Input buffered switches

Masoud Sabaei

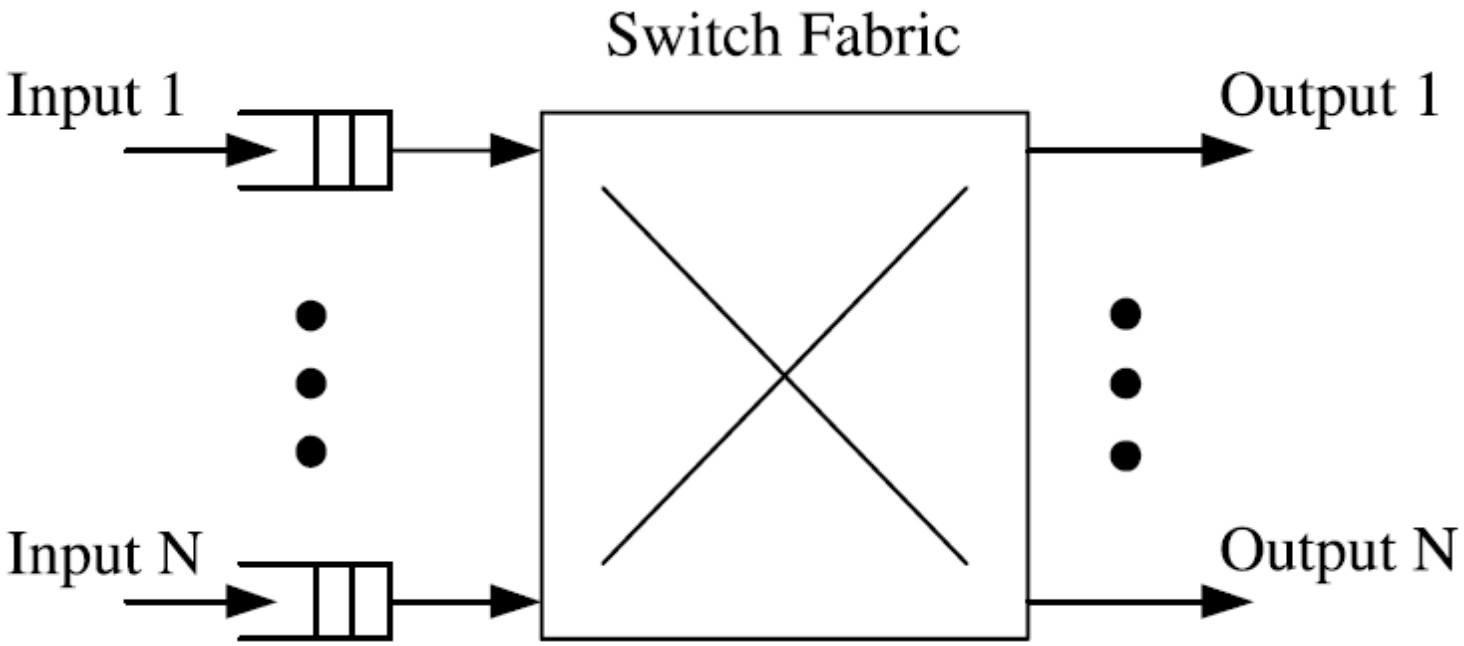
Associate professor

Department of Computer Engineering,
Amirkabir University of Technology

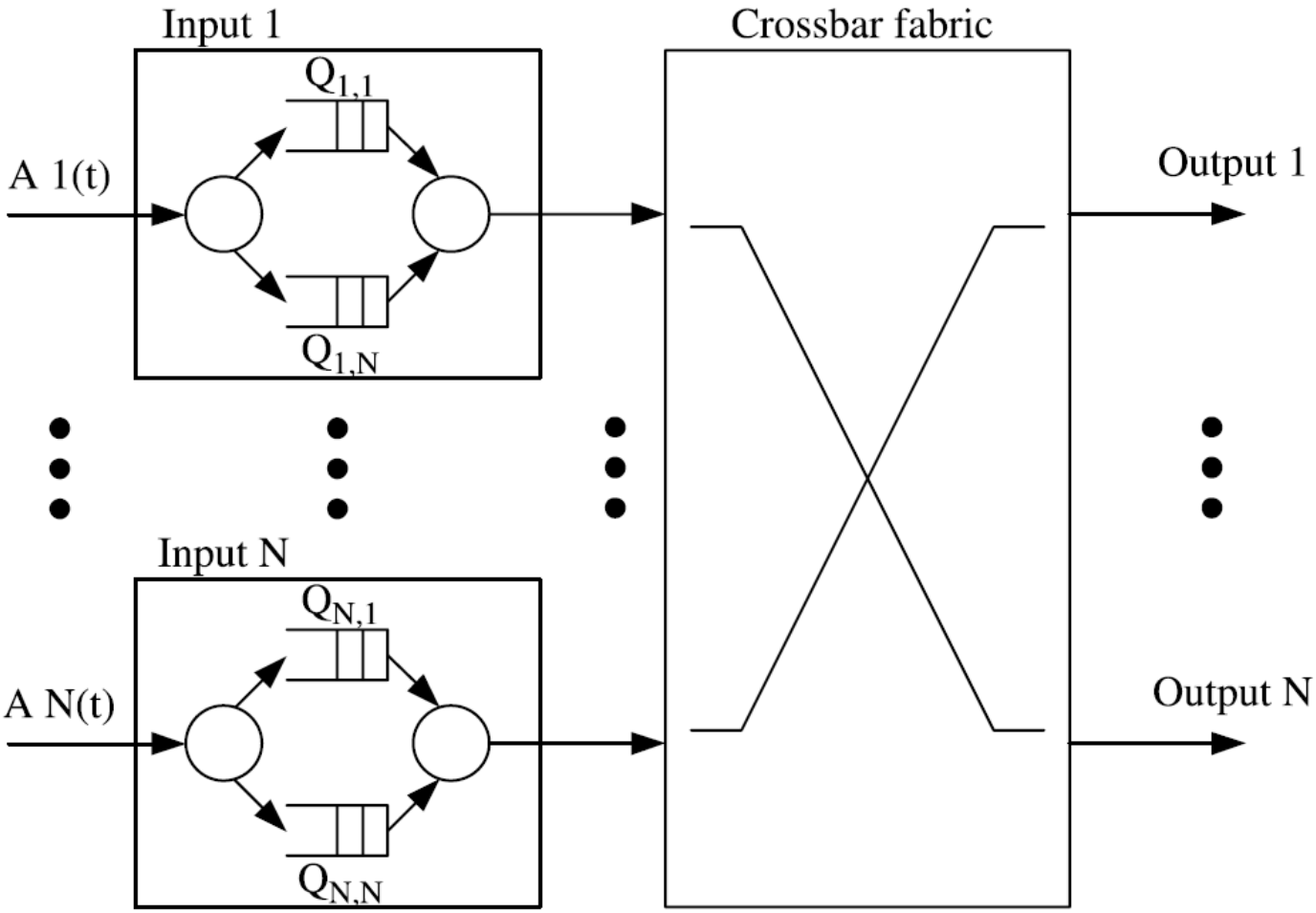
Introduction

- Input-buffered switches
  - Two major problems
    - Throughput limitation
      - Because of head-of-line (HOL) blocking
      - Need for faster switch fabric
      - Need for more paths to output ports
    - Arbitration
      - Because of out port contentions
      - Need for fast scheduling mechanisms (this chapter)

Input-buffered Switch Model With FIFO Queues



Input-buffered Structure With Virtual Output Queues



Scheduling in VOQ-based Switches

$A_i(t)$: The cell arrival process to input port i.

Q_{ij}: The queue that the arrived cell at input port i and destined to output j is put into it.

$L_{ij}(t)$: The queue length of Q_{ij}.

$A_{ij}(t)$: the arrival process from input i to output j with an arrival rate of λ_{ij}, and $A(t) = \{A_{ij}(t), 1 \le i \le N \text{ and } 1 \le j \le N\}$.

If the arrivals to each input and each output are 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.

Scheduling in VOQ-based Switches

Scheduling Algorithms:

The arbitration process of a switch decides which cells at input buffers will be transferred to outputs.

Scheduling in VOQ-based Switches

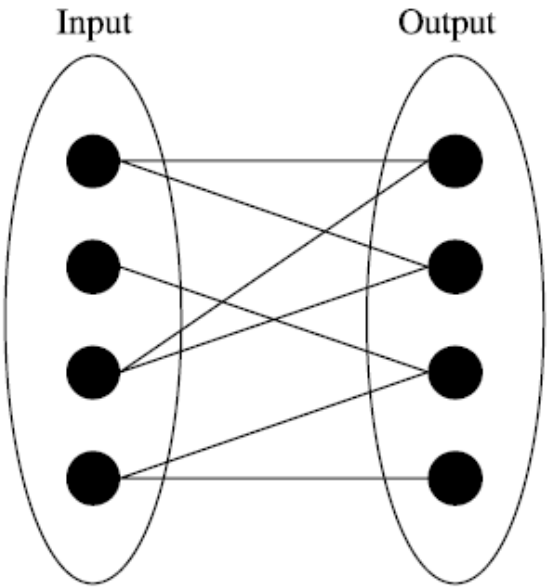
$S(t) = [s_{ij}(t)]_{N \times N}$: Service matrix

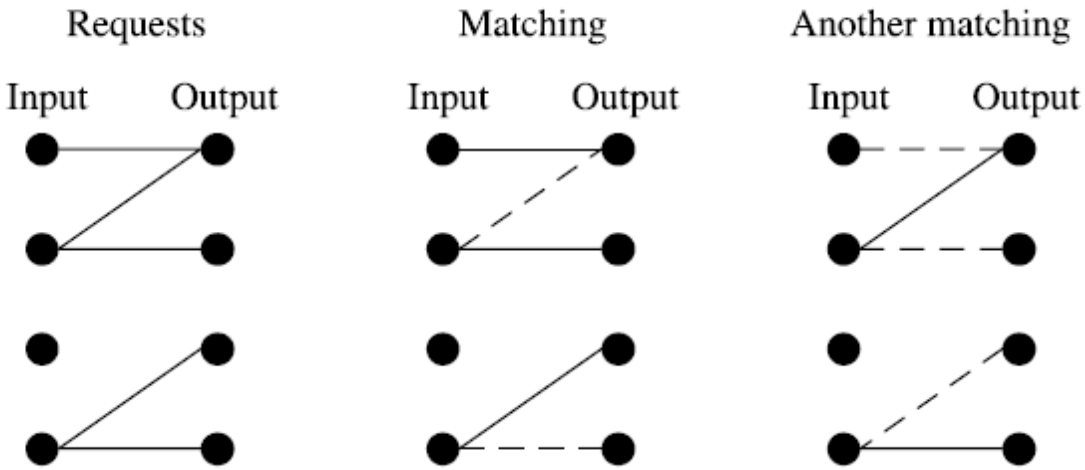
$$s_{ij}(t) = \begin{cases} 1, & \text{if a cell is transfered from } i \text{ to } j \\ 0, & \text{otherwise} \end{cases}$$

A switch is defined to be stable,
if the expected queue length is bounded, that is,

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

Bipartite Graph Matching Example





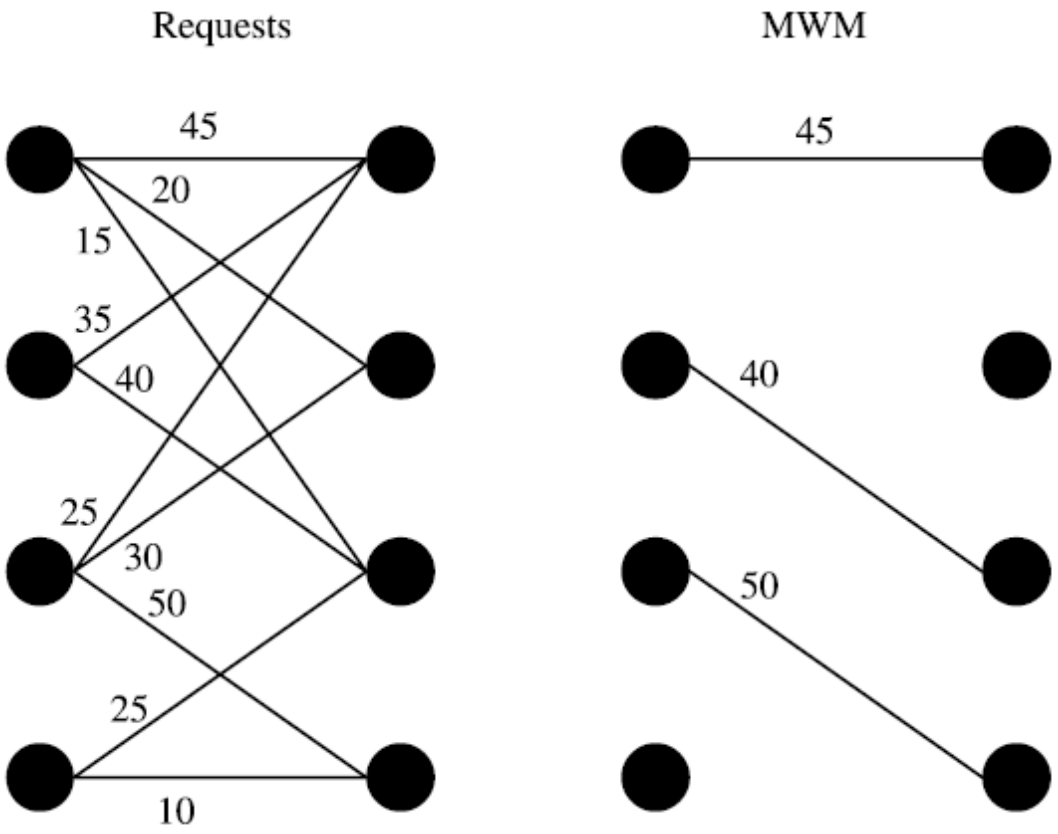
Factors to be considered in a scheduling algorithm design

- ***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.

Scheduling Algorithm

- Maximum Matching
- Maximal Matching
- Randomized Matching Algorithms
- Frame-based Matching
- Stable Matching With Speedup

Maximum Weight Matching



Maximum Weight Matching (MWM)

A maximum weight matching algorithm achieves 100 percent throughput under any admissible traffic.

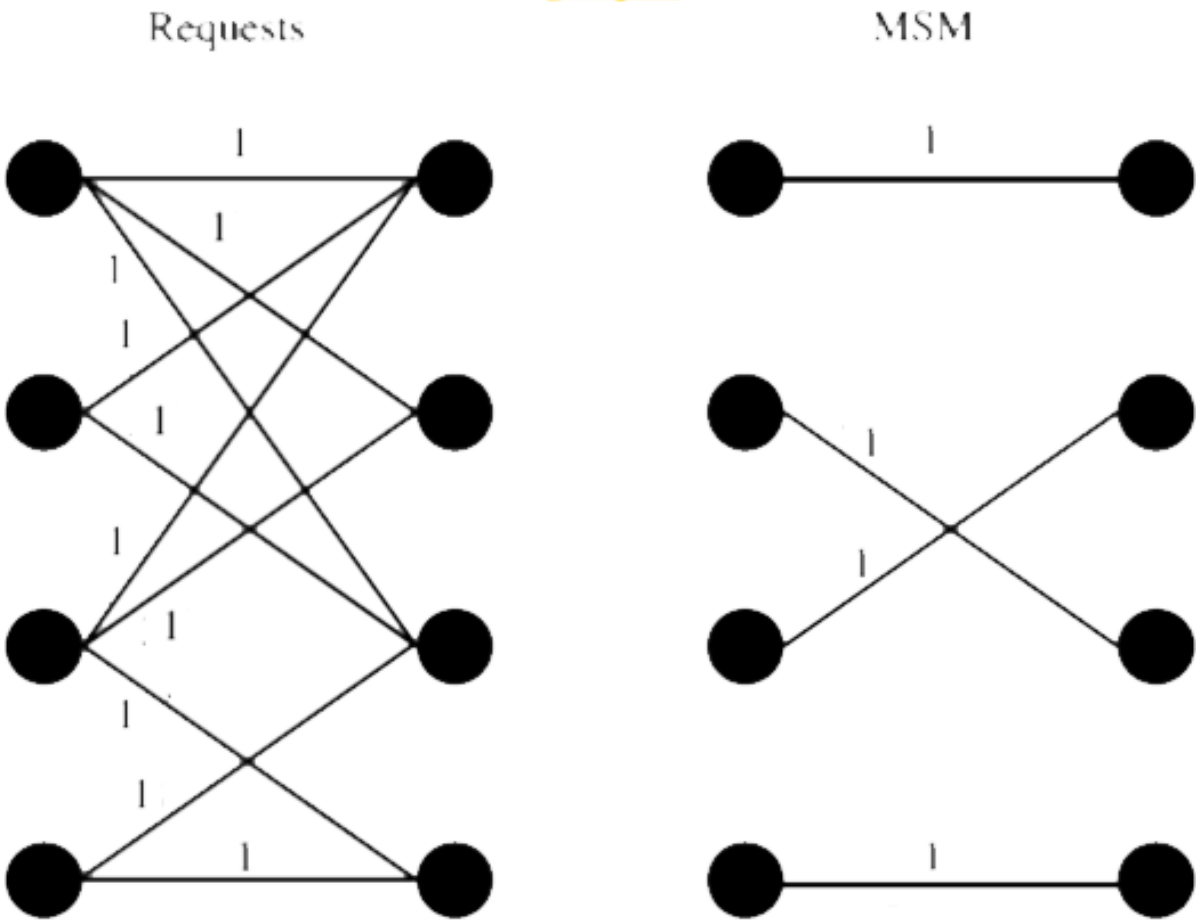
MWM can be solved in time $O(N^3)$

- LQF (longest queue first)
- OCF (oldest cell first)
- LPF (longest port first)

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

where $R_i(t) = \sum_{j=1}^{N} L_{ij}(t)$, $C_j(t) = \sum_{i=1}^{N} L_{ij}(t)$.

Maximum Size Matching



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 $O(N^{2.5})$

Maximal Matching

- **Parallel Iterative Matching (PIM)**
- **Iterative Round-Robin Matching (*i*RRM)**
- **Iterative Round-Robin with SLIP (*i*SLIP)**
- **FIRM (FCFS in round-robin matching)**
- **Dual Round-Robin Matching (DRRM)**
- **Pipelined Maximal Matching**
- **Exhaustive Dual Round-Robin Matching (EDRRM)**

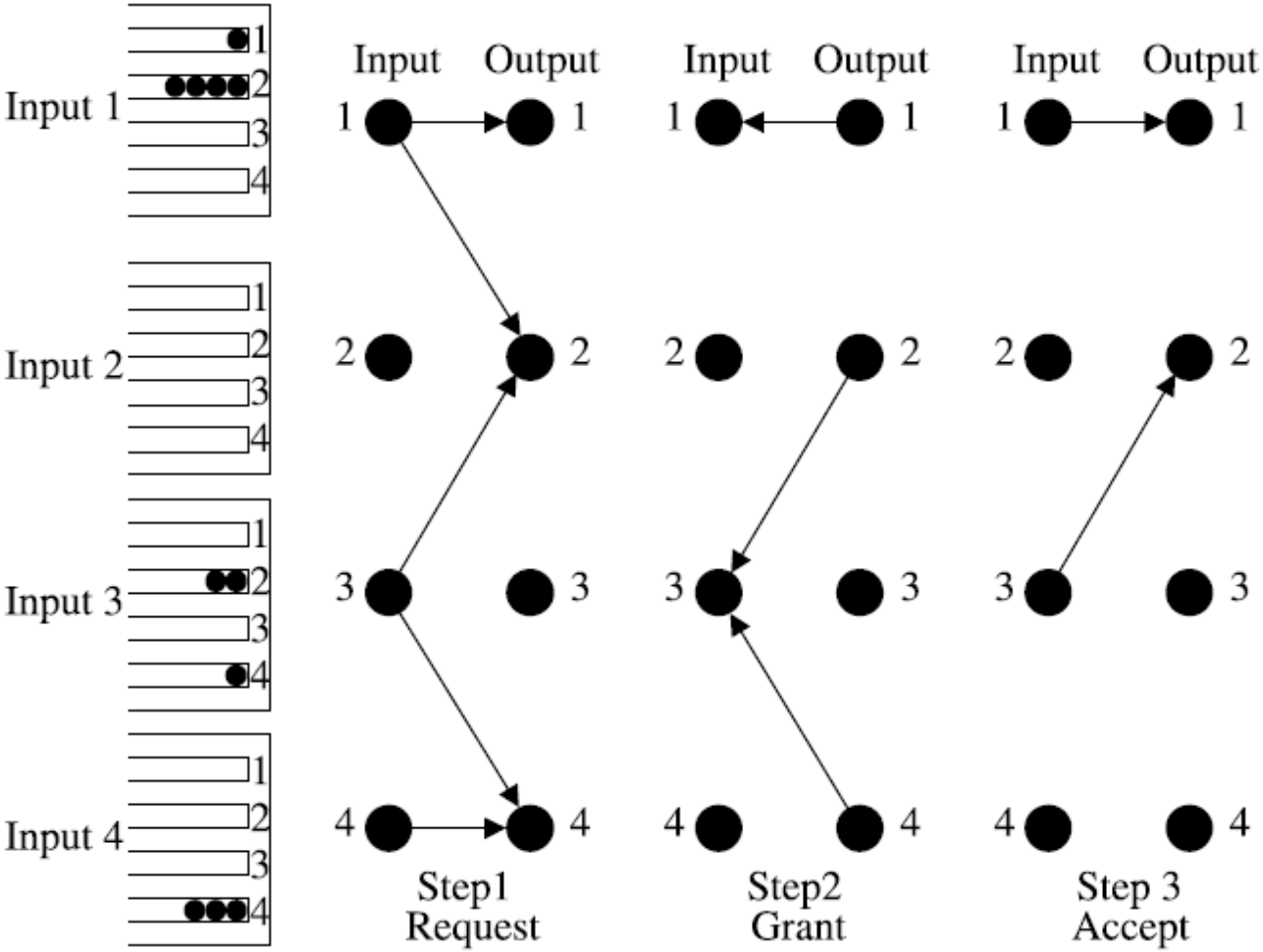
Parallel Iterative Matching (PIM)

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.

Parallel Iterative Matching (PIM)



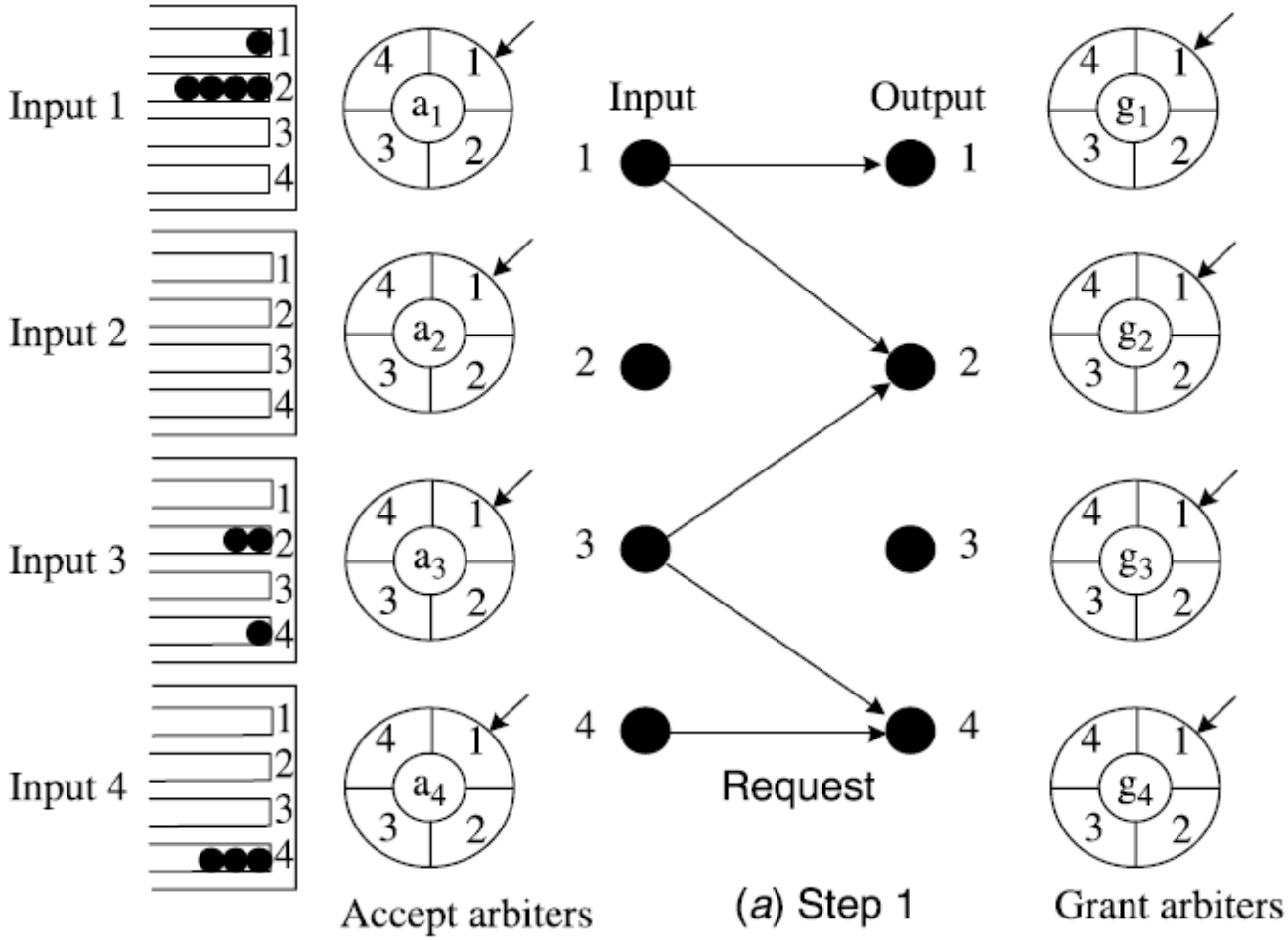
Iterative Round Robin Matching (iRRM)

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

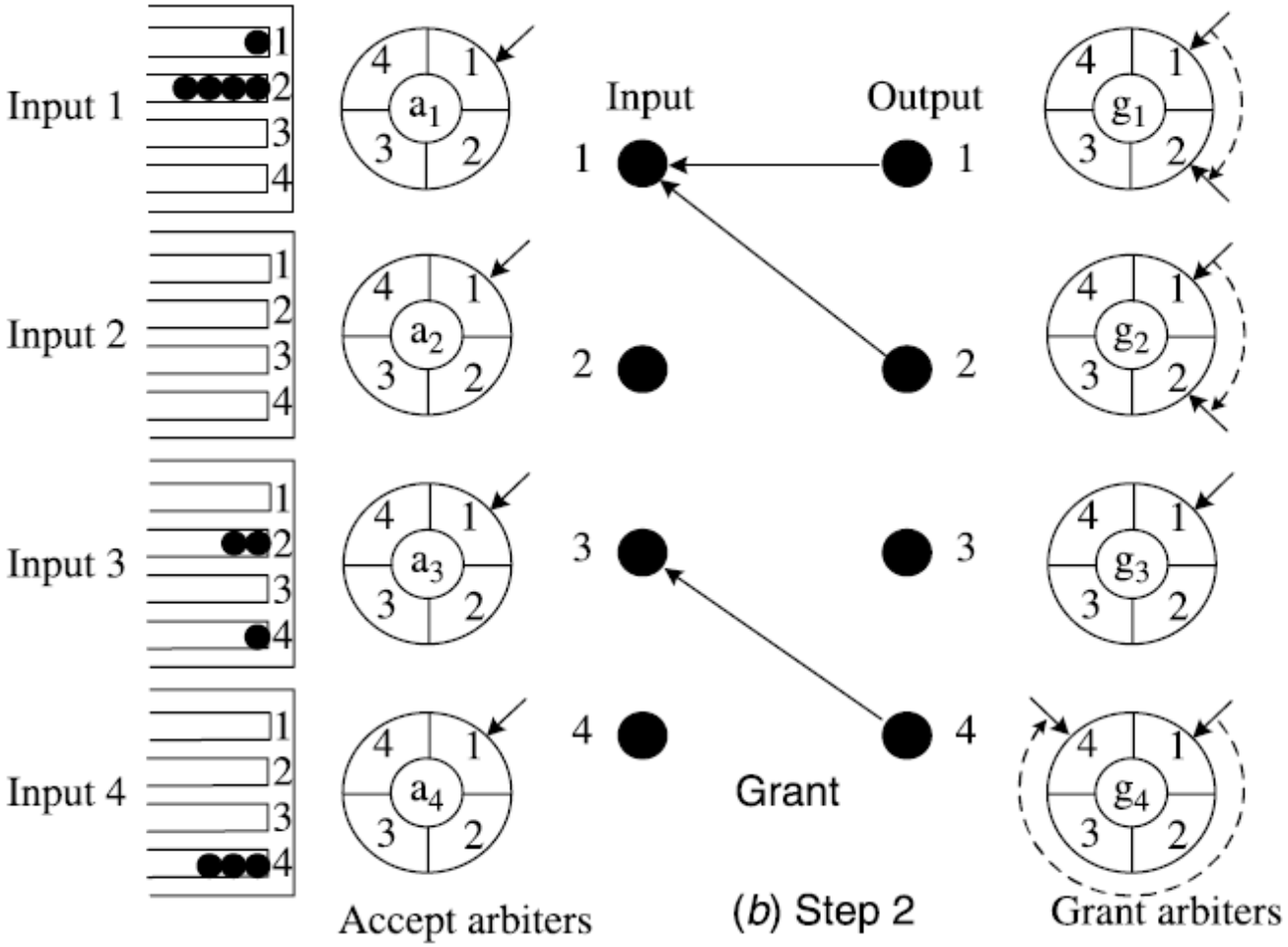
Step 2: *Grant.* If an unmatched output receives any requests, it chooses the one that appears next in a round-robin schedule starting from the highest priority element. The output notifies each input whether or not its request was granted. The pointer g_i is incremented (module N) to one location beyond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

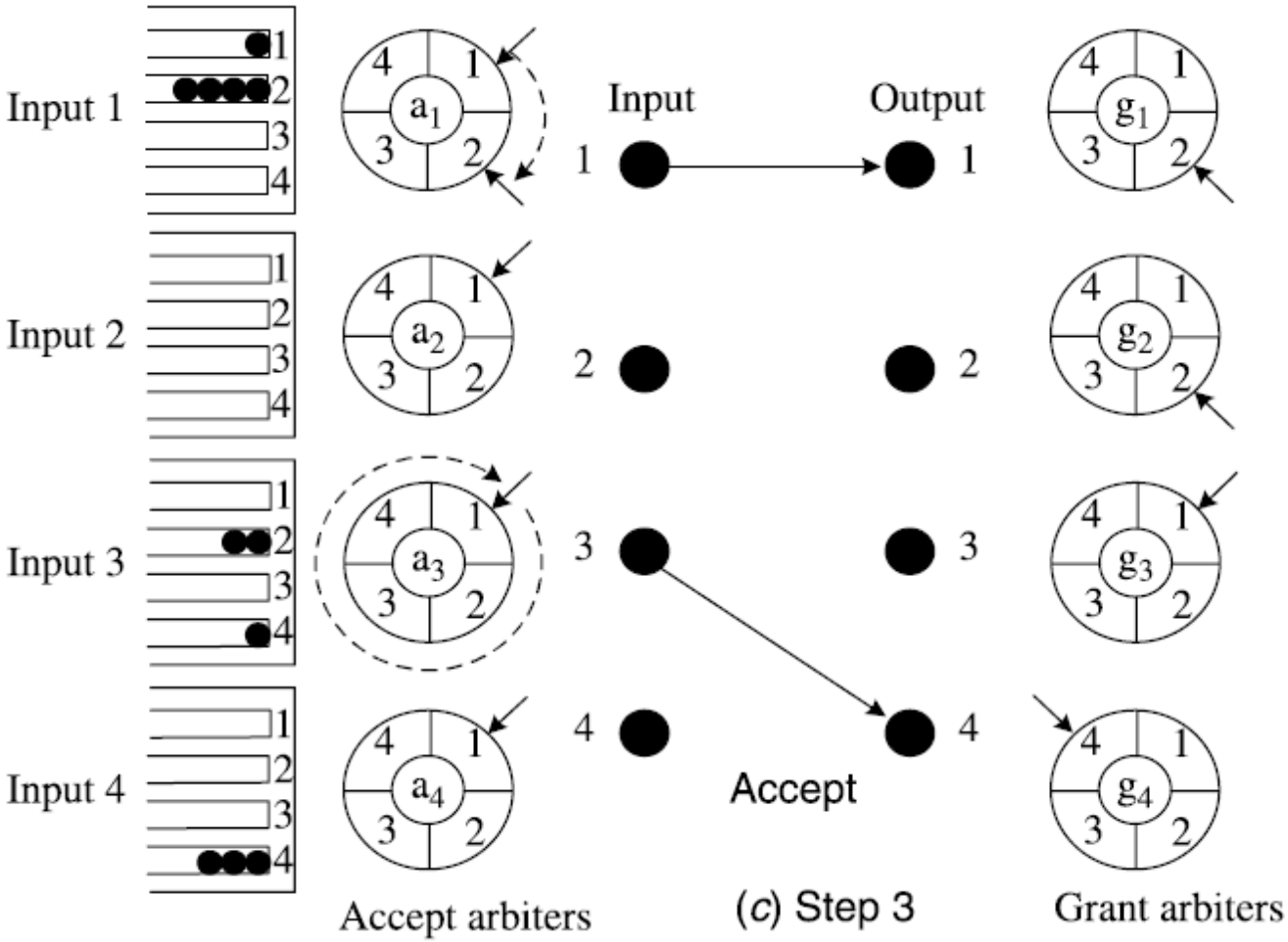
Iterative Round Robin Matching (iRRM)



Iterative Round Robin Matching (iRRM)



Iterative Round Robin Matching (iRRM)



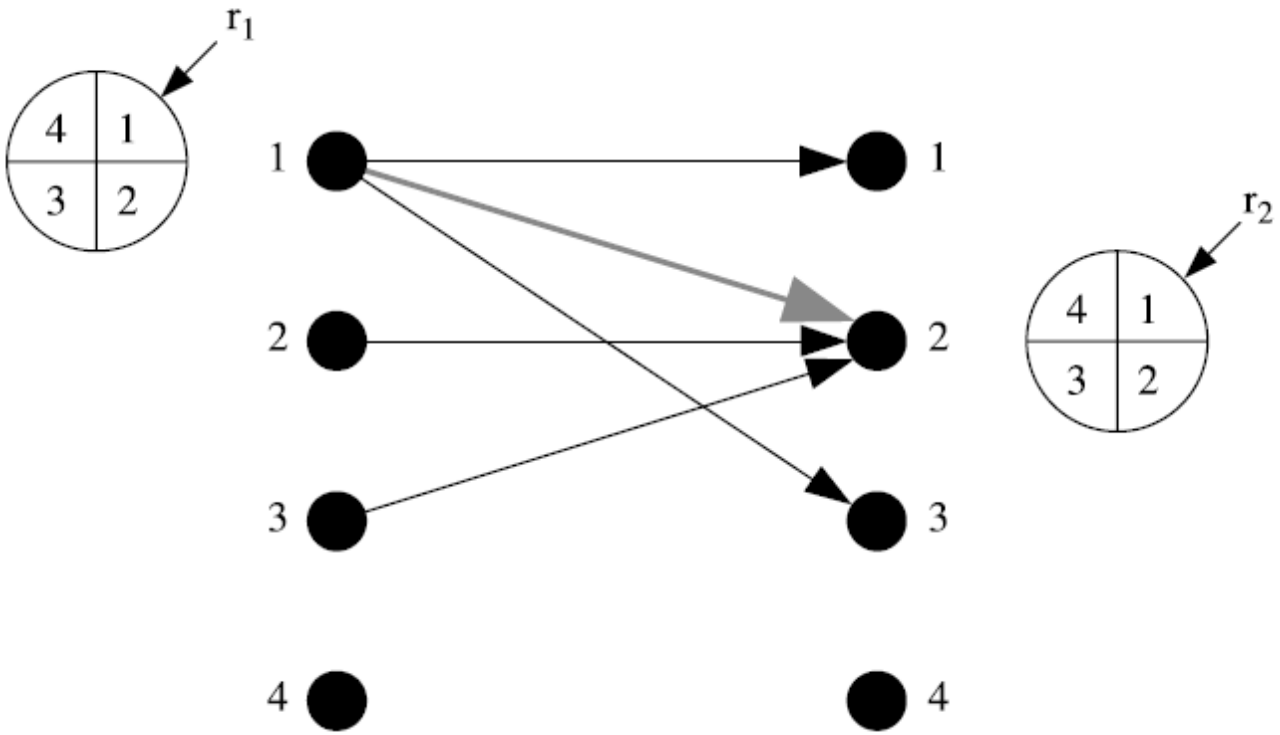
Iterative Round Robin with (iSLIP)

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

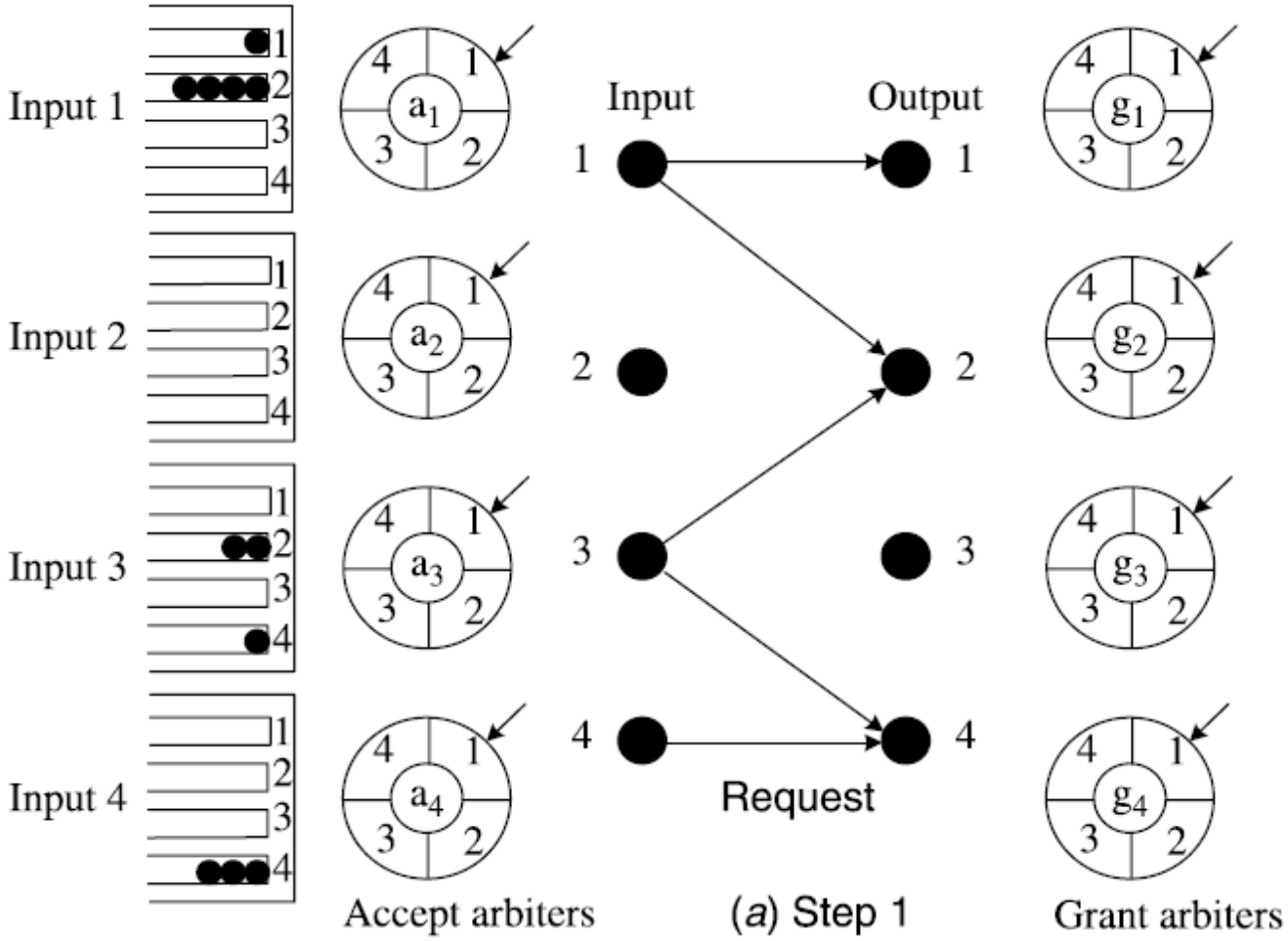
Step 2: *Grant.* If an unmatched output receives multiple requests, it chooses the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The output notifies each input whether or not its request was granted. The grant pointer g_i is incremented (module N)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

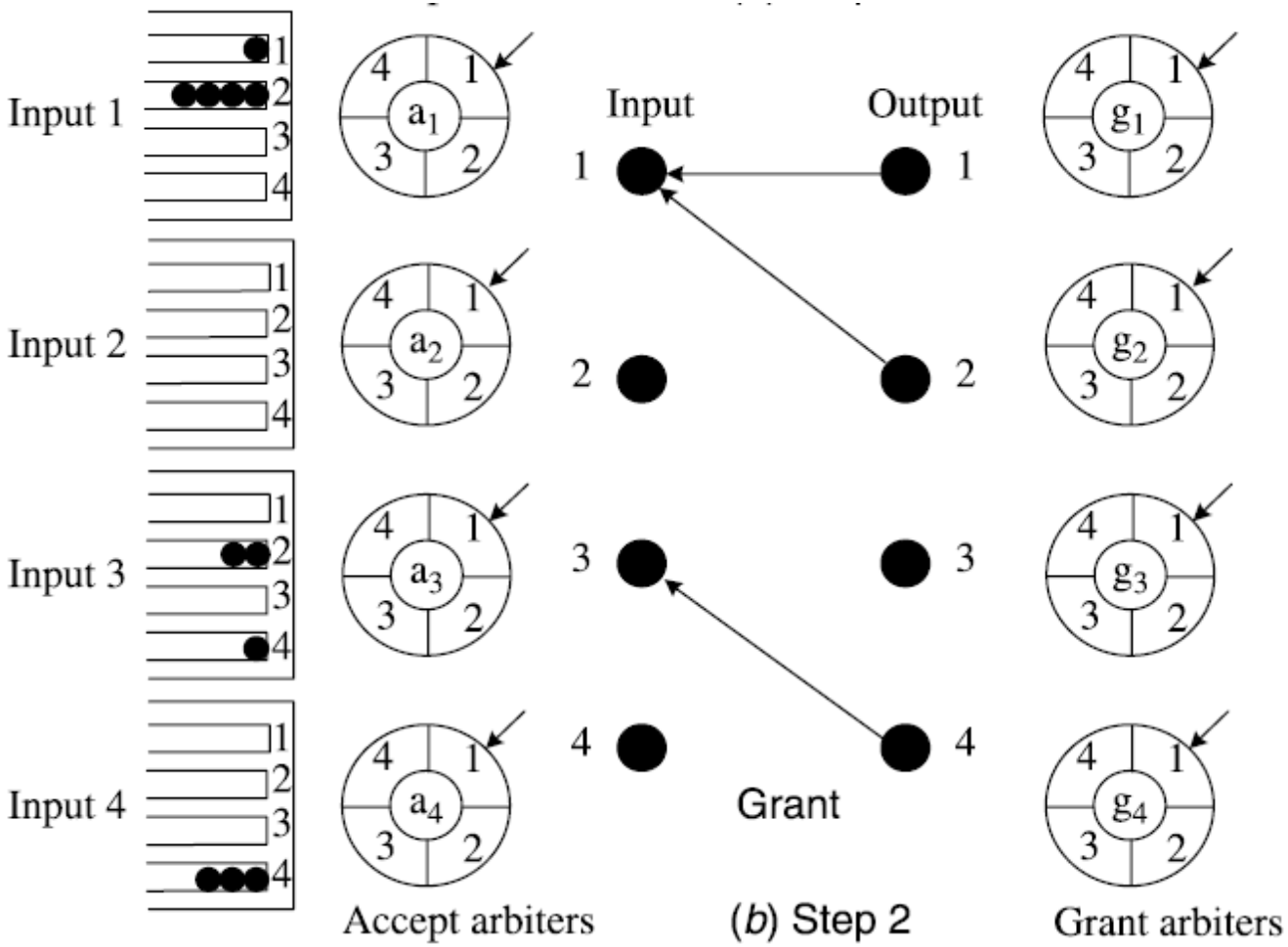
Iterative Round Robin with (iSLIP)



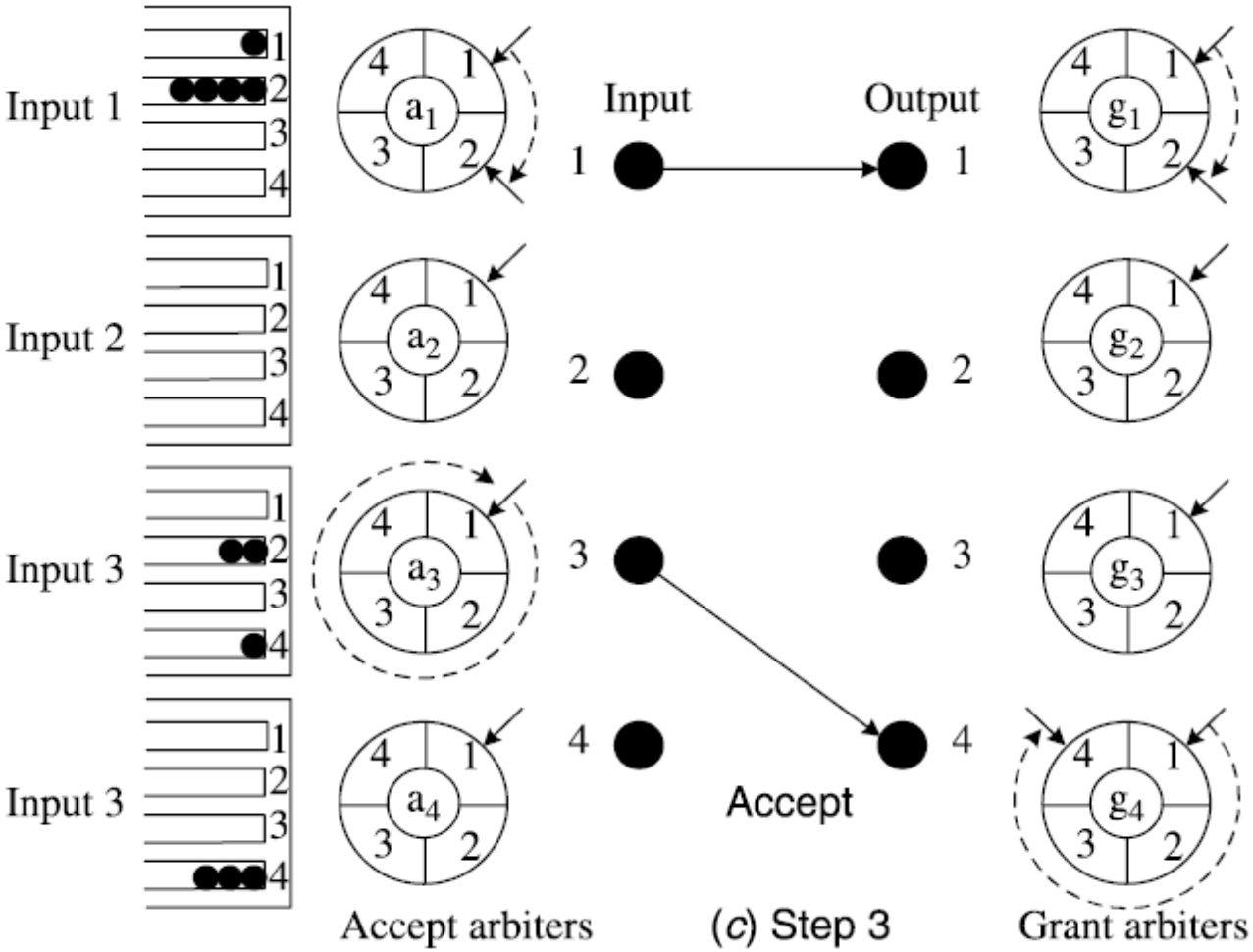
Iterative Round Robin with (iSLIP)



Iterative Round Robin with (iSLIP)



Iterative Round Robin with (iSLIP)



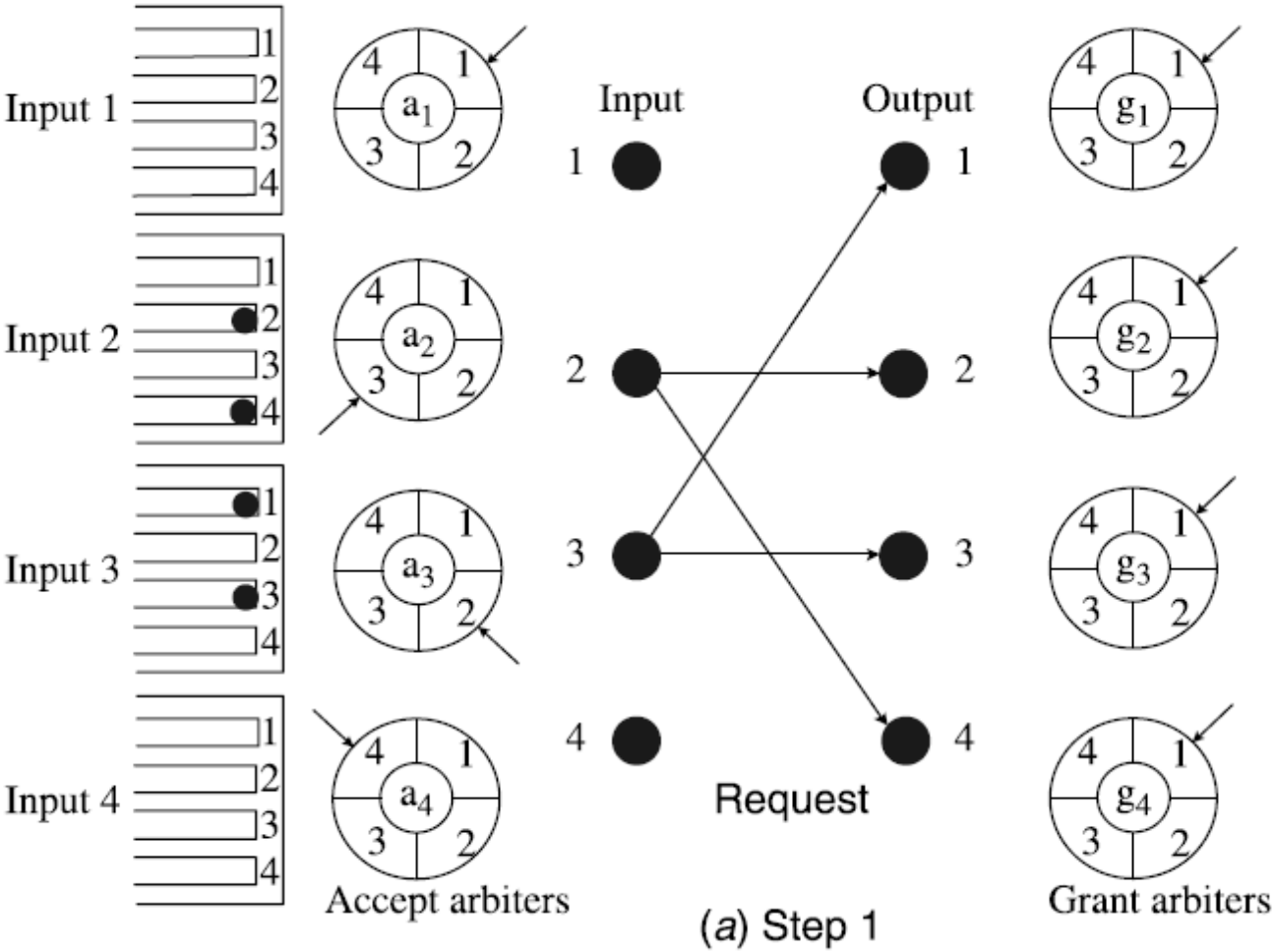
FIRM (FCFS In Round-robin Matching)

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

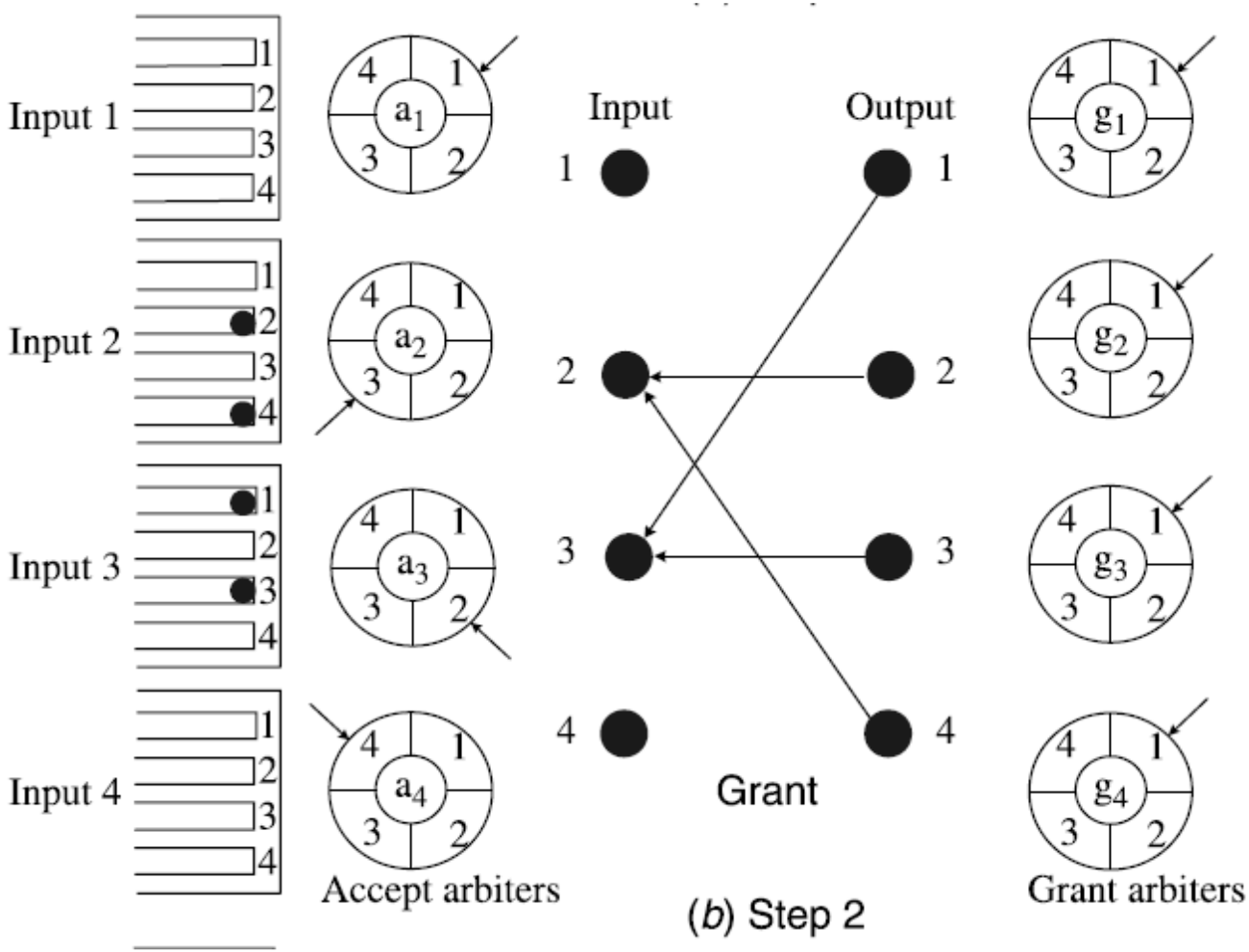
Step 2: *Grant.* If an unmatched output receives multiple requests, it chooses the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The output notifies each input whether or not its request was granted. The grant pointer g_i is incremented (modulo N)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

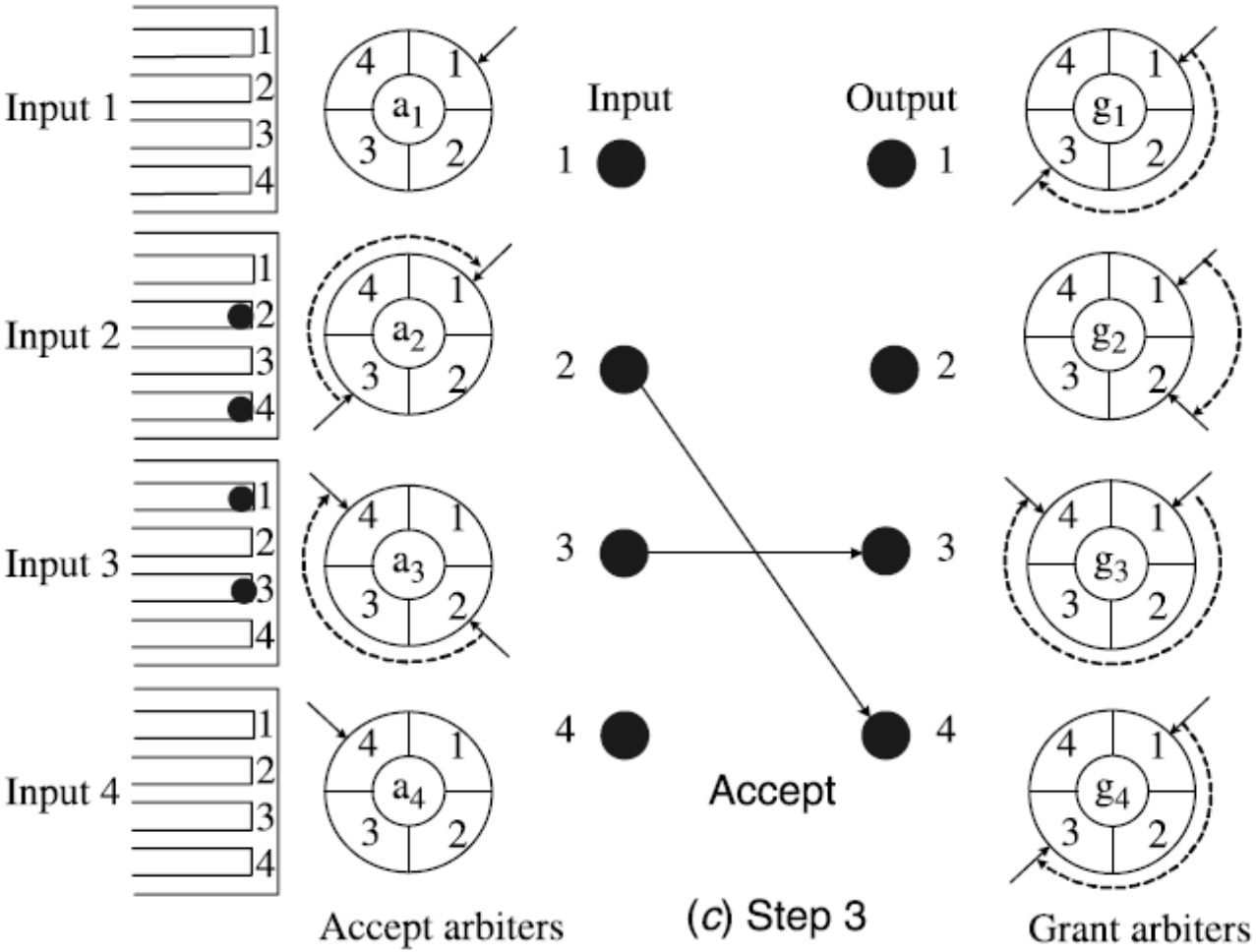
FIRM (FCFS in round-robin matching)



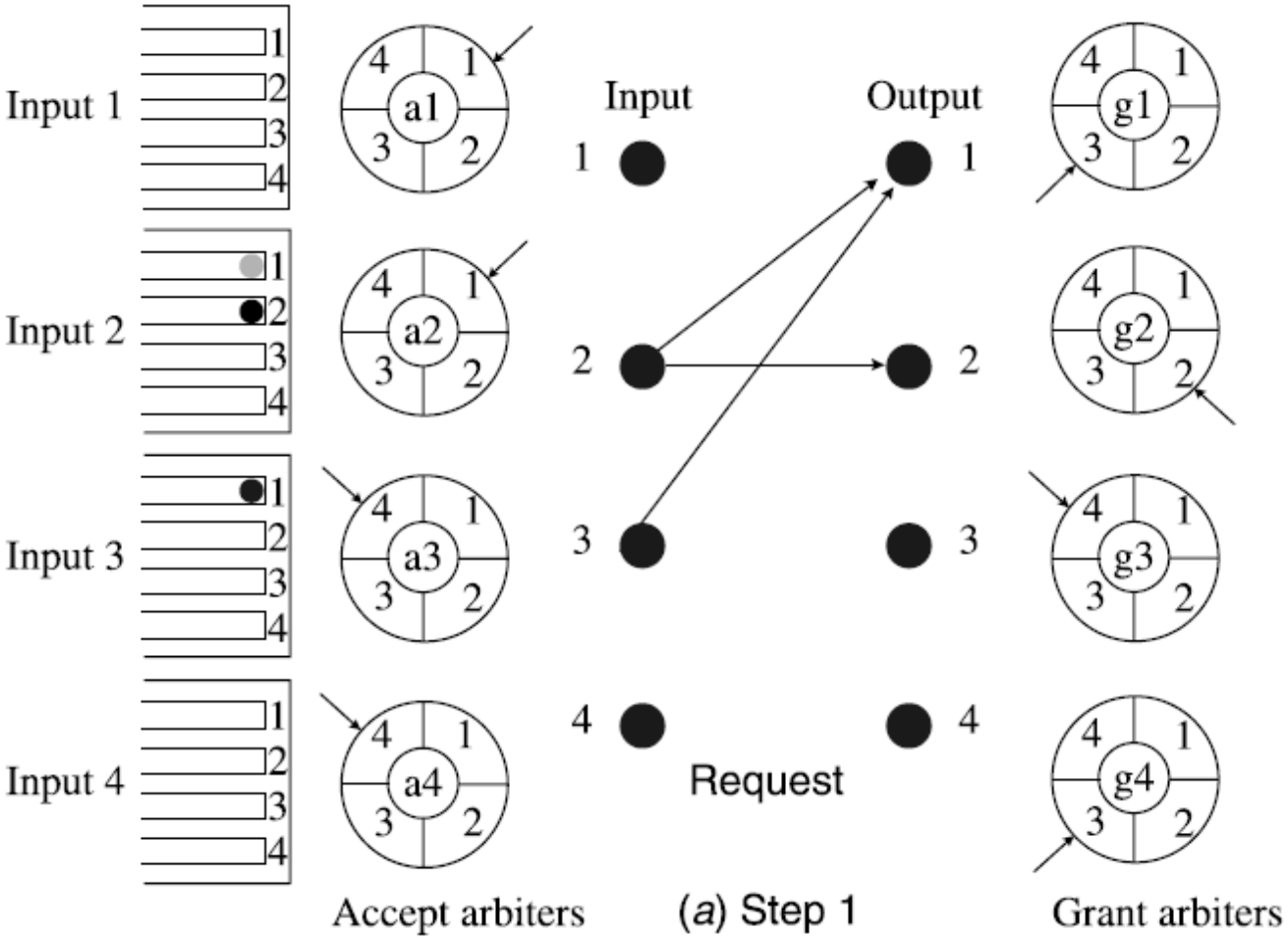
FIRM (FCFS in round-robin matching)



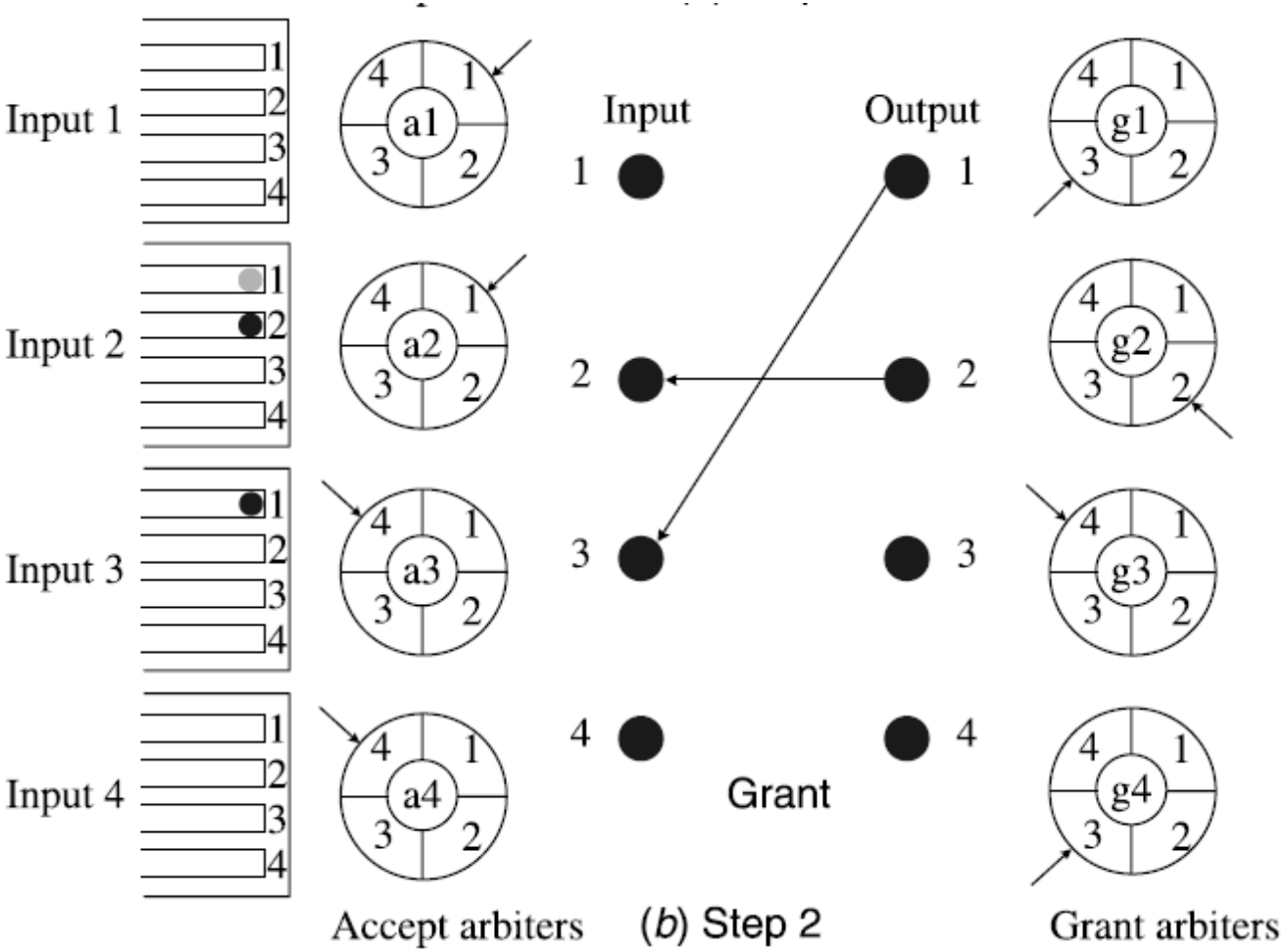
FIRM (FCFS in round-robin matching)



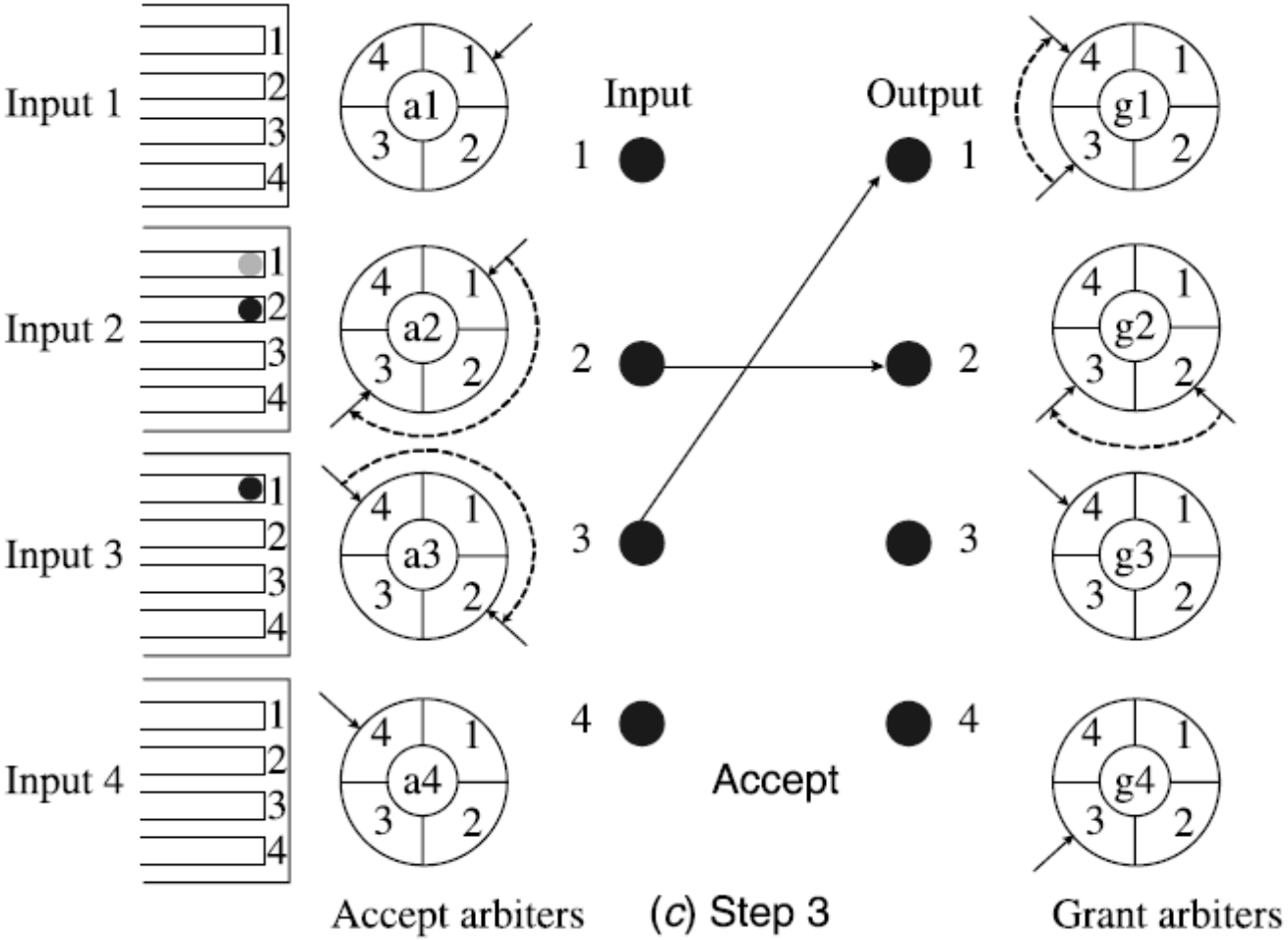
FIRM (FCFS in round-robin matching)



FIRM (FCFS in round-robin matching)



FIRM (FCFS in round-robin matching)

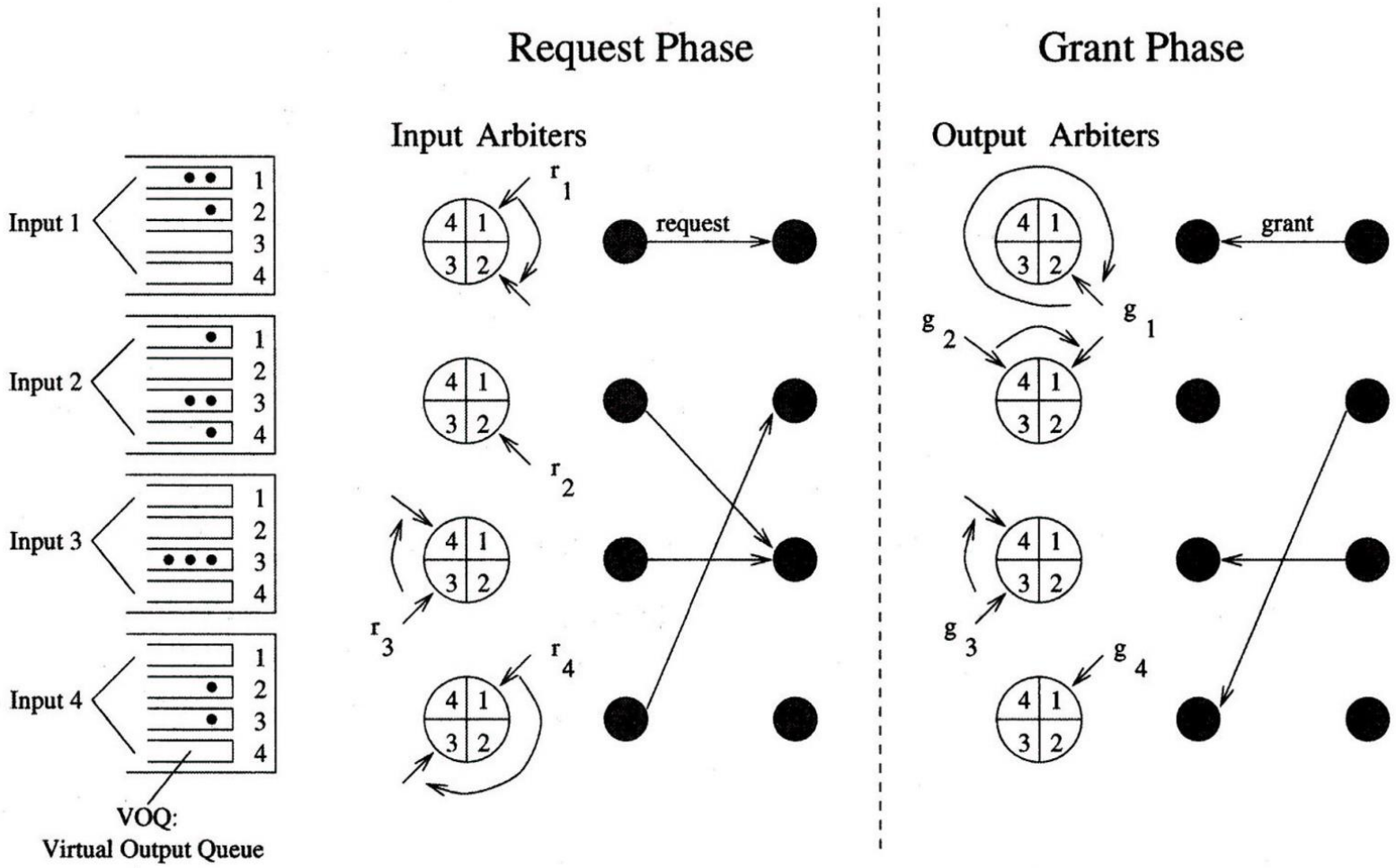


Dual Round Robin Matching (DRRM)

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.

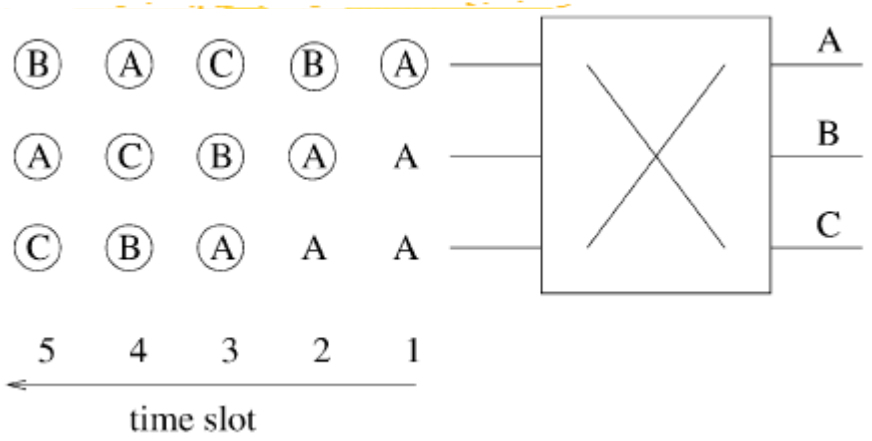
Step 2: *Grant.* If an output receives one or more requests, it chooses the one that appears next in a fixed round-robin scheduling starting from the current position of the pointer in the output arbiter. The output notifies each requesting input whether or not its request was granted. The pointer of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

Dual Round Robin Matching (DRRM)

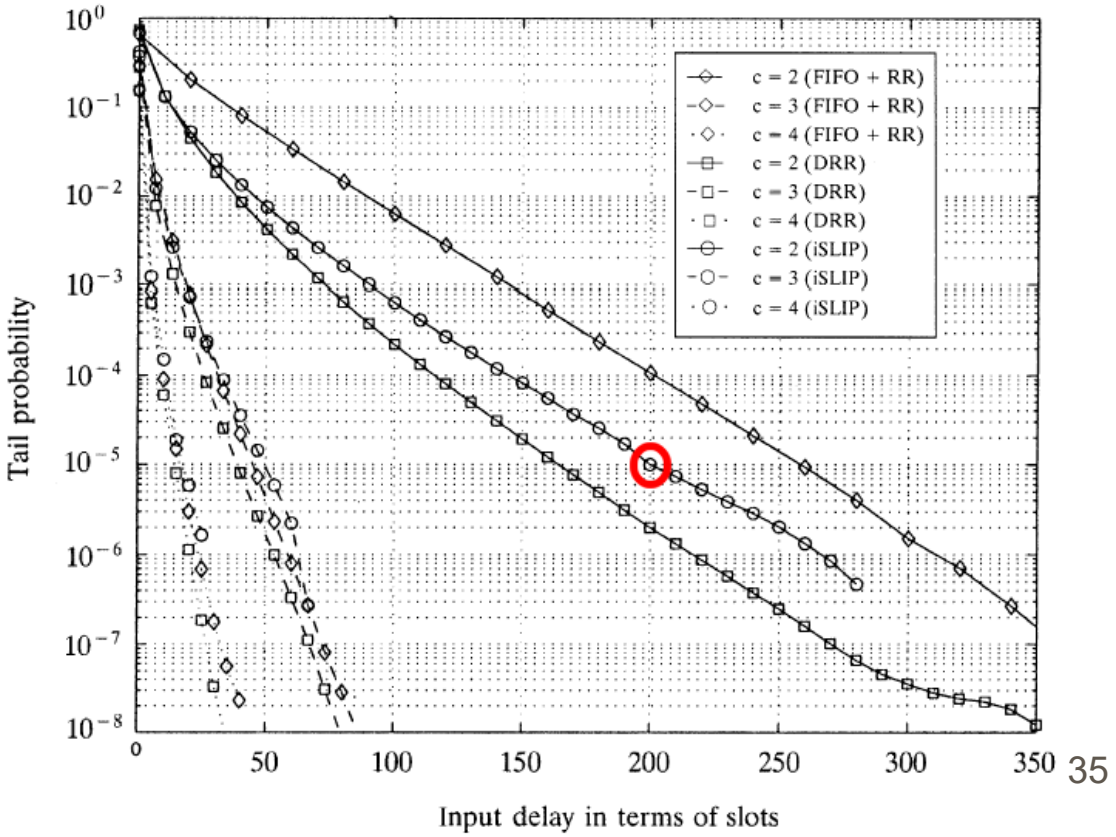


Scheduling algorithms

- Dual round robin matching (DRRM)
  - Desynchronization effect



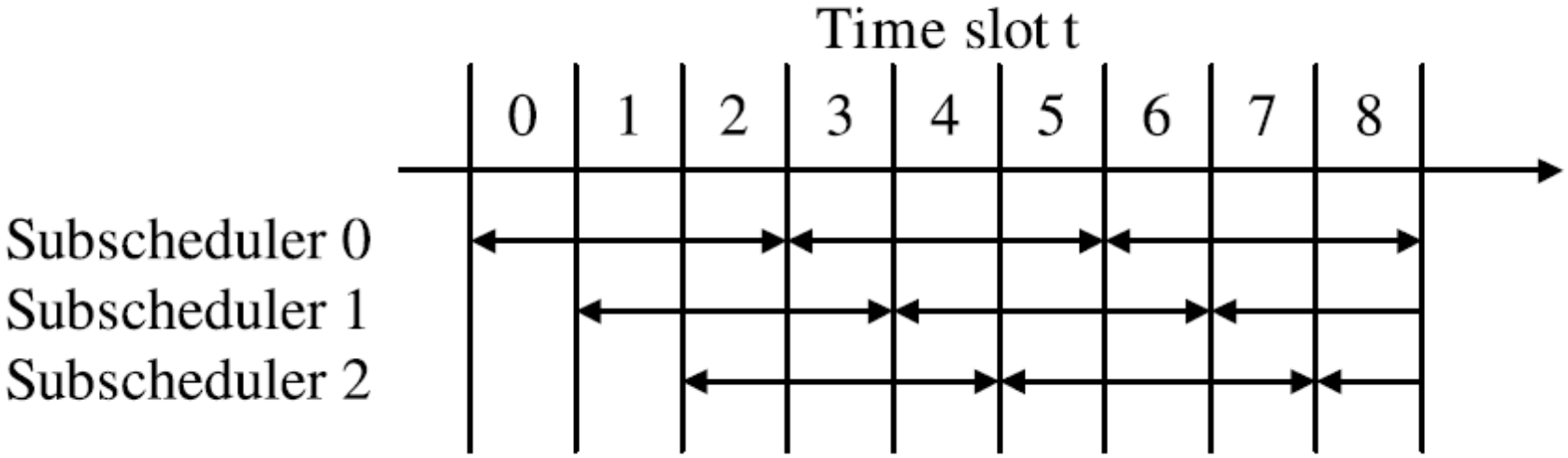
  - A comparison



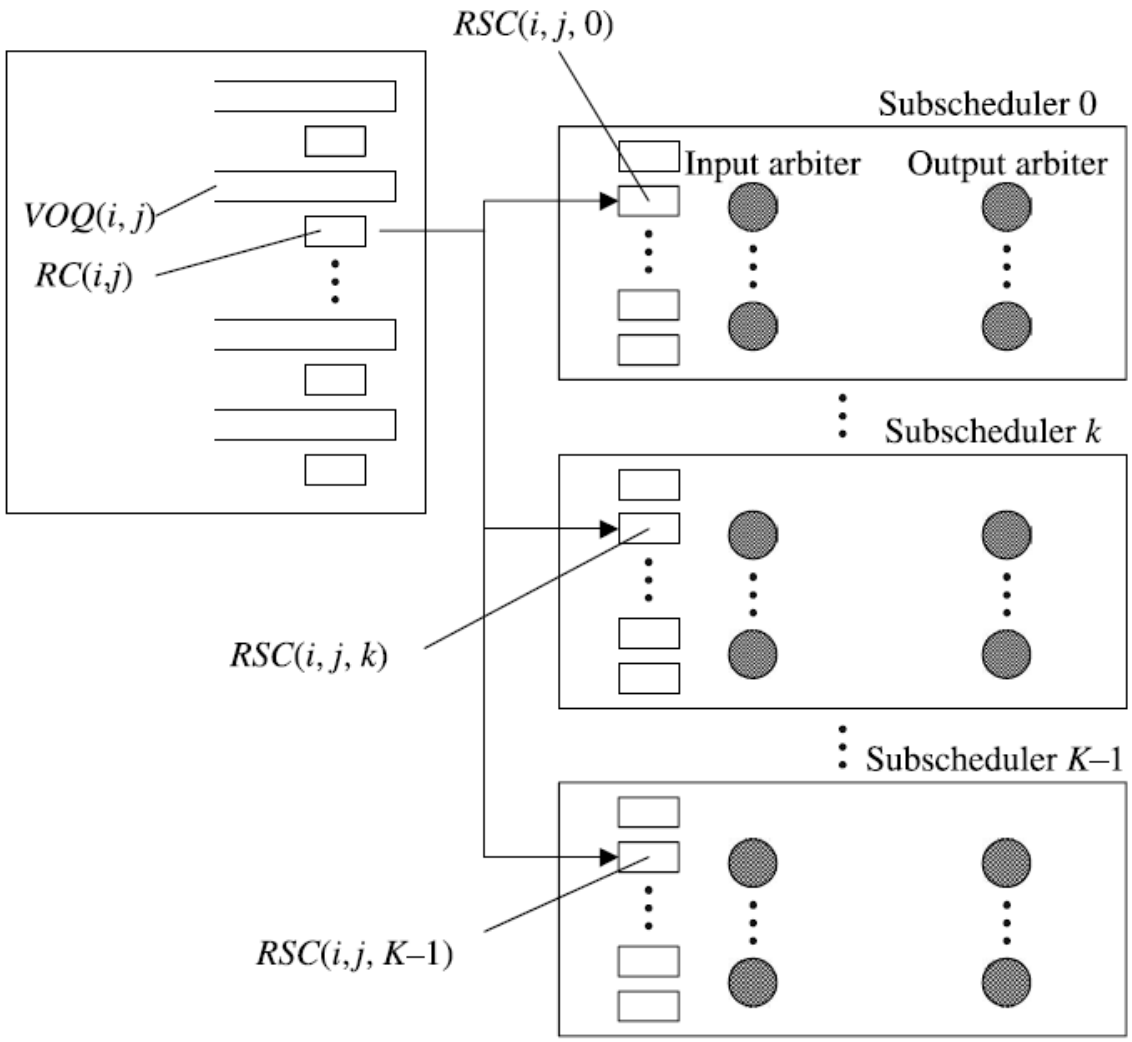
Scheduling algorithms - Pipeline technique

- Pipeline technique
  - A Bottleneck in iSLIP and DRRM:
    - Scheduling must be completed within one time slot
    - 64 bytes cells, 40 Gbits/S link -> 12.8 ns for computation!
  - Pipelining can help

Pipelined Maximal Matching



Pipelined Maximal Matching



Pipelined Maximal Matching

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.

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.

Round-Robin Greedy Scheduling (RRGS)

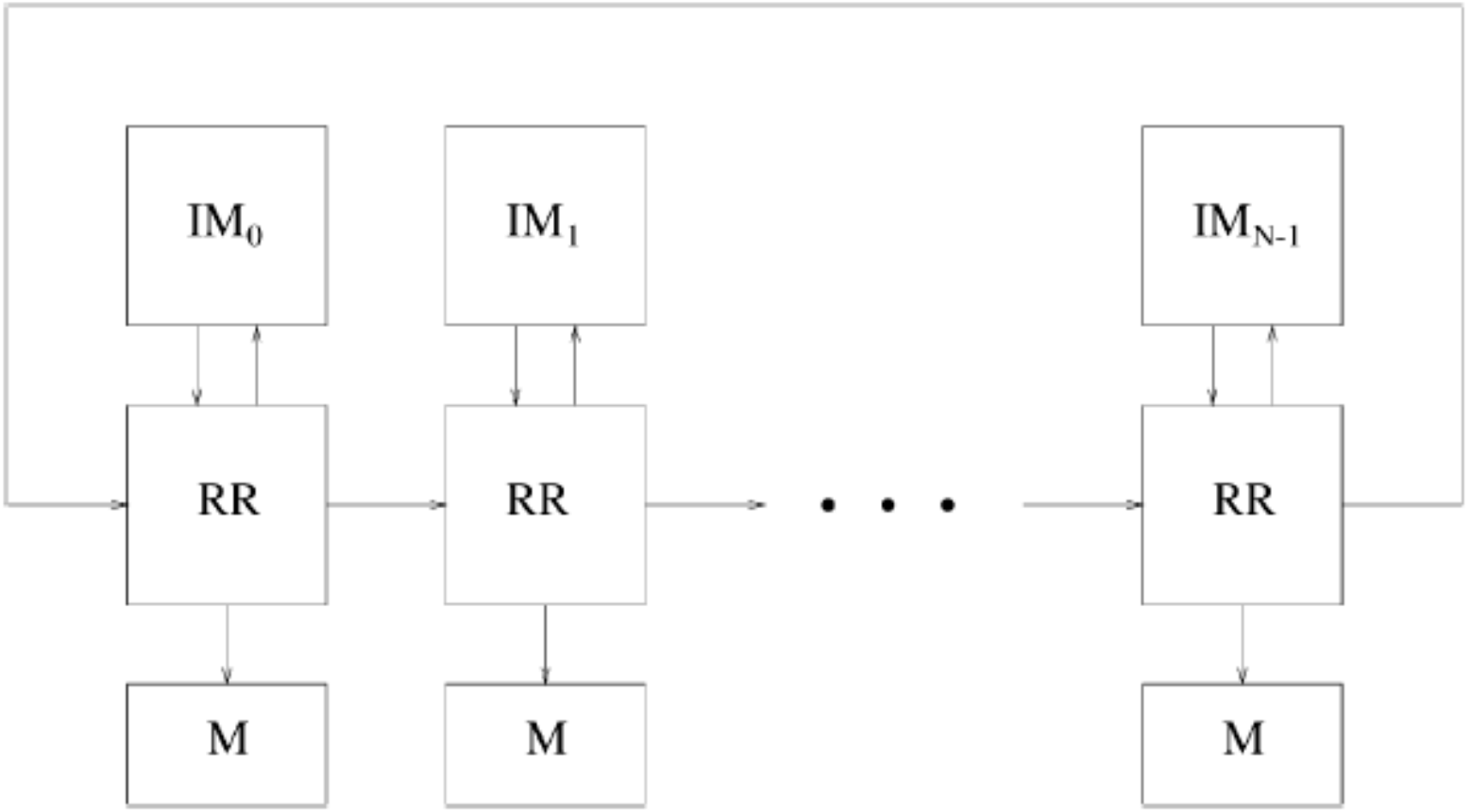
- **Step 1: $I_k = \{0, 1, \ldots, N-1\}$ is the set of all inputs; $O_k = \{0, 1, \ldots, N-1\}$ is the set of all outputs. $i = (const-k) \bmod N$ (such choice of an input that starts a schedule will enable a simple implementation).**

- **Step 2: If I_k is empty, stop; otherwise, choose the next input in a round-robin fashion according to $i = (i+1) \bmod N$.**

- **Step 3: Choose in a round-robin fashion the output j from O_k such that $(i, j) \in C_k$. If there is none, remove i from I_k and go to step 2.**

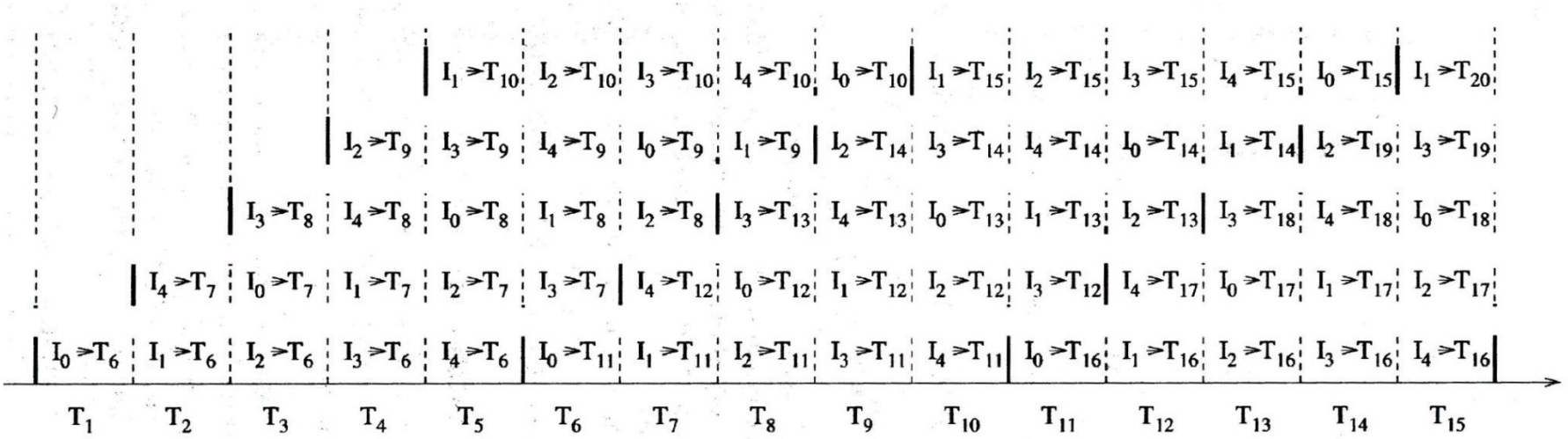
- **Step 4: Remove input i from I_k , and output j from O_k . Add (i, j) to S_k . Go to step 2.**

Central controller for RRGS protocol



Input-Buffered Switches

Round Robin Greedy Scheduling

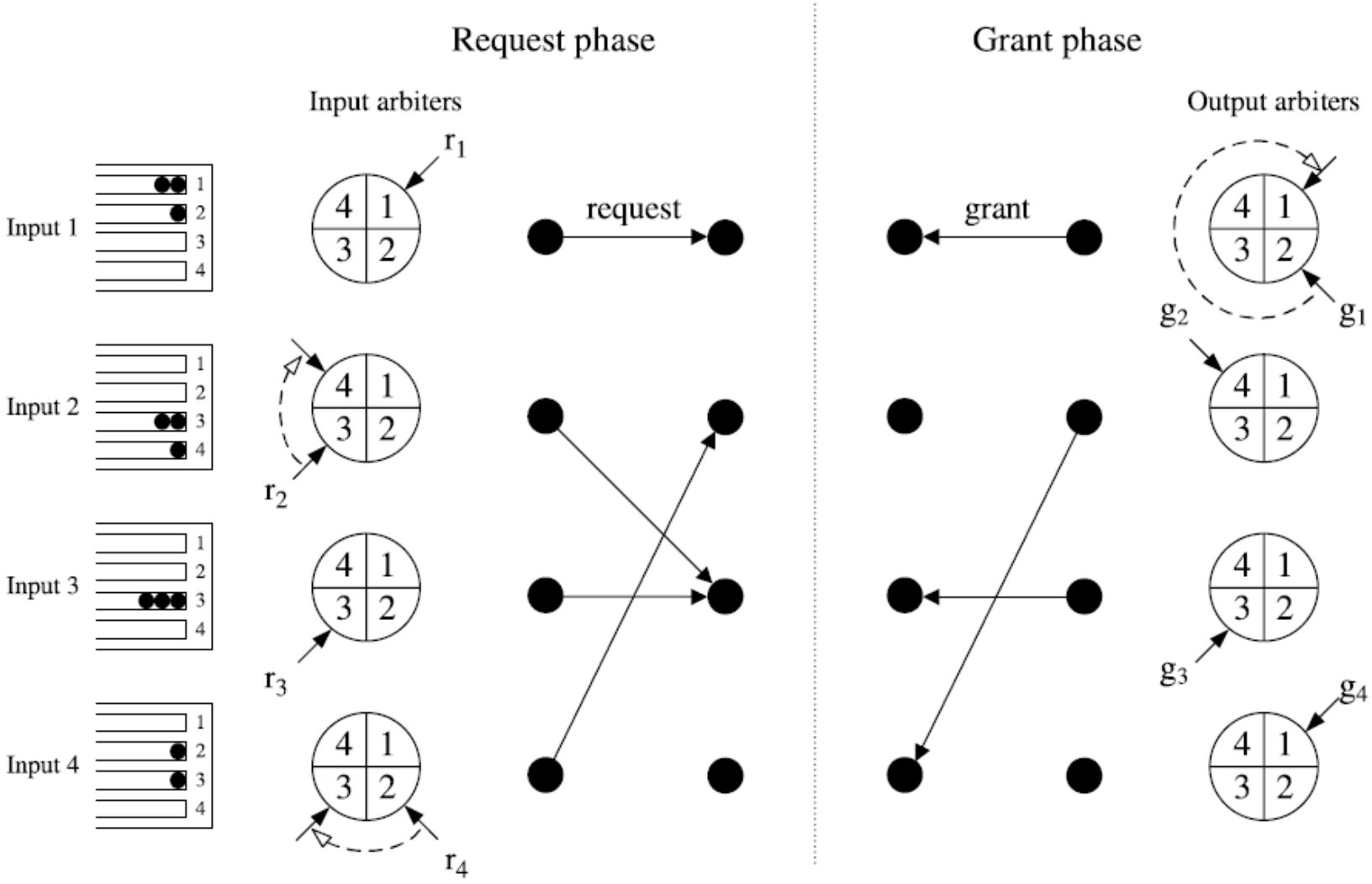


Exhaustive Dual Round-Robin Matching (EDRRM)

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.

Exhaustive Dual Round-Robin Matching (EDRRM)



Randomized Matching Algorithms

- **Randomized Algorithm with Memory**
- **De-randomized Algorithm with Memory**
- **Variant Randomize Matching Algorithms**
  - ***APSARA.***
  - ***LAURA***
  - ***SERENA***
- **Polling Based Matching Algorithms**
  - **EMHW**
  - ***HE-iSLIP***

Randomized Matching Algorithms

- *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.

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.

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

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

Variant Randomize Matching Algorithm

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.

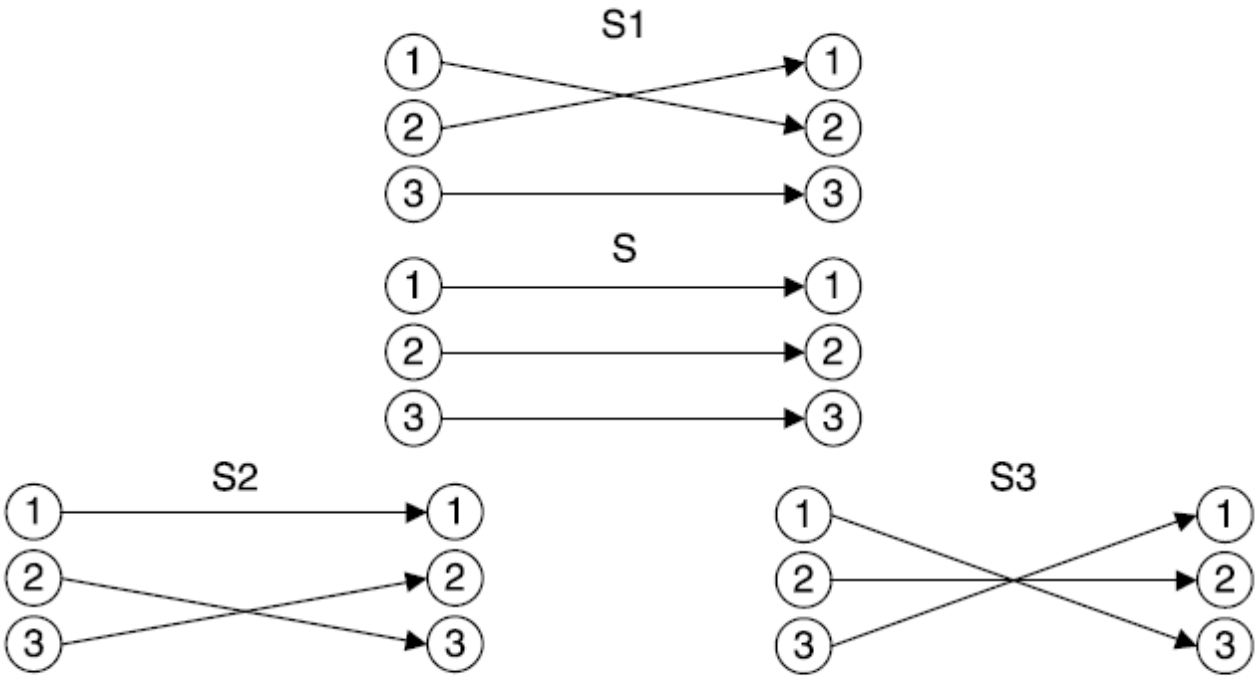


Figure 7.23 Example of neighbors in APSARA algorithms.

Variant Randomize Matching Algorithm

APSARA Algorithm:

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

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

Variant Randomize Matching Algorithm

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.

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 $t+1$.

Variant Randomize Matching Algorithm

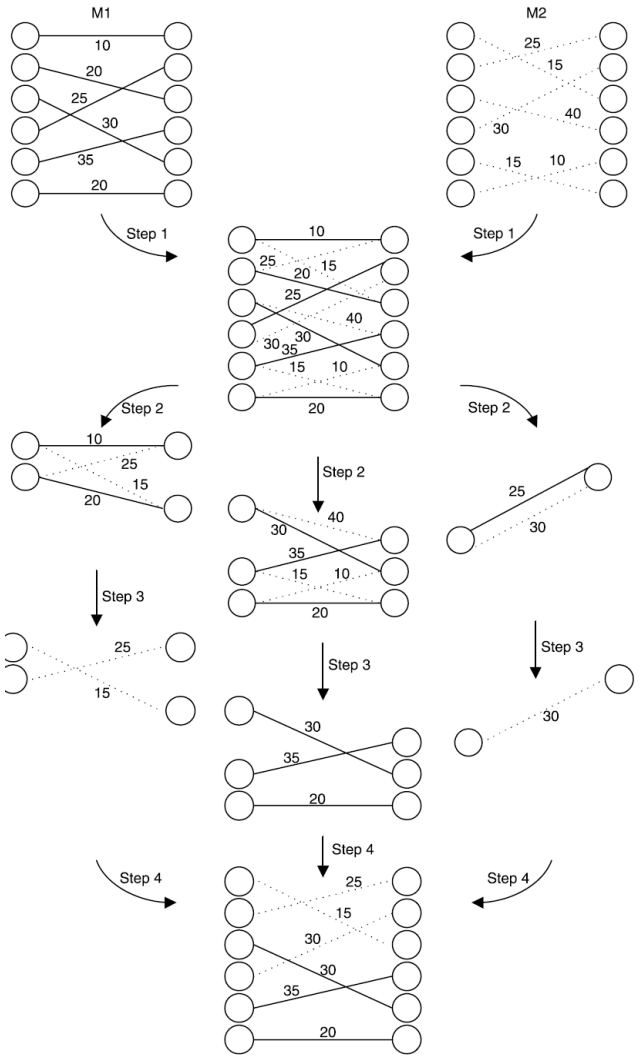


Figure 7.24 Example of the MERGE procedure.

Polling Based Matching Algorithms

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

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

Polling Based Matching Algorithms

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)

E-iSLIP

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.

Polling Based Matching Algorithms

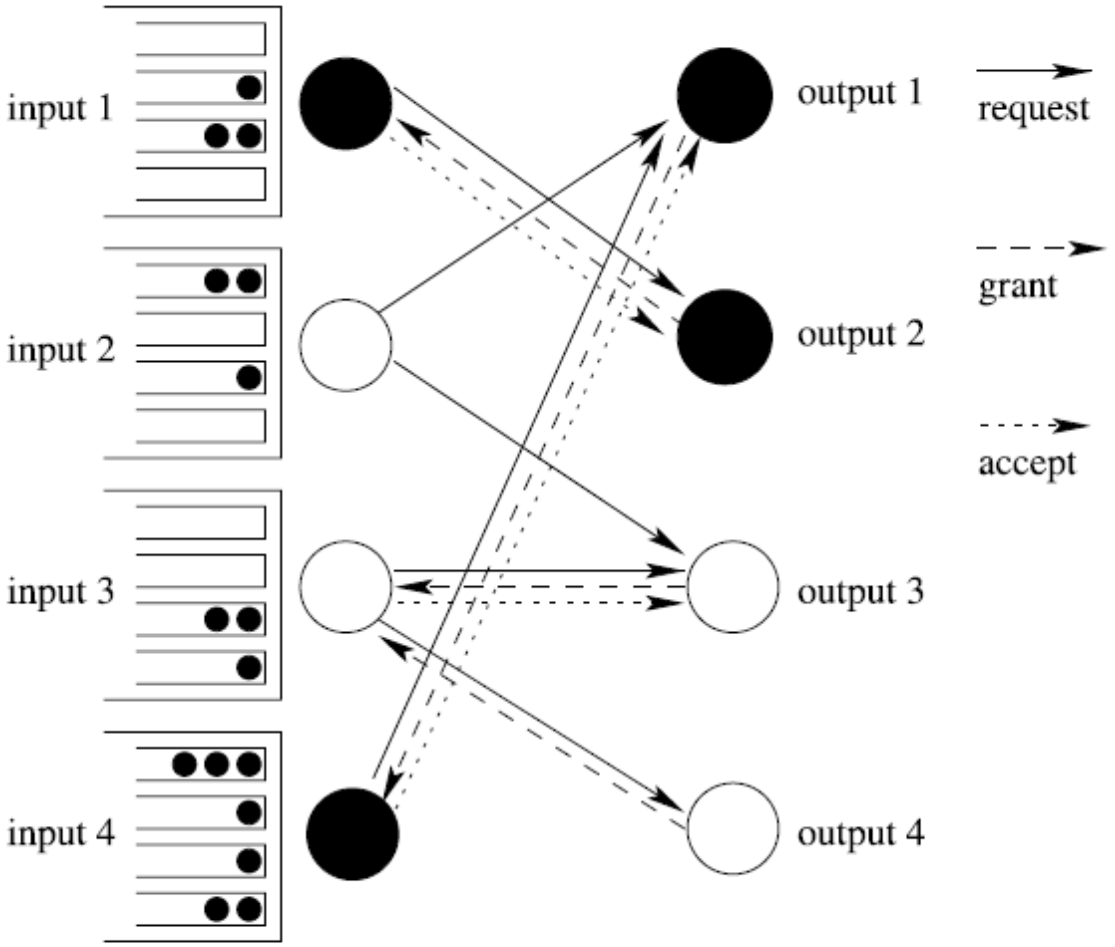


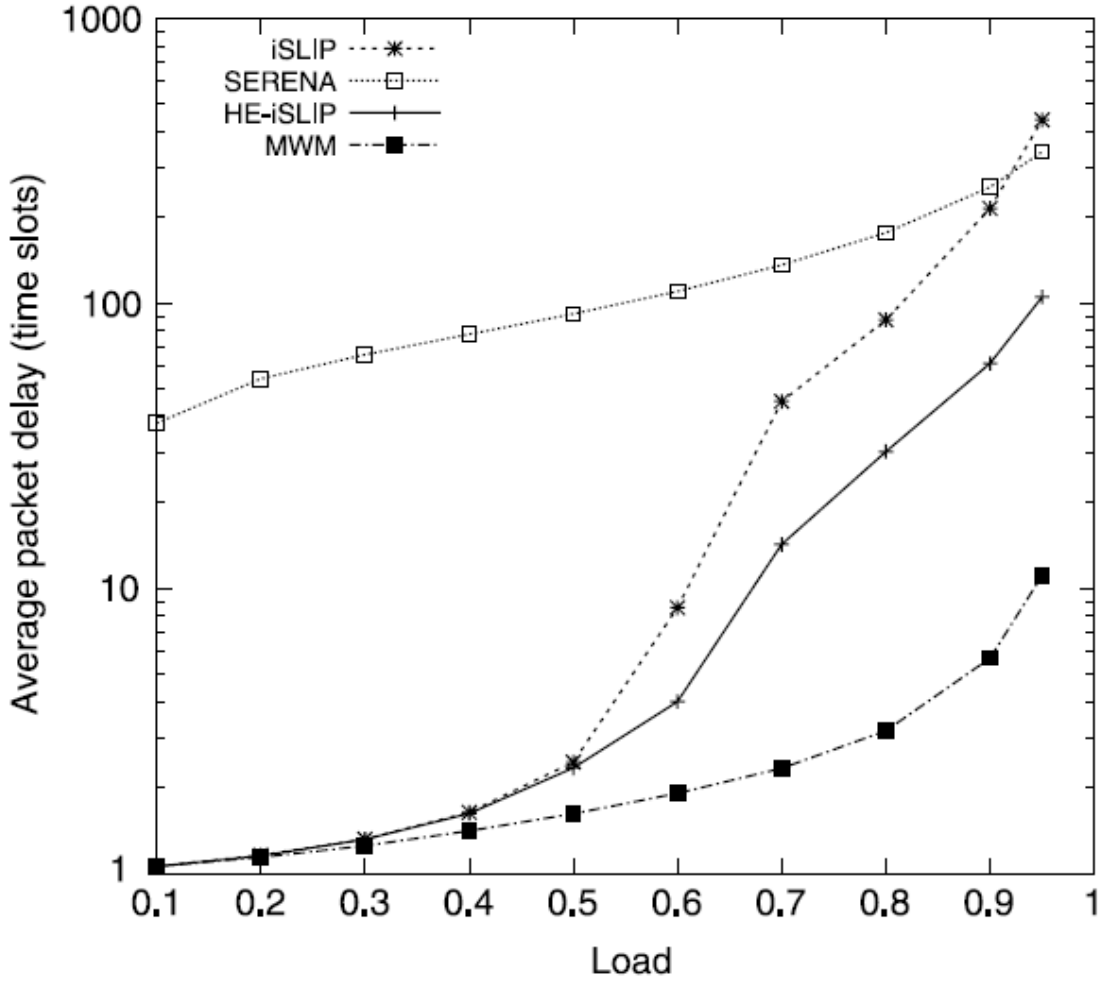
Figure 7.25 Example of E-*i*SLIP algorithm.

Polling Based Matching Algorithms

HE-iSLIP

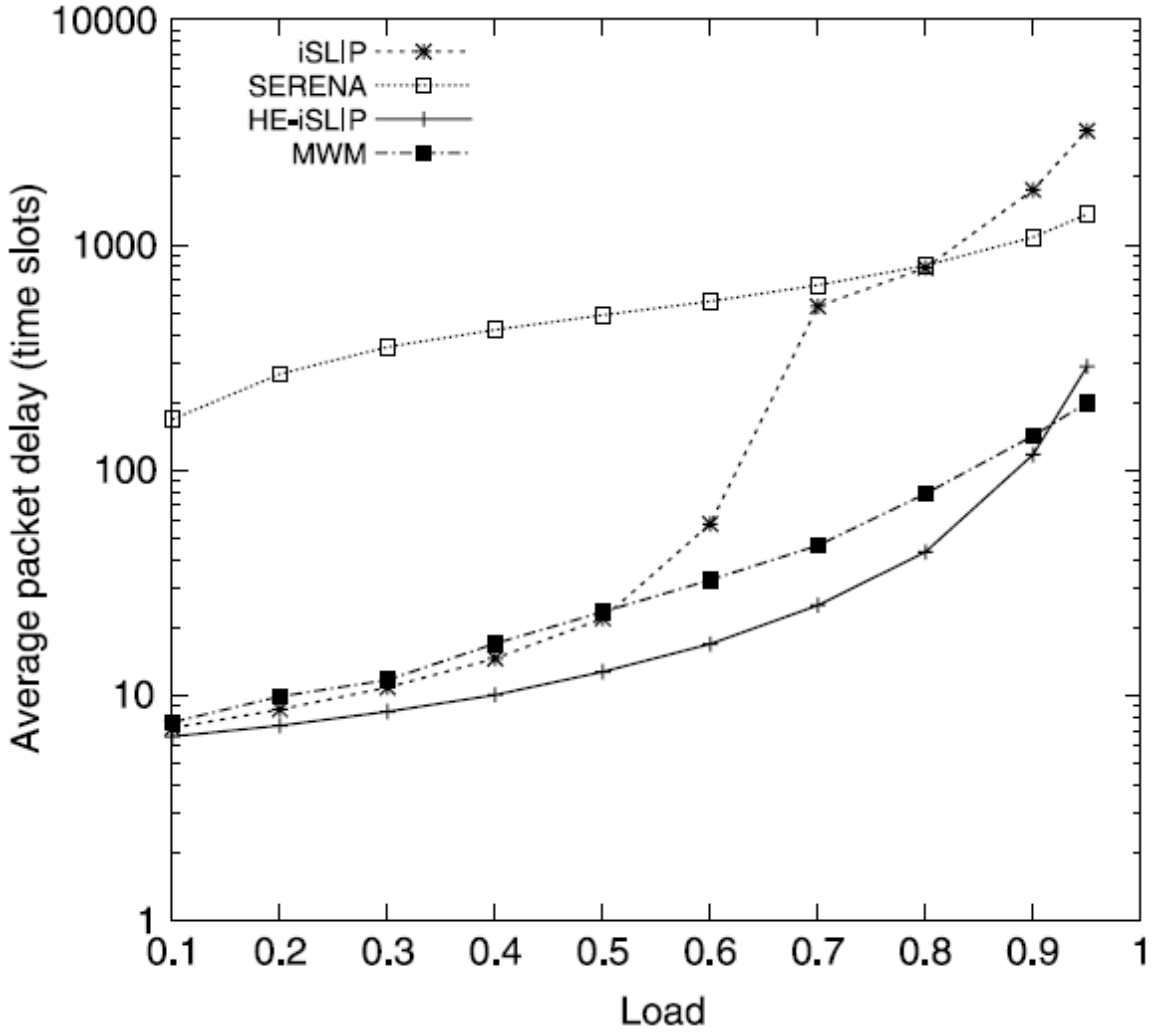
1. Run E-*i*SLIP, which generates a match $Z(t+1)$ based on the previous match $S(t)$ and updates the pointer and the state of each input and output.
2. Generate a match $H(t+1)$ by Hamiltonian walk.
3. Compare the weights of $Z(t+1)$ and $H(t+1)$. If the weight of $H(t+1)$ is larger, set $S(t+1) = H(t+1)$.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

Simulated Performance



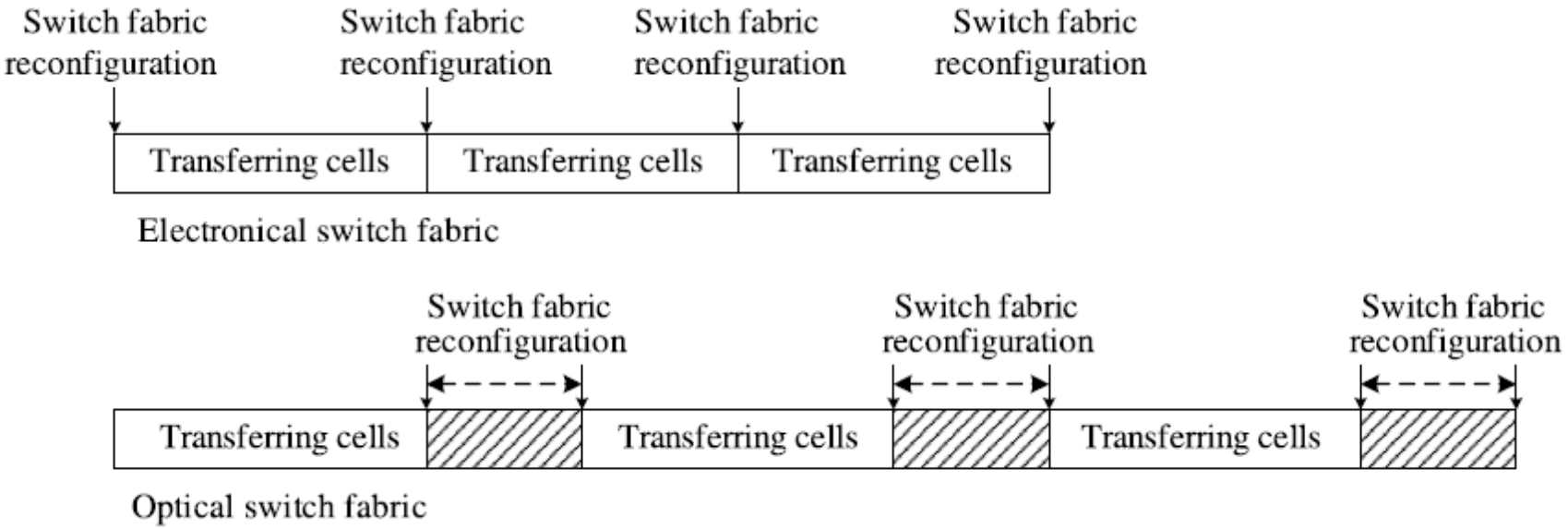
Average packet delay of *i*SLIP, SERENA, HE-*i*SLIP and MWM under uniform traffic when the packet length is 1 cell.

Simulated Performance



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

Frame-Based Matching



Frame-Based Matching

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

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

Scheduling algorithms

Output-Queuing Emulation:

The major drawback of input queuing is that the queuing delay between inputs and outputs is variable, which makes delay control more difficult.

Question:

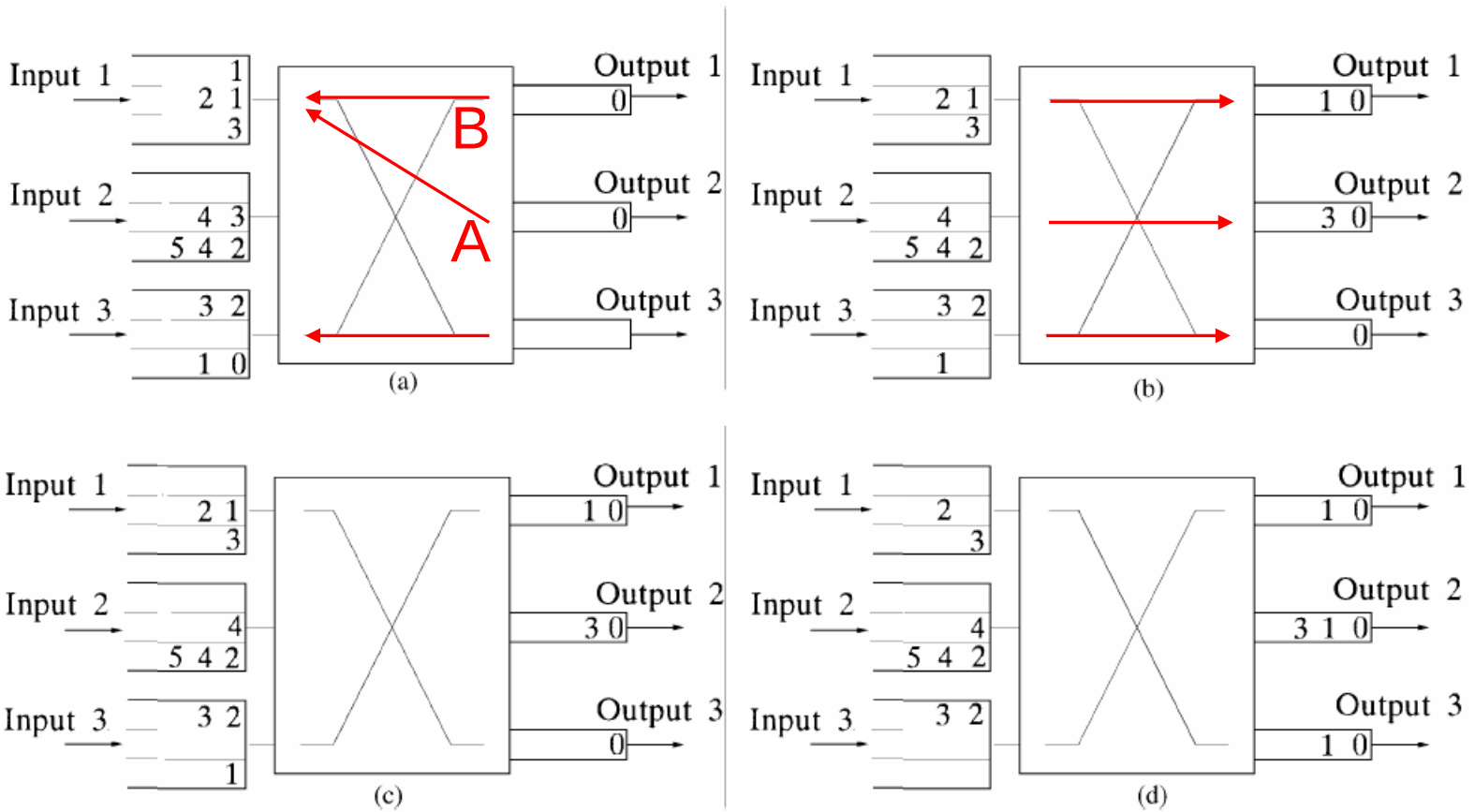
Can an input output-buffered switch with a certain speedup behave identically to an output-queued switch?

Scheduling algorithms - MUCFA

- Most urgent cell first algorithm (MUCFA)
  - Output queuing emulation
  - TL: Time to leave (not time to live!)
  - TL gets the number of cells ahead of this cell when entering
  - Most urgent cell: the cell with the smallest TL
  - The algorithm
    - Outputs send requests for the most urgent cells to the corresponding inputs
    - If an input gets multiple requests, selects the most urgent cell
    - Outputs which lose contention, request for next most urgent cell
    - These steps are repeated until no more matching is possible

Scheduling algorithms - MUCFA

- An example



Scheduling algorithms – LOOFA

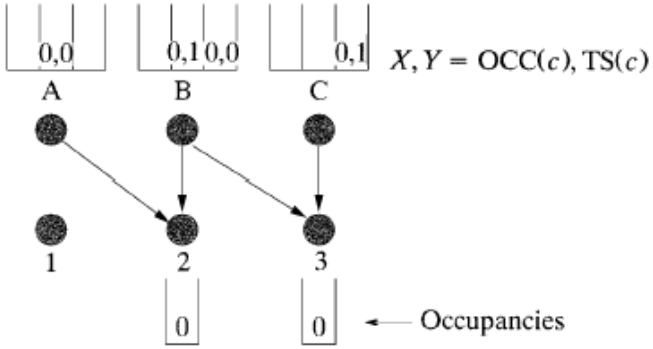
- Lowest output occupancy cell first algorithm (LOOFA)
  - 100% throughput
  - Speedup of 2
  - Bounded cell delay
  - Two versions
    - Greedy
    - Best first
  - Parameters associated with a cell c
    - Output occupancy: OCC(c)
      - The number of cells in output queue of destination port of c
    - Timestamp: TS(c)
      - Age of the cell c

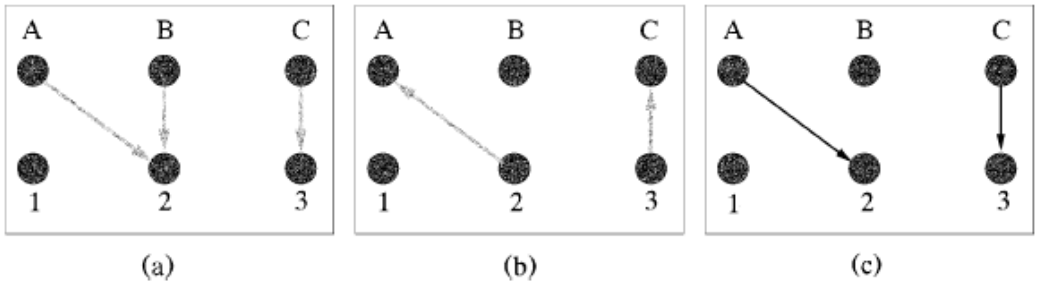
Scheduling algorithms – LOOFA

Greedy version of algorithm

- Initially, all inputs and outputs are unmatched
- Each unmatched input sends its request to the output with the lowest occupancy
- 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
- Repeat from step 2, until no more matching is possible

An example





Scheduling algorithms – LOOFA

- Best-first version of algorithm
  - Initially, all inputs and outputs are unmatched
  - Among unmatched outputs, the one having the smallest occupancy is selected. All inputs having a cell for it, send their request.
  - The output, grants the request having the smallest timestamp
  - Repeat from step 2, until no more matching is possible, or N iterations are completed