DEADLOCK FREE BUFFER ALLOCATION IN CLOSED QUEUEING NETWORKS

S. KUNDU [1] and I.F. AKYILDIZ [2]

[1] *Department of Computer Science, Louisiana State University, Baton Rouge, Louisiana 70803, U.S.A.*
[2] *School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332, U.S.A.*

Received 30 October 1987
Revised 20 July 1988

Abstract

Blocking queueing networks are of much interest in performance analysis due to their realistic modeling capability. One important feature of such networks is that they may have deadlocks which can occur if the node capacities are not sufficiently large. A necessary and sufficient condition for the node capacities is presented such that the network is deadlock free. An algorithm is given for buffer allocation in blocking queueing networks such that no deadlocks will occur assuming that the network has the special structure called cacti-graph. Additional algorithm which takes linear time in the number of nodes, is presented to find cycles in cacti networks.

Keywords: Performance evaluation, queueing networks, finite buffers, blocking, deadlock

1. Introduction

Since in actual systems the resources have a finite capacity, queueing networks with blocking must be used for performance analysis. In queueing networks with blocking, a node can be thought of as a device with a finite length queue. The network is simply a set of arbitrarily linked nodes. Blocking arises due to the limitations imposed by the capacity of these nodes. In particular, blocking occurs when the flow of jobs through one node is momentarily suspended due to the fact that another node has reached its capacity limitation.

Several papers have been published dealing with various types of blocking. Previous work regarding the blocking networks falls into three classes, Onvural and Perros [10]:

[2] Akyildiz's work was supported in part by School of Information and Computer Science, ICS, of Georgia Tech and by the Air Force Office of the Scientific Research (AFOSR) under Grant AFOSR-88-0028.

(i) *Transfer blocking*. Upon completion of the service at a node i, a job attempts to enter the destination station j. If node j is full at that moment, the job is forced to wait in node i's server until it can enter destination node j. The server remains blocked for this period of time. It cannot serve any other jobs waiting in the queue. This type of blocking has been used to model systems such as production systems and disk I/O subsystems, Akyildiz [1,2], Perros and Altiok [12].

(ii) *Service blocking*. A job at node i declares its destination node j before it starts its service. If node j is full, the i-th server is blocked before service begins. When a departure occurs from destination node j, the i-th server becomes unblocked and the job begins receiving service. This blocking type has been used to model systems such as production systems and telecommunication networks, Boxma and Konheim [5], Gordon and Newell [6], Konheim and Reiser [9].

(iii) *Rejection blocking*. Upon service completion at node i, a job attempts to join destination node j. If node j is full at that moment, the job receives a new service at node i. This is repeated until the job completes service at a time when it can proceed to station j, Balsamo and Iazeolla [3], Hordijk and van Dijk [7].

Several other investigators in recent years have published results on blocking queueing networks, Perros [11].

An important consideration in blocking queueing networks of any type is that finite node capacities and blocking can introduce the *deadlock* situation. In a simple example, deadlock may occur if a job which has finished its service at node i's server wants to join node j, whose capacity is full. That job is blocked in node i. Another job which has finished its service at j-th node now wants to proceed to the i-th node, whose capacity is also full. It blocks the j-th node. Both jobs are waiting for each other. As a result, a deadlock situation arises. Furthermore, the possibility of deadlock in a network increases with the ratio of the number of jobs in the network to the total capacity of the network. As the total number of jobs approaches the total capacity of the network, the probability of deadlock increases.

The most important issue is the allocation of node capacities in a queueing network such that deadlocks cannot occur. In this work we give a necessary and sufficient condition for a queueing network to be deadlock free, and present an algorithm for computing the capacities for the nodes such that no deadlock will occur in the network.

2. Deadlock freedom in blocking networks

Let Γ be a closed queueing network of type 1 with N nodes and K jobs where all jobs are of the same class. Each node contains $m_i \geq 1$ servers with a single

queue. There are no restrictions regarding the service time distribution and scheduling disciplines of the nodes. Let B_i be the buffer size, or capacity, of the i-th node where $B_i = Queue\ Capacity\_i + m_i$ ($for\ i = 1, \ldots, N$). There can be at most B_i jobs at node i at any time, including the jobs which are currently being serviced. A job which is serviced by the i-th node proceeds to the j-th node with probability p_{ij} for (i, $j = 1, 2, \ldots, N$), if the number of jobs at the j-th node has not exceeded that capacity B_j. Otherwise, the job is blocked at the i-th node until a job at node j has completed service and a place becomes available. This model is classified as type 1 blocking above. It is understood that once a job selects a destination (probabilistically or deterministically) it cannot change the destination. This is implication in type 1 blocking network definition. We assume that each job has a fixed class assigned to it and this cannot change because it is blocked at some point in time.

The following theorem describes a necessary and sufficient condition for a closed queueing network to be deadlock free. A *cycle* C is a sequence of nodes $(x_1, x_2, \ldots, x_j)$ such that each pair of consecutive nodes is joined by an arc (x_i, x_{i+1}), including the arc (x_j, x_1).

THEOREM 1

A closed queueing network of type 1 with finite node capacities $\{B_i : 1 \leqslant i \leqslant N\}$ is *deadlock free* if and only if for each cycle C in the network the following condition (1) holds. Simply stated, the total number of jobs in the network must be smaller than the sum of node capacities in each cycle.

$$K < \sum_{j \in C} B_j. \tag{1}$$

Proof

(i) *Necessity*. Suppose that there is a cycle $C = (1, 2, \ldots, M)$, ($M \leqslant N$), which violates the condition (1). Consider a state of the network in which each node i in C is saturated, i.e., the current number of jobs at node i, $1 \leqslant i \leqslant M$, equals its buffer capacity B_i. There is a positive probability for such a state of the network since $K \geqslant \sum_{j \in C} B_j$. Now, assume that for each node i in C, the job which is currently being serviced at i finishes and it wants to move to the next node $i + 1$ in the cycle ($M + 1 = 1$). There is also a positive probability for this to happen. This, however, results in a deadlock within the cycle C. Since there is also a positive probability that a job in another node may want to move to a node in C, eventually all nodes will be deadlocked with probability 1.

(ii) *Sufficiency*. Suppose that there is a blocking. For example, the node 1 is blocked. Then there is another node 2 such that the job at node 1 which has completed service wants to move to node 2 cannot do so. This means that node 2 is saturated and must itself be blocked. Otherwise, at some point in

the future, the current job at node 2 would move out, and the job from node 1 could then move to node 2. By repeating the above argument for node 2 and so on, we get a sequence of nodes $(1, 2 \cdots)$ with the following properties:

a) Each node i is blocked and is saturated.

b) $(i, i+1)$ is an arc of the network Γ.

Since Γ is finite, the nodes $\{i : i \geqslant 1\}$ must include a cycle, $C = \{1, 2, \ldots, M\}$, without loss of generality. Since each i, $1 \leqslant i \leqslant M$, is saturated, we have

$$K \geqslant \sum_{j \in C} B_j, \tag{2}$$

This violates the inequality (1) for the cycle C, a contradiction. This completes the proof.

If Γ is a tandem network, consisting of a single cycle, then the inequality (1) corresponds to the total buffer size $B = \sum_{i=1}^{N} B_i$ of the network being at least $(K+1)$. This can be achieved by taking $B_1 = B_2 = B_{(N-1)} = 1$, and $B_N = K - N + 2$. (Indeed, a better throughput may be achieved by allocating buffer sizes at the nodes in inverse proportion to their service rates. This has been verified in some experimental cases, but has not been established formally. The issue of buffer allocation problem for improving throughput is not considered here.)

COROLLARY

A necessary and sufficient condition for a tandem network to be deadlock free is

$$K < \sum_{i=1}^{N} B_i \tag{3}$$

3. Deadlock free buffer allocation in blocking networks

A set of buffer sizes $\{B_i : i = 1, 2, \ldots, N\}$ for which inequality (1) holds for every cycle C is called a *deadlock free buffer allocation*, or *dfba*. We denote by β the minimum value of $B = \sum_{i=1}^{N} B_i$ taken over all *dfba*'s for the network. The minimum buffering requirement of the network for avoiding deadlocks is β. It is clear that for a buffer allocation $\{B_i : 1 \leqslant i \leqslant N\}$ to be deadlock free, it needs to satisfy only those inequalities in (1) which correspond to the elementary cycles, i.e., the cycles which do not pass through the same node more than once. We assume here that each node and each arc of the network belongs to at least one cycle; a node, however, can belong to several cycles. We give below an algorithm for computing β and a corresponding buffer allocation $\{B_i : 1 \leqslant i \leqslant N\}$ for the case where the network Γ has the form of a tree of elementary cycles. Figure 1

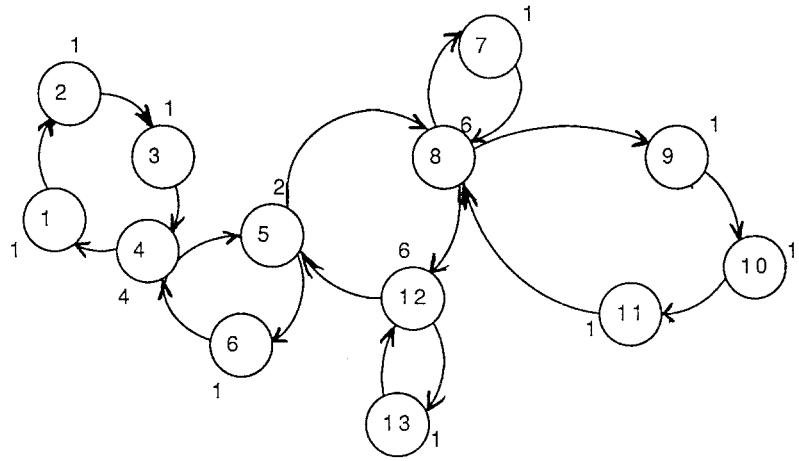

Fig. 1. A network and a deadlock free buffer allocation for $K = 6$ jobs; next to each node is the value B_i.

shows such a network, where the direction of each arc is clockwise around the cycle. Such a network is called a *cactus* network, Behzad et al. [4]. A cactus has the property that no two cycles have more than one node in common. We define the following terms to describe the algorithm:

(i) A *contact* node is a node at which two or more cycles meet. Since we assume that every node belongs to a cycle, this is the same as saying that there are 2 or more arcs leaving the node.

(ii) A *terminal* cycle is a cycle which contains at most one contact node. Unless the cactus is a cycle, a terminal cycle has exactly one contact node, and there are at least two terminal cycles in a cactus. In fig. 1, the cycles (1, 2, 3, 4) and (7, 8) are two of the four terminal cycles, with the contact nodes 4 and 8, respectively.

The algorithm *BUFFER* given below assigns values to B_i in an "outside-in" fashion. The correctness of the algorithm is based on lemma 1 which shows that given any terminal cycle C there is a deadlock free buffer allocation where all nodes other than its contact node (if any) has buffer size 1. The step (d) of the algorithm makes use of this principle. The other steps of the algorithm extends the current buffer allocation gradually until the condition (1) is satisfied for all cycles.

Algorithm BUFFER(N):

(a) Assign $B_i = 1$ initially for each node of the network.

(b) Choose a terminal cycle C of the network.

(c) If C is equal to the entire network, and C does not satisfy (1), then increase the buffer size at one of its nodes to make $\sum_{i=1}^{N} B_i = (K + 1)$, and stop.

(d) Let i be the contact node of C. If C does not satisfy (1), then increase B_i such that

$$\sum_{i \in C} B_i = (K+1)$$

(e) Delete all nodes in C from the network, except the node i.
(f) Repeat steps (b)–(e) until "stop" is encountered.

If we are given an existing buffer allocation in which the condition (1) does not hold, then the algorithm *BUFFER* can start with the given allocation instead of using the initialization $B_i = 1$, for $1 \leqslant i \leqslant N$ in step (a) above. In that case, the new buffer allocation $\{B_i' : 1 \leqslant i \leqslant N\}$ obtained by the algorithm satisfies the following conditions, and gives a minimal increament in B_i''s so that the condition (1) is satisfied.

(i) $B_i' \geqslant B_i$, for $i = 1, 2, \ldots, N$.
(ii) $\{B_i'\}$ satisfies (1), and
(iii) $\sum_{i=1}^{N} B_i'$ or equivalently, the total increase $\sum_{i=1}^{N}(B_i' - B_i)$ is minimum subject to (i).

LEMMA 1

Let $\{B_i\}$ be any buffer allocation scheme which satisfies inequality (1). Let j be the contact node of a terminal cycle C. Then there is another *dfba* $\{B_i'\}$ where each non-contact node of C has unit buffer allocation, i.e., $B_i' = 1$ for $i \neq j$ and $i \in C$ such that $\sum B_i = \sum B_i'$.

Proof

Let i be any node in C and $i \neq j$. If the buffer size B_i at i is greater than one, then decrease B_i to $B_i' = 1$ and increase the buffer size at j from B_j to $B_j' = B_j + B_i - 1$. It is easy to see that $\{B_i'\}$ is still a *dfba* and $\sum B_i = \sum B_i'$. Repeat the above process for each $i \neq j \in C$. This proves the lemma.

4. Example

We illustrate the algorithm by computing a set of B_i's for the network in fig. 1 using $K = 6$ jobs. Each row in table 1 below shows only the changes made to the buffer sizes in that step. In particular, step (e) is not shown.

5. The algorithm *CYCLE* for cacti networks

The step (b) of the algorithm for computing β requires that a terminal cycle of the network be found. The algorithm CYCLE below finds all cycles in a cacti network. A terminal cycle can be then selected by *first* finding the number of

Table 1
Illustration of the algorithm for computing the deadlock free buffer allocation $\{B_i : 1 \leqslant i \leqslant n\}$ for $K = 6$; here $\beta = 27$

Step #	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}
a)	1	1	1	1	1	1	1	1	1	1	1	1	1
b) $C = (1, 2, 3, 4)$													
d)				4									
b) $C = (4, 5, 6)$													
d)					2								
b) $C = (7, 8)$													
d)								6					
b) $C = (8, 9, 10, 11)$													
d)	no change to B_8												
b) $C = (12, 13)$													
d)												6	
b) $C = (5, 8, 12)$													
c)	no change in any of the B_i's												
Final values of B_i	1	1	1	4	2	1	1	6	1	1	1	6	1

contact nodes in the cycle. A node is a contact node if and only if its outdegree (= number of arcs leaving that node) is greater than one. Since Γ is a cactus network, and each arc of Γ belongs to a cycle, the outdegree of a node equals its indegree (= the number of arcs entering that node). When a cycle is deleted from the network in step (e) of the algorithm for β, the outdegree of each node in the cycle is decreased by one. This will allow the deletion of a terminal cycle subsequently. The algorithm takes linear time in the number of nodes in the network. The more general algorithm in Johnson [8] for finding all cycles, which works for all directed graphs and is also linear in the number of cycles and nodes, is considerably more complex. This prompted us to design the simpler algorithm *CYCLE* given here. The algorithm *CYCLE* is presented in a different format

$adj[1] = (2)$
$adj[2] = (3)$
$adj[3] = (4)$
$adj[4] = (5,1)$
$adj[5] = (8,6)$
$adj[6] = (4)$
$adj[8] = (12,9,7)$
$adj[12] = (5,13)$

(i) The adjacency lists used in the computation shown in (ii)

Illustration of the procedure CYCLE

Node visited	Stack S (top of S is on left)	Startlist
4	(4)	(4)
5	(5,4)	empty
8	(8,5,4)	
12	(12,8,5,4)	
5	cycle = (12,8,5)	
	(5,4)	(8,12)
6	(6,5,4)	
4	cycle = (6,5,4)	
	(4)	
1	(1,4)	
2	(2,1,4)	
3	(3,2,1,4)	
4	cycle = (3,2,1,4)	
	empty	
12	(8)	(12)
	...	
	...	

(ii) Part of the computation starting at node 4

Fig. 2. Illustration of the algorithm CYCLE.

than the algorithm *BUFFER* in section 4.1 as it uses a more complex control flow and data structure to achieve its efficiency. The stack S stores the nodes that have been visited but not yet outputted as part of a cycle; initially S is empty. The array visit[ ] is used for identifying the nodes which have been already

```
procedure CYCLE(N);
    begin
    if (there is no node of d[i] > 1) then
        N is a cycle
    else begin
        choose a node i such that d[i] > 1;
        initialize startlist to i;
        visit[i] ← 1;
        while (startlist is not empty) do
          begin
            choose i from startlist and remove it from startlist;
            add i to S;
            while (S is not empty) do
              begin
                i ← top(S);
                if (adj[i] is non-empty) then
                    begin
                      choose the first node j in adj[i];
                      remove j from adj[i];
                      d[i] ← d[i] - 1;
                      if (visit[j] = 1) then {a cycle is found at j}
                              begin
                                repeat {output the cycle}
                                    k ← top(S);
                                    output k;
                                    if (d(k) > 1) then
                                          add k to the beginning of startlist;
                                    pop(S);
                                until (top(S) = j);
                                output j;
                              end
                      else begin
                            visit[j] ← 1;
                            push(j, S);
                           end;
                      endif;
                else
                      pop(S);          {S is empty now}
                endwhile;
            endwhile;
        endelse;
    end.
```

visited, indicated by visit[i] = 1; initially, visit[i] = 0 for each i. The array $d[i]$ denotes the outdegree of the node i. The "*startlist*" is a list of visited contact nodes for which all cycles containing them have not been outputted and needs further processing. We assume that the network is represented as a list of adjacency lists, adj[i], one list per node. The lists adj[i] are gradually reduced to empty lists as items are deleted from them.

The algorithm CYCLE is illustrated in fig. 2 using the network in fig. 1. We show the values of the stack S and the startlist as they change in the first few iteration of the inner while-loop, assuming that the algorithm is started at the contact node 4. The nodes of the cycles are outputted in the reverse order.

References

[1] I.F. Akyildiz, On the exact an approximate throughput analysis of closed queueing networks with blocking, IEEE Transactions on Software Engineering (1988) 62–71.

[2] I.F. Akyildiz, Product form approximations for queueing networks with multiple servers and blocking, IEEE Transactions on Computers (1989) to appear.

[3] S. Balsamo and G. Iazeolla, Some equivalence properties for queueing networks with and without blocking, Proc. Performance 83 Conference, eds. A.K. Agrawala and S. Tripathi (North-Holland Publ. Co., 1983) 351–360.

[4] M. Behzad, G. Chartrand and L. Lesniak-Foster; *Graphs and Diagraphs* (Wadsworth International Group, California, 1979).

[5] O.I. Boxma and A.G. Konheim, Approximate analysis exponential queueing systems with blocking, Acta Informatica 15 (1981) 19–66.

[6] W.J. Gordon and G.F. Newell, Cyclic queueing systems with restricted queues, Operations Research 15, Nr. 2 (April 1967) 266–277.

[7] A. Hordijk and N. van Dijk, Networks of queues with blocking, *Proc. 8th Int. Symp. on Computer Performance Modelling, Measurement, and Evaluation*, Amsterdam, November 4–6, 1981.

[8] D.B. Johnson, Finding all the elementary circuits of a directed graph, SIAM Journal on Computing 4 (1975) 77–84.

[9] A.G. Konheim and M. Reiser, Finite capacity queueing systems with applications in computer modeling, SIAM Journal Computing 7, No. 2 (1978) 210–229.

[10] R.O. Onvural and H.G. Perros, On equivalences of blocking meachanisms in queueing networks with blocking, Operations Research Letters 5 (1986) 293–297.

[11] H.G. Perros, Queueing networks with blocking: a bibliography, ACM Sigmetrics Performance Evaluation Review, (1984).

[12] H.G. Perros and T. Altiok, Approximate analysis of open networks of queues with blocking: tandem configurations, IEEE Transactions on Software Engineering, Vol. SE-12, No. 3 (1986) 450–462.