A DAG-Based Algorithm for Prevention of Store-and-Forward Deadlock in Packet Networks

DAVID GELERNTER, STUDENT MEMBER, IEEE

Abstract—**Store-and-forward deadlock (SFD) occurs in packet-switched computer networks when, among some cycle of packets buffered by the communication system, each packet in the cycle waits for the use of the buffer currently occupied by the next packet in the cycle. Several techniques for the prevention of SFD are known, but all exact some cost in terms of efficient and flexible packet handling. An ideal SFD-prevention technique is as unobtrusive as possible; it imposes no routing restrictions on packets, does not require that the buffer pool on each node grow with network size, and imposes no buffer-pool partitioning. All SFD-prevention techniques described so far lack some or all of these desirable properties. The new algorithm here described has all of them; in return, it imposes other unconventional costs. Under certain circumstances it requires that packets be rerouted around areas of potential deadlock, and one arbitrarily chosen node is required to accept within finite time any packet seeking entrance to its buffer pool, even if this requires erasing some packet. It is argued nonetheless that these costs are imposed infrequently enough and are sufficiently well manageable by heuristic techniques to make this new algorithm an attractive and practical alternative to the older techniques. An implementation designed for a microprocessor network now under construction is described.**

Index Terms—**Communication software, deadlock prevention, network operating systems, packet networks, store-and-forward deadlock.**

I. INTRODUCTION

STORE-and-forward deadlock (SFD) occurs in packet-switching computer networks when some group of packets exists in the network communication system such that among this group no packet has arrived at its destination and none can make a next hop toward its destination. The simplest case of SFD is direct SFD in which, given two adjacent network nodes *Ni* and *Nj*, *Ni*'s packet buffers are full of packets destined for *Nj*, and *Nj*'s buffers are full of packets for *Ni*; as a result, no packet can move. In more complex SFD's a cycle of nodes is involved, each holding packets bound for the next node in the cycle. SFD is described more precisely in Section II.

Several algorithms for prevention of SFD on networks of arbitrary topology are known, and they will be considered in two groups.

1) The "structured buffer pool technique" is described by Raubold and Haenle in [1]; it is discussed in [2] and by Gerla and Kleinrock in their survey of flow control techniques [3]. This technique requires that every node's buffer pool be divided into buffer classes, a 0-hop class, 1-hop class, and so on through a K-hop class, where K is the length of the longest path in the network. The buffers of the i-hop class may hold only those packets that are i hops or more removed from their nodes of origin. Thus, a packet that is h hops from its source node may be buffered in any j-class such that $j \leq h$. Toueg and Ullman [4] discuss variants of this scheme and converse schemes (the forward count and forward state algorithms) in which admission to buffer classes is based on distance to destination rather than distance from source.

2) Merlin and Schweitzer [5] discuss a generalization of the structured pool technique. Their algorithm requires the construction of a directed acyclic buffer graph. The nodes of the graph are buffers, and there may be an edge between two nodes if they represent buffers that are located on one processor or on adjacent processors. Every permissible packet route must correspond to some path in the directed graph. A packet may hop from buffer b_1 to buffer b_2 only if the edge $b_1 \rightarrow b_2$ is part of the graph. (This restriction may be relaxed when the network is not congested.) Reference [6] discusses a scheme that is in a sense the converse of the technique of [5]. It requires the construction, not of an acyclic buffer graph, but of a set of buffer *cycles* containing all permissible packet routes. This algorithm relies for deadlock prevention not on an acyclic collection of paths, but on a priority protocol that ensures that every cycle must contain at all times at least one empty buffer.

All of these techniques prevent SFD, each at some cost in flexibility and efficiency of packet handling by the communication system. Obviously, SFD-prevention must be paid for somehow, but the costs imposed by the techniques discussed above are troublesome. An ideal SFD-prevention algorithm would have the following properties.

a) Packet Routing is Unrestricted. Ideally, the communication system should be free to construct packet routes dynamically without interference from the SFD-prevention algorithm. The algorithms in group 1) above do not impose routing restrictions, except for the forward-count and forward-state techniques in [4]. These require the length of a packet's route to be declared beforehand; the route may be constructed dynamically, but must not exceed the predeclared

Manuscript received February 2, 1981; revised May 27, 1981. A preliminary version of this paper was presented at the International Conference on Performance of Data Communication Systems and Their Applications, Paris, France, September 14-16, 1981.

The author is with the Department of Computer Science, State University of New York, Stony Brook, NY 11794.

0018-9340/81/1000-0709$00.75

length. In the group 2) algorithms a packet's next hop must follow an edge in the buffer graph [5] or–a more severe restriction–the unique next edge in the buffer cycle [6]. In both cases the restriction may be relaxed when the network is not congested. The severity of the restriction in [5] depends on the elaborateness and completeness of the buffer graph constructed.

b) *The Size of the Buffer Pool Required at Each Node Should be Independent of the Size of the Network.* This requirement is particularly important for networks of microcomputers that may be enlarged regularly and may ultimately grow very large. The group 1) schemes require a buffer pool on each node that is strictly larger than the longest path in the network. If communication, then, is permitted between any two network nodes, buffer pools must grow with network diameter. The pools required by the group 2) algorithms vary from node to node and are topology-dependent, but may in general grow with network size.

c) *Minimal Partitioning Should be Imposed on Buffer Pools.* The number of times that a packet seeking entrance to some node's buffer pool is refused admission despite there being empty buffers somewhere in the pool should be minimal. Congestion control may make some degree of pool partitioning necessary or desirable, but the amount of partitioning required simply for SFD-prevention should be as small as possible. The group 1) schemes impose explicit [1] or implicit [4] pool partitioning as described above. The group 2) schemes partition pools explicitly on the basis of their position in the acyclic buffer graph or in some buffer cycle. Note that minimal partitioning is desirable mainly so that pools may be used efficiently and delivery times be minimized, but also so that heavily used packet-switching software be no more complex than necessary.

The algorithm proposed below has desirable properties a), b), and c) above. It imposes new costs in return. The costs it imposes, however, are unconventional and make this scheme not strictly comparable to the other SFD-prevention techniques. Specifically, the new algorithm

i) imposes no *a priori* routing restrictions,

ii) requires a pool size per node that is independent of network size; no more than two buffers on most nodes and one on the rest are required to insure SFD-freedom,

iii) imposes no buffer-pool partitioning.

The costs of the new technique are the following.

1) Under certain conditions packets are forcibly rerouted around potential deadlocks.

2) One arbitrarily chosen node is required to accept within finite time any packet seeking entrance to its buffer pool, even if this requires its erasing some packet. This means in effect that one node will have a pool that is substantially larger than the pools on other nodes (to minimize instances of such deliberate erasure).

It will be argued that despite its costs, the algorithm's benefits make it well worth consideration. An implementation planned for a network of microcomputers now under construction at Stony Brook is discussed below.

Section II defines SFD precisely. In Section III the algorithm is described and its correctness proven; Section IV gives examples. Section V places the algorithm's logical basis in the context of other SFD-prevention algorithms. Section VI discusses the problems of livelock and of hardware failure, and the last section presents conclusions.

II. Store-and-Forward Deadlock

The formalities discussed below lead to a precise definition of SFD, and then to a series of facts about SFD that will be the basis of the correctness proof in Section III and the comparison among SFD-prevention algorithms in Section V.

Let p_i be a class of packets such that:

1) all packets in class p_i are currently buffered on the same node;

2) if one packet in class p_i is eligible for buffering after its next hop in some set B of buffers on adjacent nodes, then *each* packet in p_i is eligible for buffering in exactly the same set B of buffers after *its* next hop.

The routing algorithm, in other words, cannot distinguish among the packets in class p_i; the next-hop routing requirements of all packets in p_i are identical; p_i is in effect one packet. The buffers in set B may all be located on the same adjacent node; this will be the case if the routing algorithm assigns each packet upon arrival to a single, fixed output queue. If, on the other hand, the routing algorithm is willing to send a given packet to any of several adjacent nodes–to list one packet, in other words, in several output queues simultaneously–then the set B may contain buffers on several adjacent nodes. If buffer pools are unpartitioned, then, if B contains *some* buffer on node Nq, it contains *all* buffers on Nq. But if buffer pools are partitioned, then B may include some but not all of Nq's buffers.

Define the predicate $L(p_i)$ to mean that class p_i is deadlocked. By definition, class p_i is deadlocked when it is part of a cycle of buffer requests such that no packet in p_i can move until some packet in p_i *has* moved. This is stated precisely below.

If $P = \{p_1, p_2, \cdots, p_n\}$–i.e., P is a set of packet classes–then $L(P)$ means that all $p_i \in P$ are deadlocked.

For packet class p_i and set of classes Q_i

$$p_i w Q_i$$

means that the set of buffers in which the packets of p_i may be buffered after their next hop is currently occupied by the packet classes included in Q_i. (p_i waits for Q_i.)

If PwQ, then the set Q is equal to

$$\bigcup_i Q_i \text{ such that for some } p_i \text{ in } P,\ p_i w Q_i.$$

a is the "arrived" packet class consisting of packets that have arrived at their destination. By assumption, the packets in class ***a*** will be removed from the buffers they occupy in finite time without making further hops.

e stands for an empty buffer. The empty buffer is itself defined to be a class insofar as p_i may wait for ***e*** as it may wait for any packet class.

w^* is the transitive closure of w. Thus

$$Aw^*B$$

means that A is related to B by a string $AwS_1wS_2w\cdots wB$, in which 0 or more w operators occur. (Thus, Aw^*A is true.)

$$Aw^+B$$

means that A is related to B by the string $AwS_1wS_2w\cdots wB$, in which 1 or more w operators occur.

Finally, let $W(P)$ be the set

$$\{q \mid (q \in Q) \text{ and } (Pw^+Q)\}.$$

$W(P)$, P's "waiting set," is the set of all packet classes waited on directly or indirectly by the packet classes of P.

If $p_i \in P$, then the set $W(p_i)$ is defined to be the set $W(\{p_i\})$. That is, $W(p_i)$ consists of those classes on which p_i itself, as distinct from the other classes in the set P, waits.

The formal definition of deadlock can now be stated as follows:

$$L(P) \equiv \boldsymbol{e} \notin W(P) \text{ and } \boldsymbol{a} \notin W(P) \text{ and } Pw^+P.$$

The definition states that no packet in P can move, since neither $\boldsymbol{e}$ nor $\boldsymbol{a}$ is part of its waiting set. It further states that if P is deadlocked, it is part of a deadlocked cycle; this stipulation distinguishes deadlock from indefinite postponement. A packet is indefinitely postponed if it is waiting for the use of buffers occupied by deadlocked packets, but it is not *itself* deadlocked unless it itself blocks other deadlocked packets.

Notice that it follows from the definition that

$$L(A) \text{ and } L(B) \Rightarrow \text{if } Aw^*B, \text{ then } Bw^*A.$$

Since it is also true that Aw^*A and that w^* is transitive over packet sets, w^* is an equivalence relation over deadlocked packet sets.

1)–5) below are further facts about deadlock that follow from the definition and will be useful in the subsequent discussion.

1) $L(P) \Rightarrow \exists Q$ such that (PwQ) and $L(Q)$.

Clearly

$$(\sim L(Q)) \text{ and } (PwQ) \Rightarrow \sim L(P).$$

Hence, it follows since $\boldsymbol{e}$ and $\boldsymbol{a}$ are by definition not deadlocked, that

2) (PwQ) and $((\boldsymbol{e} \in Q)$ or $(\boldsymbol{a} \in Q)) \Rightarrow \sim L(P)$.

Clearly, also

3) $\exists Q$ such that $((\boldsymbol{e} \in Q)$ or $(\boldsymbol{a} \in Q))$ and $(Pw^+Q) \Rightarrow \sim L(P)$.

4) $L(P) \Rightarrow \forall Q$ such that Pw^+Q, $L(Q)$.

Finally, by dint of the distinction drawn between deadlock and indefinite postponement,

5) $L(P) \Rightarrow \exists Q$ such that QwP and $L(Q)$.

III. The Algorithm

Let any network N be represented by undirected graph G. Nodes in G correspond to processors in N; edges in G correspond to (assumed bidirectional) communication lines in N. The algorithm requires the construction of some directed graph G_1 with the following properties.

1) The nodes and directed edges of G_1 correspond to the nodes and the undirected edges of G.

2) G_1 is acyclic and has exactly one node with no incident incoming edges. This unique node will be called the root of G_1.

The following procedure will produce a directed graph G_1 from an undirected graph G. First, construct a directed spanning tree T of G with root r—all nodes are reachable from r via edges in T. Now assign direction to all edges in G that are not part of T such that these edges together with the edges of T remain acyclic. To accomplish this, edges between nodes at different distances from the root may be pointed away from the root, edges at the same distance from the root may be assigned either direction, so long as all edges at any given distance from the root remain acyclic. The result is a G_1 graph.

Having constructed graph G_1, construct the inverse graph G_2 by reversing the direction of all edges in G_1.

Now, every hop a packet makes in N corresponds to an edge either in G_1 or in G_2. A *G_1-packet* is a packet whose next hop will be over a G_1 edge; a G_2 packet is analogously defined. If a packet may wait simultaneously on more than one output queue for the use of whatever link becomes available first, then as long as it waits for the use of at least one G_2 link, it is a G_2 packet.

The G_1 root node (or simply the root) r has no incoming G_1 edges. All packets in r are G_1 packets because no G_2 edges leave r. The nodes without incoming edges in the G_2 graph (there may be one or many) are called G_2 roots. All packets in G_2 roots are G_2 packets because no G_1 edges leave G_2 roots.

The deadlock-free property of the algorithm derives from the following requirement: every node must at all times contain either an empty buffer or a G_2 packet. This means the following.

1) If a nonroot node Nq, after accepting some new packet, finds that every one of its buffers is now filled with a G_1 packet, then some packet in Nq must be redirected and routed out over a G_2 edge. (That is, some packet must be transformed into a G_2 packet.)

2) The root, since it can contain no G_2 packets, must always have an empty buffer. Any packet waiting for admission to r must be admitted in finite time, even if this involves erasing some packet in r.

Erasure at r may be made arbitrarily rare by increasing the size of the buffer pool on r relative to buffer pools in the rest of the network.

Note that the forced redirection of packets required by 1) occurs only when all packets in a node are G_1 packets, and therefore contending for the use of the same set of links. Such redirection generally corresponds to the forcing of some packet from a set of congested output queues to some empty queue, which may be inherently desirable. (Of course, the fact that all other packets buffered on a node are G_1 packets does not mean that all or any are waiting for the particular G_1 link that the rerouted packet would have used.)

The algorithm assumes that if there is an adjacent empty buffer in a packet's waiting set, that buffer must be used in finite time (unless the packet is shipped elsewhere).

Implementation Note: Suppose a G_1 packet p arrives at Nj

from *Ni over a* G_1 *edge*. Suppose further that p makes Nj full of G_1 packets, and thus makes a rerouting necessary. The rerouting is accomplished if p itself is chosen for rerouting, and routed back to node Ni—that is, if p is simply sent back where it came from. (The hop from Nj to Ni must be a G_2 hop.) But this means that rather than accepting and rerouting p, Nj could simply have refused p to begin with. A packet like p–any G_1 packet arriving over a G_1 link–will be referred to below as a *refusable packet*. The algorithm does not require that refusable packets be refused, but–an implementation point–if they are, a rerouting is avoided. Note that such an implementation imposes an implicit buffer pool partitioning (albeit a minor one), insofar as the last buffer in a pool full of G_1 packets is closed to a G_1 packet arriving over a G_1 link.

Note finally that if the buffer pool on r is not proportionately larger than other buffer pools, the algorithm may be considered as implementing the following pragmatic congestion control policy. When traffic in the net is such as to make deadlock possible, excess packets are drawn to the root and eliminated. (Ordinarily, of course, an implementation, by specifying a large buffer pool at the root, will be designed to avoid packet deletion, not to allow it routinely for congestion control.)

Theorem: The algorithm is SFD-free.

Proof: 1) No G_2 packet can be deadlocked.

Suppose that p is a class of G_2 packets and $L(p)$. Then $pwQ_1wQ_2\cdots$ and all Q_i are deadlocked. But since each node contains either an empty buffer or a G_2 packet, each Q_i–since by assumption it contains no empty buffer–contains a G_2 packet. (Note that if a packet is waiting for any buffer on Nq, it is waiting for *all* buffers on Nq. Thus, each Q_i will contain a G_2 packet.) Since each Q_i contains a G_2 packet, some node whose buffers hold packets in Q_{i+1} must be reachable via G_2 from some node whose buffers hold packets in Q_i. Since G_2 is acyclic, there must therefore exist some Q_n such that all packets buffered on r are part of Q_n. But, since by assumption $\boldsymbol{e}$ is contained in r, $\boldsymbol{e} \in Q_n$. Since pw^+Q_n and $\boldsymbol{e} \in Q$, $L(p)$ is false by 3) in Section II and part 1 is proven.

2) No G_1 packet can be deadlocked.

Let p be a class of G_1 packets. If p, again, is waiting for some buffer in node Nq, it is waiting for all buffers in Nq. But since every node must contain $\boldsymbol{e}$ or a G_2 packet, p can never be deadlocked, since it must always be waiting on some nondeadlocked packet. Thus, no packet, whether G_1 or G_2, can be deadlocked and the theorem is proven.

In sum, the algorithm has the following advantages.

1) There are no *a priori* routing restrictions.

2) Required buffer pool size is independent of the size of the network. Deadlock prevention requires ≥ 2 buffers on nonroots and ≥ 1 buffer on (G_1 or G_2) root nodes. (A nonroot must have at least two buffers, otherwise any packet accepted would be required to be a G_2 packet and no G_1 hops from a nonroot could ever occur. A G_2 root requires only one buffer, since it buffers only G_2 packets by definition. The G_1 root must always include an empty buffer, and therefore requires one, but no more than one, buffer.)

3) Buffer pools are unpartitioned. Any packet may be shipped to any empty buffer (with the reservation that a given implementation may choose to refuse refusable packets). The costs are as follows.

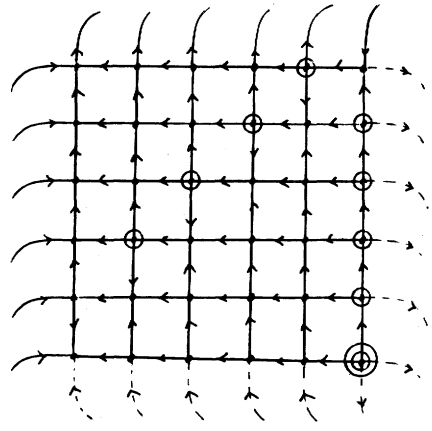

Fig. 1.

1) Packets buffered in full buffer pools on nodes other than the root are subject to rerouting.

2) Packets buffered in a full buffer pool on the root are subject to deletion.

IV. Examples

A. The Torus

Fig. 1 shows a G_1 graph for a 6×6 torus with each row and each column looping back on itself. The root is circled twice. Assuming that most packets on the torus will follow a minimum-hop route from source to destination, rerouting imposed by the algorithm is normally restricted to the following cases.

1) Suppose that packet p is at least one horizontal and one vertical hop removed from its destination Nd, and that p is therefore willing to make either a horizontal or a vertical hop so long as that hop brings it closer to Nd. A packet may be rerouted only if its next hop is a G_1 hop. Since packets like p may all potentially make their next hop over either of two adjacent links, rerouting of such packets may take place only on nodes that hold two adjacent outward-directed G_1 edges. But if, furthermore, the implementation chooses to refuse all refusable packets, then rerouting is possible only on nodes with *three* outward-directed G_1 edges; G_1 packets incoming over G_1 links will be refused rather than rerouted. Redirection of packets at least one horizontal and one vertical hop from their destinations may therefore take place only at the nodes circled in Fig. 1.

2) Suppose packet p has arrived on an axis of its destination. Such a packet will have only one choice for its next hop. It will hop from node to node along the destination's axis until it arrives. Packets in this category may be redirected whenever their next hop is a G_1 hop, and this may occur on any node with at least one outgoing G_1 link, that is, on all nodes but G_2 roots. If we assume again an implementation that refuses refusable packets, redirection of such packets is restricted to nodes with at least two outgoing G_2 links.

Thus, although in theory any G_1 packet is susceptible to rerouting on any nonroot node, the regularity of the torus graph restricts rerouting to a subset of G_1 packets.

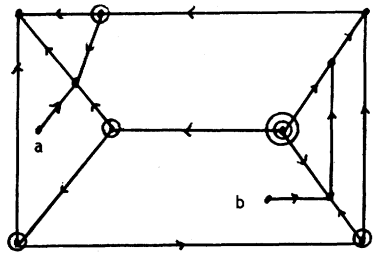

Fig. 2.

B. An Arbitrary Graph

Fig. 2 shows a G_1 graph for the Arpanet of the early 1970's [7]. Nodes of degree two are omitted. (Arpanet is discussed here only as an example of a realistic network graph.) The root is circled twice. Assuming an implementation that refuses refusable packets, packet rerouting may occur only at the four singly circled nodes—the rest lack two outbound G_1 links. (No packet buffered on a node of degree two–the majority of nodes in this case, although they are not shown in the figure–can be rerouted.) Note that nodes of degree one (the nodes labeled *a* and *b*) have been added as supplementary roots. This is permissible. Neither will be required to delete packets assuming: 1) that packets are never forwarded through *a* or *b*, and 2) that neither *a* nor *b* ever entirely fills its packet buffers with new packets.

V. The Logical Basis of SFD-Prevention

The necessary and sufficient condition for SFD that is given as the definition of SFD in Section II requires global knowledge of the disposition of packets in the net to detect. In practice, therefore, SFD-prevention algorithms choose some condition that is necessary to SFD–ordinarily the condition given in 3) in Section II–and make this condition impossible.

1) The structured buffer pool schemes generally require that each packet p has either arrived at its destination, has an empty buffer to hop to, or is waiting for the use of a buffer currently occupied by a packet that is either closer to its destination (the forward count and forward state algorithms in [4]) or farther from its source than is p itself. Since all routes are assumed finite in length, $W(p)$ therefore includes either **e** or **a** for all p, and by 3) in Section II $L(p)$ is impossible.

2) The algorithm in [5] is most easily understood as making the necessary condition given by the second part of the SFD definition

$$Pw^{+}P$$

impossible. Since all packets must hop to their destinations over an acyclic buffer graph, $p \in W(p)$ is impossible for all p, thus $L(p)$ is impossible.

3) The cycle-based algorithm in [6] requires packets to hop to their destinations over a cyclic buffer graph in which—so the protocol assures—at least one buffer is always free. It follows that for all p, $W(p)$ includes either **e** or **a**, and again by 3) in Section II, $L(p)$ is impossible.

4) The algorithm given above similarly ensures that for all p, $\mathbf{e} \in W(p)$, as given in the proof.

The necessary condition for SFD made impossible by the algorithm above can be understood in another way as well—as a generalization of a necessary condition for direct SFD. As described in the introduction, direct SFD involves two adjacent nodes, each full of packets for the other. Beginning with the assumption that all buffer pools are completely unstructured–any packet may be admitted to any empty buffer–direct SFD may be prevented simply by adding the requirement that no node ever be entirely full of packets outbound to any one adjacent node [6]. In fact, the restriction sufficient to ensure direct-SFD-freedom can be made even weaker. Direct SFD is impossible if, given any pair of adjacent nodes, at least one of the pair is never entirely full of packets for the other. (Suppose *Ni* and *Nj* form a direct SFD, but there is some packet on *Nj* that is *not* bound for *Ni*. Then this packet is not part of the direct SFD and, when it moves, its space will be available to some packet p on *Ni*, assuming only that p does not hop from *Ni* to *Nj* with the intention of making its next hop from *Nj* back to *Ni*. Thus, there is no direct SFD.)

The algorithm for prevention of general SFD given above may be understood as generalizing this weaker condition for direct-SFD-freedom. The generalization speaks not of a node's neighbors, but of a node's two disjoint *sets* of neighbors—those nodes reachable via G_1 being one set, those reachable via G_2 being the other. As stated above, it is sufficient to prevent direct SFD that given any pair of neighboring nodes, one of the pair never be full of packets for the other. It is sufficient to prevent *general* SFD that given any pair of neighboring nodes, one of the pair never be full of packets for the *set* of neighbors in which the other is included. Thus, under the algorithm, given neighboring nodes *Ni* and *Nj*, *Ni* might be full of packets for *Nj*, but then $Ni \rightarrow Nj$ must be a G_2 hop. Since $Nj \rightarrow Ni$ is therefore a G_1 hop, N_j can never be full of packets for the set of neighbors in which *Ni* is included—otherwise it would be full of G_1 packets.

VI. Livelock and Failure

Livelock as discussed by Toueg [8] is a state in which some packet is prevented indefinitely from making its next hop because of unfavorable traffic patterns. The structured buffer pool algorithms are livelockable as follows. Suppose the distance-from-source technique is used. Node *Nj* may be h hops from *Nq* and *Ni* g hops, $g < h$. Packets originating on *Nj* and passing through *Nq* are eligible for buffering in all k-hop buffer classes such that $k \leq h$. Packets originating on *Ni* and passing through *Nq* are only eligible for k-hop classes such that $k \leq g$. A stream of packets from *Nj* may therefore fill all k-classes where $k \leq g$—all classes, in other words, for which packets from *Ni* are eligible. But when all such classes are filled with packets from *Nj*, more packets from *Nj* may still be admitted, although none from *Ni* may be admitted. If a new packet from *Nj* arrives at *Nq* before each old packet from *Nj* is passed along, *Ni*'s packets will never be admitted to *Nq* and are "livelocked."

In [8] a modification of the structured pool scheme is proposed that requires all packets to be time-stamped at creation, and each packet's time-stamp to be checked against the

time-stamps of all other waiting packets before it is granted or refused admission to any node. The group 2) algorithms [5], [6], on the other hand, are in general inherently livelock-free because buffers are reserved for packets traveling given routes. One route's packets cannot exclude another route's. (In the directed buffer graph technique, buffers on nodes *Na* and *Nb* may each feed via a directed edge into a single buffer on adjacent node *Nc*. A continuous stream of packets through *Na* bound for *Nc*, together with unfair scheduling on *Nc*'s part, may indefinitely exclude packets from *Nb*. But this is not livelock. *Nc* is free at all times to admit packets from *Nb*; it simply chooses not to. In the previous example, *Nq* was forbidden by the algorithm to admit packets originating on *Ni*.)

The algorithm given above is also livelock-free because it imposes no buffer pool partitioning. So long as empty buffers are made available fairly to waiting packets, no waiting packet can be excluded indefinitely.

The fact that the algorithm allows packets to be forcibly rerouted makes it possible in theory for a packet to circle endlessly and never arrive at its destination, a situation akin to livelock. In practice, when some packet must be rerouted on *Ni*, any packet on *Ni* may be chosen for rerouting. Packets that have been rerouted once may have a header bit so indicating (the "reroute bit"), and will be avoided when again buffered in a pool where rerouting is necessary. A node that is entirely full of once-rerouted packets might choose to delete incoming packets rather than reroute any packet a second time. (A node in this state meets the algorithm's requirements by including *e* rather than a G_2 packet, as the root does.) Alternatively, incoming packets might be deleted only by a node full of twice-rerouted packets and so on. Given the expected infrequency of the situation, the procedure chosen is likely to be of little practical significance.

Given the global clock posited in [8], the algorithm can be adapted to ensure that every packet, *as long as it is not deleted* at the root, will eventually reach its destination, that is, that no packet will loop indefinitely. The global time at which a packet is created, concatenated with the unique id of the processor on which it is created, may serve as a packet's birthdate. Assuming that no processor can create more than one packet simultaneously, there is accordingly at all times a unique oldest packet in the net. When a packet must be chosen for rerouting, the youngest packet in the pool is chosen. Clearly, every packet must eventually either arrive, be deleted at the root, or become the unique oldest packet in the net. If the last-named occurs, then this unique oldest packet can never be rerouted and must therefore, unless it is deleted, arrive at its destination in finite time. (A header field counting the number of times a packet has been rerouted, concatenated with a serial number assigned sequentially by each process or to each packet it creates, may replace the global creation-time discussed above.)

To summarize, under the general group 1) schemes, a packet, although not deadlocked, may fail to reach its destination because of livelock; under the new algorithm, a packet, although not deadlocked, may fail to reach its destination because of indefinite looping or—of course—because of deletion at the root. A livelock-free version of the group 1) schemes is known [8], and as for the new algorithm, it seems likely that heuristic techniques—a header "reroute bit" or bits, increasing the size of the buffer pool on the root—will be sufficient to control the problem. In all events, the new algorithm relies on a higher level end-to-end protocol to detect and cause the retransmission of lost packets. It promises not delivery, but only probable delivery of any given packet. (This is of course the most it could promise in any case given a system in which hardware may fail.)

Hardware failures affect the algorithm in two cases. 1) All of some node *Np*'s outgoing G_2 links, or the nodes terminating them, fail. 2) The root fails, or all links leading to it fail.

In case 1), *Np* becomes a "pseudoroot," unable to reroute packets over G_2 links. It may delete excess incoming packets as the real root does, or it may initiate a reconfiguration of the network graph. In case 2), some node adjacent to the root takes over the root's functions, deleting excess incoming packets for as long as the root remains down.

VII. Comment and Conclusions

SFD-prevention has been implemented, to the author's knowledge, only on the GMDNET, which uses the structured-pool technique [2]. But SFD-prevention was considered essential on the Stony Brook Network (SBN), a torroidal network of microcomputers designed to function as a single machine or "network computer." Because several SBN nodes may cooperate in processing a single computing activity and system functions are distributed network-wide, packet traffic may be heavy at times and efficient SFD-prevention was required. The importance of an efficient algorithm is increased by the simplicity of the SBN's hardware. Each node is both host and packet-switch, and interfaces directly to its neighbors. Since there are no front-end processors, time and space given up to packet switching is lost to computing.

The algorithm described above is currently being implemented on the SBN. Answers to open questions about its efficiency and practicality will come—for our setting at least—directly from measured network performance.

Acknowledgment

The author wishes to thank Prof. A. Bernstein for extensive advice and assistance. The author also thanks Prof. J. Cherniavsky, the referees for useful comments, and his colleagues K. Bertapelle and B. Ensor for accompanying him on the long and still unconsummated search for the Free Lunch Algorithm.

References

[1] E. Raubold and J. Haenle, "A method of deadlock-free resource allocation and flow control in packet networks," in *Proc. ICCC 1976*, Toronto, Canada, Aug. 1976, p. 483.

[2] A. Giessler, J. Haenle, A. Koenig, and E. Pade, "Free buffer allocation—An investigation by simulation," *Comput. Networks*, vol. 2, p. 191, 1978.

[3] M. Gerla and L. Kleinrock, "Flow control: A comparative survey," *IEEE Trans. Commun.*, vol. COM-28, p. 553, Apr. 1980.

[4] S. Toueg and J. Ullman, "Deadlock-free packet switching networks," in *Proc. ACM Symp. Theory Comput.*, 1979, p. 89.

[5] P. Merlin and P. Schweitzer, "Deadlock avoidance in store-and-forward networks—I: Store-and-forward deadlock," *IEEE Trans. Commun.*, vol. COM-28, p. 325, Mar. 1980.

[6] D. Gelernter and K. Bertapelle, "General and topology-specific algorithms for store-and-forward deadlock prevention in packet networks," State Univ. of New York, Stony Brook, Tech. Rep. 80/018, Dec. 1980.
[7] E. Schelonka, "Resource sharing with Arpanet," in *Proc. Nat. Telecommun. Conf. 1974*, p. 1045; also in M. Abrams, R. Blanc, and I. Cotton, *Computer Networks: A Tutorial*. New York: IEEE, 1978, pp. 5–19.
[8] S. Toueg, "Deadlock- and livelock-free packet switching networks," in *Proc. ACM Symp. Theory Comput.*, 1980, p. 94.

David Gelernter (S'79) received the B.A. and M.A. degrees from Yale University, New Haven, CT, in 1976 and 1977, respectively.

Currently, he is a doctoral student in the Department of Computer Science, State University of New York, Stony Brook. His research interests include communications software, distributed programming languages and operating systems, and computer applications in clinical medicine.

The Memory System of a High-Performance Personal Computer

DOUGLAS W. CLARK, BUTLER W. LAMPSON, AND KENNETH A. PIER

***Abstract*—The memory system of the Dorado, a compact high-performance personal computer, has very high I/O bandwidth, a large paged virtual memory, a cache, and heavily pipelined control; this paper discusses all of these in detail. Relatively low-speed I/O devices transfer single words to or from the cache; fast devices, such as a color video display, transfer directly to or from main storage while the processor uses the cache. Virtual addresses are used in the cache and for all I/O transfers. The memory is controlled by a seven-stage pipeline, which can deliver a peak main-storage bandwidth of 533 million bits/s to service fast I/O devices and cache misses. Interesting problems of synchronization and scheduling in this pipeline are discussed. The paper concludes with some performance measurements that show, among other things, that the cache hit rate is over 99 percent.**

***Index Terms*—Cache, high bandwidth, memory system, pipeline, scheduling, synchronization, virtual memory.**

I. Introduction

THIS paper describes the memory system of the Dorado, a high-performance compact personal computer. This section explains the design goals for the Dorado, sketches its overall architecture, and describes the organization of the memory system. Later sections discuss in detail the cache (Section II), the main storage (Section III), interactions between the two (Section IV), and synchronization of the various parallel activities in the system (Section V). The paper concludes with a description of the physical implementation (Section VI) and some performance measurements (Section VII).

Manuscript received May 31, 1980; revised March 22, 1981.

D. W. Clark was with the Xerox Palo Alto Research Center, Palo Alto, CA 94304. He is now with the Systems Architecture Group, Digital Equipment Corporation, Tewksbury, MA 01876.

B. W. Lampson and K. A. Pier are with the Xerox Palo Alto Research Center, Palo Alto, CA 94304.

A. Goals

A high-performance successor to the Alto computer [21], the Dorado is intended to provide the hardware base for the next generation of computer system research at the Xerox Palo Alto Research Center. The Dorado is a powerful but personal computing system supporting a single user within a programming system that extends from the microinstruction level to an integrated programming environment for a high-level language. It is physically small and quiet enough to occupy space near its users in an office or laboratory setting, and inexpensive enough to be acquired in considerable numbers. These constraints on size, noise, and cost have had a major effect on the design.

The Dorado is designed to rapidly execute programs compiled into a stream of *byte codes* [19]; the microcode that does this is called an *emulator*. Byte code compilers and corresponding microcode emulators exist for Mesa, a Pascal-like system implementation language [7],[15], Interlisp, a sophisticated Lisp implementation [5],[20], and Smalltalk, an object-oriented language for experimental programming [8]. A pipelined instruction fetch unit (IFU) in the Dorado fetches bytes from such a stream, decodes them as instructions and operands, and provides the necessary control and data information to the emulator microcode in the processor; it is described in another paper [11]. Further support for fast execution comes from a very fast microcycle–60 ns–and a microinstruction set powerful enough to allow interpretation of a simple byte code in a single microcycle; these are described in a paper on the Dorado processor [12]. There is also a cache [2],[13] which has a latency of two cycles, and which can deliver a 16-bit word every cycle.

Another major goal is to support high-bandwidth input/

0018-9340/81/1000-0715$00.75