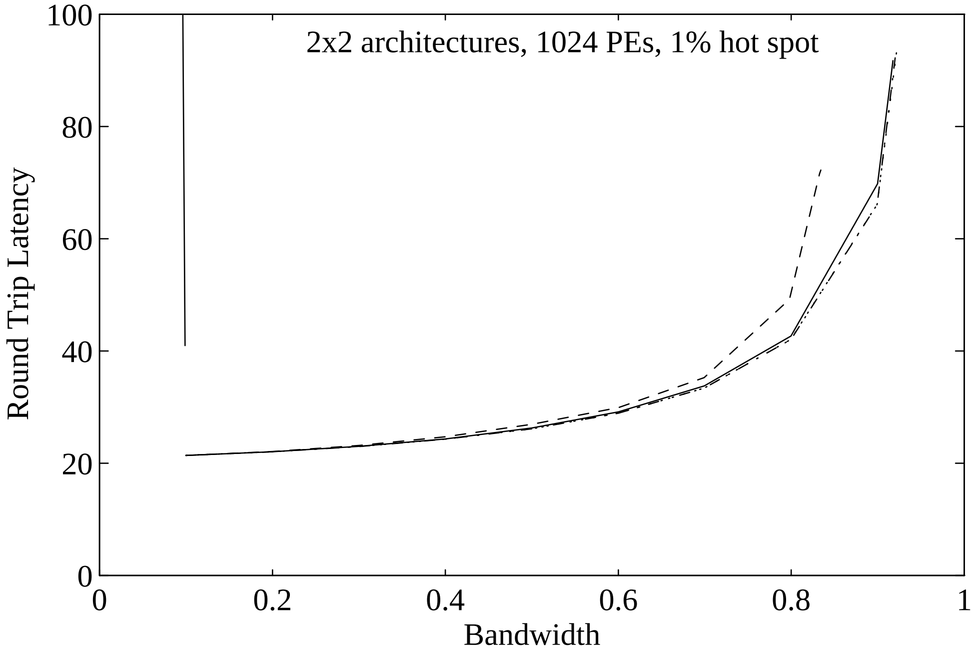


Figure 5.24: Type A and Type B 2×2 switches, latency as a function of bandwidth

1. Only half the number of stages in a network, reducing the mimimum latency experienced by the head of the message passing through the network.
2. Only a quarter the number of components in the network, and only half the number of wires, if the same width of data path is used.

But constructing a network out of 4×4 switches also has certain disadvantages:

1. The increased degree of the node requires either more expensive, higher pin-count packaging or narrowing the data path width, which has the unpleasant effects on bandwidth and latency described in section 2.5. For combining switches, narrowing the packet width may have the additional unpleasant effect of complicating the matching logic (see section 4.2.1).
2. The internal VLSI design of the nodes becomes significantly more complicated. Fair four-way arbitration analysis must be provided, which is more costly than two-way arbitration , as discussed in section 4.1.7. The total amount of logic area needed for a node increases at least linearly, and possibly quadratically for any structures where a separate structure is desired for each input/output port pair.

A greater variety of switch architectures are possible than for a 2×2 switch. Interesting designs using two-input queues must be considered, as well as the Type A, Type B and Type C designs using one-input and four-input queues. In analogy with two- and three-way combining for 2×2 switches, the multiplicity of combining in a 4×4 switch may vary from two to five. We refer to the analogue of two-and-a-half-way combining as *combining per input*, recalling the implementation in which each queue IN row has an associated CHUTE. With combining per input, in a 4×4 switch, a message may combine with four additional messages, for a total combining multiplicity of five, as long as each of those messages come from a different input port.

Some of the implementation alternatives for a 4×4 switch include:

- A Type A switch with four four-input, one-output combining queues. A design similar to that in Figure 5.11 can be used with four IN and four CHUTE rows and four-way arbitration logic at each slot. Such a design is illustrated in Figure 5.25, with a sample adder tree that implements combining per input with two stages of two- and three-input ALUs. To implement full five-way combining, much

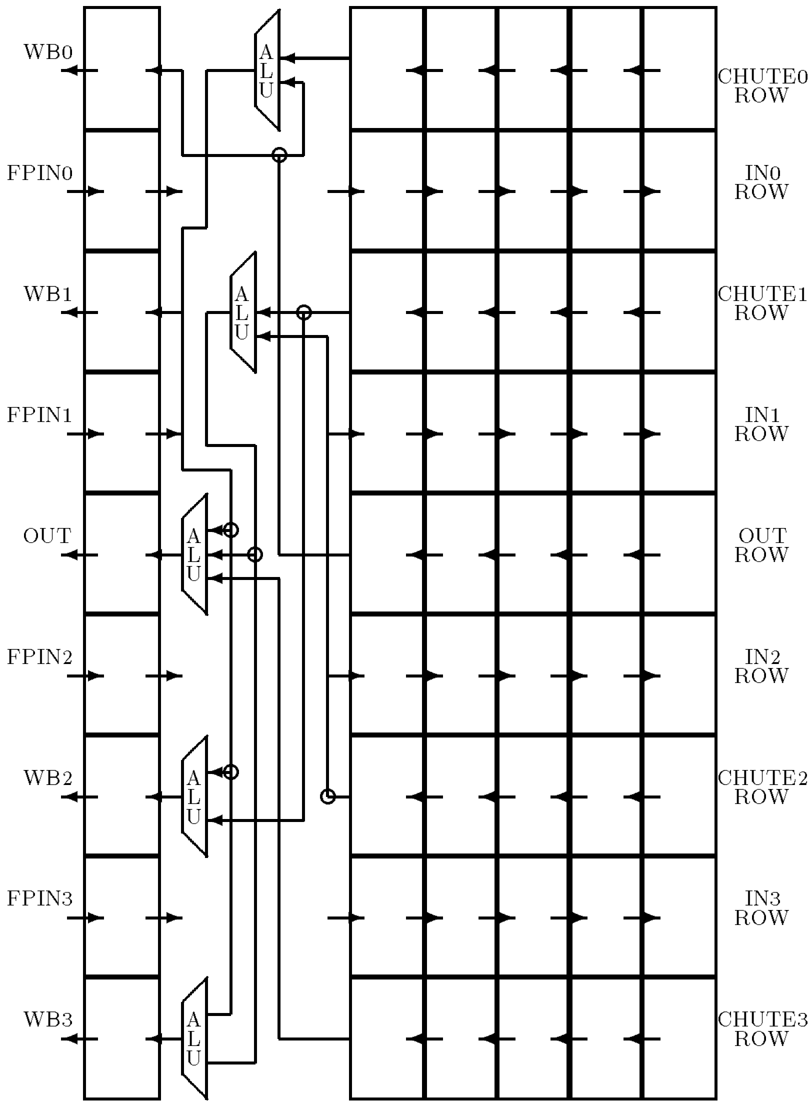


Figure 5.25: Four-input, one-output queue with combining per input.

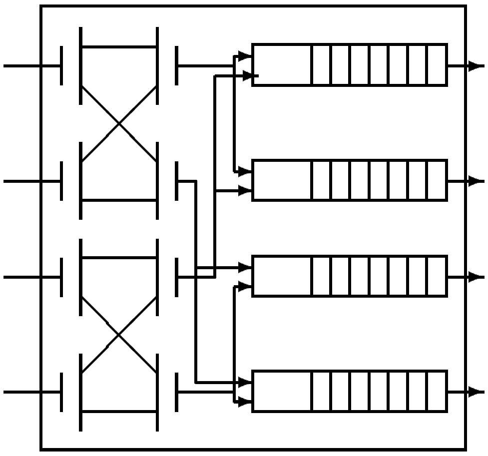


Figure 5.26: Variant Type C 4×4 combining switch using two-and-a-half-way combining queues.

more complicated multiplexing of the ALU inputs would be required, as well as connections from all IN rows to all CHUTE rows.

- A Type B switch with sixteen one-input, one-output combining queues, like those used for the design in Chapter 4.
- A Type C switch, with four one-input, one-output combining queues, again like the design in Chapter 4.
- A hybrid Type B switch, with two 'two-and-a-half-way' combining queues paired at each output, for a total of eight queues (see Figure 5.27).
- A variant Type C switch, with the two 2×2 crossbars at the input, using one 'two-and-a-half-way' combining queue at each output (see Figure 5.26). If the crossbars are fast enough, such a switch can approximate the performance of a Type A switch with two-and-a-half-way combining (see section 2.1.4).

We have simulated some of these alternatives using molasses, using parameters that are comparable to those of the 2×2 switch in Chapter 4. Messages are assumed to be composed of four packets, to model the pin limitation constraint. Storage per queue is kept at 5 messages for a Type B switch, corresponding to 20 messages per output port. Sixteen outstanding requests are allowed at each processor.

Figure 5.28 shows latency as a function of bandwidth for Type A switches capable of combining 2,3,4 and 5 messages, from any combination of inputs, for a 5 percent hot spot rate. The dotted line represents performance under uniform traffic. Figure 5.29 shows results from simulations of Type B switches capable of two-way combining in each queue (dashed line), hybrid switches capable of two-and-a-half way combining in each of the two queues at an output port (dotted line), Type A switches capable of full five-way combining (solid line), and Type A switches able to perform combining per input (dash-dot line). Latency values for switches without combining are not shown because they are off the chart, over a hundred cycles for 256 PEs and over 1000 cycles for 1024, even at the lowest offered load of 10 percent.

For 1024 PE systems, full five-way combining and combining per input Type A 4×4 switches do show improved performance compared to the 2×2 switches simulated in section 4.4.2. At a 5 percent hot spot rate, the network with 2×2 switches has a maximum bandwidth of about 65 percent and latency of 45 cycles, compared to a maximum bandwidth of 75 to 80 percent for the "high-end" 4×4 switches.

The spread between the combining switches with the most and the least combining capability is much greater for 4×4 than for 2×2 switches. Two-way combining gives comparatively very poor performance, whether in Type A or Type B switches. The hybrid Type B architecture of Figure 5.27 shows much better performance, but is still significantly worse than that of the five-way and combining per input Type A

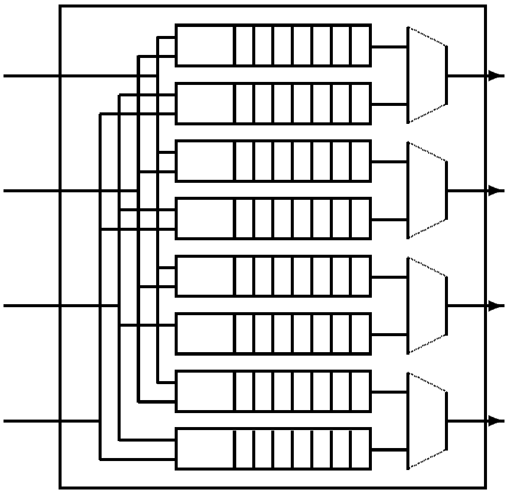
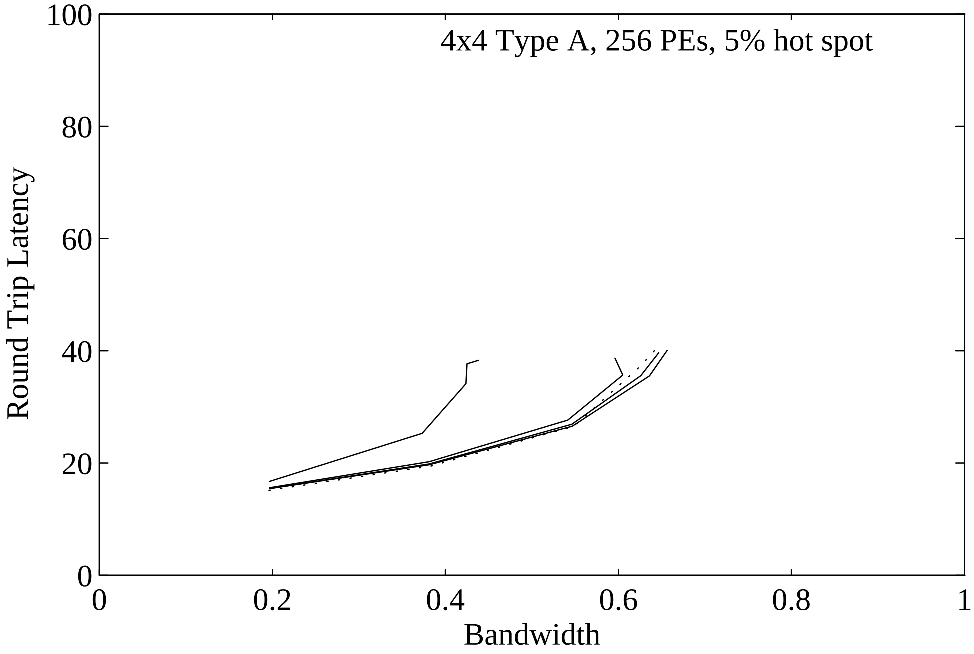


Figure 5.27: Hybrid Type B 4 × 4 combining switch using two-and-a-half-way combining queues.

switches. We have not yet simulated the variant architecture of Figure 5.26. If the crossbars in that architecture can be made fast enough to approximate the performance of a Type A architecture, its performance should approximate that of the Type A switch with three-way combining, making it a cost-effective choice for reasonable performance.



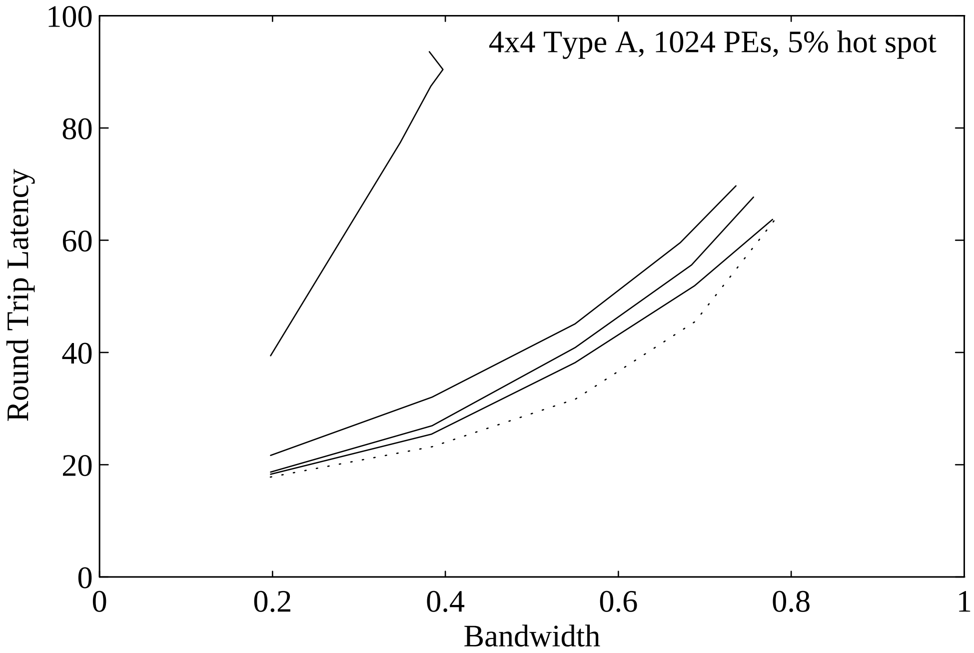


Figure 5.28: Type A 4×4 switches, latency as a function of bandwidth.

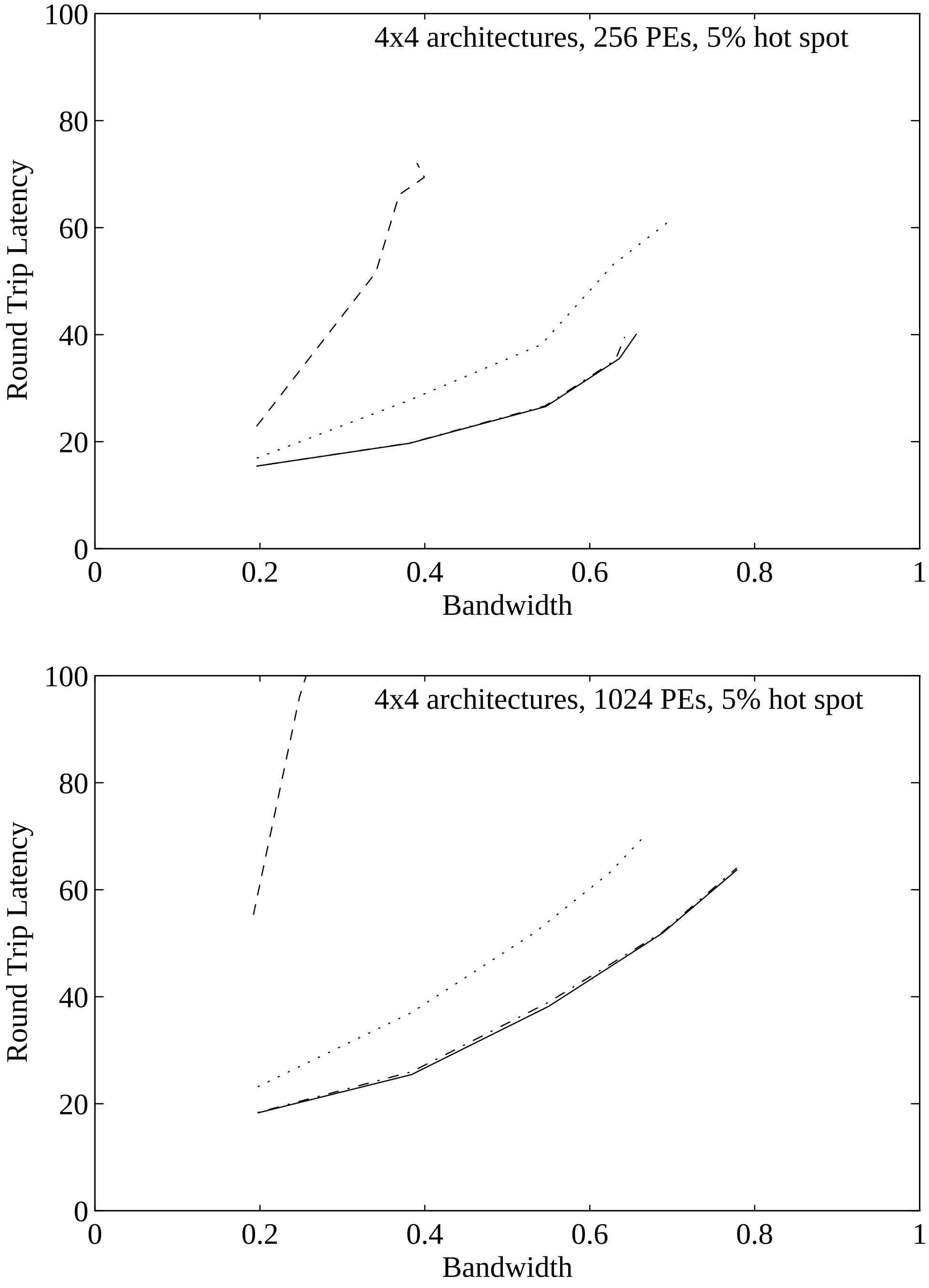


Figure 5.29: Combining options, 4×4 switches, latency as a function of bandwidth.

Chapter 6

Conclusions and Further Work

We have implemented a switch design using systolic queues that includes message combining. This switch has been designed to handle all traffic to memory in a processor-to-memory interconnection network. A 4×4 combining switch board has been in use in a 4 PE prototype since November of 1992, and functions correctly at all speeds at which the memory and processors work reliably (up to 15MHz). At this writing, we are completing the construction of a 16-PE machine using these components and will undertake a 2-year effort to measure the characteristics of the resulting system.

Our implementation experience indicates that the cost of combining is modest, certainly much lower than the estimate of between 6 and 32 times greater cost given for memory-based switch designs in [119]. Consider the three major cost factors of VLSI design area, packaging and cycle time:

- The cost of the combining logic in area and transistors was between 35 and 45 percent of the total chip cost for the FPC and between 50 and 60 for the RPC, in a switch design that, except for the inclusion of combining, was very simple. As part of a design including fault-tolerance features, for example, the combining logic would be proportionately much less.
- Combining increases the packaging cost of a node, because of the need for extra pins connecting the FPC to the RPC, and because of the increased difficulty of bit-slicing a design that does combining compared to a design that does not. In a packaging technology allowing the fabrication of an entire switch in a single package, this cost of combining goes to zero. In any case, extra stage-to-stage wires are not required.
- Most important, the cycle time of the switches need not be increased to include combining. In the CMOS technology we used for our implementation, the delay for the ALU operations required to combine and decombine messages is of roughly the same magnitude as the delay required to arbitrate and route messages, while the delay for the associative matching was not significant. By including the comparators as part of the queues and pipelining the ALU operations within the first slot of the queue, the logic for combining was done in parallel rather than in series with arbitration and routing, causing no increase in cycle time over that for a non-combining buffered switch. If changing to a different technology causes the ALU operations to have greater relative delay compared to arbitration and routing, the ALU operations can be further pipelined.

The inclusion of combining capability in the switches has little impact on the cost of scaling the network to larger sizes. Pin counts per component grow as $\log N$ in both combining and non-combining buffered networks. Cycle time is not affected by the number of stages in the network. Wait buffer storage in bits does grow as $\log^2 N$, but the chip area required for this storage is not likely to be the critical resource on these pin-limited components.

The combining queue design we implemented is an instance of a family of systolic designs with varying amounts of combining capability. Such designs can be included in a variety of switch architectures. We have investigated a number of such architectures, using both analytical techniques and simulation. Conclusions about the best switch for a parallel network are heavily dependent on characteristics of the overall architecture, such as the system size and the latency tolerance of the processor, as well as on the implementation technology. Our results should be useful to computer architects making implementation decisions in the context of a specific system design.

Future research building on the results in this dissertation could be continued in many different directions. We list some of the more important of these below.

Transfering the design from custom CMOS to other technologies.

A standard cell implementation should be carried out to increase the ease of implementation of architectural variations. A data path compiler could be designed to produce systolic queue designs with varying number of bits per item and number of items per queue. A very high performance implementation, using circuit techniques similar to those used for ECL SRAMs [20], is also of interest.

Implementing more complicated systolic queue designs.

Our simulations show that two-and-a-half-way combining queues used in Type A switches give better performance than the Type B switches. Implementing this option should be done to verify that including arbitration in each slice of the queue can be done without unacceptable costs in either area or critical path delay. The cost of pipelining the match and chute transfer logic over multiple packets, necessary if packets smaller than the size of the memory address are to be used, should also be determined by implementation.

Refining and expanding the analysis of combining queues.

The techniques developed in Chapter 5 can easily be extended to three-way combining (by assuming that a hot item turns green when one item has combined with it, and red when two have combined), and to finite queues (by including separate transition tables for the case when the queue is blocked or unblocked at the output). The finite queue analysis will only be interesting if used with iterative techniques to estimate performance in the presence of blocking, and must be checked by simulation.

Extension of design and analysis to other network topologies.

Combining can be used in any network where the message follows the same path on the return from memory, but some of the design considerations may be different. In meshes using simple dimension-order routing, inputs will have a preferred output. In fat trees, a number of different alternatives for combining on the way up or down the tree are possible. Analysis and simulation of behavior with different combining schemes needs to be done. Carrying out detailed designs for combining switches for such network architectures would illuminate the cost tradeoffs involved.

Simulation of more realistic patterns and program segments.

Most of the simulations of the effectiveness of combining, by ourselves and others, have been carried out using a single hot spot model. Using molasses, we plan to investigate multiple hot spot and partitioned models more thoroughly, and eventually to simulate short code segments.

Observation of combining in a system environment.

While the 16 PE Ultra III prototype is not large enough to show significant degradation from hot spots on real applications, memory traffic behavior from applications can be measured and synthetic applications created that stress the network in a similar way. The Ultra III prototype includes a XILINX coprocessor in the switch nodes to gather and measure information about switch behavior (see [17]).

In conclusion

We hope that our work will help to convince system designers to regard hardware combining within the interconnection network as a reasonable support for software efforts, like caches or TLBs, rather than as an option too expensive to consider.

Bibliography

[1] Ferri Abolhassan, Jorg Keller, and Wolfgang J. Paul. On the cost-effectiveness and realization of the theoretical PRAM model. Technical report, Universitat des Saarlandes, 1-6600 Saarbrucken, Germany, September 1991.

[2] Seth Abraham and Krishnan Padmanabhan. Performance of the direct binary n-cube network for multiprocessors. *IEEE Transactions on Computers*, 38(7):1000–1011, 1989.

[3] Seth Abraham and Krishnan Padmanabhan. Performance of multicomputer networks under pin-out constraints. *Journal of Parallel and Distributed Computing*, 12:237–248, 1991.

[4] Vikram S. Adve and Mary K. Vernon. Performance analysis of mesh interconnection networks with deterministic routing. Technical Report 1001b, Computer Science Department, University of Wisconsin-Madison, July 1993.

[5] Anant Agarwal et al. The MIT Alewife machine: A large-scale distributed memory multiprocessor. In M. Dubois and S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1992.

[6] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. *International Conference on Supercomputing: ACM SIGARCH Computer Architecture News*, 18(3), September 1990.

[7] Hideharu Amano and Gaye Kalidou. A Batcher double Omega network with combining. *International Conference on Parallel Processing*, I:718–719, August 1991.

[8] John B. Andrews, Carl J. Beckmann, and David K. Poulsen. Notification and multicast networks for synchronization and coherence. *Journal of Parallel and Distributed Computing*, 15, August 1992.

[9] Bruce W. Arden and Hikyu Lee. A regular network for multicomputer systems. *IEEE Transactions on Computers*, C-31(1), January 1982.

[10] M. Atiquzzaman and M. S. Akhtar. Performance of buffered multistage interconnection networks in non-uniform traffic environment. *Proceedings of the 7th International Parallel Processing Symposium*, pages 762–767, April 1993.

[11] Didier Badouel, Charles A. Wuthrich, and Eugene L. Fiume. Routing strategies and message contention on low-dimensional interconnection networks. Technical Report 258, Computer System Research Institute, University of Toronto, December 1991.

[12] Shobana Balakrishnan and Dhabaleswar K. Panda. Impact of multiple consumption channels on wormhole routed k-ary n-cube networks. *International Parallel Processing Symposium*, pages 163–167, April 1993.

[13] M. Barnett, R. Littlefield, D. G. Payne, and R. van de Geijn. Global combine on mesh architectures with wormhole routing. *Proceedings of the 7th International Parallel Processing Symposium*, pages 156–162, April 1993.

[14] Laxmi N. Bhuyan, Qing Yang, and Dharma P. Agrawal. Performance of multiprocessor interconnection networks. *IEEE Computer*, 22(2):25–37, February 1989.

[15] Richardo Bianchini, Mark E. Crovella, Leonidas Kontothanassis, and Thomas J. LeBlanc. Memory contention in scalable cache-coherent multiprocessors. Technical Report 448, University of Rochester, Rochester, New York, April 1993.

[16] Ronald Bianchini. Packaging Ultracomputers and implementing Ultracomputer prototypes. Ultracomputer Note #177, New York University, May 1992.

[17] Ronald Bianchini, Susan R. Dickey, Jan Edler, Gabriel Goodman, Allan Gottlieb, Richard Kenner, and Jiarui Wang. The Ultra III prototype. *Proceedings of the 7th International Parallel Processing Symposium Parallel Systems Fair*, pages 2–9, April 1993.

[18] Guy Blelloch. Scans as primitive parallel operations. *International Conference on Parallel Processing*, pages 355–362, August 1986.

[19] Kevin Bolding and Smaragda Konstantinidou. On the comparison of hypercube and torus networks. *International Conference on Parallel Processing*, I:62–66, August 1992.

[20] Barbara A. Chappell et al. A 2-ns cycle, 3.8-ns access 512-kb CMOS ECL SRAM with a fully pipelined architecture. *IEEE Journal of Solid-State Circuits*, 26(11), November 1991.

[21] Frederic T. Chong and Thomas F. Knight, Jr. Design and performance of multipath MIN architectures. Transit Note #64, MIT Transit Project, March 1992.

[22] Intel Corporation. Paragon XP/S Supercomputer. Product description, 1992.

[23] W. Crowther et al. Performance measurements on a 128-node Butterfly parallel processors. *International Conference on Parallel Processing*, pages 531–540, August 1985.

[24] W. Crowther, J. Goodhue, R. Gurwitz, R. Rettberg, and R. Thomas. The Butterfly (TM) parallel processor. *IEEE Computer Architecture Technical Committee Newsletter*, pages 18–45, September 1985.

[25] William J. Dally. Network and processor architecture for message-driven computers. In Robert Suaya and Graham Birtwistle, editors, *VLSI and Parallel Computation*, pages 140–222. Morgan Kaufman, 1990.

[26] William J. Dally. Wire efficient VLSI multiprocessor communication networks. In P. Losleben, editor, *Proceedings of the Stanford Conference on Advanced Research in VLSI*, pages 391–415. MIT Press, March 1987.

[27] William J. Dally. Express cubes: Improving the performance of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 40(9):1016–1023, September 1991.

[28] William J. Dally et al. Design and implementation of the Message-Driven Processor. In Thomas Knight and John Savage, editors, *Proceedings of the 1992 Brown/MIT Conference on Advanced Research in VLSI*, pages 5–25, March 1992.

[29] William J. Dally and Charles L. Seitz. The Torus routing chip. *Journal of Distributed Systems*, 1(3):187–196, 1986.

[30] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[31] William J. Dally and P. Song. Design of a self-timed VLSI multicomputer communications controller. *International Conference on Computer Design*, pages 230–234, 1987.

[32] Sivarama P. Dandamudi and Derek L. Eager. Hot-spot contention in binary hypercube networks. *IEEE Transactions on Computers*, February 1992.

[33] F. Darema-Rogers, G. F. Pfister, and K. So. Memory access patterns of parallel scientific programs. *Performance Evaluation Review*, 15(1):46–57, May 1987.

[34] André DeHon. Robust, high-speed network design for large-scale multiprocessing. Master's Thesis, MIT, February 1993.

[35] Daniel M. Dias. *Packet Communication in Delta and Related Networks*. Ph. D. thesis, Rice University, 1981.

[36] Daniel M. Dias and Manoj Kumar. Preventing congestion in multistage networks in the presence of hotspots. *International Conference on Parallel Processing*, August 1989.

[37] Susan R. Dickey and Richard Kenner. A combining switch for the NYU Ultracomputer. Ultracomputer Note #178, New York University, February 1992.

[38] Susan R. Dickey and Richard Kenner. Using qualified clocks in the NORA clocking methodology to implement a systolic queue design. *Proceedings of the Brown/MIT Conference on Advanced Research in VLSI*, pages 165–179, March 1992.

[39] Susan R. Dickey and Yue-sheng Liu. Simulation and analysis of enhanced switch architectures for interconnection networks in massively parallel shared memory machines. *Proceedings of the Second Symposium on the Frontiers of Massively Parallel Computation*, pages 487–490, October 1988.

[40] Susan R. Dickey and Ora E. Percus. Performances differences among combining switch architectures. *International Conference on Parallel Processing*, August 1992.

[41] Jianxun Ding and Laxmi N. Bhuyan. Performance evaluation of multistage interconnection networks with finite buffers. *International Conference on Parallel Processing*, August 1991.

[42] Reinhard Drefenstedt and Dietmar Schmidt. On the physical design of butterfly networks for prams. *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 202–209, 1992.

[43] Jan Edler et al. Issues related to MIMD shared-memory computers: the NYU Ultracomputer approach. *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 126–135, June 1985.

[44] Matthew Farrens, Brad Wetmore, and Allison Woodruff. Alleviation of tree saturation in multistage interconnection networks. *Supercomputing '91*, pages 400–409, November 1991.

[45] Tse-yun Feng. A survey of interconnection networks. *IEEE Computer*, 14(12):12–27, December 1981.

[46] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948–960, 1972.

[47] S. Fortune and J. Wyllie. Parallelism in random access machines. *Tenth ACM Symposium on the Theory of Computing*, pages 114–118, 1978.

[48] Patrick T. Gaughan and Sudhakar Yalamanchili. Adaptive routing protocols for hypercube interconnection networks. *IEEE Computer*, pages 12–24, May 1993.

[49] R. R. Glenn and D. V. Pryor. Instrumentation for a massively parallel MIMD application. *Journal of Parallel and Distributed Computing*, 12(3):223–236, July 1991.

[50] R. R. Glenn, D. V. Pryor, J. M. Conroy, and T. Johnson. Characterizing memory hot spots in a shared-memory MIMD machine. *Proceedings of Supercomputing '91*, pages 554–566, November 1991.

[51] L. Rodney Goke and G. J. Lipovski. Banyan networks for partitioning multiprocessor systems. *Proceedings of the 1st Annual Symposium on Computer Architecture*, pages 21–28, 1973.

[52] Nelson F. Goncalves and Hugo J. DeMan. NORA: A racefree dynamic CMOS technique for pipelined logic structures. *IEEE Journal of Solid-State Circuits*, SC-18(3):261–266, June 1983.

[53] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessing. In *Proceedings of the ASPLOS III*, pages 64–73, April 1989.

[54] Allan Gottlieb. An historical guide to the Ultracomputer literature. Ultracomputer Note #36, New York University, October 1981.

[55] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer–designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2), February 1983.

[56] Allan Gottlieb and Clyde P. Kruskal. Complexity results for permuting data and other computations on parallel processors. *Journal of the ACM*, 31(2):193–209, April1984.

[57] Allan Gottlieb and Clyde P. Kruskal. Coordinating parallel processors: A partial unification. *Computer Architecture News*, pages 16–24, October 1981.

[58] Allan Gottlieb, Boris Lubachevsky, and Larry Rudolph. Coordinating large numbers of processors. *International Conference on Parallel Processing*, 1981.

[59] Allan Gottlieb, Larry Rudolph, and Boris Lubachevsky. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM TOPLAS*, 5(2):164–189, 1983.

[60] Leo J. Guibas and Frank M. Liang. Systolic stacks, queues and counters. *MIT Conference on Advanced Research in VLSI*, pages 155–164, 1982.

[61] Wing S. Ho and Derek L. Eager. A novel strategy for controlling hot spot congestion. *International Conference on Parallel Processing*, I:14–18, August 1989.

[62] William Tsun-Yuk Hsu and Pen-Chung Yew. An effective synchronization network for hot-spot accesses. *ACM Transactions on Computer Systems*, 10(3):167–189, August 1992.

[63] Yarsun Hsu, C. Benveniste, J. Ruedinger, and C. J. Tan. Design of VLSI switch for highly parallel multiprocessor system. *IEEE Custom Integrated Circuits Conference*, pages 24.4.1–24.4.4, May 1990.

[64] Inseok S. Hwang and Aaron L. Fisher. Ultrafast compact 32-bit CMOS adders in multiple output Domino logic. *IEEE Journal of Solid-State Circuits*, 24(2):358–369, April 1989.

[65] P. G. Jansen and J. L. W. Kessels. The DIMOND: A component for the modular construction of switching networks. *IEEE Transactions on Computers*, C-29(10), October 1980.

[66] Yih-chyun Jenq. Performance analysis of a packet switch based on single-buffered Banyan network. *IEEE Journal on Selected Areas in Communications*, SAC-1(6):1014–1021, December 1983.

[67] Byung-Chang Kang, Gyung Ho Lee, and Richard Kain. Performance of multistage combining networks. *International Conference on Parallel Processing*, pages 550–553, August 1991.

[68] Byung-Chang Kang, Gyung Ho Lee, and Richard Kain. A performance bound analysis of multistage combining networks using a probabilistic model. In *Conference Proceedings: 1991 International Conference on Supercomputing*, pages 448–457, Cologne, Germany, June 1991.

[69] P. Kermani and Leonard Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3:267–286, 1979.

[70] Hyong Sok Kim and Alberto Leon-Garcia. Performance of buffered banyan networks under nonuniform traffic patterns. *IEEE Transactions on Computers*, pages 648–658, May 1990.

[71] David Klappholz. An improved design for a stochastically conflict-free memory/interconnection system. *Fourteenth Asilomar Conference On Circuits, Systems and Computers*, pages 443–448, November 1980.

[72] David Klappholz. Stochastically conflict-free data-base memory systems. *International Conference on Parallel Processing*, pages 283–289, August 1980.

[73] Leonard Kleinrock. *Queueing Systems, Volume 2: Computer Applications.* John Wiley & Sons, Inc., 1976.

[74] R. H. Krambeck, C. M. Lee, and H. S. Law. High-speed compact circuits with CMOS. *IEEE Journal of Solid-State Circuits*, SC-17(3):614–619, June 1982.

[75] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization on multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems*, 10(4):579–601, October 1988.

[76] Clyde P. Kruskal and Marc Snir. A unified theory of interconnection network structure. *Theoretical Computer Science*, 48(1):75–94, 1986.

[77] Clyde P. Kruskal and Marc Snir. The performance of multistage interconnection networks for multiprocessors. *IEEE Transactions on Computers*, 32(12):1091–1098, December 1983.

[78] Clyde P. Kruskal, Marc Snir, and Allan Weiss. On the distribution of delays in buffered multistage interconnection networks for uniform and nonuniform traffic. *International Conference on Parallel Processing*, pages 215–219, 1984.

[79] Clyde P. Kruskal, Marc Snir, and Allan Weiss. The distribution of waiting times in clocked multistage interconnection networks. *IEEE Transactions on Computers*, November 1988.

[80] Manoj Kumar and J. R. Jump. Performance enhancement in buffered Delta networks using crossbar switches and multiple links. *Journal of Parallel and Distributed Computing*, 1:81–103, 1984.

[81] Manoj Kumar and Gregory F. Pfister. The onset of hotspot contention. *International Conference on Parallel Processing*, August 1986.

[82] Lizyamma Kurian and Matthew J. Thazhuthaveetil. Effect of hot spots on multiprocessor systems using circuit switched interconnection networks. *International Conference on Parallel Processing*, August 1991.

[83] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[84] Tomas Lang and Lance Kurisaki. Nonuniform traffic spots (NUTS) in multistage interconnection networks. *Journal of Parallel and Distributed Computing*, 10:55–67, 1990.

[85] Duncan H. Lawrie. Access and alignment of data in an array processor. *IEEE Transactions on Computers*, C-24:1145–1155, December 1975.

[86] Gyung Ho Lee. A performance bound of multistage combining networks. *IEEE Transactions on Computers*, 38(10):1387–1395, October 1989.

[87] Gyung Ho Lee, Clyde P. Kruskal, and David J. Kuck. The effectiveness of combining in shared memory parallel computers in the presence of 'hot spots'. *International Conference on Parallel Processing*, pages 35–41, August 1986.

[88] Gyungho Lee. Another combining scheme to reduce hot spot contention in large-scale shared memory parallel computers. In *Lecture Notes in Computer Science, Vol 297: Supercomputing*, pages 68–79. Springer-Verlag, 1988.

[89] Yann-Hang Lee, Sandra E. Cheung, and Jih-Kwon Peir. Consecutive requests traffic model in multistage interconnection networks. *International Conference on Parallel Processing*, I:534–541, August 1991.

[90] F. Thomson Leighton. *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes.* Morgan Kaufman Publishers, Inc., 1992.

[91] Charles E. Leiserson. *Area Efficient VLSI Computation.* MIT Press, Cambridge, Massachusetts, 1983.

[92] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[93] Charles E. Leiserson et al. The network architecture of the connection machine CM-5. *ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, 1992.

[94] Daniel Lenoski et al. The Stanford Dash multiprocessor. *IEEE Computer*, pages 63–79, March 1992.

[95] R. L. Lesher and M. J. Thazhuthaveetil. Hotspot contention in non-blocking multistage interconnection networks. *International Conference on Parallel Processing*, pages 401–404, 1990.

[96] T. Lin and L. Kleinrock. Performance analysis of finite-buffered multistage interconnection networks with a general traffic pattern. *Proceedings of the 1991 ACM Sigmetrics Conference*, May 1991.

[97] T. Lin and A. Tantawi. Performance evaluation of packet-switched multistage interconnection networks under a model of hot-spot traffic. *ORSA/TIMS Joint National Meeting*, October 1990.

[98] G. J. Lipovski and Paul Vaughan. A fetch-and-op implementation for parallel computers. *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 352–372, May 1988.

[99] Yue-sheng Liu. Delta network performance and "hot spot" traffic. Ultracomputer Note #132, New York University,, January 1988.

[100] Yue-sheng Liu. *Architecture and Performance of Processor-Memory Interconnection Networks for MIMD Shared Memory Parallel Processing Systems.* Ph. D. Dissertation, New York University, Department of Computer Science, September 1990.

[101] Yue-sheng Liu, Susan R. Dickey, and Allan Gottlieb. Interconnection network switch architectures and combining strategies. Ultracomputer Note #186, New York University,, January 1988.

[102] Carver A. Mead and Lynn A. Conway. *Introduction to VLSI Systems.* Addison-Wesley, Reading, Massachusetts, 1980.

[103] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[104] Arif Merchant. Analytical models of combining banyan networks. *Performance Evaluation Review*, 20(1):205–211, June 1992.

[105] Henry Minsky. A parallel crossbar routing chip for a shared memory multiprocessor. Master's Thesis, MIT, January 1991.

[106] Debasis Mitra. Randomized parallel communications on an extension of the Omega network. *Journal of the ACM*, 34(4):802–824, October 1987.

[107] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, pages 62–76, January 1993.

[108] M. O. Noakes and William J. Dally. System design of the J-machine. In *Proceedings of the 6th MIT Conference on Advanced Research in VLSI*, pages 179–194. MIT Press, 1990.

[109] John K. Ousterhout. Switch-level delay models for digital MOS VLSI. *Proceedings of the 21st Design Automation Conference*, pages 542–548, June 1984.

[110] Janak H. Patel. Processor-memory interconnections for multiprocessors. In *Proceedings of the Sixth Annual Symposium on Computer Architecture*, pages 168–177. ACM SIGARCH, April 1979.

[111] Janak H. Patel. Performance of processor-memory interconnections for multiprocessors. *IEEE Transactions on Computers*, C-30(10):771–780, October 1981.

[112] N. M. Patel and P. G. Harrison. On hot-spot contention in interconnection networks. *Performance Evaluation Review*, 16(1):114–123, May 1988.

[113] M. C. Pease. The indirect binary n-cube microprocessor array. *IEEE Transactions on Computers*, C-26(5):548–573, May 1977.

[114] Ora E. Percus and Susan R. Dickey. Performance analysis of clock-regulated queues with output multiplexing in 2 by 2 crossbar switch architectures. *Proceedings of the Twenty-First Annual Pittsburgh Conference on Modeling and Simulation*, (3):1225–1233, May 1990.

[115] Ora E. Percus and Susan R. Dickey. Performance analysis of clock-regulated queues with output multiplexing in three different 2 by 2 crossbar switch architectures. *Journal of Parallel and Distributed Computing*, 16(1):27–40, September 1992.

[116] Ora E. Percus and J. K. Percus. Elementary properties of clock-regulated queues. *SIAM Journal on Applied Mathematics*, 50:1166–1175, 1990.

[117] Ora E. Percus and J. K. Percus. Times series transformations in clocked queueing networks. *Communications on Pure and Applied Math*, 44:1107–1119, 1991.

[118] Gregory F. Pfister et al. The IBM Research Parallel Processor Prototype (RP3): Introduction and architecture. *International Conference on Parallel Processing*, pages 764–771, 1985.

[119] Gregory F. Pfister and Allan Norton. 'Hot spot' contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, 1985.

[120] Eugene Pinsky and Paul Stirpe. Modeling and analysis of 'hot spots' in an asynchronous $n \times n$ crossbar switch. *International Conference on Parallel Processing*, I:546–549, August 1991.

[121] Abhiram G. Ranade. The Fluent abstract machine. In Jonathan Allen and F. Thomson Leighton, editors, *Proceedings of the Fifth MIT Conference on Advanced Research in VLSI*, pages 71–93, 1988.

[122] Jacob T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2:484–521, 1980.

[123] Steven L. Scott and Gurindar S. Sohi. The use of feedback in multiprocessors and its application to tree saturation control. *IEEE Transactions on Parallel and Distributed Computing*, 1(4):385–398, 1990.

[124] Charles L. Seitz. Concurrent VLSI architectures. *IEEE Transactions on Computers*, C-33(12):1247–1265, 1984.

[125] Charles L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[126] Dennis Shasha and Marc Snir. Efficient and correct execution of programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[127] Masakazu Shoji. *CMOS Digital Circuit Technology*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[128] Rajgopalan Sivaram. Queuing delays for uniform and non-uniform traffic patterns in a MIN. *Simulation Digest*, 35:17–27, Summer 1992.

[129] Marc Snir and Jon Solworth. Switch digest. Ultracomputer Hardware Note #27, New York University, 1984.

[130] Marc Snir and Jon Solworth. The Ultraswitch – a VLSI network node for parallel processing. Ultracomputer Note #39, New York University, January 1984.

[131] Herbert Sullivan and T. R. Bashkow. A large scale, homogenous, fully distributed parallel machine, I. *Proceedings of the 4th Symposum on Computer Architecture*, pages 105–117, March 1977.

[132] Yuval Tamir and Gregory L. Frazier. High-performance multi-queue buffers for VLSI communication switches. *Journal of Parallel and Distributed Computing*, 14:402–416, 1992.

[133] Yuval Tamir and Gregory L. Frazier. High-performance multi-queue buffers for VLSI communication switches. *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 343–354, May 1988.

[134] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, Inc., 1981.

[135] Thomas H. Theimer, Erwin P. Rathgeb, and Manfred N. Huber. Performance analysis of buffered banyan networks. *IEEE Transactions on Communications*, 39(2):269–277, February 1991.

[136] Robert H. Thomas. Behavior of the butterfly parallel processor in the presence of memory hot spots. *International Conference on Parallel Processing*, pages 46–50, August 1986.

[137] Nian-Feng Tzeng. Design of a novel combining structure of shared-memory multiprocessors. *International Conference on Parallel Processing*, I:1–8, August 1989.

[138] Nian-Feng Tzeng. An approach to the performance improvement of multistage interconnection networks with nonuniform traffic spots. *International Conference on Parallel Processing*, I:542–545, August 1991.

[139] Nian-Feng Tzeng. Alleviating the impact of tree saturation on multistage interconnection network performance. *Journal of Parallel and Distributed Computing*, pages 107–117, June 1991.

[140] Nian-Feng Tzeng. A cost-effective combining structure for large-scale shared-memory multiprocessors. *IEEE Transactions on Computers*, 41(11):1420–1429, November 1992.

[141] Shiwei Wang, Yarsun Hsu, and C. J. Tan. A novel message switch for highly parallel systems. *International Conference on Computer Design*, pages 150–155, 1989.

[142] Gregory V. Wilson. Implementing distributed queues on multicomputers without polling or contention. Technical Report 9125, University of Edinburgh, November 1991.

[143] Gregory V. Wilson. Using opportunistic combining networks to reduce contention in multicomputers. Technical Report 9127, University of Edinburgh, November 1991.

[144] Monica C. Wong. A combining Omega network: Performance vs. implementation. Technical Report RC 11977, IBM Research Report, June 1986.

[145] Chuan-lin Wu and Tse-yun Feng. *IEEE Tutorial: Interconnection networks for Parallel and Distributed Processing.* IEEE Computer Society Press, 1984.

[146] Chuan-lin Wu and Tse-yun Feng. On a class of multistage interconnection networks. *IEEE Transactions on Computers*, C-29(8):694–702, August 1980.

[147] David W. L. Yen, Janak H. Patel, and Edward S. Davidson. Memory interference in synchronous multiprocessor systems. *IEEE Transactions on Computers*, C-31(11), November 1982.

[148] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36:486–493, April 1987.

[149] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *International Conference on Parallel Processing*, pages 51–58, August 1986.

[150] Hyunsoo Yoon, Kyungsook Y. Lee, and Ming T. Liu. Performance analysis of multibuffered packet-switching networks in multiprocessor systems. *IEEE Transactions on Computers*, March 1990.

Index