Computer Systems Laboratory

Department of Electrical Engineering
Stanford University, CA 94305

Efficient Multiprocessor Communications: Networks, Algorithms, Simulation, and Implementation

Yen-Wen Lu

July 1996

Technical Report No. CSL-TR-96-699
(Technical Report No. STAR-1DEH694-1996-1)

Efficient Multiprocessor Communications: Networks, Algorithms, Simulation, and Implementation

By

Yen-Wen Lu

July 1996

Technical Report No. CSL-TR-96-699

(Technical Report No. STAR-1DEH694-1996-1)

Department of Electrical Engineering

Stanford University, CA 94305

This report constitutes a Ph.D. dissertation submitted to Stanford University.

Abstract

As technology and processing power continue to improve, inter-processor communication becomes a performance bottleneck in a multiprocessor network. In this dissertation, an enhanced 2-D torus with segmented reconfigurable bus (SRB) to overcome the delay due to long distance communications was proposed and analyzed. A procedure of selecting an optimal segment length and segment alignment based on minimizing the lifetime of a packet and reducing the interaction between segments was developed to design a SRB network. Simulation shows that a torus with SRB is more than twice as efficient as a traditional torus.

Efficient use of channel bandwidth is an important issue in improving network performance. The communication links between two adjacent nodes can be organized as a pair of opposite uni-directional channels, or combined into a single bi-directional channel. A modified channel arbitration scheme with hidden delay, called "token-exchange," was designed for the bi-directional channel configuration. In spite of the overhead of channel arbitration, simulation shows that bi-directional channels have significantly better latency-throughput performance and can sustain higher data bandwidth relative to uni-directional channels of the same channel width. For example, under 2% hot-spot traffic, bi-directional channels can support 80% more bandwidth without saturation compared with uni-directional channels.

An efficient, low power, wormhole data router chip for 2-D mesh and torus networks with bi-directional channels and token-exchange arbitration was designed and implemented. The token-exchange delay is fully hidden and no latency penalty occurs when there is no traffic contention; the token-exchange delay is also negligible when the contention is high. Distributed decoders and arbiters are provided for each of four IO ports, and a fully-connected 5×6 crossbar switch increases parallelism of data routing. The router also provides special hardware such as flexible header decoding and switching to support path-based multicasting. From measured results, multicasting with two destinations used only 1/3 of the energy required for unicasting. The wormhole router was fabricated using MOSIS/HP 0.6μm technology. It delivers 1.6Gb/s (50MHz) @ Vdd=2.1V, consuming an average power of 15mW.

Acknowledgments

I would like to thank my parents, to whom this dissertation is dedicated, for their unconditional support throughout my education. Without their encouragement, it would not be possible for me to achieve my goals.

I would like to express my deepest gratitude to my advisors, Professor Allen Peterson and Professor Len Tyler. Professor Peterson had given me a tremendous amount of support and guidance during my first three years at Stanford. Professor Tyler has been very patient with me for preparing my oral defense and this dissertation. I especially thank him for the time and effort he put into helping me clarify and correct many important points in this dissertation. I would like to thank Professor Michael Godfrey for letting us use his laboratory to do the chip testing and for helping us resolve all the related problems. I would also like to thank Professor Teresa Meng, Professor John Gill, and Professor Fabian Pease for serving in my reading and oral committees.

I have been fortunate to work with my colleagues in the Space, Telecommunications, and Radioscience Laboratory, and the Ultra Low Power group. Jim Burr has helped me a great deal in the direction of my research and projects. Gerard Yeh, my partner of the STARP project, is always eager to help me solve problems, and it is a pleasant experience to work with him. I would like to thank Bevan Baas for many discussions of all different topics. I would also like to thank the visiting scholars, Masataka Matsui and Kallol Bagchi, for broadening my background and knowledge.

Last but not least, I would like to thank my wife, Woan-Yu, for her loving support and understanding. She has accompanied me for countless hours to finish my projects over the last few years.

My research was supported by Intel, NASA, Toshiba, and the department of Electrical Engineering, Stanford University. They are all gratefully acknowledged.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The demand for more powerful computation is increasing rapidly in many different fields. For example, high-definition television (HDTV) with 2K $\times$ 1K pixels per frame and a frame rate of 60 frames per second will require at least 100 GOPS (1×10^{11} operations/sec) for video compression. With higher quality and larger image size, the computation requirement falls easily in the Tera OPS (1×10^{12} operations/sec) range. Facing this exploding demand, we will ask where the computation power comes from.

Different advanced techniques and computing models have been proposed and implemented to improve computational capability (Hennessy and Patterson, 1996). These computing models have different approaches and efficiency, but they all achieve their goals by increasing the concurrency of the system to some degree.

Pipelining

A serial computation can be divided into a number of steps, called stages. Each stage works at full speed on a particular part of computation. The output of one stage is the input of the next stage. With proper balance of the delay of each stage, data are fed continuously into the pipelined stages and all stages can operate concurrently with different sets of input to maximize the computation throughput.

Superscalar

When a pipeline reaches its maximum capacity, more functional units must be added in parallel to increase performance. Machines that can issue multiple independent instructions per cycle are superscalar machines. Different instructions can use different functional blocks and execute in separate pipelines concurrently if there is no hardware conflict. Therefore, a superscalar machine increases the parallelism in a processor to achieve more computation in a shorter time.

Massively Parallel Processing (MPP)

After we consume the computation power of a single processor, the next step is to combine more processors to solve problems together. Massively parallel processing represents the ultimate approach to achieving the limits of computation.

Parallel processing provides significant computational advantages for many scientific, signal processing, and image processing applications. However, it took more than 20 years for parallel computers to move from laboratory to marketplace. Even with today's most advanced technology, many challenges remain. Computation is not the only concern, however, as other issues are becoming bottlenecks. For example, as a parallel system grows, inter-processor communication may eventually dominate overall system performance. Having noticed the increasing importance of inter-processor communication, realizing its potential to limit the improvement of a parallel system, we take the question of optimal multiprocessor communication as the central problem of this dissertation.

1.1 Multiprocessor Speedup

When we have a multiprocessor system, a natural question to ask is how much performance gain we can get from the system. Speedup performance models were defined to answer this question quantitatively.

1.1.1 Speedup with a fixed load

The most common speedup model is that of the "fixed-load", which has a fixed problem size and workload. As the number of processors increases, the fixed load is distributed to all the processors in the system. Let $T(1)$ be the time required to finish the problem in one processor, and $T(n)$ be the time for n processors. Then the speedup factor for the fixed load is defined as

$$S_n = \frac{T(1)}{T(n)} \tag{1.1}$$

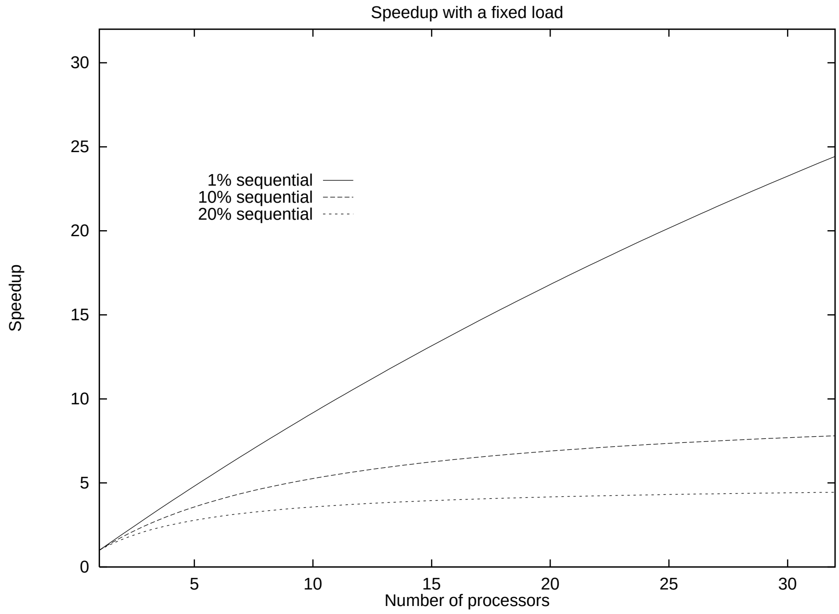


Figure 1.1: Amdahl's Law. Speedup with a fixed load

Amdahl's law is used to find the speedup factor for a fixed load. Usually the total workload W is assumed to consist of two parts: W_s is the sequential portion of the program which cannot be parallelized, and W_p is the portion which can be parallelized and evenly distributed in all available processors. The speedup factor S_n can be written as

$$S_n = \frac{W}{W_s + W_p/n} = \frac{W_s + W_p}{W_s + W_p/n} \tag{1.2}$$

When $n \to \infty$, $S_n \to W/W_s$. So the speedup is bounded by W/W_s and cannot go further even we have more processors. Fig. 1.1 shows the speedup versus the number of processors for a fixed load. When we have 10% of a program which cannot be parallelized, the speedup curve is very flat after we have more than about 20 processors. So the sequential part becomes the performance bottleneck.

1.1.2 Speedup with fixed execution time

The problem with Amdahl's law is that the workload cannot scale with the number of processors to fully utilize the available computing power, and we get a very low speedup factor. Gustafason (1988) proposed a fixed-time speedup model to scale the problem size.

Assuming the total work $W(1) = W_s + W_p$ for one processor can be done in time T, for n

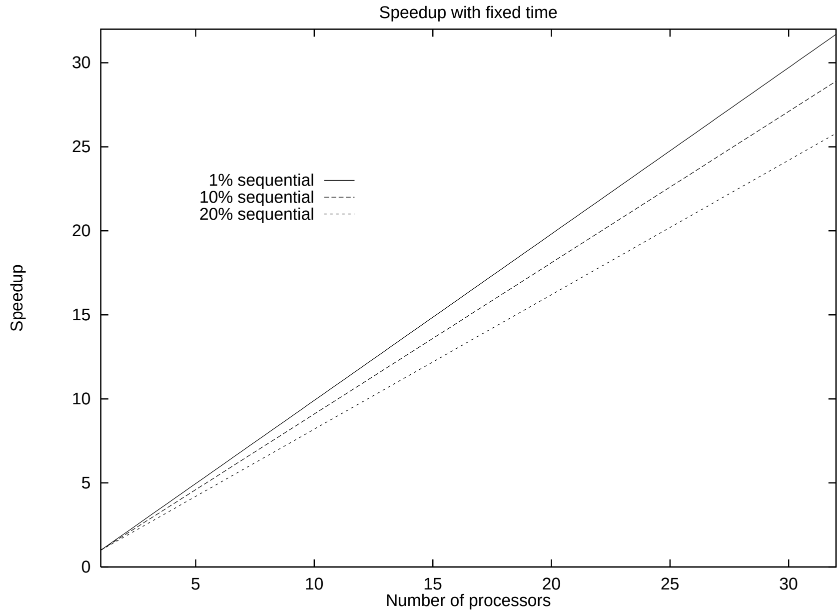


Figure 1.2: Gustafson's Law. Speedup with fixed time

processors, we can finish the total workload $W(n) = W_s + nW_p$ in the same amount of time. Therefore, the speedup S_n can be defined as

$$\begin{aligned} S_n &= \frac{W(n)}{W(1)} \\ &= \frac{W_s + nW_p}{W_s + W_p} \end{aligned} \tag{1.3}$$

Fig. 1.2 shows the speedup for the fixed-time model. With fixed-time characteristics, a very good speedup factor is achieved even when the sequential portion is as high as 20%. Because we have scaled the problem size, we can utilize more computing power by keeping all processors busy.

Gustafson's law gives us insight to design our own parallel machine for image/video processing (Chapter 7). When we have a fixed image size, the benefit of increasing the network size to solve the problem is small when we have a "sufficient" network size.[1] But if we are dealing with a larger image size, we should scale our network size to match the workload, and thereby maintain a very high speedup factor.

[1] "Sufficient" means that the time a processor spends in the parallel portion is about the same as the time in the sequential part.

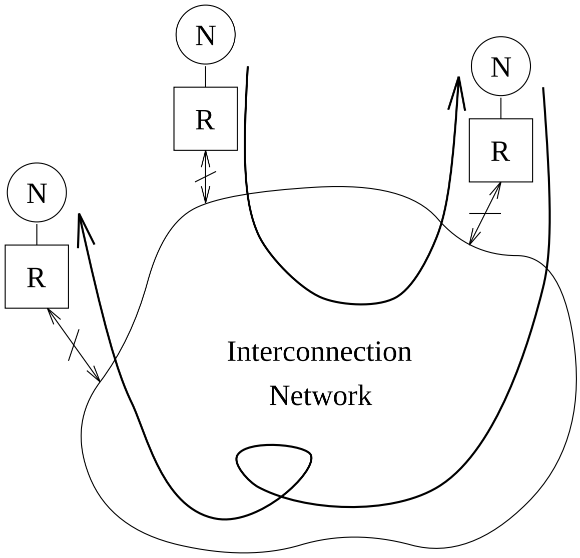


Figure 1.3: Multiprocessor networks. Communication occurs between nodes (processors and/or memory). N is the node and R is the data router.

1.2 Multiprocessor Networks

A multiprocessor network is used to connect all processing elements, memory modules, and periphery devices, and to make them work together. Fig. 1.3 illustrates a general multiprocessor network. The interconnection network is the backbone of a parallel system. All nodes collaborate with each other through the interconnection network. Chapter 2 will give an overview of interconnection networks and different network examples.

1.3 Multiprocessor Communications

Fig. 1.4 shows the total execution time of a parallel machine in terms of its two components: computation and communication. We assume that the computation can be parallelized completely, and that the computation time is inversely proportional to the number of processors. Communication time, however, is linear with the number of processors. As the number of processors increases, the computation time decreases dramatically. However, the communication time increases steadily and becomes dominant finally. Therefore, the communication overhead plays an important role in the overall system performance.

In fact, when traffic contention in a network occurs, the communication time will increase more than linearly with the number of processors. Fig. 1.5 illustrates a simple example of

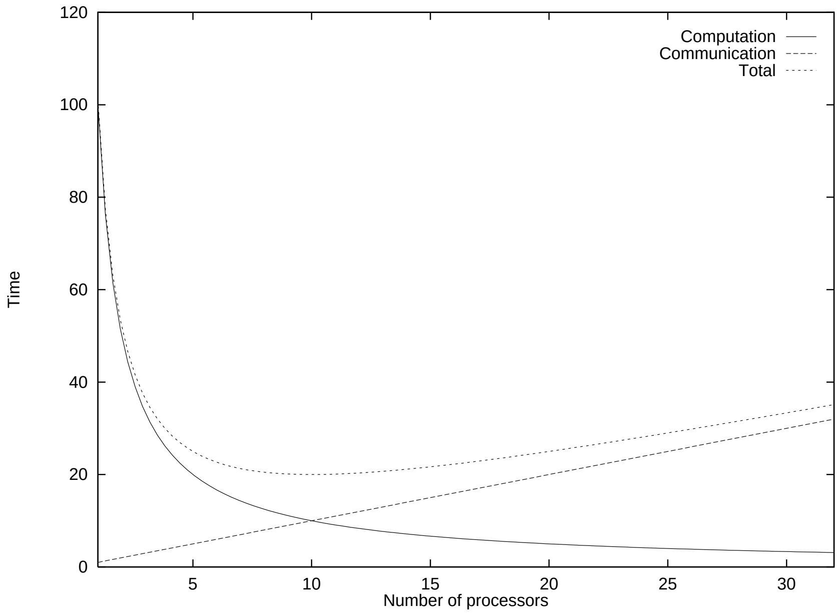


Figure 1.4: Total execution time is composed of computation time and communication time

traffic contention. Assume nodes A, B, C have data sent to nodes C, D, E, respectively. The channel setup time is t for each hop and data transmission will occupy the channel for nt where n is the packet length. The intermediate nodes can begin to forward the data to the next nodes immediately after they receive the data.[2] Without any traffic contention, the data routing can be finished at time nt (assume $n \gg 1$ to ignore the channel setup time). But data moving from node A to C and data moving from B to D will compete for the link BC. Similar contention occurs on link CD. So if nodes A, B, C start to send their data at the same time, the data routing cannot be completed until time $\sim 3nt$. However, if we can delay node B to send its data till time $\sim nt$, then there is no contention between path A $\rightarrow$ C and C $\rightarrow$ E, and the data routing will be finished at time $\sim 2nt$. This simple example shows how the routing latency can be varied due to different degrees of traffic contention.

[2]We assume the wormhole routing flow control. The detail of flow control schemes will be given in Chapter 3.

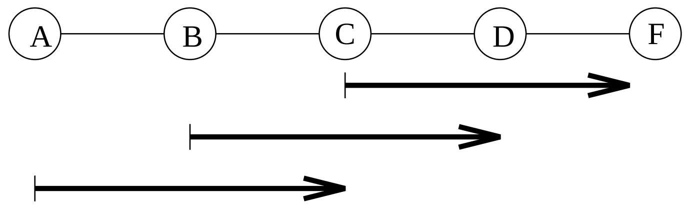


Figure 1.5: An example of traffic contention. Node A to node C, B to D, and C to E. There is traffic contention on link BC and CD.

1.4 An Overview of the Research

In the rest of this dissertation, we will discuss in detail the inter-processor communication issues in parallel systems for which the design goals for an efficient inter-processor communication architecture are high data throughput, low routing latency, low communication energy, and low implementation cost. Chapter 2 gives an overview of network properties and examples of different network topologies. Chapter 3 introduces flow control schemes, virtual channels, and channel configurations. Chapter 4 describes wormhole routing algorithms including deterministic, fully adaptive, partially adaptive, and multicasting algorithms. Simulation results of different routing algorithms and channel configurations are given in Chapter 5. An enhanced mesh or torus architecture with segmented reconfigurable bus (SRB) is introduced in Chapter 6, where also gives an optimization procedure for a torus with SRB. Chapter 7 describes the detail design and implementation of a VLSI wormhole data router for a 2-D mesh or torus. It implements bi-directional channels, dimension-order data routing, and hardware supported path-based multicasting. A completed chip implementation delivers 1.6Gb/s (50MHz) @ Vdd=2.1V and consumes an average power of 15mW. Finally, Chapter 8 summarizes the contributions of this dissertation and suggests some future work in this field.

Chapter 2

Network Interconnection

A multiprocessor network comprises nodes and communication channels, and can be represented by a uni-directional graph $G = (N, C)$, with vertices $N = \{n_1, n_2, \ldots, n_m\}$ corresponding to the nodes and edges $C = \{c_1, c_2, \ldots, c_n\}$ corresponding to the communication links between nodes. An efficient network for multiprocessors should be able to utilize the hardware resources effectively. Network performance usually is measured by latency and throughput: latency is the time required for a message to be delivered from source to destination; throughput is the data rate which is maintained in the network. Both latency and throughput are strongly dependent on the network topology and the routing algorithm employed. In this chapter, we will describe some properties of a network and introduce different network topologies.

2.1 Properties of Network Topology

There are several network properties that are directly related to network performance and complexity (Hwang, 1993). We will describe these network properties in this section and compare these properties for different networks in the next section.

Network Diameter

The network diameter is the maximum shortest path between any two nodes in the network. The diameter is the indication of the worst case latency when the traffic load is low. A designer should keep the network diameter small to reduce latency.

Bisection Width

The network bisection width is the minimum number of channels required to be removed to cut the network into two equal-sized parts. Therefore, the bisection width is related to the maximum communication bandwidth supported between two separated parts in the network.

Number of Links

The total number of links is related to the maximum total bandwidth provided in a network. In an ideal case, all the links can be used for transmitting data at the same time; this would achieve the peak throughput of the network. So the number of links is an indication of degree of concurrency of communications. The number of links also affects the cost of the network interconnection.

Node Degree

The node degree is the number of IO ports associated with each node in the network. When we design a scalable network, we would like to have the node degree independent of the network size to reduce the cost. Limited wire density and pin count in the current VLSI packaging technology restrict the node degree in feasible networks.

Symmetry

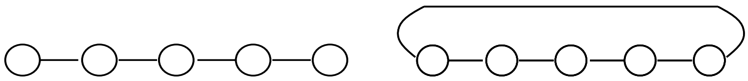
A network is symmetric if it is isomorphic to itself with any node used as the origin. Thus, in a symmetric network, the network is the same seen from any node. For uniform traffic, a symmetric network has the same traffic loading in all channels, and "hot-spot" effects, which may occur in an asymmetric network, are reduced.

Network Mapping

No matter what form of network we design, we need to map the network onto a two- or three-dimensional space for implementation. This network mapping directly affects the physical channel width and wire length. Thus propagation delay, clock speed, and transmission power all depend on the network mapping.

Data Routing

Data routing is the basic function performed by a network. Different network topologies have different characteristics as described above, as well as different node addressing,



(a) Linear Array (b) Ring

Figure 2.1: Linear Array and Ring

communication paths, and channel dependency. Therefore, different data routing algorithms are needed to match the different topologies.

Scalability

Ideally, the network performance should be scaled linearly with an increasing number of processors employed. For example, the network bisection width should be scaled with the network size to support sufficient communication bandwidth for the increased traffic. For a scalable network, the node degree should be constant, the network mapping should be compact, and the data routing should be identical for different size of networks.

2.2 Network Examples

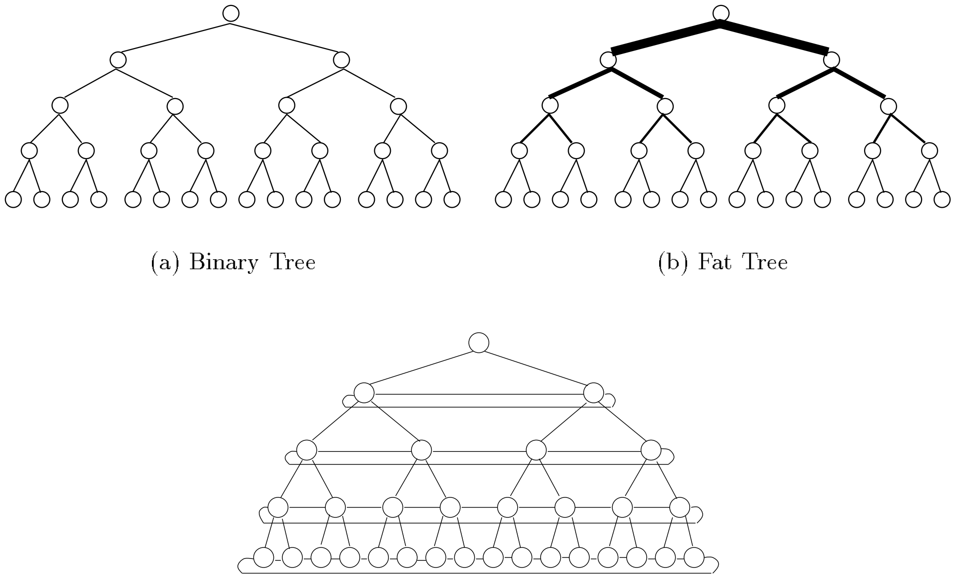
In this section, we will give a brief description of some popular networks and compare their characteristics as defined in the previous section.

Linear Array and Ring

Fig. 2.1 shows the topology of a linear array and ring. Linear arrays and rings are very simple and low-cost, but long diameter and low bisection width causes the latency to increase exponentially as the number of nodes increases, due to traffic contention.

Tree

A binary tree is shown in Fig. 2.2(a). A tree is made up a root, intermediate nodes, and leaves. Long relative distances between leaves on different branches and low bisection width are the main drawbacks of a tree network. Fat trees (Fig. 2.2(b)) have been proposed to increase the bandwidth as we ascend from leaves to the root and then reduce traffic contention near the root. Another variation of the tree structure is the



(a) Binary Tree

(b) Fat Tree

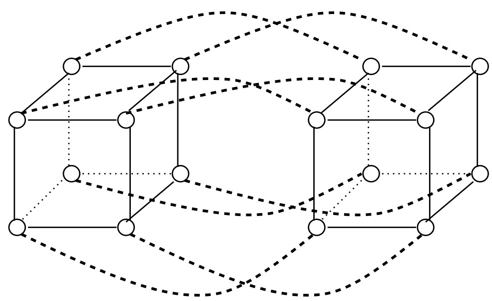
(c) X-Tree

Figure 2.2: Tree Networks

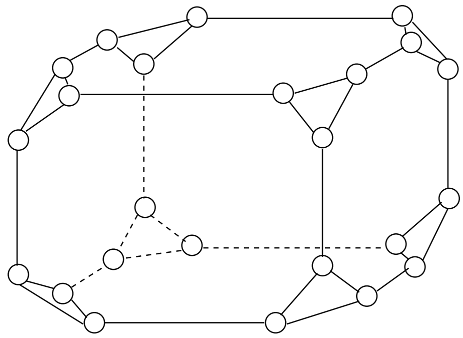
"X-tree" (Fig. 2.2(c)). In an X-tree network, all the nodes at the same level are also connected in a ring to reduce the communication across levels and the bottleneck near the tree root. The tree structures are not only limited to binary connections. The structures can be extended to multiway trees.

Hypercube and Cube-Connected Cycle (CCC)

In general, a hypercube is an n-cube with $N = 2^n$ nodes. There are n dimensions with two nodes per dimension. An order n hypercube can be constructed from two order $n-1$ hypercubes, but the node degree increases from $n-1$ to n (Fig. 2.3(a)). That the node degree depends on the network size makes hypercubes unscalable. A cube-connected cycle (CCC) is a variation of hypercubes. An n-dimensional CCC is a n-cube in which each vertex is replaced by a cycle of n nodes. Therefore, an n-CCC has $N = n2^n$ nodes, and a longer network diameter, but a constant node degree of 3. Fig. 2.3(b) shows a 3-cube topology.



(a) Construct a 4-cube from two 3-cubes



(b) 3-CCC

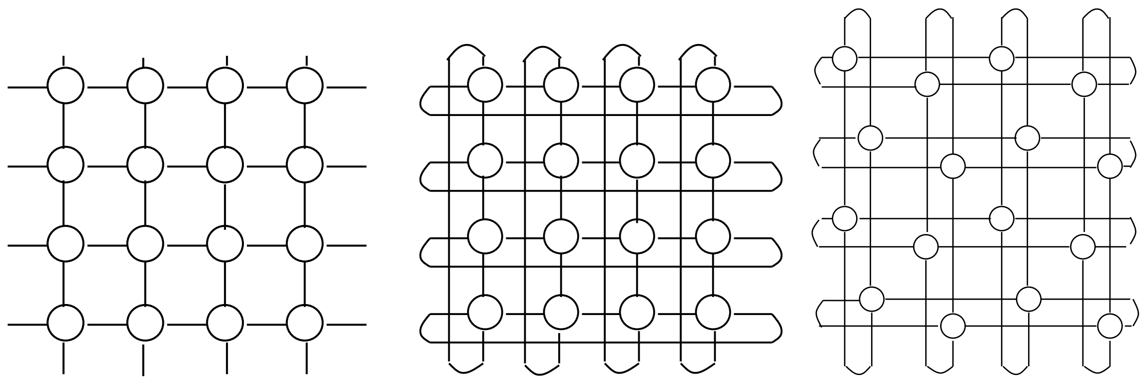
Figure 2.3: Hypercube and CCC

2-D Mesh and Torus

Fig. 2.4 shows the topology of a 2-D mesh and torus. A torus is basically a mesh with wrap-around connections in each row and column to reduce the network diameter. A folded torus, shown in Fig. 2.4(c), can be laidout easily without the long wrap-around connections which increase the propagation delay. 2-D meshes and tori are becoming popular due to their simplicity, regularity, scalability, and feasibility.

k-ary n-cube

This is a very general network topology with n dimensions and k nodes per dimension. Linear arrays, rings, hypercubes, meshes, and tori all belong to this family. Several researchers have produced results in favor of low-dimensional networks. Under the assumption of constant wire bisection width, Dally (1990b) has shown that two-dimensional k-ary n-cube networks can offer the minimum latency with a linear wire delay model. Agarwal (1991) also showed that three or four-dimensional networks perform best under some other constraints, *e.g.*, fixed node size, considerable switching delay, and so on. Several new generation multiprocessors have chosen the k-ary n-cubes family as their interconnection.



(a) Mesh (b) Torus (c) Folded Torus

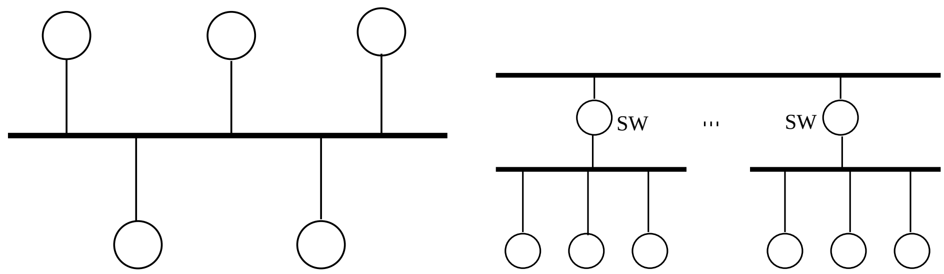
Figure 2.4: 2-D Mesh and Torus

Shared Bus

A shared bus is a very simple way to connect several hosts together (Fig. 2.5(a)). A bus is usually heavily loaded, has long propagation delay, and consumes more energy than other types of interconnections. Bus arbitration needs to be considered because only one node at a time can use the bus. The bus bandwidth is shared among multiple nodes and the traffic contention may become serious when the number of nodes increases. A hierarchical bus structure has been proposed to increase utilization and bandwidth, and to reduce contention (Fig. 2.5(b))(Mahnud, 1994).

Multi-stage Network

A multi-stage interconnect network (MIN) consists of more than one stage of switch elements which can be set up dynamically according to traffic requests. Processing nodes (or memory) are located at the ends of the MIN, and traffic is routed from one end to the other. Different switch "fabrics" and interstage connections have been proposed (Tobagi, 1990; Fen, 1981). Some examples are the *Omega* (or *Shuffle-Exchange*), *Butterfly*, and *Baseline* networks, etc., pictured in Fig. 2.6. We should note that some multi-stage networks are actually equivalent, for instance, Fig. 2.6 (a) is equivalent to (b) with some nodes in the middle stage re-ordered. All the paths in a MIN have the



(a) Shared Bus (b) Hierarchical Bus

Figure 2.5: Bus Networks

same latency, so the designer cannot take advantage of data locality in most applications.

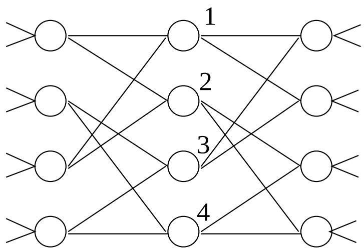
Crossbar

A crossbar switch is an interconnection in which each input is connected to each output through a path that contains a single switching node (Fig. 2.7). It offers the least traffic contention, but has the highest complexity. The cost is proportional to N^2 where N is the network size. Because of the high cost, the crossbar network is not very scalable in a large system.

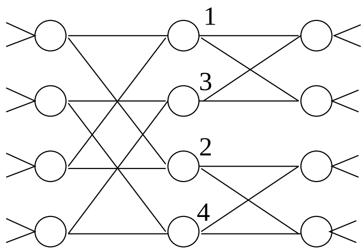
Table 2.1 provides a summary of network characteristics, showing the relation between key physical properties and network size for different network topologies. Table 2.2 surveys the topologies used in some parallel systems. Most of the recent multiprocessor systems have chosen the low dimensional k-ary n-cube as the network interconnection because of its scalability for high bandwidth, regularity for easier data routing, and simplicity for efficient implementation.

2.3 Network Category

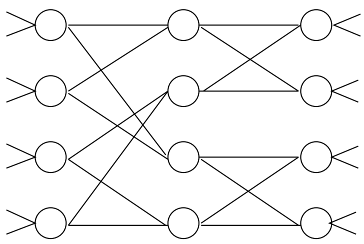
Networks also can be categorized based on interconnection status or node functionality. A network interconnection can be either *static* or *dynamic*. Static networks such as meshes and k-ary n-cubes have all their connections fixed without changing during execution. On the



(a) Omega network



(b) Butterfly network



(c) Baseline network

Figure 2.6: Some multi-stage networks examples

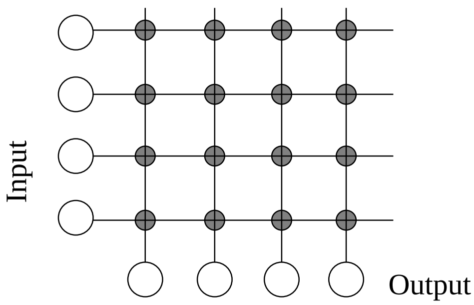


Figure 2.7: Crossbar Network

Network	size	diameter	bisection width	number of links	node degree	symmetry
Linear Array	N	$N-1$	1	$N-1$	2	No
Ring	N	$\lfloor N/2 \rfloor$	1	N	2	Yes
Binary Tree	$N = 2^n - 1$	$2(n-1)$	1	$N-1$	3	No
Hypercube	$N = 2^n$	n	2^{n-1}	$nN/2$	n	Yes
CCC	$N = n2^n$	$2n - 1 + \lfloor n/2 \rfloor$	2^{n-1}	$3N/2$	3	Yes
2-D Mesh	$N = k^2$	$2(k-1)$	k	$2(N-k)$	4	No
2-D Torus	$N = k^2$	$2\lfloor k/2 \rfloor$	$2k$	$2N$	4	Yes
k-ary n-cube	$N = k^n$	$n\lfloor k/2 \rfloor$	$2k^{n-1}$	nN	$2n$	Yes
Shared Bus	N	1	1	1	1	Yes
Omega	N	$\lg N + 1$	$N/2$	$N(\lg N + 1)$	4	No
Crossbar	N	2	N	N	2	Yes

Table 2.1: Summary of network properties

other hand, dynamic networks may change their channel configuration during the running time depending on data routing requirement, for example, multi-stage networks.

A network configuration also can be either *direct* or *indirect*. In a direct network, all the nodes are the processing units as well as switching elements, *i.e.*, the communication channels connect processors directly. Unlike direct networks, indirect networks have some intermediate nodes used for switching only. In this case, messages between processing nodes are routed through the paths set up by switching nodes. A dynamic network is usually an indirect network; for example, a multi-stage network often has its processors or memory at both ends of the network, and has the intermediate stages as switches. A dynamic network also can be a direct network; for example, a reconfigurable mesh may change its connections to the neighbors even though all the nodes are the processing units. Similarly, a static network can be either a direct or indirect network; for instance, a tree network is a direct network if all the nodes are processors, but is an indirect network if only the leaves of the tree are processors.

Machine	Year	Topology	Remarks
CMU/C.mmp	1972	Crossbar	16 processors $\times$ 16 memory
Caltech/Cosmic Cube	1983	Hypercube	64 nodes connected in a binary 6-cube
Intel/iPSC	1985	Hypercube	7 I/O ports form a 7-dim hypercube
IBM/RP3	1985	Omega	512 processors. The interconnect consists of 2 networks. A network with 128 ports (4 levels of 4×4). A combining network with 64 ports (6 levels of 2×2)
TMC/CM-2	1987	Hypercube	CM-2 is made of 8 subcubes. Each subcube contains 16 matrix boards. A matrix board has 512 processors
Cray/Y-MP	1988	Multi-stage Crossbar	8 processors and 256 memory modules connected by 4×4 and 8×8 switches and 1×8 demux
BBN/Butterfly	1989	Butterfly	A 3-stage 512×512 butterfly network constructed by 8×8 switches for a 512 processors system
TMC/CM-5	1991	Fat tree	32 to 1024 processors (max. 16384 proc) bisection width of 1024 nodes is 5GB/s
KSR-1	1991	Fat tree	2 levels of ALLCACHE Engine hierarchy. The ALLCACHE Engine is the fat tree topology. Eng:0 has bandwidth 1GB/s, Eng:1 has 1, 2, or 4GB/s
Intel/Paragon	1991	2-D Mesh	Link bandwidth: 175MB/s full duplex max bisection width: 5.6GB/s
Stanford/DASH	1992	2-D Mesh	16 clusters (4×4 mesh) Each cluster has 4 PEs May extend to 512 clusters
MIT/J-Machine	1992	3-D Mesh	1024-node ($8 \times 8 \times 16$) max limit: 65536 nodes ($32 \times 32 \times 64$)
Caltech/Mosaic C	1992	2-D Mesh	64 Mosaic chips are packaged in an 8×8 array on the circuit board. These boards can construct arbitrarily large 2-D arrays
Cray/T3D	1993	3-D Torus	2048 processors with peak 300 Gflops

Table 2.2: Topologies of existing parallel systems

Chapter 3

Routing Flow Control

Data routing is one of the most important factors in a high performance multiprocessor network. In many situations, communication between processors has become the performance bottleneck. For example, in a shared memory system, processors may wait for the memory responses which need to travel across the network. Therefore, an efficient routing flow scheme and routing algorithm are crucial to achieving good overall system performance.

3.1 Desired Routing Properties

There are three basic types of data routing faults in any kind of network: *deadlock, livelock*, and *starvation*. In a network system, all messages compete with each other for limited resources. Unless the routing conflicts are resolved satisfactorily, one of these faults will occur. A desirable routing flow control will be deadlock-free, livelock-free, and starvation-free.

Deadlock

There are four necessary conditions for deadlock to occur: *mutual exclusion, no preemption, hold and wait*, and *circular wait* (Silberschatz *et al.*, 1991). For example, resource I is occupied by process A, and process A is requesting use of resource II which is occupied by process B. If process B is requesting resource I, and if a resource cannot be released by a process until the new request has been granted, then there is a circular request dependency between process A and B, and deadlock will arise (Fig. 3.1). In

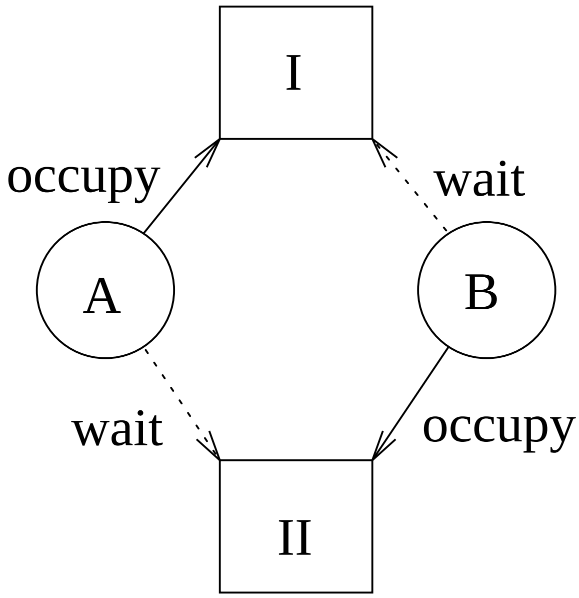


Figure 3.1: An example of deadlock: A is waiting for B to release resource II, and B is waiting for A to release resource I.

data routing, the resources may be buffers (in a store-and-forward network) or channels (in a wormhole network) depending on the routing flow control scheme (Section 3.2). When the circular waiting condition occurs, no message can move forward, and deadlock results. Avoiding deadlock is critical to solution of the data routing problem. We will discuss deadlock avoidance methods in chapter 4.

Livelock

"Livelock" occurs when a packet circulates in a network forever and never arrives at its destination. In the livelock situation, data packets which are circulating in the network consume channel bandwidth and increase traffic contention. If all packets take only the shortest paths to their destinations, *i.e.*, every hop decreases the distance to the destination, then livelock cannot occur because all packets take finite steps to reach the destinations. Livelock can occur in non-minimal routing algorithms where packets detour away from their destinations when they encounter traffic contention.

Starvation

A packet may wait for a resource indefinitely when it is competing with other packets. Starvation is caused by an unbalanced arbitration of resources. For example, a low priority packet might not gain access to a channel which is always requested by other higher priority packets. One solution to the problem of starvation is *aging*. In order to prevent unlimited waiting, aging increases the priority of packets which have waited in the network for a certain time. Aging also increases the overhead of data routing in the form of the extra hardware required to calculate the age of a packet, and the additional data field required to store the age, etc.

3.2 Routing Flow Control Schemes

Flow control is the scheme for allocating communication channels and buffers, and determining the steps to advance messages (Dally, 1990a). In a message-passing network, a message is divided into several *packets* for transmission in the network. Each packet contains its own header on routing information, and can be routed independently. A packet is divided further into several flow control digits or *flits*. A flit is the basic unit for transmission. Only the header flit contains the routing information. The rest of flits follow the header and cannot be routed independently.

Several routing control schemes have been proposed and implemented in parallel machines. In this section, we will describe and compare these different schemes.

Store-and-Forward Routing

Early multiprocessors used store-and-forward as the routing flow control. A packet was treated as an indivisible entity in a store-and-forward network. When a packet reached an intermediate node, the entire packet was buffered in the node. Then the packet was forwarded to the next node on the path to the destination only when the next node had sufficient buffers to hold the packet and the channel was free.

Wormhole Routing

In a wormhole network, only the header flit carries the route information. As the header advances along the path, all the remaining flits follow in pipeline fashion. The intermediate nodes can begin to forward the message as soon as the header has been decoded and the next node on the route has been selected. If the header is blocked, the trailing flits remain distributed in the intermediate nodes along the path.

Virtual Cut-Through Routing

Virtual cut-through routing is similar to wormhole routing in that the flits are pipelined in the network (Kermani and Kleinrock, 1979). But when the header is blocked in this instance, all the flits of the stalled packet are collected by the intermediate node where the blocking occurs.

Pipelined Circuit Switching Routing

Pipelined circuit switching (PCS) (Gaughan and Yalamanchili, 1995) is a variation of wormhole routing. In PCS, data flits do not immediately follow the header into the network. The header travels alone to find the path to its destination. When the header finally reaches the destination, an acknowledge flit returns to the source. Then data flits are then pipelined along the established path in the network.

The first difference among these routing flow control schemes is their latency (Ni and McKinley, 1993). Let L_p be the packet length, L_f be the flit length, B be the channel bandwidth, and D be the distance between the source and destination.

The latency for a *store-and-forward* network is

$$T_{SAF} = \frac{L_p}{B} \times (D+1) \tag{3.1}$$

The latency for a *wormhole* network is

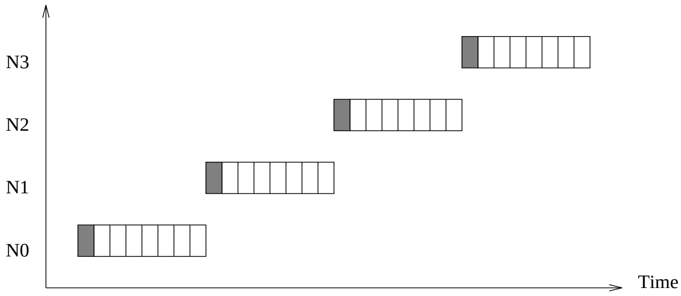
$$T_{WH} = \frac{L_p}{B} + \frac{L_f}{B} \times D \tag{3.2}$$

The latency for a *pipelined circuit switching* network is

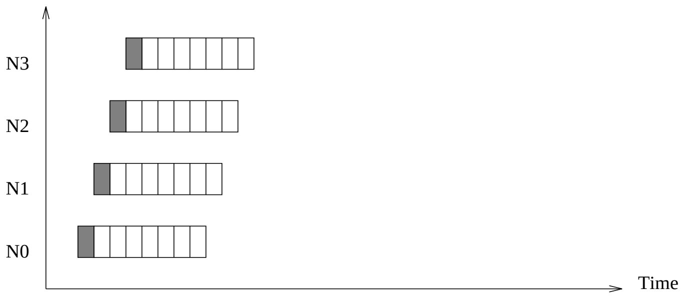
$$T_{PCS} = \frac{L_p - L_f}{B} + 3 \times \frac{L_f}{B} \times D \tag{3.3}$$

Fig. 3.2 provides a comparison of the latency for the different routing flow schemes. L_p is usually larger than L_f, therefore the latency of *store-and-forward* is larger than the latency of *wormhole* family. When $L_p \gg L_f$, the latency of wormhole routing becomes less sensitive to the distance D.

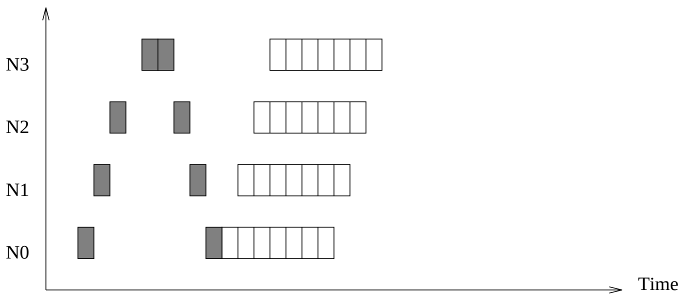
Secondly, the buffer requirement is different for these three schemes. *Store-and-forward* and *virtual cut-through routing* require buffering of the entire packet, so they need storage memory for at least an entire packet inside each node. In contrast, *wormhole* and *pipelined circuit switching routing* allow a packet to be distributed in the intermediate nodes when



(a) Store-and-Forward



(b) Wormhole



(c) Pipelined Circuit Switching

Figure 3.2: Latency comparison of different flow control schemes

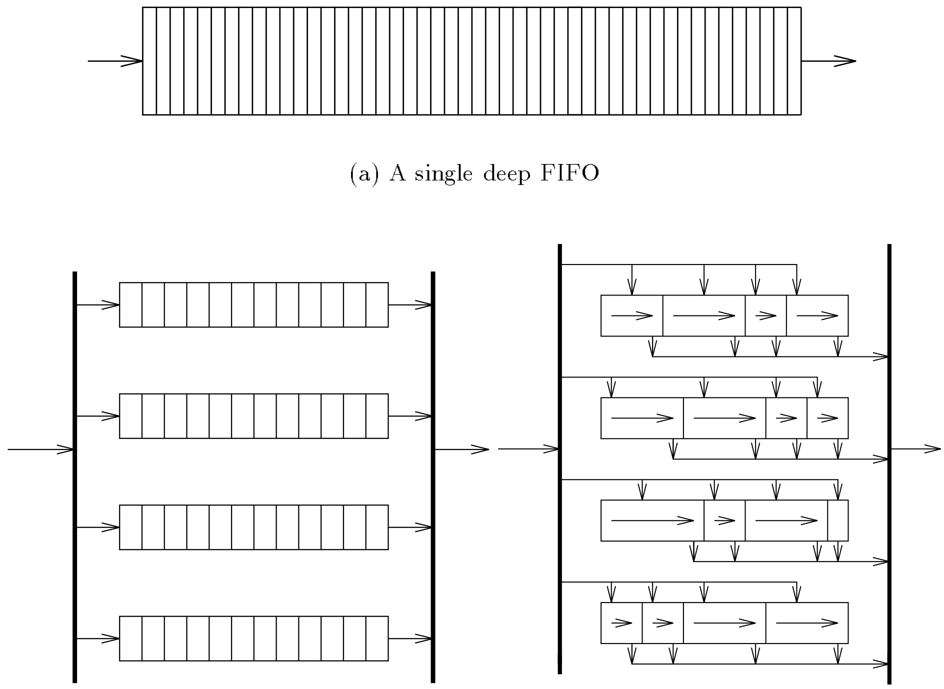
there is traffic contention, thus they require less buffering in each node since only part of a packet must be stored.

Thirdly, different routing flow control schemes have different deadlock avoidance characteristics. In *store-and-forward* and *virtual cut-through routing*, the critical resource is packet buffers, while in *wormhole* routing, the critical resource is communication channels. To avoid deadlock a designer must eliminate circular dependency among resources which are requested by packets. For example, in store-and-forward and virtual cut-through routing networks, messages can be forwarded from one buffer to the next according to a loop-free directed buffer graph which accommodates all the possible message routes (Merlin and Schweitzer, 1980). The buffers can be organized so that they are in an ascending order. When a packet resides in a given buffer, it can be stored only in a restricted set of buffers in the next node of the path such that no circular dependency in the directed buffer graph may occur. In wormhole networks, on the other hand, *channel dependency* is used for resource allocation. To prevent deadlock it is necessary to restrict the routing relation to make the channel dependency acyclic. Virtual channels have been proposed as one method to eliminate cyclic channel dependency (Dally, 1992). In PCS networks, some routing constraints imposed by wormhole routing are relaxed because only headers are traversing in the network during the setup phase of a packet route, and deadlock can be avoided as long as the header has acyclic dependency (Gaughan and Yalamanchili, 1995). The penalty is that PCS networks have longer latency than wormhole networks.

Wormhole routing flow control has been adopted by most of the new generation multiprocessors, for instance, the Intel Paragon, Stanford DASH, and so on. Therefore, we concentrate on wormhole routing in the remainder of discussion.

3.3 Virtual Channel

Virtual channels have been incorporated in multiprocessor networks as a means of preventing deadlock and also of improving performance. Several virtual channels share a physical channel. This is accomplished by time multiplexing the virtual channels on the same physical link.



(a) A single deep FIFO

(b) Several virtual channel FIFOs

(c) Several virtual channel DAMQs

Figure 3.3: FIFO queue configurations. See Sec. 3.3.1 and 3.3.2 for discussion.

3.3.1 Virtual Channel Configuration

A virtual channel is implemented by use of a buffer which can store one or more flits. Instead of having a deep FIFO structure, we may organize the buffers into several independent lanes, *i.e.*, virtual channels (Fig. 3.3). The buffer in each virtual channel can be allocated independently of other virtual channels for different packets, but the virtual channels which are associated with the same physical channel will compete with each other for the channel bandwidth.

When one virtual channel is blocked due to traffic contention, other virtual channels need not be idle since they can still make use of the physical channel. Therefore, virtual channels improve the network throughput substantially for heavily loaded traffic. Adding virtual channels and restricting the routing relation among some virtual channels can eliminate circular channel dependency to avoid deadlock (Chapter 4). Many wormhole routing algorithms are based on virtual channels for this reason. Chapter 4 describes channel dependency and some particular routing algorithms in detail.

3.3.2 Dynamically Allocated Multi-Queue

Organizing buffers such that there are several parallel lanes, *i.e.*, virtual channels, increases the efficiency of physical channels. However, if the buffers within a virtual channel are arranged as a FIFO, then it is still possible that packets in the virtual channel can be blocked unnecessarily. For example, if the packet at the head of FIFO is blocked because it requests a busy output port, then all the packets behind it will be blocked even though they are destined for output ports that are idle.

A dynamically allocated multi-queue (DAMQ) arrangement has been proposed to reduce the effect of "blocked-by-head" (Tamir and Frazier, 1992) (Fig. 3.3 (c)). Packets in a DAMQ buffer destined for different output ports can be accessed separately, and the free space reassigned dynamically to any packet. Multiple queues of packets are maintained in linked lists. Each output direction has its own linked list associated with an input virtual channel. There is another linked list to keep track of free space available. When a packet arrives at the input port, a free space is removed from the free list and linked to the tail of the list corresponding to the output port to which the packet will be routed. When the packet leaves the input buffer, this space will be returned to the free linked list. All the heads of the linked lists for different output directions in an input virtual channel can issue their requests to the router. So there may be more than one request from an input virtual channel configured as a DAMQ while there is at most one request if configured as a conventional FIFO. All the requests are arbitrated so that only one request can be granted per input and output port.

Use of a DAMQ can increase the router throughput and utilization because one of the requests from an input port must be granted if there is no output conflict with other requests from other input ports. However, we have to decide to which output port the packet will be routed when the header arrives at the input port in order to make a proper linked list. Consequently, a routing decision has to be made very quickly upon the arrival of a new packet. Moreover, after a linked list associated with an output direction is established, it is not easy to change the desired output direction of the packet. Thus, it is difficult to implement a DAMQ with adaptive routing (Section 4.2) where the routing direction may be changed due to local traffic contention.

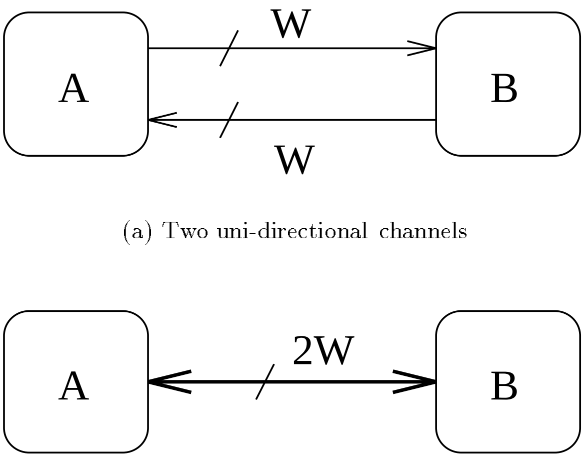
3.4 Uni-directional versus Bi-directional Channel

The links between two adjacent nodes can be organized either as a pair of opposite uni-directional channels, one for transmitting and the other for receiving (Reese *et al.*, 1994), or combined into a single bi-directional channel (Fig 3.4). The choice of different link configurations affects the efficiency of channel utilization. If the total link width is a constant $2W$, then a single bi-directional channel can have full channel width $2W$, but each uni-directional channel can only have half of the channel width. When two uni-directional channels are not fully utilized, one may be busy while the other is idle, and up to one half of the channel bandwidth is wasted. However, bi-directional channels may have longer propagation delay due to increased capacitance loading on the channel. Also, a special arbitration is necessary for bi-directional channels to prevent conflict and deadlock, and this arbitration introduces some additional overhead. Doubling the bandwidth available for data transmission halves the packet length in terms of flits because each flit size is doubled, and the same amount of information can be encoded into one half the number of flits. Buffer storage is halved also in terms of the number of flits if the total storage space is constant. Therefore if a packet is blocked in the network, it will be distributed over the same number of nodes regardless of which scheme we use.

3.4.1 Token Exchange for Bi-directional Channels

A token exchange mechanism is used for single bi-directional channels to prevent conflict caused by two neighboring nodes attempting to use the channel at the same time. A token is associated with each physical channel. Only the node with the token uses the channel to transmit data. A node without the token listens to the channel as a receiver. When a receiving node has data to transmit, it sends a request to the adjacent node which currently “owns” the token. The current “owner” can grant the request by sending an acknowledgment back to the requesting node, and the roles of these two nodes exchange (Dally and Song, 1987).

The state diagram of the token exchange is shown in Fig. 3.5. Two neighboring nodes, A and B, start at state *No_token* and *With_token*, respectively. The signal *TE* is low initially. Node A, which does not have the token, can send a request by making *TE=1*, and enter state *No_token_req*. Node B, which is in state *With_token*, goes to state *With_token_req* after *TE* goes high. Node B will wait in state *With_token_req* until the *finish* signal goes high, which means it can give up the token now. Then node B will lower *TE* and enters state *No_token*.



(a) Two uni-directional channels

(b) Single bi-directional channel

Figure 3.4: Channel configuration

Node A, which now is waiting in state *No_token_req*, senses the signal *TE* becoming low, and knows that the token has been granted. It enters state *With_token* and begins to transmit data. Fig. 3.6 shows the timing diagram of the token-exchange handshaking between node A and B where *enableA* and *enableB* designate which node is driving the signal *TE*.

One of two conditions must hold if a node is to give up its token (*finish=1*): 1) it was idle in the previous cycle, or 2) the previous flit it sent is a tail of a packet.

Theorem 1 *The token-exchange channel arbitration is conflict-free and deadlock-free.*

Proof: First, we will prove the property of "conservation of token," *i.e.*, there is one and only one token associated with a channel. From the definition of token states, only the node in the state named with *With_token* or *With_token_req* has the channel token. The token exchange takes place when *TE* goes from high to low (Fig. 3.6). This event is triggered by the state transition from *With_token_req* to *No_token* in node B, and will cause the state transition from *No_token_req* to *With_token* in node A. This sequence of transitions transfers the token from A to B without any overlap and completes in finite time (less than one clock cycle). Therefore, conservation of token holds.

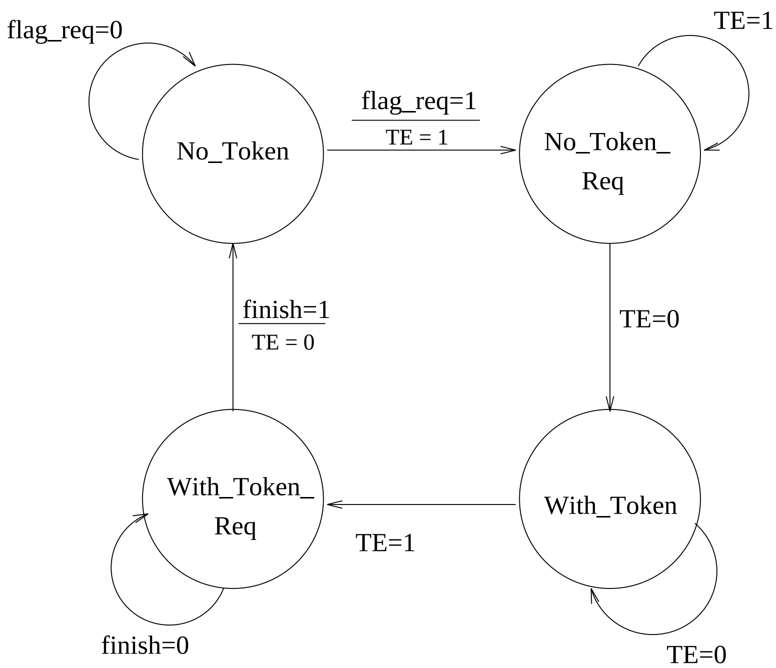


Figure 3.5: Token exchange state diagram

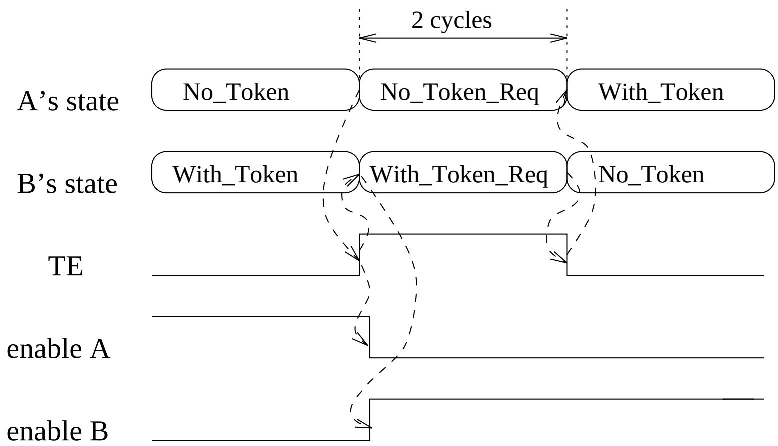


Figure 3.6: Token exchange timing diagram

The state transition from *With_token_req* to *No_token* is caused by *finish=1*. The conditions for setting *finish* guarantee that *finish* will equal one in finite time because the packet length is finite. So this state transition will complete in finite time and the token exchange can also complete in finite time.

Based on the above two properties: conservation of token and finite-time token exchange, we can conclude that it is conflict-free and deadlock-free. □

In the ideal case of no traffic contention the time for token exchange need not increase the latency. In practice, network routers are pipelined; for example, in a four-stage pipeline the elements might be an input buffer, header decoder, crossbar switch, and output control. Use of the output channel and the token-exchange are in the last stage of pipeline. The need to request the channel token is always known before the last stage, for example, in the header decoder stage. So the request for the token can be made in advance, for example, in the crossbar switch stage. Without traffic contention, the token transfer "just in time" for the next cycle, *i.e.*, the output control stage, and hide the token-exchange latency.

The other issue related to the bi-directional channel is that when we have the full bandwidth of the channel, the width of each flit is doubled compared with the uni-directional case, which means the crossbar switch inside the node must double too. But the total area and capacitance loading of a multiline switch are proportional to the square of the width of a flit, and the penalty for increasing the size of the switch is high. We can approach the problem of switching in another way. In our experience, the critical path inside a node is the header decoding. Although the crossbar usually is heavily loaded, it has very shallow logic depth, and is much faster than the path in the header decoding, especially for adaptive routing. Therefore we may keep the size of the crossbar unchanged but double the clock in the crossbar to match the bandwidth of the external channels.

We will show in simulation that in spite of the increased overhead of token exchange, the single bi-directional channel has better overall channel utilization, and thus better latency-throughput performance (see Chapter 5).

Chapter 4

Wormhole Routing Algorithms

In this chapter, we give the theoretical background and definitions required to understand wormhole data routing. We develop the routing algorithms in the following sections.

A routing algorithm determines where to send a packet according to its source, destination, and possibly local traffic conditions. An interconnection network, as defined in Chapter 2, is a uni-directed graph $G = (N, C)$, with vertices $N = \{n_1, n_2, \ldots, n_m\}$ corresponding to the nodes and edges $C = \{c_1, c_2, \ldots, c_n\}$ corresponding to the communication channels between nodes. For a channel c_i, the nodes s_i and d_i are the source and destination of c_i, such that source(c_i)=s_i and dest(c_i)=d_i, respectively (Fig. 4.1). We use the following definitions (Definition 1 to 4) from Duato (1993):

Definition 1 *Routing Function R*

A routing function $R : N \times N \rightarrow P(C)$, where $P(C)$ is the set of communication channels C, provides a set of alternative output channels to send a packet from the current node n_c to the destination n_d, i.e., $R(n_c, n_d) = \{c_{c_1}, c_{c_2}, \ldots, c_{c_p}\}$, *where source($c_{c_i}$)=$n_c$.*

A routing function defines all the permissible movements which avoid deadlock and guarantee delivery according to the current position and destination. For an adaptive routing, we

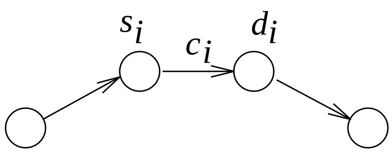


Figure 4.1: Channel c_i has a source node s_i and destination node d_i

may have more than one possible output channel supplied by the routing function if $n_c \neq n_d$. On the other hand, for a deterministic routing, there is only one possible output channel given any pair (n_c, n_d) if $n_c \neq n_d$.

Definition 2 *Local traffic F*

Local traffic F specifies the traffic conditions of the output channels in a node.

The local traffic variable F may be very simple. For example, it might indicate only whether the output channel is busy or not busy. Or it could be complicated. For example, F might represent the traffic load of each output channel, *e.g.*, available buffers left at the receiver.

Definition 3 *Selection function S*

A selection function $S : P(C \times F) \to C'$, where C' is a subset of C, considers the traffic conditions of all candidate routings to select an output channel from the set supplied by the routing function R.

The selection function affects performance, for example, by choosing a lightly loaded output to avoid traffic contention. When an output channel is the candidate of the routing function of several packets, the choice of which packet is granted use of this output channel has to be balanced in order to ensure that starvation will not occur.

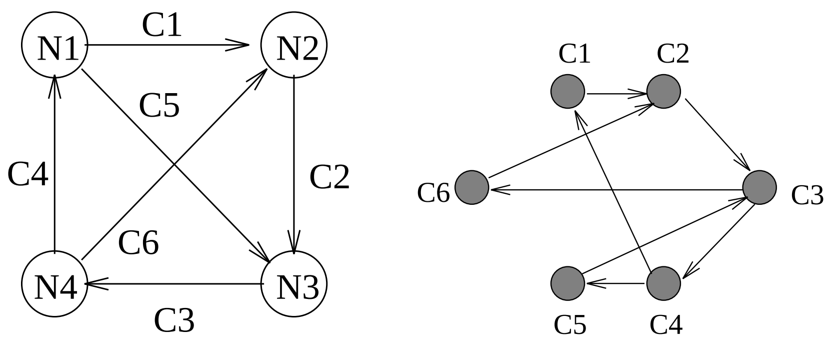
Definition 4 *Channel Dependency*

For a given network and routing function R, the channel dependency graph $D = (C, E)$ is defined as: the channels C are the vertices of D, and the edges E are the pairs of channels connected by R:

$E = \{(c_i,\ c_j) \mid c_i \in R(s_i, n)$ *and* $c_j \in R(d_i, n)$ *for some destination* $n \in N\}$

Fig. 4.2 shows an example of channel dependency D based on a network graph G. We call the above definition of channel dependency *direct* dependency because we can use c_j immediately after c_i to transmit data by the path $(c_i, c_j, \ldots)$ to the destination n. The channel dependency is very important for resolution of the deadlock issue in data routing.

Theorem 2 *(Duato, 1993) A routing function R for a given network is deadlock-free if there is no cycle in the direct channel dependency graph D.*



(a) Network graph G (b) Channel dependency D

Figure 4.2: Relation between network graph G and channel dependency D. The edge between C1 and C2 in channel dependency D indicates that C2 can be used immediately after C1 (via Node N2)

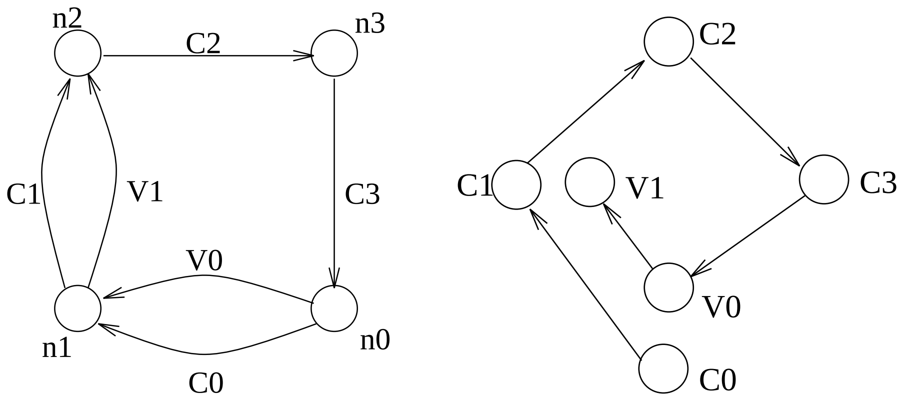
Virtual channels (Section 3.3.1) are used in many routing algorithms to break closed cycles in the channel dependency graph and thereby prevent deadlock. An example of deadlock avoidance by virtual channels is shown in Fig. 4.3. By adding virtual channels and restricting the routing function[1], we can break the circular channel dependency as in Fig. 4.3(b).

Because of the importance of k-ary n-cube networks in the recent development of multiprocessors, we will describe some specific routing algorithms for the general k-ary n-cube networks in the following sections.

4.1 Deterministic Routing

In deterministic routing, the routing path depends only on the source and destination. When a packet is injected into a network, a unique path is determined corresponding to its source and destination. In a k-ary n-cube network, the most commonly used routing algorithm is the *e-cube* algorithm, also known as the *dimension-order routing* algorithm (Dally and Seitz, 1987). The dimension-order routing algorithm has been implemented in most wormhole router designs due to its simplicity and low cost. In the dimension-order routing algorithm, messages are routed in a strictly ascending order of dimensions. Each packet is routed in one

[1]For example, if a packet at node n0 came from channel C3, then it has to use channel V0 instead of C0 to node n1.



(a) Adding virtual channels V0 and V1

(b) Virtual channel dependency

Figure 4.3: Deadlock avoidance by adding virtual channels

dimension at a time: it takes hops in *dim0* (if any), then in *dim1* (if any), and so on until it reaches its destination. Therefore, there is only one pre-determined path between any pair of source and destination nodes. If there are no wrap-around connections in each dimension, then the dimension-order routing is deadlock-free because only lower dimensional channels need to wait for higher dimensional channels, but higher dimensional channels never wait for any lower dimensional channel. Channels in the same dimension will not have circular waits in absence of wrap-around connections. So there is never a closed cycle in the channel dependency.

However, if there are wrap-around connections, then deadlock can occur. Dally has developed a deadlock-free routing algorithm for k-ary n-cubes with wrap-around connections based on adding virtual channels (Dally and Seitz, 1987). Each physical channel is divided into two virtual channels: lower and upper channels. A packet generated in a source node starts from a lower channel in the lowest dimension on its path to the destination. The packet continues in the lower channels until it goes through the wrap-around connection if necessary. After it takes the wrap-around connection, however, it jumps to upper channels. The packet returns to lower channels when it is routed to a higher dimension.

Theorem 3 *In a k-ary n-cube with wrap-around connections, two virtual channels per physical link can provide deadlock-free dimension-order routing.*

Proof: (Dally and Seitz, 1987) We label each virtual channel as $c_{(a_0,a_1,\ldots,a_{n-1}),dir,dim,vc}$, where $(a_0, a_1, \ldots, a_{n-1})$ is the coordinate of the source node of this link, dir is the virtual channel direction ($+$ or $-$), dim is the current dimension (from 0 to $n-1$), and vc indicates it is a lower ($vc = 0$) or upper ($vc = 1$) virtual channel. Then we can assign a value V to each virtual channel based on its label:

$$V(c_{(a_0,a_1,\ldots,a_{n-1}),dir,dim,vc}) = \begin{cases} 2k \cdot dim + k \cdot vc + a_{dim} & \text{if } dir = + \\ 2k \cdot dim + k \cdot vc + (k - 1 - a_{dim}) & \text{if } dir = - \end{cases} \tag{4.1}$$

The channel dependency of the dimension-order routing function R have the following possible relations:

$$\begin{array}{ll}
(c_{(a_0,a_1,\ldots,a_{dim},\ldots,a_{n-1}),+,dim,0}, & c_{(a_0,a_1,\ldots,a_{dim}+1,\ldots,a_{n-1}),+,dim,0}) \\
(c_{(a_0,a_1,\ldots,a_{dim}=k-1,\ldots,a_{n-1}),+,dim,0}, & c_{(a_0,a_1,\ldots,a_{dim}=0,\ldots,a_{n-1}),+,dim,1}) \\
(c_{(a_0,a_1,\ldots,a_{dim},\ldots,a_{n-1}),-,dim,0}, & c_{(a_0,a_1,\ldots,a_{dim}-1,\ldots,a_{n-1}),-,dim,0}) \\
(c_{(a_0,a_1,\ldots,a_{dim}=0,\ldots,a_{n-1}),-,dim,0}, & c_{(a_0,a_1,\ldots,a_{dim}=k-1,\ldots,a_{n-1}),-,dim,1}) \\
(c_{(a_0,a_1,\ldots,a_{dim},\ldots,a_{n-1}),\pm,dim,(0 or 1)}, & c_{(a_0,a_1,\ldots,a_{dim},a_{dim+1}+1,\ldots,a_{n-1}),+,dim+1,0}) \\
(c_{(a_0,a_1,\ldots,a_{dim},\ldots,a_{n-1}),\pm,dim,(0 or 1)}, & c_{(a_0,a_1,\ldots,a_{dim},a_{dim+1}-1,\ldots,a_{n-1}),-,dim+1,0})
\end{array}$$

Based on the assignment by Eq. 4.1, the channel dependency increases the virtual channel value V as a packet travels from the source to destination. Therefore, there is no cycle in the channel dependency graph, and it is deadlock-free. □

Fig. 4.4 shows examples of the dimension-order routing in a 2-D torus. For the path (3,1) → (0,1) → (0,0) → (0,3), the virtual channel values V defined by Eq. 4.1 are 3 → 10 → 11, *i.e.*, in a strictly ascending order. Because the maximum distance between the source and destination in any dimension is half of the torus size, half of the higher level virtual channels will never be used (shaded buffers in Fig. 4.4). In spite of the simplicity of the dimension-order routing algorithm, it performs well under uniform random traffic condition. However, the performance degrades quickly if there are hot spots in the traffic patterns. Furthermore, there is no capability for fault-tolerance in the dimension-order routing algorithm.

4.2 Fully Adaptive Routing

In a multiprocessor network, there are usually multiple paths from a source to a destination. In adaptive routing, the routing function R may have more than one output channel as a

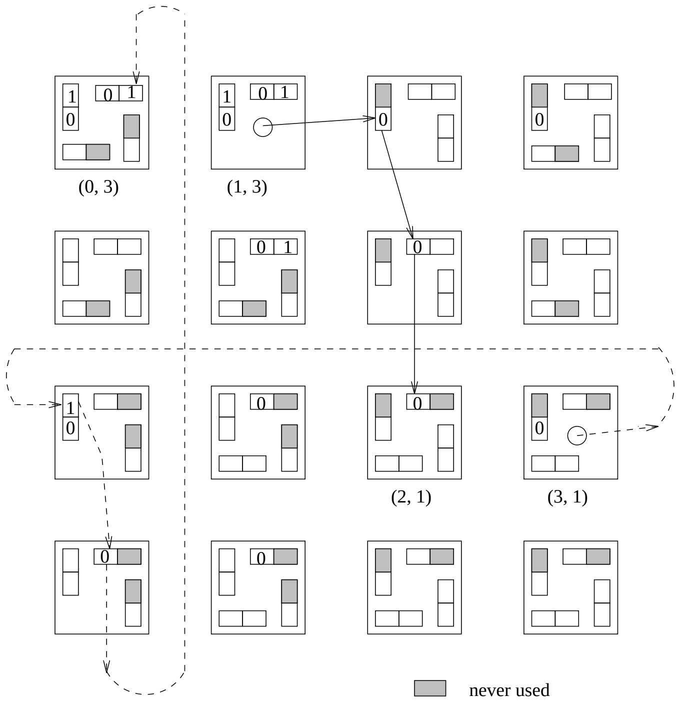


Figure 4.4: Examples of dimension-order routing in a 2-D torus. Only input buffers are shown. Some buffers are omitted for simplicity. Packets are routed from node (3,1) to (0,3) and from (1,3) to (2,1). Half of the upper channels are never used (shaded buffers) because the maximum packet distance is half the torus size due to wrap-around connections.

candidate, and the selection function S will choose one of them based on the local traffic condition F. If a routing algorithm can use all possible shortest paths between any pair of source and destination nodes, then it is a fully adaptive routing algorithm. On the other hand, if a routing algorithm can only use a subset of the possible paths, then it is a partially adaptive algorithm. We will describe some fully adaptive routing algorithms in this section, and partially adaptive routing algorithms in the next section.

4.2.1 Virtual Network Algorithm

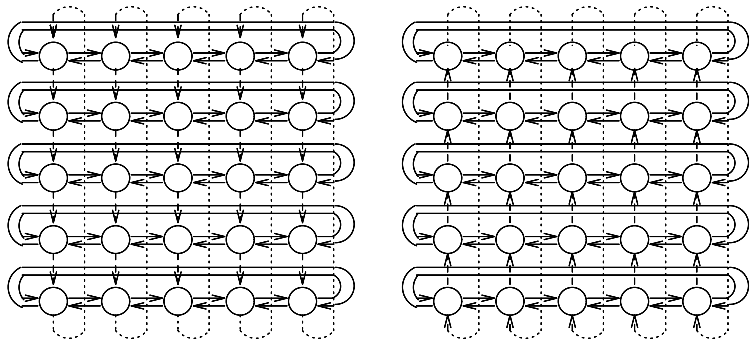
Deadlock-free adaptive routing can be achieved by partitioning a physical network into several virtual networks (Linder and Harden, 1991). A virtual network is a collection of sets of virtual channels in which there is no cyclic channel dependency. Such a virtual network can provide all possible paths of adaptive routing for this packet without deadlock. When a packet is generated, the choice of virtual network is based on its source and destination.

We formally define a virtual network as follows (Linder and Harden, 1991):

Definition 5 *(Linder and Harden, 1991) Each virtual network is identified by a vector VN, where $VN = \{d_1, d_2, \ldots, d_{n-1}\}$, and $d_i = \{$ 0 or 1 $\}$ for $i \in [1, n-1]$. A virtual network vn_{VN} is defined as*

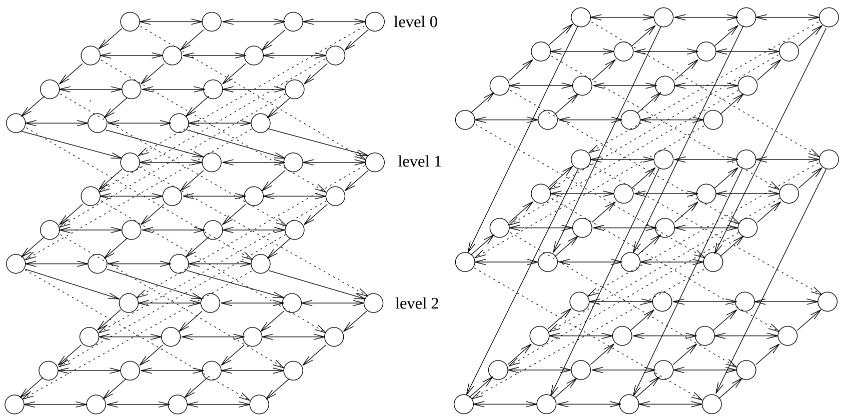
$$\begin{cases} \textit{the channels in dimension } d_i \textit{ that point exclusively in the positive direction if } d_i = 1 \\ \textit{the channels in dimension } d_i \textit{ that point exclusively in the negative direction if } d_i = 0 \end{cases}$$

Fig. 4.5 shows the example of virtual networks for a 2-D torus. In shortest-path routing, a packet never changes the direction, *e.g.*, from positive to negative, in any dimension on its path. Therefore, a packet stays in the same virtual network once it is generated. If there is no wrap-around connection, then these virtual networks are sufficient to provide deadlock-free adaptive routing. For an n-cube, there are at most n wrap-around connections which a packet may take. Thus we expand each virtual network into $n+1$ levels. A packet starts at level 0, and moves to a higher level whenever it takes a wrap-around link. Fig. 4.6 shows the three level construction of the negative virtual network in a 2-D torus. We label a virtual channel as $c_{VN,L,(a_0,\ldots,a_{n-1}),dir,dim}$, where $VN = \{d_1, d_2, \ldots, d_{n-1}\}$ is the virtual network identification, L is the level for wrap-around connections, $(a_0, a_1, \ldots, a_{n-1})$ is the coordinate of the source node of this link, dim is the current dimension (from 0 to $n-1$), and dir is the virtual channel direction ($dir = d_{dim}$ if $dim > 0$).



(a) Negative Virtual Network (b) Positive Virtual Network

Figure 4.5: Two virtual networks in a 2-D torus



(a) Negative Virtual Network (b) Positive Virtual Network

Figure 4.6: Expanding a virtual network to three logical levels in a 2-D torus

Theorem 4 *In a k-ary n-cube with wrap-around connections, the virtual network adaptive routing is deadlock-free.*

Proof: (Linder and Harden, 1991) Similar to the proof of dimension-order routing, we can assign a value to each virtual channel such that all the channel dependency is in strictly ascending order.

$$\begin{aligned} V(c_{VN,L,(a_0,\ldots,a_{n-1}),dir,dim}) &= L \cdot k^n + \sum_{i=1}^{n-1} \begin{cases} a_i \cdot k^i & \text{if } d_i = 1 \\ (k-1-a_i) \cdot k^i & \text{if } d_i = 0 \end{cases} \\ &+ \begin{cases} k & \text{if } dim \neq 0 \\ a_0 & \text{if } dim = 0 \text{ and } dir = + \\ (k-1-a_0) & \text{if } dim = 0 \text{ and } dir = - \end{cases} \end{aligned} \tag{4.2}$$

Where d_i is the vector element of VN. Based on the adaptive routing function R, we have the following channel dependency:

$$\begin{array}{lll} (c_{VN,L,(a_0,\ldots,a_{n-1}),\pm,0}, & c_{VN,L,(a_0\pm1\ldots,a_{n-1}),dir,dim}) & \\ (c_{VN,L,(a_0=k+1,\ldots,a_{n-1}),+,0}, & c_{VN,L+1,(a_0=0\ldots,a_{n-1}),dir,dim}) & \\ (c_{VN,L,(a_0=0,\ldots,a_{n-1}),-,0}, & c_{VN,L+1,(a_0=k+1\ldots,a_{n-1}),dir,dim}) & \\ (c_{VN,L,(a_0,\ldots,a_{n-1}),d_i,i}, & c_{VN,L,(a_0,\ldots,a_i+1,\ldots,a_{n-1}),dir2,dim2}) & \text{if } d_i = 1 \\ (c_{VN,L,(a_0,\ldots,a_{n-1}),d_i,i}, & c_{VN,L,(a_0,\ldots,a_i-1,\ldots,a_{n-1}),dir2,dim2}) & \text{if } d_i = 0 \\ (c_{VN,L,(a_0,\ldots,a_i=k+1,\ldots,a_{n-1}),d_i,i}, & c_{VN,L+1,(a_0,\ldots,a_i=0,\ldots,a_{n-1}),dir2,dim2}) & \text{if } d_i = 1 \\ (c_{VN,L,(a_0,\ldots,a_i=0,\ldots,a_{n-1}),d_i,i}, & c_{VN,L+1,(a_0,\ldots,a_i=k+1,\ldots,a_{n-1}),dir2,dim2}) & \text{if } d_i = 0 \end{array}$$

Based on the assignment by Eq. 4.2, all the virtual channel values are increased as a packet travels from the source to destination for all possible routes. Therefore, there is no cycle in the channel dependency graph, and it is deadlock free. □

Fig. 4.7 shows the routing examples of the virtual network adaptive routing. For the path (2,1) → (3,1) → (3,0) → (0,0) → (0,3), the virtual channel values V defined by Eq. 4.2 are 10 → 12 → 15 → 32. We can also view the path on the expanded virtual network graph (Fig. 4.8). The packet goes to a different logic level when it takes a wrap-around link.

The drawback of the virtual network approach is that it requires a large number of virtual channels because none of the virtual networks share any virtual channels. For a k-ary n-cube network, there are 2^{n-1} virtual networks, and each virtual network has $(n+1)$ levels.

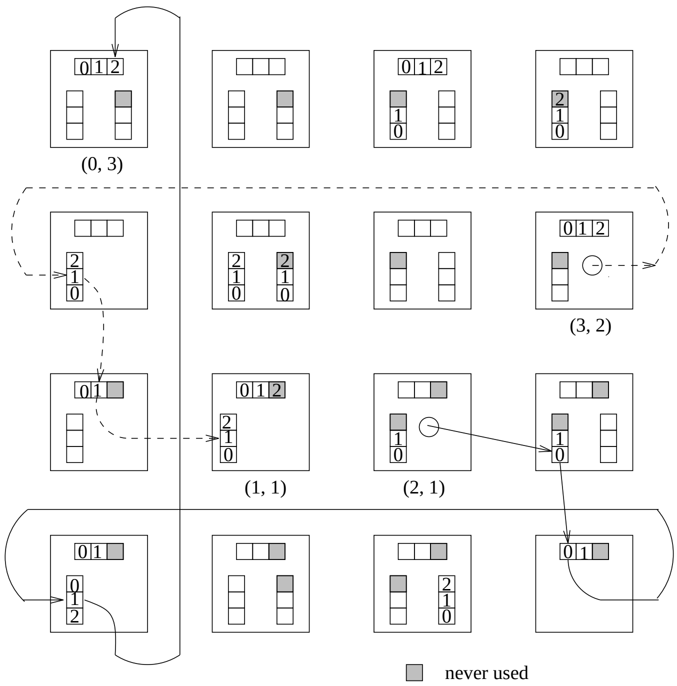


Figure 4.7: Examples of Virtual Network routing on the negative network in a 2-D torus. Packets are routed from (2,1) to (0,3) and from (3,2) to (1,1).

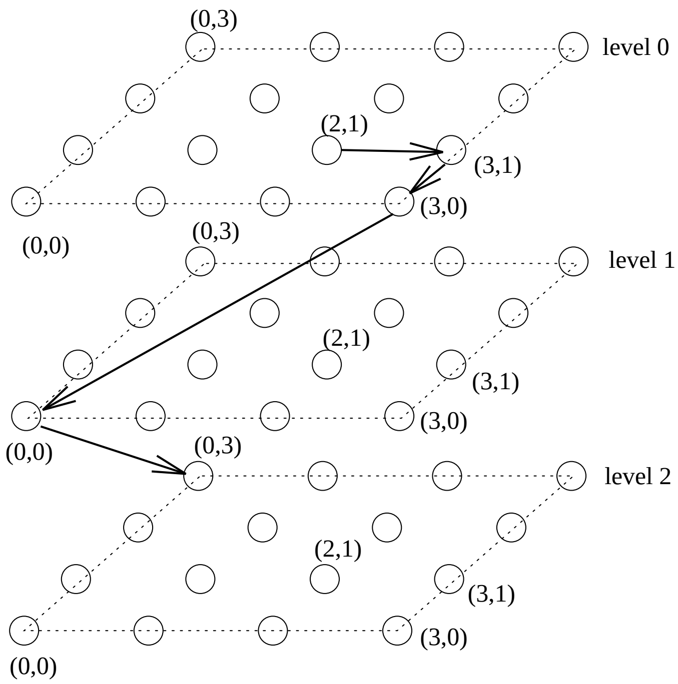


Figure 4.8: A packets is routed from (2,1) to (0,3) on the negative virtual network in a 2-D torus. The packet goes to a different logic level when it takes a wrap-around link.

Therefore, the total number of virtual channels required per port is $(n+1)2^{n-1}$ for dimension 0 and $(n+1)2^{n-2}$ for other dimensions, which may not be feasible in a high dimensional network. The other disadvantage of the virtual network algorithm is that the virtual channel utilization of higher level networks tends to be very low because the probability that a packet will take many wrap-around links on its path is small.

4.2.2 Dimension Reversal Algorithm

The dimension reversal (DR) number of a packet is the count of the number of times that this packet has been routed from a higher dimension to a lower dimension, which is the reverse of the dimension order. All packets are initialized with $DR = 0$. Whenever a packet has a dimension reversed route, the DR of the packet is incremented. Dally and Aoki (1993) developed an adaptive routing algorithm based on keeping track of the dimension reversal numbers of packets.

In the Dally and Aoki's scheme, the virtual channels associated with a physical channel are divided into two classes: adaptive and deterministic channels. A packet is generated in the adaptive channels. While in the adaptive channels, a packet may be routed in any direction. A packet labels a virtual channel with its current DR when it acquires this channel. In order to prevent deadlock, a packet with $DP = p$ cannot wait for a busy virtual channel with label $DR = q$ if $q \leq p$. If a packet reaches a node where all output channels are busy and have equal or lower DR labels, then this packet is forced to switch to the deterministic channels and follow the dimension-order routing on the rest of the path to its destination, and is not allowed to jump back to the adaptive channels.

Theorem 5 *The dimension reversal adaptive routing algorithm is deadlock free.*

Proof:(Dally and Aoki, 1993) Assuming the network is deadlocked, then there is a set of packets P in the cyclic channel dependent situation. There exists a packet p_{max} with a maximum $DR(p_{max})$. If p_{max} is waiting for some channel c occupied by packet p_j with $DR = r$ in set P, then $r \leq DR(p_{max})$. But p_{max} is not allowed to wait for a channel with equal or lower DR label. Therefore, there is contradiction and no deadlock. $\square$

When a packet is in adaptive channels, there is no restriction on which direction to route the packet. So a packet may take a hop which will *increase* the distance to its destination if all output channels on the shortest paths are busy, *i.e.*, it may follow a non-minimal routing. In non-minimal routing, however, "livelock" can become a problem (Section 3.1). In practice, we must put a limit on the maximum number of DR due to counter width, storage space, etc. When a packet reaches its DR limit, the algorithm cannot account for more dimension reversals. Consequently, the packet is directed to the deterministic channels for dimension-order routing; it will be delivered to the destination eventually. Strictly speaking, the dimension reversal adaptive routing algorithm is not really a fully adaptive routing algorithm in the sense that a packet loses freedom of adaptivity in those situations where it has to take the deterministic channels for the rest of its path. In order to support more adaptivity, it is necessary to increase the upper limit of DR and use more adaptive virtual channels. If the number of adaptive virtual channels is too small, packets easily become congested in adaptive channels and are converted to the deterministic channels. In this instance the DR algorithm behaves more like the dimension-order routing algorithm.

4.2.3 Star-Channel Algorithm

The Star-Channel algorithm is a fully adaptive, minimal wormhole routing algorithm which requires a small constant number of virtual channels per bi-directional link, independent of the size and dimension of the k-ary n-cube networks (Berman *et al.*, 1992).

The algorithm makes use of two kinds of virtual channels: *star* channels and *non-star* channels. The *star* channels are used when packets are routed on the dimension-order paths. When a packet takes some path which is not allowed by the dimension-order routing, it will use the *non-star* channels. Therefore, for a physical channel between node $(a_0, \ldots, a_i, \ldots, a_{n-1})$ and $(a_0, \ldots, (a_i + 1) \bmod k, \ldots, a_{n-1})$, there are three virtual channels associated with it:

$$c^*_{(a_0,\ldots,a_i,\ldots,a_{n-1}),+,i,0}\ ,\ c^*_{(a_0,\ldots,a_i,\ldots,a_{n-1}),+,i,1}\ ,\ c_{(a_0,\ldots,a_i,\ldots,a_{n-1}),+,i}$$

Similarly, for a link between $(a_0, \ldots, a_i, \ldots, a_{n-1})$ and $(a_0, \ldots, (a_i - 1) \bmod k, \ldots, a_{n-1})$, we may have

$$c^*_{(a_0,\ldots,a_i,\ldots,a_{n-1}),-,i,0}\ ,\ c^*_{(a_0,\ldots,a_i,\ldots,a_{n-1}),-,i,1}\ ,\ c_{(a_0,\ldots,a_i,\ldots,a_{n-1}),-,i}$$

where c^* channels are defined to be the same as the dimension-order routing in Section 4.1. Assume a packet is transmitted initially on dimension i. If dimension i is the lowest dimension in which the packet needs to be transmitted for all its minimal paths, then it goes through the star channel c^*_i in this dimension, otherwise it uses the non-star channel c_i because dimension i is not the lowest dimension and it is not allowed by the *dimension-order* routing. Each star channel c^*_i is partitioned into two levels: $c^*_{i,0}$ and $c^*_{i,1}$. If the packet has taken a wrap-around link along dimension i, then it uses $c^*_{i,1}$, if it has not, then it chooses $c^*_{i,0}$ (Berman *et al.*, 1992). A packet will enter a star channel when the current dimension is the most significant dimension defined by the *dimension-order* routing, and it is allowed to correct any other dimensions along a minimal path through non-star channels (Fig. 4.9).

Theorem 6 *The Star-Channel routing algorithm is fully adaptive and deadlock free.*

Proof: All the possible minimal paths are allowed in the routing function R, so it is fully adaptive. The formal proof of deadlock-free property can be found in (Gravano *et al.*, 1994). Here we give an intuitive explanation. There is no closed cycle within the star channels because they are defined by the *dimension-order* routing as in Section 4.1. A star-channel never waits for a particular non-star channel indefinitely because there always exists a star-channel which can be used in the next step on one of the possible shortest paths. Similarly, a non-star channel also never waits for another non-star channel indefinitely. Therefore, no unlimited waiting for non-star channels will occur under any condition. This implies that based on an acyclic star-channel dependency, the non-star channels will not introduce indefinite waiting cycles. Thus it is deadlock free. □

There are cycles in the direct channel dependency (Definition 4) of the star-channel routing algorithm. The star-channel routing algorithm is a special case of the *extended* channel dependency (Duato, 1993). The key point is to build an adaptive routing algorithm based on a deadlock-free routing with acyclic channel dependency. We will discuss a more formal construction and properties of the *extended* channel dependency in the next section. The most important property of the star-channel routing is that although a packet can use a non-star channel at any time, it never waits for a non-star channel indefinitely and there is always a star channel chosen by the routing function. If a packet is directed to a non-star channel, it must be handled in finite time in the non-star channel buffer in order to prevent deadlock. Therefore, the non-star channel buffer must be empty before it can accept any new packet, otherwise cyclic waiting may occur and packets may sit in the middle of queues forever. This more restricted rule is applied to non-star channels but not star channels in order to avoid deadlock in the star-channel routing algorithm.

4.2.4 Extended Channel Dependency

Theorem 2 states that if there is no cycle in the direct channel dependency graph, then deadlock will not occur. This is a sufficient condition for deadlock-free data routing. In this section, we will introduce the concept of the *extended* channel dependency graph which is based on both *direct* and *indirect* channel dependency to obtain a more relaxed deadlock-free condition (Duato, 1993).

Definition 4 describes direct channel dependency. Before we give the definition of indirect channel dependency, we will define the routing subfunction, then define the extended channel dependency graph and give a theorem for deadlock-free data routing (Duato, 1993).

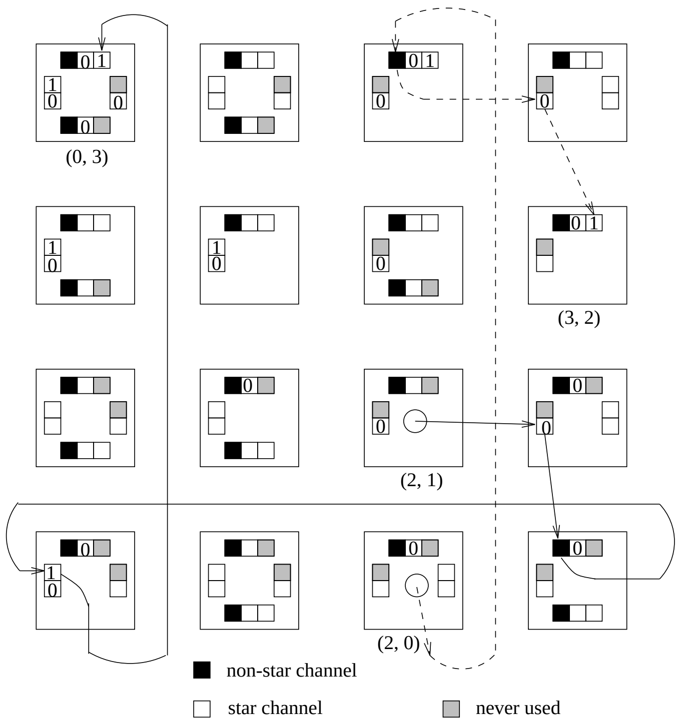


Figure 4.9: Examples of Star-Channel routing in a 2-D torus. Packets are routed from (2,0) to (3,2) and from (2,1) to (0,3).

Definition 6 *Routing Subfunction*

A routing subsection R_1 for a given routing function R and a channel subset $C_1 \subset C$ is defined as

$$R_1 : N \times N \to P(C_1)$$
$$R_1(x, y) = R(x, y) \cap C_1 \quad \forall x, y \in N$$

Definition 7 *Indirect Channel Dependency*

For a given network G=(N, C), a routing function R, a subset channel $C_1 \subset C$ which defines a routing subfunction R_1, and a pair of channels c_i, $c_j \in C_1$, there is an indirect dependency from c_i to c_j if and only if

$$\exists \bar{c}_1, \bar{c}_2, \ldots, \bar{c}_k \in \bar{C} = C - C_1 \mathit{such\ that}$$

$$\begin{cases} c_i \in R_1(s_i, n) \\ \bar{c}_1 \in R(d_i, n) \\ \bar{c}_{m+1} \in R(\bar{d}_m, n) \quad m = 1, \ldots, k-1 \\ c_j \in R_1(\bar{d}_k, n) \end{cases}$$

where s_i and d_i are the source and destination nodes of link c_i, respectively, and n is a node in N.

We can construct a path from node s_i to d_j for a packet destined to node n. Between s_i and d_j, c_i and c_j are the first and last channels and the only ones belonging to C_1. All other channels used between them belong to $\bar{C} = C - C_1$ (Fig. 4.10). Therefore, c_j will be used by a packet after using c_i and other channels in $\bar{C}$. So there is an *indirect* dependency of channel c_i and c_j.

Based on the definition of *indirect* channel dependency, we may define the *extended* channel dependency graph which has increased freedom of adaptive routing while still guaranteeing that deadlock will not occur.

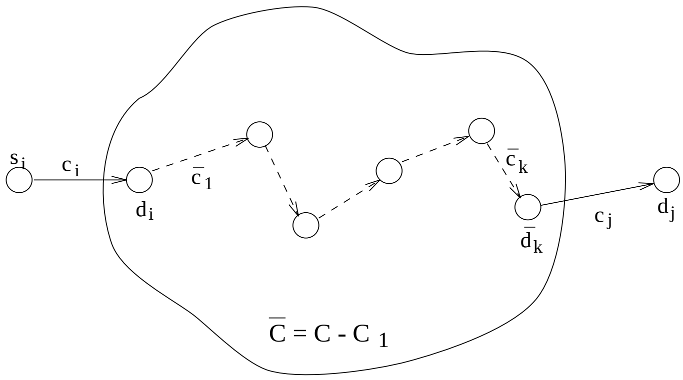


Figure 4.10: Indirect channel dependency from c_i to c_j. Solid lines $\in C_1$, and dashed lines $\in \bar{C} = C - C_1$

Definition 8 *Extended Channel Dependency Graph*

For a given network and routing subfunction R_1 of a routing function R, the extended channel dependency graph $D_E = (C_1, E_E)$ is defined as: the vertices of D_E are the set of channels C_1 which defines R_1, and the edges E_E are the pairs of channels $(c_i, c_j) \in C_1$ such that there is either a direct or indirect dependency from c_i to c_j.

A very important theorem proven by Duato (1993) shows that we may have a more relaxed rule for deadlock-free data routing.

Theorem 7 *A routing function R for a given network is deadlock-free if there exists a channel subset $C_1 \subset C$ that defines a routing subfunction R_1 which is connected such that there is no cycle in the extended channel dependency graph D_E.*

Proof: (Duato, 1993)

The key point behind the *extended* channel dependency graph is the routing subfunction R_1. We may allow cycles in the channel dependency graph in C but still stay deadlock-free as long as there is no cycle in the *extended* channel dependency graph in C_1 which defines the routing subfunction R_1. So C_1 channels are analogous to the core channels which are deadlock-free as a result of using routing function R_1. The remainder of channels $\bar{C} = C - C_1$ give more alternative paths and adaptivity while not introducing cycles in the *extended* channel dependency graph. As mentioned in Section 4.2.3, the star-channel algorithm is an example of an *extended* channel dependency graph where the star channels and the dimension-order routing function are C_1 and R_1, respectively. The non-star channels belonging to $\bar{C} = C - C_1$

are used to add adaptivity on top of the star channels. We may check that there are no cycles in the *extended* channel dependency graph of the star channels, guaranteeing that it is deadlock-free. The other very important rule is that there must be no indefinite waiting for the channels in $\bar{C}$ in order to make sure that no waiting cycles will occur in any channel. Thus the buffers of channels in $\bar{C}$ have to be empty before they accept new packets.

4.3 Partially Adaptive Routing

In a partially adaptive routing algorithm, the routing freedom is restricted to a subset of possible paths from a source to a destination. By this restriction, deadlock avoidance is easier and less expensive to implement in hardware.

4.3.1 Planar Algorithm

Restricting the routing freedom to a few dimensions at a time can reduce the requirement for virtual channels for deadlock-free routing. In the planar routing algorithm (Chien and Kim, 1992), a packet is routed adaptively in a series of two-dimensional planes. For example, a packet can be routed adaptively between dimension 0 and 1 at the beginning. After it reaches the destination's position in dimension 0, it can be routed adaptively between dimension 1 and 2, and so on.

We may define an adaptive plane A_i as a collection of some virtual channels, and divide A_i into two virtual networks A_i+ and A_i-:

$$\begin{aligned} A_i+ &= c_{i,+,2,l} + c_{i+1,\pm,0,l} \\ A_i- &= c_{i,-,2,l} + c_{i+1,\pm,1,l} \end{aligned} \tag{4.3}$$

where $c_{dim,dir,vc,l}$ labels a set of virtual channels in dimension dim with specified direction dir, virtual channel ID (vc), and level l. The level l indicates whether the wrap-around connection in this dimension has been taken or not ($l = 0$ or 1).

A packet is routed from plane $A_0, A_1, \ldots,$ to A_{n-2}. Within plane A_i, a packet is routed adaptively in the virtual network A_i+ or A_i- depending on its direction of dimension i. In each adaptive plane, the packet completes its routing path in at least one dimension, then moves to the next adaptive plane until reaches its destination. If, in plane A_i, the packet completes dimension $i+1$ first, it will stay in A_i and travel in dimension i until the distance in dimension i reduces to zero, then it will move to A_{i+2} directly.

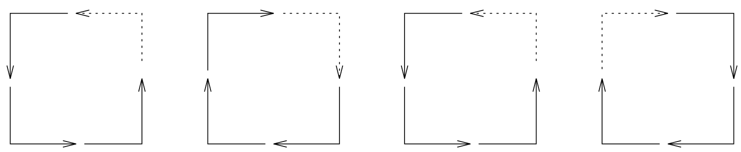
There is no cycle within each adaptive plane because we divide it into two separate virtual networks as we do in the fully adaptive virtual network routing algorithm for a 2-D torus (Section 4.2.1). There is also no cycle between different adaptive planes A_i and A_j because of the limitation of routing dimensions. So the planar algorithm is deadlock-free.

Because of the constraint of routing freedom to a 2-D plane, the requirement of virtual channels for deadlock-free routing is constant and independent of network dimension n. From Eq. 4.3, for a particular port (given *dim* and *dir*), $vc = 1 \sim 3$ and $l = 0$ or 1. So for any size of k-ary n-cubes, each port needs 6 virtual channels to avoid deadlock.

4.3.2 The Turn Model

The turn model has been proposed to avoid deadlock by prohibiting some turns to break all possible cycles in the channel dependency without adding virtual channels (Glass and Ni, 1992). The basic idea is to prevent cycles by limiting the number of turns to the smallest possible number while remaining as adaptive as possible. In fact, the dimension-order routing algorithm is a special case of the turn model where the turns from a higher dimension to a lower dimension are not allowed. Obviously, the dimension-order routing algorithm prohibits more than necessary turns and loses its adaptivity. Different combinations of turns might be eliminated in order to break cycles. For example, for a 2-D mesh, Fig. 4.11 shows two different possible ways to break cyclic dependency. In the *negative-first* routing, a negative direction channel can be followed adaptively by either a negative or positive direction channel in any dimension, but a positive direction channel is not allowed to be followed by any negative direction channel. Thus by prohibiting one quarter of all possible turns, we can prevent deadlock in an n dimensional mesh without wrap-around connections.

Based on which turns are prohibited, there is a different degree of adaptivity for different pairs of sources and destinations. For example, in the *negative-first* routing on a 2-D mesh, if the destination is at the right-upper side of the source ($x_d > x_s$, $y_d > y_s$), then it is fully adaptive from the source to destination (Fig. 4.12 (a)). On the other hand, if the destination is at the left-upper side of the source ($x_d < x_s$, $y_d > y_s$), then it is strictly deterministic if only the shortest path can be taken, or partially adaptive if non-minimal path is allowed (Fig. 4.12 (b)). In Fig. 4.12, path (c) is not allowed because it violates the *negative-first* principle.



(a) Negative-first (b) North-last

Figure 4.11: Two different possible ways to break cyclic dependency in a 2-D mesh, where the dotted turns are prohibited

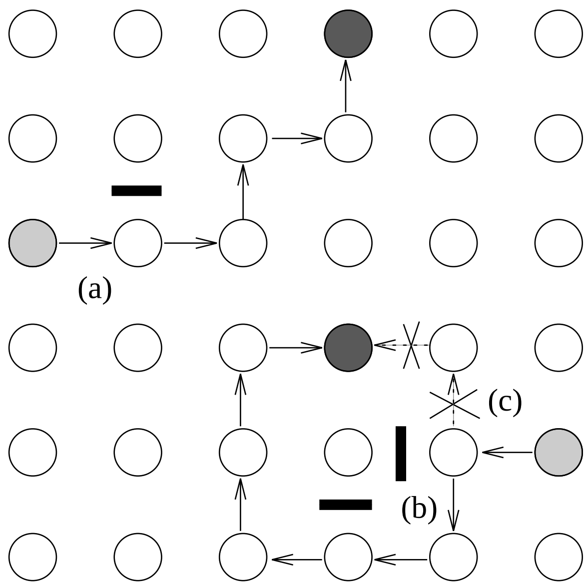


Figure 4.12: Examples of paths for the *negative-first* algorithm of the Turn model in a 2-D mesh

In an n dimensional mesh without wrap-around connections, restricting the *positive* to *negative* turns in the *negative-first* routing is sufficient to avoid deadlock. However, an n-cube with wrap-around connections requires a more complex approach to prevent deadlock. One way is to recognize the *logic* direction of a wrap-around link. For example, if the wrap-around link between node $(k-1, y)$ and $(0, y)$ is labeled as a negative direction channel, then we can apply the *negative-first* routing as described above. This approach, however, will reduce adaptivity and utilization of those wrap-around channels.

4.4 Randomized Routing

In randomized routing no particular virtual channel assignments or constraints are imposed on the routing paths. Such schemes select randomly an available virtual channel following a shortest path if such a channel is available. A "derouting" scheme[2] is used when a packet is blocked. If a packet is blocked and waiting in the same node for more than some number of cycles w, then this packet is derouted with some probability (Lu *et al.*, 1993a; Konstantinidou and Snyder, 1991). "Derouting" chooses a direction which may increase the distance to a packet's destination. The key idea here is to get around the local contention by using some non-minimal paths.

The direction chosen for derouting may be dependent on the relative position between the current node and the packet's destination: if the current node and destination are on different rows and columns, then one of the two backward paths is selected (Fig. 4.13(a)); if the current node and destination are on the same row or column, then one of the perpendicular paths to the normal path is selected (Fig. 4.13(b)) (Lu *et al.*, 1993a). Derouting has been shown to be deadlock-free for store-and-forward and virtual cut-through (Konstantinidou and Snyder, 1994). For wormhole routing, because the flits are blocked in place, a packet is possibly blocked by itself, and deadlock may occur. Therefore multi-flit buffers are necessary to prevent this situation, and we should choose the buffer size large enough to collect packets when they are blocked. In fact, for the current generation of router design, a large buffer space often is used to store at least one entire packet in order to improve performance when traffic contention occurs. We consider *virtual cut-through* flow control for the randomized data routing in the rest of this section.

[2] "Derouting" is also known as "misrouting", "non-minimal routing", and "hot-potato routing" in some literature.

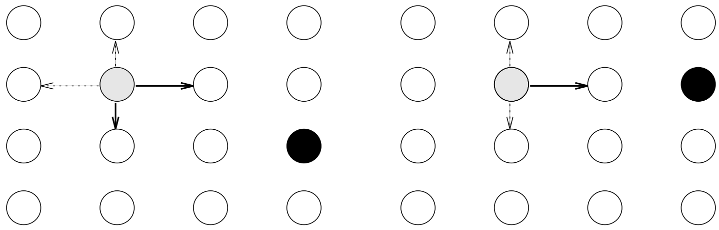


Figure 4.13: Derouting in a 2-D array when a packet is blocked. The shadow circle is the current node and the black circle is the packet's destination. Solid lines are the normal paths, and dotted lines are the derouting paths.

Theorem 8 *If the traffic contention is uniformly distributed in a wrap-around connected $N \times N$ torus network, and the blocking probability $p < 1$, then the derouting algorithm is livelock-free with probability one.*

Proof: We define the blocking probability p as the probability that an input port has no empty buffers, *i.e.*, p is the probability that a node refuses to receive new packets. We will prove livelock-free property in a 2-D torus here. It can be generalized to the n dimensional case. Without loss of generality, we set the source of a packet to be (i, j), and the destination to be (0,0). We define $u(i, j)$ as the probability of successful delivery from node (i, j) to (0,0). Because of symmetry, $u(i, j)$=$u(j, i)$, and $u(i, j)$=$u(i, N - j)$.

At node $(i, 0)$, if node $(i - 1, 0)$ has no available buffers, then the algorithm selects $(i, 1)$ or $(i, N - 1)$ as the next node on the path to the destination. If both $(i, 1)$ and $(i, N - 1)$ are available, one is selected at random. If only one of these is available, then there is only one possibility. If both have full buffers, then this packet must be delayed at $(i, 0)$ for another cycle. At node (i, j), $i, j \geq 1$, the next node along the shortest path is $(i - 1, j)$ or $(i, j - 1)$; derouting will choose $(i + 1, j)$ or $(i, j + 1)$ in case of blocking. From these considerations we can write the successful probability equations as

$$u(i, 0) = (1 - p)u(i - 1, 0) + p(1 - p^2)u(i, 1) + p^3 u(i, 0) \tag{4.4}$$

$$
\begin{aligned}
u(i,j) &= \frac{(1-p^2)}{2}[u(i-1,j)+u(i,j-1)] \\
&\quad + \frac{p^2(1-p^2)}{2}[u(i+1,j)+u(i,j+1)] \\
&\quad + p^4 u(i,j)
\end{aligned} \tag{4.5}
$$

There is a boundary condition: $u(0,0)=1$ because (0,0) is the destination. Other boundary conditions come from the wrap-around connections. For N even, $n=N/2$, we have

$$u(n+1,n+1) = u(n-1,n-1) \tag{4.6}$$

$$u(i,n+1) = u(i,n-1) \quad \text{for } i \leq n \tag{4.7}$$

$$u(n+1,j) = u(n-1,j) \quad \text{for } j \leq n \tag{4.8}$$

For N odd, $n=\lfloor N/2 \rfloor$, we have

$$u(n+1,n+1) = u(n,n) \tag{4.9}$$

$$u(i,n+1) = u(i,n) \quad \text{for } i \leq n \tag{4.10}$$

$$u(n+1,j) = u(n,j) \quad \text{for } j \leq n \tag{4.11}$$

Using Eq. 4.5 and the above boundary conditions, we can show that for $n=N/2$ (when N is even) or $n=\lfloor N/2 \rfloor$ (when N is odd),

$$u(n,n) = u(n,n-1) \quad \text{if } p \neq 1 \tag{4.12}$$

By two-dimensional mathematical induction, it is easy to show that $u(n,n) = u(n,n-1) = u(n-1,n) = \cdots = u(i,j) = \cdots = u(1,1) = u(1,0) = u(0,1) = u(0,0)$. Because $u(0,0)=1$, then

$$u(i,j) = 1 \quad \forall i,j \tag{4.13}$$

which means the successful probability for all nodes is equal to one. Therefore, the derouting algorithm can finish the data routing with probability one, *i.e.*, with livelock-free routing. □

A similar method works to calculate the latency of derouting in a 2-D torus. We define the average latency from node (i,j) to (0,0) as $l(i,j)$. By symmetry, $l(i,j) = l(j,i)$. Again,

p is the blocking probability of a node, and q is the average queue length in each node. Using the same argument as for $u(i,j)$ above, we obtain the following equations for $l(i,j)$,

$$
\begin{aligned}
l(1,0) &= 1 + (1-p)l(0,0) + p(1-p^2)[l(1,1)+q] \\
&\quad + p^3 l(1,0) && (4.14) \\
l(i,0) &= 1 + (1-p)[l(i-1,0)+q] \\
&\quad + p(1-p^2)[l(i,1)+q] + p^3 l(i,0) \quad \forall i > 1 && (4.15) \\
l(i,j) &= 1 + \frac{(1-p^2)}{2}[l(i-1,j)+l(i,j-1)+2q] \\
&\quad + \frac{p^2(1-p^2)}{2}[l(i+1,j)+l(i,j+1)+2q] \\
&\quad + p^4 l(i,j) \quad \forall i,j \geq 1 && (4.16)
\end{aligned}
$$

The boundary conditions for $l(i,j)$ are the same as $u(i,j)$ (Eq. 4.6 to Eq. 4.11), and $l(0,0)$=0.

When a packet enters a node and there are q packets waiting in the queue, then this packet will be delayed at least q cycles. This delay is caused by conflicts with other traffic, not by derouting. Therefore, if we want to measure the extra latency induced by derouting, we have to set $q = 0$. In this case, each packet can be handled immediately whenever it enters the nodes, so the delay is caused by the longer distance due to derouting.

Fig. 4.14 shows the latency penalty due to derouting in a 16×16 2-D torus. The latency is the average worst case where the distance between sources destinations is 16. It shows that the latency remains very flat and is close to 16 time units until the blocking probability is considerably large ($p > 0.1$).

4.5 Multicast Routing

All the routing algorithms we described before are unicast routing where each packet has only one destination. Multicasting is a communication process in which a message has more than one destination. Broadcasting is a special case of multicasting where a message is sent to all of the nodes in a network. An efficient multicasting algorithm should be able to deliver messages with a short latency without increasing the network load significantly. There are three methods to implement multicasting. These employ *unicast-based*, *tree-based*, and *path-based* multicasting algorithms.

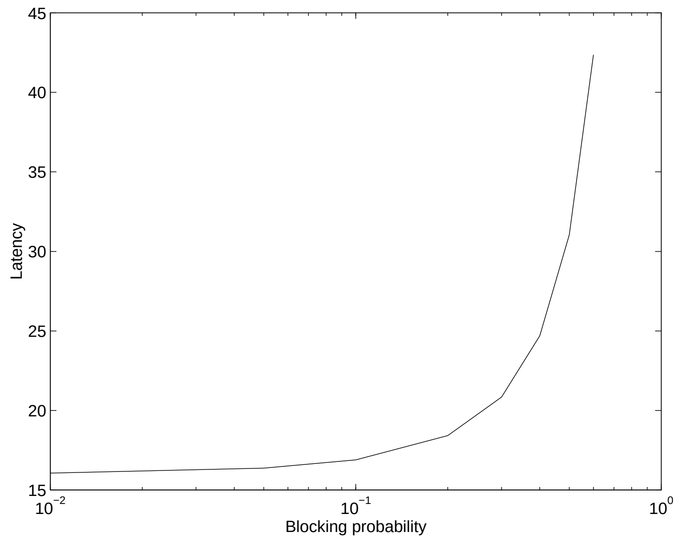


Figure 4.14: Latency penalty due to derouting in a 16 $\times$ 16 2-D torus

4.5.1 Unicast-based Multicasting

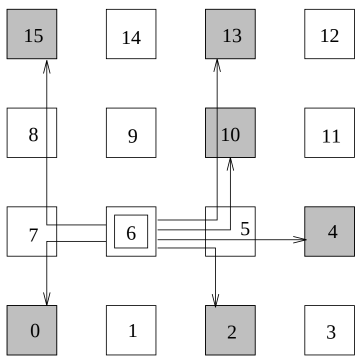
Most current routers do not have a built-in capability to support multicasting. With such routers, it is necessary to use multiple unicasting to implement multicasting in software. When a multicast request is issued by a host, multiple copies of the messages are generated, *i.e.*, one for each destination. The messages are sent to the destinations sequentially according to a unicasting algorithm such as described in the previous sections.

Fig. 4.15 (a) shows an example of the unicast-based multicasting. The unicast-based multicasting is easy to implement because it is a direct extension of a unicasting algorithm and only the source nodes need to replicate the messages. But all the packets of the same message will use network resources repeatedly, thereby increasing the network traffic load and message latency.

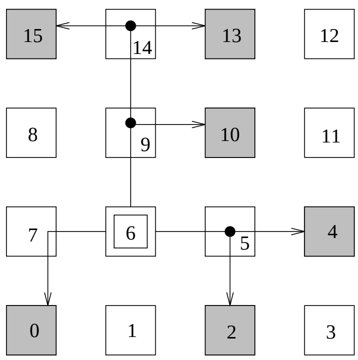
4.5.2 Tree-Based Multicasting

An alternative approach is to construct a multicasting tree that covers all destinations[3] and send the message along the paths on the tree. In tree-based multicasting, the destinations are partitioned at the source, and separate copies of the message are transmitted, one for each set of destinations or sub-trees. Packets are routed along each path. When they reach

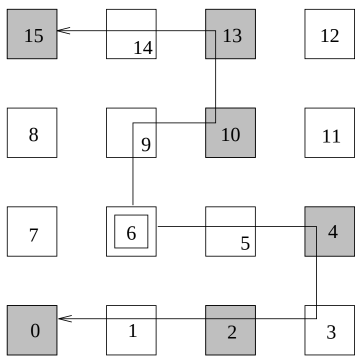
[3]The destinations can be either on the leaves or intermediate nodes of the multicast tree.



(a) Unicast-based



(b) Tree-based



(c) Path-based

Figure 4.15: Examples of different multicasting schemes. Source node is 6, destinations are [0, 2, 4, 10, 11, 15]

a branch on a sub-tree, they are replicated and each copy is forwarded to different subsets of destinations under this branch. The procedure of "replicate-forward" is repeated at every branching node of the tree until all the destinations have received their packets (Lin *et al.*, 1994). Fig. 4.15 (b) shows an example of tree-based multicasting. Because there is no storage space in the intermediate nodes on the paths, all the packets of a multicasting message on subsequent branches have to stop if any one packet which is blocked. Therefore, the blocking probability of a tree-multicasting message is much higher than a unicasting message. For example, if the blocking probability for any branch is p and there are k branches originating from the same node, then the blocking probability of the multicasting message is $1-(1-p)^k$, which is larger than p for any $k > 1$.

Deadlock may occur in tree-based multicasting even though we follow the dimension-order routing for each message. Fig. 4.16 shows a deadlock configuration in tree-based multicasting taken from Lin *et al.* (1994). In the example, a multicasting tree branches at node (1, 0) to (0, 0) and (2, 0). Another multicasting tree branches at node (2, 0) to (1, 0) and (3, 0). Neither the message entering (1, 0) nor that entering (2, 0) can move forward. The first multicast message, at (1, 0), cannot move forward because it requires the right output port at node (2, 0) which is in use. The second multicast message, at (2, 0), also cannot move forward because it requires the left output port at node (1, 0) which is occupied by the first message. Thus, there is circular waiting resulting in deadlock. A solution to prevent such blockages was proposed in (Lin *et al.*, 1994). The approach is similar to the virtual network algorithm for adaptive routing described in Section 4.2.1. A physical network is partitioned into 2^n disjoint virtual networks ($VN = \{d_0, d_1, \ldots, d_{n-1}\}$)[4]. The set of destinations is identified by the virtual network a message will take, and divided into subsets. Duplicated copies of the message are sent to each virtual network. The tree-based multicasting algorithm is applied in each virtual network to deliver the message to all the destinations in the subset.

The main disadvantage of tree-based multicasting is that the equivalent blocking probability is higher because of multiple branches on the tree. Furthermore, a packet is forwarded right after the header is received in wormhole routing, so for efficient operation the branching decision needs to be made as soon as possible. Encoding all the required branching information in the header efficiently is difficult when there are many destinations distributed in different branches. For these reasons tree-based multicasting is not suitable for wormhole routing.

[4]In Section 4.2.1, we have 2^{n-1} virtual networks and $VN = \{d_1, d_2, \ldots, d_{n-1}\}$

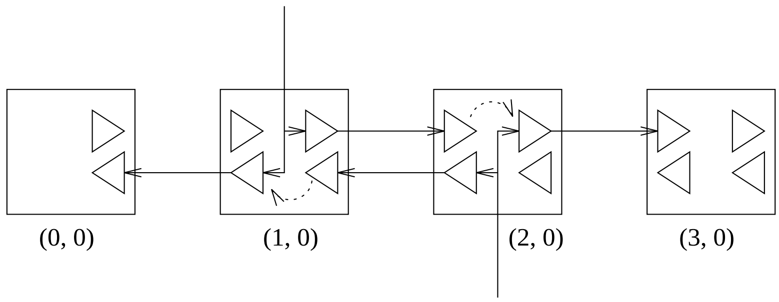


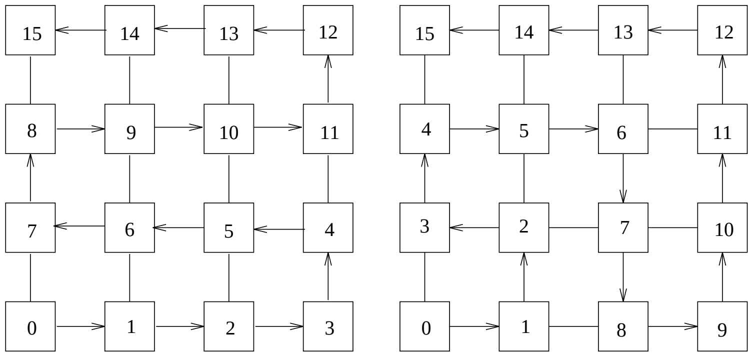
Figure 4.16: Deadlock configuration in tree-based multicasting

4.5.3 Path-Based Multicasting

In path-based multicasting, a packet is forwarded to its destinations one by one in a sequential order, so a message is never duplicated at the intermediate nodes after it departs the source node. A packet may have multiple headers: the first header indicates the first destination, the second header indicates the second destination, and so on. After the packet has reached the first destination, the first header is removed, and the remainder is both retained at the first destination and also continuously forwarded to the next destination specified by the headers. In path-based multicasting, messages usually do not travel by the shortest paths to all their destinations, but there is only one packet associated with each message in the network at any time. For example, in Fig. 4.15 (c), for the message from node 6 to nodes 0, 2, and 4, there is only one packet, and the path from node 6 to node 0 is not the shortest path. Therefore, the network load and the blocking probability of messages do not increase in the same way as tree-based multicasting.

A path-based multicasting algorithm using Hamiltonian paths was proposed by Lin *et al.* (1994). A Hamiltonian path visits every node in a graph exactly once. A label may be assigned to each node in the network according to its position on a Hamiltonian path. Fig. 4.17 shows two examples of Hamiltonian paths in a 2-D mesh. Based on the labeling of the Hamiltonian paths, it is possible to partition the network into two disjoint parts: (i) a positive network where all the links are from the nodes with smaller labels to the nodes with larger labels; and (ii) a negative network where the links are from the nodes with larger labels to the nodes with smaller labels. Fig. 4.18 shows the network partition of Fig. 4.17 (a).

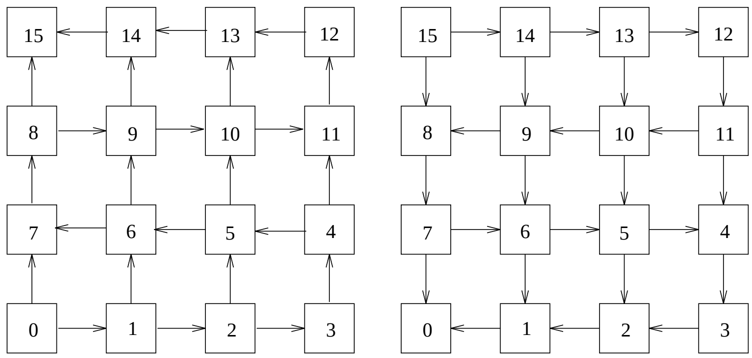
For a source and a set of destinations, the destinations can be divided into two subsets, D_H and D_L. The destinations in D_H have larger labels than that of the source, while the destinations in D_L have smaller labels than that of the source. Two copies of messages are



(a) Hamiltonian path 1 (b) Hamiltonian path 2

Figure 4.17: Two different mapping to construct Hamiltonian paths in a 2-D mesh

sent to the destinations: one is sent to the nodes in D_H using the positive network, and the other is sent to the nodes in D_L using the negative network (Fig. 4.15 (c)). The packets use the channels of the positive network in the strictly ascending order for the destinations in D_H, and those of the negative network in the strictly descending order for D_L. Therefore, there is no cyclic dependency, and deadlock-free multicasting is achieved.



(a) Positive Hamiltonian network

(b) Negative Hamiltonian network

Figure 4.18: Partition of a 2-D mesh based on the Hamiltonian path

Chapter 5

Wormhole Routing Simulation

A wormhole routing simulator was built to study different routing algorithms and some design tradeoffs, such as those among buffer size, virtual channel numbers, and channel arbitration. Throughput and latency are the main metrics for measuring performance. A good routing algorithm combined with appropriate design parameters should be able to sustain a high data throughput with low latency. In this chapter, we describe the simulator models and architectures, and some different traffic models. We also describe the results of the simulation and interpret these in terms of design issues.

5.1 Simulation Models

The wormhole network simulator can be divided into three levels: the network, the node, and the link model. The network model instantiates the network topology and interconnections, and defines individual nodes in the network. The node model defines the functions inside a router that perform data routing. The link model implements the low level data transfer, interconnection protocols, and FIFO or DAMQ buffers. In addition to the above three level models, we also have a traffic model which generates packets for the network based on the specified injection model and destination distribution. We will describe the details of each model in the following sections.

5.1.1 Network Model

The network model defines the network interconnection. Network dimension and size are declared in this level. A network configuration file contains the declaration of nodes, each port of every node, and the connection with other nodes. We can have an arbitrary network topology by specifying the interconnections in the network configuration file. The network interconnection consistency is checked automatically after reading the configuration file to make sure all the interconnections are legal and one-to-one. Two kinds of nodes can be declared: routers and hosts. The router node will be described by the node model, and the host will be described by the traffic model.

We are most interested in k-ary n-cube networks because of their generality, regularity, and simplicity (Section 2.2). A k-ary n-cube mesh network is an n-dimensional grid consisting of k^n nodes. There are k nodes in each dimension, and each node is connected to its Cartesian neighbors[1]. A torus is a mesh with wrap-around connections, *i.e.*, there is a ring in every dimension. With the physical limitation of wire density and bisection width, Dally has shown that low-dimensional cubes perform better than high-dimensional cubes in parallel processing (Section 2.2) (Dally, 1990b). Therefore, we will concentrate on the two-dimensional mesh or torus in our simulation.

5.1.2 Node Model

The node model defines the router functions and performs data routing. Fig. 5.1 shows the internal router architecture proposed and implemented in the simulator. A router is composed of header decoders, requesting units, arbiters, crossbar switches, I/O buffers, and I/O controllers. The I/O buffers and I/O controllers are described in the link model. Each function in a node is declared as a simulation module and the interface between modules is well defined. In this way, we can replace a module and implement different algorithms for a particular function.

In order to enhance the routing efficiency, each packet has the directional information in its header to indicate the direction (positive or negative) it will take in any dimension. For this we need n bits for an n-dimensional array. The directional information is encoded at the source when the packet is generated, and is never changed on the path to the destination. There are n arrival bits in the header associated with each dimension to indicate whether this dimension needs further correction or not. There are other n wrap-around bits in the header to keep track of which dimensions have taken the wrap-around connections. The wrap-around bits are critical to deadlock avoidance as described in the various routing algorithms (Chapter 4). So, a total of $3n$ bits of supplemental information are encoded in the header flit

[1] The node here contains a router and a host.

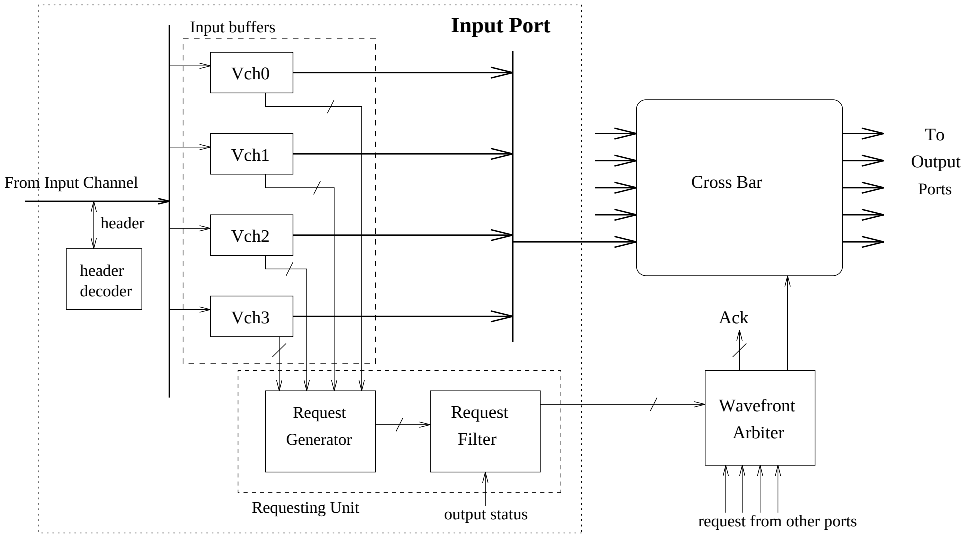


Figure 5.1: Internal router node architecture model

of a packet.

A credit feedback scheme is used for traffic congestion control. We assign a credit to each output-input virtual channel pair. The credit is used to indicate the buffer availability at the input port of the receiver and is monitored at the output port of the sender. The initial value of a credit is equal to the virtual channel buffer size. When a flit is transferred to the output port, the credit of the virtual channel it will use is decremented by one. When the receiving node reads a flit out of the corresponding input virtual channel buffers, it will send a credit back to the sender's output port. The credit for the virtual channel at the output port is incremented by one when a credit is received. Because of the propagation delay, the credit at the output port is always less than or equal to the available buffer size of the input port at the receiver. Thus with this scheme, we will not overflow an input buffer and never discard a packet.

We now describe the details of each functional block in a router.

Header Decoder The header decoder at each input port compares the coordinates of the current node and the incoming header flit's destination. We only have to check the address coordinate in the dimension from which the packet is coming. The header decoder marks the arrival bit of this dimension as "arrived" if the incoming packet's destination matches the current node in this dimension. If the input link is a wrap-around connection, the header decoder will set the wrap-around bit in this dimension for the incoming packet. If the input virtual channel is implemented as a DAMQ, the header decoder also decides the primary output direction and links the incoming flit to the proper list.

Requesting Unit There are two sub-blocks in the requesting unit: a request generator and a request filter. The request generator takes the requests from all virtual channels of the input port and selects the output port and output virtual channel for each request. If an input virtual channel is organized as a DAMQ, then there may be more than one request from a single virtual channel. If the request is issued by a header, the request generator will choose the output virtual channel based on the routing algorithm for deadlock avoidance and credit availability. If the header has passed, then the previous output virtual channel assigned for the same packet is chosen. Before the requests go to the arbiter, they go to the request filter which can block some requests. A request is blocked if (1) the chosen output virtual channel has already had another message in progress, (2) the chosen output virtual channel is full (no credit), or (3) the chosen output port buffer is full. Requests which fall into these three categories are eliminated, so they will not occupy the slot for arbitration.

Arbiter The purpose of the arbiter is to arbitrate input and output conflicts at the crossbar and to make switch utilization and throughput as high as possible. We implemented wavefront arbitration to optimize the utilization of the crossbar switch (Tamir and Chi, 1993). Fig. 5.2 shows a 5×5 wavefront arbiter. Each cell receives the request and generates the grant for the corresponding cross point in the crossbar switch. The cells are arranged in diagonal lines, *i.e.*, wavefronts. Since the requests on a wavefront are from different input ports and for different output ports, there is no conflict between the arbitration cells on the same wavefront. The order of wavefronts determines the priority of requests. The arbitration begins from the top wavefront, which has the highest priority. If a request is granted, the corresponding input and output port are disabled and no other requests can be granted for either the same input or output port. To guarantee fair arbitration, we can sort the wavefronts by the "age" of the flits issuing the requests. A more practical compromise is to put the oldest flits on the top wavefront, with other requests ordered in a random or round-robin priority.

Crossbar Switch The crossbar switch sets up physical connections from input ports to other output ports. The size of crossbar switch is $m \times m$, where m is the number of

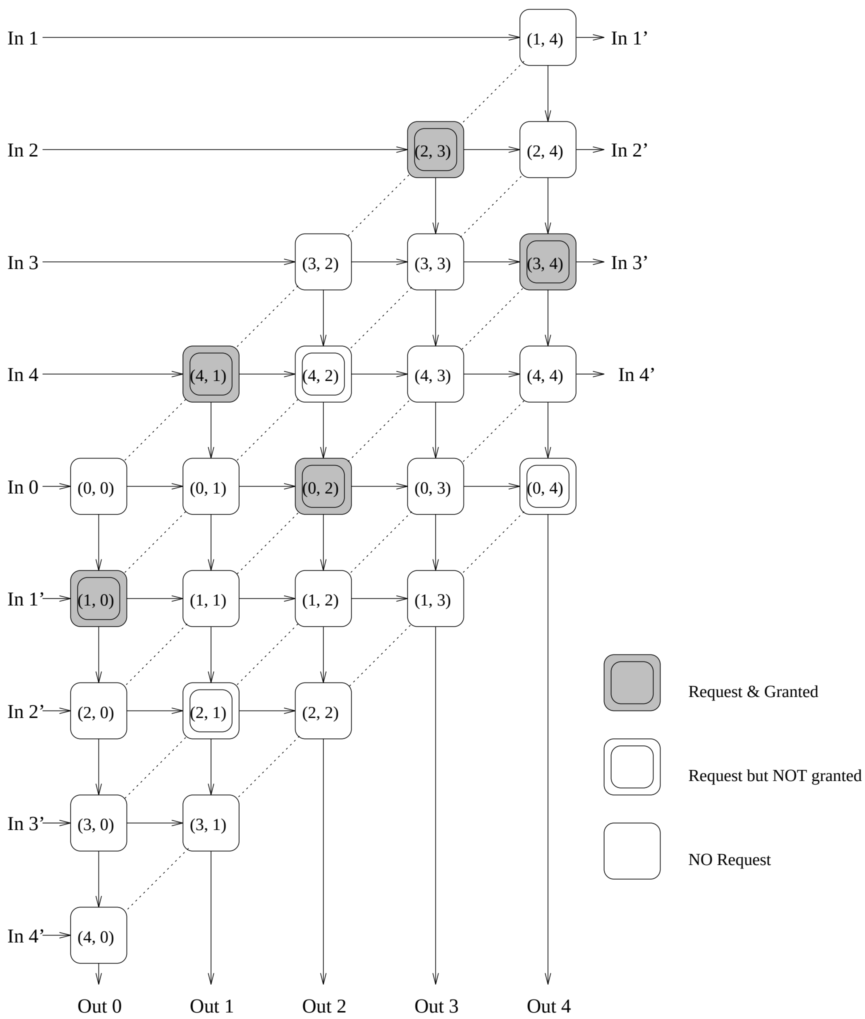


Figure 5.2: Wavefront arbitration. Each cell corresponds to a cross point in the crossbar switch.

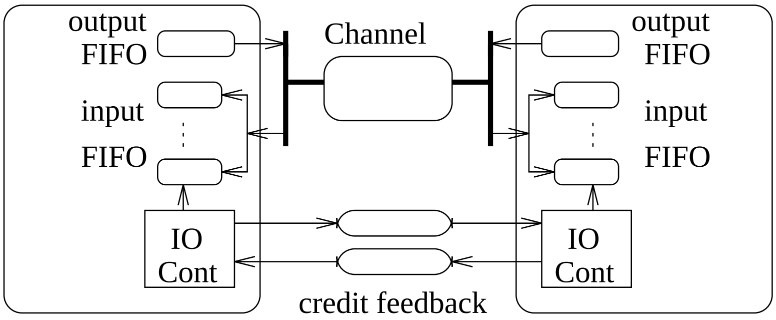


Figure 5.3: Link model

ports.

In a two-dimensional mesh or torus, each node has five I/O ports: four to the nearest neighbors and one to the host. The crossbar is a 5×5 fully-connected switch such that any input port can be connected to any other output port as long as there is no output conflict guaranteed by the arbiter.

5.1.3 Link Model

The link model includes the physical channels for data transfer, I/O controller for interconnection protocols, and I/O buffers for virtual channels (Fig. 5.3).

A physical channel is modeled as a FIFO where the sender pushes a new flit into the channel and the receiver pops a flit out of the channel. There is a delay associated with each channel. The depth of the channel FIFO model is proportional to the channel propagation delay. When the sender pushes a flit, it will add the delay to the flit's time stamp. The FIFO not-empty signal will inform the receiver to check the channel. If the flit at the head of the channel has a time stamp smaller or equal to the current time, the receiver pops out this flit and completes the transaction. We do not relay an acknowledgment back to the sender because the credit scheme described in the previous section guarantees that the buffers will not overflow and no flits will be discarded. Each flit has its own virtual channel ID. When it is popped out by the receiving node, it will be put into the input virtual channel specified by its virtual channel ID. The FIFO model of the channel is useful especially when the propagation delay is longer than the cycle time, where there may be more than one flit in the channel as a result of the transmission line effect.

The credit feedback link from a receiver back to a sender is similar to the data channel.

A credit with specified virtual channel ID is pushed into the credit feedback link. After the scheduled delay, the credit is popped out to update the output credit of the corresponding virtual channel at the sender.

The input buffers are organized as several independent lanes, or virtual channels. Each virtual channel buffer can be either a FIFO or DAMQ (Section 3.3.2) depending on the simulation conditions. The size of buffers and number of virtual channels are the simulation parameters. The output buffer is a single FIFO because the output flits are sequential on the physical channel. If the physical channel is grouped as two uni-directional channels, then a single entry latch is sufficient for the output buffer. If the physical channel is a bi-directional channel, then we need more space for the output buffer because the token-exchange may take some time and it is necessary to have the packets pass through the crossbar switch and be buffered at the output port in order to hide the latency.

5.1.4 Traffic Model

A host is connected to one of the ports in a router to inject traffic into the network. The traffic injection module emulates the host to generate packets and encode the packet headers. The injection rate is a parameter to adjust the traffic load in the network. Different injection modes can be used to generate traffic, for example, uniform injection and bursty injection. The packet length is specified in terms of number of flits including the header. Two different packet lengths can be defined and specified by some ratio to generate a mixed traffic. A flit is consumed immediately when it reaches the destination host. The injection rate and destination distribution can be specified for each host separately to model the non-uniform traffic. We may limit the maximal number of outstanding packets per host to prevent overloading the network.

5.2 Simulation Flow

Fig. 5.4 shows the simulation flow of the wormhole data routing network. The simulator has a mixed simulation scheduling scheme. Instead of the event-driven simulation, we have a global clock to control all the hosts and routers which we assume are running at the same frequency in the network. We have a bypass in each simulation step to save simulation time if we know there will be no action for some steps. In addition to the global simulation clock, we also have scheduling queues for each physical link as described in the link model. The propagation delay for each transmission is scheduled in the channel queue, and a receiving node will read the channel based on the value of the time stamp. The clock skew between nodes is simulated by adding a random synchronization delay when a flit passes across a link. The granularity of propagation delay and synchronization delay can be sub-cycle, but the scheduling is quantized to be on the cycle boundary.

Most network simulations assume it takes only a single cycle for a hop (Duato, 1993; Dally and Aoki, 1993; Boppana and Chalasani, 1993; Pifarré *et al.*, 1994; Berman *et al.*, 1992; Draper and Ghosh, 1992); this is not an accurate model for hardware implementation and cannot handle variable propagation delay. In this research, four stages of pipelining in the router are implemented as a realistic model of hardware design. The four stages are shown in Fig. 5.4: I/O Control, Header_Decoding, Request_Arbitration, and Crossbar. The fall-through latency is defined as the latency of a flit passing through a router without contention. In our simulation model, the fall-through latency is four cycles.

In the bi-directional channel configuration, it is very important for performance to hide the delay of token exchange. Since it is known whether an output port will be used at the Request_Arbitration cycle, which is two cycles earlier than the output external transmission, it is possible to issue the token request at the end of the Request_Arbitration cycle, if the chosen output port does not have the token. This hides two-cycle latency. Because of the overhead of token exchange, we do not want to have the token exchanged too often, but it is still necessary to be balanced in order to prevent starvation. A token request is acknowledged when the output port which currently has the token is idle or is sending a tail flit. This arrangement guarantees that a node can get the token in a finite time after it issues the token request, and that the token will not bounce back and forth unnecessarily, thereby reducing the overhead (Section 3.4.1).

5.3 Performance Measurement

Throughput and latency are the main metrics of network performance measurement. Throughput is a measure of the actual rate at which data are delivered to a host. The headers of packets are not included in the net data bandwidth achieved although they will consume the network bandwidth. Assume that the total channel width per port is $2w$. If the channels are bi-directional links, they have entire width $2w$ but may have the token-exchange overhead and longer propagation delay due to higher loading. If the channels are uni-directional

```
for (step=0; step < MaxStep; step++)
{
    For (all Hosts)
    {
        Generate_Packet();
        Host_Read_Channel();
    }
    For (All Routers)
    {
        Read_Channel(all input ports);
        Write_Channel(all output ports);          } I/O Control

        Crossbar_Forward();                       } Crossbar

        Requesting(all input ports);
        Arbitration();                            } Request_Arbitration

        Header_Decoder(all input ports);
        Write_Input_Buffer(all input ports);      } Header_Decoding
    }
    Cal_statistics(step);
}
```

Figure 5.4: Simulation Flow

links, each direction can only have width w, and the utilization may be lower (Section 3.4). We normalize the throughput (data bandwidth) to the factor w in our simulation results for comparison of these two channel configuration schemes.

Latency is measured based for each individual flit. Fig. 5.5 shows the delay decomposition of a single hop. The single hop delay consists of propagation delay, synchronization delay, router fall-through delay, and queuing delay (not shown). The queuing delay is due to traffic contention in the network and results in an extra delay which may be dominant for heavy traffic loads. As described in the previous section, the granularity of propagation delay and synchronization delay can be sub-cycle, and the total delay is accumulated along the path. The router fall-through delay and queuing delay are cycle-based and also accumulated in the latency calculation.

5.4 Simulation Results

The simulation was used to study network data routing for two-dimensional tori. Each simulation point has at least 10,000 cycles of operation to ensure that the statistics have reached the steady state. Data for the first 2000 cycles are discarded because they are unstable. For valid comparison, the total buffer size in a node is constant for all simulation

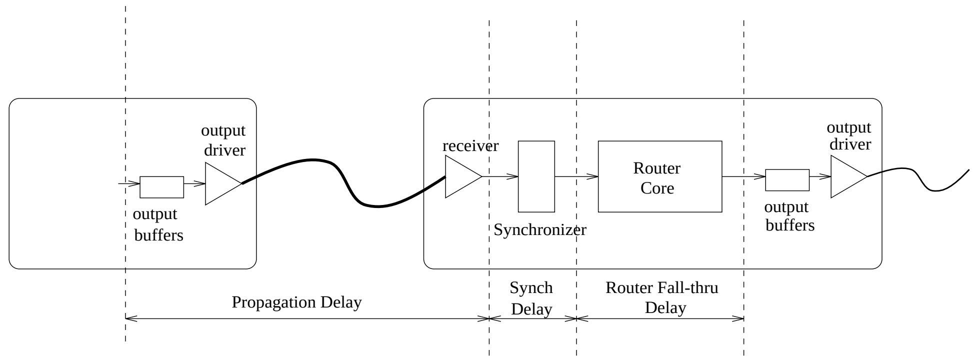


Figure 5.5: Delay components in the simulation model

conditions, *i.e.* different virtual channel numbers, routing algorithms, and so on. In the simulation, we fix the total input buffer size in a node to be $576w$, where w is the width of a uni-directional channel. For most of the simulations, the packet size is equal to $10w$ ($8w$ data plus $2w$ header). In terms of flits, the packet length is 10 and 5 flits in the uni-directional and bi-directional configuration, respectively. To prevent overloading the simulated network, we limit the maximal outstanding packets per host to be 4 in most simulations[2].

For different routing algorithms, the minimal requirements of virtual channels and buffer allocation to prevent deadlock are different (Chapter 4). For example, for the deterministic (Dimension-Order) routing, we need two levels of virtual channels in each port, while the adaptive (Dimension-Reversal or DR) algorithm needs three levels of virtual channels. Table 5.1 summarizes how we allocate the buffers for different routing algorithms on a 2-D torus, where vx_i is the number of ith level virtual channels in the x direction, for example; each virtual channel buffer size (vcb) is $32w$. It is possible to reorganize the buffers to reduce vcb and increase the number of virtual channels. For instance, vx_0=2 with vcb=$32w$ can be reorganized as vx_0=4 with vcb=$16w$. The effect of different vcb size will be shown in the simulation results.

In the simulation, we assume the clock frequency for each node is 100MHz ($10ns$ cycle time) and the propagation delay between nodes is $17ns$. Every node has its own local clock with a different phase, and the clock phase drifts by a random number within $\pm 1\%$ of the clock rate, *i.e.*, $0.1ns$/cycle. The synchronization delay is the clock difference between a sender and a receiver and takes a multiple of $10ns$.

[2]The limitation on the number of outstanding packets exists in most real parallel machines. In the real situation, we usually have request-reply type traffic patterns. Then the number of pending requests is limited

Algorithm	Buffer Allocation	Total Buffer Size
Deterministic	$vx_0 = 3,\ vx_1 = 2,\ vy_0 = 2,\ vy_1 = 2$	$2(5+4) \times 32w$
Adaptive(Virtual Net)	$vx_0 = vx_1 = vx_2 = 1,\ vy_0 = vy_1 = vy_2 = 1$ for both vn_0 and vn_1	$2(6+3) \times 32w$
Adaptive(DR)	det: $vx_0 = vx_1 = 1,\ vy_0 = vy_1 = 1$ adp: $vx_2 = 3,\ vy_2 = 2$	$2(5+4) \times 32w$
Adaptive(STAR)	$vx_0^* = vx_1^* = 2,\ vy_0^* = 2,\ vy_1^* = 1,\ vy = 2$	$2(4+5) \times 32w$

Table 5.1: Virtual channel buffer allocation for different routing algorithms in a 2-D torus.

Simulation results and discussion follow:

Effect of routing algorithms We simulated four different routing algorithms to compare their performance under different traffic loads. The four algorithms are listed in Table 5.1: There is one deterministic (Dimension-Order), and three adaptive cases, Virtual Network, Dimension Reversal (DRn)[3], and Star-Channel algorithm. Three different traffic patterns: uniform random, transpose, and hot-spot traffic, were run for each of these algorithms. The bandwidth shown in the simulation figures is the average number of bits delivered to a host per cycle divided by the channel width w.

Uniform Fig. 5.6 shows the simulation results of the uniform random traffic for different algorithms. For both uni- and bi-directional channel configurations, the deterministic algorithm is the worst among the four algorithms simulated. The Star-Channel algorithm gives the lowest latency under the same throughput (bandwidth). However, the difference between different algorithms is not very significant compared with some results reported by other researchers (Duato, 1993; Dally and Aoki, 1993; Boppana and Chalasani, 1993). The main reason for this difference is that we have a more realistic router model where the requests are arbitrated in the same way for both deterministic and adaptive routing so that the hardware complexity is about the same. A second reason is that we have four pipelining stages plus propagation delay in our model. When a flit is blocked and loses one cycle due to contention, the percentage of latency increased is smaller than that of the 1-cycle model for a single hop. So the performance difference between algorithms

by the size of request buffer which determines the maximal number of outstanding packets per host.

[3] DRn, where n is the maximum reversal number allowed

for uniform traffic is not prominent.

Transpose Transpose is a particular data pattern where node (i, j) is always sending messages to node (j, i), and vice versa. Fig. 5.7 shows the results for the transpose data pattern. In Fig. 5.7(a) (Uni-directional channels), the deterministic algorithm saturates much faster than other algorithms due to lack of flexibility in its routing paths. The Star-Channel algorithm is worse than the other adaptive algorithms in this case. The reason is the limitation of usage of non-star channels *i.e.*, the non-star channel buffer must be empty before it can accept any new packet, Section 4.2.3. The efficiency of the adaptive paths drops when we require more adaptivity, which is not very necessary in the uniform case.

For the bi-directional channel case (Fig. 5.7(b)), the deterministic algorithm actually performs best over much of the range of traffic simulated. Because of the regular data flow and the property of transpose, the deterministic algorithm always sends a packet from (i, j) to (j, j) to (j, i). So there is no channel token exchange involved in the path. The only time we may need to exchange tokens is at the source or destination where the router interfaces with the host. For adaptive routing, the regular data flow pattern is destroyed and token exchange is necessary. The overhead of token exchange makes the latency of the adaptive routing algorithms higher than that of the deterministic routing. However, since the network contention is more severe for the deterministic routing, the latency increases rapidly and the network is saturated much more quickly than the case of adaptive algorithms. So the adaptive routing achieves its advantage for heavily loaded traffic.

Hot Spot We also simulated the hot-spot effect in a network. To create a hot spot, we have every host in the network send 2% of its packets to a common destination, *i.e.*, the hot spot. In a 2-D 16×16 torus, there normally are about 0.4% of packets for each destination under uniform traffic. Thus we have five times the traffic load for the hot spot. Fig. 5.8 shows the results of hot-spot traffic. For the uni-directional case, all four algorithms saturate at about the same traffic load, which is also different from some results reported previously (Draper and Ghosh, 1992; Boppana and Chalasani, 1993). The first reason is that the bottleneck in this case is at the interface between the router and the host of the hot spot. The limited interface bandwidth will make the contention propagate quickly from the hot spot to other points in the network. This phenomenon is called "tree-saturation." Before the tree-saturation occurs, however, the deterministic algorithm is worse than other adaptive algorithms. But as is the case for uniform traffic, the difference is not very significant. When the tree-saturation sets in, most buffers are occupied by the flits destined to the hot spot, so use of different routing algorithms do not make much difference. Additionally, we have limited the number of maximal outstanding packets per host to create a more realistic host model. Since there is more contention for hot-spot messages, their "life time" is longer than normal messages. As a result, based on the packet generation probability, the percentage of hot-spot packets will become much higher than the 2% expected at the steady state. Then there are fewer normal packets which can take advantage of the adaptive routing paths. Therefore, the overall average latency is dominated by the hot-spot packets.

Effect of channel configuration Communication links can be configured either as two uni-directional channels or as a single bi-directional channel (Section 3.4). Fig. 5.9 and 5.10 show the comparison of uni- and bi-directional channels for uniform and hot-spot traffic, respectively. For lower traffic, the bi-directional channel configuration has higher latency because of token-exchange overhead. In this region, latency is dominated by the distance, and any additional delay by token exchange is significant. But when traffic load is increased, the bi-directional channel configuration performs better than the uni-directional configuration because network contention becomes dominant in the latency. For the bi-directional configuration, the effective packet length in terms of flits is half that of the uni-directional configuration, and the traffic congestion is much less in spite of the overhead of token exchange. Especially for the hot-spot traffic (Fig. 5.10), the bi-directional configuration can support much higher data bandwidth because the bottleneck at the hot-spot host is reduced due to doubling the available channel width. A similar comparison for transpose traffic is shown in Fig. 5.14. We will discuss this figure in detail in the paragraph on the effect of packet interference.

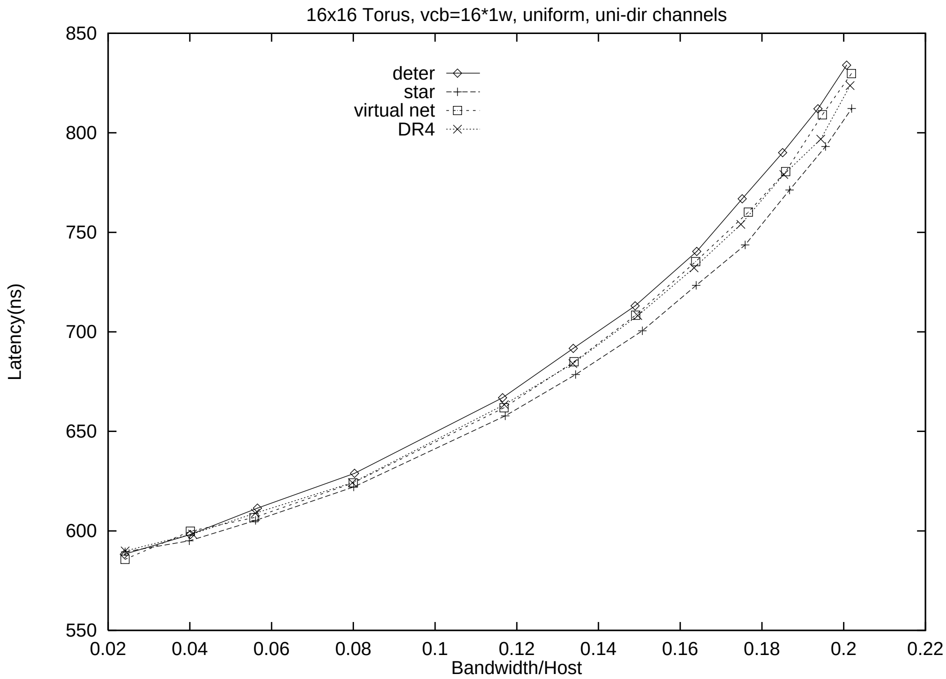
Effect of virtual channels The number of virtual channels can be increased to reduce the effect of "blocked-by-head" delays when there is traffic contention. However, we have to decrease the buffer size for each virtual channel to keep the total buffer number constant for a valid comparison (Dally, 1992). In Dally (1992), simulation results have shown that more virtual channels can increase achievable throughput. Our simulation shows similar results (Fig. 5.11), where larger buffer size means fewer virtual channels. However, when the buffer size is too small, *i.e.*, when there are too many virtual channels, the performance will be worse. The figure includes the effects of propagation delay ($17ns$) and credit feedback scheme. When the buffer size is equal to 4 flits, the round-trip link delay will make the sender stop sending more data flits because the credit is 0 and the credit update is delayed even though there is no contention. The channel will remain idle until the credit is updated. So there is an offset between the latency of vcb=4 and larger vcb at lower traffic. Some relative performance gain occurs for vcb=4 at higher traffic, but this is still not the best choice. Therefore, the virtual channel buffer size has to be large enough to hide the propagation delay of credit feedback. In the simulation, vcb=8 or 16 are the optimal buffer sizes for most traffic patterns and routing algorithms.

Effect of packet length For each packet, there is a constant overhead for header routing. When we increase the packet length, the percentage of header overhead is reduced. However, longer packets become jammed up more easily in a network. Fig. 5.12 (a) shows the network bandwidth for packet length=6, 10, and 20 flits. The total network bandwidth includes the headers of packets. For shorter packets, the latency is less at the lower network bandwidth compared with longer packets. However, shorter packets have higher header overhead. For example, the header overhead is 1/3 for length=6, but only 1/10 for length=20. Fig. 5.12 (b) shows the net data bandwidth which does not include headers. Due to header overhead, shorter packets have lower effective data bandwidth although they can achieve higher total network bandwidth. In the simulation, packet length=10 is a good compromise between traffic contention and header overhead for uni-directional channel configuration.

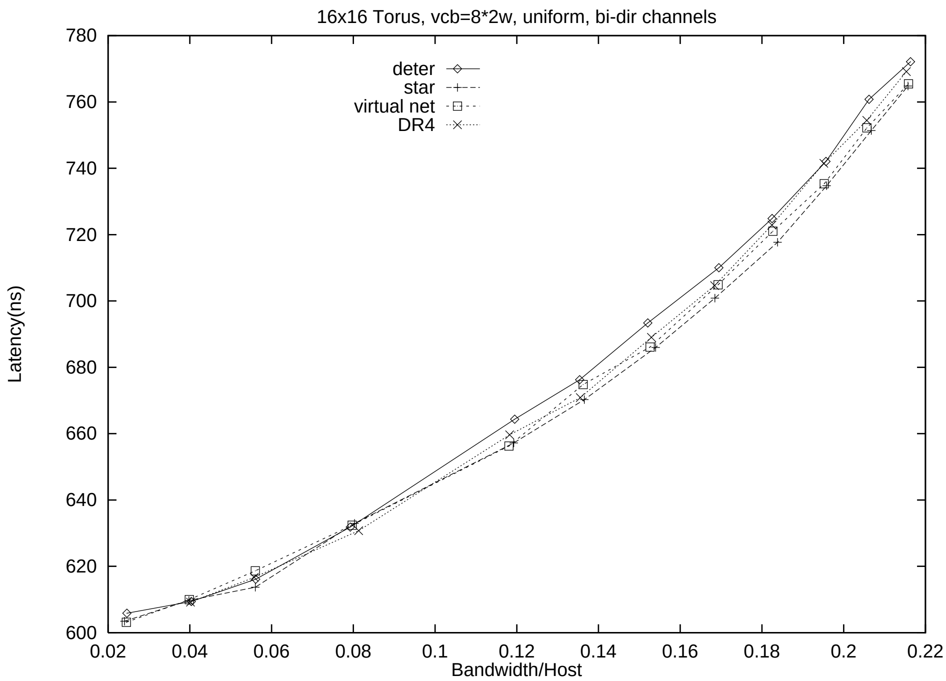
Effect of packet interference When traffic contention occurs, packets will interfere with each other. If more than one packet requests the same output port at the same time (they will request different virtual channels), only one can be granted use of the crossbar to the output port. Different arbitration schemes have been implemented and simulated. The first is that all the flits from different packets have the same privilege to issue the request and only one can be chosen by the wavefront arbiter. Then the flits from different packets may be interleaved on a physical output channel. The second scheme attempts to keep a packet together. If a packet is using some output port, then the requests by other packets will be filtered out before going to the wavefront arbiter. So the flits from the same packet will be consecutive without interruption by other packets. However, we still allow a flit to break in a packet if this flit is “older” than some threshold to prevent starvation in the second scheme.

For uniform traffic (Fig. 5.13), the performance is better when we allow a flit to break in another packet on a physical channel for the uni-directional configuration. Because all the flits compete with each other only based on their current age, it is a balanced competition and does not depend on previous arbitration results. If we try to keep a packet together, we may make some flits of other packets continue to wait even if they are older. For the bi-directional channel configuration, however, keeping a packet together is slightly better except for the deterministic algorithm.

Fig. 5.14 shows the results for transpose traffic. In this case, the benefit of keeping a packet together for the bi-directional channel configuration is even more obvious for adaptive routing. When we keep a packet together, there is no gap within a packet and the packet boundary is clear, so the overhead of token exchange is reduced. If we interleave different packets on a physical channel, a packet will spread out and idle cycles will be inserted when the packet continues to the next node (Fig. 5.15). The output port may be confused by the idle cycles and give up the token prematurely. As a result the token could be exchanged much more often, thereby increasing the overhead of token exchange.

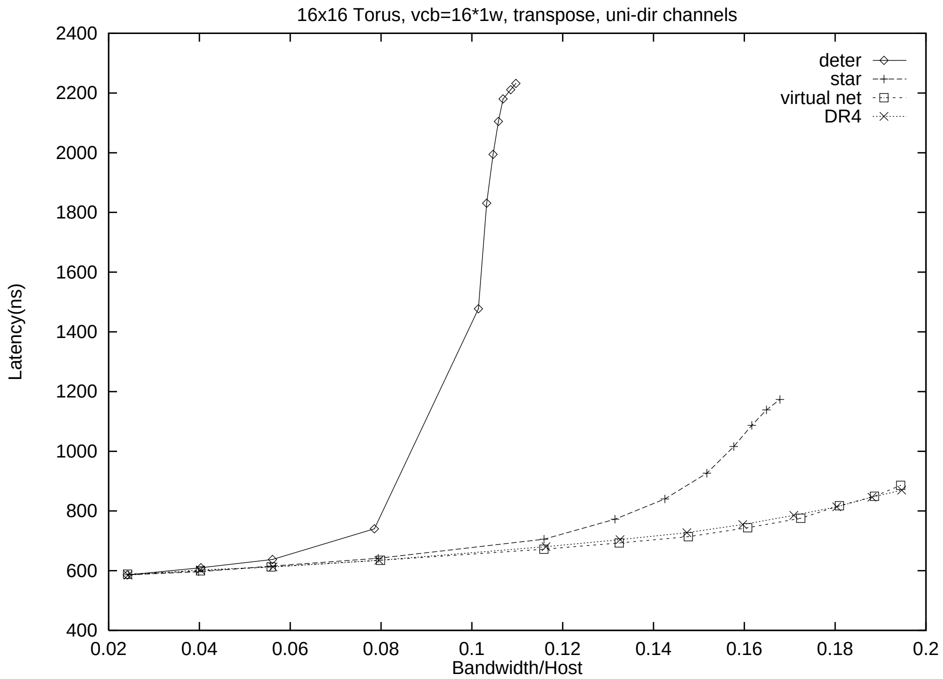


(a) Uni-directional

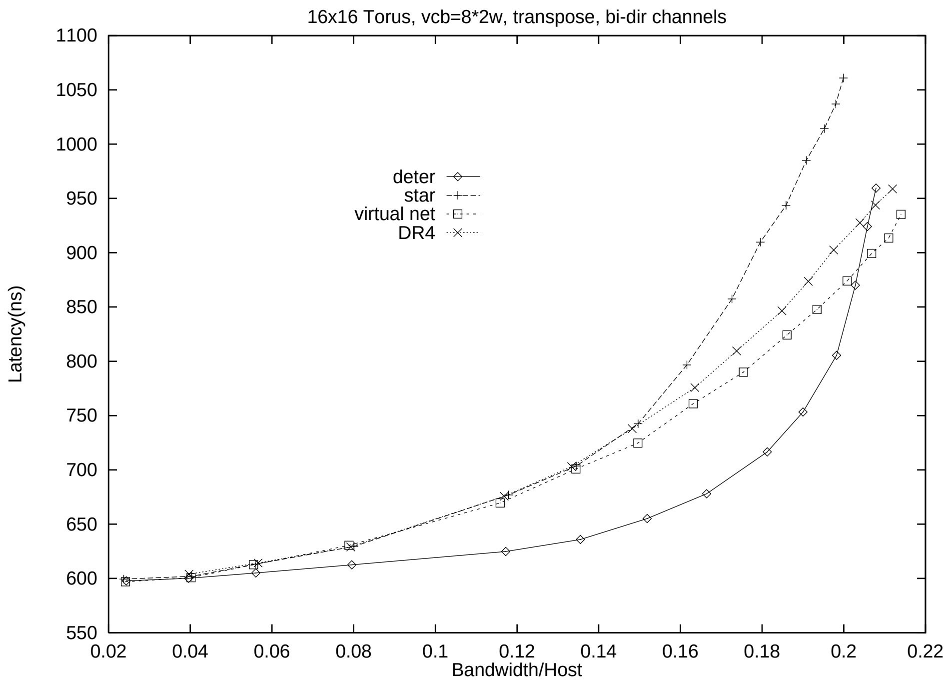


(b) Bi-directional

Figure 5.6: Latency versus throughput for different routing algorithms under uniform random traffic

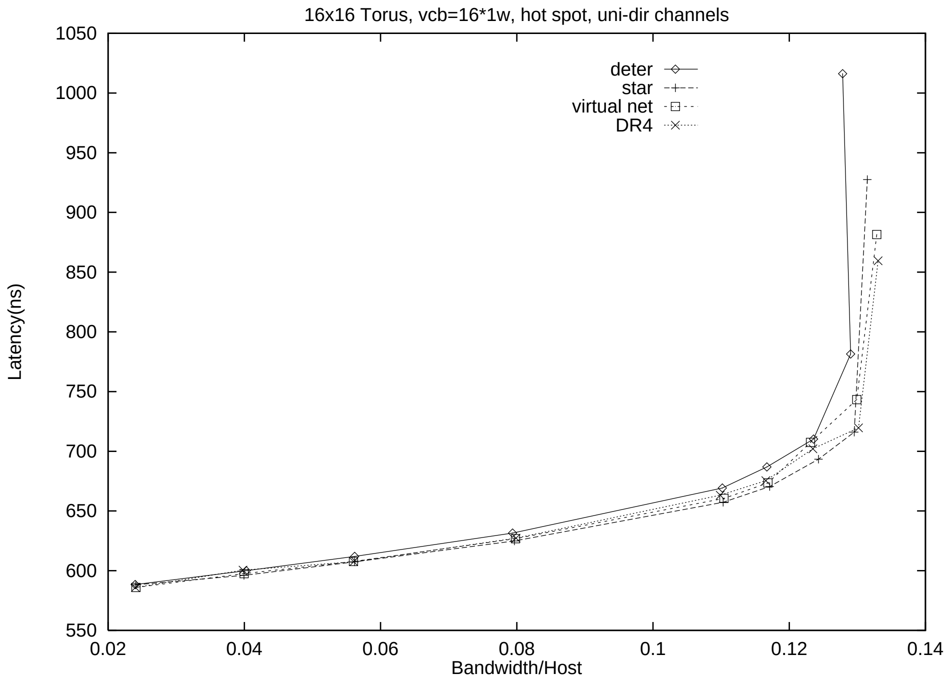


(a) Uni-directional

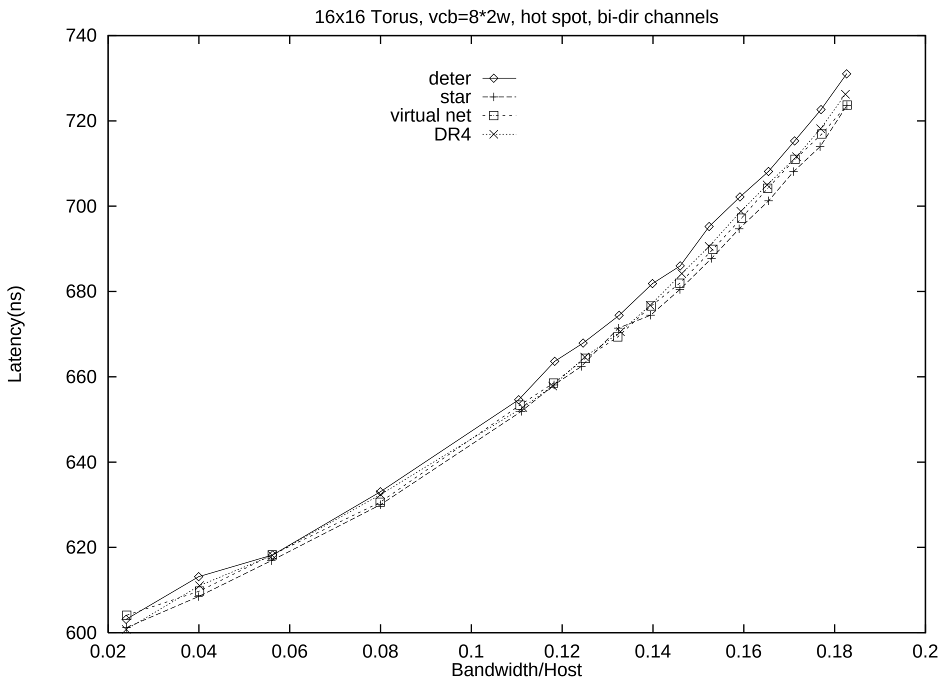


(b) Bi-directional

Figure 5.7: Latency versus throughput for different routing algorithms under transpose traffic

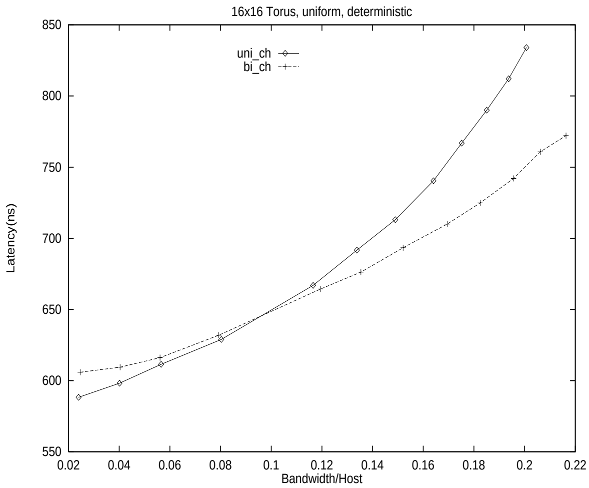


(a) Uni-directional

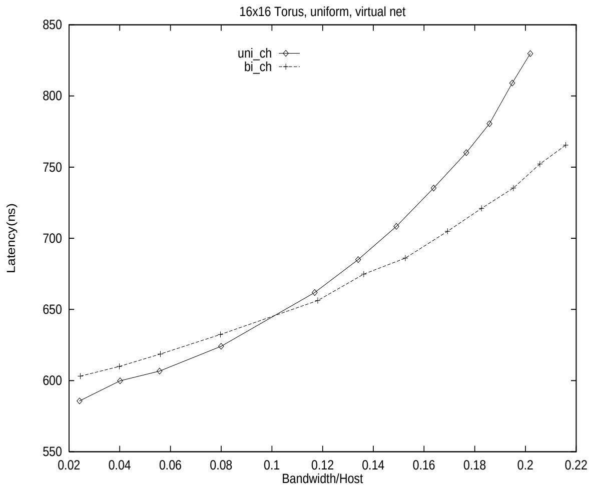


(b) Bi-directional

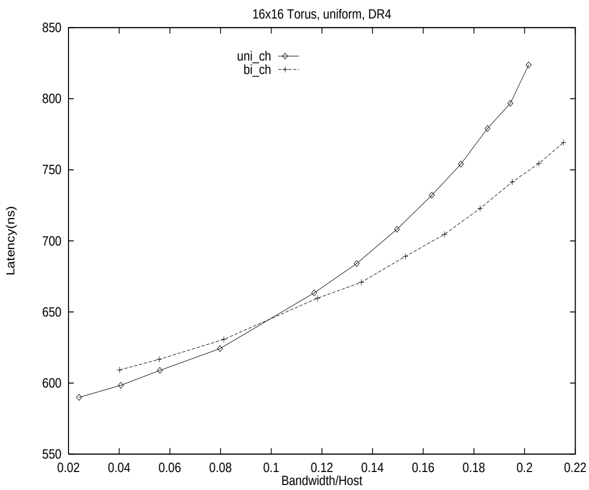
Figure 5.8: Latency versus throughput for different routing algorithms under Hot-spot traffic



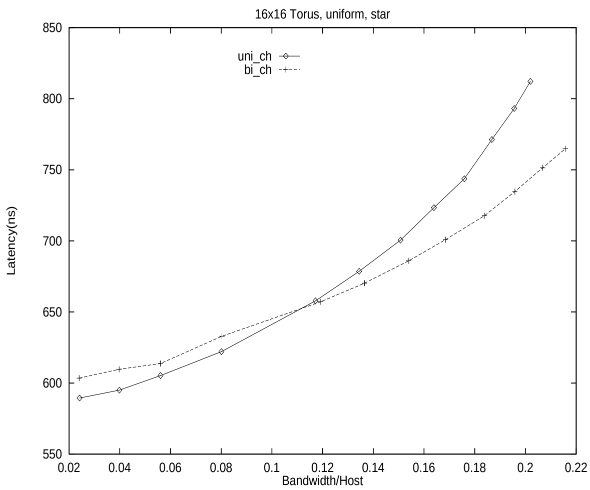
(a) Deterministic



(b) Adaptive (Virtual Networks)

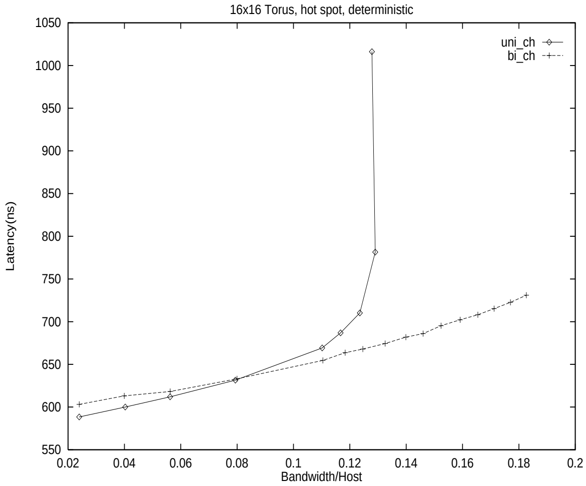


(c) Adaptive (Dimension Reversal)

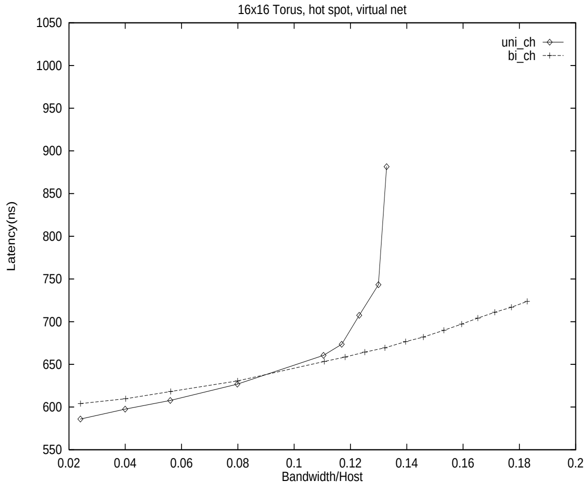


(d) Adaptive (Star Channel)

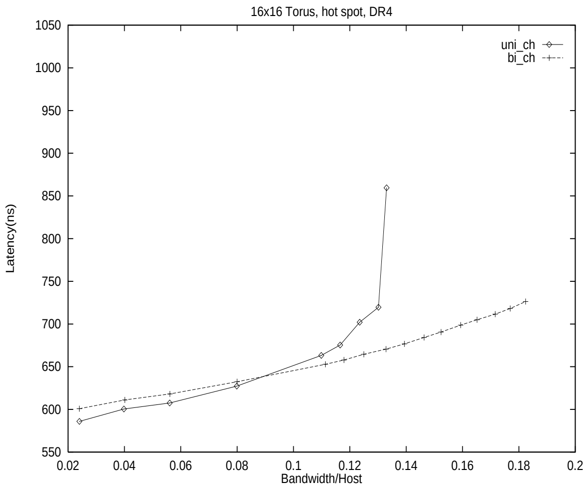
Figure 5.9: Latency versus throughput for different routing algorithms under uniform random traffic. Comparison of uni- and bi-directional channels



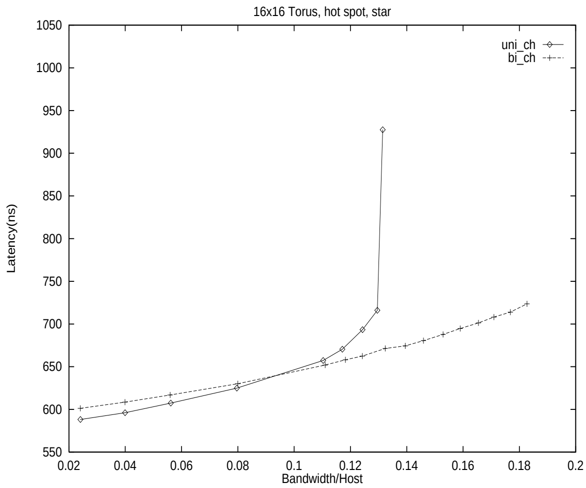
(a) Deterministic



(b) Adaptive (Virtual Networks)

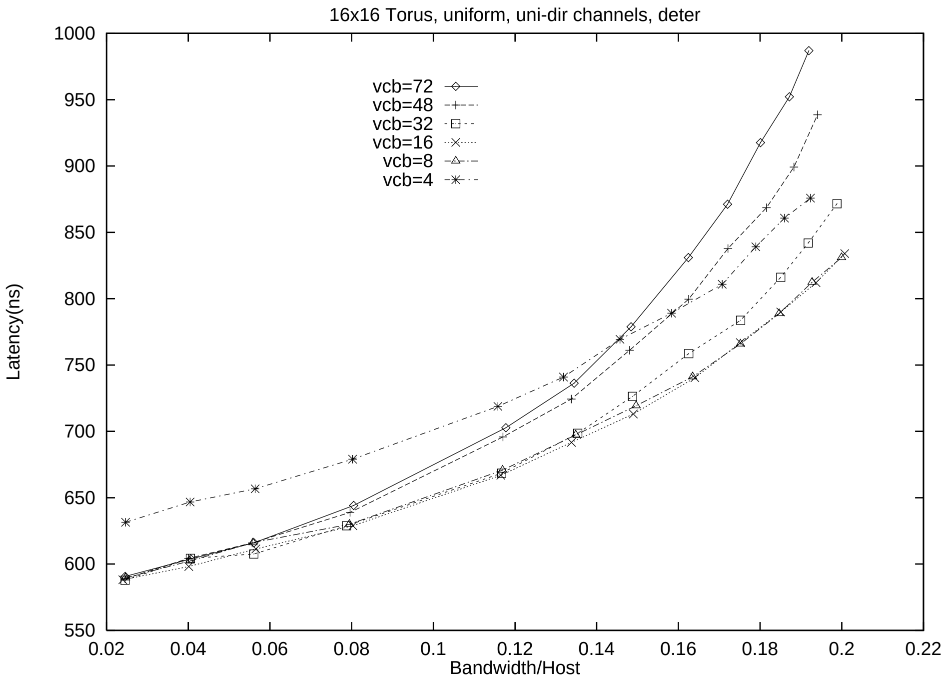


(c) Adaptive (Dimension Reversal)

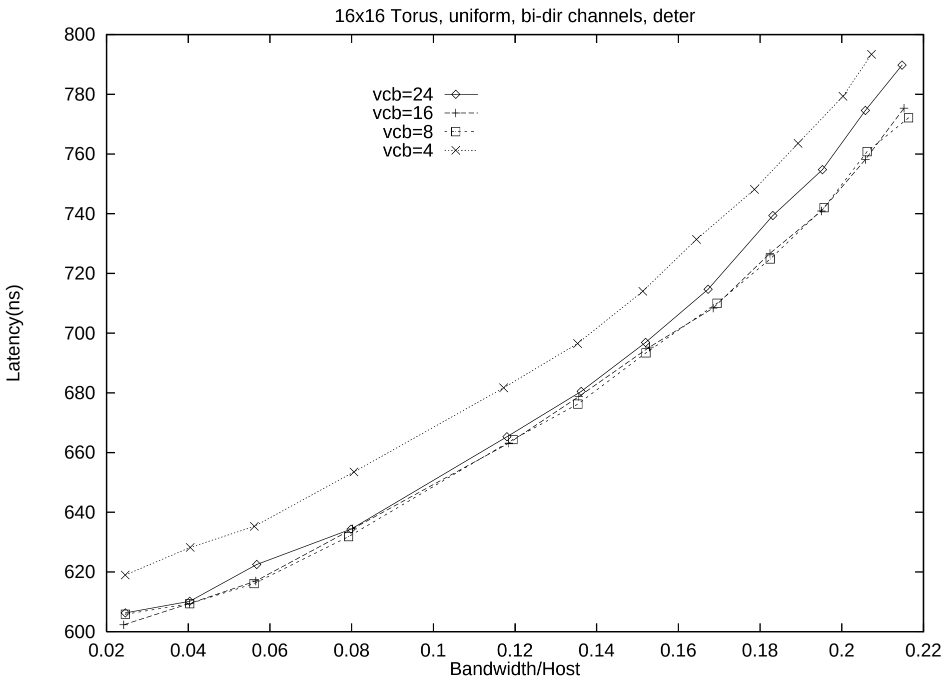


(d) Adaptive (Star Channel)

Figure 5.10: Latency versus throughput for different routing algorithms under Hot-spot traffic. Comparison of uni- and bi-directional channels

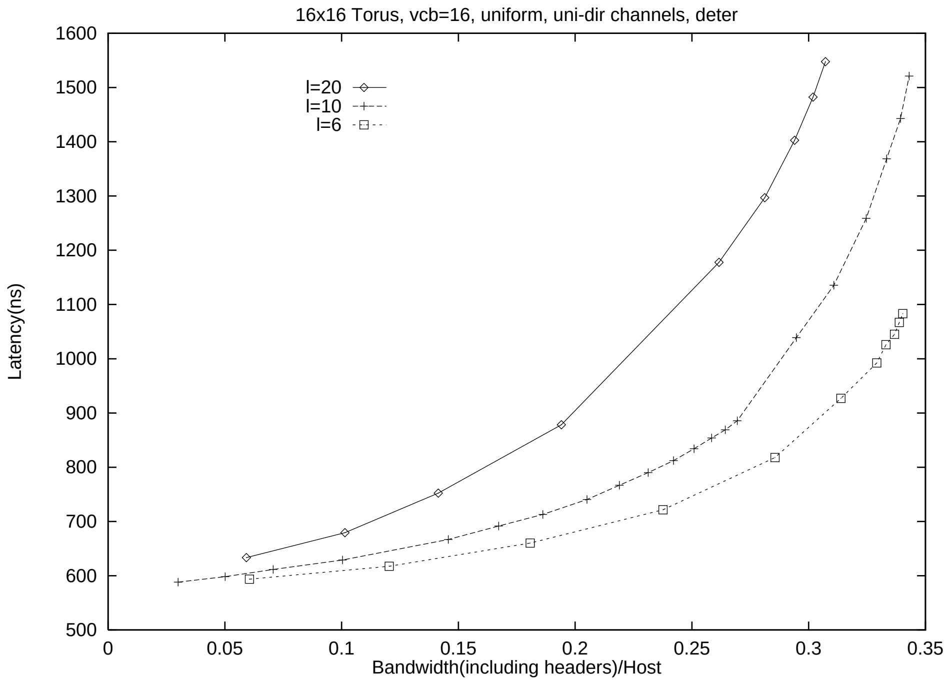


(a) Uni-directional, size unit=w

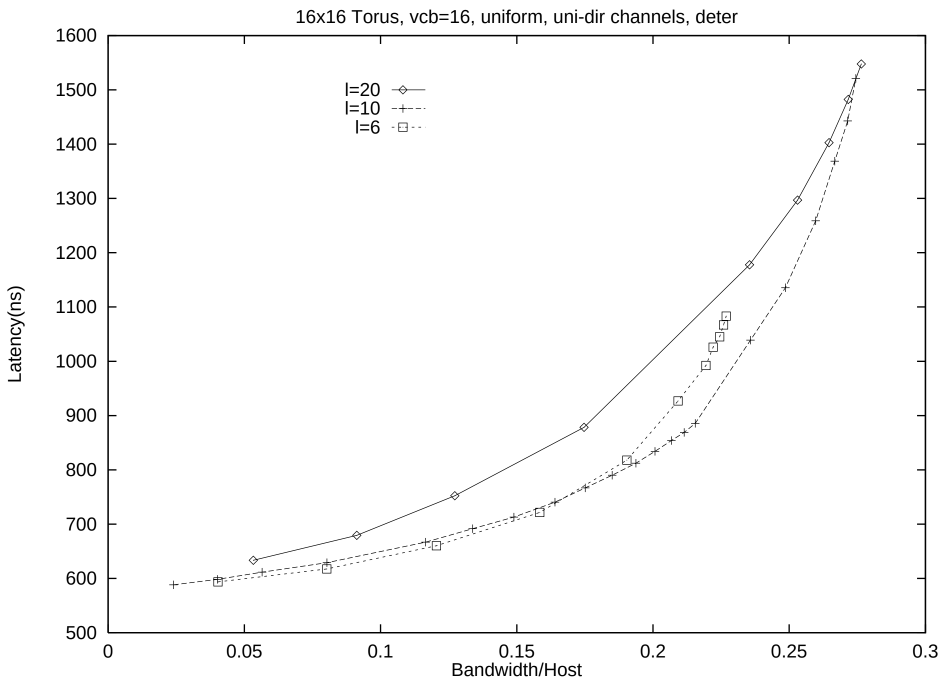


(b) Bi-directional, size unit=$2w$

Figure 5.11: Latency versus throughput for different virtual channel buffer sizes. Deterministic routing under uniform random traffic

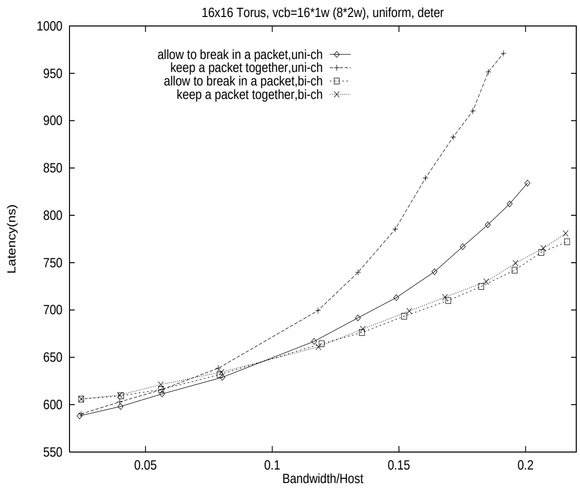


(a) Network bandwidth (including headers)

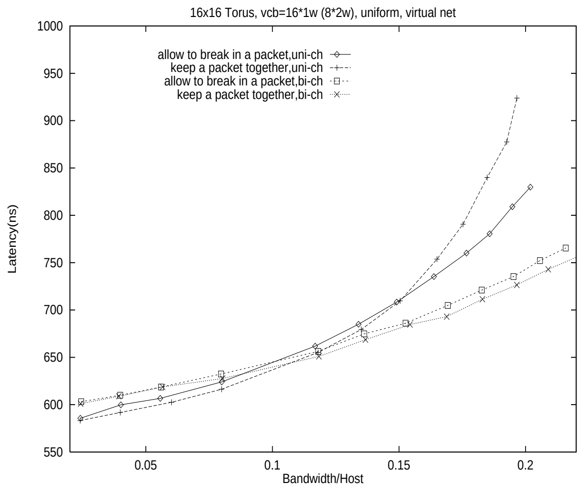


(b) Data bandwidth only

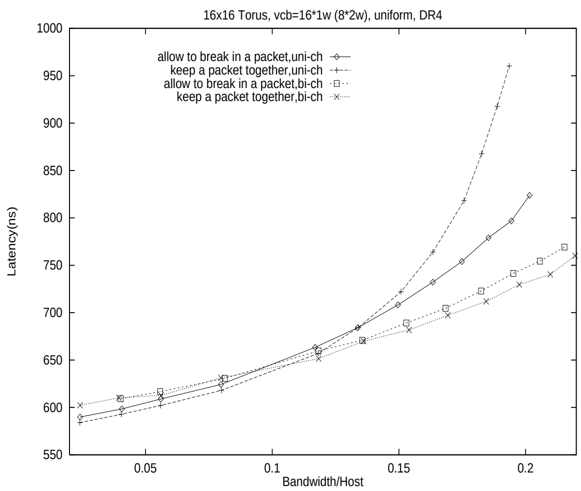
Figure 5.12: Latency for different packet length. Uni-directional channels. Deterministic routing under uniform random traffic.



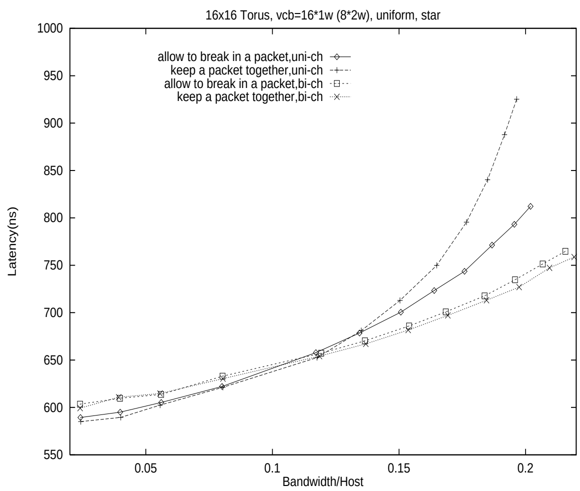
(a) Deterministic



(b) Adaptive (Virtual Networks)

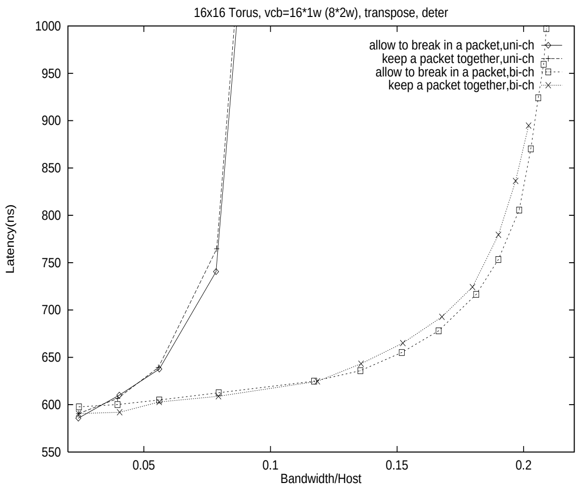


(c) Adaptive (Dimension Reversal)

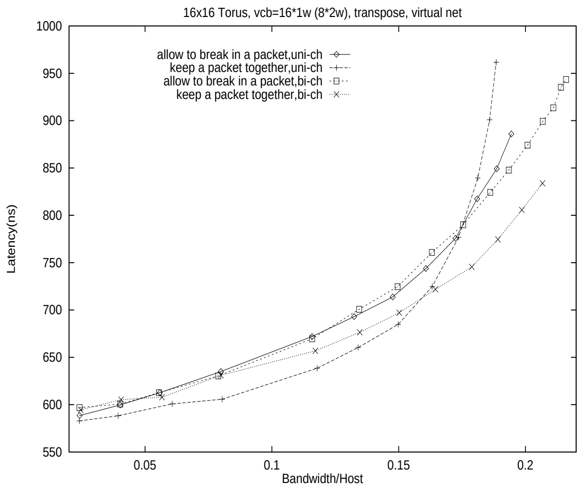


(d) Adaptive (Star Channel)

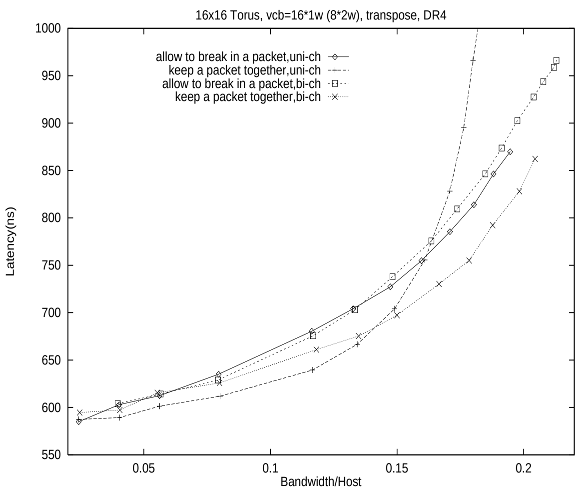
Figure 5.13: Latency versus throughput for different routing algorithms under uniform random traffic.



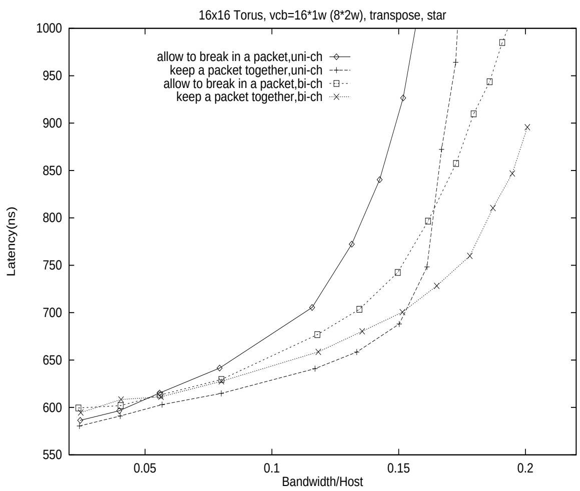
(a) Deterministic



(b) Adaptive (Virtual Networks)



(c) Adaptive (Dimension Reversal)



(d) Adaptive (Star Channel)

Figure 5.14: Latency versus throughput for different routing algorithms under transpose traffic.

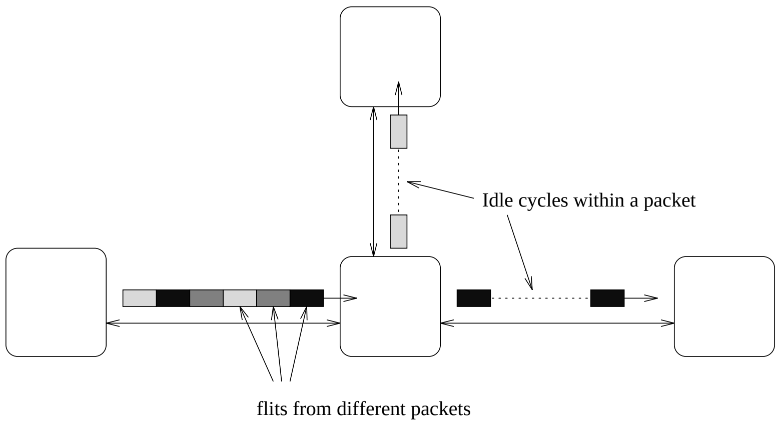


Figure 5.15: Interleaving flits from different packets will insert idle cycles in the packets when they continue to the next node.

Chapter 6

Segmented Reconfigurable Bus

Low dimensional networks (2-D mesh or torus) have been adopted in many recent multiprocessor implementations due to their simplicity, regularity, scalability, and feasibility. However, their main drawbacks are the large network diameter and possibly long distance communication between nodes.

As a result of this research, the segmented reconfigurable bus (SRB) torus architecture is proposed to overcome the delay due to "long distance" communications. In addition to the usual links between the nearest neighbors, we add the reconfigurable bus to the network as a means of reducing the latency of long distance communication, and to compensate for the large diameter of low-dimensional networks. A reconfigurable bus improves utilization, but the physical limitation in the bus length remains (Lu *et al.*, 1993b). Therefore we propose the *segmented* reconfigurable bus, and choose the segment length to mitigate the delay impact and increase resource utilization. It is necessary to arrange these segments carefully to minimize the interaction between different segments to reduce traffic contention.

In this chapter, we describe several torus architectures, including that of a torus with SRB. We also analyze and compare the delay for different interconnection models for different torus architectures. Simulation results show that the torus with SRB has the best performance in terms of routing latency as compared with other torus architectures. To increase further the performance of the SRB, an optimization procedure to select the segment length and segment alignment is developed. The results of a theoretical analysis of SRB performance are consistent with the simulation results, providing a guideline for designing a SRB torus network.

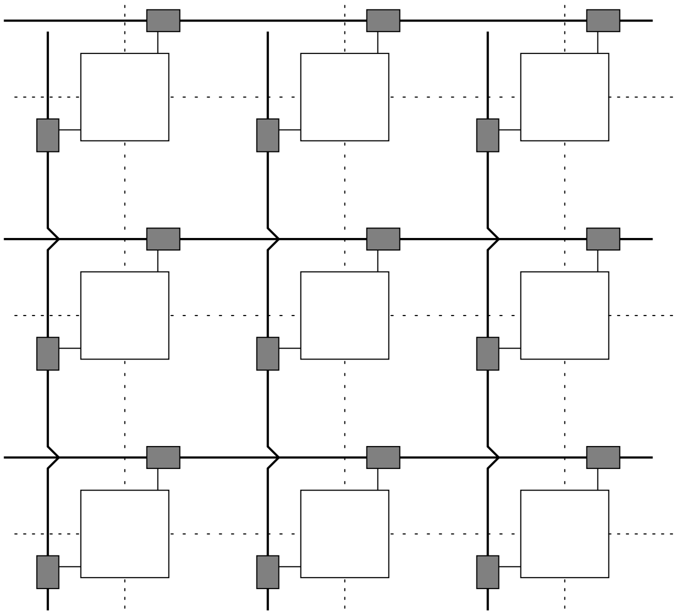


Figure 6.1: Torus structure with links and reconfigurable buses. Dotted lines are the links connecting nearest neighbors. Solid lines are the reconfigurable buses. The wrap-around connections of the torus are not shown. Also see Fig. 6.2 for detail connections.

6.1 SRB Architecture Overview

The network topology we use here is an $N \times N$ square torus. Each node contains a routing decision circuit, four I/O ports (receiver/transmitter), and k buffers. Each node is connected to its four nearest neighbors with *links*. In addition to the links between the nearest neighbors, *we add reconfigurable buses to each row and column* (Fig. 6.1). A reconfigurable bus is basically a series of links with switches to connect or disconnect these links In Fig. 6.1, the small shaded boxes represent these switches. Each reconfigurable bus is controlled by a token. Only the nodes which have the token can use the corresponding bus to send their packets. All the other nodes must use links to send packets only to their neighbors.

In order to compensate for the interconnection delay, we restrict the length of a reconfigurable bus. Each row and column is partitioned into several segments to form the SRB, and each segment has its own reconfigurable bus and token. Within a segment, only a single node can transmit onto the bus at any one time. Each segmented bus has two end points. In order to fully utilize each segment, we overlap the segmented buses, *i.e.*, two adjacent segments will share a common end point. Fig. 6.2 shows the overlap of the segments. Node

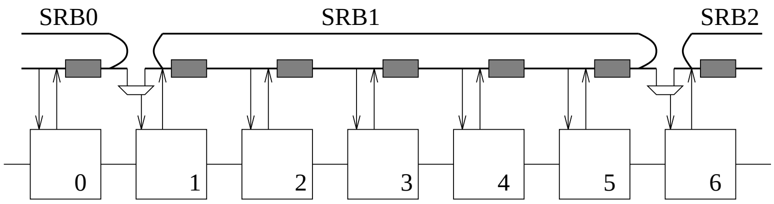


Figure 6.2: Adjacent segments share a common end points. Each segment has its own token ring without overlapping.

6 is shared by two adjacent segments, SRB1 and SRB2. Node 6 can receive packets from both SRB1 and SRB2; however, it can send packets using SRB2 only. The token of SRB1 only circulates from node 1 to node 5; node 6 belongs to the token ring of SRB2. In this arrangement, the two end points of each segmented bus belong to the different token rings, and packets in the end points can be sent by different segmented buses. To avoid confusion, we define the segment length as the length of the token ring. Thus the length of SRB1 in Fig. 6.2 is equal to 5. Fig. 6.3 is a 2-D torus with SRB. The SRB's are arranged such that the segment shift between adjacent rows (or columns) is s, and the offset between the first row and first column is t. The shift s and the offset t are the design parameters of SRB.

6.2 Routing Algorithms for Different Torus Architectures

A basic principle for efficient data routing is to reduce the distance packets travel to reach their destinations. In this section, we will briefly describe the routing algorithms on the different torus architectures: a torus with links only, a torus with global buses, a torus with reconfigurable buses, and a torus with segmented reconfigurable buses.

6.2.1 Torus with Links Only

We use the randomized routing (Section 4.4) as the basic routing algorithm for a torus with links only (torus_link). Recall that this is an adaptive routing with "derouting" for deadlock avoidance.

6.2.2 Torus with Global Buses

The links and buses can be combined together in a torus (or mesh) structure (Fig. 6.4). This has the advantage that buses can make long distance transmission much faster than links.

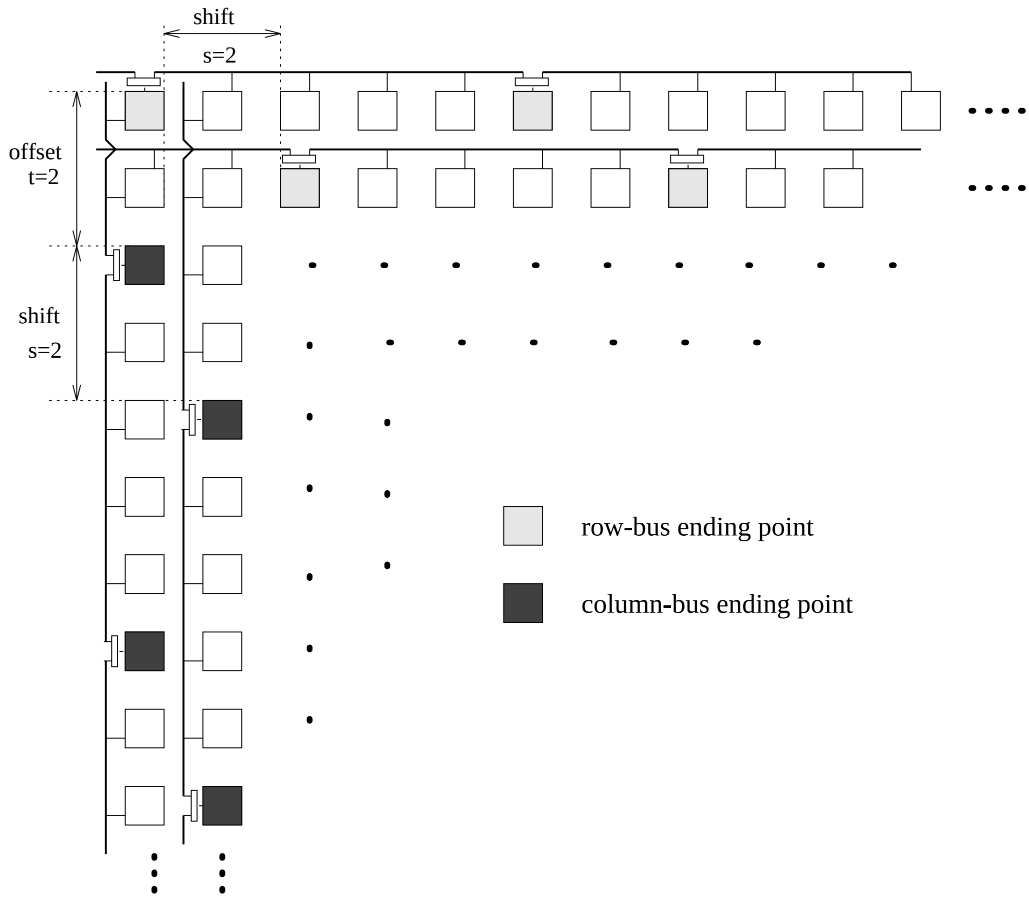


Figure 6.3: Torus with segmented reconfigurable bus, segment length=5, shift=2, offset=2. Only part of the network is shown for simplicity.

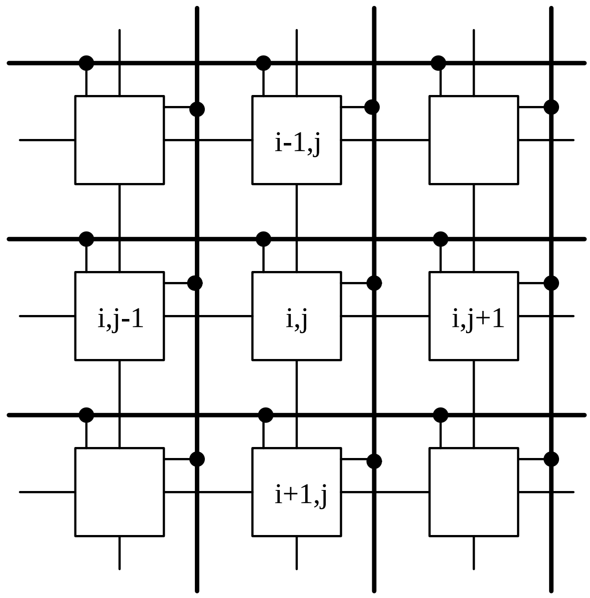


Figure 6.4: Torus with global bus

Tokens control the use of the buses, which guarantees that there is at most one node driving a bus at any time.

Algorithm 1 *Torus with global bus (torus_bus)*

- *Each row has a global bus and a row-token; each column has a global bus and a column-token.*
- *At the start, the tokens are randomly distributed within the nodes, but each row (or column) has only one row-token (or column-token).*
- *Only those nodes with tokens can use the global buses.*
- *When the node (i, j) can use the bus and has a packet whose destination is node (i', j'),*
  1. *If node (i, j) has the row-token,*
     - *if $(i, j) = (i, j')$, then the node uses link instead of bus to send the packet.*
     - *if $(i, j) \neq (i, j')$,*
       - *if the buffers in node (i, j') are not full, then node (i, j) uses the row-bus to send the packet to node (i, j').*
       - *if the buffers in node (i, j') are full, then node (i, j) does nothing.*

2. *If node (i, j) has the column-token,*
   - *if $(i, j) = (i', j)$, then the node uses link instead of bus to send the packet.*
   - *if $(i, j) \neq (i', j)$,*
     * *if the buffers in node (i', j) are not full, then node (i, j) uses the column-bus to send the packet to node (i', j).*
     * *if the buffers in node (i', j) are full, then node (i, j) does nothing.*
3. *If node (i, j) has both a row and a column-token, then it uses the row-bus first.*

- *The rules to use links are the same as torus_link.*
- *After sending the packets, those nodes with token pass their tokens to the next node,* i.e., *row-token: $(i, j) \to (i, \mathrm{mod}_N(j + 1))$, column-token: $(i, j) \to (\mathrm{mod}_N(i + 1), j)$.*

6.2.3 Torus with Reconfigurable Buses

The utilization of tokens is a crucial factor in increasing the performance of a torus with bus. In the torus_bus algorithm for the torus with global bus, some nodes possess the tokens but have nothing to send; in this instance, the bus is unused and the token is wasted. We may eliminate this situation by using the reconfigurable bus. It guarantees that the nodes receiving and keeping the tokens must have some packets to send unless all the nodes in the same row (or column) have empty buffers. Therefore the utilization of the bus is increased. A node of the torus with reconfigurable bus is shown in Fig. 6.5.

Algorithm 2 *Torus with reconfigurable bus (torus_RB)*

The rules to use links and buses are same as torus_link and torus_bus, but there are some new rules for token passing:

- *A node with empty buffers connects its (N, S) and (W, E) switches, therefore a token can bypass it,* i.e., *it will not grasp the token unless it needs to send packets.*
- *A node with row-token disconnects its (W, E) switch, and sends the token from E to its neighbor.*
- *A node with column-token disconnects its (N, S) switch, and sends the token from S to its neighbor.*

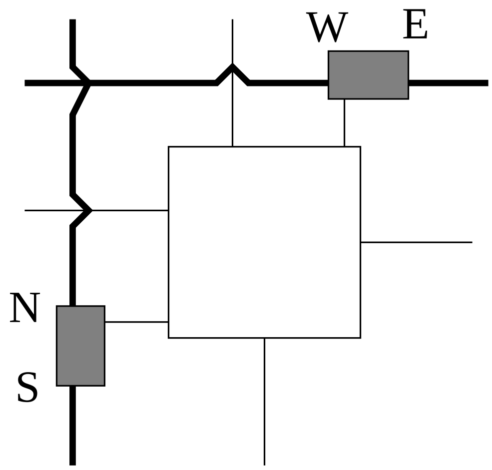


Figure 6.5: A node of the torus with reconfigurable bus.

- *If a node has some packets in its buffers, it disconnects its (N, S) and (W, E) switches,* i.e.*, it will have the opportunity to get and keep the token, and send the packet by bus in the next step.*

- *The nodes with disconnected (W, E) will keep the row-token if they receive the row-token. The nodes with disconnected (N, S) will keep the column-token if they receive the column-token.*

- *During the "packets sending" phase, all the (N, S) and (W, E) are connected as the global column and row buses respectively.*

6.2.4 Torus with Segmented Reconfigurable Buses

In practice, a reconfigurable bus comprises a series of repeaters (or transmission gates), making it difficult to implement a long reconfigurable bus without reducing throughput (the details of delay models will be given in the next section). Therefore the segmented reconfigurable bus is proposed for practical applications.

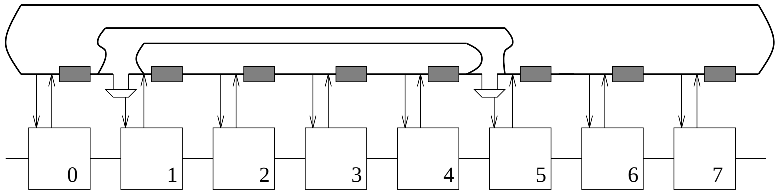


Figure 6.6: Wrap-around connections in a torus with SRB

The rules for the segmented reconfigurable bus are described as follows:

Algorithm 3 *Torus with segmented reconfigurable bus (torus_SRB)*

- *Each segment is a local bus with wrap-around connection.*
- *Each local bus has its own token which is only circulated in this local bus. The "token-passing" algorithm in each local bus bus is the same as torus_RB.*
- *Each local bus has two end points. Each local bus has two indicators to indicate if the buffers in these two end points are full or not.*
- *If a node has a token, and the intermediate destination of its packet is on this local bus, then use the bus as torus_bus.*
- *If a node has a token, but the intermediate destination is outside the local bus, then*
  - *if the buffers of the nearer end point (nearer to the destination) are not full, then use the local bus to send the packet to this end point.*
  - *if the buffers of the nearer end point are full, then use the links.*
- *The rules to use the links are the same as torus_link.*

In Fig. 6.6, there are 4 additional wires between rows (and columns) compared with the torus with links only. So the space penalty for SRB does not depend on the segment length. Also in Fig. 6.6, all the nodes are identical: they all have one input and one output port attached to a SRB.

6.3 Interconnection Delay

Interconnection delays directly affect latency and throughput of a network system, especially for a network with SRB. Before we consider the interconnection delay, we need to examine

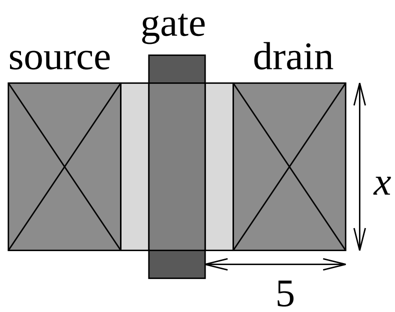


Figure 6.7: Transistor layout model

in detail the resistance and capacitance model of a transistor. Fig. 6.7 shows the layout of a transistor. Using $2\mu m$ technology parameters, typically,

$$\begin{aligned} R'_t &= 20k \cdot \frac{2}{x} = \frac{40k}{x}\ \Omega \\ C'_d &= 2 \cdot (5 + x) \cdot 0.5 + 5x \cdot 0.2 \\ &= 2x + 5\ fF \\ C'_g &= 2x \cdot 0.9 = 1.8x\ fF \end{aligned}$$

where R'_t is the transistor resistance, C'_d is the diffusion capacitance, and C'_g is the gate capacitance. Next we consider the delay models of link, long wire, bus, transmission gate, and repeater.

6.3.1 Link model

The link model is shown in Fig. 6.8.[1]

$$\tau_{link} = 2.3R_{out}(C_w l + C_{in} + C_{out}) + 2.3R_w(\frac{l}{w})C_{in}$$

[1] R_w is the wire resistance per unit square, C_w is the wire capacitance per unit length for a fixed wire width. For the following analysis, we assume the wire width is $8\mu m$, and use $2\mu m$ parameters: metal 1 area capacitance $= 26 \times 10^{-18}$, metal 1 peripheral capacitance $= 38 \times 10^{-18}$, then

$$\begin{aligned} C_{wire} &= 26 \times 8 \times l + 38 \times 2 \times (l + 8) \\ &= 284 \times l + 608 \approx .284 \times l\ (fF) \end{aligned}$$

so we choose $C_w = .284 fF/\mu m$.

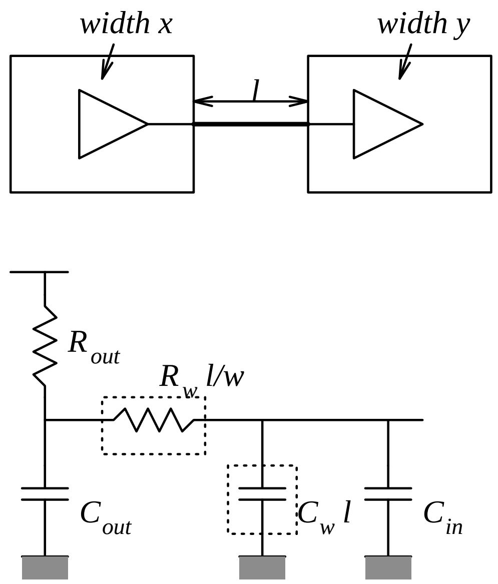


Figure 6.8: Link model, where the wire is modeled by distributed RC

$$+R_w C_w(\frac{l}{w})l \tag{6.1}$$

where R_{out}, C_{in}, C_{out} are the output resistance, input capacitance, and output capacitance of a node respectively. In our case, $R_{out} \approx R'_t = 40k/x$, $C_{in} = 2\ C'_d = 4x + 10$, and $C_{out} = 2\ C'_g = 3.6y$, where x, y are the transistor width of the output driver, input driver, respectively. We use different coefficients for lumped and distributed RC (2.3 for lumped RC, and 1 for distributed RC). (Bakoglu and Meindl, 1985)

6.3.2 Long wire model

The long wire model is the same as the link except that l is changed to $N \times l$. Then

$$\begin{aligned} \tau_{longwire} &= 2.3R_{out}(C_w Nl + C_{in} + C_{out}) \\ &\quad +2.3R_w(\frac{Nl}{w})C_{in} + N^2 R_w C_w(\frac{l}{w})l \end{aligned} \tag{6.2}$$

6.3.3 Bus model

The bus model is shown in Fig. 6.9.

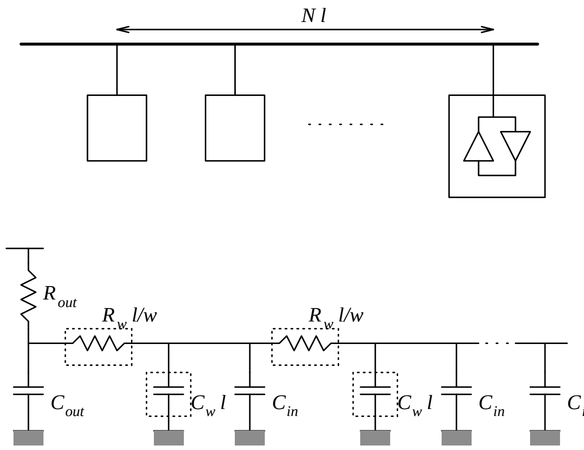


Figure 6.9: Bus model, where the parameters are the same as link model

$$
\begin{aligned}
\tau_{bus} &= NR_wC_w(\frac{l}{w})l + 2.3R_{out}(C_wNl + NC_{in} + C_{out}) \\
&\quad +2.3R_w(\frac{l}{w})\left[\sum_{i=1}^{N-1} i(C_wl + C_{in}) + NC_{in}\right] \\
&= NR_wC_w(\frac{l}{w})l + 2.3R_{out}(C_wNl + NC_{in} + C_{out}) \\
&\quad +2.3R_w(\frac{l}{w})\left[\frac{N(N-1)}{2}C_wl + \frac{N(N+1)}{2}C_{in}\right]
\end{aligned}
\tag{6.3}
$$

6.3.4 Transmission gate model

The transmission gate model and its interconnection are shown in Fig. 6.10.

$$
\begin{aligned}
\tau_{TG} &= 2.3R_{out}[C_wNl + 2(N+1)C_t + C_{out} + NC_{in}] \\
&\quad +2.3R_w(\frac{l}{w})\left[\sum_{i=1}^{N-1} iC_wl + \sum_{i=1}^{N}[(2i-1)C_t + iC_{in}]\right] \\
&\quad +2.3R_t\left[\sum_{i=1}^{N} i(C_wl + C_{in} + 2C_t)\right] \\
&\quad +NR_wC_w(\frac{l}{w})l
\end{aligned}
\tag{6.4}
$$

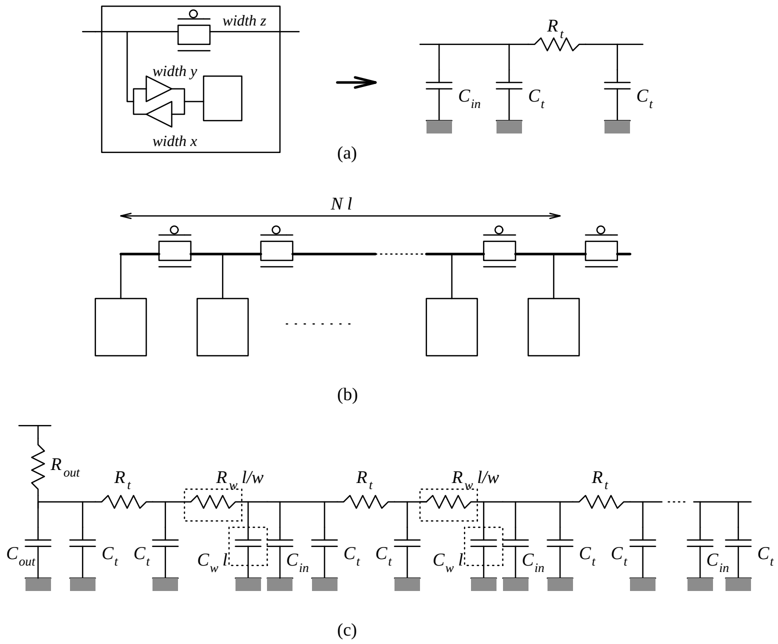


Figure 6.10: (a) Transmission gate model, where $R_t = \frac{40k}{z}$, $C_t = 4z + 10$, (b) Transmission gate interconnection, (c) Transmission gate interconnection model

$$\begin{aligned} = \; & 2.3R_{out}[C_w N l + 2(N+1)C_t + C_{out} + NC_{in}] \\ & +2.3R_w(\frac{l}{w}) \left[\frac{N(N-1)}{2}C_w l + N^2 C_t + \frac{N(N+1)}{2}C_{in}\right] \\ & +2.3R_t \left[\frac{N(N+1)}{2}(C_w l + C_{in} + 2C_t)\right] \\ & +NR_w C_w(\frac{l}{w})l \end{aligned} \tag{6.5}$$

where R_t and C_t are the resistance and capacitance of a transmission gate respectively.

6.3.5 Repeater model

The repeater used here is a tri-state driver. Its model and interconnection are shown in Fig. 6.11.

$$\begin{aligned} \tau_{repeater} = \; & 2.3R_{out}(C'_{in} + C_{out}) \\ & +2.3R_w(\frac{l}{w})[N(C'_{in} + C_t)] \\ & +2.3R_t[N(C_w l + C_t + C'_{in})] \end{aligned}$$

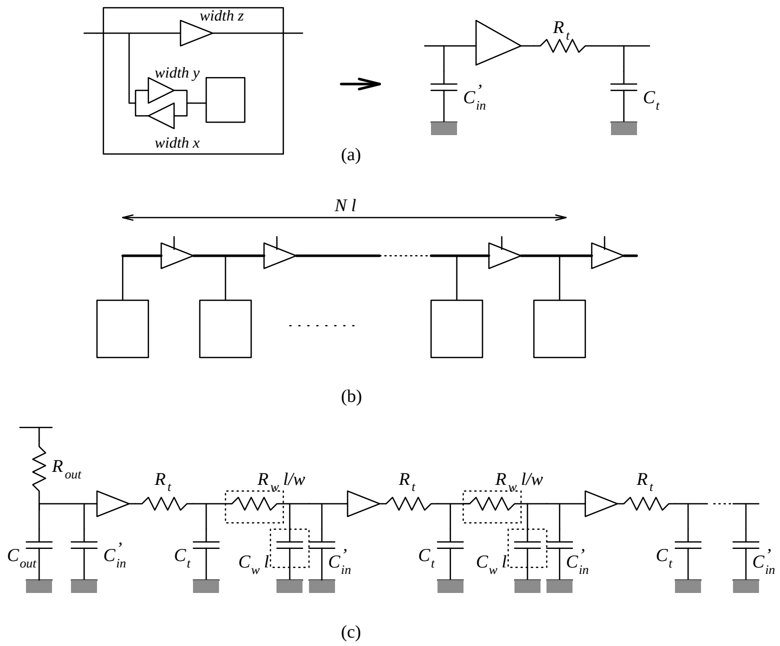


Figure 6.11: (a) repeater model, where $R_t = \frac{40k}{z}$, $C_t = 4z + 10$, $C'_{in} = C_{in} + 2C'_g$, (b) repeater interconnection, (c) repeater interconnection model

$$+NR_wC_w(\frac{l}{w}) \tag{6.6}$$

where C'_{in} is the input capacitance of a node, *i.e.*, $C'_{in} = C_{in} + 2C'_g = 4x{+}10{+}3.6y{+}3.6z$ (Fig. 6.11 (a)).

6.3.6 Delay Comparison

First, from Eq. 6.5 and 6.6, interconnection delay depends on the transistor width x, y, z (because R_t, C_t, C_{in} depend on x, y, z), therefore overall optimization requires optimal x, y, z to minimize interconnection delay. We know y is the width of the input driver transistor, which drives only a small number of circuits. Consequently the minimal width $y = 4\mu m$ (assuming $2\mu m$ technology) is sufficient. Fig. 6.12 shows delay of transmission gates and repeaters versus transistor width x for a fixed $z = 25\mu m$, and we can see the delay curve has a minimum at about $x = 20\mu m$. Fig. 6.13 shows delay versus transistor width z for a fixed $x = 20\mu m$, and the curve is very flat after z is greater than 25 μm. Therefore the choice of $x = 20\mu m$, $y = 4\mu m$, $z = 25\mu m$ will satisfy our requirements[2].

[2]The actual optimal number will depend on technology.

Links and wrap-around connections are composed of wires between two nodes. A global bus is a long wire connected to many nodes. A reconfigurable bus is links between adjacent nodes but with switches in each node to connect or disconnect those links. Transmission gates or repeaters (tri-state drivers) can be used for this kind of switch in the reconfigurable bus. Due to the square term (N^2) in Eq. 6.5, the delay of transmission gates grows much faster than that of bus and repeaters (Fig. 6.14), so a transmission gate is not a good interconnection for our reconfigurable bus from this point of view.

Fig. 6.15 compares delay of links, long wires, buses, and repeaters. The penalty for the wrap-around connection is just the wire delay. For moderate N ($N < 100$), this penalty is compensated by the reduction of both traffic congestion and diameter of the torus (from Fig. 6.18, the throughput of a long wire for $N = 80$ is about 3/10 of throughput of links, which can be compensated by reduction of diameter and congestion). Both bus and repeaters have longer delay than the long wire within the range of N in which we are interested. This implies that if we include the bus or the reconfigurable bus with the same length of the long wire, it will reduce the throughput of this system. From Fig. 6.15, we can see that, for example, the long wire delay for $N = 100$ is about equal to the delay of 25 repeaters. Therefore if we have a 100×100 torus with wrap-around connection, the maximum length of the reconfigurable bus without reducing throughput is about 25 nodes. But if we fold the torus to eliminate the long wires for wrap-around connection, the design will suffer degradation due to delay of the reconfigurable bus; thus the design is driven to a shorter bus length to reduce delay impact.

For smaller N, a bus is better than repeaters, but as N increases, the square term in bus delay (Eq. 6.3) grows faster than all the linear terms in repeater delay (Eq. 6.6), and use of repeaters is better than use of a straight bus. (Fig. 6.16)

Figs. 6.17 and 6.18 compare latency and throughput of several kinds of interconnections. The total delay of each step is the sum of interconnection delay and processing time in the node ($\tau_{delay} = \tau_{connection} + t_p$). Latency is the total delay between source and destination, and throughput is the maximum rate achievable for sending packets. It is obvious that latency and throughput are tradeoffs for these connections. For different applications, the importance of latency and throughput is different, however. For example, if a large amount of long distance data transfer is required, then latency is more important, for example, as in transposing a matrix. On the other hand, if most data transfer is local, the throughput is more significant, for example, as in systolic arrays. In the torus permutation problem, assuming that all the data are randomly distributed, then long distance and local communication are roughly of

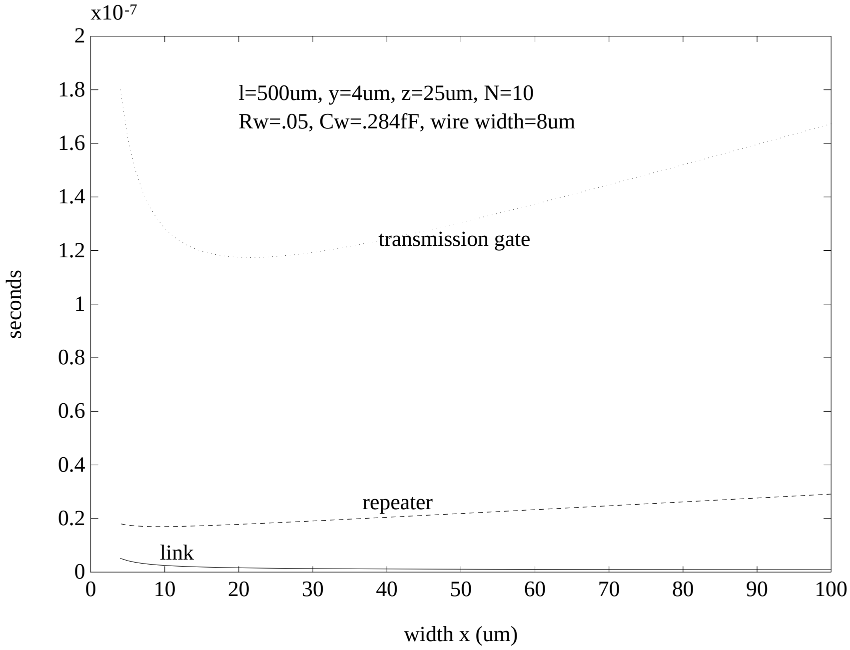


Figure 6.12: Delay versus transistor width x for a fixed width z

equal importance, so the bus (or reconfigurable bus) length must be chosen carefully in order to balance latency and throughput.

6.4 SRB Simulation and Comparison

The delay analysis in the previous section motivates the architecture with SRB. This section reports results of simulation of different torus architectures with links only, global bus connections, reconfigurable bus connections, and use of segmented reconfigurable bus connections, and compares their performance. In the simulation, we consider the 1-to-1 routing (permutation) on a two-dimensional $N \times N$ torus array with wrap-around connections.

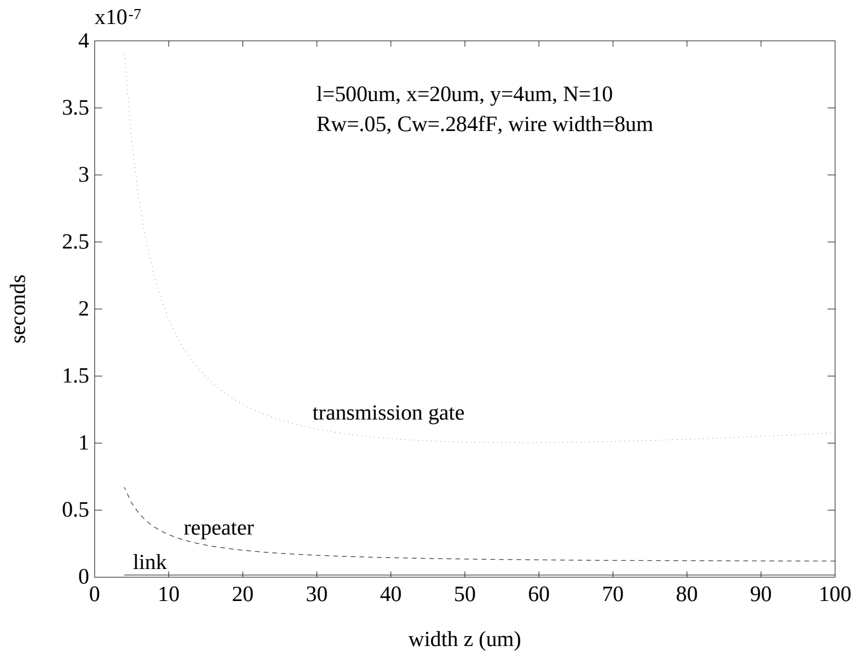


Figure 6.13: Delay versus transistor width z for a fixed width x

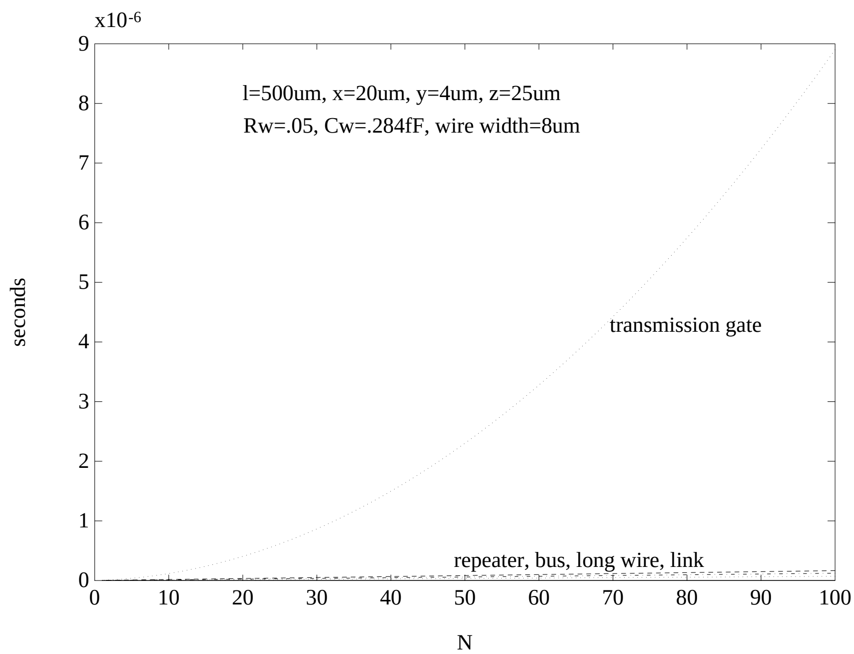


Figure 6.14: Delay versus N

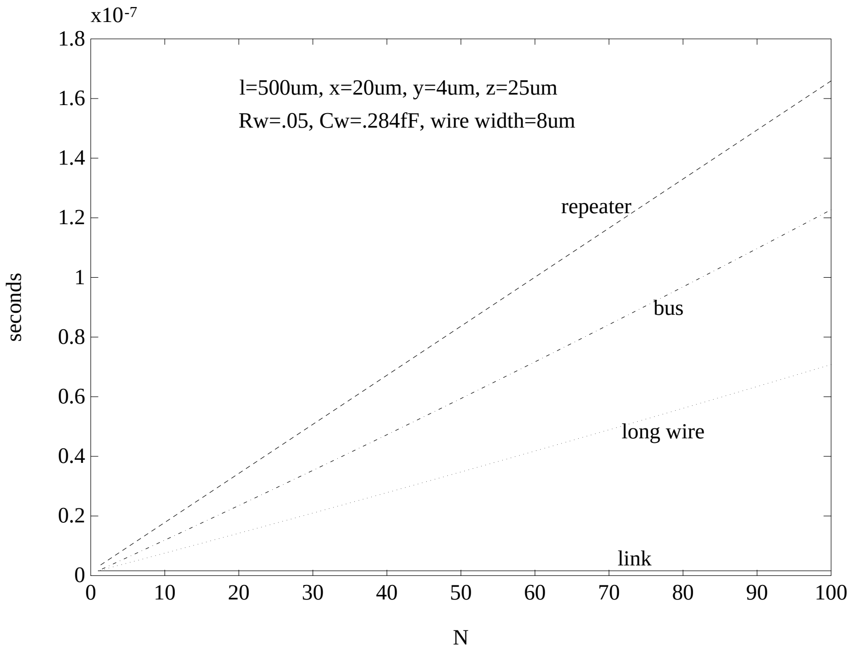


Figure 6.15: Delay versus N, excluding transmission gate

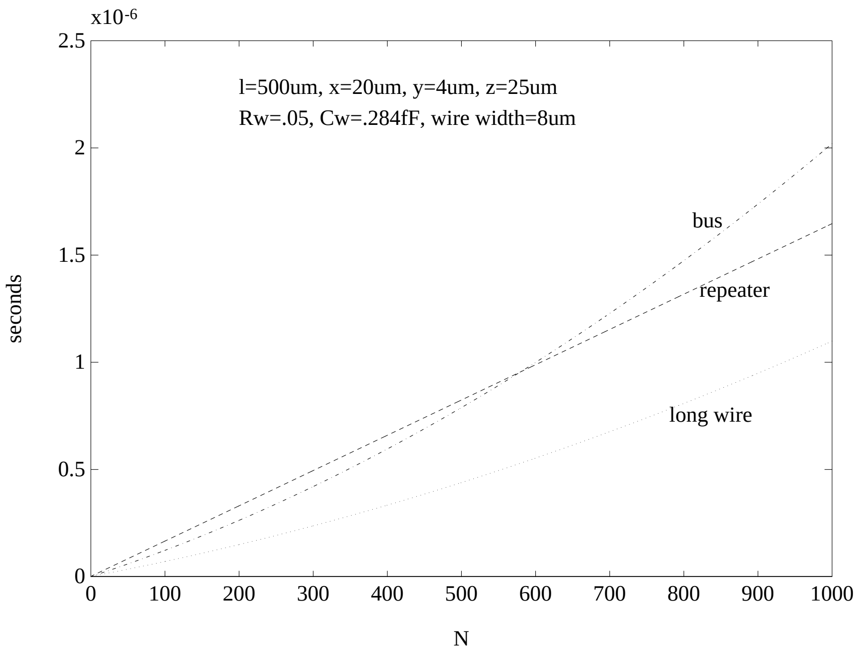


Figure 6.16: Delay versus large N

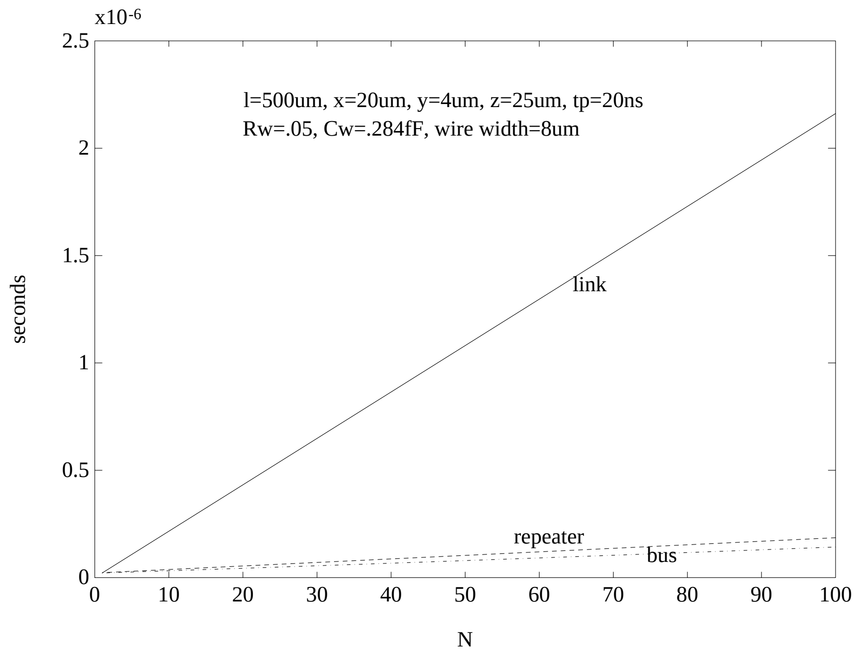


Figure 6.17: Latency versus N

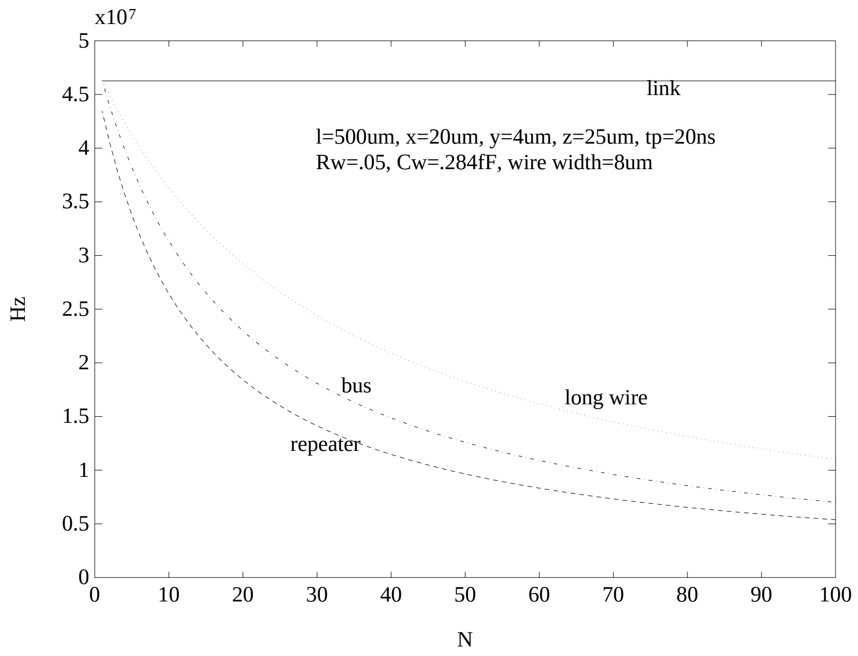


Figure 6.18: Throughput versus N

algorithm	average steps
Torus_link	1.4075N
Torus_bus	1.338N
Torus_RB	0.8144N
Torus_SRB	0.6065N

Table 6.1: Comparison of different torus architectures

Definition 9 *Permutation (1-to-1 routing)*

For an $N \times N$ torus, each node (i, j) in the torus has a packet whose destination is (i', j'). This is 1-to-1 mapping, i.e., *for every different source pair (i_1, j_1), (i_2, j_2), their packets' destinations (i'_1, j'_1), (i'_2, j'_2) are also different.*

The uniform random permutation is a permutation with all sources and destinations randomly distributed in the network. We first show the simulation results and a comparison of different torus architectures, and then interpret these.

6.4.1 Simulation Results

Fig. 6.19 shows the average number of steps to complete data transfer required by the four different architectures with $N \times N$ nodes, with their standard deviation as the error bars. The uniform random permutation is used as the data transfer pattern. Each simulation point in Fig. 6.19 is the average number of steps to complete 200 different random permutations. Torus_SRB needs the fewest steps to finish the data transfer. Torus_link and Torus_bus have very similar behavior because the bus utilization in Torus_bus is low and most packets are delivered by links. As N increases, the error bar of Torus_bus becomes larger, indicating increasing variability, and the average number of steps of Torus_bus exceed Torus_link. This is because the buffer size = 5 is too small for the configuration, and the traffic congestion delays the data transfer by both links and buses. We will discuss the detail tradeoffs later.

Fig. 6.20 shows the average steps versus different buffer size. The number of steps needed for a particular algorithm remain approximately constant after the buffer size is larger than some value, and this value is different for each algorithm.

Table 6.1 summarizes the simulation results of four different torus architectures. The average number of steps for the permutation problem appears linearly with the size N. For

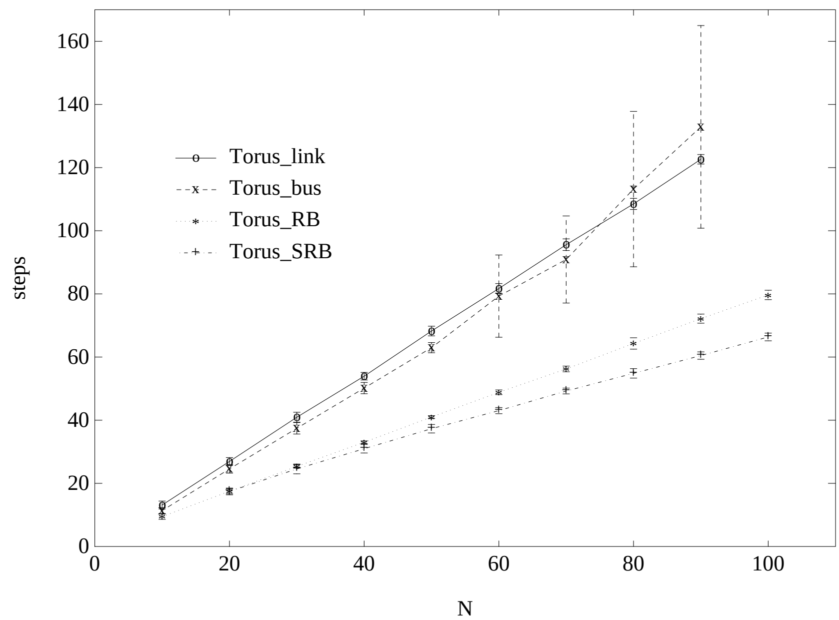


Figure 6.19: Average steps versus N, buffer size=5 for Torus_link, Torus_bus, and Torus_RB, buffer size=10 for Torus_SRB, segment length=10 for Torus_SRB

the torus_SRB, we can finish the permutation in $0.6065N$ steps on average, which is more than twice as fast as original torus architecture torus_link.

6.4.2 Discussion

Use of the torus with SRB is not only the most effective approach in terms of total steps required for a data transfer, but it is also feasible to implement in hardware for reasonable delays. The penalty for use of the SRB design is increased buffer size. This is a result of traffic congestion occurring at the end points of the local buses. The torus with SRB is basically a kind of hierarchical structure: links inside each local bus are the lowest level, segmented buses are the second level, and links between segmented buses are the highest level. Enhancing the routing ability of the lower levels increases the speed of routing, but would also increase the workload of the higher levels, causing congestion there. So balance is required to optimize a particular design.

There are several ways to mitigate the effect of traffic contention between different hierarchical levels. First, we can increase the buffer size in the nodes. Second, we can dedicate

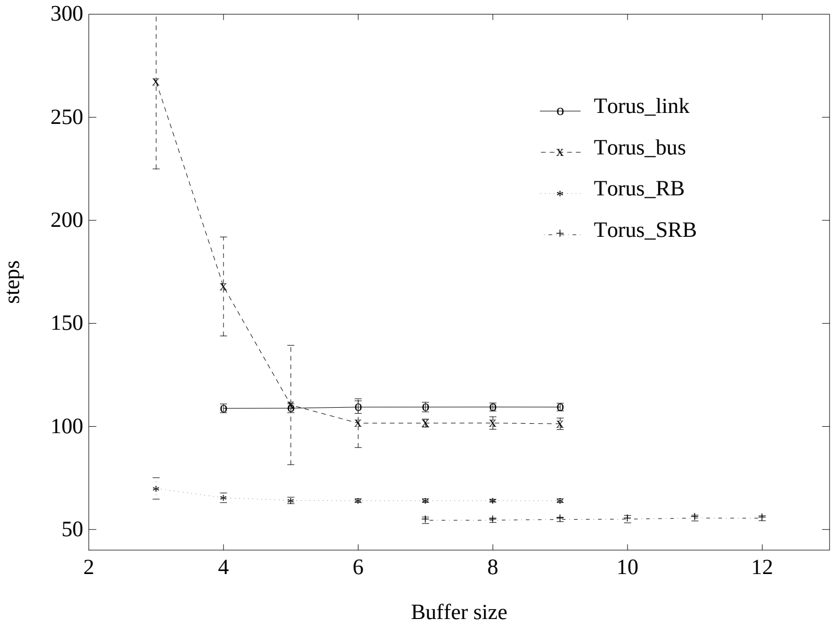


Figure 6.20: Average steps versus buffer size, $N = 80$, segment length=10 for Torus_SRB

more communication bandwidth to the highest level, *i.e.*, make the ending nodes of each local bus more powerful and increasing the number of links between them, so that they can route more than one packet in each of the steps. These two methods involve additional hardware and cost, however. The third method is to arrange the alignment of segmented buses carefully to reduce communications at the highest level (Section 6.5). Poor alignment of SRB will cause unnecessary traffic at the highest level, for example, if all the end points are aligned at the same rows or same columns, the interaction between the end points will cause additional traffic contention. Thus, achieving a balanced of workload in each hierarchical level is important to remove bottlenecks and improve performance.

Segment length is also an important performance parameter. As mentioned before, we must choose the segment length carefully to balance throughput and latency. When we design a torus with SRB, we need to consider all of these factors to choose the optimal segment length. In Section 6.5, we will show how to find the optimal segment length.

The algorithms with segmented reconfigurable bus comprise two phases: a packet sending phase and a token passing phase. We overlap these two phases to reduce communication and control overhead. Usually the token is a one-bit indicator, which is much shorter than the data packets. If we process the packet sending phase and token passing phase simultaneously, then the token passing phase must be completed sooner, and we may use the extra time to pipeline the rest of the routing procedures.

In a practical implementation, we can use repeaters to implement the reconfigurable bus in order to reduce delay (Section 6.3). This implies all data transfers by elements of the reconfigurable bus are unidirectional, so wrap-around connections are necessary in the reconfigurable bus. Furthermore, because the only required traffic on the reconfigurable bus is the token, we may use the traditional bus for packet sending and reconfigurable bus for token passing. This approach would simplify the hardware implementation without decreasing performance. Thus a near optimal torus network would consist of links for packet sending, segmented buses for packet sending, and segmented reconfigurable buses for token passing.

6.5 SRB Optimization

As discussed in the previous section, the segment length and segment alignment have crucial effects on overall performance. In this section, we now derive the methods for finding the optimal segment length and alignment (Lu *et al.*, 1993a).

6.5.1 Optimal Segment Length

Analysis

The length of each segment is a crucial design parameter for a torus with SRB. The interconnection delay of a segmented reconfigurable bus sets the maximum segment length without reducing throughput (Lu *et al.*, 1993b). Within this maximum allowable segment length, a designer can choose the optimal length that provides the shortest latency for completion of data routing. From the hierarchical concept as pointed out above, the workload balance in each level (links inside each local bus, segmented buses, and links between different segments) is an important issue with regard to bottlenecks and improving performance. Therefore, the choice of the segment length is the first step in design of an optimal torus with SRB.

Our goal here is to find a segment length L which minimizes the total steps required to complete a data transfer with arbitrary distribution. For the ith step, we define $\bar{d}_i$ as the average distance sent by each reconfigurable bus, and D_i as the total distance sent by reconfigurable buses and links. There are $2N(N/L)$ segments in the torus, each can send a packet over an average distance $\bar{d}_i$. The other $N^2 - 2N(N/L)$ nodes can use links to send packets to the neighbors with distance one. Ignoring the delay due to traffic congestion, the total distance D_i sent at step i is

$$D_i = 2(\frac{N}{L})N\bar{d_i} + (N^2 - 2(\frac{N}{L})N) \tag{6.7}$$

Let R_0 be the initial total packets distance, and R_n be the total distance remaining after the first n steps. R_0 is a function of the data transfer we want to solve.

$$R_n = R_0 - \sum_{i=0}^{n-1} D_i \tag{6.8}$$

Assume there are N^2 packets circulating in the network at step n.[3] Then each packet has the average remaining distance $\bar{r_n} = R_n/N^2$ to its destination. Each node with a bus token will send a packet via a segmented bus an average distance $\bar{d_n}$,

$$\bar{d_n} = \begin{cases} \frac{L}{2} & \text{if } \bar{r_n} \geq L \\ \frac{2}{L}\left[\frac{L}{2}\lceil x\rceil + \sum_{i=\lceil x\rceil+1}^{L/2} \frac{(\bar{r_n}+(\frac{L}{2}-i))}{2}\right] & \text{if } \frac{L}{2} \leq \bar{r_n} < L \\ & \text{where } x = \bar{r_n} - L/2 \\ \frac{2}{L}\left[\bar{r_n}\lfloor x\rfloor + \sum_{i=\lfloor x\rfloor+1}^{L/2} \frac{(\bar{r_n}+(\frac{L}{2}-i))}{2}\right] & \text{if } 0 \leq \bar{r_n} < \frac{L}{2} \\ & \text{where } x = L/2 - \bar{r_n} \end{cases} \tag{6.9}$$

Simulation

If we use random permutation (Def. 9) as an example, the initial total distance R_0 with wrap-around connection is $N^3/2$, and $\bar{r_n}$ can be written as

$$\bar{r_n} = \frac{N}{2} - \sum_{i=1}^{n}(1 + \frac{2}{L}(\bar{d_i} - 1)) \tag{6.10}$$

From equations 6.7 to 6.10, for each choice of segment length L, we may find a minimum n called n_o to make $R_{n_o} \leq 0$, which indicates that in the absence of traffic contention, the

[3]This assumption may not be true for the permutation problem because some packets can be consumed by their destinations before step n, but we also assume there are N^2 nodes sending packets at step n. Actually there are fewer than N^2 nodes sending fewer than N^2 packets, so these two assumptions can compensate each other and get the approximately equivalent results.

data routing can finish at the n_oth step. A smaller n_o implies a shorter latency. The optimal segment length L_{opt} should be chosen in order to achieve the minimum n_o.

Fig. 6.21 compares simulation and analysis results of the uniform random permutation for $N = 60$ and 80. Without considering the effect of traffic contention, the analysis results yield fewer steps to completion than the simulation results, but the curve shapes are very similar for both cases. The minima of simulation and analysis occur at about the same length, which gives confidence that our prediction of the optimal length is accurate. From these results, we infer that the optimal length is about $N/5$ or $N/4$ for the random permutation problem, and that performance does not change significantly if we choose a longer segment length. For moderate N ($N \leq 100$), this would be within the maximum allowable segment length[4] (Lu *et al.*, 1993b). The optimal length may be slightly different for the other problems, but the shape of the latency versus segment length curve is not expected to change very much. Because random permutation is a basic traffic pattern for a large class of communication problems, when we minimize the latency of the uniform random permutation problem, we minimize the life time of packets, and we increase network throughput and capacity in many applications.

6.5.2 Optimal Segment Alignment

The end points of a segmented reconfigurable bus are a kind of "hot spot" in a network. They carry much more of the workload than ordinary nodes. Proper arrangement of these end points can help to remove the effect of "hot spots" and reduce unnecessary traffic contention.

Ending points equation

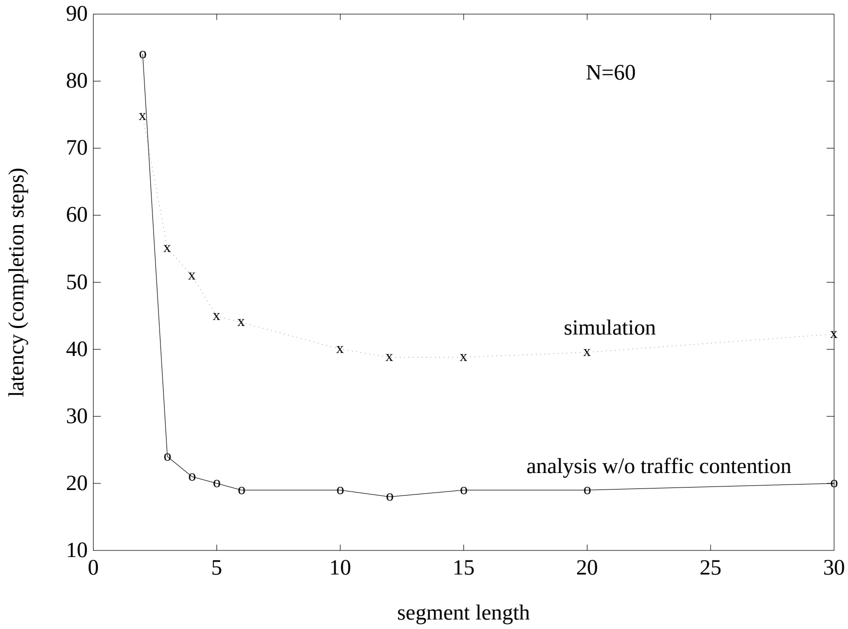
Let the segments on the $(i + 1)$th row (or $(i + 1)$th column) be shifted s nodes from the segments on the ith row (or ith column). The offset between the first row and first column segments is t (Fig. 6.3). Then we may express the positions of the end points as

$$\text{row end points:} \quad [i, \text{mod}_N(i \cdot s + n_1 \cdot L)] \tag{6.11}$$

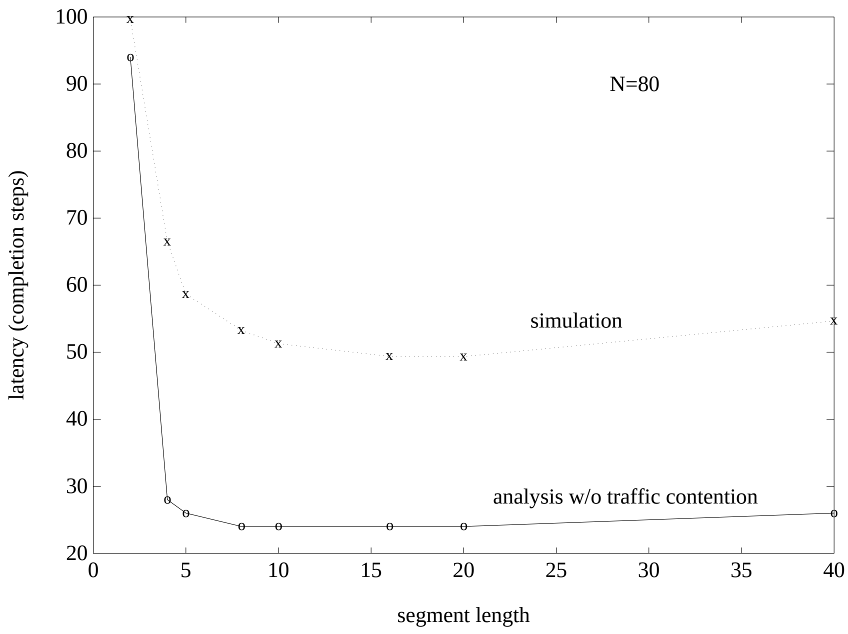
$$\text{column end points:} \quad [\text{mod}_N(j \cdot s + n_2 \cdot L + t), j] \tag{6.12}$$

where N is the torus size, L is the segment length, s is the segment shift between adjacent rows (or columns), t is the segment offset between the first row and first column, n_1, n_2 =

[4]The maximum allowable segment length is about 25 to 30 nodes for 2 μm technology.



(a) Latency vs. segment length, N=60



(b) Latency vs. segment length, N=80

Figure 6.21: Analysis and simulation of segment length.

$0 \sim (N/L-1)$, and $i,\ j = 0 \sim (N-1)$.

If we choose N to be an integer multiple of L, *i.e.*, $N = mL$, then we can simplify the above equation as

$$\text{row end points:} \quad [i, \mathrm{mod}_L(i \cdot s)] \tag{6.13}$$

$$\text{column end points:} \quad [\mathrm{mod}_L(j \cdot s + t), j] \tag{6.14}$$

where $i, j = 0 \sim (L-1)$. The distribution pattern of those end points is duplicated in the torus with a period of L nodes. So we only have to investigate the properties of a $L \times L$ region instead of the entire $N \times N$ torus.

Intuitively, if we can make the distance between any two end points as large as possible, then we can reduce the interaction between them and reduce traffic contention due to these hot spots. Given a segment alignment (given s and t), a criterion is needed to measure whether it is a good alignment or not. We define two factors as the metrics: the distribution factor (f_d) and the neighboring factor (f_n). The *distribution factor* measures the workload among the end points. If their workload is not uniform, then there may be a "hottest spot" in these hot spots, and it will become the bottleneck and reduce performance. The *neighboring factor* measures the effect between different end points. This takes the distance between end points into account, *i.e.*, the longer distance implies the smaller effect between them.

Distribution factor f_d

For each end point in a $L \times L$ region, we may define a distance distribution function which describes the relative distance to all other end points in this region.

Definition 10 *Distance distribution function*

We mark the $2L$ end points in a $L \times L$ region according to some arbitrary order from 1 to $2L$. For each end point, define a two-dimensional function as

$$dis_i(l) = k$$

This means for the ith end point, there are k other end points whose distance to this ith end point is l.

We obtain a set of distribution functions, one for each end point in the $L \times L$ region. This is similar to a stochastic process, each distribution function corresponds to a realization of the process (but this is a deterministic function, not random). We have $2L$ such sample processes, and we can take the ensemble average $dis_{ave}(l)$ as

$$dis_{ave}(l) = \frac{1}{2L}\sum_{i=1}^{2L} dis_i(l) \tag{6.15}$$

Using this ensemble average, we may calculate the difference between $dis_i(l)$ and $dis_{ave}(l)$. Define $\sigma_i^2 = \mathrm{var}[dis_i]$, then we have

$$\sigma_i^2 \equiv \frac{1}{L}\sum_{l=0}^{L}(dis_i(l) - dis_{ave}(l))^2 \tag{6.16}$$

We define the distribution factor f_d as the sum of the distribution variance of the total $2L$ end points.

$$f_d \equiv \sum_{i=1}^{2L} \sigma_i^2 \tag{6.17}$$

From this definition, f_d is a measure of the degree of uniformity of the distribution of those end points. If every end point has a similar distribution function dis_i, then it is also close to the ensemble average dis_{ave}, and resulting in a small variance σ_i^2. Therefore, small f_d (small total variance) implies that all the σ_i^2 are also small and that all the end points have approximately the same workload. Thus f_d is sensitive to a combination of two of the critical network parameters, the degree to which the load is balanced and to occurrence of hot spots.

Neighboring factor f_n

Even though small f_d indicates that each end point has about the same workload, it does not guarantee that this workload is small. We still have to investigate the traffic contention caused by the distribution of the end points. The closer together these end points are, the more serious traffic congestion will be. Our approach is to find a penalty coefficient corresponding to some distance between two ending points. The larger distance will have a smaller penalty.

For a certain segment in a row, there are $(N - L)$ nodes outside this segment. The probability for a node to send a packet to another node in the same segment by the segmented bus is about L/N. Therefore the probability for a node to send a packet to one of the end points of this segment is about $(1 - L/N)/2$. We define the penalty for distance l as the probability for an end point a to receive a packet from another end point b whose distance to node a is l.

When two end points coincide at the same node (a row bus and a column bus share the same node as their end points), we may regard this node as the end point a for a row bus and the end point b for a column bus. Then the probability for a to receive packets from b is just the probability for b to receive packets through the column bus (because a and b are coincidence). So the penalty p_0 for the coincidence of two end points is $(1 - N/L)/2$. When two ending points are adjacent, $l = 1$, the probability for one end point to receive packets from the other one is $1/4$ because each node has four neighbors and assume the traffic is uniform. Whenever we increase the distance between two end points by one, this probability is reduced by $1/4$.[5] Then the penalty for distance l is $(1/4)^l$ for $l \geq 1$. From the penalty definition and distance distribution functions, we may get the total penalty for the ith end point

$$P_i = \frac{1}{2}(1 - \frac{L}{N})dis_i(0) + \sum_{l=1}^{L}(\frac{1}{4})^l dis_i(l) \tag{6.18}$$

Now we define the neighboring factor f_n as the maximum total penalty among all the end points.

$$\begin{aligned} f_n &= \max_i P_i \\ &= \max_i[\frac{1}{2}(1 - \frac{L}{N})dis_i(0) + \sum_{l=1}^{L}(\frac{1}{4})^l dis_i(l)] \end{aligned} \tag{6.19}$$

This factor can give us the heaviest load in the network, which will be the hottest spot and bottleneck of performance. The smaller f_n indicates all the ending points are separated far enough and traffic contention caused by nearer end points is reduced.

Criterion for optimal segment alignment

As mentioned previously, we need to consider both the distribution factor f_d and the neighboring factor f_n in order to evaluate the overall traffic condition for a network. We define an alignment metric η as

[5] The exact probability is more complicated, but the approximation is sufficient for our purpose.

shift	offset						
	1	2	3	4	5	6	7
1	**0.5145**	0.1252	0.0453	0.0272	0.0224	0.0211	0.0207
2	0.2228	0.2228	0.3570	0.2257	0.2238	0.3547	0.2236
3	0.3182	0.3147	0.3158	0.3127	0.3122	0.3182	0.3147
4	0.0804	0.0089	0.0113	0.0804	**0.0065**	**0.0019**	0.0089
5	0.2272	0.2295	0.3446	0.2272	0.2295	0.3446	0.2272
6	0.1227	0.0183	0.0189	0.1228	0.2557	0.1227	0.0183
7	0.2238	0.2238	0.3547	0.2257	0.2236	0.3570	0.2199

Table 6.2: Alignment metric η, $\alpha = .5$, $\beta = 2$, $N = 60$, segment length=15

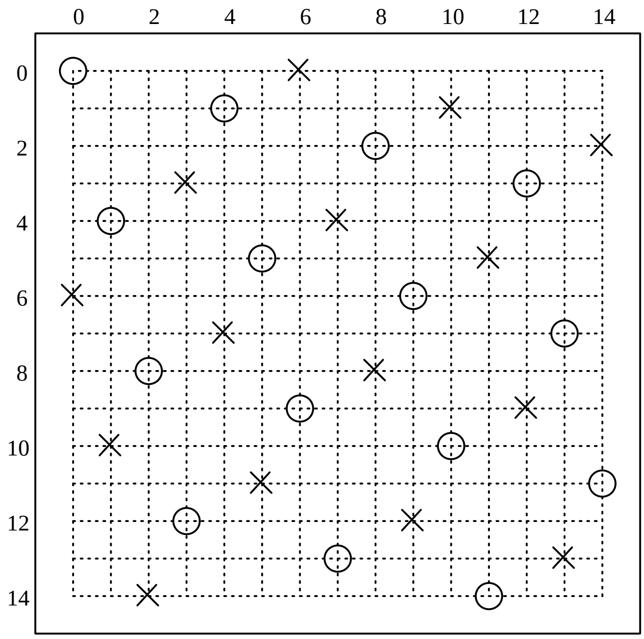
$$\eta = (f_d)^\alpha \cdot (f_n)^\beta \tag{6.20}$$

where α and β are the weighting factors for $\log(f_d)$ and $\log(f_n)$. For any positive α and β, traffic contention is reduced as η is decreased. Table 6.2 shows the metric η for $\alpha = .5$ and $\beta = 2$. Compared with simulation results of random permutation in Table 6.3,[6] the metric η is consistent with the simulation. The fluctuation of simulation gave us some exceptions compared with Table 6.2 and Table 6.3, but a smaller average number of steps basically corresponds to a smaller η. Hence the minimal η will correspond to the best performance. We found the minimal η always corresponds to the minimal f_d and f_n, so the optimal alignment metric would not be very sensitive to the different choices of α and β. Table 6.4 lists the optimal segment alignment for different segment length. Fig. 6.22 shows the examples of “good” alignment and “poor” alignment. A good alignment (Fig. 6.22(a)) has a uniform distribution and larger distance between end points. A poor alignment (Fig. 6.22(b)) is not very uniform and has shorter distance between adjacent end points, and coincidences of row and column end points (‘o’ and ‘x’ occur at the same place).

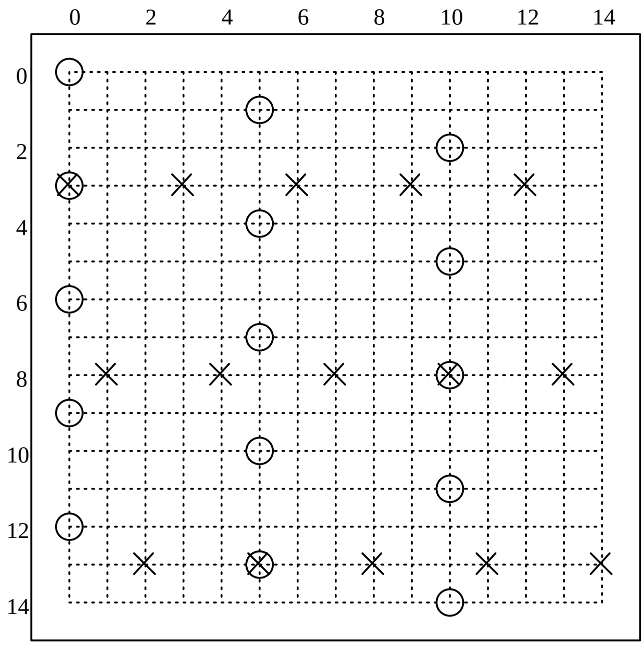
6.6 Summary

A reconfigurable bus is a very flexible structure for the mesh-connected multiprocessors. Many algorithms for different problems have been developed in networks with reconfigurable buses (Wang *et al.*, 1990; Snyder, 1982; Miller *et al.*, 1988). But the interconnection delay

[6]The standard deviation of simulation is ≈ 2.5 % of the mean value. Assuming that these values are normally distributed, the 95 % confidence interval is within 0.3 % of the mean value.



(a) Good alignment, shift=4, offset=6



(b) Bad alignment, shift=5, offset=3

Figure 6.22: Examples of segment alignment, $L = 15$, 'o': row end point, 'x': column end point. (a) shift=4, offset=6, distribution is uniform and distance between end points is large. (b) shift=5, offset=3, distribution is not uniform and there are three coincidences of 'o' and 'x'. Cf. Fig. 6.3.

shift	offset						
	1	2	3	4	5	6	7
1	**44.431**	42.170	40.815	40.135	39.780	39.705	39.895
2	41.640	41.643	43.410	41.845	41.652	43.420	41.750
3	42.151	42.580	42.675	42.595	42.685	42.555	42.550
4	40.553	39.750	40.110	40.495	**39.280**	**39.225**	40.050
5	40.535	40.575	44.805	40.590	40.650	44.795	40.345
6	40.140	40.340	40.095	40.270	44.420	40.285	40.620
7	40.440	40.475	42.715	40.623	40.580	42.750	40.450

Table 6.3: Average latency (completion steps), $N = 60$, segment length=15, buffer size=10. Simulation results are matched with the analytic results in Table 6.2

length L	10	11	12	13	14	15	16	17	18	19	20
shift	1	1	5	1	6	4	7	1	8	1	9
offset	5	5	3	6	4	6	4	8	5	9	5

Table 6.4: Optimal segment alignment for different segment length

of a reconfigurable bus will negatively impact the performance of these algorithms. We propose to use the segmented reconfigurable bus (SRB) to reduce the interconnection delay. Implementation of a reconfigurable bus by use of repeaters is also an important issue. The faster the available repeater, the smaller the delay for the reconfigurable bus, giving the designer more freedom to choose the proper segment length to achieve the best performance. We have shown that the torus with SRB is the fastest among the different torus architectures for the permutation problem in our simulation.

We also proposed a procedure to design the optimal torus with SRB. Given the communication distance distribution, we may find an optimal segment length within the feasible maximum length to achieve the best resource utilization and minimize the latency. After finding the optimal segment length, we arrange the segment alignment so that the interaction between the end points of these segments can be minimized, and we can reduce the unnecessary traffic congestion resulting from the hot spots in the network. We have shown that the best segment length for random permutation is about $N/5$ to $N/4$, and a good segment alignment may have 10% improvement over a poor alignment.

Chapter 7

Wormhole Router Design

Router hardware implementation has direct effects on the performance of multiprocessor communications. We designed and implemented a wormhole data router chip for 2-D mesh and torus networks with bi-directional channels and token-exchange arbitration. In our design, the token-exchange delay is fully hidden, and no latency penalty occurs in the absence of traffic contention. There are distributed decoders and arbiters for each IO port, and a fully-connected crossbar switch to increase parallelism of data routing. The router also has hardware support for path-based multicasting. From measured results, we will show that multicasting communication is much more energy-efficient than unicasting. The wormhole router was fabricated using MOSIS/HP 0.6μm technology. It can deliver 1.6Gb/s (50MHz) @ Vdd=2.1V and consumes an average power of 15mW.

7.1 Network Architecture

7.1.1 Network Review

As mentioned in Chapter 2, the network architecture is a primary factor affecting the performance of a parallel system. Due to hardware limitations, *e.g.*, bisection width, wire density, and interconnection delay, some network architectures are better than others from the implementation point of view (Dally, 1990b; Agarwal, 1991). For example, although high-dimensional hypercubes possess nice properties in which they have multiple IO connections and smaller network diameter, their channel width per IO port is much smaller than the channel width of a low-dimensional mesh network when the total bisection width is fixed. For a constant

Machine	Year	Topology
CMU/C.mmp	1972	Crossbar
Caltech/Cosmic Cube	1983	Hypercube
Intel/iPSC	1985	Hypercube
IBM/RP3	1985	Omega
TMC/CM-2	1987	Hypercube
Cray/Y-MP	1988	Multi-stage
BBN/Butterfly	1989	Butterfly
TMC/CM-5	1991	Fat tree
Intel/Paragon	1991	2-D Mesh
Stanford/DASH	1992	2-D Mesh
MIT/J-Machine	1992	3-D Mesh
Caltech/Mosaic C	1992	2-D Mesh
Cray/T3D	1993	3-D Torus
Stanford/STARP	1995	2-D Mesh or Torus

Table 7.1: Topologies of existing parallel systems

data flow, packets in a high-dimensional hypercube must be longer as compared with packets in a low-dimensional mesh, and longer packets suffer more traffic contention, resulting in a deterioration of throughput performance and increased latency.

Table 7.1 summarizes the network topologies of some parallel machines in the past twenty years. There has been some evolution over this time. Before 1990, there was considerable diversity in the network topologies people chose to implement. Hypercubes and multi-stage networks were particularly popular because of their good algorithmic properties for a wide range of applications. But in the 1990's, as VLSI technology improved, the component and wire density are increasing, and efficient implementation became more important. Low-dimensional networks (2-D or 3-D mesh) now dominate current multiprocessor topology implementations.

7.1.2 STARP Network Architecture

The STARP (Stanford Tilable and Reconfigurable Processor) is an array processor targeted for motion estimation in video compression application. While most machines in Table 7.1 are general purpose computers, STARP is a signal processing machine. There are important differences between STARP and general purpose machines in terms of network and communications:

- IO: in STARP, the IO stream comes mainly from video data which is a nearly constant high-bandwidth data flow. In general purpose machines, IO is much more bursty and unpredictable.

- Traffic Pattern: in STARP, the traffic pattern is essentially fixed by the motion estimation searching algorithm. In general purpose machines, however, the traffic patterns are very dependent on applications and data partitioning, and patterns may interfere with each other.

- Data Mapping: in STARP, a 2-D image maps directly onto a 2-D mesh or torus array. Therefore, we choose a 2-D torus as the network topology for STARP to preserve data locality and increase communication efficiency. In general purpose machines, depending on applications and topologies, data mapping may be less efficient and require more long distance communication.

- Multicasting: in STARP, the data stream is shared by several nodes for the purpose of performing image motion estimation. As a result, the benefit of using multicasting to reduce traffic is very high. In general purpose machines, multicasting messages are mostly invalidation signals for cache coherence; these usually are short packets and will not cause much traffic overhead. Consequently, the demand for hardware supported multicasting in general purpose machines is less than in STARP.

The STARP network architecture is shown in Fig. 7.1. It can be configured either as a 2-D mesh or a 2-D torus with nearest neighbor connections. Each STARP node contains a processing element and a data router. The processing element has a datapath optimized for motion estimation. There are five on-chip SRAM memory banks (each bank is 128×32 bits), and a global controller. The data router carries out data communication between nodes. The router performs wormhole data routing with hardware supported path-based multicast. Bi-directional channels are designed and implemented based on the simulation results in Chapter 5. The design and implementation of the STARP chip was done in collaboration with Gerard Yeh. The datapath and SRAM were designed by Gerard Yeh. In the rest of this chapter, we concentrate on the wormhole data router, and give the details of the router design.

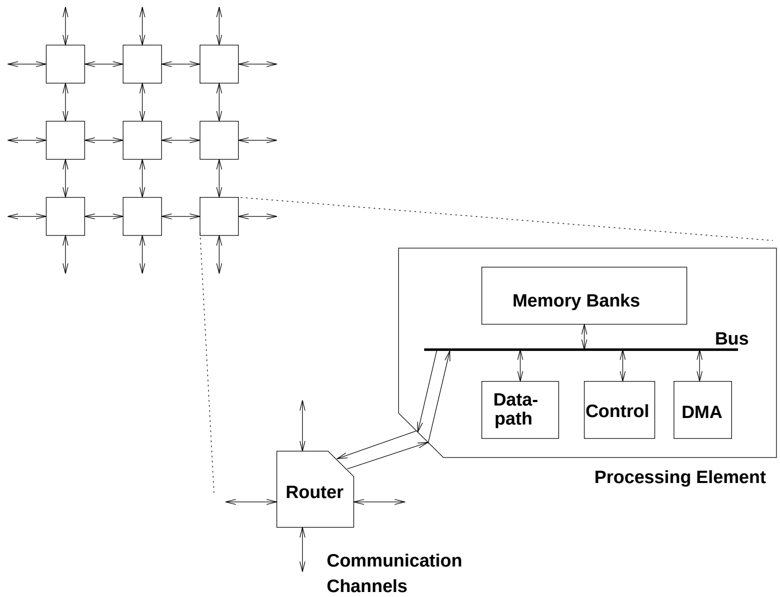


Figure 7.1: Network architecture of STARP

7.2 Previous Router Designs

Many wormhole routers have been designed and implemented for parallel systems. Table 7.2 summarizes a few implementations of the recent years (Dally and Seitz, 1986; Segucgi *et al.*, 1991; Traylor and Dunning, 1992; Reese *et al.*, 1994).

In 1986, Dally and Seitz (1986) designed the first wormhole routing chip for a 2-D torus (uni-directional in each dimension) with a throughput as high as 32Mb/s/port. As VLSI technology advanced, new generations of wormhole router provided more I/O ports with higher data throughput. But from Table 7.2, all of them implemented the uni-directional channel configuration, which is easier to design but supports less data bandwidth if there is traffic contention (see simulation results in Chapter 5); none support multicasting which reduces traffic congestion and communication energy dramatically. Therefore, in STARP, it was decided to implement bi-directional channels with token-exchange channel arbitration to optimize channel utilization, and to include hardware supported path-based multicasting.

7.3 Data Format of Different Levels

In this section, we describe the data format of three different levels: channel level, packet level, and flit level, in STARP.

	Year	IO Ports	Ch. Confg.	Multicast	Tech	BW
Dally & Seitz	85	3×3 Ports	uni-dir	No	$3\mu m$	32Mb/s
Seguchi	91	3×3 Ports	uni-dir	No	$0.8\mu m$	1.2Gb/s
Traylor et al.	92	5×5 Ports	uni-dir	No	$1\mu m$	1.6Gb/s
Reese et al.	94	5×5 Ports	uni-dir	No	$0.6\mu m$ BiCMOS	3.2Gb/s
STARP	95	5×6 Ports	bi-dir	Yes	$0.6\mu m$	1.6Gb/s

Table 7.2: Some previous wormhole router designs

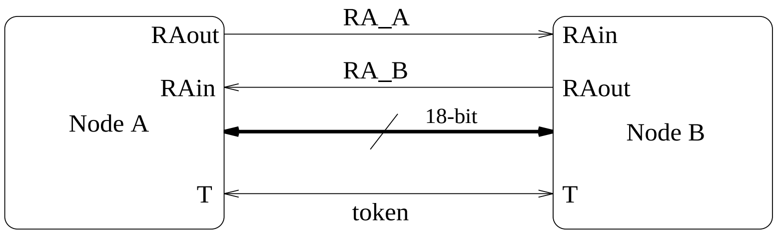


Figure 7.2: Channel format

7.3.1 Channel Format

Data channels are the physical media for transfer of information between nodes. Based on the simulation results for channel configurations in Chapter 5, we chose to implement bi-directional channels due to their capability to sustain a high data bandwidth.

To use the bi-directional channel configuration, a special channel arbitration scheme called "token-exchange" was designed. Section 3.4 describes the details of the algorithm. The basic idea was to prevent conflict of channel usage by neighboring nodes and to reduce the overhead caused by this arbitration.

Fig. 7.2 shows the channel format between two nodes in STARP. There is a one-bit signal T for "token-exchange" arbitration. Two-bit hand-shaking signals (RAin and RAout) are used for packet synchronization and contention control. Details of packet hand-shaking are given in Section 7.5. An 18-bit bi-directional channel (16-bit data and 2-bit control) is the physical channel for data transmission and reception.

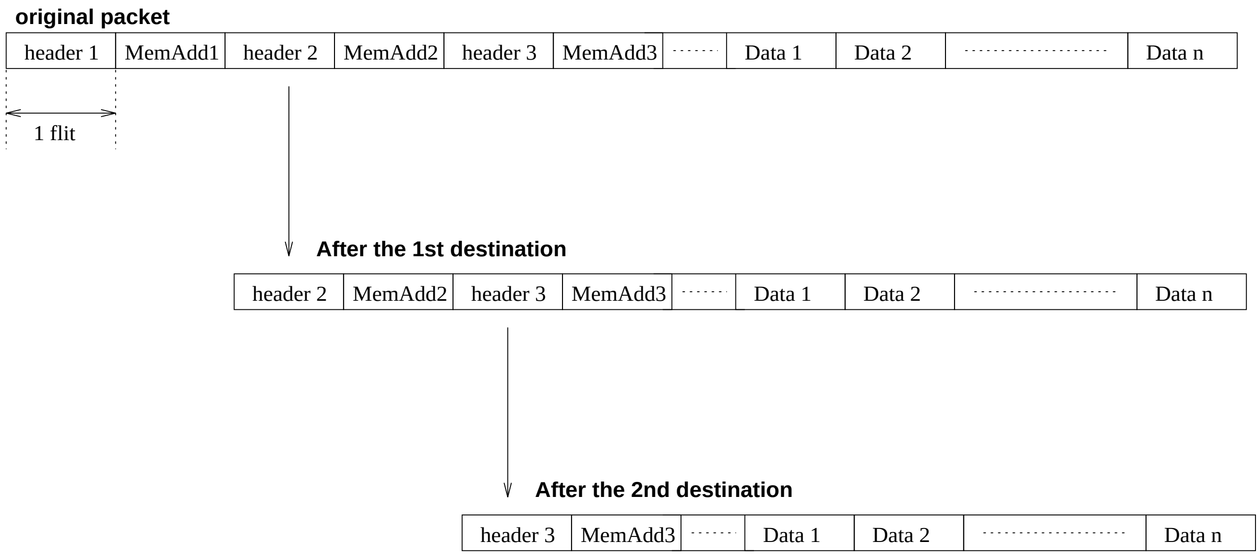


Figure 7.3: Packet format

7.3.2 Packet Format

A packet is divided into several flits for transmission. The packet length is arbitrary according to the communication protocol. In practice, however, the packet length is limited due to the finite counter size in the host interface which generates the outgoing packets.

In STARP, a packet is composed of three kinds of flits: *Header*, *MemAdd*, and *Data* flits. The definition of each type of flit is as follows:

- Header: packet destination *node* address.
- MemAdd: *memory* address to be written at the destination node.
- Data.

A basic packet must have one *Header* flit, followed by a *MemAdd* flit, and then one or more *Data* flits. Path-based multicasting is supported by the design (Section 4.5). A multicasting packet has multiple headers, with each *Header* flit followed by a *MemAdd* flit that indicates the memory address associated with the destination node specified by the *Header* flit. Fig. 7.3 shows the format of a multicasting packet. After reaching the first destination, *header1* and *MemAdd1* are removed, and the remainder of the packet is forwarded toward the second destination, and so on.

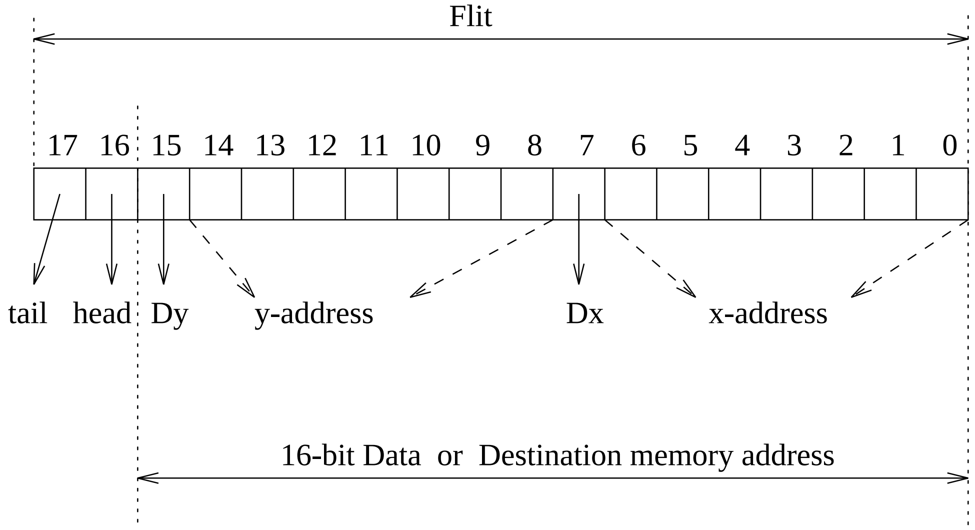


Figure 7.4: Flit format

7.3.3 Flit Format

A flit is the most basic unit for transmission. In the physical design, the flit size is equal to the communication channel width. In STARP, the channel is 18 bits wide for all ports. Fig. 7.4 shows the flit format. There are 2 bits for control signals indicating whether the flit is a tail or a header, and 16 bits for data or address. The tail bit is set only for the last flit of a packet, so it is a very low activity signal. One explicit head bit is necessary to indicate the *Header* flit because we require more than one *Header* to support multicasting. By using the additional head bit, we can specify an arbitrary number of destinations in a multicasting packet.

If a flit is a *Data* or *MemAdd* flit, it carries a 16-bit data or memory address, respectively. If the flit is a *Header* flit, bits 7 and 15 are the directional bits (Dx, Dy) which specify the direction it will travel in the corresponding dimension (0 indicates negative, 1 indicates positive). Dx and Dy are calculated at the source, and never change along the path. Therefore, the intermediate nodes do not have to re-decode the address to compute Dx and Dy. Bits 0 to 6 and bits 8 to 14 are the x-address and y-address for the destination, respectively. The maximal addressable network size is $2^7 \times 2^7$.

7.4 Router Architecture

The design goals for this router are: 1) reduce fall-through latency, 2) maintain high throughput (one flit/cycle/port), 3) increase parallelism, and 4) reduce power consumption. The router has five ports: four external I/O ports with the nearest neighbors on the mesh network, and one port with the local host (Fig. 7.5). The main blocks in the router are: Host interface, header decoders, arbiters, I/O controllers, and crossbar switches.

7.4.1 Host Interface

The host interface is the block between the router core and the local host (Fig. 7.6). There are two kinds of modules in the interface: Source Output Control (SrOut) and Source Input Control (SrIn).

SrOut controls the path from the local host to the network. It receives the commands from the host and injects packets into the router core. We will describe the details of commands in the later section (Section 7.6). SrOut executes the following tasks:

- Generating the outgoing packet headers including *Header* and *MemAdd* flits.
- Setting SrOut counter for packet length (how many *Data* flits in the packet).
- Issuing memory read to fetch data for outgoing packets.
- Generating the tail bit for the last flit in a packet.

Similarly, SrIn controls the path from the network to the local host. It receives the incoming packets destined for the local host. As shown in Fig. 7.6, we have two SrIn modules to increase the data bandwidth to the local memory. Sometimes, especially when there are multicasting packets, there may be more than one packet arriving at the same destination at the same time. If there are two such packets whose destination memory addresses are in the different memory banks, then there is no conflict between them. Thus two SrIn modules can operate simultaneously and double the bandwidth to the host to reduce the bottleneck effect caused by host. The main tasks of SrIn are:

- Filtering out header information before writing packets to memory. SrIn will keep monitoring the head bit from the router core. If it sees head=1, it will delete 2 consecutive flits (*Header* and *MemAdd*) before writing to memory.

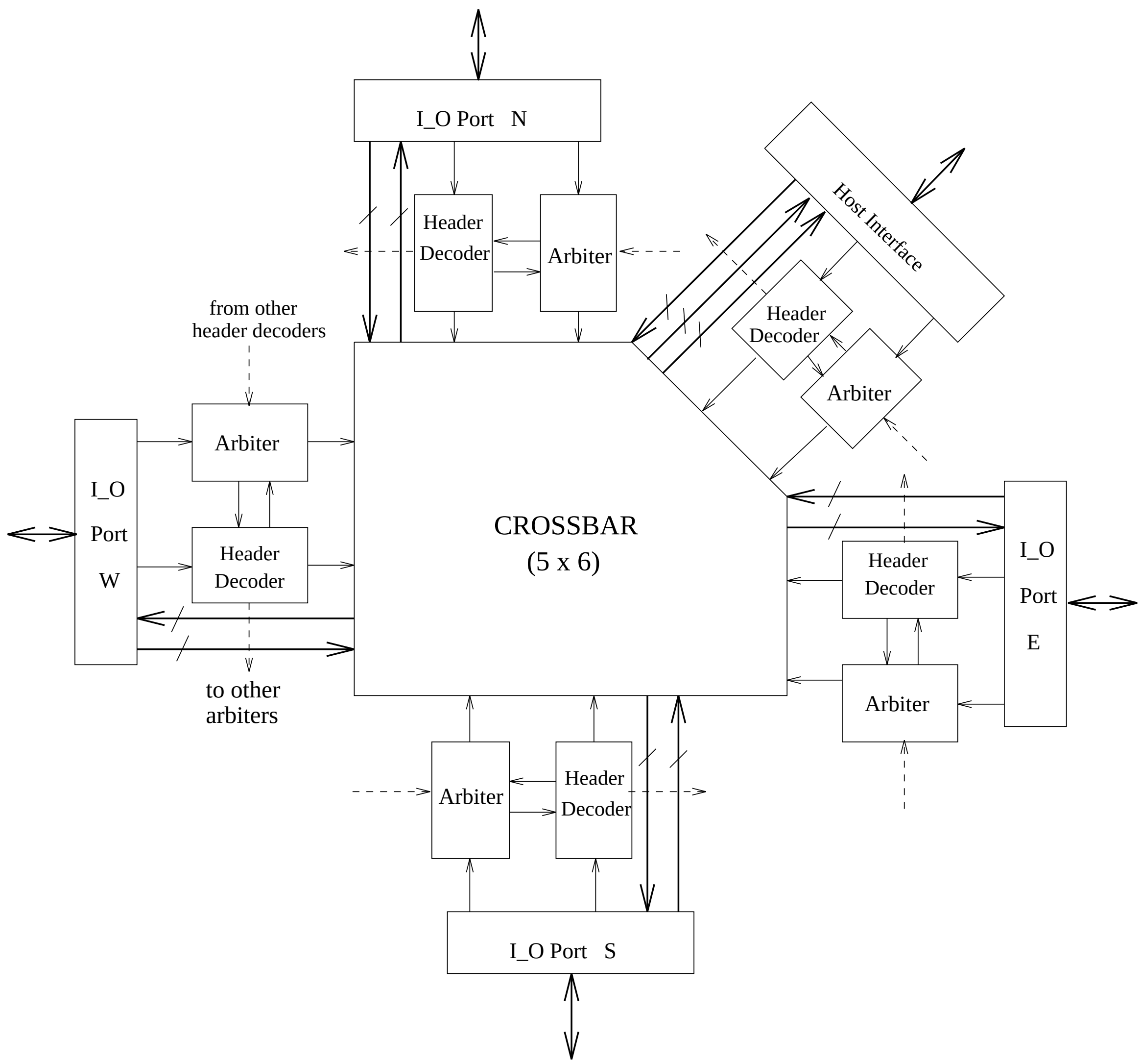


Figure 7.5: Global router architecture

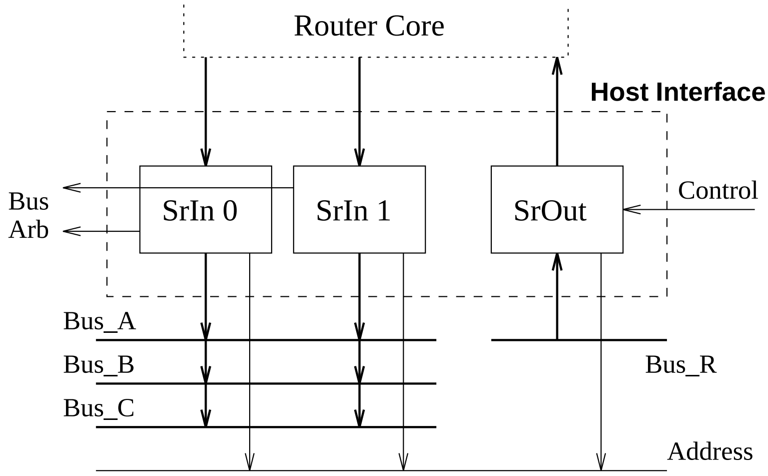


Figure 7.6: Host interface architecture

- Arbitrating data bus conflicts.
- Issuing memory write commands.

The memory buses are 32 bits, while the communication channels are only 16 bits. So for both SrOut and SrIn, two flits correspond to one memory access. Thus, the maximal memory bandwidth consumed by the router is less than half of the total memory bandwidth available. The processing element can share the memory bandwidth with the router by properly interleaving their memory access.

7.4.2 Port Architecture

The router has four identical ports (North, East, West, South) connected to the nearest neighbors. Each port contains: I/O control, input and output FIFOs, a header decoder, and an arbiter (Fig. 7.7). We have distributed header decoders and arbiters in each port to increase parallelism of data routing. As long as there is no output conflict, (*e.g.*, no more than one packet is heading toward the same output port), different packets can proceed at the same time.

I/O Control and FIFO

The I/O control implements the link-level communication protocol and token-exchange arbitration (Section 3.4). We have assumed the network is synchronous, *i.e.*, all the nodes

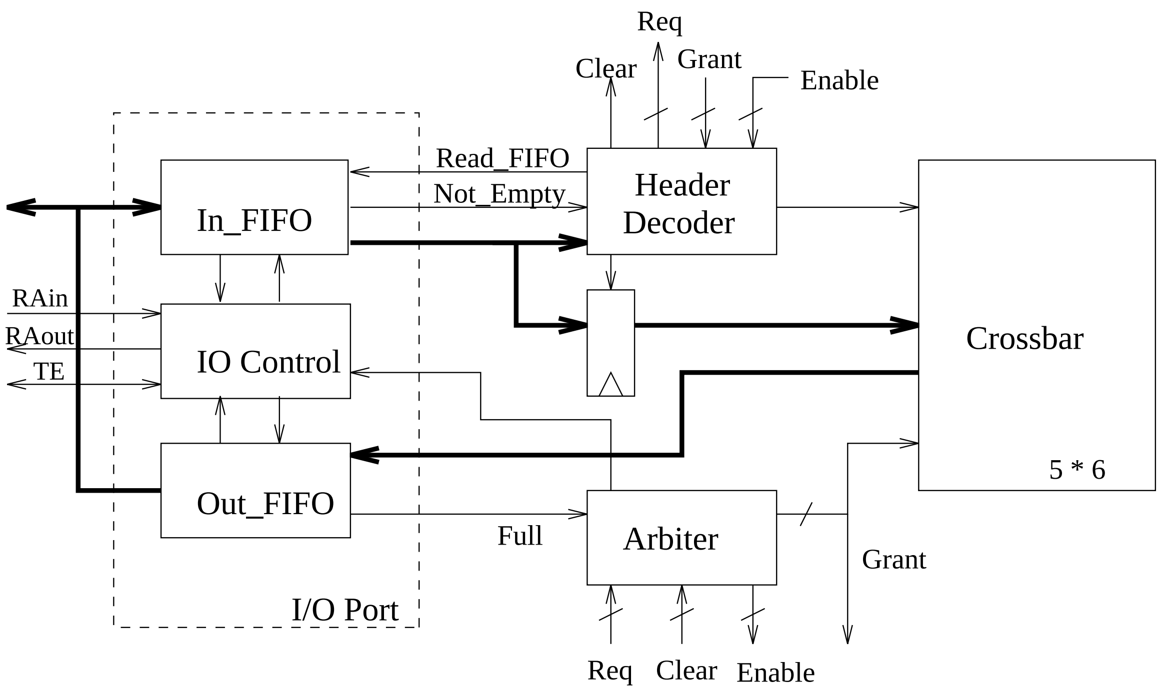


Figure 7.7: Router port architecture, only one port is shown

are driven by the same clock and have the same frequency and phase.[1] The communication hand-shaking occurs at the packet level instead of flit level (Dally, 1990a; Hwang, 1993) in order to reduce hand-shaking overhead and increase data throughput. When a port acts as an output port, its RAout is the *Req* signal, and RAin is the *Ack* signal from its neighbor. On the other hand, when a port acts as an input port, its RAout is the *Ack* signal, and RAin is the *Req* from the neighbor. When an I/O port changes from output to input, its RAout and RAin also change their roles. Fig. 7.8 shows the interaction of *Req* and *Ack*. When there is a packet at the output port (*i.e.*, the output FIFO is not empty), the *Req* signal, the RAout of the sender, will go high in order to notify the neighbor that there is an incoming data stream. Because they are synchronous, the receiver can sample and latch the incoming data as soon as the signal is stable. *Req* stays high if the sender keeps sending the following flits of the same packet. When the receiver's input FIFO is full, it will set *Ack* (its RAout) to be high to signal the sender. The sender will stop sending data and lower *Req* to prevent buffer overflow and flit loss. After the receiver has available space in its input FIFO, it will set *Ack* to low

[1]Synchronous networks can be achieved by 1) distributing a global clock with carefully balanced delay, 2) PLL in each local node, or 3) Source Synchronous Driver/Receiver, where the local clock is sent along with data.

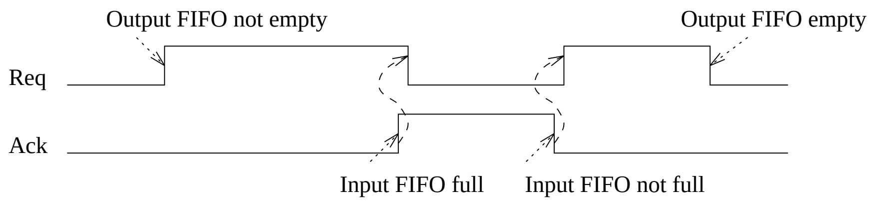


Figure 7.8: Channel hand-shaking in the packet level. *Req* is the RAout of the sending node and *Ack* is the RAout of the receiving node

and the sender will resume data transmission by setting *Req* high again. When the sender finishes the entire packet (output FIFO empty), it will set *Req* to low to tell the receiver that no more data are on the channel.

Each I/O FIFO is two-flits deep. For an input FIFO, if there is space for only one flit, then when a new flit arrives, the *Ack* (input FIFO full) signal will be high, and will slow down the transmission rate of the sender even though there is no traffic contention. So we need at least two-flit FIFO to eliminate this false full phenomenon. For an output port, although we can hide the token-exchange delay by pre-issuing token request (Section 7.5), we need some extra buffering when traffic contention happens. Therefore, we choose two flits per I/O FIFO to reduce hardware cost while maintaining functionality.

Header decoder

The header decoder determines to which output port a packet will be routed based on the routing algorithm. The design implements the dimension-order deterministic algorithm because of its simplicity and efficiency. When a header flit comes into the header decoder, it first compares the local host address with the packet destination address. If they are identical, this packet has reached its destination and the host interface is signaled that a packet has arrived. If the x addresses are different, the packet will be sent to x dimension according to Dx; if the x addresses are the same but y addresses are different, the packet will be sent to y dimension according to Dy. We can formulate the operation as follows:

$$\begin{aligned} R_x &= X_p \oplus X_n \\ R_y &= Y_p \oplus Y_n \end{aligned} \tag{7.1}$$

$$\text{If } \overline{dest} \equiv R_x + R_y = 0, \text{ then send to the local host,}$$

$$\text{else if } R_x \neq 0, \text{ then send to } x \text{ (E or W) according to Dx bit,}$$

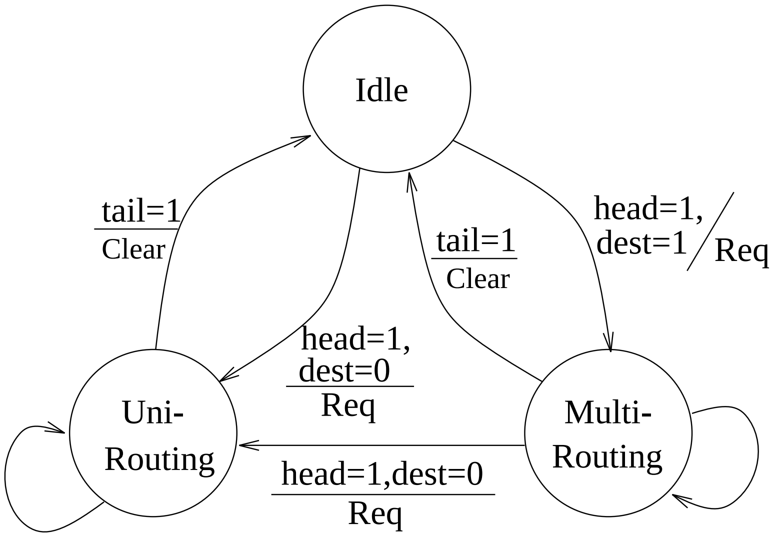


Figure 7.9: State diagram of the header decoder

else if $R_x = 0$, then send to y (N or S) according to Dy bit

where $X_p(Y_p)$ and $X_n(Y_n)$ are the $x(y)$ addresses of the packet destination and local node, respectively. And $\oplus$ is XOR operation.

The header decoder also was designed to support path-based multicasting. Fig. 7.9 shows the state diagram for the header decoder. When a new packet comes in, the header decoder changes from the idle state to either the uni-routing state or the multi-routing state, depending on whether or not this packet has reached its destination. If dest=0, the current node is only an intermediate node. In this case, the decoder goes to uni-routing state and the entire packet is forwarded to the next node. When dest=1, the packet has reached it destination (or the first destination on the list), and the header decoder goes to the multi-routing state. In the multi-routing state, the header decoder continues to monitor the next head bit in the packet, and issues a routing request if a second header arrives. The header decoder goes to the uni-routing state after the second header has been decoded. After the tail bit arrives, the header decoder goes back to the idle state, clears the switches, and waits for the next packet.

Arbiter

The arbiter receives and arbitrates requests from the header decoders of all input ports. When the arbiter receives requests, it first checks its internal state to see if the switch to the output has been used. If the switch is free and there is only one request at that time, then the port issuing the request can use the switch immediately. If the switch is available but there are more than one request from different ports, then round-robin priority is used to decide which one is allowed to use the switch, and all the other requesting ports wait. If the switch to the output port is being used, all the requests are denied. Because we do not implement virtual channels, packets to the same output port can not be interleaved with each other.

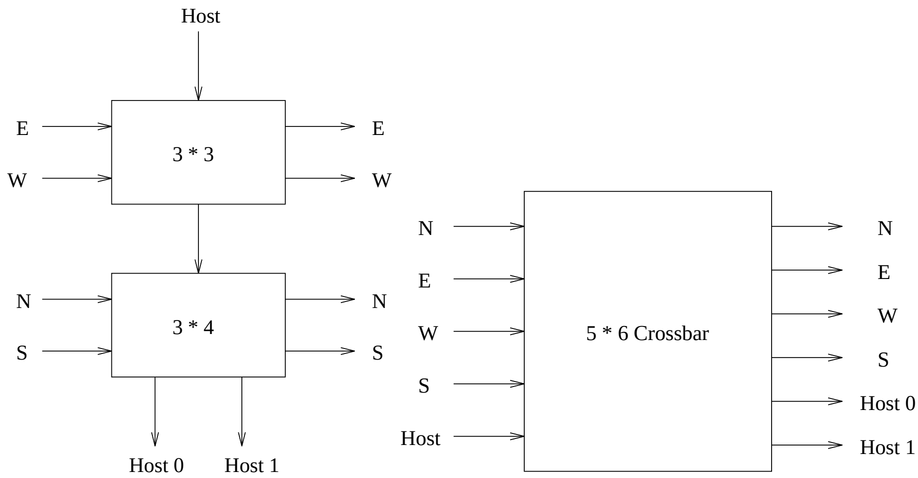
When there is a packet passing through the switch and the corresponding output FIFO is full, the arbiter associated with this output port will be notified. A disable signal (Enable=0) will be sent from the arbiter back to the header decoder of the input port from which the packet came. The disable signal stops the packet until there is space available at the output FIFO. When the transmission of the packet finishes, a clear signal is sent to the arbiter to reset the switch, and the arbiter is ready for the new request.

7.4.3 Crossbar Switch Architecture

The crossbar is a fully-connected 5×6 switch used to set up a physical connection path from an input port to any output port. Each input or output port in the crossbar is 18 bits wide (16-bit data plus 2-bit control) with the format of a flit. One extra output port to the local host provides 32 bits per cycle bandwidth ($2 \times$ 16-bit data width) in order to reduce host bottleneck. To support multicasting, an input port is able to connect to more than one output port. All connected paths can transfer data simultaneously.

Fig. 7.10 shows two different crossbar architectures: a two-stage CR and a fully-connected CR. Table 7.3 summarizes the comparison of these two CR architectures. The two-stage CR has a smaller area. However, it has strict routing limitation , *i.e.*, it can only route data from E or W to N or S , but cannot route data in the reverse directions (from Y to X)[2]. Due to this limitation, the two-stage CR cannot support path-based multicasting. In addition, the 2-stage CR also has less parallelism, *e.g.*, packets may get blocked even though there is no output conflict. Therefore, we chose to implement the fully-connected CR and pay the area penalty. More details of crossbar switch implementation will be discussed later (Section 7.7).

[2]For dimension-order unicasting routing, the routing limitation is not a problem because Y to X routing never occurs.



(a) 2-stage CR (b) Fully-Connected CR

Figure 7.10: Two different crossbar architectures

	2-stage CR	Fully-Connected CR
Area	Small	Large
Connectivity	Cannot route from Y to X	Provide all connections
Delay	Longer critical path	Same for all connections
Multicasting	No	Yes
Parallelism	Less	More

Table 7.3: Comparison of 2-stage CR and fully-connected CR

7.5 Router Timing

The main design goal of the router is to make the fall-through latency as small as possible while maintaining the throughput at one flit/cycle/port. In this section, we will show the details of our timing design and latency-hiding technique.

7.5.1 I/O Timing

The I/O control timing includes I/O channel timing for data transmission and token-exchange timing for channel arbitration.

I/O Channel Timing

Fig. 7.11 shows the I/O channel timing. In cycle 1, Out_state_s1, which changes from *Idle* to *Sending*, triggers fifo_out and RA(req), and puts data (Flit 1) on the channel. In cycle 2, In_state_s2 of the receiving node changes from *Idle* to *Receiving* due to RA(req)=HIGH. The channel data will be latched by data_in signal. If the input FIFO of the receiving node is full, RA(ack) will become HIGH to stop the sender; Out_state_s2 becomes *Idle* and RA(req) becomes LOW as shown in cycle 3. The sender can resume transmission in cycle 4 when it sees RA(ack) is LOW indicating that there is space available in the input FIFO. When the sender finishes, Out_state_s1 returns to *Idle* and RA(req) becomes LOW. The receiver also returns to *Idle* to complete the transaction in cycles 5 and 6.

Without traffic contention, the input FIFO is never full, the data throughput is one flit/cycle, and RA(req) only changes at the packet header, which goes from LOW to HIGH, and at the tail, which goes from HIGH to LOW. Hand-shaking is in the packet level. A second hand-shaking signal, RA(ack), is necessary for contention control to prevent buffer overflow and packet loss.

Token-Exchange Timing

In order to prevent conflict, token exchange is used for channel arbitration to determine which can use the channel. There is a one-bit signal TE between two neighboring nodes. Fig. 7.12 shows the circuit of the token-exchange interface. Both nodes are listening to TE all the time, but only the node whose Tstate_s2 is either *no_token* or *with_token_req* state can drive TE. That is,

$$\text{flag\_enable}=(\text{Tstate\_s2}==\textit{no\_token}) \;||\; (\text{Tstate\_s2}==\textit{with\_token\_req})$$

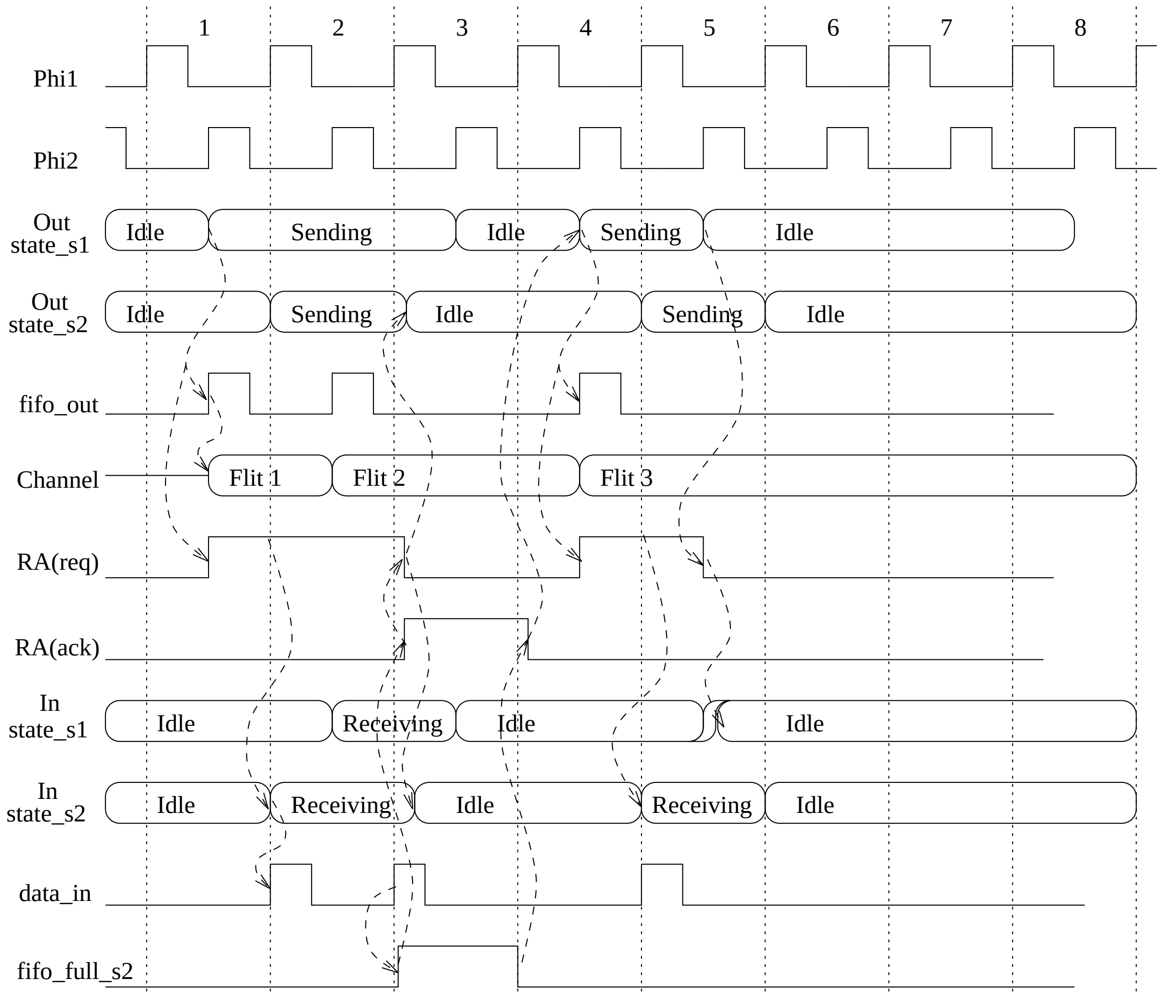


Figure 7.11: I/O control and channel timing diagram

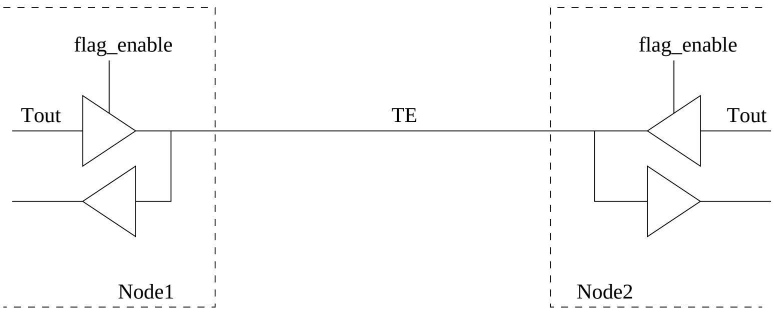


Figure 7.12: Token-exchange interface circuit diagram

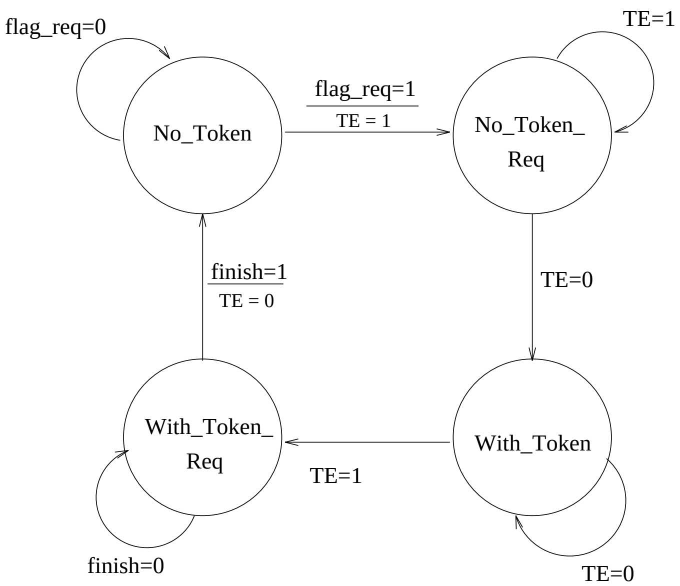


Figure 7.13: Token-exchange state diagram

The flag_enable signal controls the tri-state driver of T_out in Fig. 7.12. Fig. 7.13 shows the token-exchange state diagram; Fig. 7.14 is the timing diagram. Details of token-exchange algorithm were described in Section 3.4.1. As shown in the timing diagram, TE is triggered by node1_Tout or node2_Tout depending on their flag_enable states. It takes two cycles for token exchange (from Phi_2 of cycle 1 to Phi_2 of cycle 3, Fig. 7.14). We will show how to hide this token-exchange latency in the next section (router core timing). In the absence of traffic contention, there is no penalty for token-exchange arbitration in our design.

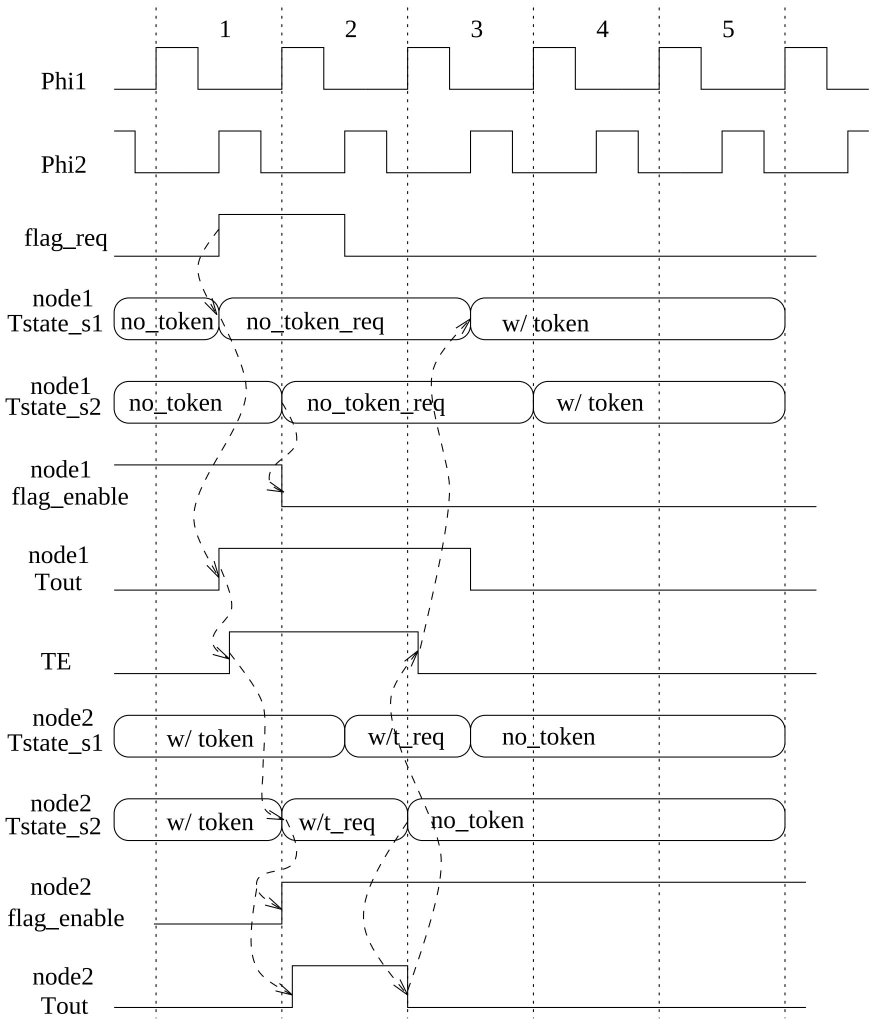


Figure 7.14: Token-exchange timing diagram

7.5.2 Router Core Timing

There are three major pipelining stages in the router core: header decoder, arbiter, and crossbar. We will describe their detail timing in the following paragraphs.

Header Decoder Timing

The header decoder takes a flit from the input FIFO when both data_in_s2 and enable_s2 are HIGH (Fig. 7.15). In cycle 1, if the incoming flit's head bit=HIGH, decoding circuitry will send req_to_Arb=HIGH to the requested output port. When grant_from_Arb is HIGH in cycle 2, the decoder will lower the req_to_Arb signal to indicate that the path to the output port has been established. If the output FIFO buffer is full due to traffic contention, grant_from_Arb will become LOW and the enable signals also become LOW; the header decoder will stop reading new flits from the input FIFO as shown in cycles 3 and 4. When the blocking at the output port is removed, grant_from_Arb and enable become HIGH again, and the header decoder resumes. When the tail of the packet arrives at the header decoder in cycle 5, the clear_sw signal is HIGH one cycle later to clear the arbiter, crossbar, and reset grant_from_Arb=LOW at the end of cycle 6 to complete this packet.

Arbiter Timing

Fig. 7.16 shows the arbiter timing diagram. In cycle 1, the arbiter receives the request from a header decoder,[3] and checks its internal switch_state. If the switch_state is LOW (the switch is free now), it can give grant to both header decoder and crossbar to establish the path in cycle 2. When the output FIFO is full (output_full=HIGH), the arbiter will disable the grant and enable signals to the header decoder and crossbar to stop further transmission of data until output_full is LOW as shown in cycles 4 and 5. When a clear_switch signal is received by the arbiter, it resets its switch_state and clears all grant signals in cycle 6.

Crossbar Switch Timing

The crossbar is a physical switch for establishing paths from any input port to any other output port(s). One input port may be connected to more than one output port when the packet is a multicasting packet and the current node is one of the destinations.

[3] In fact, the arbiter may receive more than one request from different header decoders. Then the round-robin priority arbitration is used.

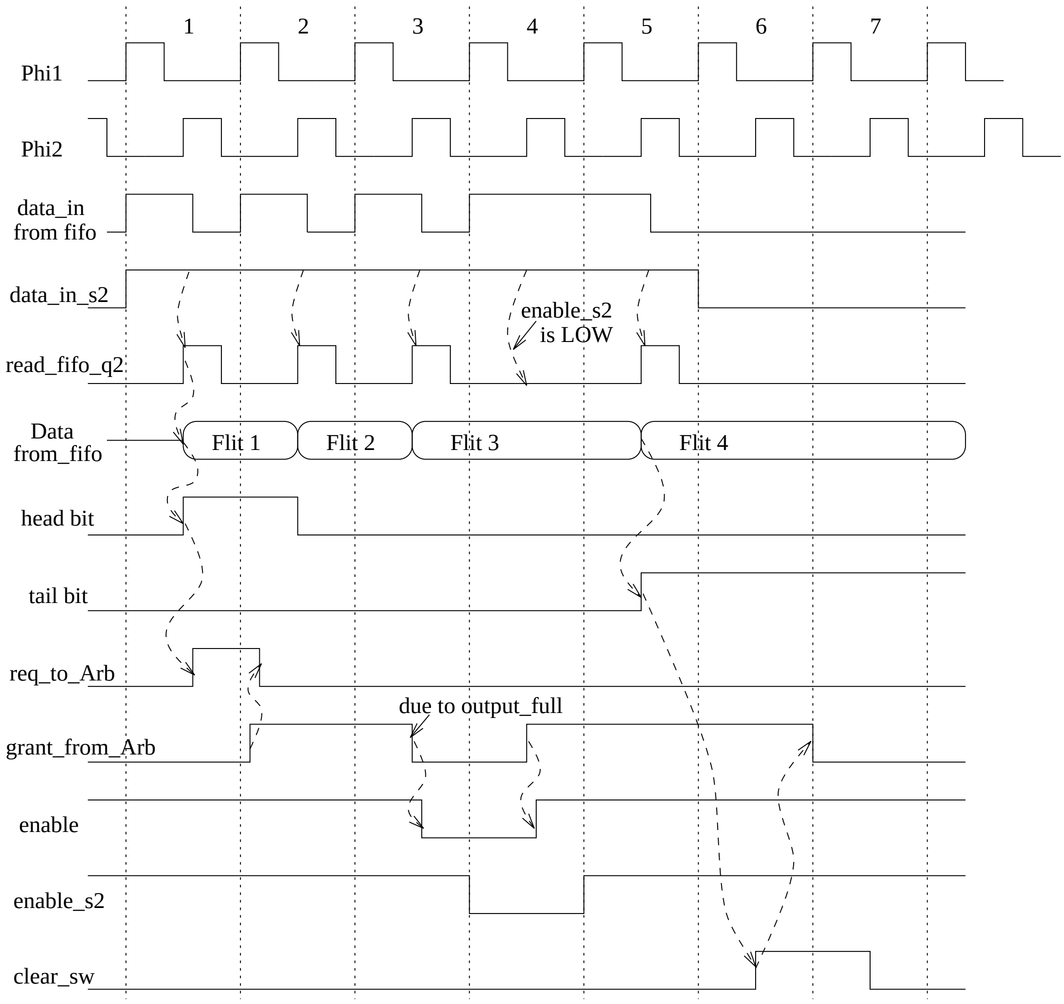


Figure 7.15: Header decoder timing diagram

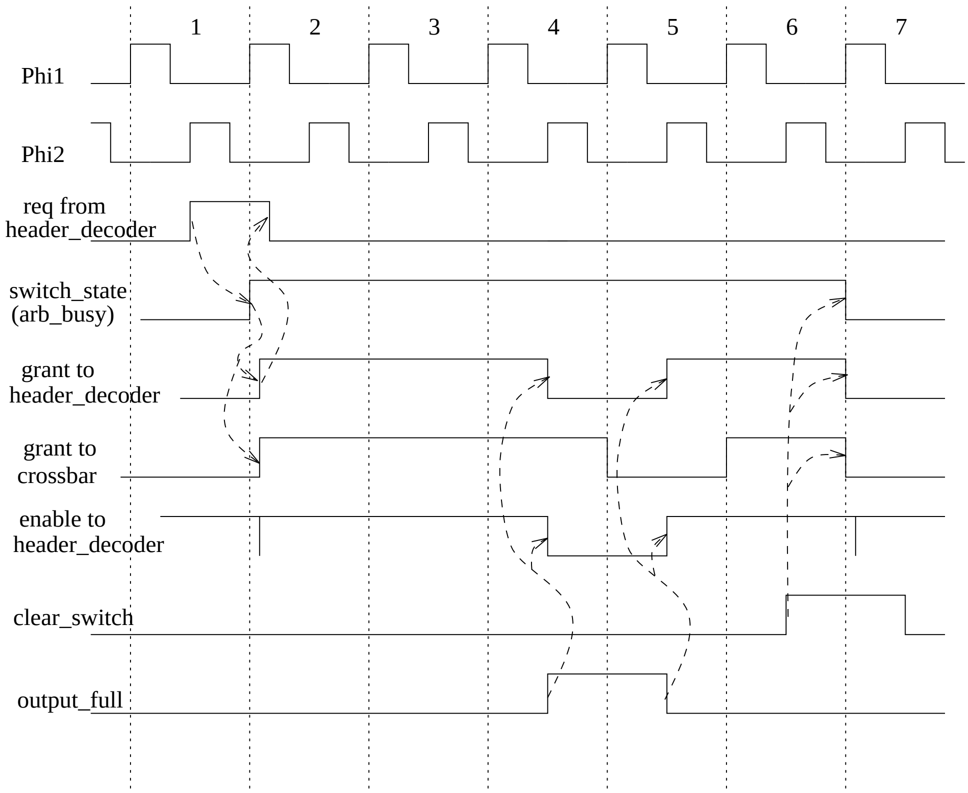


Figure 7.16: Arbiter timing diagram

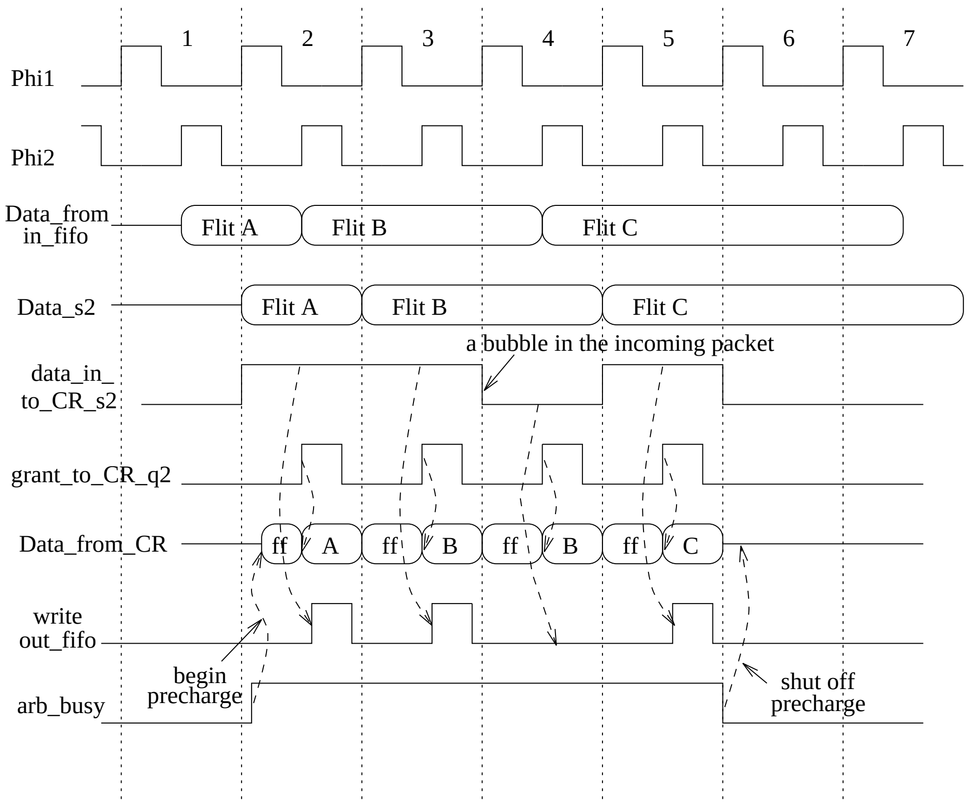


Figure 7.17: Crossbar timing diagram. ff in Data_from_CR is the precharged value.

Fig. 7.17 shows the timing of the crossbar switch. Data from the input FIFO is latched once before entering the crossbar (Data_s2). Precharge circuits are used in the crossbar; details of the circuit issues are in Section 7.7.2. The crossbar begins to precharge when arb_busy from the arbiter is HIGH because we know some packets are going to use the crossbar in the next cycles. When arb_busy=HIGH, the circuits are precharged during Phi_1, and evaluated during Phi_2 controlled by grant_to_CR_q2. Data is latched in the output FIFO at the falling edge of Phi_2 by write_out_fifo signal. The crossbar shuts off precharge when arb_busy=LOW to save stand-by power.

Router Core Timing

Based on the timing diagrams of the functional blocks described in the previous paragraphs, we can summarize the timing of the router core as shown in Fig. 7.18. The router fall-through latency is 3 cycles (from In_channel to Out_channel), and the throughput is one flit/cycle.

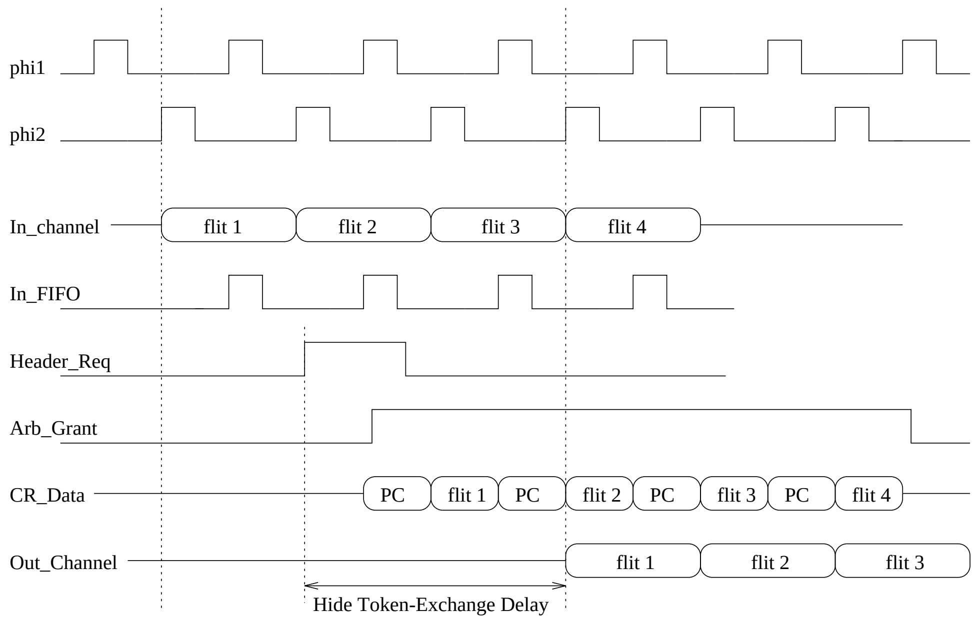


Figure 7.18: Router core timing diagram

From Fig. 7.18 timing diagram, we observed that Header_Req is two cycles earlier than Out_channel. It is known which output port will be used two cycles before the data actually arrives. Also from Fig. 7.14, the token-exchange timing diagram, we know there are two cycles delay for token exchange. Therefore, the token request can be pre-issued at the Header_Req cycle; if there is no contention, the node receives the token two cycles later and sends out the data without delays associated with the token exchange process.

7.6 Router Instruction Sets

We defined a set of instructions for STARP to perform data routing. There are two groups of instructions for the router: one for setting up the router status, called status instructions, and the other for generating data packet, called packet instructions. Each instruction is 16 bits with 4-bit opcode (bits 15:12). Depending on the opcode, different fields of operand (bits 11:0) have different meaning.

7.6.1 Status Instructions

Status instructions are used to set up the router states, for example, I/O tokens, test mode, and so on. The opcode for the status instructions is "ROUT". The fields of operand are defined as follows:

$$\underbrace{11}_{M} - \underbrace{10}_{T} - \underbrace{09}_{S1} - \underbrace{08}_{S2} - \underbrace{07 - 06}_{S} - \underbrace{05 - 04}_{W} - \underbrace{03 - 02}_{E} - \underbrace{01 - 00}_{N}$$

$$\begin{array}{lllll} M & = & 0, (S1, S2) & = & \text{don't care} \quad \text{Multicasting OFF} \\ M & = & 1, (S1, S2) & = & \text{don't care} \quad \text{Multicasting ON} \\ T & = & 0, (S1, S2) & = & (1, 0) \quad \text{set token mode} \\ & & & & (1, 1) \quad \text{set ID mode} \\ T & = & 1, (S1, S2) & = & (0, 0) \quad \text{SW\_set mode} \\ & & & & (0, 1) \quad \text{SW\_reset mode} \\ & & & & (1, 0) \quad \text{Host\_reset mode} \\ & & & & (1, 1) \quad \text{Host\_reset mode} \end{array}$$

Multicasting ON or OFF is set in SrOut control only. It affects only the generation of outgoing packets from the host. The router can handle any kind of packets independent of the status of SrOut control. Table 7.4 summarizes the functions and operations of each mode described above. Some notes are given as follows:

- When we initialize the I/O port token state for several routers connected in a network, we must assume that each channel can be initialized with only one token.

- In SW_set mode, the decoded results R_x and R_y (Eq. 7.1) are overwritten by the pre-set values R_{xt} and R_{yt} specified in the fields of operand corresponding to the I/O port.

- Similarly, in Host_set mode, the decoded results R_x and R_y are overwritten by the pre-set values R_{xt} and R_{yt} specified in the operand bits 1:0.

- Either SW_set mode or Host_set mode will turn on the loop-back testing mode of the router. The details of loop-back mode will be given in Section 7.8.

Mode	Operations
set token	Set I/O port token state bits [7:0] are associated with 4 I/O ports (NEWS) as specified. (00) is no token and (11) is with token state
set ID	Set the local node address bits [7:4] and [3:0] are the x and y address, respectively.
SW_set	Set router switches to the test mode. The header decoders in the I/O ports (NEWS) will be disable. Decoded results are pre-set by this instruction in bits [7:0]
SW_reset	Reset router switches to the normal mode.
Host_set	Set host switch to the test mode. The header decoders in the host port is disable. Decoded results are pre-set in bit [1:0]
Host_reset	Reset host switch to the normal mode.

Table 7.4: Summary of status instructions modes

7.6.2 Packet Instructions

The outgoing packets are generated at SrOut by writing some packet information, for example, destination address, memory location, packet length, into the system registers. The packet instructions are used to write these system registers. The opcode for the packet instructions is "LSRI". The fields of the operand contains a 4-bit register ID and an 8-bit data.

There are four system registers designed for router packet instructions. Each system register is 8 bits wide. The functions for the system registers are listed as follows:

Register ID	Data stored
reg#0	dest[7:0]
reg#1	dest[15:8]
reg#2	local[7:0]
reg#3	local[15:8]

The corresponding header format is :

Header[2:0]	=	dest[2:0];	// x-address
Header[6:3]	=	4'b0000;	
Header[7]	=	dest[3];	// Dx
Header[10:8]	=	dest[6:4];	// y-address
Header[14:11]	=	4'b0000;	
Header[15]	=	dest[7];	// Dy
MemAdd[3:0]	=	4'b0000;	
MemAdd[11:4]	=	dest[15:8];	
MemAdd[15:12]	=	4'b0000;	

In the STARP prototype, we can specify only 3-bit x and y addresses for the packet destination, although the maximal addressable range in a header is 7-bit in each dimension. In a full implementation, it would be necessary to increase the number of system registers or increase the size of each register.

System registers #2 and #3 are used for local host memory address and packet length. Source mem_address and SrOut counter are defined as follows:

Source mem_address[11:2]	=	local[9:0];
SrOut counter	=	local[15:10];

Source mem_address is the memory location at the source node where the data of the outgoing packet is stored. SrOut counter is equivalent to the packet length. The counter is decremented by one whenever the memory is read until the counter value reaches zero. The counter is 6-bit wide, and every memory access corresponds to two flits, so the maximal packet length is 128 data flits.

7.6.3 Instruction Examples

A simple example of the router instructions is shown below. In this example, a unicasting packet is generated in node (0, 0) with all testing modes set. The packet has 16 data flits plus 2 header flits (*Header* and *MemAdd*). The packet data is read from local memory bank R0 address 0. The destination of this packet is node (0, 1) with memory location at bank B address 4.

```
ROUT 300   // set ID_x=0, ID_y=0
ROUT 2ff   // set with token
ROUT 602   // H_set, Rx_test=0, Ry_test=1
ROUT 400   // SW_set, Rx_test=Ry_test=0 for all ports, multicast off
LSRI 2 00  // source mem addr = R0 @ 0
LSRI 3 20  // Source counter = 8
LSRI 0 10  // dest: x=0, Dx=0, y=1, Dy=0
LSRI 1 81  // dest: bank B @ 4
```

The next example shows a multicasting packet with two destinations: dest1 node (0, 0) and dest2 node (1, 1). The destination memory locations are bank A address 0 and bank C address 0, respectively.

```
ROUT 602   // H_set, Rx_test=0, Ry_test=1
ROUT d00   // reset SW, multicast on
NOP
LSRI 0 80  // dest1: x=0, Dx=0, y=0, Dy=1
LSRI 1 40  // dest1: bank A @ 0
LSRI 2 81  // Source mem R1 @ 4
LSRI 3 10  // Source counter = 4
LSRI 0 99  // dest2: x=1, Dx=1, y=1, Dy=1
LSRI 1 c0  // dest2: bank C @ 0
```

7.7 Chip Implementation

In the implementation of STARP, we combined standard cell design and custom design. In this section, we describe our design methodology and discuss some circuit issues.

7.7.1 Design Methodology

Fig. 7.19 shows our design methodology of STARP. We started from the high level C language model for algorithmic and architectural trade-off study. At this level, we could evaluate the performance of different architectures, for example, channel configurations, routing algorithms, buffer size, etc., because simulation efficiency is high. After finalizing the architecture and other high level details, we began the hardware modeling and simulation in Verilog, a Hardware Description Language. The whole chip Register-Transfer Level (RTL) model was created in Verilog. We verified design details: functionality, timing, interface, and complexity. We can also achieve a better view of block partitioning of the chip in this phase of the design.

The physical design has two parts: custom blocks and standard cells synthesis. The custom blocks include memory, datapath, and crossbar switch. We designed, laidout (with MAGIC), and simulated (with HSPICE) these blocks to optimize performance and minimize area. We also designed and laidout our own standard cell library with separate well biases to improve low voltage performance (Section 7.7.2). All the cells were simulated and calibrated by HSPICE. These cells were used to synthesize the control logic blocks by Synopsys which reads in the Verilog HDL codes. A silicon compiler (Lager) was used to generate the layout from the synthesis results. All the blocks were extracted into SIM files which is the transistor level with RC model to verify functionality and timing (IRSIM). Global routing put all the blocks together following design rule checks (DRC), Vdd and GND checks, and a well check (to prevent floating wells), etc. The entire chip was extracted again into a SIM file for IRSIM simulation.

At each level of simulation, test vectors were created by Snooper from Verilog. Snooper records all the changes of input and output signals of a block, or an entire chip, as the stimulus and assertion of simulation. Final chip test vectors were created with Snooper for chip testing.

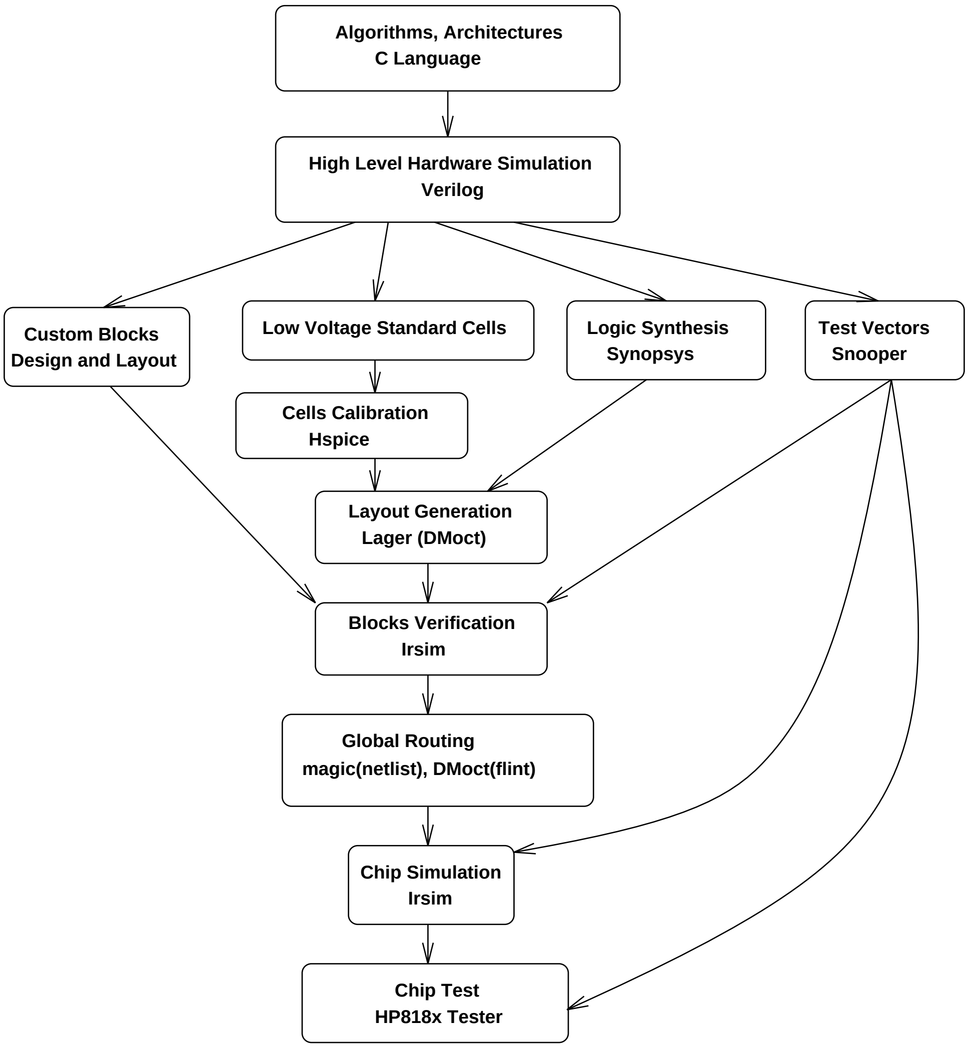


Figure 7.19: Design methodology and flow of STARP

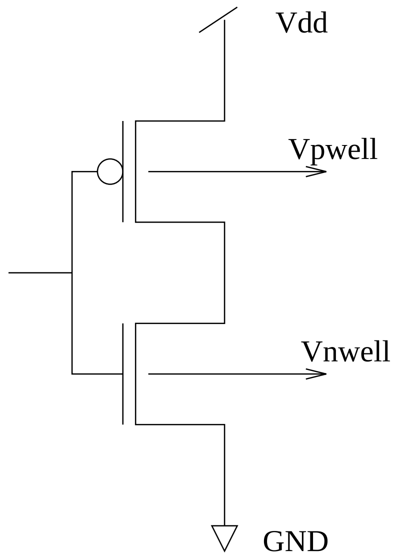
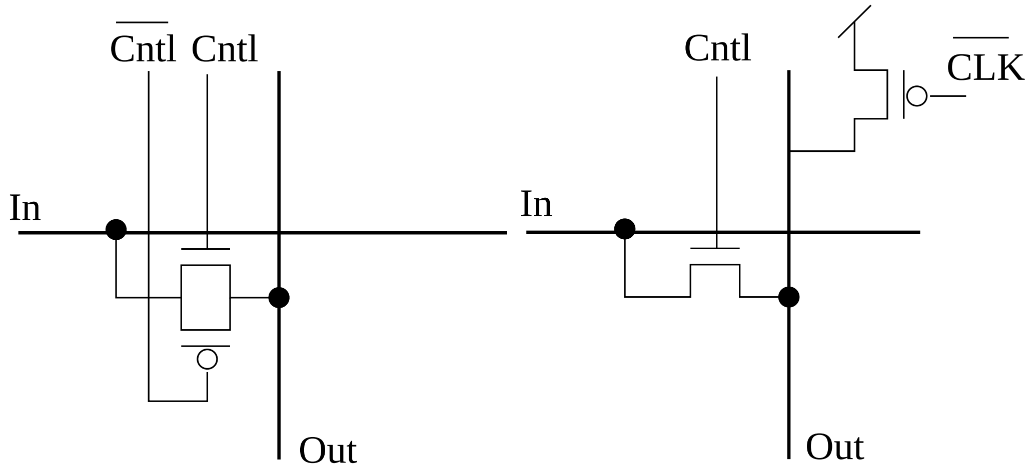


Figure 7.20: CMOS inverter model. Vpwell and Vnwell can be biased separately from Vdd and GND to improve the performance at low Vdd

7.7.2 Circuit Issues

We used the Stanford Ultra Low Power (ULP) CMOS technique at the device level (Burr and Shott, 1994) to design the STARP chip. Fig. 7.20 shows a CMOS inverter model. In the traditional CMOS design, Vpwell is connected to Vdd, and Vnwell is connected to GND to guarantee that the diffusion and substrate junction is reverse-biased. In our approach, we have separate well biases: Vpwell and Vnwell have separate contacts, distinct from Vdd and GND, allows separate biases to be applied to the diffusion and substrate junctions. According to the body effect, the transistor threshold voltage V_{th} will be reduced when a forward bias is applied to the diffusion and substrate junction. Performance of transistors can be enhanced due to the reduction of V_{th}.

The well bias must be controlled carefully to prevent latch-up. Too much forward bias to the junction will result in a large current flowing through the substrate inducing latch-up. A slightly forward biased junction (V_bias $<$ 0.5V) is helpful to reduce gate delay and diffusion capacitance. But the current leakage because of lower V_{th} will cause some increase in DC power, and even malfunction in some dynamic circuits. Consequently, it is necessary to restrict the circuit styles in designing a low power VLSI system using the ULP technology.



(a) Crossbar with static circuit (b) Crossbar with dynamic circuit

Figure 7.21: Crossbar switch circuits. 1-bit switching unit implemented in static and dynamic circuits

Crossbar Switch

Most of the router circuits were synthesized using Synopsys. But the crossbar switch (CR) was fully custom designed and laidout because it is very regular and needs to be very compact.

Fig. 7.21 is the basic crossbar switch unit implemented in two different circuit styles, one for static and another for dynamic circuits. Fig. 7.21(a) shows the static transmission gate switch with both PMOS and NMOS. Fig. 7.21(b) is the dynamic switch with NMOS pass gate and PMOS precharge. In practice, because there are several NMOS pass gates on an output bit line, we put a small PMOS keeper at the output line for the dynamic CR to prevent the leakage through these NMOS and accidental discharge of the output line (Fig. 7.22). Many references in VLSI design provide a thorough discussion of static and dynamic logic, *e.g.*, (Weste and Eshraghian, 1985). Basically, dynamic logic requires fewer transistors, smaller area, and less capacitance loading.

After physically layingout the crossbar circuits, we found that the CR area is dominated by wires; therefore we hid the transistors beneath the wires, and the size of the resulting circuits is determined by the number of wires. The wires include input, output, and control signals. Fig. 7.23 shows the floor plan of the crossbar switch. The area difference between

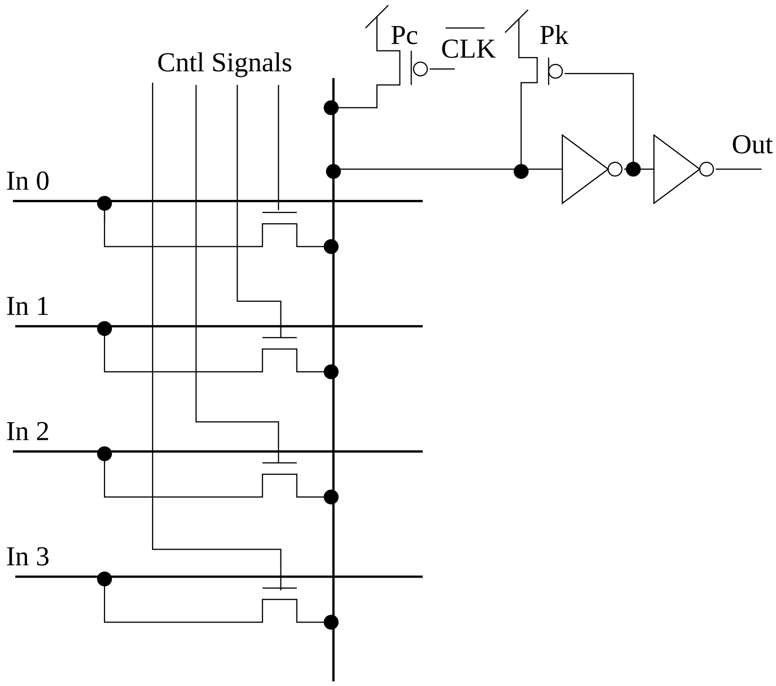


Figure 7.22: Dynamic crossbar switch with a PMOS keeper at the output to prevent leakage

the static and dynamic CR results from the need for twice the number of control wires for the static CR. So we estimate the area of a general CR as follows:

$$\text{Area} = (m + \frac{c}{2} + \Delta) \times (b \times n + c) \times b/2 \quad w^2$$

where m and n are the number of input and output ports, respectively. The remaining variables are as follows: Δ is the precharge overhead, c is the number of control signals, b is the number of bits per port, and w is a single wire pitch. Therefore, the area estimates for the 5×6 crossbars are:

$$\text{Area\_dynamic} = (5 + 10 + 5) \times (18 \times 6 + 20) \times 9 = 23040 \cdot w^2$$

$$\text{Area\_static} = (5 + 20) \times (18 \times 6 + 40) \times 9 = 33300 \cdot w^2$$

where $m = 5$, $n = 6$, $b = 18$, and $\Delta = 5$, $c = 20$ for the dynamic case and $\Delta = 0$, $c = 40$ for the static case. The area of the dynamic CR is about 30% less than that of static CR. Therefore, we decided to implement the dynamic CR in order to compensate for the larger area of a fully connected CR .

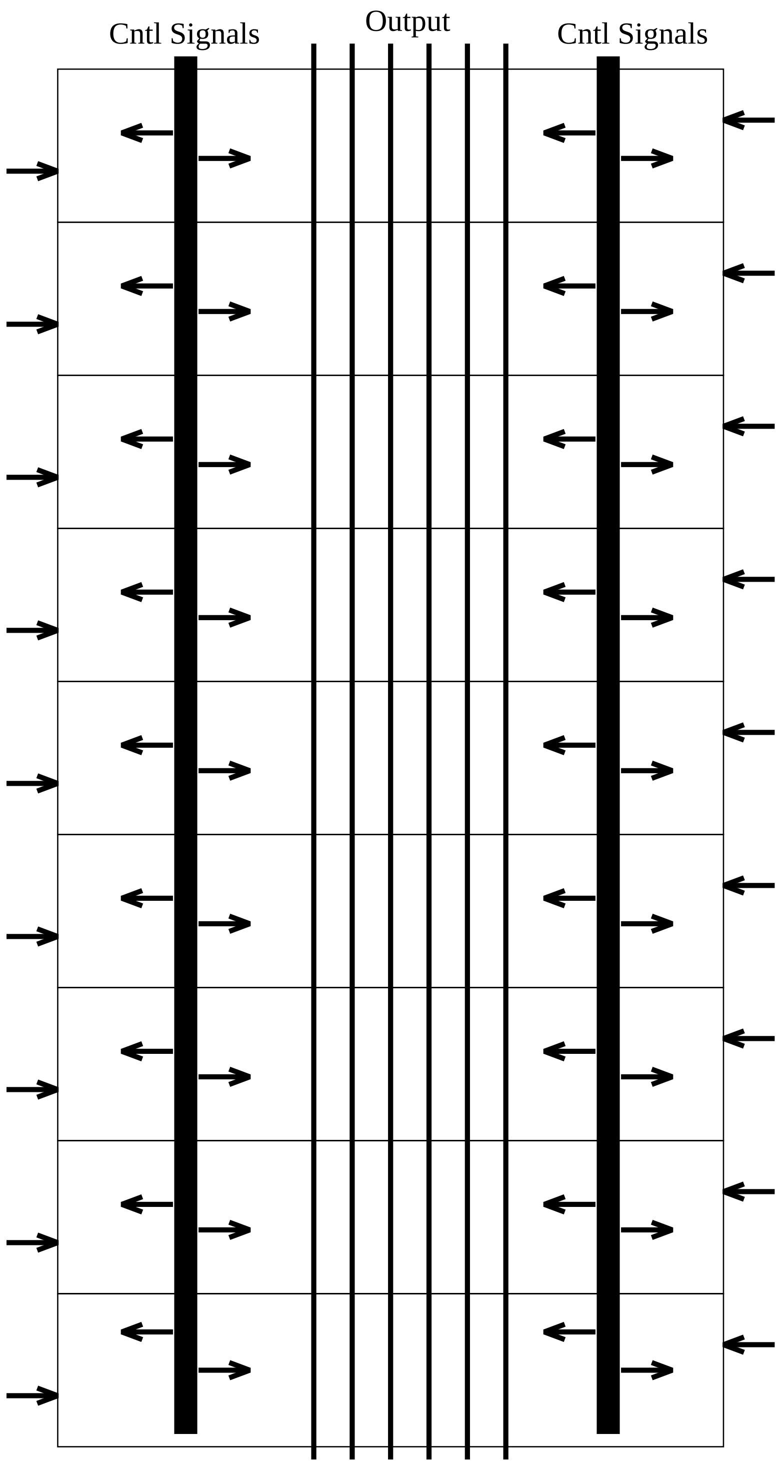


Figure 7.23: Crossbar switch floor plan

7.7.3 Chip Fabrication

STARP was fabricated using MOSIS/HP 0.6μm CMOS technology. The die size is $6mm \times 4.8mm$ containing about 250,000 transistors. The wormhole router occupies 30% of the die area and has about 50,000 transistors. The STARP chip was packaged in a 132-pin PGA. Due to the limit of pin count, only one I/O port has real I/O pads off chip. Fig. 7.24 shows the chip micrograph.[4]

7.8 Testing Issues

Testability is an important issue in VLSI design. We have built in a scan path which allows us to access the internal states of the chip. Each part of the chip (router, memory, and datapath) can be tested separately to isolate problems if any. We also have several testing modes which allow us to simplify the testing procedures.

7.8.1 Loop-back Mode Testing

The loop-back mode testing was designed for single chip testing. The purpose is to emulate incoming packets from the network. Fig. 7.25 shows the diagram of loop-back testing. All the header decoders can be disabled, and the decoding results can be pre-set by instructions (Section 7.6). Therefore, by using the status instructions, we can easily test all I/O ports and all multicasting combinations.

The loop-back mode is on when either SW_set or Host_set mode is on. In the loop-back mode, the I/O port will "hand-shake" with itself. Outgoing packets from an output FIFO are looped back to the input FIFO of the same I/O port with proper control signaling to simulate incoming packets from the network. We also can introduce network traffic contention by disabling the channel token, and the packet should stop in the FIFOs while waiting for the token. Then we can test the router behavior under traffic contention. The loop-back packets can be stored in the local memory and read out later for debugging.

7.8.2 Tester Setup

We used an HP8180 data generator and HP8182 data analyzer as our testing platform. We constructed a test board to connect with the HP testers. The high frequency signals on the

[4]The MAD_Unit and memory banks were designed by Gerard Yeh.

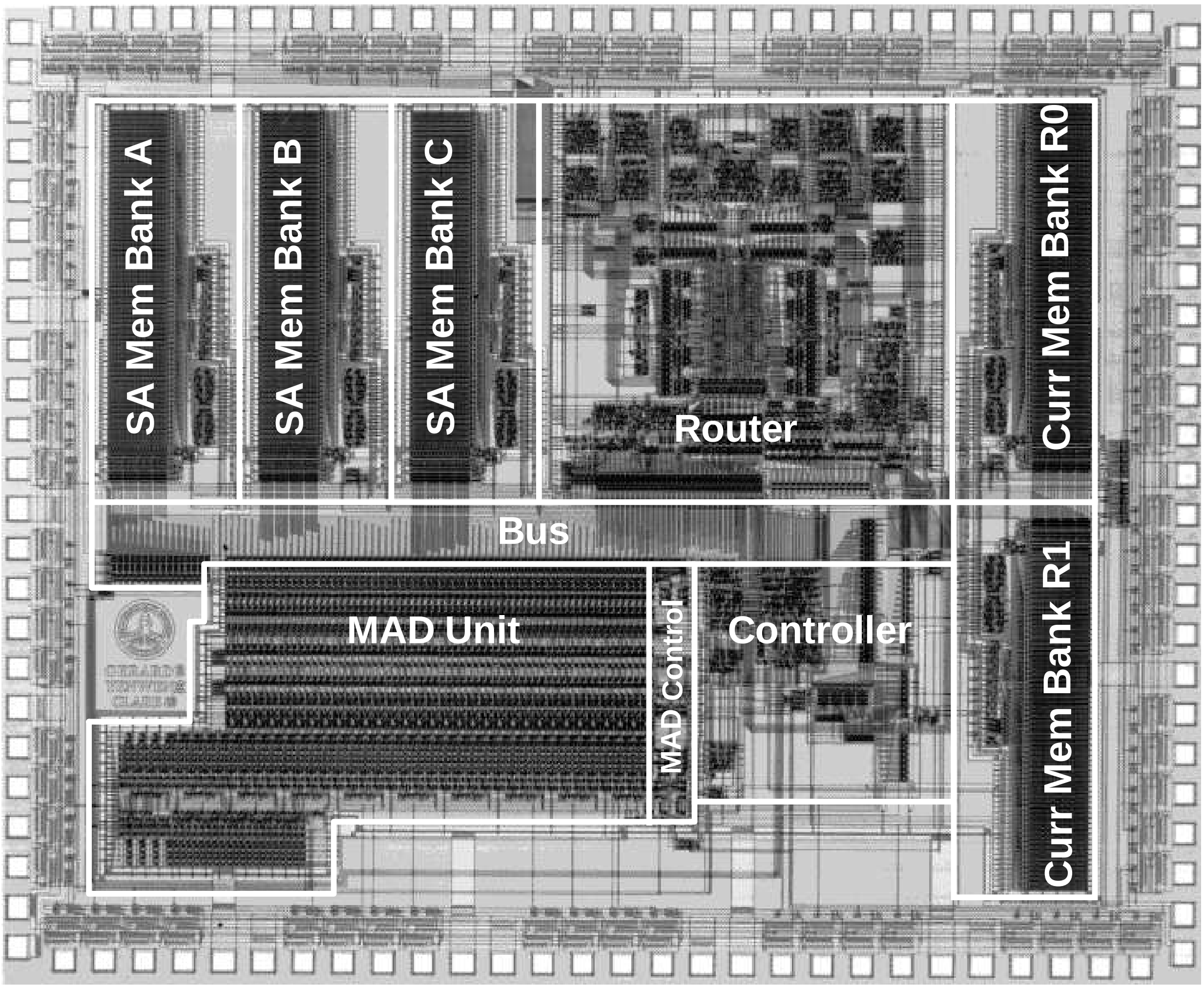


Figure 7.24: STARP chip micrograph

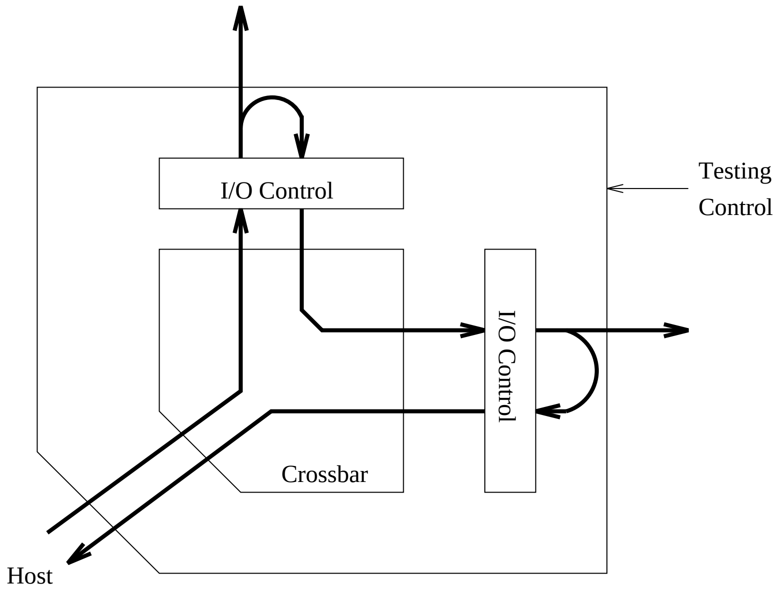


Figure 7.25: Loop-back mode test

board (clocks, input instructions, result data bus, and router I/O channels) are matched and terminated. A PC served as the host to upload testing vectors and download test results to and from the HP tester via GPIB interface. A translation program was written to convert IRSIM test vectors for the HP tester.

7.8.3 Testing Results

The STARP chip was tested and verified. There were about 20 test vectors (each vector has 50 to 300 instructions) for functionality and performance tests. STARP was fully functional for all test vectors. Performance tests were conducted with a wide range of frequency and power supply. Separate well bias effects were also tested for low Vdd.

Fig. 7.26 shows the performance testing results. Each point was measured at different Vdd and the maximum frequency achievable for the given Vdd. In testing, we can achieve 50MHz at Vdd=2.1V.[5] We achieve this performance because we have well-pipelined and controlled critical paths in our design. Fig. 7.26(a) is the plot of average power versus frequency. We measured only the router core (including host interface and bus) power because the interconnection power is very dependent on the physical interconnection medium. We tested both unicasting and multicasting (with 2 destinations) with loop-back mode. There are more

[5] 50MHz is the maximum frequency in the HP testers

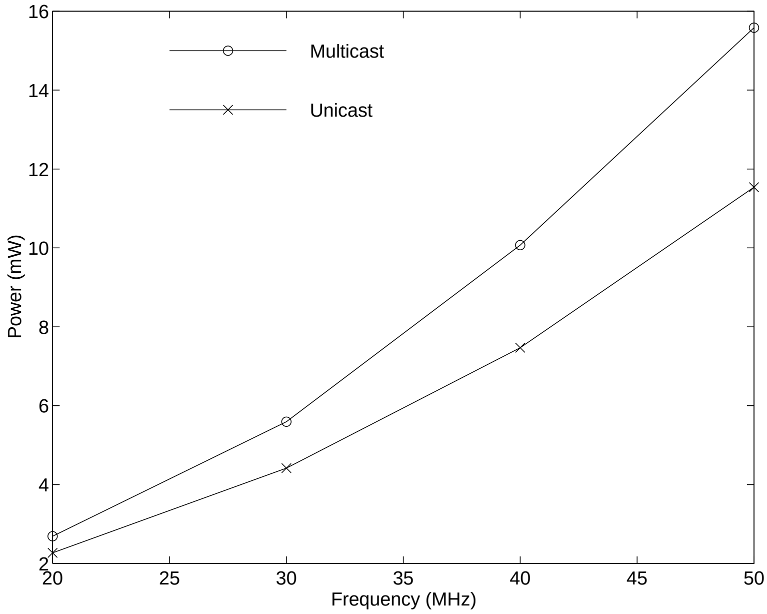
activities in both control and crossbar switch for multicasting packets, so the average power for multicasting is higher than unicasting. But multicasting can deliver more data bandwidth compared with unicasting. Data rate is equal to frequency $\times$ (# of dest) $\times$ 16bits. So at 50MHz the peak data rate = 1.6Gb/s for multicasting. Fig. 7.26(b) shows the plot of energy/bit versus data rate. There is an obvious energy advantage of multicasting over unicasting. Multicasting supports a higher data rate with lower energy per bit because there is more resource sharing. So multicasting not only reduces network traffic contention, but also reduces communication energy.

Fig. 7.27 shows the plot of energy $\times$ delay ($E \cdot T$) versus Vdd. $E \cdot T$ is a metric for optimization trade-off. We wish to minimize the energy while maintaining reasonable performance. Again, multicasting has lower $E \cdot T$ curve than unicasting because multicasting is more energy-efficient with higher date rate.

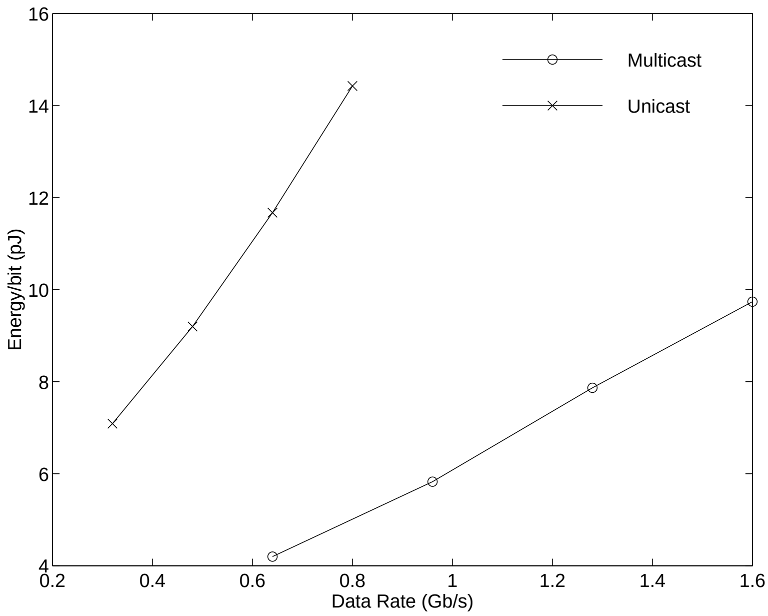
Fig. 7.28 shows the performance improvement versus well bias. When we increase the well bias, we forward bias the transistor junctions and reduce the threshold voltage V_{th}. The y-axis in the plot is the frequency improvement normalized to the frequency without separate bias. The improvement is very linear with the bias voltage. We have larger improvement when we reduce the supply voltage Vdd because we have a more significant difference in (Vdd-V_{th}) when we change V_{th} for small Vdd. So this low power technique is especially good for low voltage operation. For Vdd=1.3V, we achieve more than 30% improvement with bias=0.4V.

7.9 Summary

Table 7.5 gives the allocation of the power and area for each block in the router. The power percentage number was estimated from IRSIM simulation. As we expected, clocks, crossbar switch, and FIFOs consume most of the power (67.7%) due to their high activity and heavy loading. The other significant power consumers are the Host_Interface and Bus drivers, for the same reason. The Arbiter uses particularly little power because of its low activity: it has transitions only when the switch is set or reset. From the column of areas, the Crossbar and FIFOs occupy the majority of router area although reduction of the Crossbar area received special attention (Section 7.7.2). The total effective area for the items listed is about 48% of the router area. The rest of the area is used for power distribution, clock distribution, and routing wires.



(a) Core power vs. Frequency



(b) Core energy vs. Data Rate

Figure 7.26: Chip measurement results

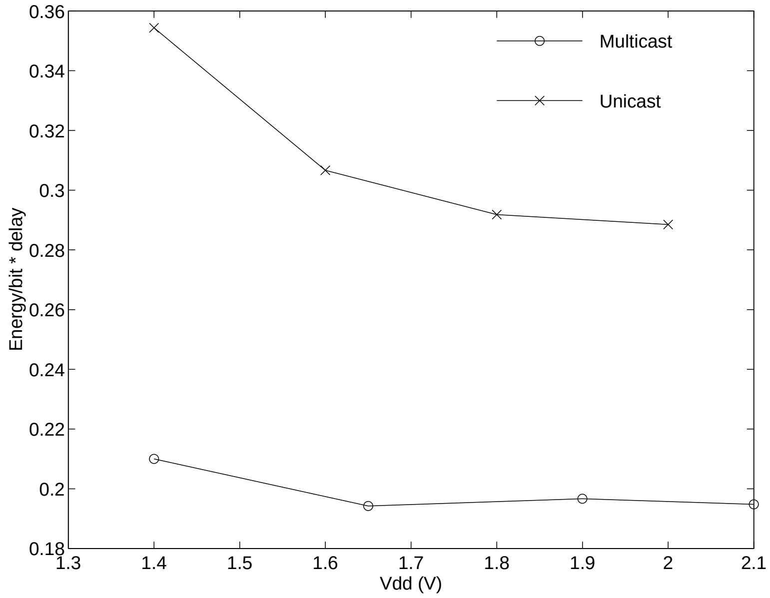


Figure 7.27: Core energy×delay vs. Vdd

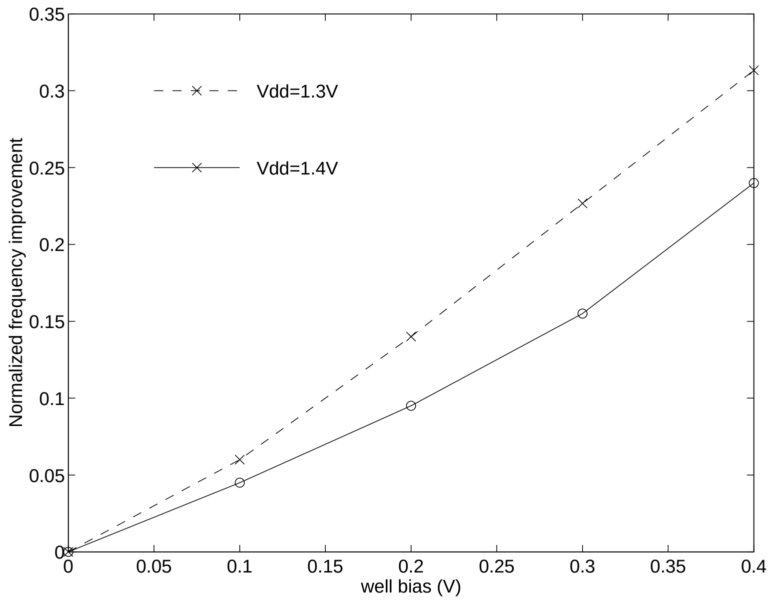


Figure 7.28: Performance improvement vs. well bias

	Power %	Area %
Clock	28.6%	
CR + FIFO	39.1%	24.4%
Host_Interface	21.9%	6.4%
Bus Drivers		4.0%
Header_Decoder	6.8%	4.75%
Arbiter	0.5%	5.23%
IO_Control	2.6%	3.01%
Total	100%	47.79%

Table 7.5: Estimated power and area allocation of the router in STARP

Feature	Path-based multicasting
	Bi-directional 16-bit data channels
	Token-exchange channel arbitration
	Deterministic routing
Latency	3 cycles/hop
Throughput	1 flit/cycle/IO port
Performance	50MHz @ Vdd=2.1V
	(MOSIS HP 0.6μm)
Ave. Power (Core)	15mW (measured)
Energy (Core)	20pJ/bit (measured)
Max. BW	1.6Gb/s

Table 7.6: STARP wormhole router summary

The wormhole router was fabricated using MOSIS/HP 0.6μm technology. It can deliver 1.6Gb/s (50MHz) @ Vdd=2.1V and consumes an average power of 15mW. Table 7.6 gives the summary of the wormhole router.

Chapter 8

Contributions and Suggestions for Future Work

8.1 Contributions

The principle contributions of this dissertation are:

1. A wormhole routing simulator was written to study different routing algorithms and some design tradeoffs: for example, buffer size, virtual channel numbers, channel arbitration, and so on. Four algorithms were implemented and simulated: Deterministic (dimension-order), Virtual Network, Dimension Reversal, and Star-Channel algorithm. Three different traffic patterns were included in the simulation: uniform, transpose, and hot-spot. In general, the adaptive algorithms perform better than the deterministic algorithm. The difference between algorithms depends on traffic patterns: for example, it is obvious that adaptive routing is superior for transpose traffic, but not significant for uniform traffic.

2. Different channel configurations (uni-directional and bi-directional channels) were simulated and compared. A token-exchange channel arbitration scheme for bi-directional channels was developed and implemented. The token-exchange scheme was proved to be conflict-free and deadlock-free. In spite of the channel arbitration overhead, simulation shows that bi-directional channels have significantly better latency-throughput performance, and can sustain higher data bandwidths relative to uni-directional channels of the same physical channel width.

3. An enhanced mesh (or torus) architecture with segmented reconfigurable bus (SRB) is proposed to overcome the delay due to long distance communication in a low-dimensional array. An optimization procedure to select the segment length and segment alignment has been formulated. The results of a theoretical analysis of SRB performance are consistent with the simulation results, and provide a guideline for designing a SRB torus network.

4. A low power, wormhole data router chip for 2-D mesh and torus networks with bi-directional channels and token-exchange arbitration was designed and implemented. In this design, the token-exchange delay is fully hidden with no increase in latency when there is no traffic contention. Distributed decoders and arbiters are provided for each IO port, and a fully-connected crossbar switch increases parallelism of data routing. The router also provides hardware support for path-based multicasting. From measured results, the multicasting communication with two destinations is three times as energy-efficient as unicasting. The wormhole router was fabricated using MOSIS/HP 0.6μm technology. It can deliver 1.6Gb/s (50MHz) @ Vdd=2.1V and consumes an average power of 15mW.

8.2 Suggestions for Future Work

Networks, routing flow control, routing algorithms, and router design are all crucial for efficient multiprocessor communications. There are many other critical components for a parallel system to achieve efficient inter-processors communication. For example, high-speed interconnection is essential for high-throughput and low-latency systems. Low voltage swing interconnection and optical interconnection have been seen as potentially useful for future systems.

Fault-tolerance is also an important issue for an efficient and reliable parallel machine. Routing data in a network with faulty nodes or channels is especially important in a large scale machine. Deterministic routing is more susceptible to faulty nodes and channels than adaptive routing. Complexity, redundancy, and efficiency are the tradeoffs for the router design.

Accurate traffic modeling can also give us insight into the network behavior and performance for different applications. Packet interference in different traffic patterns plays a significant role on overall throughput-latency performance. In addition to fast data routers and high speed communication links, a systematic method to generate a traffic flow with less contention for different problems is also desirable to improve data routing efficiency.

References

Agarwal, A. "Limits on interconnection network performance." *IEEE Trans. on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 398–412, Oct. 1991.

Bakoglu, H. B. and J. D. Meindl. "Optimal interconnect circuits for VLSI." *IEEE Trans. on Electron Devices*, vol. ED-32, pp. 903–909, May 1985.

Berman, P. E., L. Gravano, G. D. Pifarré, and J. L. C. Sanz. "Adaptive deadlock- and livelock-free routing with all minimal paths in torus networks." In *Proc. 4th ACM SPAA*, pp. 3–12, 1992.

Boppana, R. V. and S. Chalasani. "A comparison of adaptive wormhole routing algorithms." In *Proc. 20th Int. Symp. on Comput. Arch.*, pp. 351–360, 1993.

Burr, J. and J. Shott. "A 200mV Self-Testing Encoder/Decoder using Stanford Ultra-Low-Power CMOS." In *Proc. IEEE International Solid-State Circuits Conference*, pp. 84–85, 1994.

Chien, A. A. and J. H. Kim. "Planar-adaptive routing: low-cost adaptive networks for multiprocessors." In *Proc. 19th Int. Symp. on Comput. Arch.*, pp. 268–277, May 1992.

Dally, W. J. "Network and processor architecture for message-driven computers." In Suaya, R. and G. Birtwistle, editors, *VLSI and Parallel Computation*, pp. 140–222, San Mateo, CA., 1990. Morgan Kaufmann Publishers, Inc.

Dally, W. J. "Performance analysis of k-ary n-cube interconnection networks." *IEEE Trans. on Comput.*, vol. C-39, no. 6, pp. 775–785, June 1990.

Dally, W. J. "Virtual-channel flow control." *IEEE Trans. on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 194–205, Mar. 1992.

Dally, W. J. and H. Aoki. "Deadlock-free adaptive routing in multicomputer networks using virtual channels." *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 4, pp. 466–475, Apr. 1993.

Dally, W. J. and C. L. Seitz. "The torus routing chip." *Distributed Computing*, vol. 1, no. 3, pp. 187–196, Oct. 1986.

Dally, W. J. and C. L. Seitz. "Deadlock-free message routing on multiprocessor interconnection networks." *IEEE Trans. on Comput.*, vol. C-36, no. 5, pp. 547–553, May 1987.

Dally, W. J. and P. Song. "Design of a self-time VLSI multicomputer communication controller." In *Proc. Int. Conf. Computer Design*, pp. 230–234, 1987.

Draper, J. T. and J. Ghosh. "Multipath E-cube algorithms (MECA) for adaptive wormhole routing and broadcasting in k-ary n-cubes." In *Proceedings of 6th International Parallel Processing Symposium*, pp. 407–410, 1992.

Duato, Jose. "A new theory of deadlock-free adaptive routing in wormhole networks." *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 12, pp. 1320–1331, Dec. 1993.

Fen, T. Y. "A survey of interconnection networks." *IEEE Computer*, pp. 12–27, Dec. 1981.

Gaughan, P. T. and S. Yalamanchili. "A family of fault-tolerant routing protocols for direct multiprocessor networks." *IEEE Trans. on Parallel and Distributed Systems*, vol. 6, no. 5, pp. 482–497, May 1995.

Glass, C. J. and L. M. Ni. "The turn model for adaptive routing." In *Proc. 19th Int. Symp. on Comput. Arch.*, pp. 278–287, 1992.

Gravano, L., G. D. Pifarré, P. E. Berman, and J. L. C. Sanz. "Adaptive deadlock- and livelock-free routing with all minimal paths in torus networks." *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 12, pp. 1233–1251, Dec. 1994.

Gustafson, J. L. "Reevaluating Amdahl's Law." *Commun. ACM*, vol. 31, no. 5, pp. 532–533, May 1988.

Hennessy, J. L. and D. A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufnann, San Francisco, CA., second edition, 1996.

Hwang, K. *Advanced Computer Architecture.* McGraw-Hill, New York, 1993.

Kermani, P. and L. Kleinrock. "Virtual cut-through: a new computer communications switching technique." *Computer Networks*, vol. 3, no. 4, pp. 267–286, Oct. 1979.

Konstantinidou, S. and L. Snyder. "Chaos router: architecture and performance." In *Proc. 18th Int. Symp. on Comput. Arch.*, pp. 212–221, 1991.

Konstantinidou, S. and L. Snyder. "The chaos router." *IEEE Trans. on Comput.*, vol. C-43, no. 12, pp. 1386–1397, Dec. 1994.

Lin, X., P. K. McKinley, and L. M. Ni. "Deadlock-free multicast wormhole routing in 2-D mesh multicomputers." *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 793–804, Aug. 1994.

Linder, D. H. and J. C. Harden. "An adaptive and fault tolerant wormhole routing strategy for k-ary n-cubes." *IEEE Trans. on Comput.*, vol. C-40, no. 1, pp. 2–12, Jan. 1991.

Lu, Y. W., J. B. Burr, and A. M. Peterson. "Optimization of the torus with segmented reconfigurable bus for data routing." In *Proc. Int. Conf. on Parallel and Distributed Systems*, Dec. 1993.

Lu, Y. W., J. B. Burr, and A. M. Peterson. "Permutation on the mesh with reconfigurable bus: algorithms and practical considerations." In *Proceedings of 7th International Parallel Processing Symposium*, Apr. 13-16 1993.

Mahnud, S. M. "Performance analysis of multilevel bus networks for hierarchical multiprocessors." *IEEE Trans. on Comput.*, vol. 43, no. 7, pp. 789–805, July 1994.

Merlin, P. M. and P. J. Schweitzer. "Deadlock avoidance in store-and-forward networks I: Store-and-forward deadlock." *IEEE Trans. on Commun.*, vol. COM-28, no. 3, pp. 345–354, Mar. 1980.

Miller, R., V. K. Prasanna-Kumar, D. Reisis, and Q. F. Stout. "Meshes with reconfigurable buses." In *Advanced Research in VLSI. Proceedings of the Fifth MIT Conference*, pp. 163–178, Mar. 1988.

Ni, L. M. and P. K. McKinley. "A survey of wormhole routing techniques in direct networks." *IEEE Computer*, vol. 26, no. 2, pp. 62–76, Feb. 1993.

Pifarré, G. D., L. Gravano, S. A. Felperin, and J. L. C. Sanz. "Fully adaptive minimal deadlock-free packet routing in hypercubes, meshes, and other networks: algorithms and simulations." *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 3, pp. 247–263, Mar. 1994.

Reese, E. A., H. Wilson, D. Nedwek, J. Jex, M. Khaira, T. Burton, P. Nag, H. Kumar, C. Dike, D. Finan, and M. Haycock. "A Phase-tolerant 3.8GB/s data-communication router for a multiprocessor supercomputer backplane." In *Proc. IEEE Int. Solid-State Circuits Conf.*, pp. 296–297, 1994.

Segucgi, Y., S. Komori, H. Takata, T. Tamura, F. Asai, T. Tokuda, and H. Terada. "A flexible router chip for massively parallel data-driven computer." In *Proc. Symposium on VLSI Circuits*, pp. 27–28, 1991.

Silberschatz, A., J. Peterson, and P. Calvin. *Operating System Concepts.* Addison Wesley, Reading, MA., third edition, 1991.

Snyder, L. "Introduction to the configurable, highly parallel computer." *IEEE Computer*, pp. 47–56, Jan. 1982.

Tamir, Y. and H. C. Chi. "Symmetric crossbar arbiters for VLSI communication switches." *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 13–27, Jan. 1993.

Tamir, Y. and G. L. Frazier. "Dynamically-allocated multi-queue buffers for VLSI communication switches." *IEEE Trans. on Comput.*, vol. 41, no. 6, pp. 725–737, June 1992.

Tobagi, F. A. "Fast packet switch architectures for broadband integrated services digital networks." *Proceeding of the IEEE*, vol. 78, no. 1, pp. 133–167, Jan. 1990.

Traylor, R. and D. Dunning. "Routing chip for Intel Paragon Parallel Supercomputer." In *Hot Chip IV*, 1992.

Wang, B. F., G. H. Chen, and F. C. Lin. "Constant time sorting on a processor array with a reconfigurable bus system." *Information Processing Letters 34*, pp. 182–192, Apr. 1990.

Weste, N. and K. Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley, Reading, MA., 1985.