CHAPTER 3

The SLIP Algorithm with Multiple Iterations

1 Introduction

In this chapter we consider the SLIP algorithm with multiple iterations per cell time. We shall call the generic algorithm “iterative SLIP” (*i*-SLIP). When the number of iterations *n* is known, we shall call the algorithm *n*-SLIP. For example, Chapter 2 focussed on 1-SLIP.

With more than one iteration, the iterative SLIP algorithm improves upon the performance of 1-SLIP: each iteration attempts to add connections not made by earlier iterations.

We begin this section with a description of *i*-SLIP, emphasizing the differences from non-iterative SLIP. In particular, we pay careful attention to the way that the arbiter pointers are updated. In Section 3 we present some results from a simulation study of *i*-SLIP. As we found for 1-SLIP, iterative SLIP is stable for all admissible uniform i.i.d. Bernoulli arrivals. We find that the performance improves as we increase the number of iterations up to about log_2N, for an NxN switch. Once again, we shall see that *desynchronization* of the output arbiters with respect to each other plays an important rôle in achieving low latency. However, we will also see that the basic *i*-SLIP algorithm tends to do a worse job of desynchronizing the arbiters as the number of iterations increase.

In Section 4 we try to improve upon the basic iterative SLIP algorithm by changing the rules for updating the pointers so that desynchronization is improved when the number of iterations is increased. We consider the extra complexity that these changes introduce.

Finally, in Section 5 we describe an implementation of iterative SLIP, showing that although it may take longer to execute, the implementation is only slightly more complex than the implementation of non-iterative SLIP.

2 The Iterative SLIP Matching Algorithm

2.1 Description

The *i*-SLIP algorithm is an enhancement of the SLIP algorithm described in Chapter 2, but has a number of differences specific to its iterative behavior. As before, at the beginning of each cell time, the match process begins over. All inputs and outputs are initially unmatched and only those inputs and outputs not matched at the end of one iteration are eligible for matching in the next. Connections made in one iteration are never removed by a later iteration, even if a larger sized match would result. The three steps of each iteration operate in parallel on each output and input and 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 fixed, 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 to the highest priority element of the round-robin schedule is incremented (modulo 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 unmatched input receives a grant, it accepts the one that appears next in a fixed, round-robin schedule starting from the highest priority element. The pointer a_i to the highest priority element of the round-robin schedule is incremented (modulo N) to one location beyond the accepted output *only if this input was matched in the first iteration.*

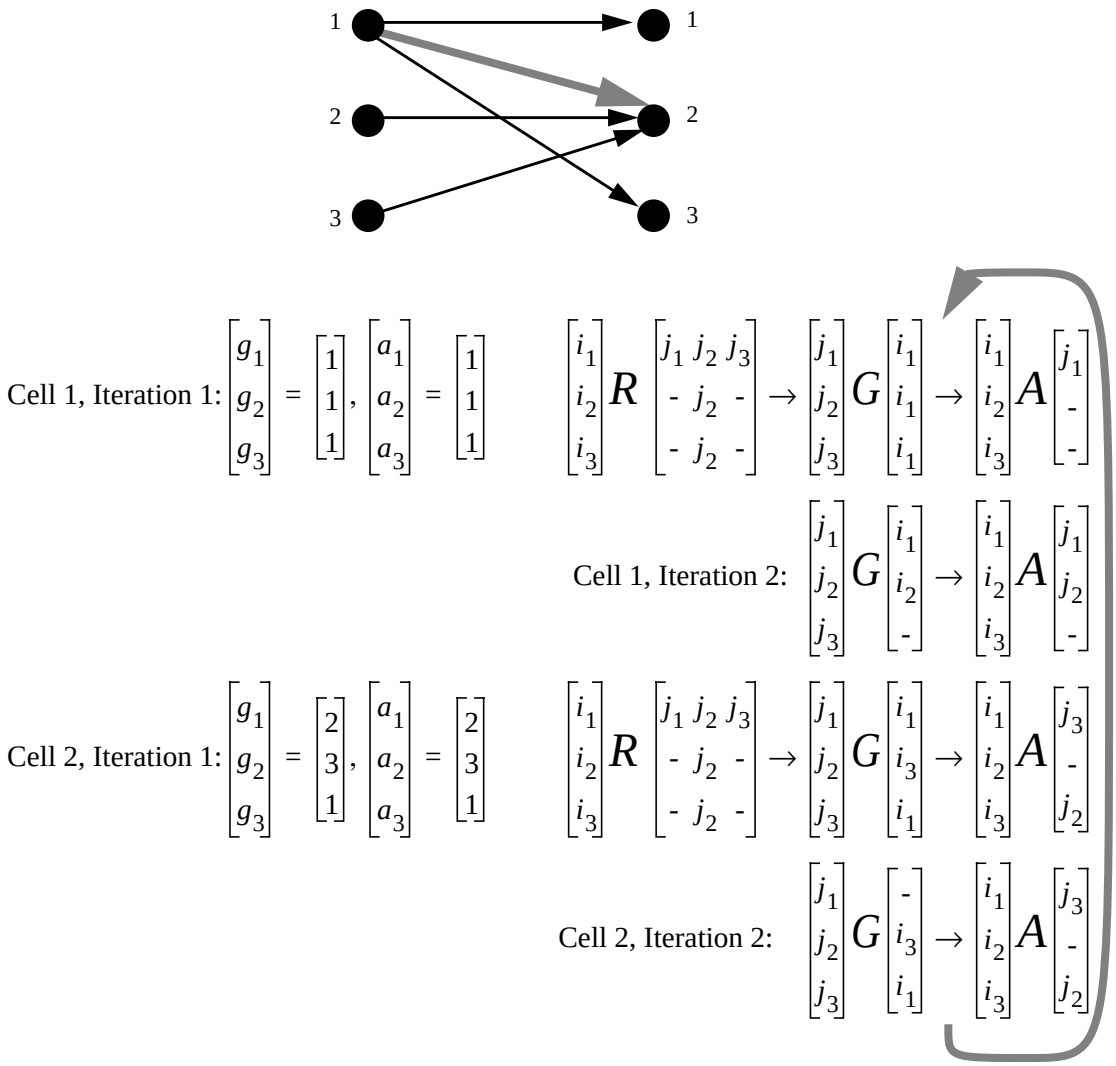


FIGURE 3.1 Example of starvation, if pointers are updated after every iteration. The 3x3 switch is heavily loaded, i.e. all active connections have an offered load of 1 cell per cell time. The sequence of grants and accepts repeats after 2 cell times, even though the (highlighted) connection from input 1 to output 2 has not been made. Hence, this connection will be starved indefinitely.

2.2 Updating Pointers

Note that pointers g_i and a_i are only updated for matches found in the first iteration. Connections made in subsequent iterations do not cause the pointers to be updated. This is to avoid starvation. To understand how starvation can occur, we refer to the example of a 3x3 switch with 5 active and heavily loaded connections, shown in Figure 3.1. The switch is scheduled with the *2*-SLIP algorithm, except in this case the pointers are updated after both iterations. The figure shows the sequence of decisions by the grant and accept arbiters; for this traffic pattern, they form a repetitive

cycle in which *the highlighted connection from input 1 to output 2 is never served*. Each time the round-robin arbiter at output 2 grants to input 1, input 1 chooses to accept output 1 instead.

Starvation is eliminated if the pointers are not updated after the first iteration. In the example, output 2 would continue to grant to input 1 with highest priority until it is successful.

2.3 Properties

The *i*-SLIP algorithm has the following properties:

Property 1. Connections matched in the first iteration become the lowest priority in the next cell time. This is the same as Property 1 of 1-SLIP described in Chapter 2 Section 3.

Property 2. No connection is starved. As with 1-SLIP, and because of the requirement that pointers are not updated after the first iteration, an output will continue to grant to the highest priority requesting input until it is successful.

Property 3. For *i*-SLIP with 1 iteration, and under heavy load, queues with a common output all have the same throughput. This is the same as in Chapter 2 Section 3.

Property 4. For *i*-SLIP with more than one iteration, and under heavy load, queues with a common output may each have a different throughput. An example of this property is shown in Figure 3.2 for a heavily loaded 3x3 switch scheduled using 2-SLIP. The state of the grant and accept arbiters forms a cycle that repeats every three cell times. Note that although all non-empty queues are served, Q(2,3) is served *twice* per cycle whereas Q(1,3) is served only *once*.

Property 5. The algorithm will converge in at most N iterations. Each iteration will schedule zero, one or more connections. If zero connections are scheduled in an iteration then the algorithm has converged: no more connections can be added with more iterations. Therefore, the slowest convergence will occur if exactly one connection is scheduled in each iteration. At most N connections can be scheduled (one to every input and one to every output) which means the algorithm will converge in at most N iterations.

Property 6. The algorithm will not necessarily converge to a maximum sized match. At best, it will find a *maximal* match: the largest size match without removing connections made in earlier iterations.

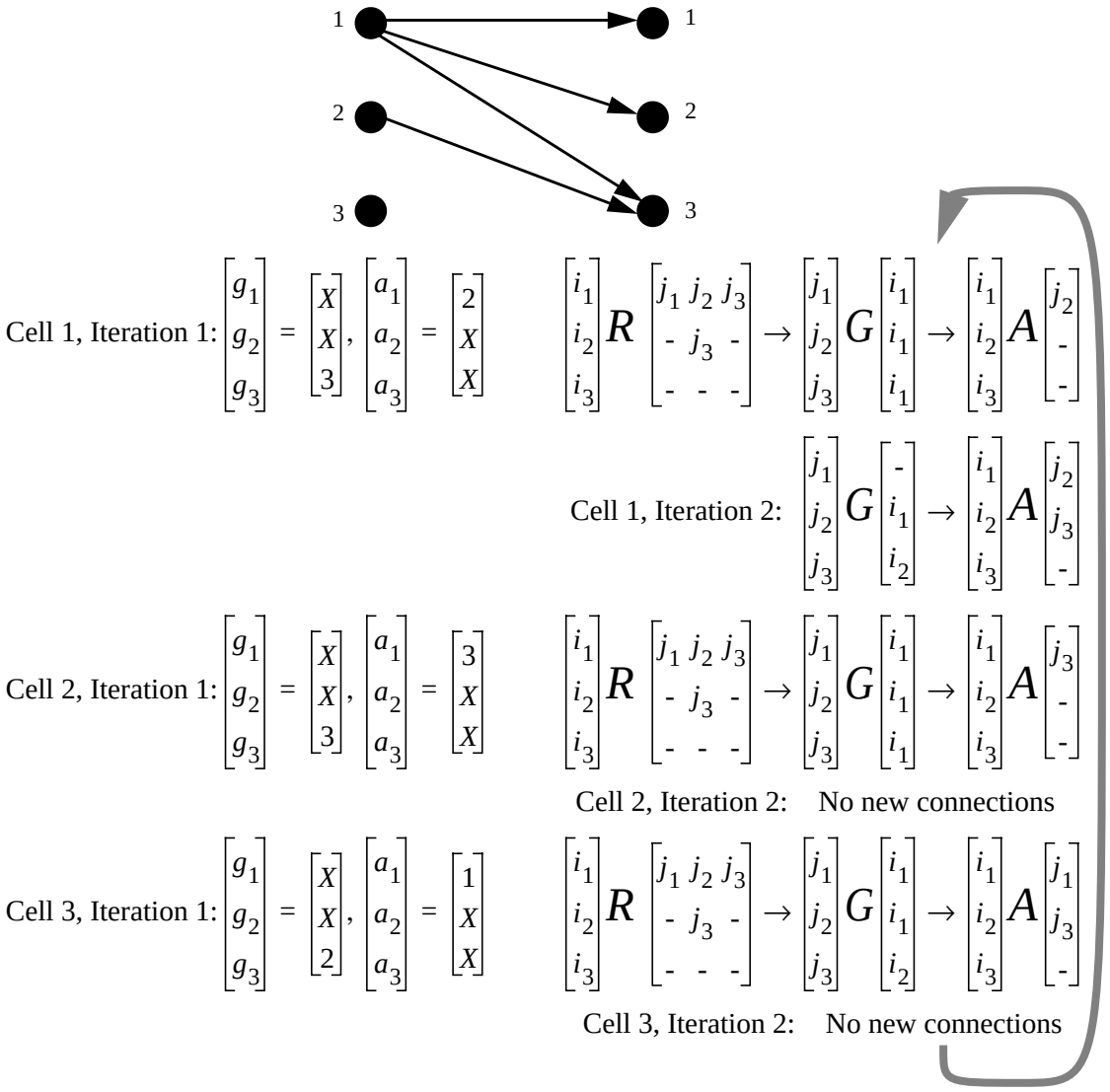


FIGURE 3.2 Example of unequal service under heavy load for 2 inputs that share an output.The 3x3 switch is heavily loaded, i.e. all four active connections have an offered load of 1 cell per cell time. The sequence of grants and accepts for 2-SLIP repeats after 3 cell times. During each cycle, Q(2,3) is served twice whereas Q(1,3) is served only once. Note that for 1-SLIP, only the 1st cell time would be different and Q(2,3) would be served only once per cycle.

3 Simulated Performance of Iterative SLIP

3.1 How Many Iterations?

Now that we have an iterative algorithm, we need to decide how many iterations to perform

during each cell time. Ideally, from Property 5 above we would like to perform N iterations. However, in practice there may be insufficient time for N iterations, and so we need to consider the penalty of performing only i iterations, where $i < \mathrm{N}$. In fact, because of the desynchronization of the arbiters, i-SLIP will usually converge in fewer than N iterations. An interesting example of this is shown in Figure 3.3. In the first cell time, the algorithm takes N iterations to converge, but thereafter converges in one less iteration each cell time. After N cell times, the arbiters have become totally desynchronized and the algorithm will converge in a single iteration.

How many iterations should we use: it clearly doesn't always take N? One option is to always run the algorithm to completion, resulting in a scheduling time that varies from cell to cell. In some applications this may be acceptable. In others, such as in an ATM switch, it is desirable to maintain a fixed scheduling time and to try and fit as many iterations into that time as possible.

Under simulation, we have found that for an NxN switch it takes *about* $\log_2 N$ iterations for i-SLIP to converge. This is similar to the results obtained for PIM in [3], in which the authors prove that

$$\mathrm{E}(I) \leq \log_2 N + \frac{4}{3}, \tag{1}$$

where I is the number of iterations that PIM takes to converge. However, although for all the stationary arrival processes we have considered $\mathrm{E}(I) < \log_2 N$ for i-SLIP, we have not been able to prove that this relation holds in general.

As an example, Figure 3.4 compares the number of iterations required for PIM and i-SLIP to converge under uniform i.i.d. Bernoulli arrivals.

3.2 Bernoulli Traffic

To illustrate the improvement in performance of i-SLIP when the number of iterations is increased, Figure 3.5 shows the average queueing delay for 1, 2 and 4 iterations under uniform i.i.d. Bernoulli arrivals. We find that multiple iterations of i-SLIP significantly increase the size of the match and therefore reduce the queueing delay[1]. In fact, n-SLIP is stable for all n under admis-

1. Although not shown, we find that increasing i above 4 for a 16x16 switch leads to a negligible performance improvement.

Cell 1, Iteration 1:

$$\begin{bmatrix} g_1 \\ g_2 \\ : \\ g_N \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ : \\ 1 \end{bmatrix}, \begin{bmatrix} a_1 \\ a_2 \\ : \\ a_N \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ : \\ 1 \end{bmatrix} \qquad \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} R \begin{bmatrix} j_1\ j_2\ \cdots\ j_N \\ j_1\ j_2\ \cdots\ j_N \\ :\ :\ \ldots\ : \\ j_1\ j_2\ \cdots\ j_N \end{bmatrix} \rightarrow \begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix} G \begin{bmatrix} i_1 \\ i_1 \\ : \\ i_1 \end{bmatrix} \rightarrow \ i_1 \, A \, j_1$$

Iteration N:

$$\begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix} G \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} \rightarrow \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} A \begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix}$$

Cell 2, Iteration 1:

$$\begin{bmatrix} g_1 \\ g_2 \\ : \\ g_N \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ : \\ 1 \end{bmatrix}, \begin{bmatrix} a_1 \\ a_2 \\ : \\ a_N \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ : \\ 1 \end{bmatrix} \qquad \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} R \begin{bmatrix} j_1\ j_2\ \cdots\ j_N \\ j_1\ j_2\ \cdots\ j_N \\ :\ :\ \ldots\ : \\ j_1\ j_2\ \cdots\ j_N \end{bmatrix} \rightarrow \begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix} G \begin{bmatrix} i_1 \\ i_1 \\ : \\ i_1 \end{bmatrix} \rightarrow \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} A \begin{bmatrix} j_2 \\ j_1 \end{bmatrix}$$

Iteration N-1:

$$\begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix} G \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} \rightarrow \begin{bmatrix} i_1 \\ i_2 \\ i_3 \\ : \\ i_N \end{bmatrix} A \begin{bmatrix} j_2 \\ j_1 \\ j_3 \\ : \\ j_N \end{bmatrix}$$

:

:

Cell N, Iteration 1:

$$\begin{bmatrix} g_1 \\ g_2 \\ : \\ g_N \end{bmatrix} = \begin{bmatrix} N \\ N-1 \\ : \\ 1 \end{bmatrix}, \begin{bmatrix} a_1 \\ a_2 \\ : \\ a_N \end{bmatrix} = \begin{bmatrix} N \\ N-1 \\ : \\ 1 \end{bmatrix} \qquad \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} R \begin{bmatrix} j_1\ j_2\ \cdots\ j_N \\ j_1\ j_2\ \cdots\ j_N \\ :\ :\ \ldots\ : \\ j_1\ j_2\ \cdots\ j_N \end{bmatrix} \rightarrow \begin{bmatrix} j_1 \\ j_2 \\ : \\ j_N \end{bmatrix} G \begin{bmatrix} i_N \\ i_{N-1} \\ : \\ i_1 \end{bmatrix} \rightarrow \begin{bmatrix} i_1 \\ i_2 \\ : \\ i_N \end{bmatrix} A \begin{bmatrix} j_N \\ j_{N-1} \\ : \\ j_1 \end{bmatrix}$$

FIGURE 3.3 Example of the number of iterations required to converge for a heavily loaded NxN switch. All input queues remain non-empty for the duration of the example. In the first cell time, the arbiters are all synchronized. During each cell time, one more arbiter is desynchronized from the others. After N cell times, all arbiters are desynchronized and a maximum sized match is found in a single iteration.

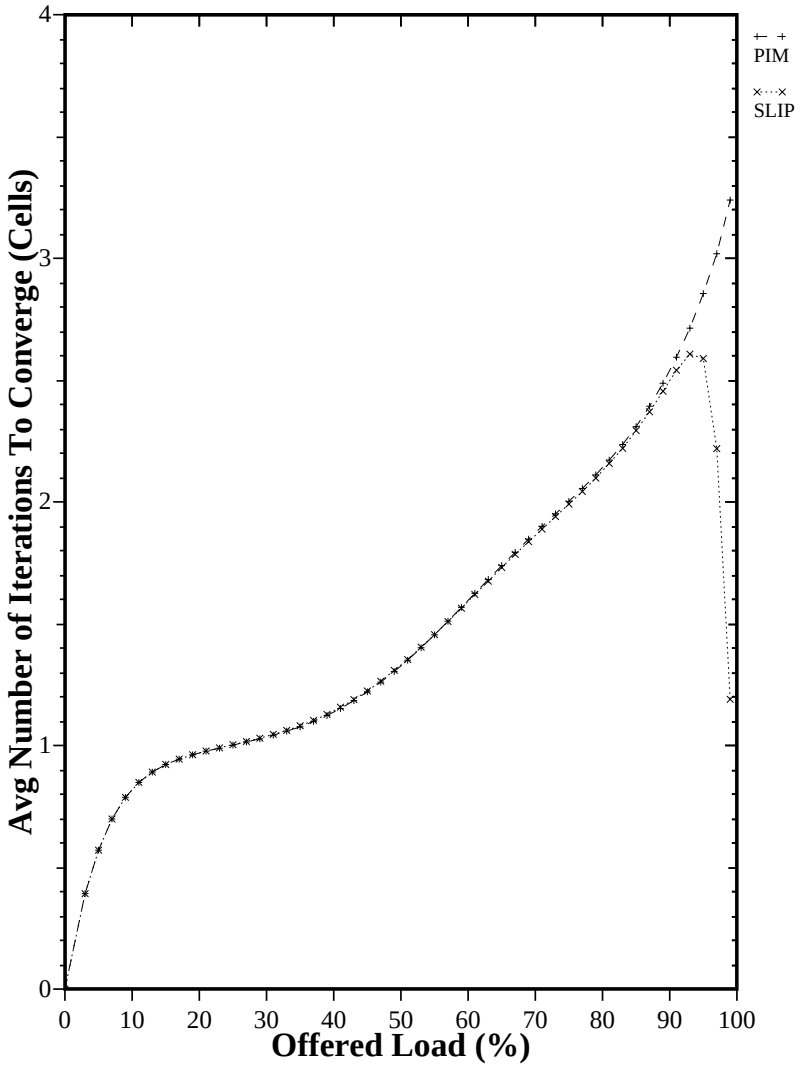


FIGURE 3.4 An example of the number of iterations for *i*-SLIP and PIM to converge for uniform i.i.d. Bernoulli traffic as a function of the offered load for a 16x16 switch. Each algorithm is run to completion during each cell time to determine how many iterations are required before no more connections can be added.

sible uniform i.i.d. Bernoulli arrivals. This should come as no surprise: in Chapter 2 we saw that 1-SLIP is stable under these conditions. Intuitively, the size of the match increases with the number of iterations: each new iteration potentially adds connections not made by earlier iterations. As a result, for a given set of queue occupancies *(n+1)*-SLIP can provide an instantaneous match closer to the maximum sized match than *n*-SLIP. This is illustrated in Figure 3.6 which compares the size of each *i*-SLIP matching with the size of the maximum matching for the same instantaneous queue occupancies. Under low offered load, the 1-SLIP arbiters move randomly and the ratio of the

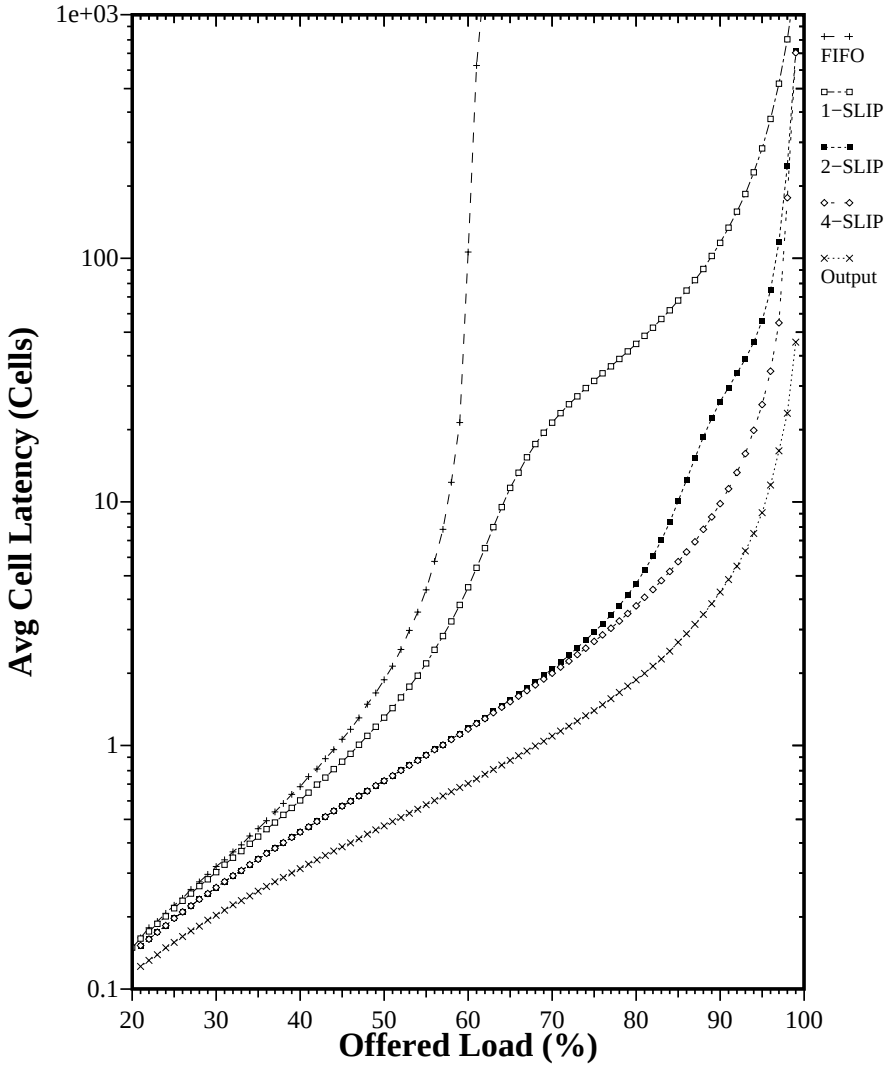


FIGURE 3.5 Performance of *i*-SLIP for 1,2 and 4 iterations compared with FIFO and output queueing for i.i.d Bernoulli arrivals with destinations uniformly distributed over all outputs. Results obtained using simulation for a 16x16 switch. The graph shows the average delay per cell, measured in cell times, between arriving at the input buffers and departing from the switch.

match size to the maximum match size decreases with increased offered load. But when the load exceeds approximately 65%, the ratio begins to increase linearly. This is the result of desynchronization of the output arbiters which leads to a better and better match as the load increases. 2-SLIP and 4-SLIP behave similarly and, as expected, the ratio increases with the number of iterations indicating that the matching gets closer to the maximum sized match. But only up to a point: for an 16x16 switch under this traffic load, increasing the number of iterations beyond four does not measurably increase the average match size.

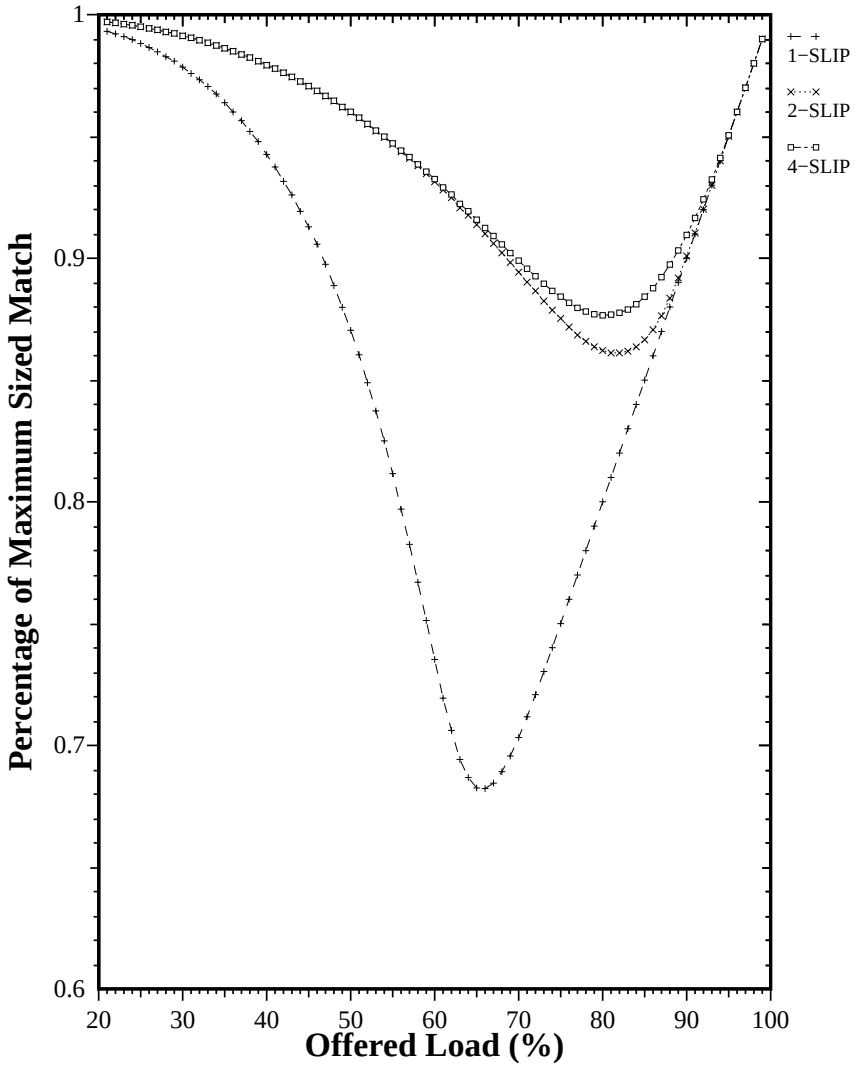


FIGURE 3.6 Comparison of the match size for *i*-SLIP with the size of a maximum sized match for the same set of requests. Results are for a 16x16 switch and uniform i.i.d. Bernoulli arrivals.

Although the average queueing delay is reduced by increasing the number of iterations, the synchronization of the output arbiters *increases*, as shown in Figure 3.7. This unfortunate behavior is the consequence of only allowing the arbiters to be updated after the first iteration. With 1-SLIP, every successful connection moves an arbiter's pointer, leading to a rapid desynchronization under high offered load. For 2-SLIP and 4-SLIP, only those connections made by the first iteration move the arbiter's pointer, leading to a slower rate of desynchronization. Later in this chapter we will consider ways that *i*-SLIP can be modified to reduce the arbiter synchronization.

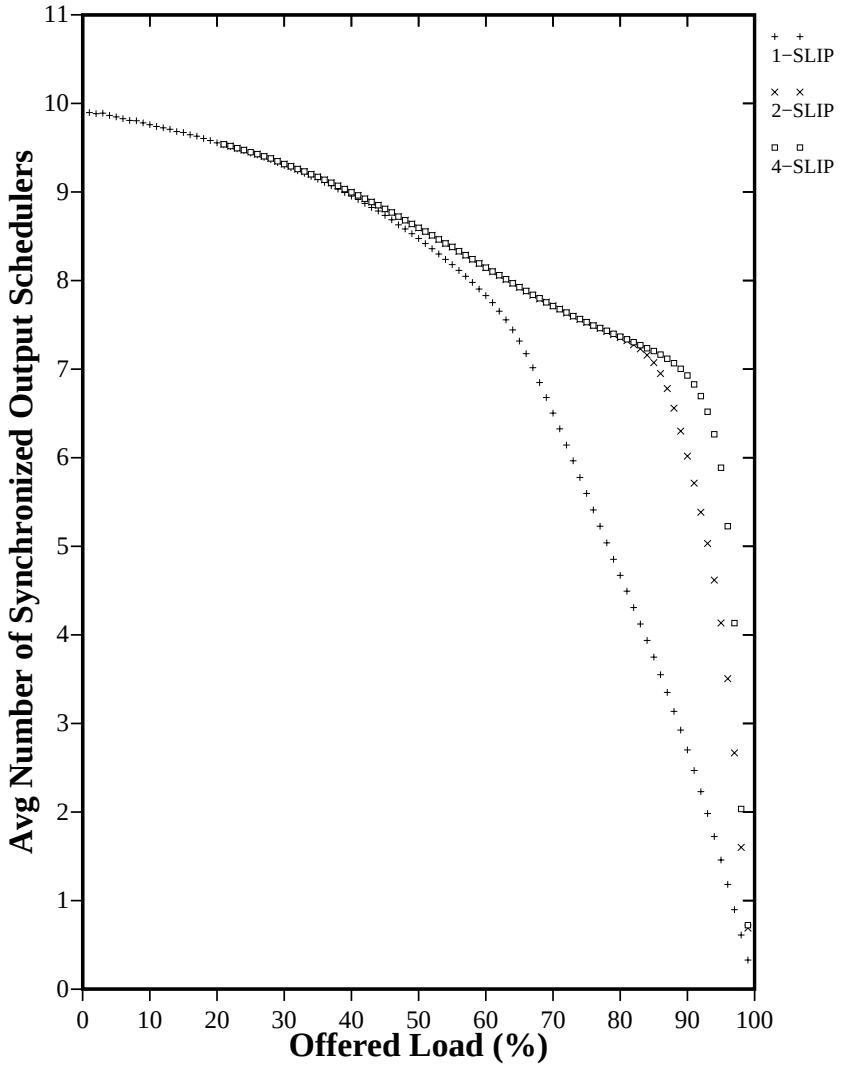


FIGURE 3.7 Average number of synchronized output schedulers as a function of offered load for uniform i.i.d. Bernoulli arrivals. Scheduling is with 1-SLIP, 2-SLIP and 4-SLIP.

3.3 Bursty Traffic

As we did for 1-SLIP, we illustrate the effect of burstiness on *i*-SLIP using an on-off arrival process modulated by a 2-state Markov-chain. The source alternately produces a burst of full cells (all with the same destination) followed by an idle period of empty cells. The bursts and idle periods contain a geometrically distributed number of cells.

Figure 3.8 shows the performance of *i*-SLIP under this arrival process for a 16x16 switch, comparing the performance for 1, 2 and 4 iterations. As we would expect, the increased burst size

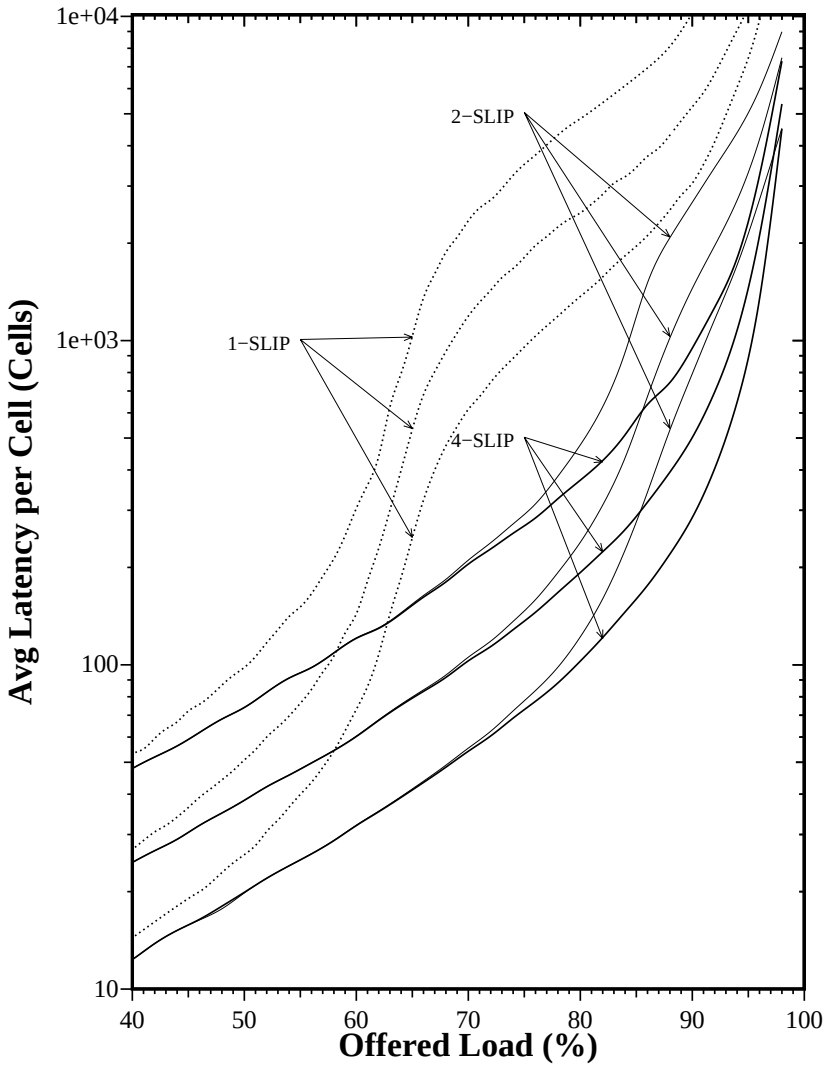


FIGURE 3.8 Performance of *i*-SLIP for 1, 2 and 4 iterations under bursty arrivals. Arrival process is a 2-state Markov-modulated on-off process. Average burst lengths are 16, 32 and 64 cells.

leads to a higher queueing delay whereas an increased number of iterations leads to a lower queueing delay. In all three cases, the average latency is *proportional* to the expected burst length. As pointed out in Chapter 2, the performance for bursty traffic is not heavily influenced by the queueing policy.

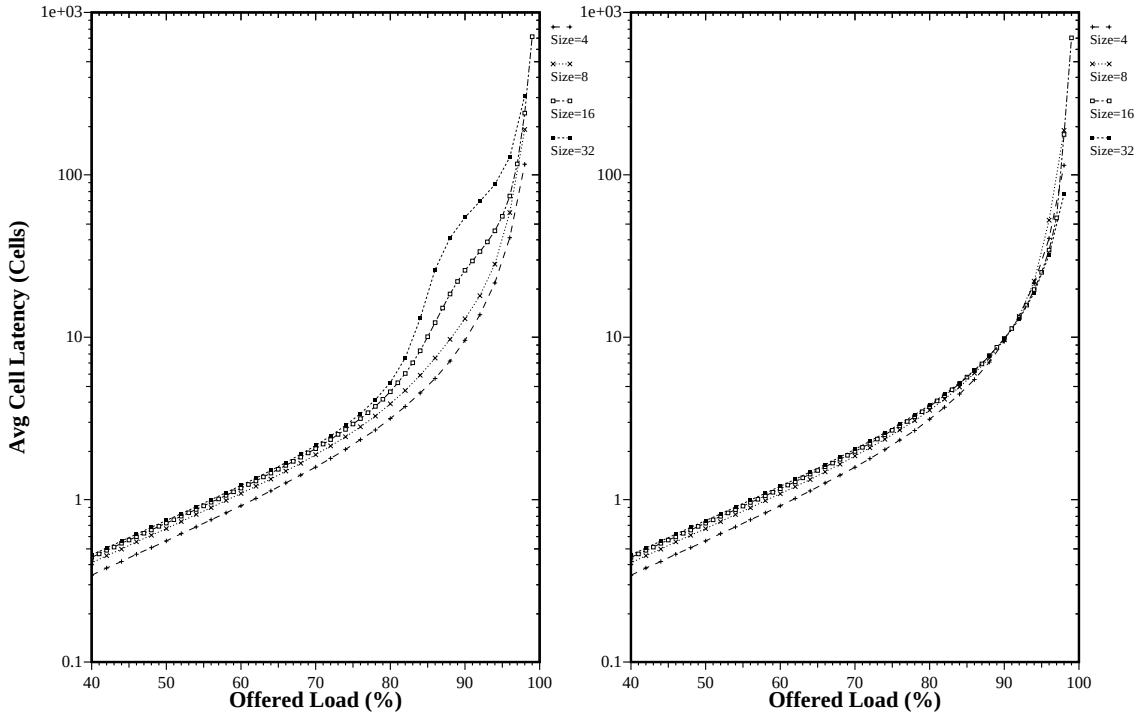


FIGURE 3.9 The performance of *i*-SLIP as function of switch size. Uniform i.i.d. Bernoulli arrivals.

3.4 As a Function of Switch Size

In Chapter 2 we found that 1-SLIP performance degrades as the switch size increases. As shown in Figure 3.9(a), we find similar behavior for 2-SLIP. Under low offered load the average cell latency approaches a constant, whereas under high load the delay is approximately proportional to N.

Although similar under low offered load, 4-SLIP exhibits quite different behavior under high offered load: the average latency can actually *decrease* with N. This is shown in Figure 3.9(b). Under an offered load below approximately 80% the ordering is strict: increasing *N* increases average latency. Between 80% and 100% offered load, the ordering changes — at 99% offered load the ordering has reversed and the average latency decreases strictly with *N*.

It should be noted that the *maxsize* algorithm does not exhibit the same behavior. The average latency for the *maxsize* algorithm always increases with *N*.

Because of the difficulty of analyzing this algorithm with more than a single iteration, we offer only a heuristic explanation for this result. We believe the result to be the combination of two effects. First, for switches up to a size 64x64, the algorithm almost always converges in fewer than 4 iterations. This means that in 4 iterations, the match size is close to the *maximal* match: the largest match possible without rearranging connections. Second, the number of possible matches under heavy load equals approximately *N*! which grows rapidly with *N*. Moreover, as *N* increases it becomes more likely that a *maximal* match equals a maximum match. Hence, with sufficient iterations and as *N* increases, it becomes more likely that the SLIP algorithm will find a match close or equal to a maximum sized match. By comparing the size of the match with the maximum sized match, we have found this to be the case.

It should be noted that the PIM algorithm exhibits similar behavior. This supports our explanation: PIM like *i*-SLIP, converges on a *maximal* match.

4 Variations of Iterative SLIP

In Chapter 2 we found that 1-SLIP performs well because of the desynchronization of the output arbiters under high offered load. As shown in Figure 3.7 *i*-SLIP is less effective at desynchronizing its arbiters, because the pointers can only be updated for connections made by the first iteration.

In this section we consider two variations on *i*-SLIP that allow the pointers to be updated after every iteration. Both variations are significantly more complex to implement than basic *i*-SLIP and the performance improvements are inconclusive. We believe that these algorithms require further study.

4.1 Iterative SLIP with LRU Accept Arbiters

Figure 3.1 showed how starvation could occur if we allowed the pointers to be updated after every iteration. In the example, the problem was not that output 2 never grants to input 1. Rather,

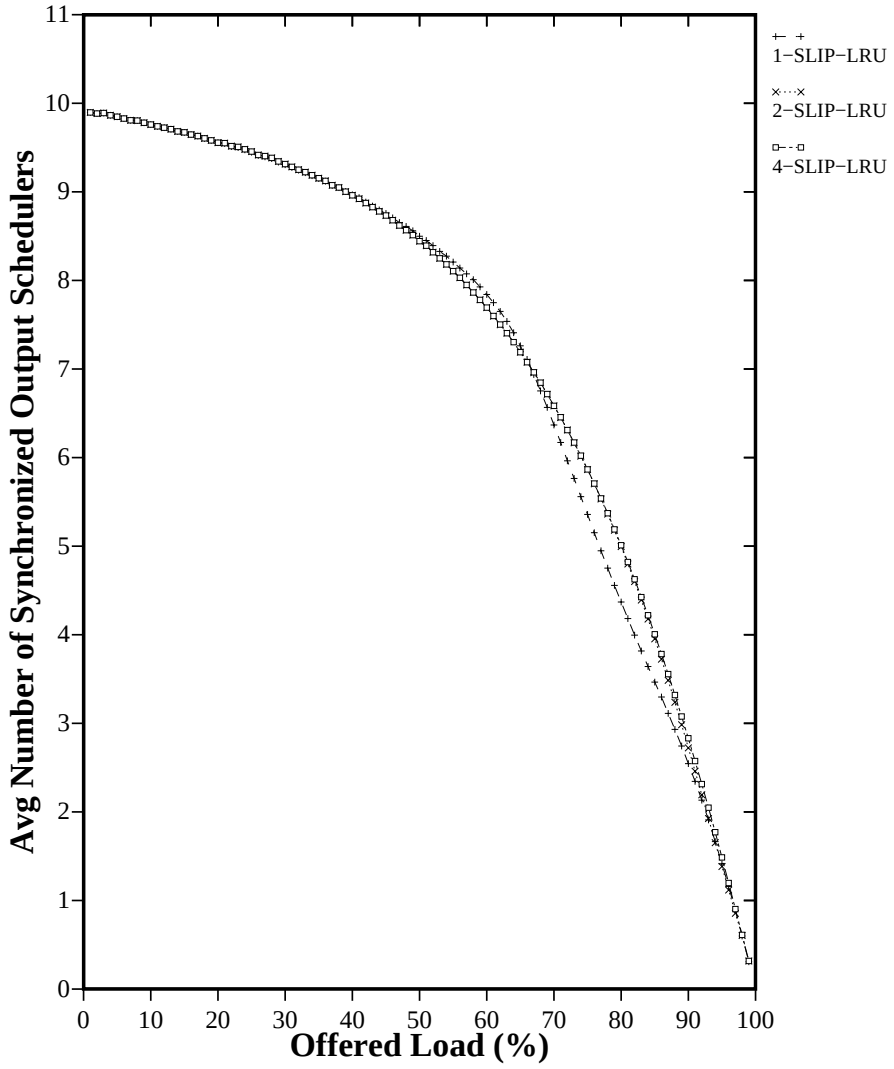


FIGURE 3.10 *i*-SLIP-LRU algorithm under uniform i.i.d. Bernoulli traffic. For 1 iteration, the number of desynchronized schedulers is almost identical to 1-SLIP. However, the number of schedulers increases only slightly with the number of iterations. This is quite different from i-SLIP for 2 and 4 iterations (see Figure 3.7)

when input 1 received a grant, it never accepted it. If we can encourage input 1 to accept output 2 if it has not done so recently, then we can prevent the connection from being starved. One way to achieve this is for the input arbiter to give highest priority to the least recently used (LRU) output[1]. We call this algorithm “iterative SLIP output arbiters with LRU input arbiters” (*i*-SLIP-LRU).

The *i*-SLIP-LRU algorithm allows us to update the pointers after each iteration so that every established connection will help desynchronize the output arbiters. Figure 3.10 demonstrates that

1. Starvation could also be avoided by using random selection at the input arbiter.

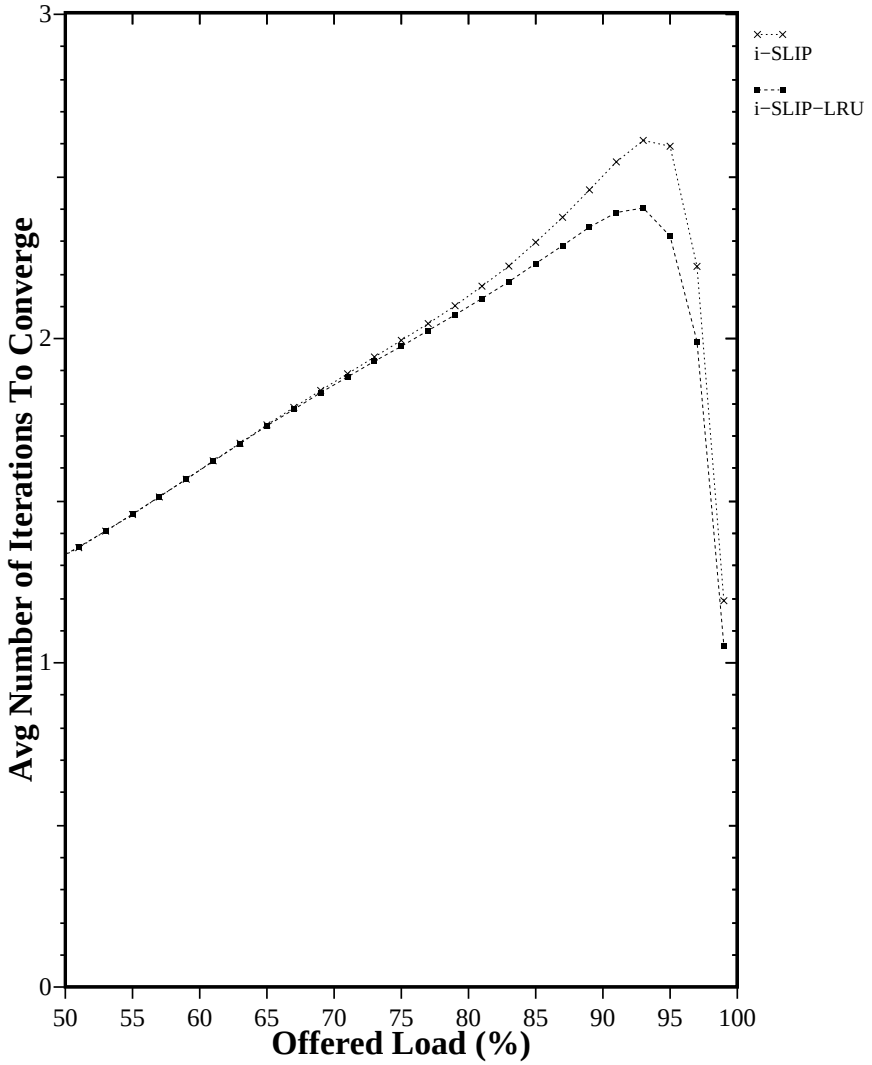


FIGURE 3.11 An example of the average number of iterations for *i*-SLIP-LRU to converge for uniform i.i.d. Bernoulli traffic as a function of the offered load. The algorithm is run to completion during each cell time to determine how many iterations are performed before no more connections can be added.

unlike *i*-SLIP, the number of synchronized schedulers for *i*-SLIP-LRU increases only very slightly with the number of iterations. This is beneficial in two ways: it means that for a fixed number of iterations, *i*-SLIP-LRU will provide higher performance, or alternatively, if run to completion *i*-SLIP-LRU will converge faster (Figure 3.11).

4.2 Separate Pointers for each Iteration

An alternative way to prevent starvation in iterative SLIP is to maintain a separate pointer for each iteration. For *n*-SLIP, output *j* now maintains *n* grant pointers, $g_j(1)$, …, $g_j(n)$. During the first iteration, the arbiter uses $g_j(1)$, updating the pointer if and only if a connection is established in this iteration. In the next iteration, the arbiter uses $g_j(2)$, and so on. In other words, the arbiter pointer is updated at the end of every iteration and for every connection.

We now show that no queue is starved of service by this algorithm. Assume that the queue Q(*i*,*j*) at input *i* is non-empty and thus requests service from output *j*. Now assume that we are in iteration *k*. Output *j* is using the pointer $g_j(k)$, and we shall assume that currently $g_j(k) = i$. During iteration *k* of every cell time, output *j* will grant to input *i* until either (i) input *i* accepts output *j* and Q(*i*,*j*) is served, or (ii) output *j* serves Q(*i*,*j*) in iteration $k' \neq k$, and Q(*i*,*j*) becomes empty. If Q(*i*,*j*) does not become empty in iteration k', then eventually it will be served in iteration *k* and $g_j(k)$ will be updated. So, in either case, Q(*i*,*j*) is served and $g_j(k)$ will eventually move. This means every non-empty queue will eventually be served and none will be starved of service indefinitely.

The advantage of this algorithm is that the pointers at each iteration tend to become desynchronized. More importantly, every connection that is established helps desynchronize the arbiters.

We find that for the arrival processes described in Section 3, this variation on the basic SLIP algorithm does *not* improve performance. It therefore does not seem worth the extra complexity of maintaining multiple pointers at each output. We believe that this requires further study.

5 Implementing Iterative SLIP

To conclude the description of *i*-SLIP, we consider the complexity of implementing the algorithm in hardware. Implementation of *i*-SLIP is very similar to non-iterative SLIP, described in Chapter 2 Section 7. Figure 3.12 shows how for an NxN switch, 2N arbiters and an N^2-bit memory

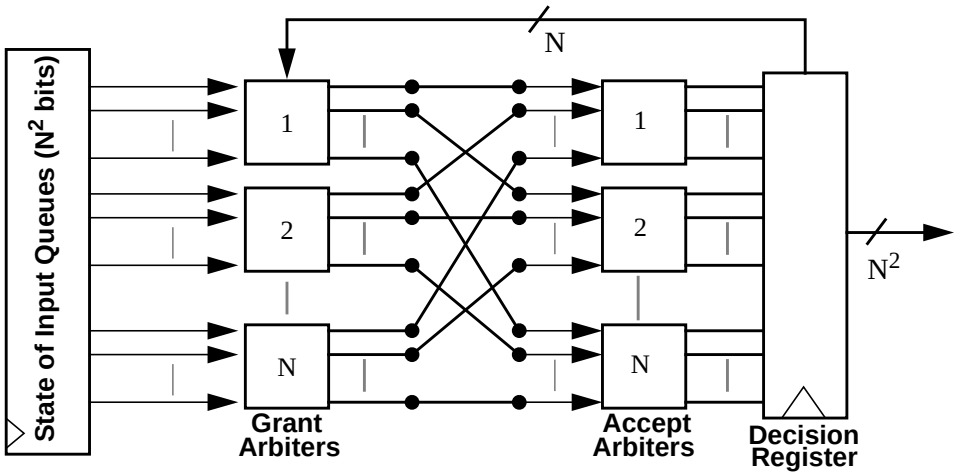


FIGURE 3.12 Interconnection of 2N arbiters to implement *i*-SLIP for an NxN switch.

may be interconnected to implement *i*-SLIP. The arbiters are almost identical to those used for non-iterative SLIP. They differ in two ways:

1. When an input or output is matched, its arbiter is disabled in subsequent iterations, preventing it from making additional matches. This is simple for the accept arbiter: whenever it makes a decision, it is disabled in all further iterations. However, it is not known whether a grant arbiter's decision is accepted until the end of the iteration. The decision register must feedback an indication to the grant arbiters. This is shown in Figure 3.12.
2. The arbiters only update pointers a_i and g_i after the first iteration.

As before, the complexity of *i*-SLIP is dominated by the arbiters. In fact, the number of gates required to implement *i*-SLIP is almost identical to non-iterative SLIP.

In some implementations it may be desirable to reduce the number of arbiters, sharing them among the grant and accept steps of the algorithm. An implementation using only N arbiters is shown in Figure 3.13. The results from the arbiters in the grant phase are registered and fed back for the accept phase. The number of gates for this implementation is almost halved, but with the performance penalty of an extra clock delay through the holding register.

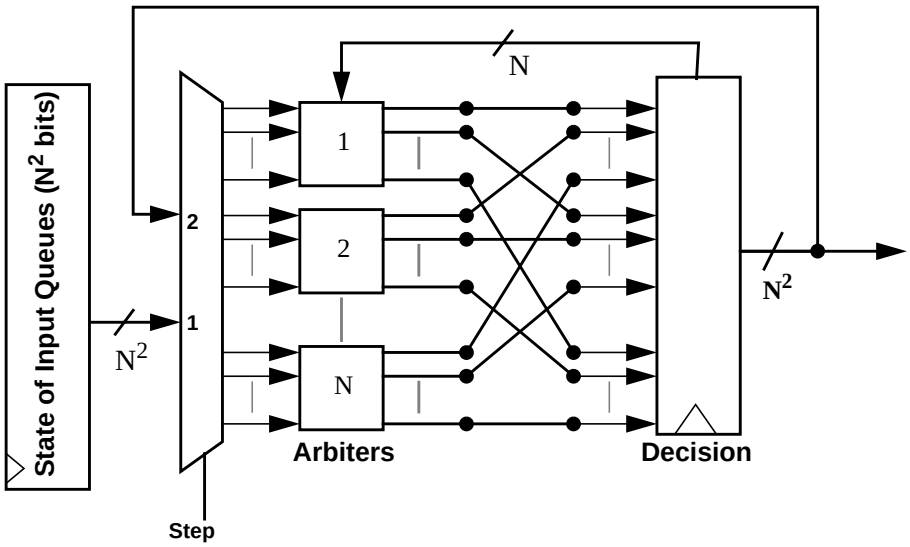


FIGURE 3.13 Interconnection of N arbiters to implement i-SLIP for an NxN switch. Each arbiter is used for both input and output arbitration. In this case, each arbiter contains *two* registers to hold pointers g_i and a_i.

CHAPTER 4

Weighted Matching Algorithms

1 Introduction

In this chapter we describe algorithms that consider more than one bit of information per queue, for example the occupancy of the queue, or the waiting time of queued cells. These algorithms find the maximum or maximal *weight* matching, giving preference to queues with a larger occupancy, or to cells that have been waiting longest.

We saw in Chapter 1 that maximizing the size of the match is not necessarily desirable as this can lead to instability for an offered load below capacity, and can lead to starvation for an offered load above capacity. This was demonstrated in Chapter 2 where we considered simple arrival patterns that are unstable for both the *maxsize* and SLIP algorithms, even for a 2x2 switch. The reason that these algorithms become unstable is that they only consider one bit of information per input queue: whether the queue is empty or non-empty. As we shall see, the maximum weight algorithms are stable over a wider range of workloads.

We start by describing two maximum weight matching algorithms, *longest queue first* (LQF) and *oldest cell first* (OCF) and consider their performance. We prove that the LQF algorithm is stable under i.i.d. arrivals and conjecture that both algorithms, although too complex to implement in hardware, are stable under all admissible offered loads.

We describe the more practical, parallel and iterative algorithms, *i*-LQF and *i*-OCF which attempt to find maximal weight matchings in a similar manner to *i*-SLIP and present schematic implementations of both algorithms.

Finally, we describe an interesting class of algorithms that solve the *stable marriage problem*. Solutions to this well-studied problem find a special kind of weighted bipartite matching called a *stable* matching. Although stable matchings are generally different from either a maximum sized or maximum weight match, they provide good performance and are readily implemented in hardware.

2 Maximum Weight Matching

Figure 1.2 shows an example of a matching on a weighted bipartite graph. The maximum *weight* matching, M is one that maximizes $\sum_{(i,j) \in M} w_{i,j}$, where $w_{i,j}$ is the weight assigned to the edge between vertices i and j. As with the maximum size matching, the maximum weight matching for a bipartite graph can be found by solving an equivalent network flow problem. The most efficient known algorithm for solving this problem converges in $O(N^2 \log_3 N)$ running time [41].

We now consider two types of maximum weight matching that may be used to schedule cells in an input-queued switch: LQF and OCF.

In the LQF algorithm, preferential service is given to input queues that are more heavily occupied. As illustrated in Figure 4.1, this is achieved by defining $w_{i,j}(t)$ to be equal to the queue occupancy $L_{i,j}(t)$.

The OCF algorithm gives preferential service to cells that have been queued for a long time. This is achieved by defining $w_{i,j}(t)$ to be equal to the waiting time $W_{i,j}(t)$ of the cell at the head of queue $Q(i,j)$.

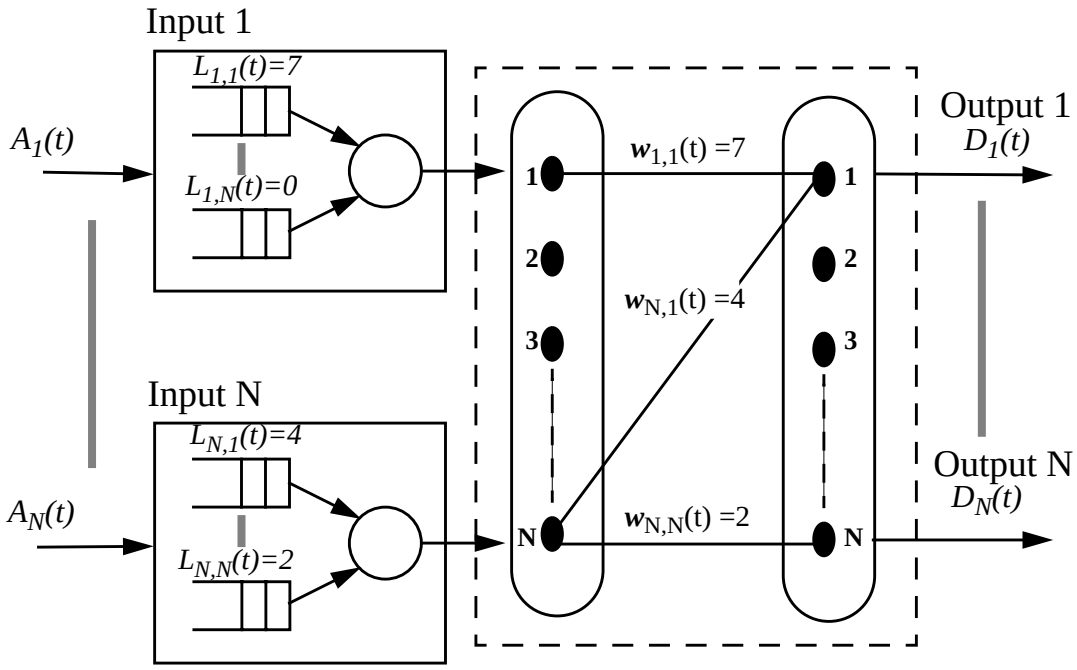


FIGURE 4.1 Example of weights for the LQF maximum weight matching algorithm.

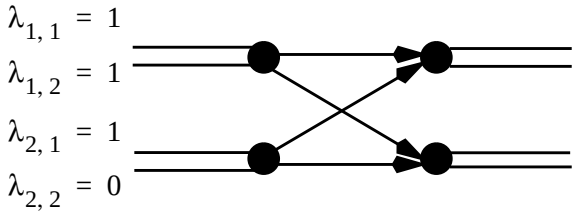


FIGURE 4.2 Example of 2x2 switch for which, using the LQF algorithm, inadmissible traffic may lead to the starvation of an input queue.

2.1 Starvation with LQF

Under inadmissible traffic it is possible for the LQF algorithm to permanently starve an input queue. As a simple example, consider the 2x2 switch shown in Figure 4.2. Three queues have an arrival rate equal to their capacity and will be unstable, growing without bound. Now assume that

all three queues have grown to a length of two cells. Further assume that a single cell arrives at the fourth queue, *Q(2,2)*, but that no further cells arrive at this queue. Because of the large arrival rate, the other queues will never contain fewer than two cells and so *Q(2,2)* will never be served.

OCF, however, cannot starve a queue under any offered load. Cells at the head of queues that have not been served recently increase in weight until, eventually, they are served.

2.2 Performance of LQF and OCF Algorithms

2.2.1 Uniform Workload

If all switch inputs and outputs are identically loaded, the LQF algorithm has an *average* performance identical to the *maxsize* algorithm, Figure 4.3(a). This is because there is no benefit, on average, in distinguishing between different input queues when they all have identical average arrival rates. We have found this to be the case for a range of uniform workloads with and without correlated arrivals. The OCF algorithm has a slightly worse average behavior than LQF.

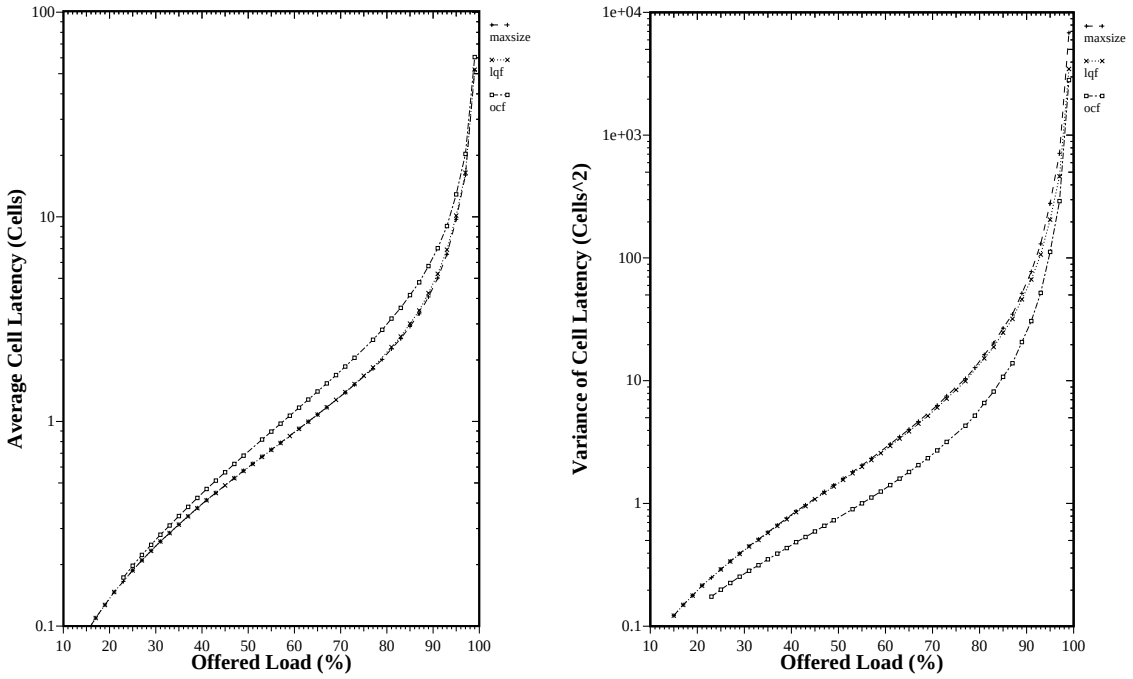
However, because during fluctuations in waiting time it favors cells that have been waiting longest, the *variance* in cell latency is lower for the OCF algorithm. This is illustrated in Figure 4.3(b), where the variance of cell latency is plotted against the offered load.

2.2.2 Non-Uniform Workload

The difference between the maximum weight and maximum size matching algorithms is more marked under a non-uniform workload, particularly when not all flows are active.

We illustrate this by way of a simple example: an arbitrary arrival pattern for a 4x4 switch, shown in Figure 4.4. We find that for the *maxsize* algorithm the switch is unstable for $\lambda > 0.31$, whereas for the LQF and OCF algorithms, the switch is stable for all admissible values of λ.

The LQF and OCF algorithms maintain much closer average queue lengths than the *maxsize* algorithm. Figure 4.5 shows the average cell latency through each of the 16 input queues in the 4x4 switch of Figure 4.4, with all average rates $\lambda = 0.30$. Even though the *maxsize* algorithm is stable for this workload, the average queue lengths differ widely. This difference increases as the



(a) Average cell latency vs. offered load. (b) Variance in cell latency vs. offered load.

FIGURE 4.3 16x16 switch scheduled using the LQF, OCF and *maxsize* scheduling algorithms.

All flows have identical average arrival rate $\lambda = \lambda_{i,j}$ such that $\sum_{i \in I} \lambda_{i,j} < 1, \sum_{j \in J} \lambda_{i,j} < 1$

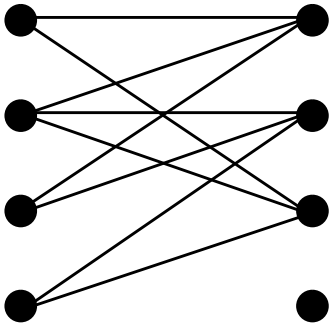


FIGURE 4.4 Example of 4x4 switch with non-uniform traffic pattern. Although admissible, this traffic pattern can be unstable for the maximum scheduling algorithm. It is stable for the LQF and OCF algorithms.

offered load increases, and it is the three queues $Q(2, 1)$, $Q(2, 2)$, $Q(2, 3)$ that become unstable

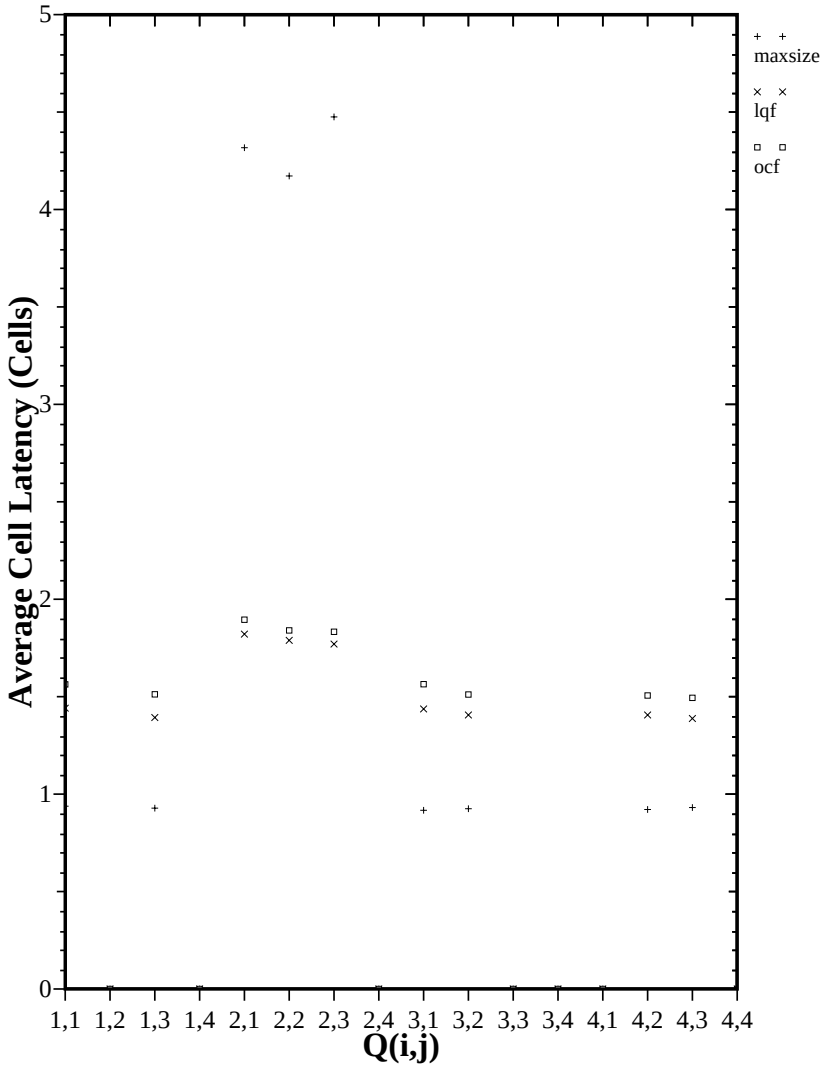


FIGURE 4.5 Variation in queue lengths under the non-uniform workload shown in Figure 4.4, with i.i.d. Bernoulli arrivals and $\lambda = 0.30$.

when the average arrival rate exceeds $\lambda = 0.31$. Both LQF and OCF maintain average queue lengths that are almost identical independent of the arrival rate.

2.2.3 Stability of 2x2 Switch

In this section we consider the stability of a 2x2 switch, scheduled with a maximum weight matching algorithm.

We start with a simple observation: if an arrival pattern is such that all input queues are permanently occupied, the switch will be stable. This is because the switch can serve two queues in every cell time.

Theorem 4.1: *If* $L_{1,1}(n), L_{1,2}(n), L_{2,1}(n), L_{2,2}(n) > 0$ *for all n, LQF and OCF are stable for a 2x2 switch.*

Proof: Assume that at time *n* the switch is stable. i.e.

$$L(n) \equiv L_{1,1}(n) + L_{1,2}(n) + L_{2,1}(n) + L_{2,2}(n) < \infty \quad (1)$$

Because $L_{1,1}(n), L_{1,2}(n), L_{2,1}(n), L_{2,2}(n) > 0$, the LQF and OCF algorithms will always serve exactly 2 queues. There can be no more than two arrivals to the switch in a cell time, so

$$L(n+1) \leq L(n) \quad (2)$$

and so the switch must be stable at time *n*+1. Hence $L(n)$ cannot become unbounded, and the switch is stable. ❒

In general, not all input queues will be permanently occupied. At any time, one or more queues may be empty and, as a result, the scheduling algorithm may select to serve fewer than two queues. Recall that it was when one queue was empty that the maximum size matching algorithm (and SLIP and PIM) could become unstable. However, below we show that for a 2x2 switch LQF and OCF are stable when one or more queues are permanently empty.

Theorem 4.2: *LQF and OCF are stable for the 2x2 switch with three active flows illustrated in Figure 4.6 and independent arrivals.*

Proof: Appendix 3 Section 1 finds sufficient conditions on a scheduling algorithm so that this switch is stable for independent arrivals:

1. If $L_{1,1}(n) = 0$, then set crossbar to configuration *B*.
2. Else, if $L_{2,1}(n) = 0$ and/or $L_{1,2}(n) = 0$, then set crossbar to configuration *A*.
3. Else, set crossbar configuration to either *A* or *B*.

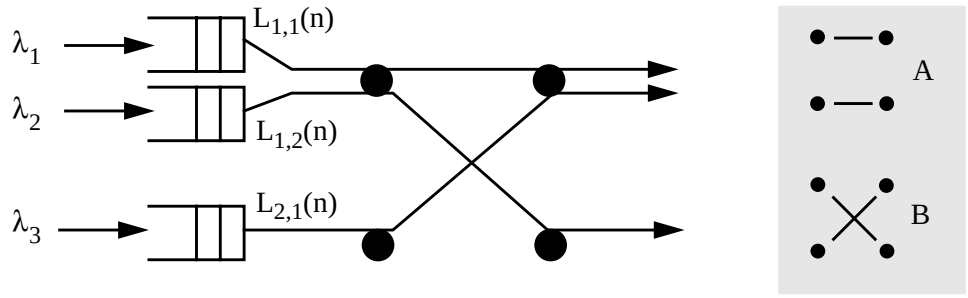


FIGURE 4.6 2x2 switch with three active flows. The two possible switch configurations, *A* and *B*, are shown.

LQF clearly satisfies conditions (1) and (3) above, but not (2). Define an algorithm, *G*, that is exactly the same as LQF under conditions (1) and (3), but also satisfies condition (2). Clearly, *G* is stable. LQF serves $Q(1, 2)$ and $Q(2, 1)$ at least as often as *G*, so under LQF these two queues must be stable. It remains to be shown that $Q(1, 1)$ is stable under LQF. Assume that under LQF, $Q(1, 1)$ is *unstable*. For this to be the case $Q(1, 1)$ must grow so that at some time

$$L_{1,1}(n) = 1 + L_{1,2}(n) + L_{2,1}(n), \tag{3}$$

and $Q(1, 1)$ will then be served continuously until

$$L_{1,1}(n) \le L_{1,2}(n) + L_{2,1}(n). \tag{4}$$

During the time that $Q(1, 1)$ is served continuously, its queue length cannot increase. Therefore, $L_{1,1}(n)$ is bounded by $1 + L_{1,2}(n) + L_{2,1}(n)$. $Q(1, 2)$ and $Q(2, 1)$ are stable, so $1 + L_{1,2}(n) + L_{2,1}(n)$ is bounded, which means that $Q(1, 1)$ is stable. An analogous argument based on waiting times holds for OCF. ❒

It is interesting to consider why *maxsize* can be unstable for this switch (Chapter 1, Section 3.3). Although the conditions above are not strictly necessary for the switch to be stable, they give an intuitive explanation. If condition (2) above is true, the *maxsize* algorithm may select either configuration *A* or *B*, even if one of the queues grows without bound.

Theorem 4.3: *For any arrival process to a 2x2 switch, if for some n*

$$\left| \{L_{1,1}(n) + L_{2,2}(n)\} - \{L_{1,2}(n) + L_{2,1}(n)\} \right| \leq 3 \tag{5}$$

then for all $n' \geq n$

$$\left| \{L_{1,1}(n') + L_{2,2}(n')\} - \{L_{1,2}(n') + L_{2,1}(n')\} \right| \leq 4\,. \tag{6}$$

Proof: See Appendix 3 Section 2. ❒

To summarize these results for a 2x2 switch:

1. For any arrival process, it is not possible for all 4 input queues to become unstable simultaneously.
2. When arrivals are i.i.d. and only three flows are active, it is not possible for any queue to become unstable.
3. For any arrival process, if the queues are initially empty, the occupancies of the two sets of input queues: $L_{1,1}(n) + L_{2,2}(n)$ and $L_{1,2}(n) + L_{2,1}(n)$ can differ by at most four cells. This means that if instability occurs, at least one queue from each set must be unstable.

This leads us to make the following conjecture, as yet unproved.

Conjecture: *For a 2x2 switch, LQF and OCF are stable for all ergodic, admissible arrival processes.*

2.2.4 Stability of NxN Switch

Theorem 4.4: *LQF is stable for all admissible i.id. arrival processes.*

Proof: The proof is given in appendix 4. In summary, we show that for an NxN switch scheduled using the LQF algorithm, there is a negative single-step drift in the sum of the squares of the state. In particular,

$$\mathrm{E}\left[\underline{L}^T(n+1)\underline{L}(n+1) - \underline{L}^T(n)\underline{L}(n) \middle|\ \underline{L}(n)\right] \leq -\varepsilon\|\underline{L}(n)\| + k\,, \tag{7}$$

$k > 0, \varepsilon > 0$. ❒

The term $-\varepsilon\|\underline{L}(n)\|$ indicates that whenever the occupancy of the input queues is large enough, the expected drift is negative. Should $\|\underline{L}(n)\|$ become very large, the downward drift also becomes large, and so the stability is quite "strong". This leads us to the following conjecture.

Conjecture: *LQF and OCF are stable for all ergodic, admissible arrival processes.*

One possible definition of stability for ergodic arrivals is

$$\lim_{T \to \infty} \frac{1}{T} \mathrm{E}\left[\sum_{n=0}^{T} \|\underline{L}(n)\|\right] < \infty . \quad (8)$$

Although we have not been able to find admissible arrival processes for which an NxN switch is unstable using the LQF and OCF, we have not been able to prove that this conjecture is true in general. This remains an open problem.

3 Iterative Maximal Weight Matching Algorithms

A goal of this work is to determine fast scheduling algorithms that can be readily implemented in hardware. Unfortunately, the maximum weight matching algorithms, LQF and OCF, are very complex to implement, and require an $O(N^2 \log_a N)$ running time.

As an alternative, we now consider iterative approximations to LQF and OCF, based on the SLIP and PIM algorithms, that are designed to be readily implemented in hardware and to quickly find a maximal weight match. These algorithms are called respectively, *i*-LQF and *i*-OCF.

3.1 *i*-LQF

Like PIM and SLIP, *i*-LQF is an iterative algorithm consisting of N output and N input arbiters operating in parallel. The scheduler maintains an N^2 word memory; each entry indicates the occupancy of an input queue, $L_{i,j}(t)$. The word width, *b*, is determined by the maximum queue length, L_{max}

$$2^b \geq L_{max} . \quad (9)$$

As before, at the beginning of each cell time the match process begins over. All inputs and outputs are initially unmatched and only those inputs and outputs not matched at the end of one iteration are eligible for matching in the next. Connections made in one iteration are never removed by a later iteration, even if a larger sized match would result. The three steps of each iteration are as follows:

Step 1. *Request*. Each unmatched input sends a request word of width 2^b bits to each output for which it has a queued cell, indicating the number of cells that it has queued to that output.

Step 2. *Grant*. If an unmatched output receives any requests, it chooses the largest valued request. Ties are broken randomly.

Step 3. *Accept*. If an unmatched input receives one or more grants, it accepts the one to which it made the largest valued request. Ties are broken randomly.

3.2 Properties

The *i*-LQF algorithm has the following properties:

Property 1. Independent of the number of iterations, the longest input queue is always served. The longest input queue will lead to the largest request in the first iteration, which must be granted, resulting in the largest grant. This must be accepted.

Property 2. As with *i*-SLIP, the algorithm converges in at most N iterations. If during some iteration no connection is made, the algorithm has converged and no further connections are possible. So, prior to convergence at least one connection is added per iteration. There are N inputs and N outputs, requiring at most N iterations to converge.

Property 3. For an inadmissible offered load, an input queue may be starved. This is the same as for LQF, described in Section 2.1.

3.3 *i*-OCF

The *i*-OCF algorithm eliminates the starvation problem of *i*-LQF by favoring cells with a longer waiting time. *i*-OCF differs from *i*-LQF only in Step 1: the value of the request from input *i* to output *j* equals the waiting time, $W_{i,j}(t)$, of the cell at the head of queue $Q(i, j)$.

3.4 Properties

The *i*-OCF algorithm has the following properties:

Property 1. Independent of the number of iterations, the cell (*C)* that has been waiting the longest time in the input queues is served. First note that *C* must be at the head of its FIFO queue. The input will make the largest request to the scheduler in the first iteration on behalf of *C,* which must be granted, resulting in the largest grant. This must be accepted.

Property 2. As with *i*-LQF, the algorithm converges in at most N iterations.

Property 3. No input queue can be starved indefinitely.

3.5 Performance of *i*-LQF and *i*-OCF

3.5.1 Uniform Workload

Both *i*-LQF and *i*-OCF have worse throughput-delay performance than LQF and OCF. This is to be expected — neither of the iterative algorithms will remove connections in an attempt to maximize the match. The performance of both algorithms under uniform i.i.d. Bernoulli arrivals is illustrated in Figure 4.7. Once again, both iterative algorithms are stable up to an offered load of 100%, albeit with a slightly larger latency than for the maximum weight algorithms.

3.5.2 Nonuniform Workload

We have not found a workload for which the *i*-LQF and *i*-OCF are unstable; under non-uniform arrival patterns, both algorithms exhibit similar throughput-delay performance to the LQF and OCF algorithms. We illustrate this by way of the same simple (but arbitrary) example as before, shown in Figure 4.4. Our results in Figure 4.8 show not only that the *i*-LQF and *i*-OCF algorithms are stable, but that their performance is very close to the maximum weight algorithms.

3.5.3 Stability for 2x2 Switch

Conjecture: *With sufficient iterations, i-LQF and i-OCF are stable for a 2x2 switch with all admissible workloads.*

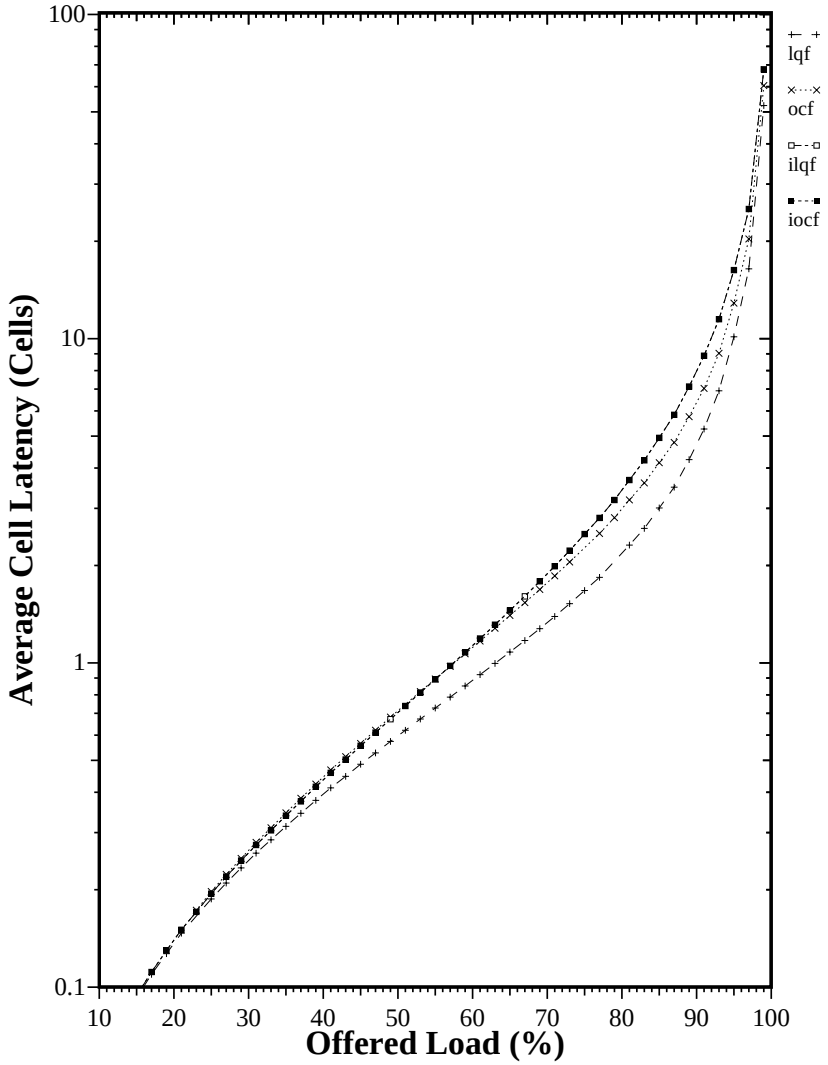


FIGURE 4.7 Performance of *i*-LQF and *i*-OCF algorithms for uniform i.i.d. Bernoulli arrivals, compared with LQF and OCF algorithms.

3.6 Implementation of *i*-LQF and *i*-OCF

Both *i*-LQF and *i*-OCF are relatively simple to implement in hardware, although more complex than the *i*-SLIP algorithm described in Chapter 3. The main difference is that the simple priority encoders that perform arbitration in SLIP are replaced by more complex comparators.

Figure 4.10 shows a schematic design for *i*-LQF. The design consists of 2N arbiters.[1] Each

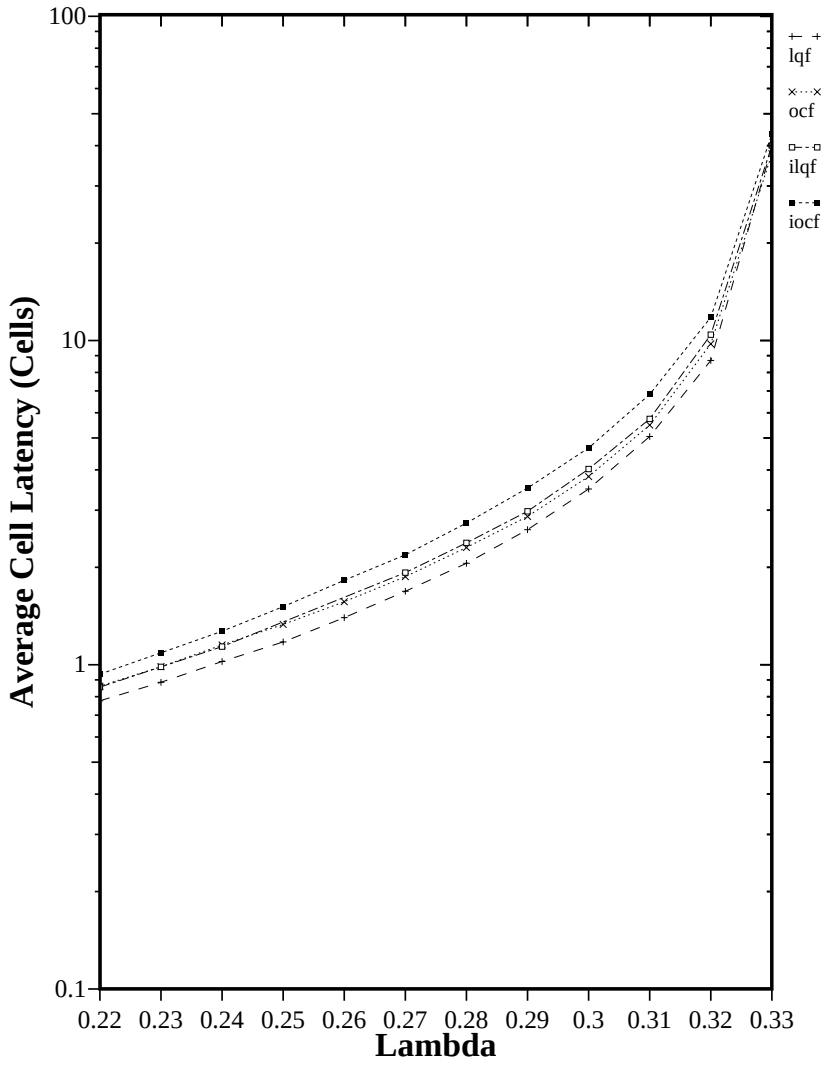


FIGURE 4.8 Performance of *i*-LQF and *i*-OCF algorithms under the non-uniform workload, shown in Figure 4.4.

For *i*-LQF, the registers for output *j* maintain the occupancy values for each input queue $Q(i \in I, j)$ and similarly, the registers at input *i* maintain the occupancy values for each input queue $Q(i, j \in J)$. If output *j* grants to input *i*, then grant arbiter *j* enables the register containing $L_{i,j}(t)$ at accept arbiter *i*. The finite state machine prevents matched inputs and outputs from par-

1. As with SLIP, the arbiters may be reduced to N by sharing them between the grant and accept stages.

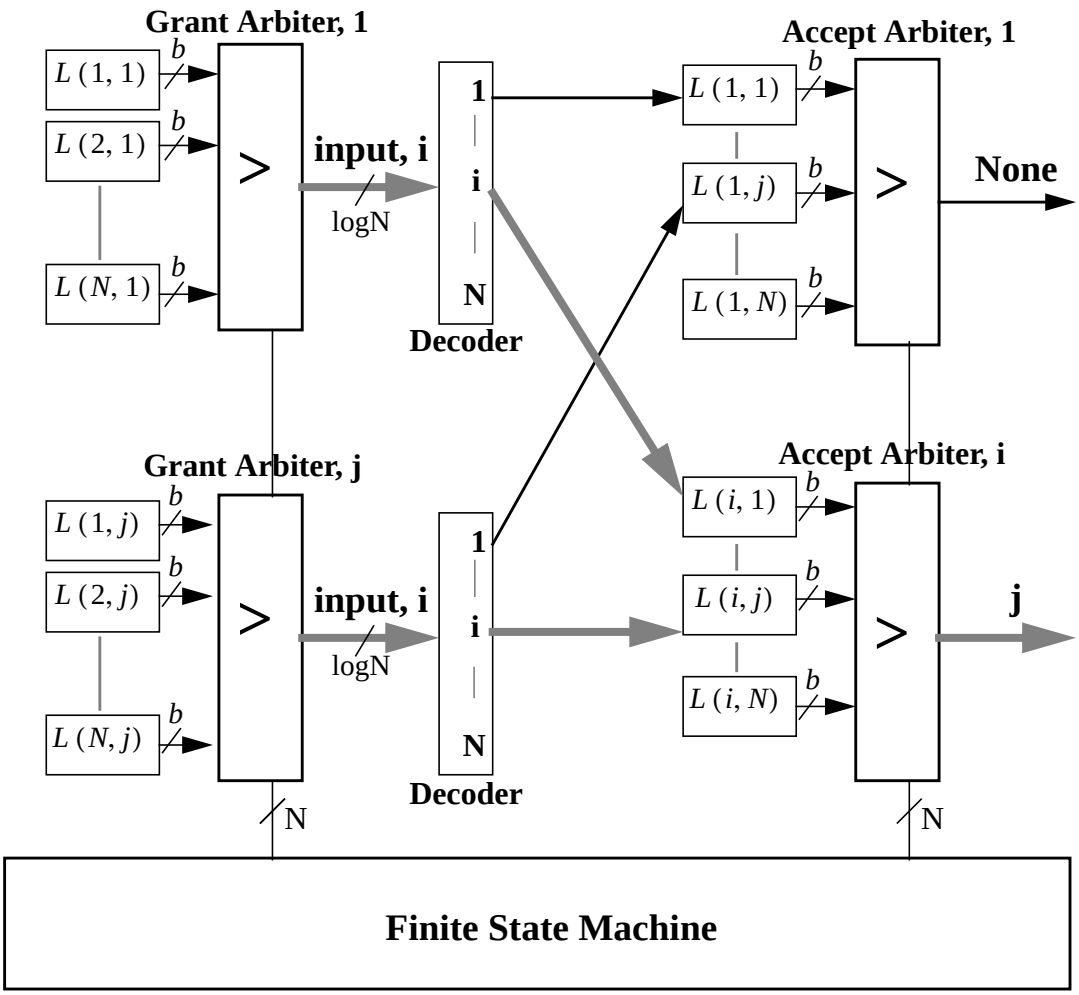


FIGURE 4.9 Implementation of NxN *i*-LQF algorithm using 2N arbiters. For brevity, only grant arbiters 1 and *j* and accept arbiters 1 and *i* are shown here.

ticipating in future iterations by disabling the corresponding arbiters. The implementation for *i*-OCF is exactly the same as for *i*-LQF, except that the registers hold cell waiting times, rather than queue occupancies.

The implementation for *i*-LQF is clearly more complex than for *i*-SLIP, both in the number of gates required to implement the arbiters and in the size of the registers required to hold the queue occupancy values. *i*-LQF also requires more state to be updated at the beginning and end of each cell time. In a cell time, at most one queue at each input can increase its occupancy and by only one cell. The input must indicate to the central scheduler which queue has increased; the scheduler

must increment the corresponding value. Likewise, in a cell time, at most one queue at each input may decrease its occupancy, and by only one cell. The input does not need to indicate which queue has changed: it is the one that the scheduler selected during its arbitration. The scheduler must decrement the corresponding value.

Choosing the value for *b* is an important design decision, affecting the number of different queue lengths that can be distinguished. If the size of the input queues is large, or if the gate count is to be minimized, it may be required that $2^b < L_{max}$. Two possible modifications to the algorithm in this case are: (1) if the occupancy $L_{i,j}(t) \geq 2^b$, issue a request of size 2^b; or (2) if the maximum queue size $L_{max} = 2^b \times 2^n$, issue a request of size $L_{i,j}(t)/2^n$, which is readily achieved in hardware by truncating the lowest *n*-bits.

It is interesting to note that although for small values of N and *b* the complexity of *i*-LQF is greater than for *i*-SLIP, its complexity increases more slowly with N. In Chapter 2 we found that the number of gates required to implement *i*-SLIP increases with N^4. *i*-LQF consists of 2N comparators, each comprising $O(\log b \log N)$ gates and N^2 registers each with $O(b)$. For small *b* and N, the comparators will dominate the total number of gates, increasing with $N \log N$. With sufficiently large N, the total gate count is dominated by the registers which increase with N^2. In both cases, this increase is at a slower rate than for *i*-SLIP.

4 Stable Marriages

The *stable marriage problem* was first introduced in 1962 by Gale and Shapley [13] and since then has been studied extensively [15], [26]. Solutions to the stable marriage problem find a *stable* and complete matching on a bipartite graph and can therefore be used to schedule cells in an input-queued switch. More importantly, there exists a well known algorithm (the Gale-Shapley Algorithm — GSA) that is feasible to implement in hardware and will always find a stable matching in *N* iterations.

We begin this section with a description of the stable marriage problem and describe the GSA. We then consider two different ways that the GSA can be used to schedule cells in an input-queued switch; one is a variation of LQF and the other a variation of OCF. We finish the section with a description of the implementation of these algorithms.

4.1 The Stable Marriage Problem

Although rather dated, the classical problem is stated in terms of two equal-sized sets: a set of N men and a set of N women all of whom wish to get married to a member of the other set. Each man independently creates an ordered preference list, ranking each of the N women. Each woman does the same, ranking each of the N men. The aim is to find a *stable* matching between the set of men and women so that each man is matched to a woman and each woman is matched to a man. A match is *unstable* if there is a couple who are not matched to each other, yet both prefer the other to their partner in the matching. A *stable* matching is any matching that is not *unstable*.

4.2 The Gale-Shapley Algorithm

In their original paper, Gale and Shapley prove that:

1. Every instance of the stable marriage problem admits at least one stable matching. They prove this with an algorithm, GSA, that will always find a matching in $O(N^2)$ running time.
2. The GSA has two distinct versions: the male-optimal GSA and the female-optimal GSA. The male-optimal GSA will simultaneously give all the men the best partner and all the women the worst partner that they could have in any stable matching; and vice-versa for the female-optimal GSA.

The Gale-Shapley Algorithm is usually expressed in terms of a series of marriage proposals. In the male-optimal version, the proposals are always from men to women. We will describe here the male-optimal version of the algorithm.[1] Initially, each person is *free* and may become *engaged* as the algorithm progresses. Women who become engaged never become free again, whereas engaged men may be rejected by their partner and become free again.

1. The female-optimal algorithm is obtained by simply reversing the roles of the sexes.

Each man proposes to the women, one at a time, in the order that they appear in his preference list. If the woman that he proposes to is free, she will accept his proposal. If she is already engaged, but prefers the new proposal, she will reject her previous partner in favor of the new man. The rejected man is now free and will resume making proposals to the remaining women in his preference list. The algorithm terminates when everyone is engaged.

The somewhat surprising findings of Gale and Shapley were that the algorithm will always converge before any man reaches the end of his preference list, and that on completion of the algorithm the engaged couples always form a stable matching.

4.3 Analogy to Switch Scheduling

A stable matching is an example of a bipartite graph matching. As described in Chapter 1, scheduling cells in an input-queued switch is analogous to finding a bipartite graph matching between the set of switch inputs and outputs. So if, for example, we assign the set of men to represent the switch outputs and the set of women to represent the switch inputs, a stable matching will represent a legal switch configuration.

Next we shall consider the principle ways that the GSA differs from the iterative algorithms discussed earlier in this thesis. We then describe two algorithms, GS-LQF and GS-OCF, that are based on the GSA and may be used for scheduling cells in an input-queued switch.

There are three main ways in which the GSA differs from the iterative maximum weight algorithms described earlier in this chapter:

1. A *stable* matching is in general different from a maximum sized or maximum weight matching. It is not clear that a stable matching will lead to an efficient use of switch bandwidth, or that it will prevent connections from being starved of service. In particular, we know that the GSA will favor either outputs or inputs. Later, we will consider an algorithm that provide a more egalitarian matching.
2. The stable marriage problem and the GSA are usually defined for a complete matching in which every input and every output is matched. It is not always the case in an input-queued switch that a complete match is possible: most often, only a subset of the input-queues are occupied. This means that some inputs and outputs will have missing entries in their preference lists. Fortunately, with a small modification, the GSA algorithm will

still work.[1] However, it is known that just a few missing elements in preference lists can lead to a large reduction in the size of a stable matching [15].

3. In the iterative algorithms *i*-SLIP, *i*-LQF and *i*-OCF, once a connection has been accepted it is not rejected by a later iteration. In the GSA, connections made in one iteration may be rejected by a later iteration. In principle, by rearranging connections the GSA can find a larger sized or weight match than iterative algorithms that do not remove connections established in an earlier iteration. The benefit of this is not clear: in (1) above we saw that the stable matching does no attempt to maximize either the size or the weight of a match.

4.3.1 The GS-LQF Algorithm

GS-LQF (GSA with LQF preference lists) is the Gale-Shapley algorithm with preference lists based on the occupancy of the input queues.

Output j determines its preference list $\mathbf{R}_{\mathrm{O}}(j)$ based on the occupancy of each of the N input queues, $L(i,j), \forall i \in \mathbf{I}$

$$\mathbf{R}_{\mathrm{O}}(j) = [i_1, i_2, \ldots, i_N], \quad \text{where: } L(i_1,j) \geq L(i_2,j) \geq \ldots \geq L(i_N,j). \tag{10}$$

Similarly, input i determines its preference list

$$\mathbf{R}_{\mathrm{I}}(i) = [j_1, j_2, \ldots, j_N], \quad \text{where: } L(i,j_1) \geq L(i,j_2) \geq \ldots \geq L(i,j_N). \tag{11}$$

4.3.2 The GS-OCF Algorithm

GS-OCF (GSA with OCF preference lists) is the Gale-Shapley algorithm with preference lists based on the waiting time of the cells at the head of each input queue.

Output j determines its preference list $\mathbf{R}_{\mathrm{O}}(j)$ based on the waiting times of the cells at the head of each of the N input queues, $W(i,j), \forall i \in \mathbf{I}$

$$\mathbf{R}_{\mathrm{O}}(j) = [i_1, i_2, \ldots, i_N], \quad \text{where: } W(i_1,j) \geq W(i_2,j) \geq \ldots \geq W(i_N,j). \tag{12}$$

Similarly, input i determines its preference list

$$\mathbf{R}_{\mathrm{I}}(i) = [j_1, j_2, \ldots, j_N], \quad \text{where: } W(i,j_1) \geq W(i,j_2) \geq \ldots \geq W(i,j_N). \tag{13}$$

1. It may be shown that the algorithm will partition the inputs and outputs into those that are matched in all stable matchings and those that are never matched [15].

4.4 Performance of GS-LQF and GS-OCF Algorithms

The performance of the GSA depends on the weight of the stable matching. In general, the relationship between a stable matching and a maximal weight matching is unknown. In fact, it is not intuitively obvious that a stable matching algorithm will perform well in this application, particularly when several of the input queues are empty.

Surprisingly, we have found through simulation that the performance of the GS-LQF and GS-OCF[1] algorithms are, respectively, *indistinguishable* from the performance of *i*-LQF and *i*-OCF. It remains an open problem whether the performance is identical for all traffic patterns.

4.5 Implementation of GS-LQF and GS-OCF

We now describe a parallel, iterative version of the GSA to match inputs to outputs in an input-queued switch. The algorithm is relatively simple to implement in hardware, although more complex than the *i*-SLIP algorithm described in Chapter 3.

Figure 4.10 shows the schematic design of GS-LQF which is almost identical to the implementation of *i*-OCF in Figure 4.10. As with *i*-LQF, the design consists of 2N arbiters. Each grant arbiter maintains a register value for each of the requesting input queues. For GS-LQF, the registers at output *j* maintain the preference list: the occupancy values for each input queue, $\mathbf{R}_O(j)$ in Equation 11. Similarly, the registers at input *i* maintain the elements of $\mathbf{R}_I(i)$. The Finite State Machine controls the preference lists at each arbiter by enabling only those that are active in a particular iteration.

In the first iteration, all of the entries in the arbiters' preference lists are enabled. Each output will grant ("propose") to the input which makes the largest request. For example, in Figure 4.10, grant arbiters 1 and *j* both grant to input *i*. Grants are single bit values, used to enable entries in an accept arbiter's preference list. The input may or may not have already accepted a grant. If it has not, then it accepts the largest enabled entry in its preference list. If it has, then it only accepts the new value if it exceeds the value of the previously accepted grant.

1. Our simulations only considered the output-optimal algorithms.

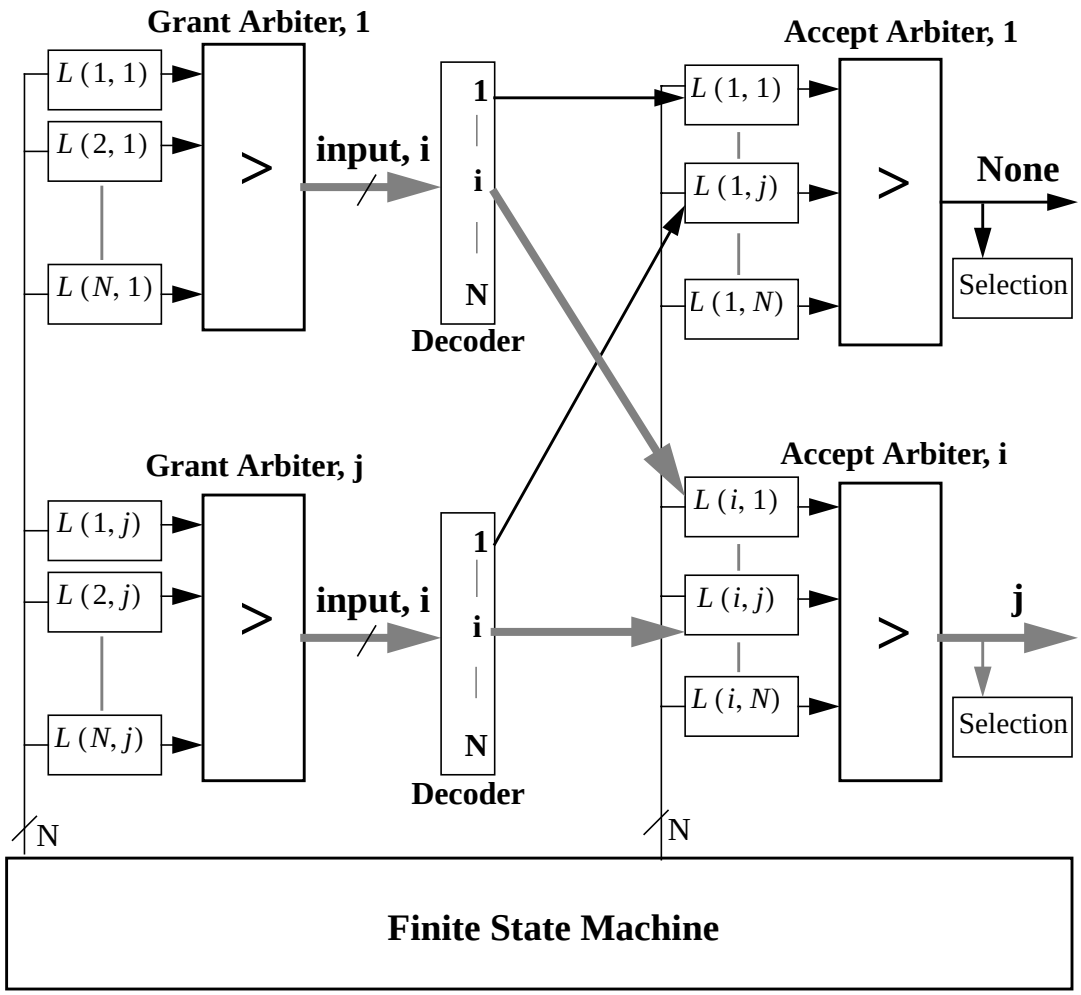


FIGURE 4.10 Implementation of NxN GS-LQF algorithm using 2N arbiters. For brevity, only grant arbiters 1 and j and accept arbiters 1 and i are shown here.

As with the implementation of *i*-LQF the complexity of the implementation of GS-LQF depends on the number of bits, *b*, used to represent $L_{i,j}(t)$. This will determine the number of gates required to register the preference lists, $O(bN^2)$ and the number of gates required to implement each comparator, $O(\log N \log b)$.

4.6 Egalitarian Stable Marriage

In [15], the authors describe an algorithm to find an egalitarian stable marriage. Whereas the male-optimal GSA simultaneously:

$$\begin{aligned} &\text{minimizes} \quad \sum_{(m,w) \in M} R_m(m,w), \quad \text{and} \\ &\text{maximizes} \quad \sum_{(m,w) \in M} R_w(w,m) \end{aligned} \tag{14}$$

the egalitarian stable matching

$$\text{minimizes} \quad \sum_{(m,w) \in M} [R_m(m,w) + R_w(w,m)], \tag{15}$$

where:

$R_m(m,w)$ = the position of woman w in man m's preference list,

$R_w(w,m)$ = the position of man m in woman w's preference list.

If we could use this algorithm to schedule cells, it would remove the decision as to whether to give preference to either inputs or outputs. Unfortunately, the best known algorithm for finding an egalitarian matching requires an $O(N^4)$ running time and is impractical to implement in hardware.