CHAPTER 7

INPUT-BUFFERED SWITCHES

Fixed-length switching technology is widely accepted as an efficient approach to achieving high switching efficiency for high-speed packet switches. Variable-length IP packets are segmented into fixed-length "cells" at inputs and are reassembled at the outputs. When high-speed packet switches were constructed for the first time, they used, either internal shared buffer or input buffer and suffered the problem of throughput limitation. As a result, historically most research focused on the output buffering architecture. Since the initial demand for switch capacity was in the range of a few hundred Mbit/s to a few Gbit/s, output buffered switches seemed to be a good choice because of their high throughput/delay performance and memory utilization (for shared memory switches). In the first few years of deploying ATM switches, output buffered switches (including shared memory switches) dominated the market. However, as the demand for large-capacity switches increased rapidly (either line rates or the switch port number increased), the speed requirement for the memory had to increase accordingly. This limits the capacity of output buffered switches. Although output buffered switches have optimal delay-throughput performance for all traffic distributions, the N-times speed-up for the memory operation speed limits the scalability of this architecture. Therefore, in order to build larger-scale and higher-speed switches, people now have focused on input-buffered, or combined-input-output-buffered switches with advanced scheduling and routing techniques, which are the main subjects of this chapter.

Input buffered switches are desirable for high-speed switching, since the internal operation speed is only moderately higher than the input line speed. But there are two problems: (1) throughput limitation due to the head-of-line (HOL) blocking (throughput limited to 58.6 percent for FIFO buffering), and (2) the need of arbitrating cells due to output port contention. The first problem can be circumvented by moderately increasing the switch fabric's operation speed or the number of routing paths to each output port (i.e., allowing multiple cells arriving at the output port at the same time slot). The second problem is resolved by novel, fast arbitration (i.e., scheduling) schemes that are described in this chapter. According to Moore's Law, memory density doubles every 18 months. But the memory speed increases at a much slower rate. For instance, the memory speed is 4 ns for state-of-the-art CMOS static RAM compared to 5 ns one or two years ago. On the other hand, the speed of logic circuits has increased at a much higher rate than that of memory. Recently, much research has been devoted to devising faster scheduling schemes to arbitrate cells from input ports to output ports.

Some scheduling schemes even consider per-flow scheduling at the input ports to meet the delay/throughput requirements for each flow, which of course greatly increases implementation complexity and cost. Scheduling cells on a per-flow basis at input ports is much more difficult than at output ports. For example, at an output port, cells (or packets) can be time-stamped with values based on their allocated bandwidth and transmitted in an ascending order of their time stamp values. However, at an input port, scheduling cells must take output port contention into account. Thus, it makes the problem so complicated that so far no feasible scheme has been devised.

7.1 SCHEDULING IN VOQ-BASED SWITCHES

The basic input buffer switch model is shown in Figure 7.1. A FIFO queue is implemented in front of each input of the switch fabric, and is used to store incoming packets. They are then scheduled to transmit to the switch fabric. Because of the HOL blocking, Section 5.3 shows that the overall throughput of the input buffer structure is limited to 58.6 percent under uniform traffic, and even worse under non-uniform traffic.

Because of the throughput limitation from the FIFO queue structure, virtual output queue (VOQ) structure, as shown in Figure 7.2, has been widely used to eliminate the HOL blocking and thus improves the system throughout. In each input buffer, there are N FIFO queues (N is the switch size), each corresponding to an output port, or N^2 FIFO queues in total. In other words, packets/cells arriving at input port i and destined for output port j are stored in $VOQ_{i,j}$ (i.e., $Q_{i,j}$ in Fig. 7.2). The HOL cell of each VOQ can be scheduled for transmission in every time slot. However, there will be at most one cell among the N VOQs being selected for transmission.

The cell arrival to input port i is a stochastic process $A_i(t)$. Within each time slot, there is at most one arrived at each input port. The cell that arrived at input port i and destined to output j is put into queue Q_{ij}. At time slot t, the queue length of Q_{ij} is denoted as $L_{ij}(t)$.

$A_{ij}(t)$ denotes the arrival process from input i to output j with an arrival rate of λ_{ij}, and $A(t) = \{A_{ij}(t), 1 \leq i \leq N \text{ and } 1 \leq j \leq N\}$. If the arrivals to each input and each output are

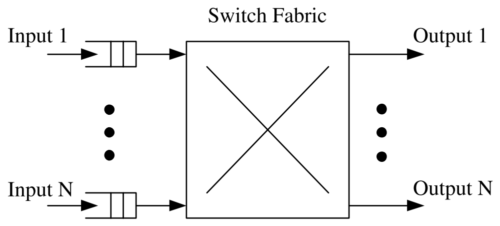


Figure 7.1 Input-buffered switch model with FIFO queues.

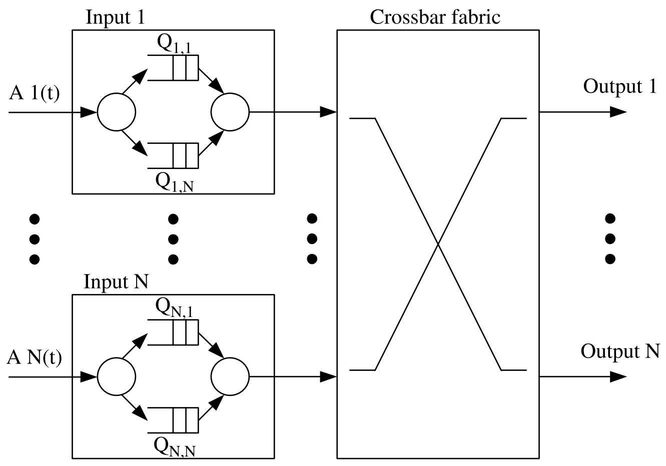


Figure 7.2 Input-buffered structure with virtual output queues.

admissible, that is,

$$\sum_{i=1}^{N} \lambda_{ij} < 1, \ \forall j \qquad \text{and} \qquad \sum_{j=1}^{N} \lambda_{ij} < 1, \ \forall i$$

then the set $A(t)$ is admissible. Otherwise, it is not admissible. The arrival rate matrix is denoted as $\Lambda = [\lambda_{ij}]$.

Let the service matrix $S(t) = [s_{ij}(t)]_{N \times N}$ represent the matchings at time slot t with each element:

$$s_{ij}(t) = \begin{cases} 1, & \text{if a cell is transfered from } i \text{ to } j \\ 0, & \text{otherwise} \end{cases}$$

A cell arrival that is an independent process satisfies the following two conditions: (1) Cell arrivals to each input port are independent and identically distributed; (2) The arrivals to an input port are independent from other input ports. If the arrival rates are equal, and the destinations are uniformly distributed among all output ports, then the arrival distribution is said to be uniform.

Throughput and delay are used to evaluate a switch's performance. Throughput is defined to be the average number of cells transmitted in a time slot, and delay is defined to be the time experienced by a cell from arrival to departure. A switch is defined to be stable, if the expected queue length is bounded, that is,

$$E\left[\sum_{ij} L_{ij}\right] < \infty, \ \forall t.$$

If a switch is stable under any independent and admissible input traffic, then the switch can achieve 100 percent throughput.

Sophisticated scheduling algorithms are required to schedule cells to the switch fabric in each time slot. These scheduling algorithms can be modeled as a bipartite graph matching

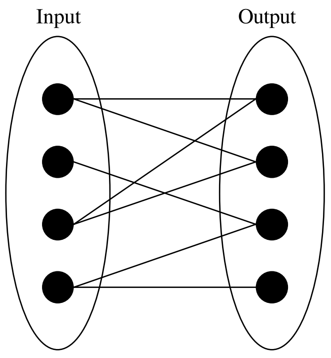


Figure 7.3 Bipartite graph matching example.

problem. In Figure 7.3, N nodes on the left stand for the N input ports, while the N nodes on the right stand for N output ports. The edge between an input port and an output port denotes that there are requests for cells to be transferred between them. A scheduler is responsible for selecting a set of the edges (also called matches) from at most N^2 edges, where each input is connected to at most one output and each output is connected to at most one input. A matching example is shown in Figure 7.4 where the dotted lines represent the requests that are not granted. A matching of input–output can be represented as a permutation matrix $M = (M_{i,j}), i,j \leq N$, where $M_{i,j} = 1$ if input i is matched to output j in the matching.

To select a proper scheduling algorithm, several factors must be considered:

Efficiency. The algorithm should achieve high throughput and low delay. In other words, select a set of matches with more edges in each time slot.

Fairness. The algorithm should avoid the starvation of each VOQ.

Stability. The expected occupancy of each VOQ should remain finite for any admissible traffic pattern.

Implementation Complexity. The algorithm should be easy for hardware implementation. High implementing complexity will cause long scheduling time, which further limits the line speed of the switch.

With the above design objectives, many scheduling algorithms have been proposed. We will examine some of these algorithms in detail.

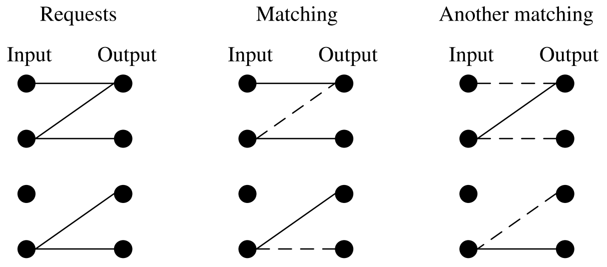


Figure 7.4 Matching example.

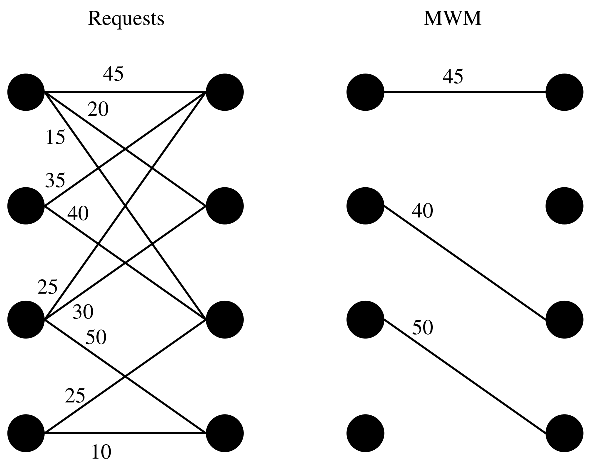


Figure 7.5 Maximum weight match example.

7.2 MAXIMUM MATCHING

7.2.1 Maximum Weight Matching

In a bipartite graph, we define $w_{i,j}$ as the weight of edge $e_{i,j}$, from input i to output j. Weight of a VOQ refers usually, but not restricted to, the length (number of packets in backlog) of the VOQ. The maximum weight matching (MWM) M for a bipartite graph is one that maximizes $\sum_{e_{(i,j)} \in M} w_{i,j}$. Figure 7.5 shows an example of a maximum weight match.

Theorem 1: A maximum weight matching algorithm achieves 100 percent throughput under any admissible traffic [1, 2].

MWM can be solved in time $O(N^3)$ [3], which is, too large for a high-speed packet switch. By carefully selecting the weight of each edge, or using approximations of MWM, we can reduce the complexity of computing maximum weight matches.

LQF (longest queue first) and OCF (oldest cell first) [1] are two MWM algorithms that were proposed early on. LQF uses the queue length $L_{ij}(t)$ as the weight $w_{ij}(t)$ and OCF uses the HOL cell's waiting time as the weight $w_{ij}(t)$. Under admissible traffic, these two algorithms can achieve 100 percent throughput. Under inadmissible traffic, starvation can occur for LQF, but not for OCQ. In order to reduce the complexity of LQF, the LPF (longest port first) algorithm is proposed [4] with weight $w_{ij}(t)$ defined to be the port occupancy:

$$w_{ij}(t) = \begin{cases} R_i(t) + C_j(t), & L_{ij}(t) > 0 \\ 0, & \text{otherwise} \end{cases}$$

where $R_i(t) = \sum_{j=1}^{N} L_{ij}(t)$, $C_j(t) = \sum_{i=1}^{N} L_{ij}(t)$. Since the weight of LPF is not equal to the queue length, it has the advantage of both maximum size matching (MSM) and MWM.

7.2.2 Approximate MWM

In the work of Shah and Kopikare [5], a class of approximations to MWM, 1-APRX, was proposed and defined as follows. Let the weight of a schedule obtained by a scheduling algorithm B be W^B. Let the weight of the maximum weight match for the same switch state be W^*. B is defined to be a 1-APRX to MWM, if the following property is always true: $W^B \geq W^* - f(W^*)$, where $f(\cdot)$ is a sub-linear function, that is, $\lim_{x \to \infty} [f(x)/x] = 0$ for any switch state.

Theorem 2: Let $W^*(t)$ denote the weight of maximum weight matching scheduling at time t, with respect to switch state $Q(t)$ (where $Q(t) = [Q_{i,j}(t)]$ and $Q_{i,j}(t)$ denotes the number of cells in $VOQ_{i,j}$). Let B be a 1-APRX to MWM and $W^B(t)$ denote its weight at time t. Further, B has a property that,

$$W^B(t) \geq W^*(t) - f(W^*(t)), \qquad \forall t, \tag{7.1}$$

where $f(\cdot)$ is a sub-linear function. Then the scheduling algorithm B is stable under any admissible Bernoulli i.i.d. input traffic.

Theorem 2 can be used to prove the stability of some matching algorithms that are not MWM and with lower complexity. Examples are the de-randomized matching algorithm with memory in Section 7.5.2 and the Exhaustive Service Matching with Hamiltonian Walk in Section 7.4.4.

7.2.3 Maximum Size Matching

Maximum size matching (MSM) finds the match containing the maximum number of edges. Obviously, maximum size matching is a special case of the maximum weight matching when the weight of each edge is 1. The time complexity of MSM is $O(N^{2.5})$ in survey [6]. Figure 7.6 shows an example of a maximum size match.

It has been shown by simulation that MSM delivers 100 percent throughput when the traffic is permissible uniform [7]. However, under admissible nonuniform traffic, it can lead to instability and unfairness, and under impermissible distribution, it can lead to starvation for some ports [1].

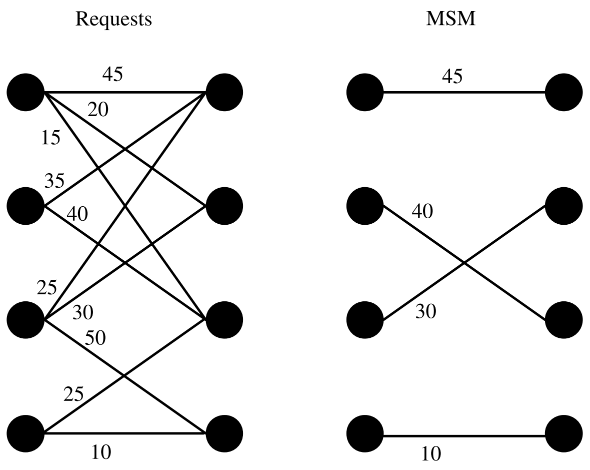


Figure 7.6 Maximum size match (MSM) example.

7.3 MAXIMAL MATCHING

A maximal matching algorithm leads to a maximal match by adding connections incrementally, without removing connections made earlier in the matching process. In a maximal match, if a nonempty input is not matched to any output, all destination outputs of the cells waiting in this input must have been matched to other inputs. Figure 7.7 shows an example of a maximal match. Generally speaking, a maximal match has fewer matched edges than a maximum size match, but is simpler to implement.

Maximal matching algorithms have lower implementation complexity than MWM. However, simulation results show that maximal matching achieves 100 percent throughput under uniform traffic, but not under nonuniform traffic. Speedup is needed to guarantee 100 percent throughput when a maximal matching algorithm is used. A switch with a speedup of s can transfer up to s cells from each input and up to s cells to each output within a time slot. An output-queued switch has a speedup of N and an input-queued switch has a speedup of 1. Usually, when $1 < s < N$, the switch is called a combined input- and output-queued (CIOQ) switch, since cells need to be buffered at the inputs before switching as well as at the outputs after switching.

Theorem 3: Under any admissible traffic, a CIOQ switch using any maximal matching algorithm achieves 100 percent throughput for any speedup $s \geq 2$ [8, 9].

One way to implement maximal matching is to use iterative matching algorithms that use multiple iterations to converge on a maximal match. In each iteration, at least one more connection is added to the match before a maximal match is achieved. Therefore, a maximal match can always be found within N iterations. Some multiple iteration matching algorithms converge faster and require fewer iterations, for example, $\log_2 N$. The main difference between various iterative matching algorithms is the rationale for selection of input ports to grant, and output ports to accept, when scheduling.

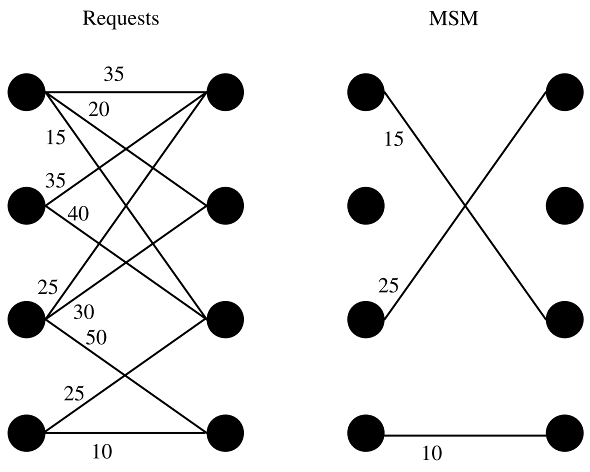


Figure 7.7 Maximal match example.

7.3.1 Parallel Iterative Matching (PIM)

The PIM scheme [10] uses random selection to solve the contention between inputs and outputs. Input cells are first queued in VOQs. Each iteration consists of three steps. There can be up to N iterations in each time slot. Initially, all inputs and outputs are unmatched and only those inputs and outputs that are not matched at the end of an iteration will be eligible to participate in the next matching iteration. The three steps in each iteration operate in parallel on each input and output as follows.

Step 1: *Request.* Each unmatched input sends a request to every output for which it has a queued cell.

Step 2: *Grant.* If an unmatched output receives multiple requests, it grants one by randomly selecting a request over all requests. Each request has an equal probability of being granted.

Step 3: *Accept.* If an input receives multiple grants, it selects one to accept in a fair manner and notifies the output.

Figure 7.8 shows an example of PIM in the first iteration. In step 1, input 1 requests output 1 and 2, input 3 requests output 2 and 4, input 4 requests output 4. In step 2, output 1 grants the only request from input 1, output 2 randomly chooses input 3 to grant, and output 4 chooses input 3. In the last step 3, input 1 accepts the only grant from output 1 and input 3 randomly chooses output 2 to accept.

It has been shown that each iteration resolves, on average, at least 25 percent of the remaining unresolved requests. Thus, the algorithm converges at $O(\log N)$ iterations, where N denotes the number of ports. No memory or state is used to keep track of how recently a connection was made in the past since, at the beginning of a cell time slot, the match begins over, independently of the matches that were made in previous cell time slots. On the other hand, PIM does not perform well for a single iteration. The throughput of PIM for one iteration is about 63 percent under uniform traffic. The reason is as follows:

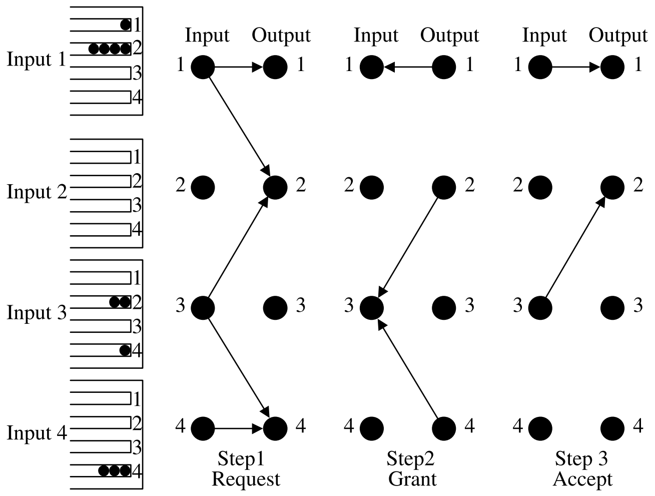


Figure 7.8 Parallel iterative matching (PIM) example.

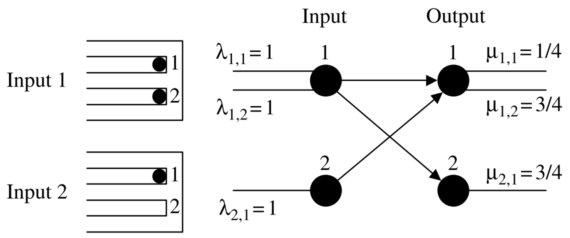


Figure 7.9 Example of unfairness in PIM under heavy, oversubscribed load with more than one iteration.

Assume that each VOQ of an $N \times N$ switch is nonempty. With PIM, each output will receive N requests, one from each input. The probability that an output selects a particular input to grant is $1/N$, so that the probability that an input is not selected by an output is $1 - (1/N)$. If an input is granted by at least one output, this input will randomly select one of them to accept and be matched. In other words, if an input does not receive any grant, it will not be matched. This will happen when none of the outputs selects this input to grant, with a probability of $[1 - (1/N)]^N$, or $1/e$ as $N \to \infty$. Therefore, the throughput tends to $1 - (1/e) \simeq 0.63$ as $N \to \infty$.

Thus, under uniform traffic, PIM achieves 63 percent and 100 percent throughput for 1 and N iterations, respectively. Despite the 100 percent throughput PIM achieves with N iterations. When the switch is oversubscribed, PIM can lead to unfairness between connections [11]. Figure 7.9 shows such a situation. In the figure, input port 1 has cells to output port 1 and 2 in every time slot and similarly input port 2 has cells to output port 1. Under PIM, input port 1 will only accept output port 1 for a quarter of the time since output port 1 should first grant input port 1 and then input port 1 should accept output port 1. However, since no input port competes with input port 1 at output port 2, input port 1 will accept output port 2 during the other three quarters of the time. This results in unfairness between traffic from input port 1 to output port 1 and from input port 1 to output port 2. Moreover, implementing a random function at high speed can be expensive.

7.3.2 Iterative Round-Robin Matching (*i*RRM)

The *i*RRM scheme [12] works similar to PIM, but uses the round-robin schedulers instead of random selection at both inputs and outputs. Each scheduler maintains a pointer pointing at the port that has the highest priority. Such a pointer is named accept pointer a_i at input i and grant pointer g_j at output j. The steps for this algorithm are as follows:

Step 1: *Request.* Each unmatched input sends a request to every output for which it has a queued cell.

Step 2: *Grant.* If an unmatched output receives any requests, it chooses the one that appears next in a round-robin schedule starting from the highest priority element. The output notifies each input whether or not its request was granted. The pointer g_i is incremented (module N) to one location beyond the granted input.

Step 3: *Accept.* If an input receives multiple grants, it accepts the one that appears next in its round-robin schedule starting from the highest priority element. Similarly, the pointer a_j is incremented (module N) to one location beyond the accepted output.

An example is shown in Figure 7.10. In this example, we assume that the initial value of each grant pointer is input 1 (e.g., $g_i = 1$). Similarly, each accept pointer is initially pointing to output 1 (e.g., $a_j = 1$). During step 1, the inputs request transmission to all outputs that they have a cell destined for. In step 2, among all received requests, each grant scheduler selects the requesting input that is nearest to the one currently pointed to. Output 1 chooses input 1, output 2 chooses input 1, output 3 has no requests, and output 4 chooses input 3. Then, each grant pointer moves one position beyond the selected one. In this case, $g_1 = 2$, $g_2 = 2$, $g_3 = 1$, and $g_4 = 4$. In step 3, each accept pointer decides which grant is accepted in a similar way as the grant pointers did. In this example, input 1 accepts output 1, and input 3 accepts output 4; then $a_1 = 2$, $a_2 = 1$, $a_3 = 1$, and $a_4 = 1$. Notice that the pointer a_3 accepted the grant issued by output 4, so the pointer returns to position 1.

Although *i*RRM brings good fairness by using a round-robin policy, it actually does not achieve much higher throughput than PIM under 1 iteration. This is predominantly due to the output pointer update mechanism. Considering the situation when input port i with accept pointer pointing to j has cells for all output ports, and the grant pointers in all output point to i, with one iteration, only one cell will be transferred in the system from input port i to output port j. It is worse that all output pointers are updated to $i + 1$ identically. This phenomenon in *i*RRM is called output synchronization. It significantly degrades the throughput of *i*RRM. An example is given in Figure 7.11. The 2×2 switch with one iteration *i*RRM will only achieve 50 percent throughput under heavy load.

7.3.3 Iterative Round-Robin with SLIP (*i*SLIP)

As illustrated in the above section, although easily implemented in hardware, the *i*RRM, in some cases, suffers from output synchronization. An enhanced scheme (*i*SLIP) was presented in the work of McNeown [11].

The steps for this scheme are as follows:

Step 1: *Request.* Each unmatched input sends a request to every output for which it has a queued cell.

Step 2: *Grant.* If an unmatched output receives multiple requests, it chooses the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The output notifies each input whether or not its request was granted. The grant pointer g_i is incremented (module N) to one location beyond the granted input if and only if the grant is accepted in step 3 of the first iteration.

Step 3: *Accept.* If an input receives multiple grants, it accepts the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The pointer a_j is incremented (modulo N) to one location beyond the accepted output. The accept pointers a_i are only updated in the first iteration.

The main difference compared to *i*RRM is that in *i*SLIP, the grant pointers update their positions only if their grants are accepted. Also, the output grant pointer and input accept pointer will only be updated during the first iteration, since if the pointers are updated in each iteration, some connection may be starved indefinitely. Figure 7.12 shows an example. It depicts a request graph in the first time slot. Assume that the accept pointer of input port 1 points to output port 1, and the grant pointer of output 2 points to input port 1. During the first iteration of that time slot, the input port 1 will accept output port 1, and in the second iteration, output port 2 will grant input port 2, since input port 1 has been connected.

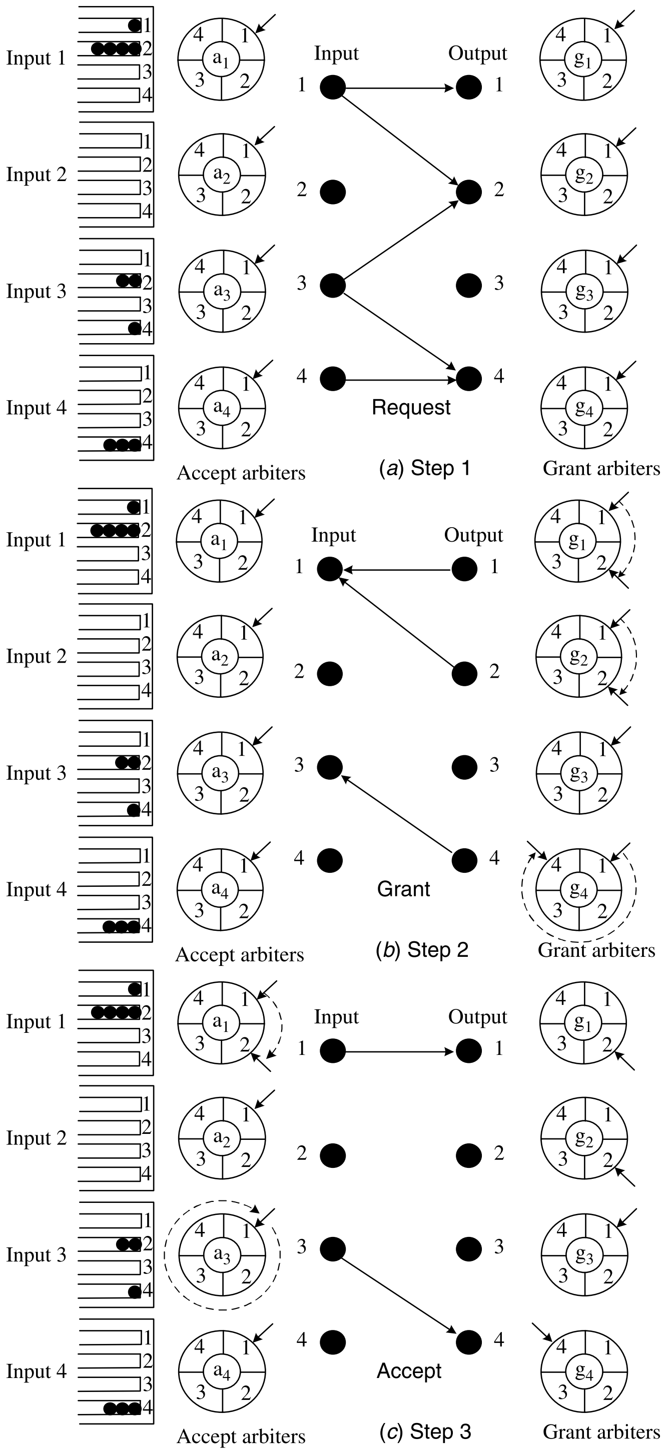


Figure 7.10 *i*RRM example.

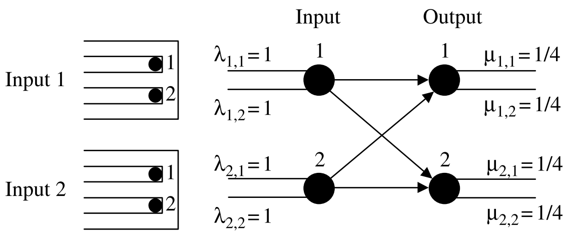


Figure 7.11 Synchronization of output arbiters leads to a throughput of just 50 percent in *i*RRM under heavy, oversubscribed load with more than one iteration.

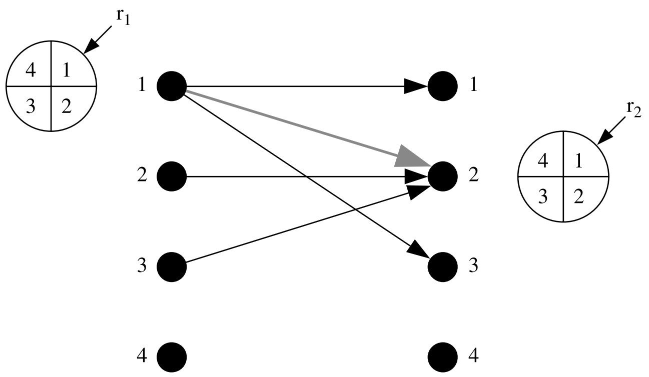


Figure 7.12 Request graph when *i*SLIP will cause starvation if pointers are updated in every iteration.

When the input port 2 finally accepts output port 2, the grant pointer in output port 2 will be updated beyond input port 1 to port 3. So the connection between input port 1 and output port 2 may be postponed indefinitely.

However, in this scheme of updating pointer only in first iteration, starvation is avoided because every requesting pair will be served within 2N time slots, and the newest connected pair is assigned the lowest priority.

Because of the round-robin motion of the pointers, the algorithm provides a fair allocation of bandwidth among all flows. This scheme contains 2N arbiters, where each arbiter is implementable with low complexity. The throughput offered with this algorithm is 100 percent under uniform traffic for any number of iterations due to the desynchronization effect (see Section 7.3.5). A matching example of this scheme is shown in Figure 7.13. Considering the example from the *i*RRM discussion, initially all pointers a_j and g_i are set to 1. In step 2 of *i*SLIP, the output accepts the request that is closer to the pointed input in a clockwise direction; however, in a manner different from *i*RRM, the pointers g_i are not updated in this step. They wait for the acceptance result. In step 3, the inputs accept the grant that is closer to the one pointed to by a_i. The accept pointers change to one position beyond the accepted one, $a_1 = 2$, $a_2 = 1$, $a_3 = 1$, and $a_4 = 1$. Then, after the accept pointers decide which grant is accepted, the grant pointers change to one position beyond the accepted grant (i.e., a non-accepted grant produces no change in a grant pointer position). The new values for these pointers are $g_1 = 2$, $g_2 = 1$, $g_3 = 1$, and $g_4 = 4$. In the

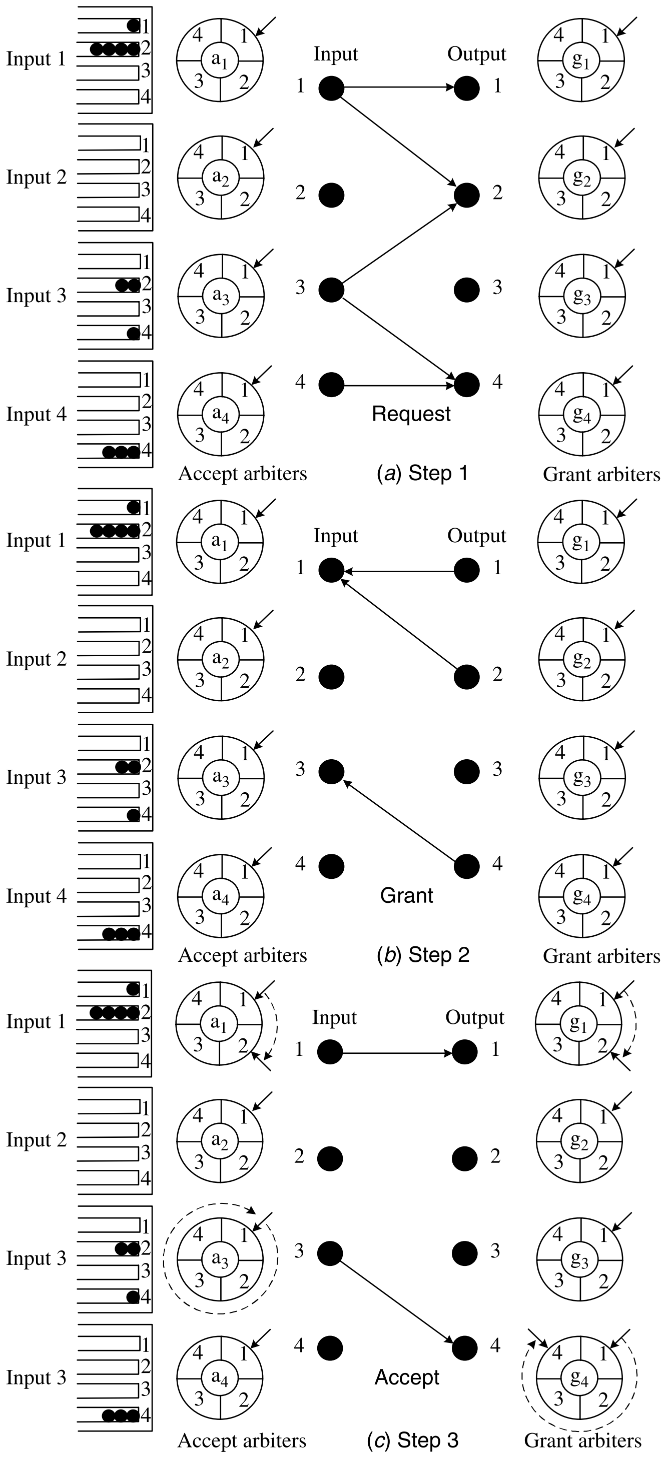


Figure 7.13 *i*SLIP example.

following iterations, only the unmatched input and outputs are considered and the pointers are not modified (i.e., updating occurs in the first iteration only).

*i*SLIP can achieve 100 percent throughput with a single iteration under uniform Bernoulli independent and identically distributed (i.i.d.) arrival traffic [13, 14]. This is because, within a finite time, the pointer of each output arbiter will point to an input different from all other output pointers, so that every input will have one cell to be served in every time slot. We will show this by using a ball game.

Under heavy traffic, the *i*SLIP scheduling algorithm reduces to the following rules:

- In any time slot, each input sends requests to every output.
- At any output, if the pointer points to k in a time slot, this output grants the request from the kth input. If the grant is selected to be accepted by the kth input arbiter, in the next time slot the pointer points to $(k+1) \bmod N$; if it is not selected, in the next time slot the pointer will be still at k.
- At any input k, if only one output arbiter pointer points to k, this output will be selected by input k; if there are $m (m > 1)$ output arbiter pointers pointing to input k, one of them will be selected.

We define a vector $X_i = (x_{1,i}, \ldots, x_{k,i}, \ldots, x_{N,i})$ to express the state of input arbiters; in time slot i, there are $x_{k,i}$ output arbiter pointers pointing to input k, $k = 1, \ldots, N$, $0 \leq x_{k,i} \leq N$, $\sum_{k=1}^{N} x_{k,i} = N$.

If at time slot i, $x_{k,i} = 1$, $k = 1, \ldots, N$, which indicates that each input is pointed by an output arbiter pointer, then the throughput is 100 percent in this time slot. We will now proceed to show $X_i = (1, 1, \ldots, 1)$ for all $i \geq N - 1$. Thus a throughput of 100 percent can be sustained indefinitely after $N - 1$ time slots.

To simplify the notation, we will drop the mod N, that is, $(k + l) \bmod N$ will be represented by $k + l$. Using the *i*SLIP arbitration rules to the vector X_i, we get:

$$x_{k,i+1} = \begin{cases} 0 & x_{k,i} \leq 1, x_{k-1,i} = 0 \\ x_{k,i} - 1 & x_{k,i} > 1, x_{k-1,i} = 0 \\ 1 & x_{k,i} \leq 1, x_{k-1,i} > 0 \\ x_{k,i} & x_{k,i} > 1, x_{k-1,i} > 0 \end{cases} \tag{7.2}$$

By cyclically shifting X_i to the left by one slot every slot time, we get another vector $Y_i = (y_{1,i}, \ldots, y_{k,i}, \ldots, y_{N,i})$. This Y_i is defined as follows: in time slot 0,

$$Y_0 = X_0 \tag{7.3}$$

that is, $y_{k,0} = x_{k,0}$, $k = 1, \ldots, N$, and in time slot $m \geq 0$,

$$y_{k,m} = x_{k+m,m}, \quad k = 1, \ldots, N \tag{7.4}$$

At any time slot, $y_{k,i}$ represents the state of one input arbiter. If, and only if, in time slot i, $y_{k,i} = 1$ for all $k = 1, \ldots, N$, then $x_{k,i}$ also equals to 1 for all k. Therefore it is sufficient to show that $Y_i = (1, 1, \ldots, 1)$ for all $i \geq N - 1$ to prove the 100 percent throughput of *i*SLIP under uniform traffic.

According to (7.2), (7.3), and (7.4), we get

$$y_{k,i+1} = \begin{cases} 0 & y_{k+1,i} \leq 1, y_{k,i} = 0 \quad (\textit{condition 1}) \\ y_{k+1,i} - 1 & y_{k+1,i} > 1, y_{k,i} = 0 \quad (\textit{condition 2}) \\ 1 & y_{k+1,i} \leq 1, y_{k,i} > 0 \quad (\textit{condition 3}) \\ y_{k+1,i} & y_{k+1,i} > 1, y_{k,i} > 0 \quad (\textit{condition 4}) \end{cases} \tag{7.5}$$

From (7.5), by considering the third and fourth conditions, we can conclude that whenever y_k is larger than 0, it will always be larger than 0 after that. According to this conclusion, if in time slot i, $y_{k,i} = 1$ for all $k = 1, \ldots, N$, then for any time slot $j > i$, all $y_{k,j}$ will always be larger than 0. Since there are N inputs and N outputs, and $\sum_{k=1}^{N} y_k = N$, after time slot i, $y_k = 1, k = 1, \ldots, N$, which indicates that $x_k = 1$ for all k and the throughput will always be 100 percent. We will next prove that with any initial state Y_0, in a finite number of time slots M, where M is no more than $N - 1$, we will always have $y_{k,M} = 1, k = 1, \ldots, N$.

The state vector Y and its state transitions can be expressed as a game shown in Figure 7.14. In the game, we have N balls placed in N boxes. In time slot i, there are $y_{k,i}$ balls in the kth box, $k = 1, \ldots, N$. We will show that no matter how many balls there are in each box at the beginning, after at most $N - 1$ time slots, every box will always contain exactly one ball.

The rule that determines the movement of the balls in the boxes is as follows:

In time slot i, if box k is occupied and has m balls, $m > 0$, then in time slot $i + 1$, one of the m balls will stay in box k and the others, if any, will move to box $k - 1$. Since all the balls are identical, without losing generality we will require that the ball that arrived at box k earliest will stay in and occupy box k, and the others, if any, will move to box $k - 1$. If more than one ball arrives at an empty box, one of them is picked arbitrarily to stay there. Thus if a ball is put into an empty box, it stays there indefinitely.

Figure 7.14 shows an example of the movement of balls. In the figure, black balls are those that occupy a box permanently and white balls keep moving until they find an empty box and occupy it, at which point they turn black. We will prove that each of the N balls will find a box to occupy permanently in no more than $N - 1$ time slots, so that every box will always have one ball in it.

We will now show that the game corresponds to the state transitions of Y_i as defined in (7.5). By following the rules above, and by knowing how many balls there are in box k and box $k + 1$ in time slot i ($y_{k,i}$ and $y_{k+1,i}$), we can get the number of balls in box k in time slot $i + 1$ ($y_{k,i+1}$), which is identical with (7.5):

Condition 1. If in time slot i, box k is empty and box $k + 1$ has at most one ball, then in time slot $i + 1$, box k is still empty.

Condition 2. If in time slot i, box k is empty and box $k + 1$ has j balls, $j > 1$, then in time slot $i + 1$, one of these j balls stays in box $k + 1$ and box k will have the other $j - 1$ balls.

Condition 3. If in time slot i, box k has j balls, $j > 1$, and box $k + 1$ has at most one ball, then in time slot $i + 1$, no ball will move from box $k + 1$ to box k, and only one ball (which permanently occupies box k) will stay in box k.

Condition 4. If in time slot i, there are m balls in box k and j balls in box $k + 1$, $m > 1$ and $j > 1$, then in time slot $i + 1$, one of the m balls (which permanently occupies box k) stays in box k and others move to box $k - 1$, one of the j balls (which permanently occupies box $k + 1$) stays in box $k + 1$ and $j - 1$ balls move to box k; box k will then have j balls.

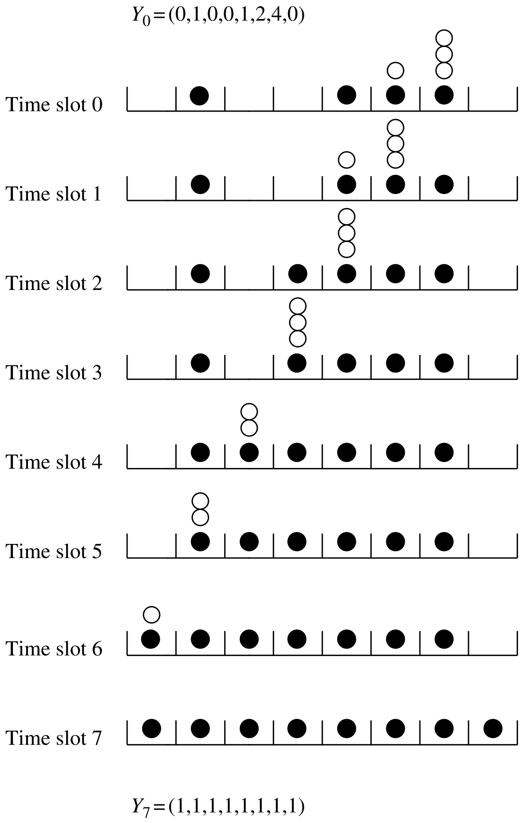


Figure 7.14 States of the system in 8 time slots when $N = 8$ and with the initial state $Y_0 = (0, 1, 0, 0, 1, 2, 4, 0)$.

In time slot 0, if a solitary ball occupies a box, it means that it has already found its final box. What we need to show is that if a ball does not occupy a box in time slot 0, it will find its box within $N - 1$ time slots.

Suppose in time slot 0, box k is occupied and there is a white ball (named ball B) in it; then, in the next time slot, ball B must move to box $k - 1$. We will next use a proof by contradiction. Assume that until time slot $N - 1$, ball B still cannot find its own box to occupy; this means it has moved in every time slot and traveled $N - 1$ boxes, all of which were occupied. Since box k is already occupied, all N boxes are occupied by N balls. With ball B, there will be a total of $N + 1$ balls in the system, which is impossible. So the assumption is wrong and ball B will find a box to occupy within $N - 1$ time slots.

Therefore, we conclude that any ball can find a box to occupy within $N - 1$ time slots, and from time slot $N - 1$, each box has one ball in it. Thus $Y_i, y_{k,i} = 1, k = 1, \ldots, N, i \geq N - 1$, for any Y_0, which indicates that after time slot $N - 1$, each output arbiter pointer will point to a different input, and will continue to do so indefinitely. This guarantees a throughput of 100 percent.

7.3.4 FIRM

Similar to *i*RRM and *i*SLIP, FIRM [15] also implements the round-robin scheduler to update input accept and output grant pointers. The only difference in this scheme lies in that the output grant pointers update their positions to the one beyond the granted input port if the grant is accepted in step 3, and update to the granted input port if the grant is not accepted in step 3. The steps for one iteration of this scheme are as follows:

Step 1: *Request.* Each unmatched input sends a request to every output for which it has a queued cell.

Step 2: *Grant.* If an unmatched output receives multiple requests, it chooses the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The output notifies each input whether or not its request was granted. The grant pointer g_i is incremented (modulo N) to one location beyond the granted input if the grant is accepted in step 3. It is placed to the granted input if the grant is not accepted in step 3.

Step 3: *Accept.* If an input receives multiple grants, it accepts the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The pointer a_j is incremented (modulo N) to one location beyond the accepted output. The accept pointers a_i are only updated in the first iteration.

Compared to *i*SLIP, FIRM is fairer as it approximates first come first serve (FCFS) closer to the improved update scheme of the output grant pointer. An example is shown in Figures 7.15 and 7.16.

In the cycle 0 of the example given in Figure 7.15, input 2 has queued cells for outputs 2 and 4, while input 3 has queued cells for outputs 1 and 3. No cells are queued in inputs 1 and 4. Initially, all pointers a_j and g_i are set as shown in Figure 7.15*a*. In step 1, input 2 sends requests to outputs 2 and 4 while input 3 sends requests to outputs 1 and 3. In step 2, the outputs grant the request that is closest to the pointed input by the grant pointer in the clockwise direction. So, input 2 receives grants from outputs 2 and 4; and input 3 receives grants from outputs 1 and 3. In step 3, the inputs accept the grant that is closest to the pointed output by the accept pointer. Hence, input 2 accepts the grant from output 4 and input 3 accepts the grant from output 3. As a result, a_2 and a_3 are updated and now point to outputs 1 and 4, respectively, (a_1 and a_4 are not updated). However, in a manner different from *i*SLIP, the grant pointers of outputs 3 and 4 are updated to inputs 4 and 3, respectively, since their grants were accepted and those of outputs 1 and 2 are updated to inputs 3 and 2, respectively, since their grants were not accepted. This completes cycle 0 of FIRM.

Figure 7.16 shows the situation at the beginning of cycle 1. New cells arrive from input 2 to output 1. Due to the difference in updating the output grant pointer in cycle 0, FIRM will grant input 2 in output 2 and input 3 in output 1, the two granted cells are both old cells that arrived in cycle 0. This example shows that FIRM is better in performing FCFS of arriving cells and is fairer, compared to *i*SLIP and *i*RRM.

7.3.5 Dual Round-Robin Matching (DRRM)

The DRRM scheme [16, 17] works similar to *i*SLIP, using the round-robin selection instead of random selection. But it starts the round-robin selection at inputs and only sends one

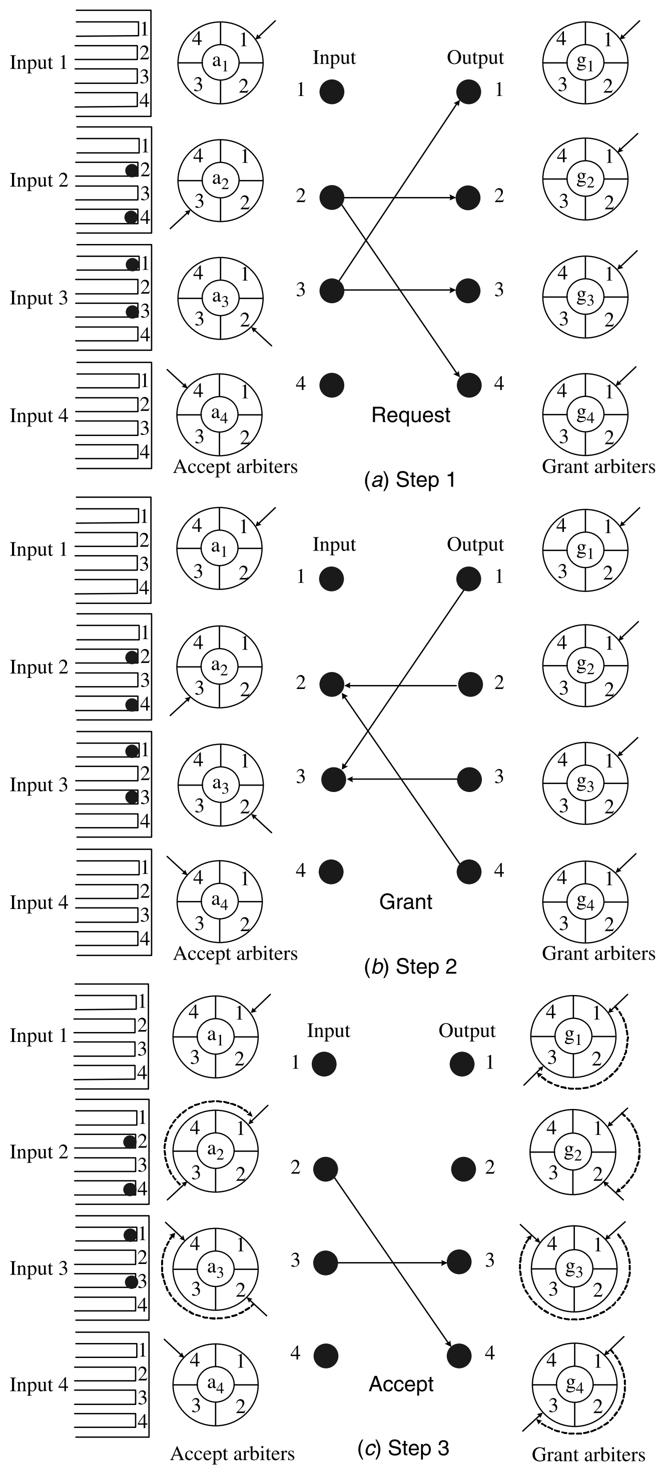


Figure 7.15 Example of FIRM–cycle 0.

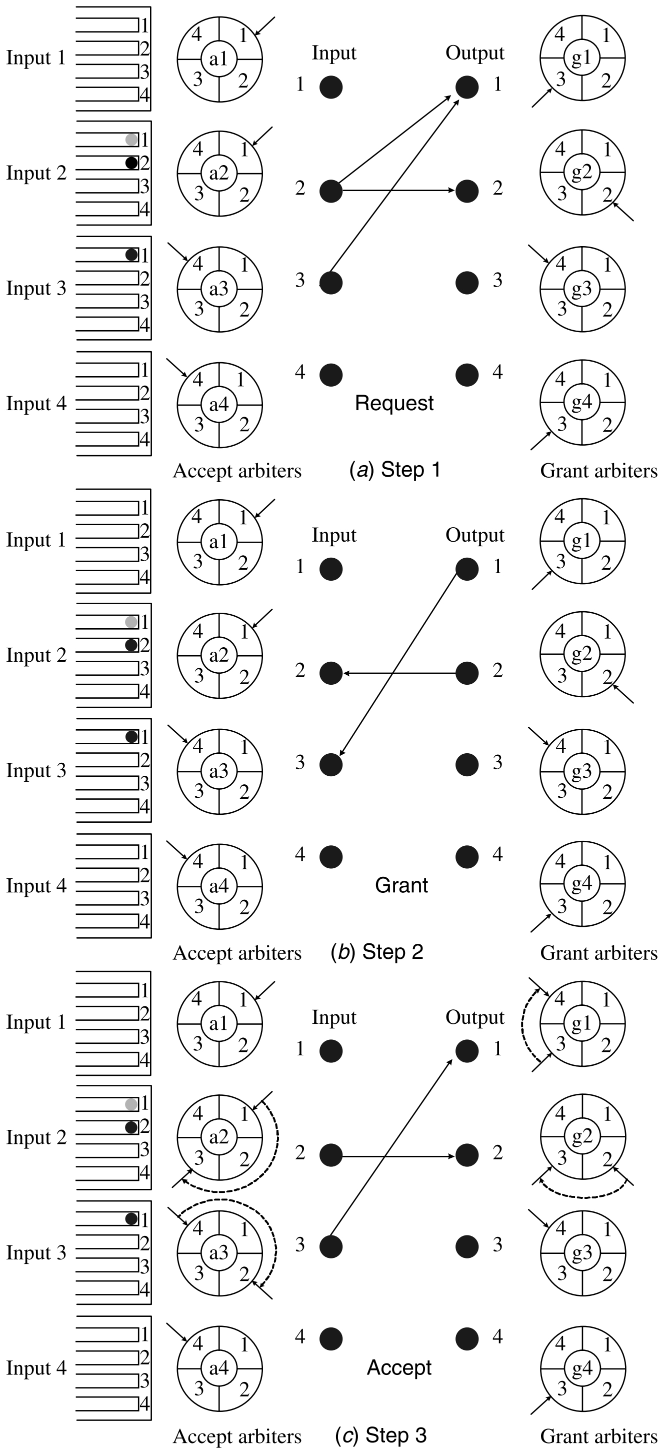


Figure 7.16 Example of FIRM–cycle 1.

request from each non-empty input port. An input arbiter is used to select a non-empty VOQ according to the round-robin service discipline. After the selection, each input sends a request, if any, to the destined output arbiter. An output arbiter receives up to N requests. It chooses one of them based on the round-robin service discipline, and sends a grant to the winner input port. Because of the two sets of independent round-robin arbiters, this arbitration scheme is called dual round-robin matching (DRRM).

The dual round-robin matching (DRRM) has two steps in a cycle:

Step 1: *Request.* Each input sends an output request corresponding to the first nonempty VOQ in a fixed round-robin order, starting from the current position of the pointer in the input arbiter. The pointer remains unchanged if the selected output is not granted in step 2. The pointer of the input arbiter is incremented by one location beyond the selected output if and only if the request is granted in step 2.

Step 2: *Grant.* If an output receives one or more requests, it chooses the one that appears next in a fixed round-robin scheduling starting from the current position of the pointer in the output arbiter. The output notifies each requesting input whether or not its request was granted. The pointer of the output arbiter is incremented to one location beyond the granted input. If there are no requests, the pointer remains where it is.

Because each input sends at most one request and receives at most one grant in each time slot, it is not necessary for input ports to conduct a third accept phase.

Figure 7.17 shows an example of the DRRM algorithm. In the request phase, each input chooses a VOQ and sends a request to an output arbiter. Assume input 1 has cells destined for both outputs 1 and 2. Since its round-robin pointer r_1 is pointing to 1, input arbiter 1 sends a request to output 1 and finally updates its pointer to 2 when its grant is accepted by output 1. Let us consider output 3 in the grant phase. Since its round-robin pointer g_3 is

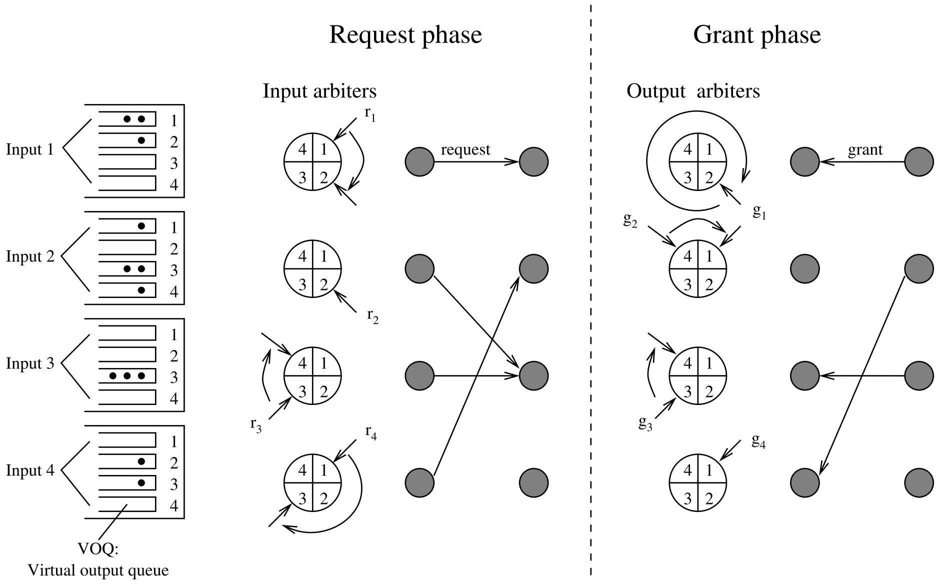
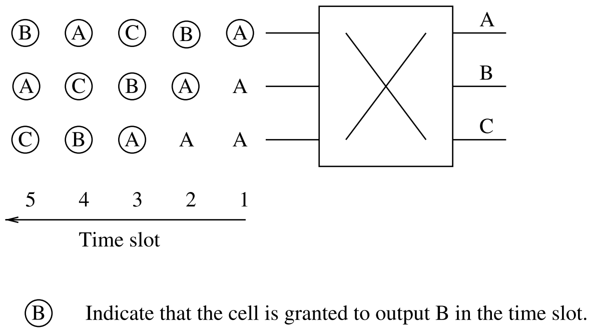


Figure 7.17 Example of DRRM scheduling algorithm.



B A C B A
A C B A A
C B A A A
A
B
C
5 4 3 2 1
Time slot
B Indicate that the cell is granted to output B in the time slot.

Figure 7.18 Desynchronization effect of DRRM under the fully loaded situation.

pointing to 3, output arbiter 3 grants input 3 and updates its pointer to 4. Similar to *i*SLIP, DRRM also has the desynchronization effect. The input arbiters granted in different time slots have different pointer values, and each of them requests a different output, resulting in desynchronization. However, the DRRM scheme requires less time to perform arbitration and is easier to implement. This is because less information exchange is needed between input arbiters and output arbiters. In other words, DRRM saves initial transmission time required to send requests from inputs to outputs in *i*SLIP.

Consider the fully loaded situation in which every VOQ always has cells. Figure 7.18 shows the HOL cells chosen from each input port in different time slots. In time slot 1, each input port chooses a cell destined for output A. Among those three cells, only one (the first one in this example) is granted and the other two have to wait at the HOL. The round-robin pointer of the first input advances to point to output B in time slot 2, and a cell destined for B is chosen and then granted because there are no contenders. The other two inputs have their HOL cells unchanged, both destined for output A. Only one of them (the one from the second input) is granted and the other has to wait until the third time slot. At that time, the round-robin pointers among the three inputs have been desynchronized and point to C, B, and A, respectively. As a result, all three cells chosen are granted.

Figure 7.19 shows the tail probability under FIFO+RR (FIFO for input selection and RR for round-robin arbitration), DRRM, and *i*SLIP arbitration schemes. The switch size is 256 and the average burst length is 10 cell slots (with the ON–OFF model). The performances of DRRM and *i*SLIP are comparable at speedup of 2, while all three schemes have almost the same performance as speedup $s \geq 3$.

7.3.6 Pipelined Maximal Matching

Maximal matching is widely adopted in the scheduling of input buffered switches, either with one cycle matching or multiple iterative matching. However, the computing complexity of maximal matching is, too high to complete within a single time slot when the switch size increases or the line rate becomes high.

In the work of Oki et al. [18], a scheduling algorithm called pipelined maximal matching (PMM) based on pipeline was proposed to resolve the problem. The PMM scheme relaxes the computation time for maximal matching to more than one time slot. As shown in Figure 7.20, the PMM scheduler consists of N^2 request counters (RC) and K subschedulers. Each subscheduler has N^2 request subcounters (RSC). As shown in Figure 7.21, each

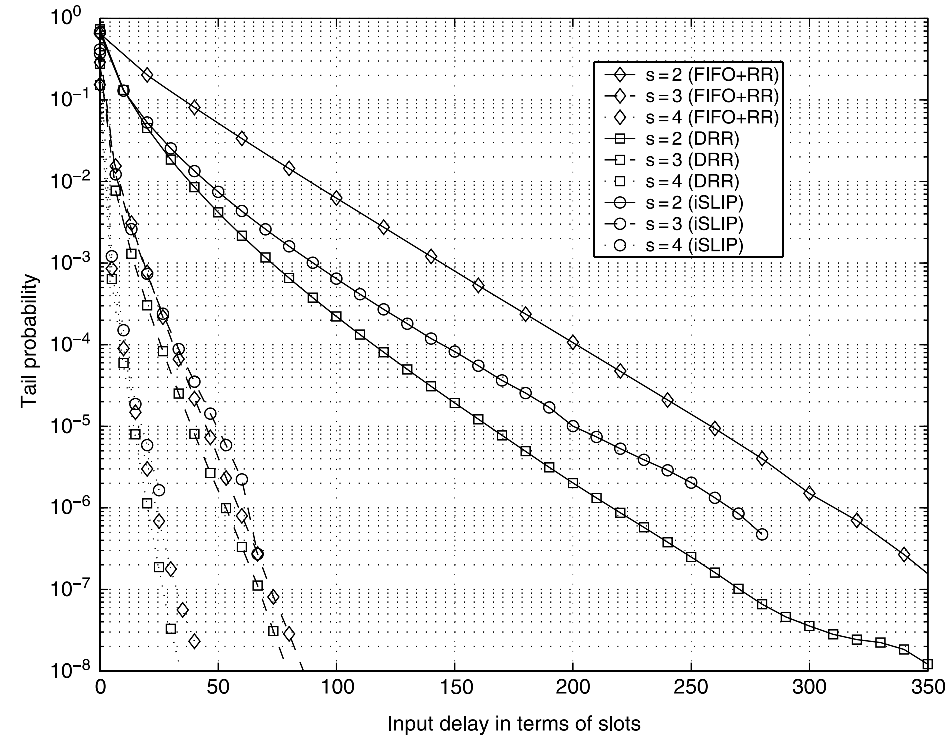


Figure 7.19 Comparison on tail probability of input delay under three arbitration schemes.

subscheduler operates the maximal matching in a pipelined manner and takes K time slots to complete the matching. After each time slot, one of the K subschedulers finishes a matching, which is used in the next time slot. DRRM is implemented in the subscheduler to achieve a maximal matching. Since K time slots are provided for one scheduling, multiple iterative DRRM (*i*DRRM) can be adopted.

Before a detailed description of PMM, several notations are first defined: $RC(i, j)$ denotes the request counter associated with $VOQ(i, j)$. $C(i, j)$ keeps the value of $RC(i, j)$, which is the number of accumulated requests associated with $VOQ(i, j)$ that have not been sent to any subscheduler. $RSC(i, j, k)$ denotes the request subcounter in the kth subscheduler that is associated with $VOQ(i, j)$, and $SC(i, j, k)$ keeps its value. The $SC(i, j, k)$ is the number of remaining requests that are dispatched from $RC(i, j)$ and not transferred yet. Each $SC(i, j, k)$ is limited to SC_{max}. It is found that when $SC_{max} = 1$, the delay performance is best. At initial time, each $C(i, j)$ and $SC(i, j, k)$ are set to zero. PMM operates as follows.

Phase 1. When a new cell enters $VOQ(i, j)$, the counter value of $RC(i, j)$ is increased as $C(i, j) = C(i, j) + 1$.

Phase 2. At the beginning of every time slot t, if $C(i, j) > 0$ and $SC(i, j, k) < SC_{\max}$, where $k = t \bmod K$, then $C(i, j) = C(i, j) - 1$ and $SC(i, j, k) = SC(i, j, k) + 1$. Otherwise, $C(i, j)$ and $SC(i, j, k)$ are not changed.

Phase 3. At $Kl + k \leq t < K(l + 1) + k$, where l is an integer, subscheduler k operates the maximal matching according to the adopted algorithm.

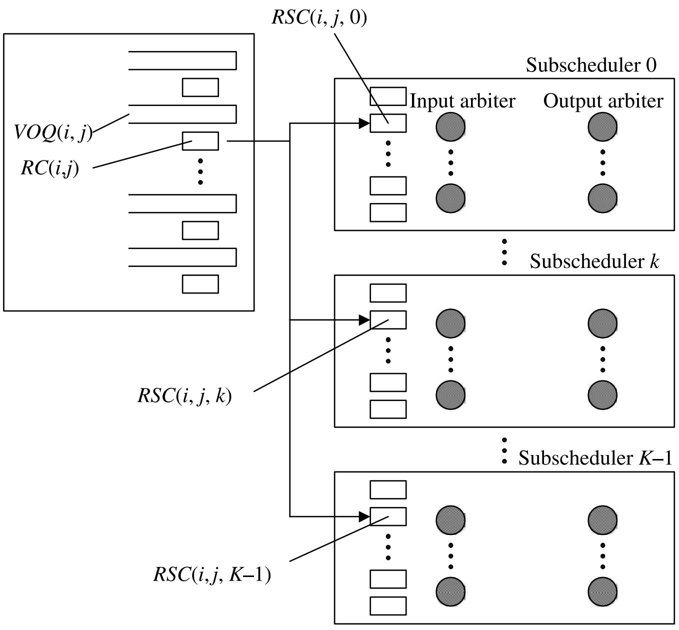


Figure 7.20 Structure of the PMM scheduler.

Phase 4. By the end of every time slot t, subscheduler k, where $k = (t - (K - 1)) \bmod K$, completes the matching. When input–output pair (i, j) is matched, $SC(i, j, k) = SC(i, j, k) - 1$. The HOL cell in $VOQ(i, j)$ is sent to output j at the next time slot. This ensures that cells from the same VOQ are transmitted in sequence, since only the HOL cell of each VOQ can be transferred when any request is granted.

Since the scheduling timing constraint has been relaxed, each subscheduler is now allowed to take several time slots to complete a maximal matching. By implementing the iterative DRRM, PMM can approximate the performance of *i*DRRM, providing 100 percent throughput under uniform traffic and fairness for best-effort traffic. Besides, cell order of the same VOQ is preserved.

Apart from PMM, other algorithms based on pipeline have been proposed, such as the Round-Robin Greedy Scheduling (RRGS) [19], but PMM is better in providing fairness and scalability.

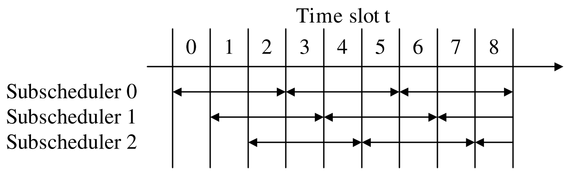


Figure 7.21 Timing diagram of PMM with three subscheduler.

7.3.7 Exhaustive Dual Round-Robin Matching (EDRRM)

In most scheduling algorithms, inputs and outputs are matched in each time slot. This is similar to the limited service policy with a limit of 1 [20] in a polling system. In order to improve the performance under non-uniform and burst traffic, some improved scheduling methods modify the limit-1 service policy to the limit-∞ policy so that whenever an input is matched to an output, all cells in the corresponding VOQ will be transferred in the following time slots before any other VOQ at the same input can be served. This is called the exhaustive service policy [20] in polling systems.

Combining the exhaustive service policy with DRRM, the exhaustive DRRM (EDRRM) [21] was proposed. In EDRRM, the input pointer is not updated until the current VOQ is exhausted. Besides the exhaustive scheduling, EDRRM has two other differences compared to DRRM. First, the pointer in the output arbiter always points to the latest matched input and does not update its location after a service because the output has no idea if the currently served VOQ will become empty after this service. Second, in EDRRM, if an input sends a request to an output but gets no grant, the input will update its pointer of input arbiter to the next location beyond the requested output, while in DRRM, this pointer will remain where it is until it gets a grant. The reason for this is because in EDRRM, if an input cannot get a grant from an output, it means that the output is most likely being matched with another input for all the cells waiting in the same VOQ. Therefore, it is better to update the pointer of input arbiter to search for another free output, rather than to wait for this busy one. The detailed description of EDRRM is shown below.

Step 1: *Request.* Each input moves its pointer to the first nonempty VOQ in a fixed round-robin order, starting from the current position of the pointer in the input arbiter, and sends a request to the output corresponding to this VOQ. The pointer of the input arbiter is incremented by one location beyond the selected output if the request is not granted in step 2, or if the request is granted and after one cell is served, this VOQ becomes empty. Otherwise, if there are remaining cells in this VOQ after sending one cell, the pointer remains at this nonempty VOQ.

Step 2: *Grant.* If an output receives one or more requests, it chooses the one that appears next in a fixed round-robin scheduling starting from the current position of the pointer in the output arbiter. The pointer is moved to this position. The output notifies each requesting input whether or not its request was granted. The pointer of the output arbiter remains at the granted input. If there are no requests, the pointer remains where it is.

Figure 7.22 shows an example of the EDRRM arbitration algorithm, where r_1, r_2, r_3, and r_4 are arbiter pointers for inputs 1, 2, 3, and 4, and g_1, g_2, g_3, and g_4 are arbiter pointers for outputs 1, 2, 3, and 4. At the beginning of the time slot, r_1 points to output 1 while g_1 does not point to input 1, which means that in the last time slot, input 1 was not matched to output 1, and now input 1 requests output 1 for a new service. Similarly, r_2 requests output 3 for a new service. Since r_3 points to output 3 and g_3 points to input 3, it is possible that in the last time slot input 3 was matched to output 3 and in this time slot output 3 will transfer the next cell from input 3 because the VOQ is not empty. Input 4 and output 2 have a similar situation as input 3 and output 3. In the grant phase, output 1 grants the only request it receives from input 1 and updates g_1 to 1, output 2 grants a request from input 4 and output 3 grants a request from input 3. The request from input 2 to output 3 is not granted, so r_2

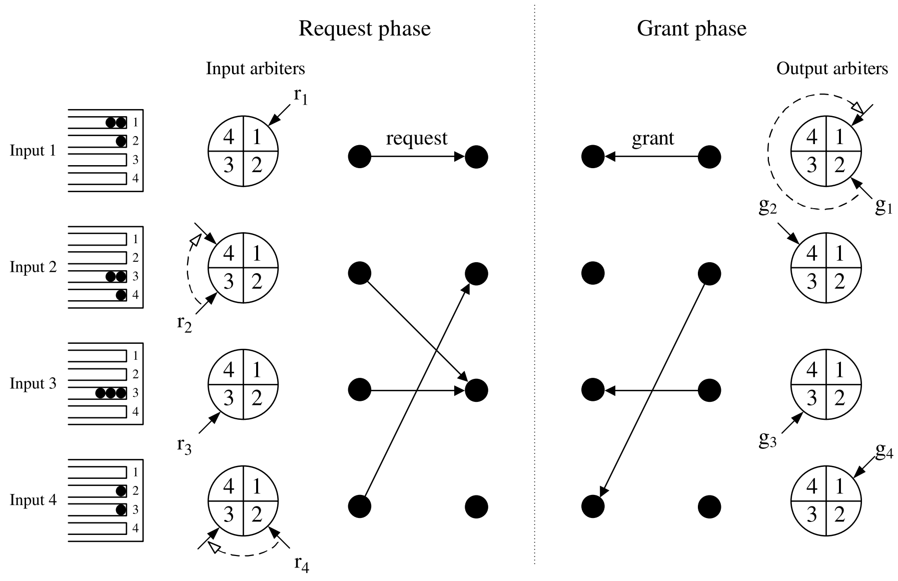


Figure 7.22 Example of EDRRM scheduling algorithm.

moves to 4. By the end of this time slot, the first VOQ of input 1 and the third VOQ of input 3 are still nonempty so that r_1 and r_3 are not updated. r_4 is updated to 3 because the second VOQ of input 4 has become empty.

7.4 RANDOMIZED MATCHING ALGORITHMS

Determining the maximum weight matching essentially involves a search procedure, which can take many iterations and be time-consuming. Since the goal is to design high-performance schedulers for high aggregate bandwidth switches, algorithms that involve, too many iterations are unattractive [22].

In this section, we will introduce several randomized matching algorithms using memory or arrivals. The usage of memory and arrivals are based on two observations:

Using memory. In each time slot, there can be at most one cell that arrives at each input, and at most one cell that departs from each input. Therefore, the queue length of each VOQ does not change much during successive time slots. If we use the queue length as the weight of a connection, it is quite possible that a heavy connection will continue to be heavy over a few time slots. With this observation, matching algorithms with memory use the match (or part of the match) in the previous time slot as a candidate of the new match.

Using arrivals. Since the increase of the queue length is based on the arrivals, it might be helpful to use the knowledge of recent arrivals in finding a match.

Since the decision is not based upon the complete knowledge of a large state space, but upon a few random samples of the state, the decision process can be significantly simplified.

By using a randomized matching algorithm, we may not find "the best" match, but the match could be good enough to make the switch stable.

7.4.1 Randomized Algorithm with Memory

The randomized algorithm with memory presented by Tassiulas [23] is a very simple matching scheme that achieves 100 percent throughput. The disadvantage of this algorithm is its high average delay.

At time t, let $Q(t) = [q_{ij}]_{N \times N}$, where q_{ij} is the queue length of VOQ_{ij}. The weight of a match $M(t)$, which is the sum of the lengths of all matched VOQs, is denoted by $W(t) = \langle M(t), Q(t) \rangle$.

Randomized Algorithm with Memory

1. Let $S(t)$ be the schedule used at time t.
2. At time $t+1$, uniformly select a match $R(t+1)$ at random from the set of all $N!$ possible matches.
3. Let

$$S(t+1) = \arg \max_{S \in \{S(t), R(t+1)\}} \langle S, Q(t+1) \rangle. \tag{7.6}$$

   The function of arg selects the S that makes $\langle S, Q(t+1) \rangle$ achieve its maximum in the above equation.

Theorem 4: The randomized algorithm with memory is stable under any Bernoulli i.i.d. admissible arrival traffic [23].

7.4.2 De-randomized Algorithm with Memory

In [21], a matching algorithm was presented to de-randomize the randomized algorithm with memory by using Hamiltonian walk.

A Hamiltonian walk is a walk which visits every vertex of a graph exactly once. For a $N \times N$ switch, the total number of possible matches is $N!$. If those matches are mapped on to a graph with $N!$ vertices so that each vertex corresponds to a match, a Hamiltonian walk on the graph visits each vertex exactly once every $N!$ time slots. The vertex that is visited at time t is denoted by $H(t)$. The complexity of generating $H(t+1)$ from $H(t)$ is $O(1)$ [24].

De-randomized Algorithm with Memory

1. Let $S(t)$ be the match used at time t.
2. At time $t+1$, let $R(t+1) = H(t)$, the match visited by the Hamiltonian walk.
3. Let

$$S(t+1) = \arg \max_{S \in \{S(t), R(t+1)\}} \langle S, Q(t+1) \rangle. \tag{7.7}$$

Theorem 5: An input-queued switch using the de-randomized algorithm with memory is stable under all admissible Bernoulli i.i.d. input traffic [22].

7.4.3 Variant Randomize Matching Algorithms

The fact that matching algorithms with simple ideas, such as randomized and de-randomized algorithms with memory, achieve 100 percent throughput as MWM does shows an important insight. That is, to achieve 100 percent throughput, it is not necessary to find "the best" match in each time slot. However, "better" matches do lead to better delay performance. Simulation results show that the randomized and de-randomized algorithms with memory have very high delay. In order to improve the delay performance, extra work is needed to find "better" matches. In [22], three algorithms with much better delay performance and higher complexity were proved to be stable.

APSARA. The APSARA algorithm [25] employs the following two ideas:

1. Use of memory.
2. Exploring neighbors in parallel. The neighbors are defined so that it is easy to compute them using hardware parallelism.

In APSARA, the "neighbors" of the current match are considered as candidates of the match in the next time slot. A match S' is defined as a neighbor of a match S if, and only if, there are two input–output pairs in S, say input i_1 to output j_1 and input i_2 to output j_2, switching their connections so that in S' input i_1 connects to output j_2 and i_2 connects to output j_1. All other input–output pairs are the same under S and S'. We denote the set of all the neighbors of a match S as $N(S)$. As shown in Figure 7.23 [22], the matching S for a 3×3 switch and its three neighbors $S1$, $S2$, and $S3$ are given below: $S = (1, 2, 3), S1 = (2, 1, 3), S2 = (1, 3, 2)$, and $S3 = (3, 2, 1)$.

Let $S(t)$ be the matching determined by APSARA at time t. Let $H(t + 1)$ be the match corresponding to the Hamiltonian walk at time $t + 1$. At time $t + 1$, APSARA does

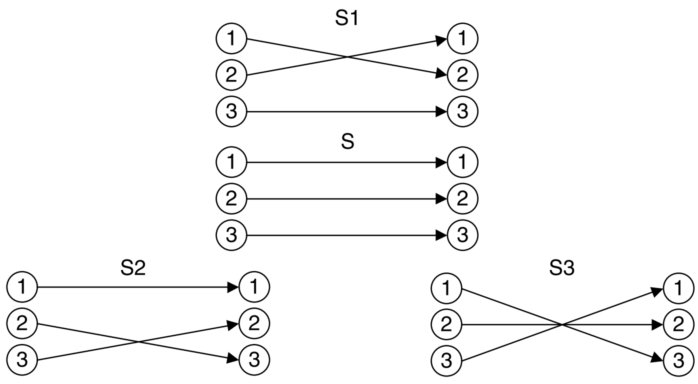


Figure 7.23 Example of neighbors in APSARA algorithms.

the following:

1. Determine $N(S(t))$ and $H(t)$.
2. Let $M(t+1) = N(S(t)) \cup H(t+1) \cup S(t)$. Compute the weight $\langle S', Q(t+1)\rangle$ for all $S' \in M(t+1)$.
3. Determine the match at $t+1$ by

$$S(t+1) = \arg \max_{S' \in M(t+1)} \langle S', Q(t+1)\rangle. \tag{7.8}$$

APSARA requires the computation of the weight of neighbors. Each such computation is easy to implement. However, computing the weights of all $\binom{N}{2}$ neighbors requires a lot of space in hardware for large values of N. To overcome this, two variations were considered in the work of Giaccone et al. [22] by reducing the number of neighbors considered in each time slot.

LAURA. There are three main features in the design of LAURA [25]:

1. Use of memory.
2. Nonuniform random sampling.
3. A merging procedure for weight augmentation.

Most of a matching's weight is typically contained in a few edges. Thus, it is more important to choose edges at random than it is to choose matchings at random. Equally, it is more important to remember the few good edges of the matching at time t for use in time $t+1$ than it is to remember the entire matching at time t [26].

The randomized and de-randomized algorithms with memory provide poor delay performance because they carry matches between time slots via memory. When the weight of a heavy match resides in a few heavy edges, it is more important to remember the heavy edges rather than the whole match. This observation motivates LAURA, which iteratively augments the weight of the current match by combining its heavy edges with the heavy edges of a randomly chosen match.

Let $S(t)$ be the match used by LAURA at time t. At time $t+1$ LAURA does the following:

1. Generate a random match $R(t+1)$ based on the procedure in [22].
2. Use $S(t+1) = MERGE(R(t+1), S(t))$ as the schedule for time $t+1$.

The MERGE Procedure. Given a bipartite graph and two matches $M1$ and $M2$ for this graph, the MERGE procedure returns a match M whose edges belong either, to $M1$ or to $M2$. MERGE works as follows and an example is shown in Figure 7.24.

Color the edges of $M1$ red and $M2$ green. Start at output node j_1 and follow the red edge to an input node, say i_1. From input node i_1 follow the (only) green edge to its output node, say j_2. If $j_2 = j_1$, stop. Otherwise, continue to trace a path of alternating red and green edges until j_1 is visited again. This gives a "cycle" in the subgraph of red and green edges.

Suppose the above cycle does not cover all the red and green edges. Then there exists an output j outside this cycle. Starting from j, repeat the above procedure to find another cycle. In this fashion, find all cycles of red and green edges. Suppose there are m cycles,

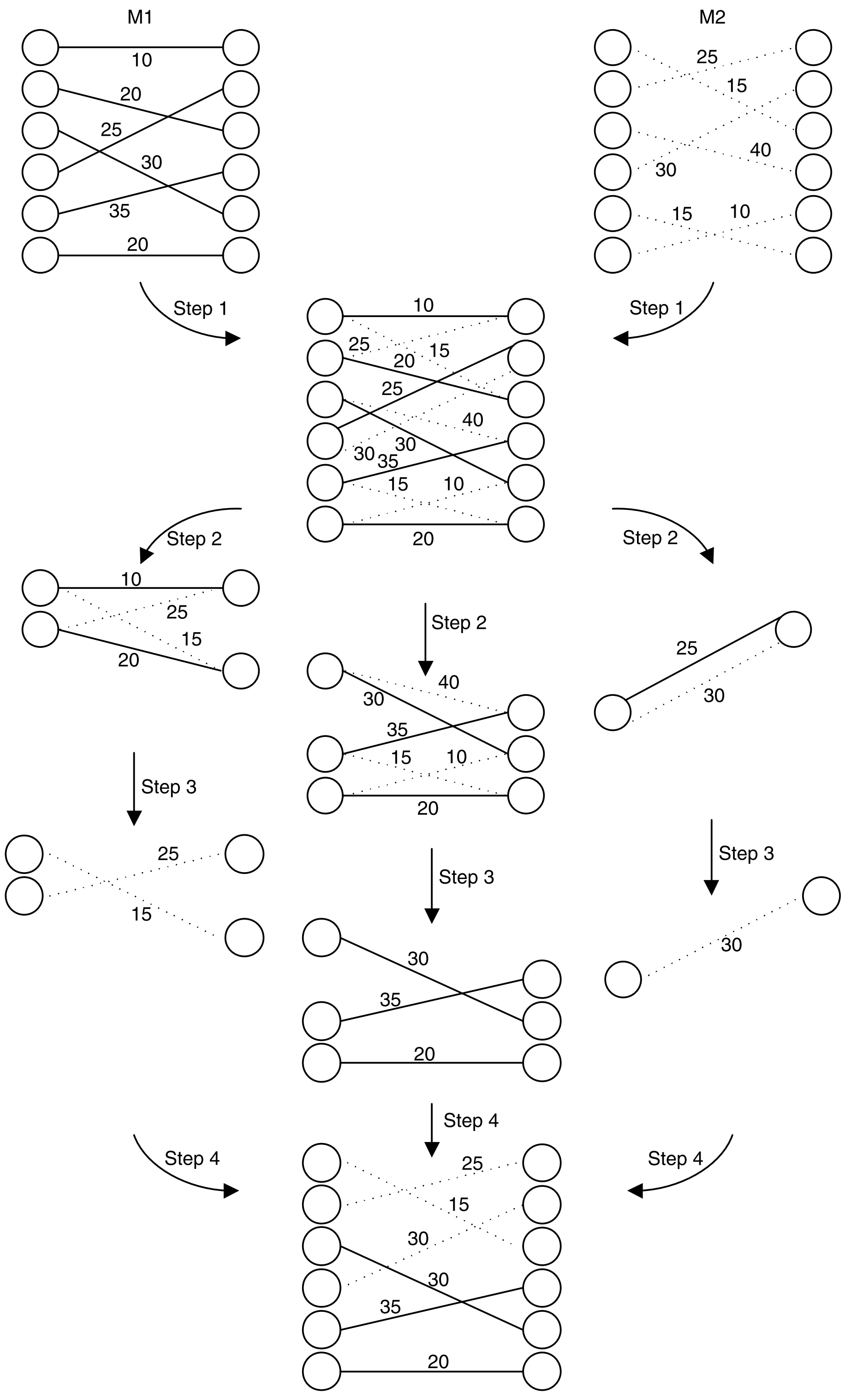


Figure 7.24 Example of the MERGE procedure.

$C_1, \dots, C_m$ at the end. Then each cycle, C_i, contains two matches: G_i with green edges and R_i with red edges. The MERGE procedure returns the match

$$M = \cup_{i=1}^{m} \arg \max_{S \in \{G_i, R_i\}} \langle S, Q(t) \rangle. \tag{7.9}$$

The complexity of the MERGE procedure is $O(N)$.

SERENA. SERENA is a variant of LAURA that uses packet arrival times as a source of randomness. The basic version of LAURA merges the past schedule with a randomly generated matching. In contrast, SERENA considers the edges that received arrivals in the previous time slot and merges them with the past matching to obtain a higher-weight matching [26].

SERENA [25] is based on the following ideas:

1. Use of memory.
2. Exploiting the randomness of arrivals.
3. A merging procedure, involving new arrivals.

In SERENA, in order to provide information about recent traffic load, arrival patterns are used to generate a new match.

Let $S(t)$ be the match used by SERENA at time t. Let $A(t+1) = [A_{ij}(t+1)]$ denote the arrival graph, where $A_{ij}(t+1) = 1$ indicates arrival at input i destined to output j. At time $t+1$:

1. Turn $A(t+1)$ into a full match.
2. Use $S(t+1) = MERGE(S(t), A(t+1))$ as the schedule.

In the second step, the MERGE procedure combines two full matches (with N connections each) into a new match. However, it is possible that $A(t+1)$ is not a match when more than one input has arrivals destined to one output. Therefore, it is necessary to convert $A(t+1)$ into a match in the first step as follows. If an output has more than one arrival edge, pick the edge with the highest weight and discard the remaining edges. At the end of this process, each output is matched with at most one input. After that, if $A(t+1)$ is not a full match, simply connect the remaining input–output pairs by adding edges in a round-robin fashion, without considering their weights.

The complexity of SERENA is $O(N)$. In the work of Giaccone et al. [22], the authors preferred SERENA against APSARA and LAURA because of its good performance and low implementation complexity.

7.4.4 Polling Based Matching Algorithms

A polling model [20] is a system of multiple queues accessed in a cyclic order by a single server. A polling system can have different service disciplines, such as the exhaustive service and the limited service. When the server switches to a queue, it serves some customers with a limited service discipline, while it serves all other customers with an exhaustive service discipline. Usually, the exhaustive service discipline is more efficient than other service disciplines.

Exhaustive service match with Hamiltonian walk (EMHW), presented in the work of Li [14] and Li et al. [27], is a class of matching algorithms inspired by exhaustive service polling systems. In an exhaustive service matching algorithm, when an input is matched to an output, all the packets waiting in the corresponding VOQ will be served continuously before any other VOQ related to the input and the output can be served. EMHW achieves stability and low packet delay with low implementation complexity.

EMHW is defined as follows:

1. Let $S(t)$ be the schedule used at time t.
2. At time $t+1$, generate a match $Z(t+1)$ by means of the exhaustive service matching algorithm, based on the previous schedule $S(t)$, and $H(t+1)$, the match visited by a Hamiltonian walk.
3. Let

$$S(t+1) = \arg \max_{S \in \{Z(t+1), H(t+1)\}} \langle S, Q(t+1) \rangle. \tag{7.10}$$

Theorem 6. An EMHW is stable under any admissible Bernoulli i.i.d. input traffic [14, 27].

The stability of EMHW is achieved due to two efforts. Unlike most other matching algorithms, which try to find efficient matches in each time slot, exhaustive service matching achieves efficiency by minimizing the matching overhead over multiple time slots. Cells forwarded to outputs are held in reassembly buffers that can only leave the switch when all cells belonging to the same packet are received so that the packet is reassembled. The total delay a packet suffers, from the time it arrives at the input to the time it departs at the output, includes the cell delay incurred traversing the switch and the time needed for packet reassembly. In exhaustive service matching, since all the cells belonging to the same packet are transferred to the output continuously, the packet delay is significantly reduced. The stability of EMHW is guaranteed by introducing matches generated by a Hamiltonian walk. This lower bounds the weight of matches, hence guaranteeing stability.

HE-iSLIP. Exhaustive schemes can be used in conjunction with existing matching algorithms, such as *i*SLIP. The time complexity of exhaustive service *i*SLIP with Hamiltonian walk (HE-*i*SLIP), which belongs to the class of stable algorithm EMHW, is as low as $O(\log N)$. Simulation results show that HE-*i*SLIP achieves very good packet delay performance.

In EMHW, an input (output) is busy if it is matched to an output (input), otherwise it is free. For exhaustive service *i*SLIP (E-*i*SLIP), at the beginning of each time slot, each input (output), which was busy (i.e., matched) in the previous time slot, checks its state by checking the corresponding VOQ. If the VOQ has just been emptied, the input (output) changes its state to free and increments its pointer to one location beyond the matched output (input). Otherwise, the input (output) keeps its state as busy and does not update its pointer. A detailed description of the three step E-*i*SLIP algorithm follows:

Step 1: *Request.* Each free input sends a request to every output for which it has a queued cell. Each busy input sends a request to the matched output.

Step 2: *Grant.* If an output (either free or busy) receives any requests, it chooses one of them in a fixed round-robin order starting from the current position of the pointer. The output notifies each input whether or not its request was granted. Note that the output pointer points to the granted input if the grant is accepted in Step 3.

Step 3: *Accept.* If an input receives any grant, it sets its state to busy, and accepts one of the multiple grants in a fixed round-robin order starting from the current position of the pointer. The input pointer then points to the matched output.

In E-*i*SLIP, free outputs only get requests from free inputs, and free inputs only get grants from free outputs.

Figure 7.25 shows an example of E-*i*SLIP in a time slot. By the end of the last time slot, input 1 is matched to output 2, and input 4 is matched to output 1. At the beginning of the current time slot, none of their corresponding VOQs was empty, and the other inputs and outputs were free. Therefore, input 1 and input 4 only send one request each, to output 2 and output 4, respectively. Input 2 and input 3 send requests for non-empty VOQs, to outputs 1 and 3, and outputs 3 and 4, respectively. Outputs 1 and 2 grant inputs to which they are matched, and inputs 1 and 4 accept their grants. Output 3 gets two requests. It grants input 3 according to the status of its round robin pointer (not shown). Output 4 only gets one request from input 3, and it grants the request. Input 3 receives two grants, and it accepts output 3 according to the status of its round robin pointer (again, not shown). Input 3 and output 3 are matched and change their state to busy. At the beginning of the next time slot, if input 1 does not have a new arrival to output 2, the corresponding VOQ will be empty and input 1 and output 2 will set their states to free.

HE-*i*SLIP does the following in each time slot $t + 1$:

1. Run E-*i*SLIP, which generates a match $Z(t + 1)$ based on the previous match $S(t)$ and updates the pointer and the state of each input and output.
2. Generate a match $H(t + 1)$ by Hamiltonian walk.

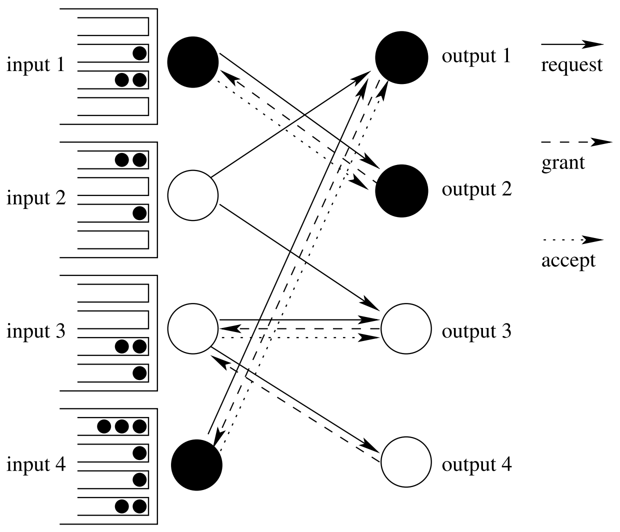


Figure 7.25 Example of E-*i*SLIP algorithm.

3. Compare the weights of $Z(t+1)$ and $H(t+1)$. If the weight of $H(t+1)$ is larger, set $S(t+1) = H(t+1)$. For each matched input (output), set its state to busy, and update its pointer to the output (input) with which it is matched. Unmatched inputs and outputs set their states to free and do not update their pointers. If the weight of $Z(t+1)$ is larger, simply set $S(t+1) = Z(t+1)$. No further pointer or state updating is needed since it has been done in the first step.

Cell Delay Performance Analysis. The cell delay performance of E-*i*SLIP under uniform traffic can be analyzed by using an exhaustive random polling system model [28]. The expression of the average cell delay $E(T)$ is as follows:

$$E(T) = \frac{1}{2}\left[\frac{\delta^2}{r} + \frac{\sigma^2}{(1-N\mu)\mu} + \frac{Nr(1-\mu)}{1-N\mu} + \frac{(N-1)r}{1-N\mu}\right] \tag{7.11}$$

where μ is the arrival rate for a VOQ, σ^2 is the variance of the arrival process for a VOQ, and $r = E(S)$, $\delta^2 = E(S^2) - E^2(S)$. Here S is the switch-over time, the time taken for the server to switch from one VOQ after service completion to another VOQ for a new service period. The expressions of $E(S)$ and $E(S^2)$ are as below. The details can be found in the work of Li [14] and Li et al. [29].

$$E(S) = \sum_{n=1}^{\infty} n(1-Q_S)^n Q_S = \frac{1-Q_S}{Q_S}. \tag{7.12}$$

and

$$E(S^2) = \sum_{n=1}^{\infty} n^2(1-Q_S)^n Q_S = \frac{1-Q_S}{Q_S}\left[\frac{2(1-Q_S)}{Q_S} + 1\right], \tag{7.13}$$

where

$$\begin{aligned} Q_S &= \sum_{m=1}^{N} \binom{N-1}{m-1} \rho^{N-m}(1-\rho)^{m-1}[1-(1-\rho)^m] \sum_{i=0}^{m-1} \binom{m-1}{i} (1-w)^{m-i-1} w^i \frac{1}{i+1} \\ &= 1 - \sum_{m=1}^{N} \binom{N-1}{m-1} \rho^{N-m}(1-\rho)^{m-1}(1-w)^m \end{aligned} \tag{7.14}$$

and

$$w = \frac{1}{m}\left[1-(1-\rho)^m\right]. \tag{7.15}$$

When N is large,

$$E(T) \to E(S)\,\frac{N}{1-\rho}. \tag{7.16}$$

Numerical results show that for all $\rho < 1$, the average switch-over time, $E(S)$, is around 0.58 time slots when N is large. Therefore, for a fixed ρ, $E(T)$ is linear in N when N is large.

7.4.5 Simulated Performance

In this section, we will show some simulation results under uniform and nonuniform arrival traffic for the delay performance of a 32 by 32 VOQ switch with benchmark stable algorithms, MWM, *i*SLIP, the de-randomized matching algorithm, SERENA and HE-*i*SLIP with implementation complexity $O(N^3)$, $O(\log N)$, $O(\log N)$, $O(N)$ and $O(\log N)$, respectively.

In fixed-length switches, variable-length IP packets are segmented into fixed-length cells at the inputs, and the cells are placed in the corresponding VOQ. When a cell is transferred to its destination output, it will stay in a buffer and wait for the other cells of the same packet. After the complete reception of all the cells of the same packet, these cells will be reassembled into a packet. The delay a packet suffers before it is reassembled into a packet and delivered to its destination includes the cell delay, and the waiting time at the output reassembly buffer, which is often ignored by many researchers. For a more realistic evaluation of switch performance, we consider the following average delays in our simulation.

Cell delay. The delay a cell suffers from the time it enters the system to the time it is transferred from the input to its destined output.

Reassembly delay. The delay a cell suffers from the time it is transferred to it destined output to the time it is reassembled and departs the system.

Packet delay. As in [30], the packet delay of a packet is measured from the time when the last cell of a packet enters the system to the time it departs.

Under Uniform Traffic. Three different packet patterns are considered in this section, as follows:

Pattern 1. The packet length is fixed with a size of 1 cell. This also allows for comparison with the cell delay used in many papers.

Pattern 2. The packet length is fixed with a size of 10 cells.

Pattern 3. Based on the Internet traffic measurements by Claffy et al. [31], where 60 percent of the packets are 44 bytes, 20 percent are 552 bytes, and the rest are 1500 bytes. In the simulation, the packet size distribution is defined as follows: the size of 60 percent of the packets is 1 cell, the size of 20 percent of the packets is 13 cells, and the size of other 20 percent packets is 34 cells. This assumes a cell payload of 44 bytes. The average packet size is 10 cells.

Simulation results under packet patterns 1, 2, and 3 are shown in Figures 7.26, 7.27, and 7.28, respectively.

The packet delay of the de-randomized matching algorithm is always much higher than 10,000 cell time slots and are therefore not shown in the figures. We can see that under uniform traffic, the packet delay of HE-*i*SLIP is always lower than *i*SLIP and SERENA. In the figures, SERENA has the highest packet delay when the traffic load is low to moderately high. Under heavy load, *i*SLIP has the highest packet delay. MWM has a higher packet delay than HE-*i*SLIP for packet patterns 2 and 3.

As shown in Figure 7.26, MWM has the lowest delay when the packet size is 1 cell, but has higher packet delay than HE-*i*SLIP when the average packet size is >1, as shown in Figures 7.27 and 7.28. Figures 7.29 and 7.30 explain why this happens. The cell delay

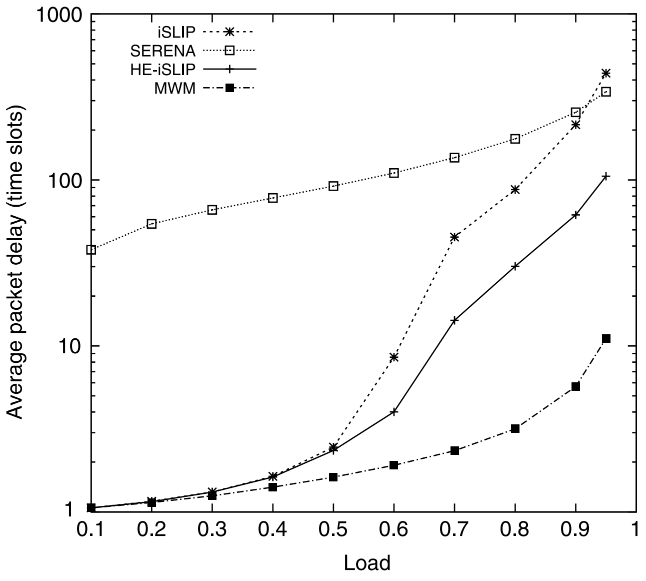


Figure 7.26 Average packet delay of *i*SLIP, SERENA, HE-*i*SLIP and MWM under uniform traffic when the packet length is 1 cell.

of MWM is always lower than HE-*i*SLIP, but its reassembly delay is much higher. In HE-*i*SLIP, the cells in the same packet are usually transferred continuously. The only exception is when the match generated by Hamiltonian walk is picked because of its larger weight. However, this does not happen very often. Therefore, the reassembly delay of HE-*i*SLIP is always close to half of the packet length.

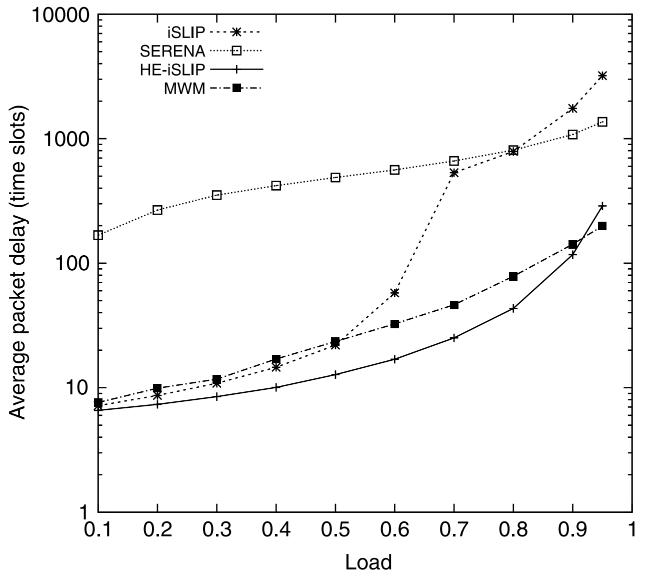


Figure 7.27 Average packet delay of *i*SLIP, SERENA, HE-*i*SLIP and MWM under uniform traffic when the packet length is 10 cells.

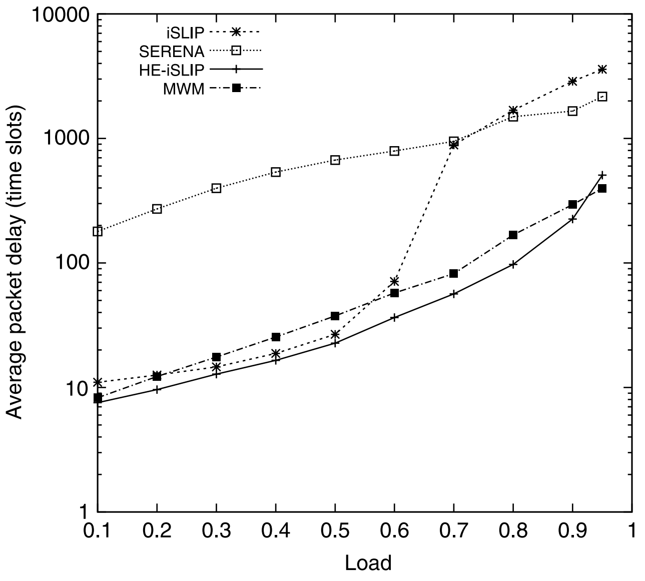


Figure 7.28 Average packet delay of *i*SLIP, SERENA, HE-*i*SLIP and MWM under uniform traffic with variable packet length.

Performance Under Nonuniform Traffic. Two typical nonuniform traffic patterns, diagonal and hotspot, are considered in this section. The packet length is assumed to be 1 cell in the simulation. *i*SLIP is not included in the performance comparison since it is not stable under nonuniform traffic.

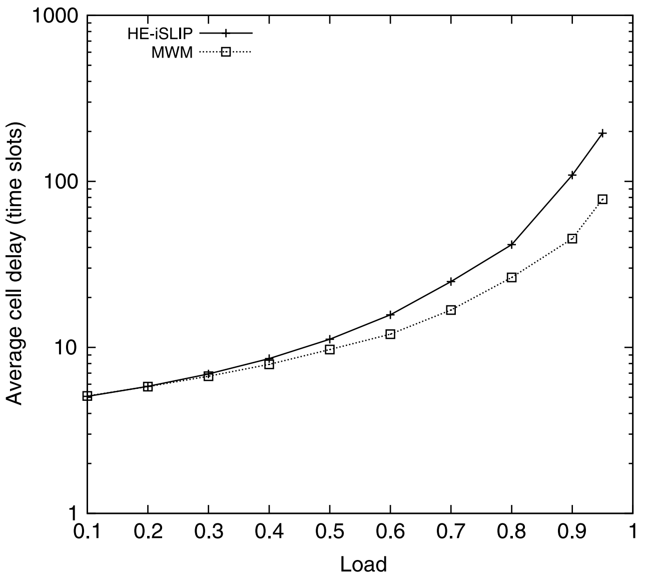


Figure 7.29 Average cell delay of HE-*i*SLIP and MWM under uniform traffic when the packet length is 10 cells.

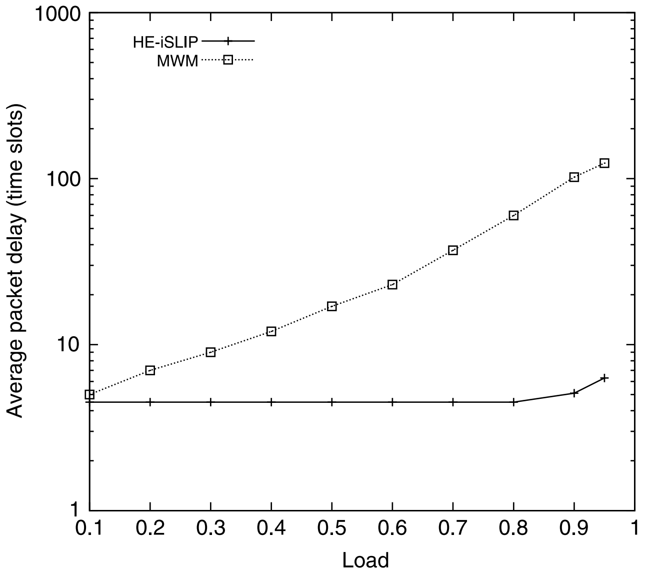


Figure 7.30 Average reassembly delay of HE-*i*SLIP and MWM under uniform traffic with variable packet length.

With the diagonal traffic pattern [21, 30], the arrival rate for each input is the same. For input i a fraction f of arrivals are destined to output i, and other arrivals are destined to output $(i + 1) \mod N$.

Table 7.1 shows the average delays, under diagonal traffic, of the de-randomized matching algorithms, SERENA, HE-*i*SLIP and MWM, respectively. The arrival rate to each input is 0.85. MWM has the best delay performance under diagonal traffic. The delay of SERENA is lower than HE-*i*SLIP when f is large and similar to HE-*i*SLIP when f is small. The delay of the de-randomized matching algorithm is much higher than those of other schemes.

In the hotspot traffic pattern, the arrival rate for each input is identical. For input i, a fraction p, $1/N \leq p < 1$, of arrivals are destined to output i, and other arrivals are uniformly destined to other outputs [21, 29]. Figure 7.31 shows the average delay of HE-*i*SLIP and SERENA for a 32×32 switch for different values of p when the arrival rate is 0.95 and the packet size is 1 cell. The delay for the de-randomized algorithm with memory is, too high and therefore not shown in the figure. The simulation results show that HE-*i*SLIP always has a lower delay than SERENA, but higher delay than MWM.

Compared to HE-*i*SLIP, the delay of SERENA is lower under diagonal traffic but higher under the hotspot traffic pattern. This can be explained by the fact that SERENA takes

TABLE 7.1 Average Delay of a 32 x 32 Switch Under Diagonal Traffic Pattern

f	0.1	0.2	0.3	0.4	0.5
De-randomized	1164	425.4	532.1	369.2	371.9
SERENA	3.945	6.606	8.570	9.517	10.08
HE-*i*SLIP	3.167	8.502	30.54	56.22	86.13
MWM	1.848	2.532	3.115	3.337	3.440

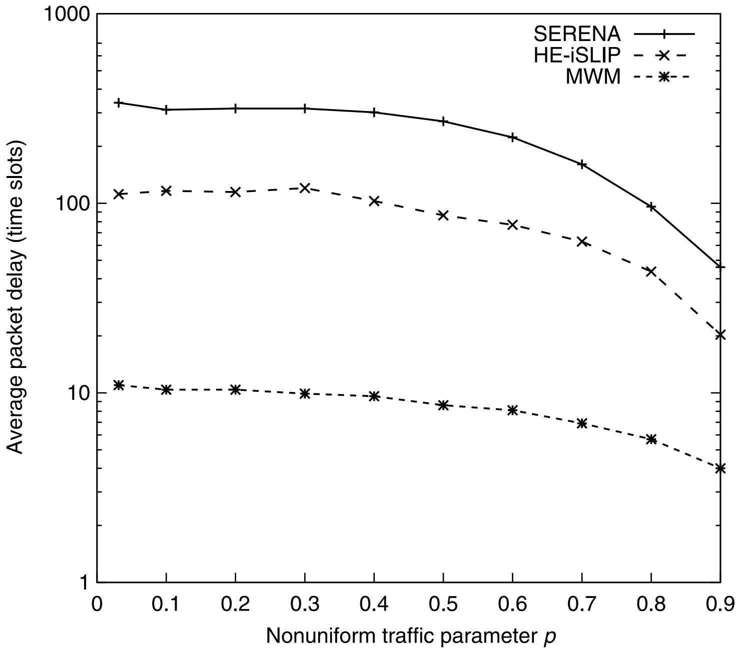


Figure 7.31 Average packet delay of SERENA, HE-*i*SLIP and MWM under the hotspot traffic pattern.

the arrival pattern at each time slot into account to generate the new match. However, if there is more than one arrival destined to the same output, only one of them, which is randomly selected, can be considered. Under diagonal traffic, only two inputs can have traffic to a given output. This makes it relatively easy for a new match to adapt to the arrival pattern. Indeed, SERENA is particularly suitable for a traffic pattern with which each output is always fed by only a few inputs. When traffic pattern is such that many inputs feed a given output, the SERENA algorithm is less effective, as in the hotspot or uniform traffic pattern case, since arrivals in a given slot give less indication of a good match.

7.5 FRAME-BASED MATCHING

By taking advantage of the tremendous transmission and switching capacity of optical fibers, many researchers have explored the possibility of building optical switching fabrics while packet processing and storage are still handed by electronics. As compared to electronic switch fabrics, a pair of E/O & O/E converters can be saved in optical switch fabrics, thus reducing the system cost.

Packet scheduling in optical switch fabrics is quite different from that in electronic switch fabrics as shown in Figure 7.32. In an electronic switch fabric, the switch fabric is reconfigured without any delay. Cells can be transferred back-to-back, and packet scheduling has to be completed before the switch configuration. But in an optical switch fabric, the switch reconfiguration is much slower, and is not negligible. During the switch

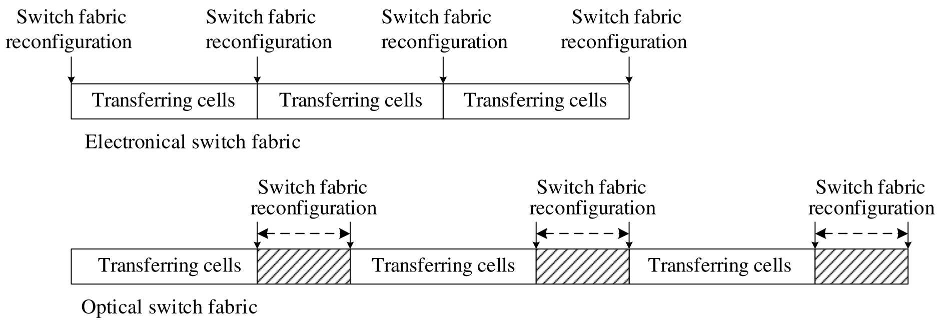


Figure 7.32 Comparison of operations between an electronic and optical switch fabric.

fabric reconfiguration, no data can be transferred. These reconfiguration overheads range from milliseconds for bubble and free-space MEMS (Microelectromechanical Systems) switches, to 10 μs for MEMS wavelength switches, and 10 ns for opto-electronic switch techniques [32].

Taking into account the non-zero reconfiguration overheads, a different framework for packet scheduling in optical switches has been studied. Instead of scheduling at every time slot, multiple cells are merged together to form a large frame and scheduled and switched as a group, so as to reduce the switching overhead. This type of scheduling algorithms is called frame-based matching algorithms. The following questions are then raised: How many cells should we schedule and switch together? How can we schedule full frames and partially filled frames? What is the performance difference between traditional algorithms and frame-based matching algorithms? Frame-based matching has the following advantages:

1. Low overhead. For a 40 Gbps interface, a typical cell size (64 byte) only lasts 12.8 ns. If the reconfiguration overhead is 20 ns, the switch utilization is only about 40 percent. But if 100 cells form a frame and are switched together, the switch utilization is improved to 98 percent.
2. The time required for scheduling frames is much more relaxed and more complex but higher performance scheduling algorithms can be explored in frame-based matching. With the above example, the scheduling time can be extended from 12.8 ns to 128 ns.

Research on frame-based matching is also important for electronic switch fabrics. As the line speed increases from 10 Gbps to 40 Gbps to 100 Gbps or more, the cell slot becomes smaller and smaller, and frame-based matching can be used to eliminate the constraints with the price of a larger delay.

In the following, we introduce some work related to frame-based matching including reducing the reconfiguration frequency, fixed-size frame-based matching, and asynchronous variable-size frame-based matching.

7.5.1 Reducing the Reconfiguration Frequency

The objective of frame-based matching is to lower the bandwidth waste by reducing the reconfiguration frequency. The algorithms presented in [32] can exactly emulate an unconstrained (zero overhead) switch in a switch with reconfiguration overhead, such as an optical switch. That is, any scheduling algorithms designed for an unconstrained switch can be exactly emulated in an optical switch fabric. The emulation is executed in the following steps. First, in an unconstrained switch, acknowledged requests within T time slots are accumulated. Second, in the switch with reconfiguration overhead, an algorithm is executed, and a set of N_s switch schedules are generated to acknowledge the same batch of requests. Finally, at each output port, packets are reordered to exactly emulate the same departure process in the unconstrained switch.

For instance, suppose $T = 5$ in a 2×2 switch and the unconstrained switch fabric configurations in consecutive T time slots are

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

In another way, we can keep the switch fabric configuration

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

for three time slots, and then

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

for two time slots. The two types of configuration can transfer the same cells over the five time slots from inputs to corresponding outputs, but the second type only requires one reconfiguration of the switch fabric. Obviously, in a 2×2 switch, some combinations of the two basic switch configurations

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

can achieve the same scheduling results for any given T. Then what is the minimum N_s in a $N \times N$ switch?

Exact Covering. The problem of emulation is identical to the "cover" problem in math.

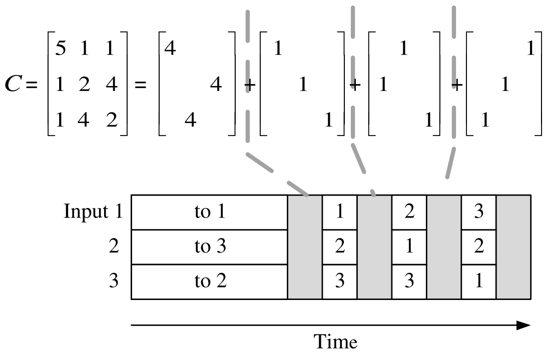
Cover: Matrix A is covered by a set of switch configurations $P(1), \ldots, P(N_s)$ and corresponding weights $\phi(1), \ldots, \phi(N_s)$ if

$$\sum_{k=1}^{N_s} \phi(k) p_{i,j}(k) \geq a_{i,j}, \qquad \forall i, \quad j \in \{1, \ldots, N\},$$

where $a_{i,j}$ and $p_{i,j}(k)$ are elements of matrix A and $P(k)$, respectively, and $\phi(k)$ is the weight of a switch configuration matrix $P(k)$. $P(k)$ is a $N \times N$ permutation matrix, which means each element of $P(k)$ is, either 0 or 1, and there is only one 1 in each row and each column.

In the case of equality for all i and j, the switch configurations exactly cover A.

For a given $N \times N$ switch fabric, if all the requests are cumulated to form a matrix $C(T)$, where $C(T)$ is a matrix whose rows and columns sum to T. The number of necessary and sufficient switch configurations N_s that can exactly cover any $C(T)$ is $N^2 - 2N + 2$.



$C = \begin{bmatrix} 5 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 4 & 2 \end{bmatrix} = \begin{bmatrix} 4 & & \\ & & 4 \\ & 4 & \end{bmatrix} + \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} + \begin{bmatrix} & 1 & \\ 1 & & \\ & & 1 \end{bmatrix} + \begin{bmatrix} & & 1 \\ & 1 & \\ 1 & & \end{bmatrix}$
Input 1
2
3
to 1
to 3
to 2
1 2 3
2 1 2
3 3 1
Time

Figure 7.33 Example of exact covering $\{T = 7\}$.

Figure 7.33 shows an example of exact covering with $T = 7$. The matrix C is exactly covered with four switch configurations. The switch configuration

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

is kept for four time slots and others are one time slot.

Complex algorithms that achieve an exact covering are typical and can be found in the work of Inukai [33] and Chang et al. [34]. They perform $O(N^2)$ maximum size matchings, and the complexity of each maximum size matching is $O(N^{2.5})$, so the complexity of exact covering is $O(N^{4.5})$.

In the exact covering, there are no empty time slots during transferring cells. Thus the required speedup to compensate for empty slots speedup $S_{\min} = 1$, and the total speedup of the switch fabric $S = T/[T - \delta(N^2 - 2N + 2)]$, where δ is the switch reconfiguration overhead in time slots.

Minimum Switching. Exact covering provides a bound where $S_{\min} = 1$, but $O(N^2)$ switch configurations are needed. Whereas, the minimum switching configurations that can cover any arbitrary matrix C is N. These N permutation matrixes do not have any 1s in the same row and the same column.

Obviously, if none of the elements of C is zero, any switching configurations that are less than N cannot cover C. For example, the matrix

$$\begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$$

cannot be covered with less than two different switch configurations.

For the same matrix shown in Figure 7.33, it can be covered with three configurations as shown in Figure 7.34. These switch configurations do not exactly cover matrix C, and there are empty time slots, so the speedup is required to compensate these empty time slots. To transmit a general cumulative schedule matrix $C(T)$ in N switch configurations, $S_{\min}$ must be at least $\Omega(\log N)$ for $T > N$. At the same time, to cover a general cumulative schedule matrix $C(T)$ with N switch configurations, $S_{\min} = 4T(4 + \log_2 N)$ is sufficient.

$$C=\begin{bmatrix}5&1&1\\1&2&4\\1&4&2\end{bmatrix}=\begin{bmatrix}5&&\\&&4\\&4&\end{bmatrix}+\begin{bmatrix}&1&\\1&&\\&&2\end{bmatrix}+\begin{bmatrix}&&1\\&2&\\1&&\end{bmatrix}$$

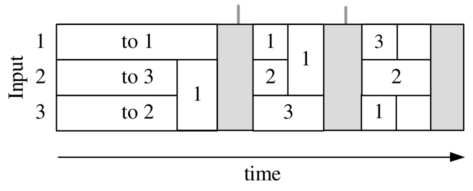


Figure 7.34 Example of minimum covering $\{T = 9\}$.

The algorithm with N configurations is also proposed in the work of Towles and Dally [32]. The algorithm is mainly made of N maximum size matchings and thus the total time complexity is $O(N^{3.5})$.

DOUBLE. As described in the previous section, using the minimal number of switchings requires a speedup of at least $\log N$. In this section, we show that by allowing $2N$ switchings, the minimum speedup $S_{\min}$ can be reduced to approximately two. Most importantly, the minimum speedup is no longer a function of N. This approach has the advantage of the EXACT algorithm, a small constant speedup, combined with a number of switchings that grows linearly with N.

DOUBLE [35] algorithm works as follows:

Step 1. *Split C.* Define and $N \times N$ matrix A such that $a_{i,j} = \lfloor c_{i,j}/(T/N) \rfloor$.

Step 2. *Color A.* Construct the bipartite multigraph G_A from A (the number of edges between vertices is equal to the value of the corresponding entry of A). Find a minimal edge-coloring of A. Set $i \leftarrow 1$.

Step 3. *Schedule coarse.* For a specific color in the edge-coloring of G_A, construct a switch configuration $P(i)$ from the edges assigned that color. Set $\phi(i) \leftarrow \lceil T/N \rceil$ and $i \leftarrow i + 1$. Repeat step 3 for each of the colors in G_A.

Step 4. *Schedule fine.* Find any N nonoverlapping switch schedules $P(N + 1), \ldots, P(2N)$ and set $\phi(N + 1), \ldots, \phi(2N)$ to $\lceil T/N \rceil$.

DOUBLE works by separating C into coarse and fine matrices and devotes N configurations to each. The algorithm first generates the coarse matrix A by dividing the elements of C by T/N and taking the floor. The rows and columns of A sum to at most N, thus the corresponding bipartite multigraph can be edge-colored in N colors. Each subset of edges assigned to a particular color forms a matching, which is weighted by $\lceil T/N \rceil$. The fine matrix for C does not need to be explicitly computed because its elements are guaranteed to be less than $\lceil T/N \rceil$. Thus, any N configurations that collectively represent every entry of C, each weighted by $\lceil T/N \rceil$, can be used to cover the fine portion.

An example execution of DOUBLE is shown in Figure 7.35. The algorithm begins by creating the coarse matrix A by dividing each element in C by T/N and taking the floor. So, in the example, entry (1,1) of A contains $\lfloor 16/(T/N) \rfloor = 16/(16/4) = 4$. The resulting matrix A has row and column sums ≤ 4, ensuring that it can be edge colored with four colors

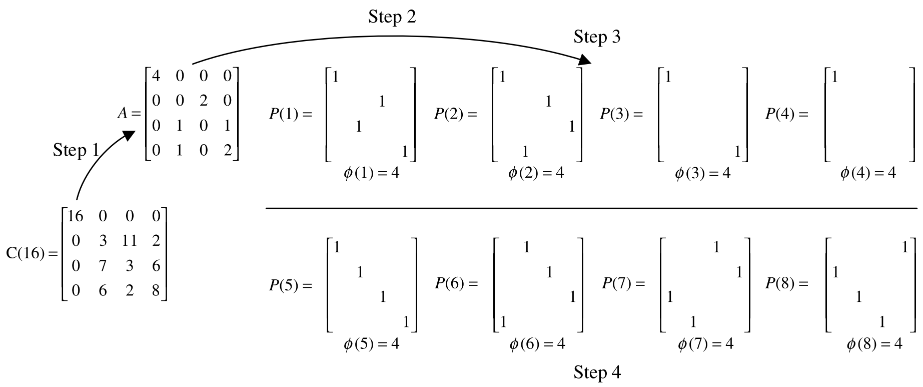


Figure 7.35 Example of DOUBLE $\{N = 4, T = 16\}$.

(step 2). Then, the edges assigned to each color are converted to schedules in step 3. For example, $P(1)$ corresponds to the subset of edges assigned to color 1 during step 2. Also, some of the schedules may not be complete permutations because the row and column sums of A are less than N, such as $P(3)$ and $P(4)$, but it is still guaranteed that all the elements of A are covered. In general, step 3 creates at most N matchings with weight $\lceil T/N \rceil$, for a total weight of approximately T.

Step 4 picks four nonoverlapping schedules, $P(5)$ through $P(8)$, and each is assigned a weight of $\lceil T/N \rceil = 4$. In general, step 4 creates the same total weight as step 3: approximately T. Therefore, the total weight to schedule $C(T)$ using DOUBLE is approximately $2T$ and $S_{\min} = 2$.

In brief, exact covering generates $N_s = N^2 - 2N + 2$ schedules for every T time slots, and no speedup is required. The complexity of exact covering is $O(N^{4.5})$. Minimum switching generates only N schedules to cover the batch of requests, but requires a speedup of $\Omega(\log N)$. The complexity of minimum switching is $O(N^{3.5})$. DOUBLE generates $N_s = 2N$ schedules for every T time slots and a speedup of two is required. The DOUBLE algorithm produces schedules with these properties in $O(N^2 \log N)$ time using the edge-coloring algorithm of [36]. When the port number N and the reconfiguration overhead δ are large, the delay of exact covering will not be accepted. When the bandwidth is expensive, the $\Omega(\log N)$ speedup of minimum switching will cause, too much bandwidth waste. DOUBLE will be a better tradeoff between delay and bandwidth for a large range of N and T.

7.5.2 Fixed Size Synchronous Frame-Based Matching

Li et al. [37] have described a fixed-size synchronous frame-based matching scheme, where every K time slots are grouped into a frame. When $K = 1$, the frame-based matching becomes the cell-based matching. In a frame-based matching, we can assume $K \gg 1$. The matching is computed for each frame and the switch fabric is only updated on frame boundaries. Since the time to compute the new matching set is not limited to one time slot, more complex scheduling can be executed in frame-based matching.

Figure 7.36 shows the working process of the synchronous frame-based matching. Each matching for the next frame is computed in m time slots and the result is available at the end of each frame. Then, at the beginning of the next frame, L time slots are needed to reconfigure the switch fabric. In the following $K - L$ time slots, valid packets are switched from matched

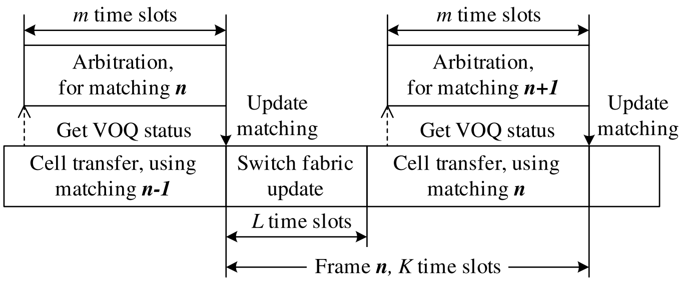


Figure 7.36 Fixed-size synchronous frame-based matching scheme.

input ports to output ports. L time slots overhead are introduced by reconfiguration in each frame. At least $K/(K-L)$ speedup is required to compensate the bandwidth during the reconfiguration. In the following, three kinds of fixed-size synchronous frame-based matching schemes are studied.

Frame-Based Maximum Weight Matching. In the cell-based scheduling, maximum weight matching (MWM) has been proved to be stable for any admissible traffic that satisfies the strong law of large numbers (SLLN). Naturally, MWM may be used in frame-based matching. But in frame-based matching, we cannot achieve maximum weight at each time slot. Instead, as shown in Figure 7.36, we can compute a matching that achieves the maximum weight at the time slot of getting VOQ status. The frame-based MWM is still stable under any admissible Bernoulli i.i.d. traffic.

The complexity of MWM is $O(N^3)$, which is not practically implemented even with the relaxed timing available under frame-based matching.

Frame-Based Maximal Weight Matching. Maximal weight matching algorithm is proposed to approximate MWM with less complexity. The most straightforward maximal weight matching algorithm is to sort all N^2 VOQs by their weight and always select the VOQs with the largest weight for service. Ties can be broken randomly. The complexity of this sorting operation is $O(N^2 \log N)$. Sorting VOQs in a distributed manner at each input line card can further reduce the time complexity. The details of this algorithm are as follows.

Step 1: At m time slots before a new frame starts, get weights for all VOQs. At each input, sort all N VOQs by their weights in decreasing order. Let $h = N$.

Step 2: Consider the h VOQs at the top of the h sorted lists, and select the one with the largest weight and match the corresponding input and output. Delete the sorted list of the corresponding input and all VOQs destined to the corresponding output.

Step 3: $h = h - 1$. If $h > 0$, go to step 2.

Step 4: Update the matching set at the boundary of a new frame.

The complexity of step 1 is $O(\log N)$. Step 2 also takes $O(\log N)$ steps, and at most N executions are needed. So the total time complexity is $O(N \log N)$.

Figure 7.37 shows an example of frame-based maximal weight matching where the port number is four. During iteration 1, the weight of $L(1,2)$ is the maximum, so the match

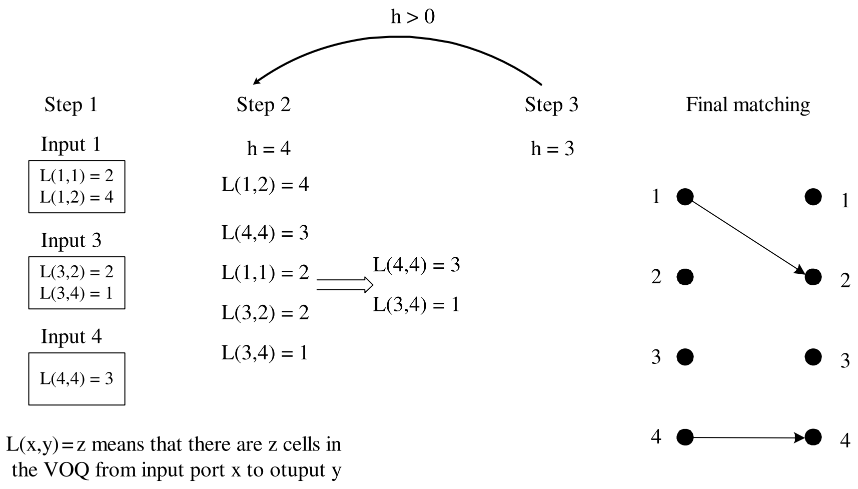


Figure 7.37 Example of frame-based maximal weight matching.

between input port 1 and output 2 is established. Then $L(1,2)$, $L(1,1)$, and $L(3,2)$ are removed, and only $L(4,4)$ and $L(3,4)$ are left in the following iterations.

Frame-Based Multiple Iteration Weighted Matching. In a cell-based electronic switch, practical matching schemes, such as PIM, iSLIP, and DRRM, use multiple iterations to converge on a maximal size matching. Similarly, multiple iterative weighted matching scheme, such as longest queue first (*i*LQF) and oldest queue first (*i*OQF), converge on a maximal weight matching within $\log N$ iterations.

In an optical switch, a frame-based multiple iteration weighted matching can be used to converge on a frame-based maximal weight matching. Simulation results show that by using $\log N$ iterations, the schedule can achieve almost the same performance as that of a frame-based maximal weight matching. A frame-based multiple iterative weighted matching scheme works as the following in every iteration.

Step 1: Each unmatched input sends requests for all nonempty VOQs along with their weights.

Step 2: If an unmatched output receives any request, it selects the request with the largest weight and sends a grant to the corresponding input. Ties are broken randomly.

Step 3: If an unmatched input receives multiple grants, it accepts the grant corresponding to the largest weight. Ties are broken randomly. The corresponding input and output port are set to be matched and excluded from the following iterations.

Figure 7.38 shows an example of frame-based multiple iteration weighted matching. The value upon each edge is the weight from an input to the corresponding output. Each output grants an input with the highest weight, and each input accepts an output with the highest weight.

The complexity of one iteration is $O(\log N)$, and $\log N$ iterations are needed to converge the frame-based maximal weight matching. Therefore, the complexity of this scheme is as low as $O(\log^2 N)$. Moreover, this algorithm can be implemented in a distributed manner and is more practical.

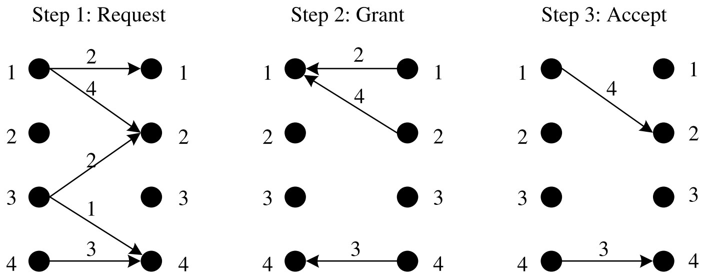


Figure 7.38 Example of frame-based maximal weight matching.

When the configuration time L is set to 0, our simulation results show that the throughput of a frame-based MWM algorithm under uniform traffic is 100 percent, and is close to 100 percent under all non-uniform traffic patterns considered by Li et al. [21] except the diagonal traffic pattern. According to previous studies, the diagonal traffic pattern is an extremely unbalanced nonuniform pattern, in which for an input i, a ratio p of arrivals is destined to output i, and $1 - p$ to output $(i + 1) \bmod N$. Switches usually have worse performance under diagonal traffic than under other more balanced traffic patterns. The throughput of frame-based maximal weight matching under diagonal traffic is always higher than 88 percent. Moreover, the simulated throughput performance of the frame-based multiple iteration weighted matching is almost the same as that of the frame-based maximal weight matching.

Delay performance is measured in time slots, where one time slot is the time to transfer one fixed-size packet. The performance of frame-based multiple iteration weighted matching is quite close to the performance of the frame-based maximal weight matching. Figure 7.39 shows the average delay of the frame-based maximal weight matching for different switch sizes when the reconfiguration time L is 0 or 10, and the frame length K is 50 time slots. We can infer that under light load, the average delay with nonzero L is close to the sum of L and the average delay when L is zero. Note that the average delay is close to $NK/2$. This result is consistent with the intuition that under a moderate load regime, the arriving packet has to wait for $N/2$ frames before its own VOQ is served. Therefore, when K becomes large, the measured delay will increase linearly. For example, when $K = 200$, the corresponding delay is about four times than shown in Figure 7.39.

If a cell slot time is 100 ns, a delay of 1400 time slots is about 140 μs. Therefore, a future reduction in switch reconfiguration time to 10 μs will make it feasible. Given a switch size N, reconfiguration time L and the required delay, the corresponding frame size can then be determined.

7.5.3 Asynchronous Variable-Size Frame-Based Matching

In fixed-size synchronous frame-based matching schemes, the switch fabric is updated for each frame instead of each time slot to reduce the reconfiguration frequency. However, if only part of the connections is reconfigured when necessary and others keep transferring cells, which is corresponding to asynchronous frame matching, the overhead can be reduced further, especially under heavy offered load. This variation is feasible. For instance, MEMS-based optical switches can reconfigure a subset of input and output ports while the other input-output ports continue to switch packets. Exhaustive service matching and limited

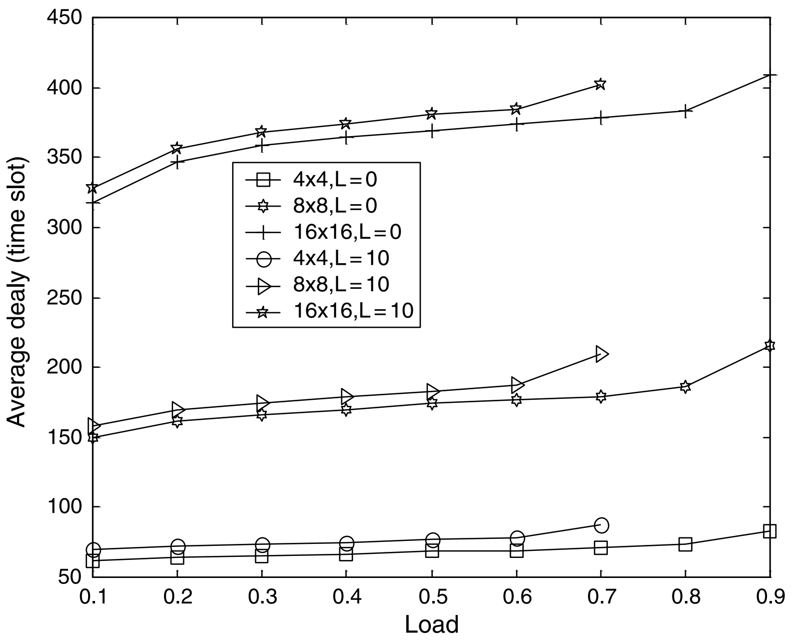


Figure 7.39 Average delay of frame-based maximal weight matching switch under uniform traffic when $K = 50$.

service matching algorithms [21] can be extended for asynchronous variable-size frame matching [37]. Under uniform traffic, we can expect that exhaustive service matching and limited service matching will lead to similar performance. Under nonuniform traffic, limited service matching can avoid unfairness and instability.

For an optical switch, exhaustive service dual round-robin matching (EDRRM) can be modified so that arbitration will only be done when necessary. In the original EDRRM, arbitration is done by input arbiters and output arbiters based on the round-robin service discipline. When an output is matched to an input, this output is locked by the input. When the VOQ under service is emptied, the corresponding input sends a message to the

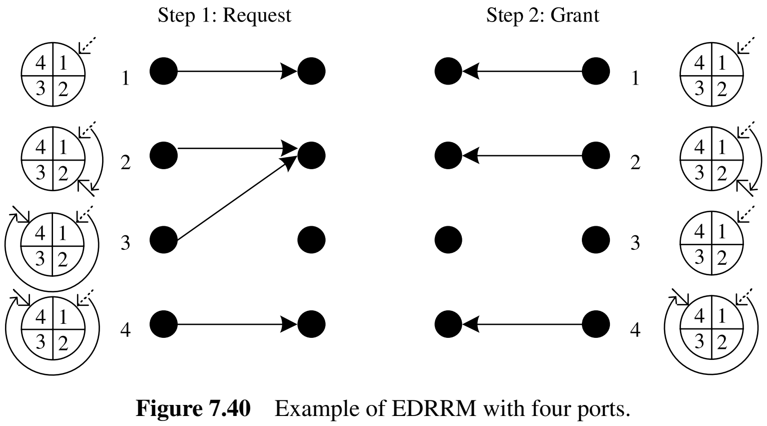


Figure 7.40 Example of EDRRM with four ports.

locked output to release it. The released input and output increase their arbiter pointers by one location, so that the VOQ just being served will have the lowest priority in the next arbitration. A locked output cannot grant any request from other inputs. An input sends requests to outputs if and only if it is not matched to any output. When a match is found, only this new connection will be updated which leads to a reconfiguration time, and all other connections stay uninterrupted. In order to reduce the frequency of switch fabric reconfigurations, in frame-based matching EDRRM is modified as follows. When a VOQ is completely served and the corresponding input or output has not successfully found a new match, the connection for this VOQ will not be disconnected. In this way, new arrivals to this VOQ can still be transferred before the switch fabric is updated.

Figure 7.40 shows an example of EDRRM where the port number is four. At the beginning, all the pointers are pointing at port 1. In step 1, each input port sends a request. In step 2, each output port grants the requests. Finally, there are three matchings in total. Input port 1 locks its pointer to output port 1 because its request is granted by output port 1. At the same time, output port 1 locks its pointer to input port 1. In the following time slots, output port 1 will always grant input port 1 if there is a request from input port 1 to output port 1. Input port 2 and 4 update their pointers similar to input port 1. The request of input port 3 is not granted by output port 2, so input port 3 updates its pointer to 4, and thus makes the VOQ to output port 3 become the lowest priority to send a request. Output port 3 does not update its pointer because it does not receive any requests.

To further reduce the bandwidth overhead, one possible variation of EDRRM is to start searching for the next matching when the number of cells waiting in the VOQ under service drops below a threshold. Additionally, since the arbitration time is not necessarily limited to one time slot, a higher complex matching scheme can be used to improve the performance and this will be a future topic of research.

When the reconfiguration time L is larger than zero in an optical switch, throughput is expected to degrade. Simulated throughput with different values of non-zero L and different switch size is shown in Table 7.2. It shows that throughput is relatively insensitive to L. The throughput can be improved by an appropriate speedup.

When the reconfiguration overhead $L > 0$, the variable frame schemes have better delay performance than fixed frame schemes under low and moderate loads. This is due to the fact that under low loads, fixed frames are often not filled, which leads to unnecessary additional delay. In variable-size frame-based matching schemes, the frame size adapts to the load of each VOQ, which means most frames are accumulated to be fully filled.

The delay performance of an optical EDRRM switch under uniform traffic can be analyzed by using an exhaustive random polling system model. The model can be applied to predict the performance of switches with, too large size to be simulated. The performance

TABLE 7.2 Throughput of an EDRRM Optical Switch

L (time slots)	0	10	100	1000
4×4 switch	0.9926	0.9913	0.9760	0.8946
8×8 switch	0.9408	0.9399	0.99383	0.9232
16×16 switch	0.9680	0.9673	0.9587	0.9279
32×32 switch	0.9794	0.9768	0.9636	0.9153

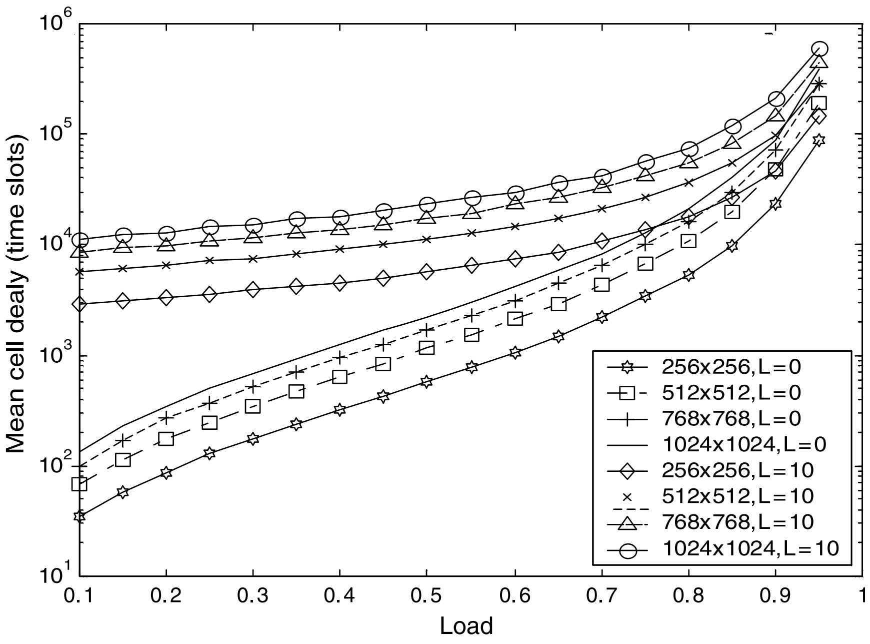


Figure 7.41 Average cell delay of the EDRRM with large switch size and varied L under uniform traffic.

analysis with L is a straightforward extension of the analysis without introducing L in [29]. When N goes to infinity, the average switch over time and the average delay converge to a limit for $\rho < 1$.

$$\lim_{N\to\infty} E(S) = \frac{1-e^{-\rho}}{1-e^{-(1-\rho)(1-e^{-\rho})}} - 1 + L,$$

and

$$E(T) \to E(S)\frac{N-\rho}{1-\rho} + \frac{2-\rho}{2(1-\rho)} \approx E(S)\frac{N}{1-\rho}.$$

Figure 7.41 shows the calculated average delay $E(T)$ of four switches of large size when the reconfiguration time L is 0 and 10. Compared with switches with zero configuration overhead, the delay is approximately increased by $LN/(1-\rho)$. For instance, when $N = 256$, $L = 10$, and $\rho = 0.1$, the additional delay caused by non-zero L is about 2844 time slots. The delay can be lowered with a speedup to compensate the configuration overhead.

7.6 STABLE MATCHING WITH SPEEDUP

The stable matching problem is a bipartite matching, proposed by Gale and Shapley [38]. An existing algorithm to solve the problem is GSA (Gale-Shapley algorithm) with a lower complexity bound of $O(N^2)$ [39]. In the input-buffered matching algorithms, GSA tries to find stable input–output port matching by using predefined input port priority lists and

TABLE 7.3 Comparison of Stable Marriage Algorithms

Algorithm	Input Preference List	Output Preference List	Speedup	Complexity
MUCFA	Urgent value	Urgent value	4	$\omega(N^2)$
CCF	Output occupancy	Urgent value	2	$O(N)$
LOOFA	Output occupancy	Arrival time	2	$O(N)$

output port priority lists, which are mainly used to solve input and output port contention. A match is considered to be stable, when all the matched input and output ports cannot find a better matching with a higher priority from those unmatched input and output ports.

In this section, we discuss three stable matching algorithms: most-urgent-cell-first algorithm (MUCFA), critical cell first (CCF), last-in-highest-priority (LIHP), and lowest-output-occupancy-cell-first (LOOFA). Table 7.3 summarizes the operations and complexity.

7.6.1 Output-Queuing Emulation with Speedup of 4

The most urgent cell first algorithm (MUCFA) scheme [40] schedules cells according to the "urgency". A shadow switch with output queuing is considered in the definition of the "urgency" of a cell. The urgency is also called the time to leave (TL), which indicates the time from the present that the cell will depart from the OQ switch. This value is calculated when a cell arrives. Since the buffers of the output-queued switch are FIFO, the urgency of a cell at any time equals the number of cells ahead of it in the output buffer at that time. It gets decremented after every time slot. Each output has the record of the urgency value of every cell destined for it. The algorithm is run as follows:

1. At the beginning of each phase, each output sends a request for the most urgent cell (i.e., the one with the smallest TL) to the corresponding input.
2. If an input receives more than one request, then it will grant to that output whose cell has the smallest urgency number. If there is a tie between two or more outputs, a supporting scheme is used. For example, the output with the smallest port number wins, or the winner is selected in a round-robin fashion.
3. Outputs that lose contention will send a request for their next most urgent cell.
4. The above steps run iteratively until no more matching is possible. Then cells are transferred and MUCFA goes to the next phase.

An example is shown in Figure 7.42. Each number represents a queued cell, and the number itself indicates the urgency of the cell. Each input maintains three VOQs, one for each output. Part (*a*) shows the initial state of the first matching phase. Output 1 sends a request to input 1 since the HOL cell in $VOQ_{1,1}$ is the most urgent for it. Output 2 sends a request to input 1 since the HOL cell in $VOQ_{1,2}$ is the most urgent for it. Output 3 sends a request to input 3 since the HOL cell in $VOQ_{3,3}$ is the most urgent for it. Part (*b*) illustrates matching results of the first phase, where cells from $VOQ_{1,1}$, $VOQ_{2,2}$, and $VOQ_{3,3}$ are transferred. Part (*c*) shows the initial state of the second phase, while part (*d*) gives the matching results of the second phase, in which HOL cells from $VOQ_{1,2}$ and $VOQ_{3,3}$ are matched.

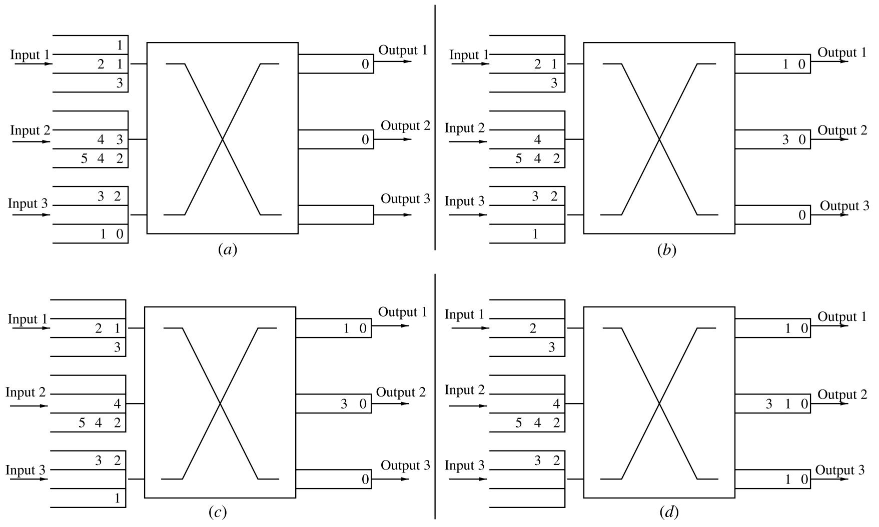


Figure 7.42 An example of two phases of MUCFA.

It has been shown that, under an internal speedup of 4, a switch with virtual-output queuing and MUCFA scheduling can behave identically to an output-queued switch, regardless of the nature of the arrival traffic.

7.6.2 Output-Queuing Emulation with Speedup of 2

This category of algorithms is based on an implementation of priority lists for each arbiter to select a matching pair [41]. The input priority list is formed by positioning each arriving cell at a particular place in the input queue. The relative ordering among other queued cells remains unchanged. This kind of queue is called a push-in queue. Some metrics are used for each arriving cell to determine the location. Furthermore, if cells are removed from the queue in an arbitrary order, we call it a push-in arbitrary out (PIAO) queue. If the cell at the head of queue is always removed next, we call it a push-in first out (PIFO) queue.

The algorithms described in this section also assume a shadow output-queued switch, based on which the following terms are defined:

1. Time to leave – $TL(c)$ is the time slot in which cell c would leave the shadow OQ switch. Of course, $TL(c)$ is also the time slot in which cell c must leave from the real switch for the identical behavior to be achieved.
2. Output cushion – $OC(c)$ is the number of cells waiting in the output buffer at cell c's output port that have a lower TL value than cell c. If a cell has a small (or zero) output cushion, then it is urgent to be delivered to its output so that it can depart when its TL is reached. Conversely, if a cell has a large output cushion, it may be temporarily set aside while more urgent cells are delivered to their outputs. Since the switch is work-conserving, a cell's output cushion is decremented after every time slot. A cell's output cushion increases only when a newly arriving cell is destined for the same output and has a more urgent TL.

3. Input thread – $IT(c)$ is the number of cells ahead of cell c in its input priority list. $IT(c)$ represents the number of cells currently at the input that have to be transferred to their outputs more urgently than cell c. A cell's input thread is decremented only when a cell ahead of it is transferred from the input, and is possibly incremented when a new cell arrives. It would be undesirable for a cell to simultaneously have a large input thread and a small output cushion – the cells ahead of it at the input may prevent it from reaching its output before its TL. This motivates the definition of slackness.
4. Slackness – $L(c)$ equals the output cushion of cell c minus its input thread, that is, $L(c) = OC(c) - IT(c)$. Slackness is a measure of how large a cell's output cushion is with respect to its input thread. If a cell's slackness is small, then it is urgent to be transferred to its output. Conversely, if a cell has a large slackness, then it may be kept at the input for a while.

Critical Cell First (CCF). CCF is a scheme for inserting cells in input queues that are PIFO queues. An arriving cell is inserted as far from the head of its input queue as possible such that the input thread of the cell is not larger than its output cushion (i.e., a positive slackness). Suppose that cell c arrives at input port P. Let x be the number of cells waiting in the output buffer at cell c's output port. Those cells have a lower TL value than cell c or the output cushion $OC(c)$ of c. Insert cell c into $(x + 1)$th position from the front of the input queue at P. As shown in Figure 7.43, each cell is represented by its destined output port and the time to leave. For example, cell (B,4) is destined for output B and has a TL value equal to 4. Part (*a*) shows the initial state of the input queues. Part (*b*) shows the insertion of two incoming cells (C,4) and (B,4) to ports Y and Z, respectively. Cell (C,4) is inserted at the third place of port Y and cell (B,4) at the second place of port Z. Hence, upon arrival, both cells have zero slackness. If the size of the priority list is less than x cells, then place c at the end of the input priority list. In this case, cell c has a positive slackness. Therefore, every cell has a non-negative slackness on arrival.

Last in Highest Priority (LIHP). LIHP is also a scheme for inserting cells at input queues. It was proposed mainly to show and demonstrate the sufficient speedup to make an input–output queued switch emulate an output-queued switch. LIHP places a newly arriving

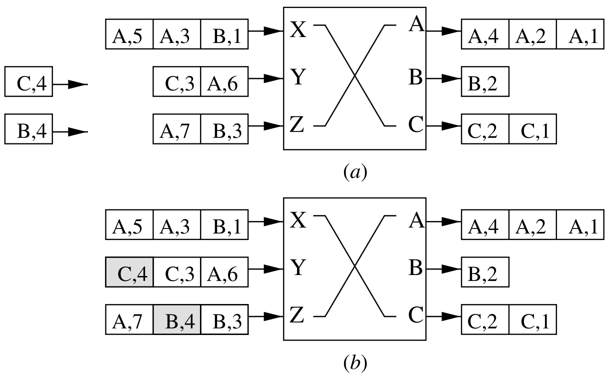


Figure 7.43 Example of CCF priority placement.

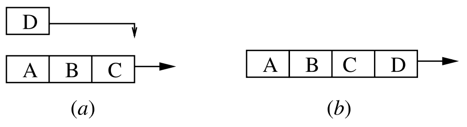


Figure 7.44 Example of placement for LIHP. (*a*) Initial state; (*b*) New incoming cell is placed at the highest priority position..

cell right at the front of the input priority list, providing a zero input thread ($IT(c) = 0$) for the arriving cell. See Figure 7.44 for an example. The scheduling in every arbitration phase is a stable matching based on the time to leave value and the position in its input priority list of each queued cell.

The necessary and sufficient speedup is $2 - 1/N$ for a $N \times N$ input-and-output queued switch to exactly emulate a $N \times N$ output-queued switch with FIFO service discipline.

The necessary condition can be shown by the example as shown below. Since the speedup $2 - (1/N)$ represents a non-integral distribution of arbitration phases per slot between one and two, we first describe how scheduling phases are distributed. A speedup of $2 - (1/N)$ corresponds to having a truncated time slot out of every N time slots; the truncated time slot has just one scheduling phase, whereas the other $N - 1$ time slots have two scheduling phases each. Figure 7.45 shows the difference between one-phased and two-phased time slots. We assume that the scheduling algorithm does not know in advance whether a time slot is truncated.

Recall that a cell is represented as a tuple (P, TL), where P represents which output port the cell is destined to and TL represents the time to leave for the cell. For example, the cell $(C,7)$ must be scheduled for port C before the end of time slot 7.

The input traffic pattern that provides the lower bound for an $N \times N$ input–output queued switch is given as follows. The traffic pattern N spans time slots, the last of which is truncated:

1. In the first time slot, all input ports receive cells destined for the same output port, P_1.
2. In the second time slot, the input port that had the lowest time to leave in the previous time slot does not receive any more cells. In addition, the rest of the input ports receive cells destined for the same output port, P_2.

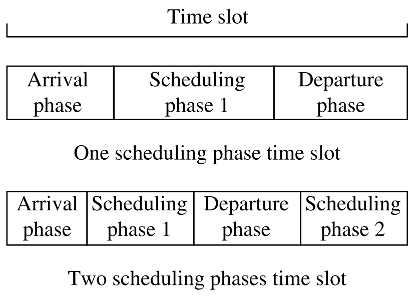


Figure 7.45 One scheduling phase and two scheduling-phase time slots.

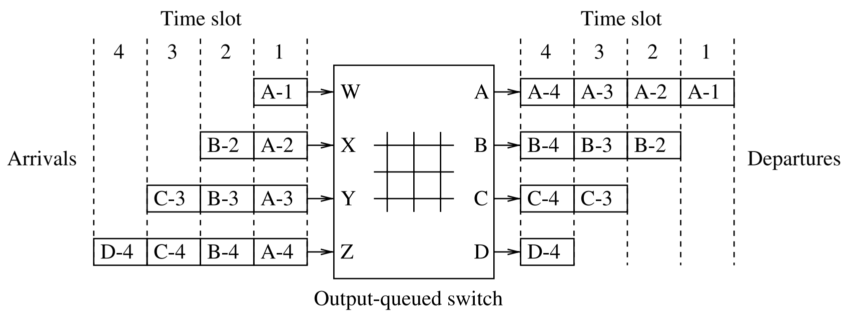


Figure 7.46 Lower bound input traffic pattern for a 4 × 4 switch.

3. In the ith time slot, the input ports that had the lowest time to leave in each of the $i - 1$ previous time slots do not receive any more cells. In addition, the rest of the input ports must receive cells destined for the same output port, P_i.

One can repeat the traffic pattern just mentioned as many times as is required to create arbitrarily long traffic patterns. Figure 7.46 shows the above sequence of cells for a 4 × 4 switch. The departure events from the output-queued switch are depicted on the right, and the arrival events are on the left. For simplicity, we present the proof of our lower bound on this 4 × 4 switch instead of a general $N \times N$ switch.

Figure 7.47 shows the only possible schedule for transferring these cells across in seven phases. Of the four time slots, the last one is truncated, giving a total of seven phases. Cell A-1 must leave the input side during the first phase, since the input–output queued switch does not know whether the first time slot is truncated. Similarly, cells B-2, C-3, and D-4 must leave during the third, fifth, and seventh phases, respectively (see Fig. 7.47*a*). Cell A-2 must leave the input side by the end of the third phase. However, it cannot leave during the first or the third phase because of contention. Therefore, it must depart during the second phase. Similarly, cells B-3 and C-4 must depart during the fourth and sixth phases, respectively (see Fig. 7.47*b*). Continuing this elimination process (see Fig. 7.47*c* and *d*), there is only one possible scheduling order. For this input traffic pattern, the switch needs all seven phases in four time slots, which corresponds to a minimum speedup of $7/4$ (or $2 - 1/4$). The proof of the general case for a $N \times N$ switch is a straightforward extension of the 4 × 4 example.

7.6.3 Lowest Output Occupancy Cell First (LOOFA)

The LOOFA is a work-conserving scheduling algorithm [42]. It provides 100 percent throughput and a cell delay bound for feasible traffic, using a speedup of 2. An input-and-output queued architecture is considered. Two versions of this scheme were presented: the greedy and the best-first. This scheme considers three different parameters associated with a cell, say cell c, to perform a match: the number of cells in its destined output queue or output occupancy $OCC(c)$, the time stamp of a cell or cell age $TS(c)$, and the smallest port number to break ties. Under the speedup of 2, each time slot has two phases. During each phase, the greedy version of this algorithm works as follows (see Figure 7.48 for

Phase	PA	PB	PC	PD
1	**A-1**			
2				
3		**B-2**		
4				
5			**C-3**	
6				
7				**D-4**

(*a*)

Phase	PA	PB	PC	PD
1	A-1			
2	**A-2**			
3		B-2		
4		**B-3**		
5			C-3	
6			**C-4**	
7				D-4

(*b*)

Phase	PA	PB	PC	PD
1	A-1			
2	A-2			
3	**A-3**	B-2		
4		B-3		
5		**B-4**	**C-3**	
6			C-4	
7				D-4

(*c*)

Phase	PA	PB	PC	PD
1	A-1			
2	A-2			
3	A-3	B-2		
4	**A-4**	B-3		
5		B-4	C-3	
6			C-4	
7				D-4

(*d*)

Figure 7.47 Scheduling order for the lower bound input traffic pattern in Figure 7.46.

an example):

1. Initially, all inputs and outputs are unmatched.
2. Each unmatched input selects an active VOQ (i.e., a VOQ that has at least one cell queued) going to the unmatched output with the lowest occupancy, and sends a request to that output. Ties are broken by selecting the smallest output port number. See part (*a*) in Figure 7.48.
3. Each output, on receiving requests from multiple inputs, selecting the one with the smallest *OCC* and sends the grant to that input. Ties are broken by selecting the smallest port number.
4. Return to step 2 until no more connections can be made.

An example of the greedy version is shown in Figure 7.48. The tuple "x, y" in the VOQ represents the output occupancy $OCC(c)$ and the timestamp $TS(c)$ of cell c, respectively. In the upper part of the figure, the arrows indicate the destination for all different cells at the input ports. The gray arrows in the lower part of the figure indicate the exchange of requests and grants. The black arrows indicate the final match. Part (*a*) shows that each input sends a request to the output with the lowest occupancy. Output 2 receives two requests, one from A and the other from B, while output 3 receives a request from input C. Part (*b*) illustrates that, between the two requests, output 2 chooses input A, the one with lower *TS*. Output 3 chooses the only request, input C.

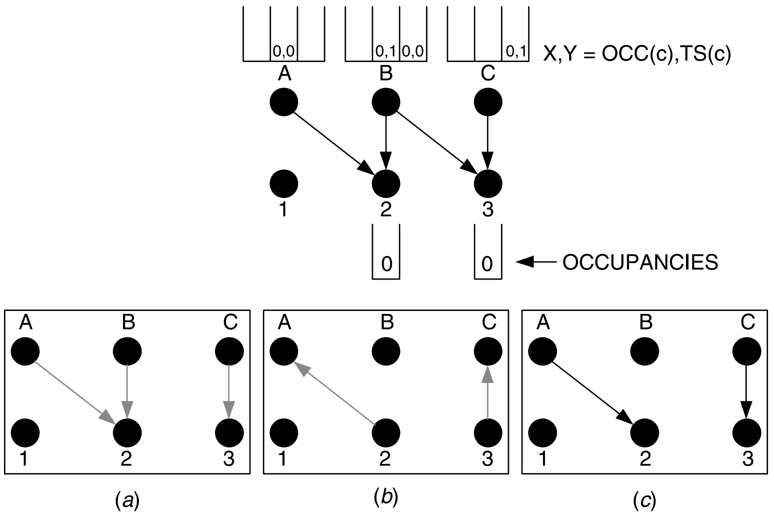


Figure 7.48 Matching example with the greedy LOOFA.

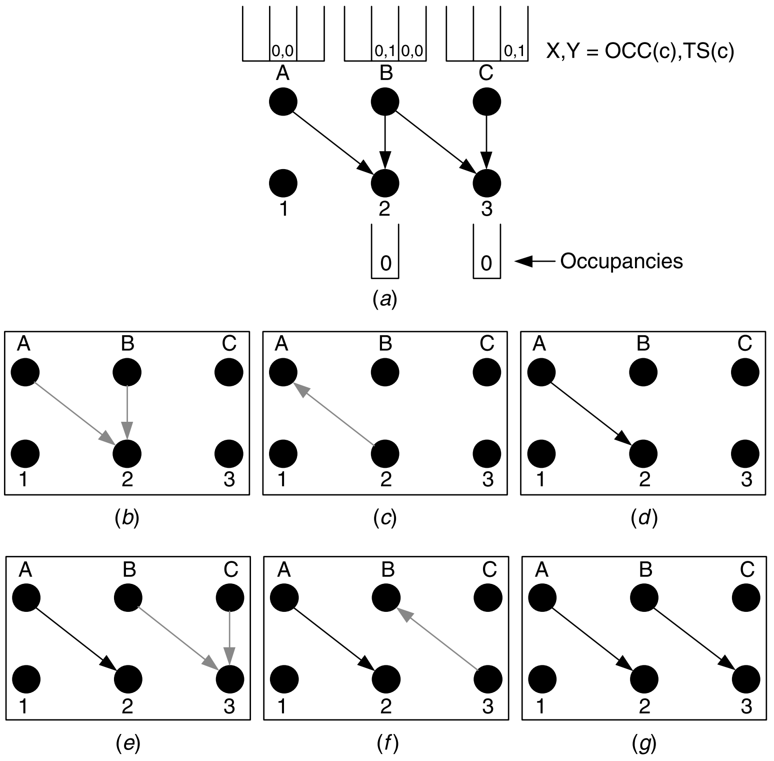


Figure 7.49 Matching example with the best-first version of LOOFA.

The best-first version works as follows:

1. Initially, all inputs and outputs are unmatched.
2. Among all unmatched outputs, the output with the lowest occupancy is selected. Ties are broken by selecting the smallest output port number. All inputs that have a cell destined for the selected output send a request to it.
3. The output selects the cell request input with the smallest time stamp and sends the grant to the input. Ties are broken by selecting the smaller input port number.
4. Return to step 2 until no more connections can be made (or N iterations are completed).

Figure 7.49 shows a matching example with the best-first version example. The selection of the output with the lowest $OCC(c)$ results in a tie. Outputs 2 and 3 have the lowest OCC. This tie is broken by selecting output 2 since this port number is the smaller number. Therefore, inputs A and B send a request to this output as shown in part (b), while part (c) illustrates that output 2 grants the oldest cell, input A. Part (d) shows the matching result after the first iteration. The second iteration begins in part (e) when output 3 is chosen as the unmatched output port with the lowest OCC with requests from inputs B and C. Input B is chosen in part (f) for its lowest $TS(c)$. Part (g) depicts the final match.

Both algorithms achieve a maximal matching, with the greedy version achieving it in less iterations. On the other hand, it has been proven that, when combined with the oldest-cell-first input selection scheme, the best-first version provides delay bounds for rate-controlled input traffic under a speedup of 2. Denote D_a and D_o as the arbitration delay and the output queuing delay of any cell. It can be shown that $D_a \leq 4N/(S-1)$ and $D_o \leq 2N$ cell slots, where S is the speedup factor.

REFERENCES

[1] N. McKeown, A. Mekkittikul, V. Anantharam, and J. Walrand, "Achieving 100% throughput in an input-queued switch," *IEEE Transactions on Communications*, vol. 47, no. 8, pp. 1260–1267 (Aug. 1999).

[2] L. Tassiulas and A. Ephremides, "Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks," *IEEE Transactions on Automatic control*, vol. 37, no. 12, pp. 1936–1949 (Dec. 1992).

[3] R. E. Tarjan, "Data structures and network algorithms," in *Proc. Society for Industrial and Applied Mathematics*, Pennsylvania (Nov. 1983).

[4] A. Mekkittikul and N. Mckeown, "A practical scheduling algorithm to achieve 100% throughput in input-queued switches," *INFOCOM'98*, San Francisco, California, pp. 792–799 (Mar. 1998).

[5] D. Shah and M. Kopikare, "Delay bounds for approximate maximum weight matching algorithms for input queued switches," in *Proc. IEEE INFOCOM'02*, New York, pp. 1024–1031 (June 2002).

[6] J. E. Hopcroft and R. M. Karp, "An $n^{\frac{5}{2}}$ algorithm for maximum matchings in bipartite graphs," SIAM, J. Comput., vol. 2, no. 4, pp. 225–231 (Dec. 1973).

[7] N. McKeown, "Scheduling algorithms for input-queued cell switches," Ph.D. thesis, UC Berkeley, May 1995.

[8] J. G. Dai and B. Prabhakar, "The throughput of data switches with and without speedup," in *Proc. IEEE INFOCOM'00*, Tel Aviv, Israel, pp. 556–564 (Mar. 2000).

[9] E. Leonardi, M. Mellia, F. Neri, and M. A. Marsan, "On the stability of input-queued switches with speed-up," *IEEE/ACM Transactions on Networking*, vol. 9, no. 1, pp. 104–118 (Feb. 2001).

[10] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker, "High speed switch scheduling for local area networks," *ACM Transactions on Computer Systems*, vol. 11, no. 4, pp. 319–352 (Nov. 1993).

[11] N. McKeown, "The iSLIP scheduling algorithm for input-queued switches," *IEEE/ACM Transactions on Networking*, vol. 7, no. 2, pp. 188–201 (Apr. 1999).

[12] N. McKeown, P. Varaiya, and J. Warland, "Scheduling cells in an input-queued switch," *IEE Electronics Letters*, vol. 29, issue 25, pp. 2174–2175 (Dec. 1993).

[13] Y. Li, S. Panwar, and H. J. Chao, "On the performance of a dual round-robin switch," in *Proc. IEEE INFOCOM 2001*, Anchorage, Alaska, vol. 3, pp. 1688–1697 (Apr. 2001).

[14] Y. Li, "Design and analysis of schedulers for high speed input queued switches," Ph.D. Dissertation, Polytechnic University, Brooklyn, New York, Jan. 2004.

[15] D. N. Serpanos and P. I. Antoniadis, "FIRM: a class of distributed scheduling algorithms for high-speed ATM switches with multiple input queues," in *Proc. IEEE INFOCOM'00*, Tel Aviv, Israel, pp. 548–554 (Mar. 2000).

[16] H. J. Chao and J. S. Park, "Centralized contention resolution schemes for a large-capacity optical ATM switch," in *Proc. IEEE ATM Workshop*, Fairfax, Virginia (May 1998).

[17] H. J. Chao, "Satur: A terabit packet switch using dual round-robin," *IEEE Communications Magazine*, vol. 38, no. 12, pp. 78–84 (Dec. 2000).

[18] E. Oki, R. Rojas-Cessa, and H. J. Chao, "A pipeline-based maximal-sized matching scheme for high-speed input-buffered switches," *IEICE Transactions on Communications*, vol. E85-B, no. 7, pp. 1302–1311 (July 2002).

[19] A. Smiljanic, R. Fan, and G. Ramamurthy, "RRGS-round-robin greedy scheduling for electronic/optical terabit switches," in *Proc. IEEE GLOBECOM'99*, Rio de Janeireo, Brazil, pp. 1244–1250 (Dec. 1999).

[20] H. Takagi, "Queueing analysis of polling models: an update," in *Stochastic Analysis of Computer and Communication Systems*. Elsevier Science Inc., New York; pp. 267–318, 1990.

[21] Y. Li, S. Panwar, and H. J. Chao, "The dual round-robin matching switch with exhaustive service," in *Proc. High Performace Switching and Routing (HPSR) 2002*, Kobe, Japan (May 2002).

[22] P. Giaccone, B. Prabhakar, and D. Shah, "Towards simple, high-performance schedulers for high-aggregate bandwidth switches," in *Proc. IEEE INFOCOM'02*, New York, vol. 3, pp. 1160–1169 (2002).

[23] L. Tassiulas, "Linear complexity algorithms for maximum throughput in radio networks and input queued switches," in *Proc. IEEE INFOCOM'98*, San Francisco, California, vol. 2, pp. 533–539 (Mar. 1998).

[24] A. Nijenhuis and H. Wilf, *Combinatorial Algorithms: for Computers and Calculators*. Academic Press, Orlando, Florida, 1978.

[25] P. Giaccone, B. Prabhakar, and D. Shah, "Randomized scheduling algorithms for high-aggregate bandwidth switches," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 4, pp. 546–559 (May 2003).

[26] D. Shah, P. Giaccone, and B. Prabhakar, "Efficient randomized algorithms for input-queued switch scheduling," *IEEE Micro*, vol. 22, no. 1, pp. 10–18 (Jan. 2002).

[27] Y. Li, S. Panwar, and H. J. Chao, “Exhaustive service matching algorithms for input queued switches,” in *Proc. Workshop on High Performance Switching and Routing (HPSR 2004)*, Phoenix, Arizona (Apr. 2004).

[28] L. Kleinrock and H. Levy, “The analysis of random polling systems,” *Operations Research*, vol. 36, no. 5, pp. 716–732 (Sept. 1988).

[29] Y. Li, S. Panwar, and H. J. Chao, “Performance analysis of a dual round robin matching switch with exhaustive service,” in *Proc. IEEE GLOBECOM’02*, Taipei, Taiwan, vol. 3, pp. 2292–2297 (Nov. 2002).

[30] F. A. Marsan, A. Bianco, P. Giaccone, E. Leonardi, and E. Neri, “Packet scheduling in input-queued cell-based switches,” in *Proc. IEEE INFOCOM’01*, Anchorage, Alaska, vol. 2, pp. 1085–1094 (Apr. 2001).

[31] K. Claffy, G. Miller, and K. Thompson, “The nature of the beast: recent traffic measurements from an Internet backbone,” in *Proc. INET’98*, Geneva, Switzerland, pp. 21–24 (July 1998).

[32] B. Towles and W. Dally, “Guaranteed scheduling for switches with configuration overhead,” in *Proc. IEEE INFOCOM’02*, New York, vol. 1, pp. 342–351 (June 2002).

[33] T. Inukai, “An efficient SS/TDMA time slot assignment algorithm,” *IEEE Transactions on Communications*, vol. 27, no. 10, pp. 1449–1455 (Oct. 1979).

[34] C. S. Chang, W. J. Chen, and H. Y. Huang, “Birkhoff–von Neumann input buffered crossbar switches,” in *Proc. IEEE INFOCOM’00*, Tel Aviv, Israel, pp. 1614–1623 (Mar. 2000).

[35] B. Towles and W. J. Dally, “Guaranteed scheduling for switches with configuration overhead,” *IEEE/ACM Transactions on Networking*, vol. 11, no. 5, pp. 835–847 (Oct. 2003).

[36] R. Cole and J. Hopcroft, “On edge coloring bipartite graph,” *SIAM Journal on Computing*, vol. 11, no. 3, pp. 540–546 (1982).

[37] Y. Li, S. Panwar, and H. J. Chao, “Frame-based matching algorithms for optical switches,” in *Proc. IEEE Workshop on High Performance Switching and Routing (HPSR 2003)*, Torino, Italy, pp. 97–102 (June 2003).

[38] D. Gale and L. S. Shapley, “College admission and the stability of marriage,” *American Mathematical Monthly*, vol. 69, no. 1, pp. 9–15 (1962).

[39] D. Gusfield and R. Irving, *The Stable Marriage Problem: Structure and Algorithms*. The MIT Press, Cambridge, Massachusetts, 1989.

[40] B. Prabhakar and N. McKeown, “On the speedup required for combined input and output queued switching,” *Automatica*, vol. 35, issue 12, pp. 1909–1920 (Dec. 1999).

[41] S. T. Chuang, A. Goel, N. McKeown, and B. Prabhakar, “Matching output queuing with a combined input/output-queued switch,” *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1030–1039 (June 1999).

[42] P. Krishna, N. Patel, A. Charny, and R. Simcoe, “On the speedup required for work-conversing crossbar switches,” *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1057–1066 (June 1999).