MATHEMATICS OF OPERATIONS RESEARCH
Vol. xx, No. x, Xxxxxxx 200x, pp. xxx–xxx
ISSN 0364-765X | EISSN 1526-5471 |0x|xx0x|0xxx

DOI 10.1287/moor.xxxx.xxxx

Lex-Optimal On-Line Multiclass Scheduling with Hard Deadlines

Bruce Hajek
University of Illinois at Urbana-Champaign
email: b-hajek@uiuc.edu www.uiuc.edu/~b-hajek

Pierre Seri
University of Illinois at Urbana-Champaign
email: p-seri@uiuc.edu

On-line scheduling of unit-length packets with hard deadlines by a single server in slotted time is considered. First, the throughput optimal scheduling policies are characterized. Then multiclass packets are considered in which each packet has an M-bit class identifier, and a new optimality property called lex-optimality (short for lexicographic optimality) is defined for on-line scheduling policies. Lex-optimality is a hierarchical sequence of M throughput optimality properties. The lex-optimal policies that do not drop packets early are characterized. Both characterizations involve identification of a "no-regret subset" of the set of packets available for scheduling in a given slot.

A lex-optimal scheduling algorithm is presented with complexity per packet $O(MB)$, where M is the log of the number of priority classes, and B is the maximum buffer size. The algorithm requires no more packets to be buffered than any on-line throughput optimal scheduling policy. Simulation results are presented which illustrate that lex-optimality combines elements of pure priority and nested priority scheduling.

Key words: on-line scheduling; priority; deadlines; multiclass queues; competitive optimality

MSC2000 Subject Classification: Primary: 90B35 (scheduling theory, deterministic ; Secondary: 68M20 (Performance evaluation; queueing; scheduling)

OR/MS subject classification: Primary: Queues/Priority ; Secondary: Queues/Algorithms

History: Received: September 9, 2000; revised: March 19, 2002 and September 18, 2004.

1. Introduction. The demand for real-time applications and systems has grown markedly in recent years. The growth is fueled by the remarkable increase in the processing speed of microchips. The high processing speed allows for the implementation of real-time applications with tighter timing constraints. As a result, real-time applications and systems are being developed, or are available, for industrial areas as diverse as transportation, robotics, manufacturing, defense, and telecommunications [6, 12]. While the literature on real-time scheduling is extensive [2, 17, 18], on-line scheduling of multi-class packets with deadlines remains to be fully understood.

We consider on-line scheduling of unit-length packets with hard deadlines by a single server in slotted time. We adopt the use of the word "packet" in this context, as is common in the literature on scheduling in communication networks [3, 4, 9, 14, 16], whereas the word "job," "task," or "call" is used in other literature. By "hard deadlines" is meant that if a packet is not scheduled before its deadline, it is dropped from the system. By "on-line" is meant that whatever the arrivals up to the current time slot, the future arrivals are arbitrary and no knowledge of the future arrivals is available to the scheduler. Policies based on the assumption of periodic arrival streams, such as the Rate Monotonic policy of Liu and Layland [10], are not on-line by this definition, since periodicity yields prior knowledge about future arrivals. The most relevant prior work for the setting we consider is based on weighted throughput and competitive optimality. The relationship of this work to ours is discussed at the end of Section 2.

A scheduling policy is *throughput optimal* if for every $k \geq 1$ it schedules at least as many packets in the first k slots as any other scheduling policy. The Earliest Deadline First (EDF) policy schedules, in each slot with packets available, a packet with earliest deadline. Jackson (1955) established that EDF is throughput optimal. A proof by induction is easy: At the end of each slot, EDF leaves in the system a

set of packets that dominates the set left in the system by any other nonidling scheduling policy.

A key motivation of this paper is that EDF is not the only throughput optimal policy. For example, suppose in the current time slot there are three packets available for scheduling: packet p_a which will expire at the end of the slot if not scheduled in the slot, and packets p_b and p_c which will expire at the end of the next slot. The EDF policy specifies that packet p_a is to be scheduled. However, no matter what the future arrivals are, at most two of the three packets can ever be scheduled. Thus, without loss of throughput optimality, p_b, which may be more valuable than p_a, could be scheduled in the current slot instead of p_a. Policy S-OPT of Ling and Shroff [9] is an on-line throughput optimal policy different from EDF.

Unless we specify otherwise, scheduling policies are permitted to drop some packets strictly before their deadline. Such early dropping of packets can be useful for reducing the required buffer space, reducing the time until possible negative acknowledgments are sent, or reducing the computational complexity of scheduling.

The first contribution of the paper is to characterize the on-line, throughput optimal scheduling policies. This contribution indicates how much leeway is available for designing scheduling policies without sacrificing throughput optimality. The characterization involves the identification of a "no-regret subset" of the set of packets available for scheduling in a given slot. A policy that doesn't drop packets early is throughput optimal if and only if, whenever it has packets available to schedule in a slot, it schedules one from the no-regret subset. In general an on-line policy is shown to be throughput optimal if whenever it drops a packet early, it doesn't decrease the number of packets it could schedule in the future in the absence of future arrivals, and whenever it has packets available for scheduling it schedules one from the no-regret subset.

The second contribution of the paper is to present a new optimality criterion, called lex-optimality, for an on-line policy that schedules multiclass packets, and to present a characterization of all no-early-dropping lex-optimal scheduling policies. It is assumed that each packet carries an M bit class identifier in its header, for some fixed $M \geq 1$. Lex-optimality is defined by a hierarchical sequence of throughput optimality properties. The lex-optimal policies that do not drop packets early are characterized. The characterization also involves a "no-regret subset" of the set of packets available for scheduling in a given slot.

The final contribution of the paper is to present a lex-optimal scheduling algorithm, called D_{lex}. Algorithm D_{lex} has complexity per packet $O(MB)$, where M is the log of the number of priority classes, and B is the maximum buffer size. The algorithm requires no more packets to be buffered than any other on-line throughput optimal scheduling policy.

The remainder of this paper is organized as follows. Section 2 presents the three contributions just described, as well as a discussion of the related work which is based on the use of weighted throughput as an objective function. Sections 3 and 4 present the proofs of the theorems in Section 2. Section 5 presents simulations for a lex-optimal scheduling policy, and can be read independently of Sections 3 and 4. The simulations illustrate that lex-optimal scheduling combines elements of pure priority and nested priority scheduling. Closing remarks are given in Section 6.

2. Main results.

2.1 Throughput optimal policies. Consider a single server with its associated buffer, multiplexing several real-time input streams of packets into a single output stream. Let time be slotted, with slot k the time interval from k to $k+1$, and suppose at most one packet can be scheduled per time slot. Adopt the *early arrival model* of Hunter [7], so that packets arrive early during a time slot, are stored in the buffer, and may be scheduled in the time slot of arrival. Let each packet p arrive with an integer deadline $p.d$. If p arrives in slot k, it is either scheduled or dropped in one of the slots $\{k, k+1, \ldots, p.d-1\}$. Once scheduled or dropped, a packet leaves the system and doesn't return. Say a packet p is dropped early if p is dropped before slot $p.d-1$. The *laxity* $p.l$ of a packet p, in a time slot k, is the amount of time left before the packet expires, given by $p.l = p.d - k$. If the current time slot is k and p is in the system in slot k then p can be scheduled in any of the $p.l$ time slots $k, \ldots, k + p.l - 1$. For example, if the current time slot is 9, then a packet p with deadline 12 has laxity 3, and can be scheduled in any of the time slots 9, 10, or 11. The laxity of a packet in the system at any given time is at least 1, since packets are

dropped once their deadlines are reached.

An *arrival sequence* $\mathcal{A} = (A_1, A_2, \ldots)$ consists of disjoint sets of packets $A_1, A_2, \ldots$, with A_k denoting the set of packets arriving in slot k. A scheduling policy maps each arrival sequence into a sequence of packets scheduled and dropped in the time slots. Given an arrival sequence $\mathcal{A}$, a scheduling policy π, and a time slot k, with $k \geq 1$, let $S_k(\pi, \mathcal{A})$ denote the set of packets that are available for scheduling in the slot, and let $Q_k(\pi; \mathcal{A})$ denote the set of packets that are dropped during the slot. When there is no ambiguity, the reference to the scheduling policy, time slot, and arrival sequence may be dropped; therefore, the notation S denotes the set of available packets and Q denotes the set of packets dropped.

During a particular time slot, for the purposes of scheduling packets from some set, it may be that only the laxities of the packets are important. In such cases we use the expression, *set of packets with laxities*, to stress the fact that only the laxities of the packets matter; the time slot in which the packets are observed is irrelevant, and is assumed to be the "current" time slot.

Let A be a set of packets with laxities. If at least one packet in A has laxity l, let the *excess index* $i(l)$ of l be given by $i(l) = l - |\{p \in A : p.l \leq l\}|$, where "$|F|$" denotes the cardinality of a set F. Thus $i(l)$ indicates the excess number of time slots available for laxity l, in the sense that if $i(l) \geq 0$, then all the packets with laxity less than or equal to l could be scheduled by their deadline, with $i(l)$ slots not used. On the other hand, if $i(l)$ is negative, then $-i(l)$ is the minimum number of packets from A with laxity less than or equal to l which must eventually be dropped. Let $i^* = \min\{i(l) : l \in A\})$, and let l^* be the minimum laxity in A that has excess index i^*. Define the subset $\Phi(A)$ of A to be the set of packets in A that have laxity less than or equal to l^*. For example, if the multiset of laxities in A is equal to $\{3, 3, 5, 7, 7, 7, 9\}$, i.e, if A contains two packets with laxity 3, one packet with laxity 5, three packets with laxity 7, and one packet with laxity 9, then $i(3) = 1$, $i(5) = 2$, $i(7) = 1$, and $i(9) = 2$. Thus, $i^* = 1$, $l^* = 3$, and $\Phi(A) = \{3, 3\}$. If the minimum excess index i^* for A is positive, then all packets of A can be scheduled by their deadlines. Call A *schedulable* in such case. If i^* is negative, then i^* is the minimum number of packets from $\Phi(A)$, and also the minimum total number of packets from A, which must eventually be dropped. We call the subset $\Phi(A)$ of A the "no-regret subset" of A for the following reason.

THEOREM 2.1 *An on-line, no-early-dropping scheduling policy is throughput optimal if and only if the policy schedules a packet in $\Phi(S)$ whenever S is not empty.*

Theorem 2.1 clearly implies that EDF is throughput optimal: by its definition $\Phi(S)$ contains all packets of S that have the minimum laxity in S, or, equivalently, all packets of S with the earliest deadline in S. However, Theorem 2.1 concerns only no-early-dropping scheduling policies, so it does not establish the throughput optimality of scheduling policies that may drop some packets early, such as S-OPT of Ling and Shroff [9], or policy D_{lex}, which is described later. A characterization of the on-line, throughput optimal scheduling policies is given next.

Let A be a set of packets with laxities, and τ be a subset of $\{1, 2, \ldots\}$. Set A *can cover* τ if there exists an assignment of packets in A to elements in τ so that all elements in τ are assigned a packet of A; no packet of A is assigned to more than one element of τ; and, finally, if a packet p in A is assigned to an element t of τ, then $p.l \geq t$. For example, if the multiset of laxities in A is $\{1, 3, 5, 6\}$, then A can cover $\{1, 4, 6\}$, but cannot cover $\{1, 7\}$. Note that a set A of packets with laxities is *schedulable* if and only if A covers $\{1, \ldots, |A|\}$.

If a set of packets with laxities C can cover every subset τ of $\{1, 2, \ldots\}$ that A can cover, then say C *dominates* A, and write $C \succeq A$. If $C \succeq A$, and $A \succeq C$, then say A and C are *equivalent*, and write $A \equiv C$.

The following theorem characterizes the on-line, throughput optimal scheduling policies. Given a set A and a subset B of A, write $A - B$ for the complement of B in A. Also, if $a \in A$ write $A - a$ for $A - \{a\}$.

THEOREM 2.2 *An on-line scheduling policy is throughput optimal if and only if the following is true in every time slot: if S is not empty then $S \equiv S - Q$ and the policy schedules a packet in $\Phi(S - Q)$. Furthermore, for any arrival sequence $\mathcal{A}$ and slot k, an on-line throughput optimal policy π minimizes the buffer occupancy $|S_k(\pi, \mathcal{A}) - Q_k(\pi, \mathcal{A})|$, over all such policies, if and only if $S_k(\pi, \mathcal{A}) - Q_k(\pi, \mathcal{A})$ is schedulable.*

Thus, a throughput optimal policy can drop some packets early only if the dropping does not reduce the ability of the set of packets in the system to cover sets of integers. Furthermore, in order to achieve minimum buffer occupancy, in a time slot, over all on-line, throughput optimal scheduling policies, an on-line, throughput optimal scheduling policy must drop enough packets so that $S - Q$ is schedulable, subject to the constraint $S - Q \equiv S$.

The "if" halves of Theorems 2.1 and 2.2 are also true for off-line throughput optimal policies. However, since off-line throughput optimal scheduling policies have more leeway than on-line throughput optimal policies, the "only if" halves of the theorems are not true for off-line scheduling policies.

The following simple greedy algorithm finds a subset Q of S such that $S-Q \equiv S$, and it simultaneously constructs the latest maximal cardinality set τ^* of slots that the packets within S can cover. Process the packets of S one at a time in an arbitrary order. Process the jth packet as follows. If each integer from one up to the laxity of the packet was already assigned one of the first $j-1$ packets processed, add the packet to Q. Otherwise assign the packet to the largest unassigned integer less than or equal to the laxity of the packet, and do not add the packet to Q. Finally, let τ^* be the set of integers that are assigned a packet during the execution of the algorithm. This algorithm can easily be used to devise an on-line, throughput optimal scheduling policy as follows. In each time slot, first use the the algorithm to construct $S - Q$, then compute $\Phi(S - Q)$, and schedule a packet from it. However, such an approach does not lead to a low complexity throughput optimal scheduling policy since it doesn't retain a data structure from slot to slot.

2.2 Lex-optimal policies. Consider scheduling of multiclass packets. Suppose for some $M \geq 1$ that each packet p has an M-bit class identifier $p.c$. The first bit of $p.c$, $p.c[1]$, is the most significant bit. For any k, such that $1 \leq k \leq M$, refer to $p.c[k]$ as p's kth bit. The class of a packet can be viewed as an integer given in binary integer representation, so that two classes can be compared using the order on integers. Given two packets p and q, p has priority higher than q, if the integer value of $p.c$ is smaller than that of $q.c$, which is denoted $p.c < q.c$. Equivalently, $p.c < q.c$ if the vector $p.c$ is lexicographically smaller than $q.c$. Packet p has priority higher than or equal to that of q, if $p.c < q.c$ or $p.c = q.c$, which is denoted $p.c \leq q.c$. Lex-optimality, a new criteria for an on-line scheduling policy, is defined next.

Definition *A scheduling policy is order 1 lex-optimal provided the policy is on-line, and whatever the arrival sequence, the policy schedules in time slots* 1 *through* k *at least as many packets that have first bit equal to* 0 *as any on-line scheduling policy, for any* $k \geq 1$. *For any* j *such that* $2 \leq j \leq M$, *a scheduling policy is* order j *lex-optimal provided the policy is order* $j-1$ *lex-optimal, and whatever the arrival sequence, the policy schedules in time slots* 1 *through* k *at least as many packets that have* j*th bit equal to* 0 *as any order* $j-1$ *lex-optimal scheduling policy, for any* $k \geq 1$. *An order* M *scheduling policy is simply referred to as a lex-optimal scheduling policy.*

As a first step towards understanding the definition of lex-optimality, consider the two scenarios pictured in Figure 1. For the first scenario there are two packets available for scheduling in a time slot:

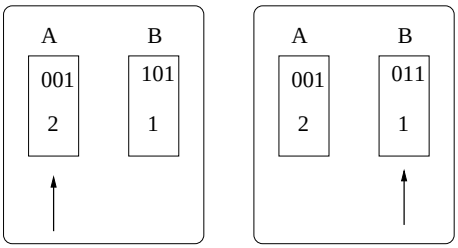

Figure 1: Two samples of lex-optimal packet selection

packet A with laxity 2 and class identifier 001 and packet B with laxity 1 and class identifier 101. Note that the more valuable packet, A, has higher laxity. Which packet will be scheduled under a lex-optimal strategy? The answer is that A must be scheduled, since of the two packets it is the only one with first class bit equal to 0. Thus any lex-optimal policy, in fact any order 1 lex-optimal scheduling policy, must schedule packet A in the slot.

The second scenario is the same except that B has class 011. Again A is the more valuable packet,

but in this scenario, any lex-optimal policy, in fact any order 1 lex-optimal policy, must schedule B in the slot. Otherwise, if there happens to be no arrivals in the next slot, the policy would have no packet with first class bit 0 to schedule in the next slot. Note that in both scenarios, the more valuable packet has a larger laxity. In the first scenario, the difference in priorities is large enough that the more valuable packet is scheduled first, while in the second scenario the difference in priorities is small enough that the packet with the earlier deadline is scheduled first.

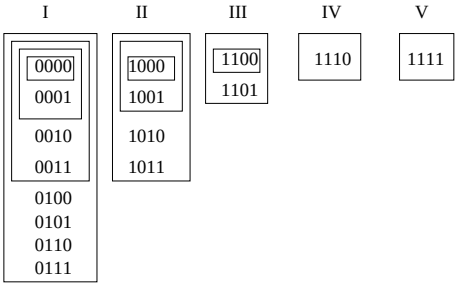

Figure 2: The relationship among priority classes for $M = 4$

Figure 2 illustrates the relationship among priority classes in more detail, in the particular case $M = 4$. The 16 priority classes, are grouped into five subgroups, numbered by Roman numerals. Any lex-optimal policy treats the five subgroups of packets in a static priority fashion: it must always schedule a packet from the first nonempty subgroup, except that scheduling group V packets is optional. For example, packets in groups III, IV, and V have no effect on packets in groups I and II. The boxes pictured within the subgroups indicate that lex-optimal policies treat packets within a subgroup in a nested priority fashion. For example, subject to maximizing throughput for group I packets, a lex-optimal policy maximizes throughput for packets with class in {0000, 0001, 0010, 0011}, and subject to that, it maximizes the throughput for packets with class in {0000, 0001}, and subject to that it maximizes the throughput for packets with class 0000. Additional insight into the nature of the definition of lex-optimality is provided in Section 5, which illustrates the performance of lex-optimal scheduling.

Turn next to the characterization of the no-early-dropping lex-optimal scheduling policies. The approach is similar to that adopted in the characterization of on-line, no-early-dropping throughput optimal scheduling policies: a "no-regret subset" of S for lex-optimality is identified.

Given a set of a packets A with laxities and possibly different class identifiers, let $\tilde{A}$ denote the of packets in A that have class identifiers with at least one bit equal to 0. Also, for $1 \leq n \leq M$, let $A^{(n)}$ denote the set of packets in A that have nth bit equal to 0, and let the subset $\Gamma^n(A)$ of A be defined as follows:

$$\Gamma^1(A) \quad = \quad \begin{cases} \Phi(A^{(1)}), & \text{if } A^{(1)} \neq \emptyset; \\ A, & \text{otherwise;} \end{cases}$$

and for $2 \leq n \leq M$,

$$\Gamma^n(A) \quad = \quad \begin{cases} \Phi(\Gamma^{n-1}(A)^{(n)}), & \text{if } \Gamma^{n-1}(A)^{(n)} \neq \emptyset; \\ \Gamma^{n-1}(A), & \text{otherwise.} \end{cases}$$

Define $\Gamma(A)$ to be $\Gamma^M(A)$, and call $\Gamma(A)$ the "no-regret subset" of A for lex-optimality, because of the following theorem.

THEOREM 2.3 *A no-early-dropping scheduling policy is lex-optimal if and only if the policy is on-line, and schedules a packet in* $\Gamma(S)$ *whenever* $\tilde{S}$ *is not empty.*

Theorem 2.3 is similar to Theorem 2.1, except that the necessary and sufficient condition is used whenever $\tilde{S}$, instead of S, is not empty. The definition of lex-optimality is such that packets with class identifiers equal to the all 1's binary vector can be dropped from the system without loss of optimality. These packets are ignored in the definition of lex-optimality, which is why the necessary condition of Theorem 2.3 is active only if S contains a packet that is not from the all 1's class. Theorem 2.3 not only characterizes the no-early-dropping lex-optimal policies, it also shows that lex-optimal policies exist.

An intuitive explanation for the definition of $\Gamma(A)$ follows. Theorem 2.3 implies that if $1 \leq n \leq M$ then a policy is order n lex-optimal if and only if the policy is on-line, and schedules a packet in $\Gamma^n(S)$ whenever S contains a packet with at least one of its first n bits equal to zero. Therefore, the hierarchical nature of the definition of lex-optimality requires that $\Gamma^1(A) \supset \Gamma^2(A) \supset \cdots \supset \Gamma^M(A)$. If all packets in $\Gamma^{n-1}(A)$ have nth bit equal to 1, then order n lex-optimality treats all packets in $\Gamma^{n-1}(A)$ the same, so it is reasonable that $\Gamma^n(A) = \Gamma^{n-1}(A)$ in this case. On the other hand, if some packets in $\Gamma^{n-1}(A)$ have nth bit equal to 0, then order n lex-optimality would require scheduling one of those packets. Furthermore, application of Φ to this set of packets to take into account the effect of laxities is reasonable, given Theorem 2.1.

In contrast to the results in the single class model, the set of all lex-optimal scheduling policies (which are permitted to drop packets early) is not characterized in this paper. Rather, in the next section we describe a simple and efficient lex-optimal scheduling algorithm, Algorithm D_{lex}, with the minimum buffer requirements.

2.3 A lex-optimal scheduling algorithm. So far we have discussed scheduling policies, which can be rather complex mappings which specify packets to be scheduled or dropped early. In contrast, in this section, we consider a particular lex-optimal scheduling algorithm. By scheduling algorithm we mean a computational recipe for implementing a scheduling policy. Our objective is Algorithm D_{lex}. However, first we present a related single-class throughput optimal algorithm called Algorithm D_s, for the purpose of illustrating in a simple setting the ideas underlying D_{lex}.

Fix the data structure of Algorithm D_s as follows. Assign each packet p an additional parameter $p.v$, called the virtual laxity of p. Maintain the packets in a linear buffer B. If there are $L \geq 1$ packets in the buffer, let $b_1, b_2, \ldots, b_L$ denote the packets, ordered from the head of B to the tail. In contrast to the algorithms of Ling and Shroff [9] and Peha and Tobagi [14], the packets occupy consecutive positions of B, starting from the first.

Figure 3 illustrates Algorithm D_s processing a new arrival and then scheduling a packet. Initially

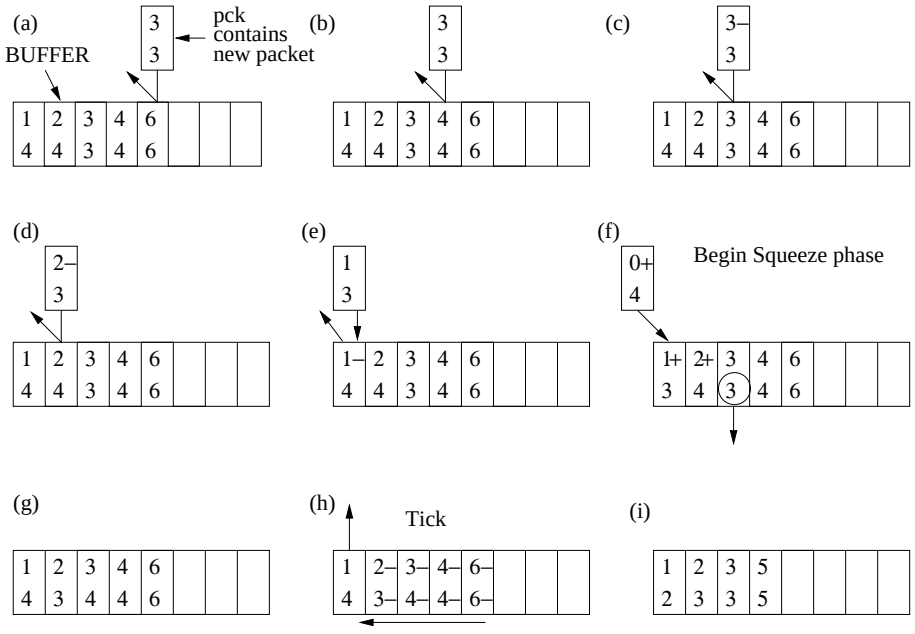

Figure 3: A sample run for Algorithm D_s.

there are five packets in the buffer with virtual laxities 1,2,3,4,and 6, written at the top of the packets and laxities 4,4,3,4 and 6, written at the bottom of the packets. The new packet has laxity, and hence initial virtual laxity, 3. The new packet begins at the tail of B and slides past the two packets with larger laxities, as shown in (a)-(c) of the figure. Upon meeting the packet with the same laxity, there is an optional swap between the two packets. In the example shown, there is no swap at that time. As the packet continues toward the head of B, its virtual laxity is decreased from 3 to 2. The minus sign

in part (c) of Figure 3 indicates that said decrease is about to happen. Such decrease occurs whenever the two packets meeting have the same virtual laxity. This process continues until the head of B has been reached, resulting in the configuration shown in part (f). The second half of the insertion process is the Squeeze phase. The outstanding packet is inserted in the first position of B. The displaced packet from the first position moves to the second position, and so on. However, the packet in the third position cannot be moved to the fourth position since it has laxity 3. That packet is hence dropped from B, as indicated in part (f). If a packet is dropped, as in this example, the virtual laxities of the packets preceding it are each increased by one. This completes the insertion algorithm, and the result is shown in part (g). Parts (h) and (i) of the figure indicate that when a packet is to be scheduled, it is taken from the head of the buffer. The other packets move one position towards the head of the buffer, and their laxities and virtual laxities decrease by one for the next time slot.

Algorithm D_s stems from a simple observation: when two packets have equal laxity, it does not affect throughput optimality to reduce the laxity of one packet by 1, since only one packet can be scheduled per time slot. Virtual laxities are introduced because the laxities cannot be modified by a scheduling algorithm.

Figure 4 shows the pseudo-code for Algorithm D_s. The set of instructions of Figure 4 are executed at the beginning of each time slot. Algorithm D_s is divided into two procedures: Insert and Tick. Procedure Insert consists of two phases, the compete and squeeze phases, and is called for each new arrival.

Algorithm D_s {
for (each new arrival p) Insert(p);
Tick; }
Procedure Insert(p **)** {
Compete phase:
$pos = L =$ number of packets in B; $p.v = p.l$; $pck = p$;
while ($pos \geq 1$)
    if $pck.v > b_{pos}.v$, swap pck and b_{pos};
    else if $pck.v == b_{pos}.v$, swap pck and b_{pos} if desired (the swap is optional), and decrease $pck.v$ by 1;
    $pos = pos - 1$;
Squeeze phase:
$j = 0$;
while ($pck.l > j$ and $j \leq L - 1$)
    swap pck and b_{j+1}; $j = j + 1$;
if ($j == L$ and $pck.l > L$) let $b_{L+1} = pck$;
else for $i \in \{1, \ldots, j\}$ increase $b_i.v$ by 1 and drop pck; }
Procedure Tick() {
if B is not empty, schedule b_1, shift the remaining packets in B, if any, towards the head of B, and decrease all laxities and all virtual laxities by 1. }

Figure 4: Algorithm D_s, for updating B and scheduling in one time slot.

During the compete phase the variable pos counts from L down to zero. The variable pck represents a packet, initially the new arrival, competing for insertion into the buffer at position pos. The squeeze phase shifts some—possibly all—packets of B by one slot towards the tail of B, and inserts pck in the slot freed at the head of B. A packet may be dropped during the shift to ensure that no packet is at a position of B that is larger than its laxity, and, therefore, that if the packets were scheduled according to their position in B, b_1 first, then no packet would expire before receiving service.

In each time slot, once procedure Insert has been executed for all new arrivals, procedure Tick is called. Procedure Tick schedules packet b_1, shifts the remaining packets in B, if any, by one position towards the head of B, and decreases these packets' virtual laxities by 1. The shift by one position ensures that the first position of B, which is freed once packet b_1 is scheduled, is filled at the beginning of the next time slot. The laxities and virtual laxities of remaining packets are decreased by 1 to reflect the passing of time.

Algorithm D_s is meant to be as close as possible to the new multiclass algorithm, Algorithm D_{lex}, which is given later in this section. Therefore, the squeeze phase of D_s is a single class version of the squeeze phase of D_{lex}. To obtain a simpler, on-line, throughput optimal dropping policy you could modify the squeeze phase of D_s as follows: when *pos* reaches 0, if $pck.v \geq 1$, run the squeeze phase of D_s (and no packet is dropped); on the other hand, if $pck.v = 0$, drop *pck*.

The optimality property of Algorithm D_s is stated formally in the following theorem.

THEOREM 2.4 *Algorithm D_s is an on-line, throughput optimal scheduling algorithm which minimizes the buffer occupancy over all on-line, throughput optimal scheduling policies.*

At this point we are prepared to discuss the lex-optimal algorithm, Algorithm D_{lex}. Maintain a data structure for Algorithm D_{lex} as follows. Assign each packet p a vector $p.v$ of M virtual laxities, such that the jth virtual laxity of a packet p, written $p.v[j]$, is finite if and only if the jth bit of $p.c$ is 0. Maintain the packets in a sorted buffer B, as in Algorithm D_s. Figure 5 illustrates how D_{lex} processes a new arrival, and schedules a packet, for a system with $M = 4$. The first four rows of numbers indicate

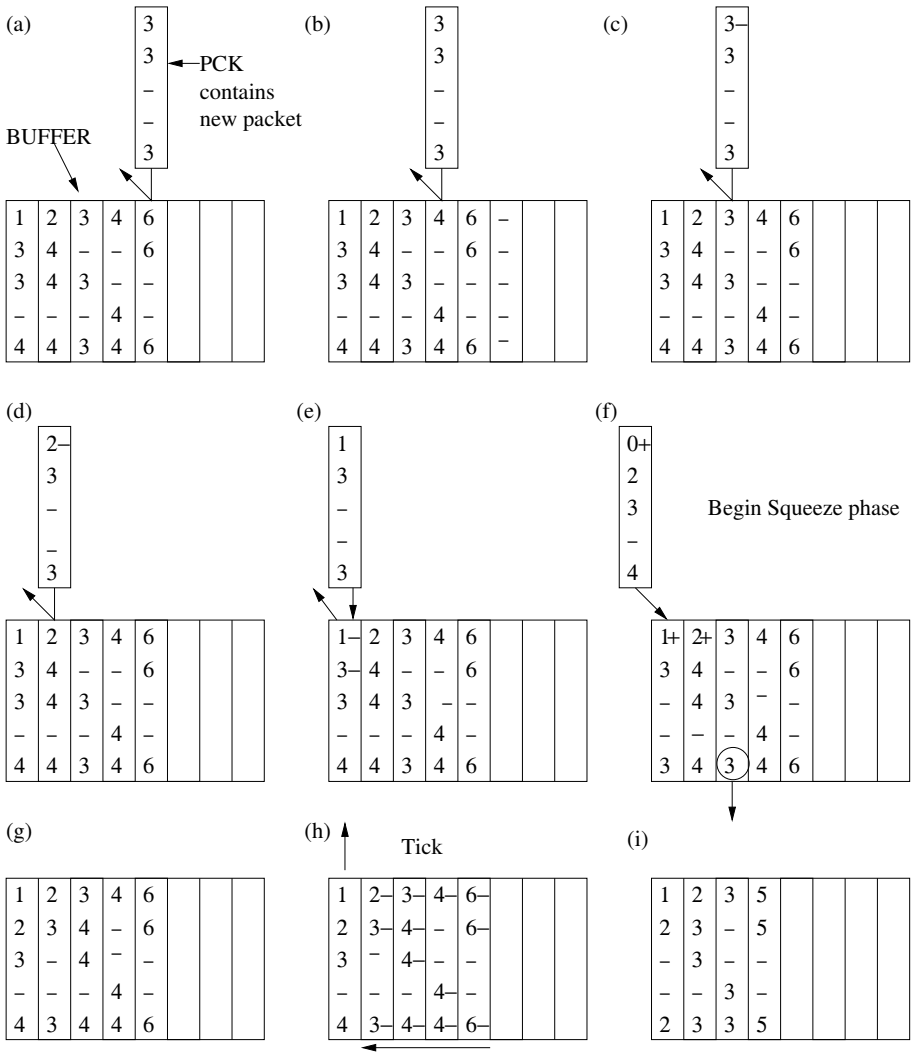

Figure 5: A sample run for Algorithm D_{lex}.

the four virtual laxities for each packet, and the last row indicates the laxity of each packet in the buffer. The dashes in the figure indicate entries that are infinity. The new arrival has laxity 3, and class 0011. Its corresponding initial virtual laxity vector is therefore set to $33\infty\infty$ as shown in part (a). Note that the first virtual laxities and laxities of the packets in the buffer and the new arrival in Figure 5 are the same as the virtual laxities and laxities of the packets in the example of Figure 3. The insertion process

is also similar to that of D_s. When two packets are compared, the one with the lexicographically smaller virtual laxity vector is promoted, and continues towards the head end of the buffer. If the initial portion of the virtual laxity vectors match, the coordinates in the common prefix are each decreased by one as a result. Similarly, when a packet is dropped, as indicated in part (f), the coordinates of the common prefix of virtual laxities of the preceding packets are all increased by one. The Squeeze phase then terminates the insertion process, much the same as for Algorithm D_s.

Pseudo-code for Algorithm D_{lex} is given in Figure 6. The following notation is used. Given a vector u, the dimension of u is denoted by $|u|$. Further, for any vectors u and u', $u \cap u'$ is equal to $NULL$ if $u[1] \neq u'[1]$; otherwise $u \cap u'$ is the common prefix, from $u[1]$, of u and u' with the largest dimension. By definition, $|NULL| = 0$. For example, if $u = (05748)$ and $v = (0537)$, $u \cap v = (05)$; thus, $|u \cap v| = 2$.

Algorithm D_{lex} {
for (each new arrival p) M-Insert(p);
M-Tick(); }
Procedure M-Insert(p) {
$pos = L =$ number of packets in B;
for ($i \in \{1, \ldots, M\}$)
    if $p.c[i] == 0$, $p.v[i] = p.l$; otherwise, $p.v[i] = \infty$.
$pck = p$;
Compete phase:
while ($pos \geq 1$)
    if $pck.v$ is (lexicographically) larger than $b_{pos}.v$, swap pck and b_{pos};
    else if $pck.v == b_{pos}.v$, swap pck and b_{pos} if desired (the swap is optional).
    Decrease the first $|b_{pos}.v \cap pck.v|$ coordinates of $pck.v$ by 1;
    $pos = pos - 1$;
Squeeze phase:
$j = 0$;
while ($pck.l > j$ and $j \leq L - 1$)
    swap pck and b_{j+1}; $j = j + 1$;
if ($j == L$ and $pck.l > L$) let $b_{L+1} = pck$;
else for $i \in \{1, \ldots, j\}$ increase the first $|b_j.c \cap pck.c|$ coordinates of $b_i.v$ by 1, and drop packet pck; }
Procedure M-Tick() {
if B is not empty, schedule b_1, shift the remaining packets in B, if any, towards the head of B, and decrease all laxities and all virtual laxities by 1. }

Figure 6: Algorithm D_{lex}, for updating B and scheduling in one time slot.

As indicated in Figure 6, Algorithm D_{lex} is also divided into two procedures: M-Insert, which is executed for each new arrival, and M-Tick, which is executed once in Algorithm D_{lex}, after all calls to M-Insert, if any. Procedure M-Insert consists of two phases: the compete and squeeze phases, which are delimited by the labels in Figure 6.

The compete phase of procedure M-Insert is similar to that of procedure Insert. For each new arrival pck, the compete phase determines a relative order on packets in B and the new arrival. During the execution of the compete phase, the variable pos goes from L, the number of packets in B, to 0. For each value of pos, the virtual laxity vectors of pck and of b_{pos} are compared; the packet with lexicographically smaller virtual laxity vector is promoted towards the head of B. Packet pck competes with the packet at position pos of B. If $pck.v$ and $b_{pos}.v$ have a common prefix, then the virtual laxities of the packet promoted, from the first virtual laxity to the last common virtual laxity, are decreased by 1.

The squeeze phase of procedure M-Insert starts once variable pos reaches 0: the packets in B, if any, are shifted by one position towards the tail, and pck is inserted at the head of B. During the shift of packets in B, a packet is dropped if its position in B is equal to its laxity, and the virtual laxities of the packets preceding the dropped packet are adjusted. The dropping ensures that if the packets in B were scheduled according to their order in B, starting from b_1, then no packet would expire before being scheduled. Also, the adjustment of virtual laxity vectors in case of dropping is necessary to correct the excess decrease in virtual laxities that might have been caused by the dropped packet. Procedure Tick

is executed after all new arrivals have been considered for insertion in B. The packet b_1 is scheduled, and the remaining packets in B, if any, are shifted by one position towards the head of B, and their virtual laxities are decreased by 1. The shift by one position is made to ensure that the positions of B are occupied starting from position 1. The laxities and virtual laxities are decreased to account for the passing of time.

Algorithms D_{lex} and D_s are based on the observation that virtual laxities ought to be distinct in a set of packets. Since lex-optimality requires that the throughput of multiple sets of packets be maximized, a vector of virtual laxities is needed in D_{lex}. The comparison of the virtual laxity vectors of *pck* and b_{pos} in Algorithm D_{lex} is lexicographical, reflecting the recursive nature of lex-optimality.

We state the lex-optimality of D_{lex} as a theorem. The proof is given in Section 4.

THEOREM 2.5 *Policy D_{lex} is lex-optimal, and has buffer occupancy lower than or equal to that of any on-line, throughput optimal scheduling policy.*

Theorem 2.5 is remarkable, given the simplicity of Algorithm D_{lex}. This algorithm has complexity $O(ML)$ per new arrival, where L is the number of packets in B at the time of arrival. The value of M is expected to be small in many applications: the number of priority classes grows exponentially with M. For example, when M is equal to 5, 32 different priority classes are allowed. This makes policy D_{lex} attractive for multiclass scheduling in the model considered herein.

2.4 Related work based on weighted throughput. As mentioned in the introduction, there is a large literature on scheduling with hard deadlines. A large portion of the literature either assumes that the arrivals are constrained in such a way that no packet has to be dropped before its deadline, or the value per unit service time is the same for all packets, or only off-line algorithms are considered.

The previous work concerning algorithms for on-line multiclass scheduling with hard deadlines and possible overload, focuses on competitively optimal algorithms with weighted throughput as an objective function. A key paper in this area is the one of S. Baruah et al. [1], which follows pioneering work on scheduling heuristics by Locke [11]. Following Locke, Baruah et al. [1] allow packets of any length, and suppose that packets are labeled with different numerical values. The value density of a packet is the ratio of its value to its length. Remarkably enough, it is shown that any on-line scheduling policy cannot, for a worst case arrival sequence depending on the policy, achieve cumulative value better than a factor $1/(1+\sqrt{m})^2$ of the value of the best off-line scheduling policy, where m is the ratio of the largest to the smallest value density. This factor is thus an upper bound on the "competitive ratio" of any on-line scheduling policy. This number is surprisingly small for moderate or large m. For example, if $m = 10$, so that some packets are 10 times more valuable per unit length than others, then no on-line scheduling policy has competitive ratio better than 5.6%. The bound on the competitive ratio is 0.25 if all packets have the same value per unit length ($m = 1$), but are still allowed to have different lengths. Further, in that case, the bound is achievable [1].

Recently Hajek [5] showed that for on-line scheduling of unit-length packets in slotted time, the competitive factor of any algorithm is less than or equal to the inverse golden ratio, $(\sqrt{5}-1)/2 \approx 0.61804$. This upper bound is not very close to one, so the practical performance of competitively optimal policies might not be satisfactory in practical applications. It was also shown that a static priority policy, which simply schedules the largest value available packet in each slot, achieves competitive factor 0.5.

In contrast, our formulation of lex-optimality does not involve numerical weights. In fact, it does not use a single objective function. This is consistent with current efforts for providing different quality of service on the Internet. The main proposals being considered for differential quality of service on the Internet are based on partitioning packets into different classes that are handled differently by routers (see Peterson and Davie [15, Sect. 6.5.3]). Numerical values are not assigned to the packets.

Still, one can view lex-optimal policies as maximizers of an expected weighted throughput, in a limiting sense, as follows. Given a packet p with M-bit class identifier $p.c$, we define the value $p.w$ by

$$p.w = (1 - c[1]) + (1 - c[2])\epsilon + (1 - c[3])\epsilon^2 + \ldots + (1 - c[M])\epsilon^{M-1}$$

where ϵ is a very small number. Suppose the arrival sequence is random with a sufficiently rich probability distribution, let policy π^ϵ be a policy that minimizes the expected sum of values of the packets scheduled

in some specified finite time interval beginning with slot one. Then for any arrival sequence and ϵ sufficiently small, π^ϵ schedules the same packets during the interval as a lex-optimal policy.

This paper concerns on-line policies. The corresponding off-line problem, for the same setting of discrete-time, unit length packets, is rather straight-forward. Virtually all optimality criteria are equivalent. Indeed, suppose the arrival sequence for slots $1, \ldots, N$ is known in advance. Let $p_1, \ldots, p_M$ denote all the packets that will arrive, ordered in decreasing order of priority (so p_1 has the highest priority, p_2 has the second highest priority, etc). Then a single schedule can be found to maximize, simultaneously for all m, the number of packets in $\{p_1, \ldots, p_m\}$ that are scheduled in slots $1, \ldots, N$, [8]. Indeed, any schedule corresponds to a matching on the bipartite graph with the M packets as one set of nodes and the slots $1, \ldots, N$ as the other set of nodes. There is an edge between a packet p and a slot k if k is included in the interval of slots beginning with the arrival slot of p and ending with slot $p_d - 1$, the last slot before the deadline of p. The well known labeling algorithm (see [13]) for finding bipartite matchings can be used to first schedule p_1, then to attempt to schedule p_2, then to attempt to schedule p_3, and so on. Once a packet is scheduled it will remain scheduled in subsequent steps of the labeling algorithm, although its slot assignment can be changed. After m packets have been considered, the maximum possible number of the first m packets will have been scheduled. Peha and Tobagi [14] provided an algorithm which exploits the special structure of the matching problem to provide the (off-line) maximum weighted throughput schedule for M packets using $O(M^2)$ computations.

3. Proofs of the characterization theorems. This section presents the proofs of Theorems 2.1 through 2.3, which characterize no-early-dropping on-line TO policies, on-line TO policies, and no-early-dropping lex-optimal policies, respectively.

3.1 Proofs of Theorem 2.1 and 2.2. Theorem 2.1 is now considered. The following observations show that the analysis can focus on nonidling scheduling policies: first, by definition, a throughput optimal scheduling policy must be nonidling. Second, a scheduling policy is throughput optimal if and only if the policy is nonidling, and schedules as many packets as any nonidling scheduling policy in time slots 1 through k, for any $k \geq 1$. Therefore, the remainder of this section focuses on nonidling scheduling policies. A series of important properties of sets of packets with laxities is now discussed. Each result is stated as a lemma and proved.

Given a set of packets with laxities A, $T(A)$ denotes the same set of packets observed one time slot later: the laxities are decreased by 1, and the packets with laxity 1 in A are dropped. For example, if $A = \{1, 3, 3, 4\}$, then $T(A) = \{2, 2, 3\}$, where the set of packets with laxities A is equated with the multiset of the laxities of packets in A. This makes sense when only laxities of packets in A matter. In like manner, for any $k \geq 2$, $T^k(A)$ is the set of packets A observed k time slots later. As a special case, $T^0(A) = A$ and $T^1(A) = T(A)$.

Given two sets A and C of packets with laxities, the notation $A \sim C$ means that A and C are identical as sets of packets, but may be observed in different time slots: A and C contain the same packets, and if C is the set observed at a later time, then no packets in A have expired at the time C is observed.

LEMMA 3.1 *Let A be a set of packets with laxities, and l be the largest laxity in $\Phi(A)$. Then $T^l(A - \Phi(A)) \sim A - \Phi(A)$, and $T^l(A - \Phi(A))$ is schedulable.*

PROOF. The result is vacuous if $A - \Phi(A)$ is empty; therefore, it is assumed that this set is not empty. Clearly, packets in $A - \Phi(A)$ have laxity greater than l, which implies that $T^l(A - \Phi(A)) \sim A - \Phi(A)$. Let the packets in A be ordered as $p_1, \ldots, p_{|A|}$ so that $p_1.l \leq \ldots \leq p_{|A|}.l$. Let p_r be the last packet of $\Phi(A)$ in the order. Then, by the definition of $\Phi(A)$, l has the minimum excess index over all laxities in A, and the excess index of l is $l - r$. Thus, for any j such that $1 \leq j \leq |A|$, the excess index of $p_j.l$ is greater than or equal to $l - r$. Further, there are at least j packets in A with laxity less than or equal to $p_j.l$; thus, $p_j.l - j$ is greater than or equal to the excess index of $p_j.l$. Hence, $p_j.l - j \geq l - r$, so that $p_j.l \geq l + j - r$. This implies that the packets $p_{r+1}, \ldots, p_{|A|}$ can *all* be scheduled after a delay of l time units. □

LEMMA 3.2 *Let A be a set of packets with laxities, and let p and q be packets of A, such that $p.l \leq q.l$. Then $A - p \succeq A - q$.*

Lemma 3.2 is trivial.

LEMMA 3.3 *Let A and C be two sets of packets with laxities, such that $A \succeq C$. Then, $T(A) \succeq T(C)$. As a consequence, if $A \equiv C$, then $T(A) \equiv T(C)$.*

PROOF. Given a subset τ of $\{1, 2, \ldots\}$, let τ' be the set obtained by increasing each element of τ by 1. Clearly, for any set of packets with laxities D, $T(D)$ can cover τ if and only if D can cover τ'. Therefore, $T(C)$ can cover τ only if C can cover τ', thus, only if A can cover τ', and, consequently, only if $T(A)$ can cover τ. □

LEMMA 3.4 *Let A, C, and D be sets of packets with laxities, such that D is disjoint from both A and C. If $A \succeq C$, then $A \cup D \succeq C \cup D$. Therefore, if $A \equiv C$, then $A \cup D \equiv C \cup D$.*

PROOF. Any set that $C \cup D$ can cover can be partitioned into two subsets: a first subset that C can cover, and a second that D can cover. Then A can cover the first subset, so $A \cup D$ can cover the union of both subsets. □

LEMMA 3.5 *Let A and C be two sets of packets with laxities, let p be a minimum laxity packet in A, and let q be any packet in C. If $A \succeq C$, then $T(A-p) \succeq T(C-q)$. Therefore, if $A \equiv C$ and q is a minimum laxity packet in C, then $T(A-p) \equiv T(C-q)$.*

PROOF. Let τ be a set that $T(C-q)$ can cover, and τ' be the set obtained by increasing each element in τ by 1. Then $C-q$ can cover τ', so C can cover $\tau' \cup \{1\}$. Thus, A can cover $\tau' \cup \{1\}$, and can do so by assigning p to $\{1\}$. This implies that $A-p$ can cover τ', and, therefore, that $T(A-p)$ can cover τ. □

LEMMA 3.6 *Let A be a set of packets with laxities, p and q be packets of A, such that $p \in \Phi(A)$. Then (i) $T(A-p) \succeq T(A-q)$; further, (ii) if $T(A-p) \equiv T(A-q)$, then q is also in $\Phi(A)$.*

PROOF. Part (i) of Lemma 3.6 is considered first. By Lemma 3.3, it suffices to show that $A-p \succeq A-q$. By Lemma 3.2, p can be taken as a packet of largest laxity in $\Phi(A)$, and it can be assumed that $q.l < p.l$. Hence $q \in \Phi(A)$. Let the packets in $\Phi(A)$ be ordered as $p_1, \ldots, p_r$, with $r = |\Phi(A)|$, $p_1.l \leq \ldots \leq p_r.l$, and $p = p_r$. The excess index of $p_r.l$ is less than that of $p_j.l$, for any j, such that: $1 \leq j \leq r$ and $p_j.l < p_r.l$. Thus, for any such j, $p_j.l - j > p_r.l - r$. If $1 \leq j \leq r-1$ and $p_j.l = p_r.l$, then again $p_j.l - j > p_r.l - r$. So for any j, such that $1 \leq j \leq r-1$, $p_j.l \geq p_r.l + j - r + 1$. Hence, packets $p_1, \ldots, p_{r-1}$ can cover $\{\max\{p_r.l - r + 2, 1\}, \ldots, p_r.l\}$. Therefore, $\Phi(A) - p_r$ can cover any set that a subset of $\Phi(A)$ of cardinality $r-1$ can cover. So $\Phi(A) - p \succeq \Phi(A) - q$. Then, by Lemma 3.4 (taking $D = A - \Phi(A)$), $A - p \succeq A - q$. Thus, by Lemma 3.3, $T(A-p) \succeq T(A-q)$.

Part (ii) of Lemma 3.6 follows from Lemma 3.1: packets in $A - \Phi(A)$ are *all* needed to cover $\{l + 1, \ldots, l + |A - \Phi(A)|\}$. □

LEMMA 3.7 *Let A and C be sets of packets with laxities. Let $p \in \Phi(A)$ and $q \in C$. If $A \succeq C$, then $T(A-p) \succeq T(C-q)$. Therefore, if $A \equiv C$ and $q \in \Phi(C)$, $T(A-p) \equiv T(C-q)$.*

PROOF. Let p' be a smallest laxity packet in A. Then, by Lemmas 3.6 and 3.5, $T(A-p) \succeq T(A-p') \succeq T(C-q)$. □

The "if" part of Theorem 2.1 follows from the following proposition.

PROPOSITION 3.1 *Let π be a no-early-dropping scheduling policy that schedules a packet in $\Phi(S)$ whenever S is not empty. Then, for any nonidling policy π', in any time slot, $S(\pi) \succeq S(\pi')$. Thus, whenever policy π' schedules a packet; so does policy π.*

PROOF. By definition, $S_1(\pi) = S_1(\pi')$. For the sake of argument by induction, let $k \geq 1$ and suppose $S_k(\pi) \succeq S_k(\pi')$. If $S_k(\pi') = \emptyset$, then $S_{k+1}(\pi') \subset S_{k+1}(\pi)$, so $S_{k+1}(\pi) \succeq S_{k+1}(\pi')$. On the other hand, if $S_k(\pi') \neq \emptyset$, then $S_k(\pi) \neq \emptyset$. Let p denote the packet scheduled by π and p' denote the packet scheduled by π'. Lemma 3.7, applied to $S_k(\pi)$ and $S_k(\pi')$, yields $T(S_k(\pi) - p) \succeq T(S_k(\pi') - p')$. Finally, Lemma

3.4, with D equal to the set of arrivals in time slot $k+1$, gives the desired result $S_{k+1}(\pi) \succeq S_{k+1}(\pi')$. The proof by induction is complete. □

Proposition 3.1 shows that the "if" part of Theorem 2.1 is valid. Also, the following result is a simple corollary to Proposition 3.1.

COROLLARY 3.1 *Let π and π' be two no-early-dropping scheduling policies that schedule a packet in $\Phi(S)$ whenever S is not empty. In any time slot, $S(\pi) \equiv S(\pi')$.*

The "only if" part of Theorem 2.1 is discussed. Let π be a scheduling policy that, in some time slot k, schedules a packet in $S_k - \Phi(S_k)$. Let l be the largest laxity of $\Phi(S_k)$, and let the subsequent arrivals consist of a single packet of laxity 1 in each of the next $l-1$ time slots and no arrivals in later time slots. Had policy π scheduled a packet in $\Phi(S_k)$, the new arrivals could have been scheduled in the next $l-1$ time slots, and, by Lemma 3.1, l time slots later, policy π would have had the schedulable set $S_k - \Phi(S_k)$. However, since, in time slot k, policy π schedules a packet in $S_k - \Phi(S_k)$, whatever the decisions in the intermediary time slots, l times slots later, policy π has fewer packets than it could have had. This completes the proof of Theorem 2.1.

The proof of Theorem 2.2 is now discussed. The proof of the"if" part of Theorem 2.2 is a minor variation of the proof of the "if" part of Theorem 2.1, and is based on a simple induction on time, as follows. Let π be a scheduling policy such that, in any time slot, $S - Q \equiv S$, and π schedules a packet in $\Phi(S-Q)$ whenever $S - Q \neq \emptyset$, and let π' be an on-line, throughput optimal no-early-dropping scheduling policy. By definition, $S_1(\pi) \equiv S_1(\pi')$. For the sake of argument by induction, let $k \geq 1$ and suppose $S_k(\pi) \equiv S_k(\pi')$. Then, $S_k(\pi) - Q_k(\pi) \equiv S_k(\pi')$, so, by Lemmas 3.4 and 3.7, $S_{k+1}(\pi) \equiv S_{k+1}(\pi')$. The induction argument is complete, showing that $S_k(\pi) \equiv S_k(\pi')$ for all $k \geq 1$. This implies the "if" part of Theorem 2.2.

The "only if" part of Theorem 2.2 is now tackled, using the following remarkable relation between equivalence and cardinality.

LEMMA 3.8 *Let A be a set of packets with laxities, and r be the largest cardinality of any schedulable subset of A. Then, for any subset C of A, $C \equiv A$ if and only if C contains a schedulable set of cardinality r.*

PROOF. The "only if" part of Lemma 3.8 is obvious. For the "if" part, let C be schedulable with cardinality r, and—for the sake of argument by contradiction—let τ be any set of smallest cardinality among the sets that A can cover, but that C cannot cover. Let the elements of τ be $t_1, \ldots, t_{|\tau|}$, ordered from the smallest to the largest. The following holds.

The set C can cover $\tau - t_1$; otherwise, $\tau - t_1$ would be a set, with cardinality less than $|\tau|$, that A can cover, but that C cannot cover. Since C can cover $\tau - t_1$ but cannot cover τ, exactly $|\tau| - 1$ packets of C have laxity larger than or equal to t_1, and these packets can cover $\{t_1 + 1, \ldots, t_1 + |\tau| - 1\}$. Since A can cover τ, A contains at least $|\tau|$ packets with laxity larger than or equal to t_1. Thus, let p be a packet of $A - C$ that has laxity larger than or equal to t_1. Clearly, the set $C \cup \{p\}$ is schedulable, and has cardinality $r+1$; this contradicts the definition of r. □

Turning to the proof of the "only if" part of Theorem 2.2, by Lemma 3.8, if $S - Q$ is not equivalent to S, and there are no arrivals in the subsequent time slots, then fewer packets would be scheduled by any scheduling policy starting with $S - Q$ than would be scheduled by EDF starting with S. This completes the proof of the "only if" part of Theorem 2.2. Further, the following result is a simple corollary to the "only if" part of Theorem 2.2.

COROLLARY 3.2 *Given two on-line, throughput optimal scheduling policies π and π', whatever the arrival sequence, in every time slot, $S(\pi) \equiv S(\pi') \equiv S(\pi) - Q(\pi) \equiv S(\pi') - Q(\pi')$.*

The last bit of Theorem 2.2 is proved next. Consider a particular arrival sequence and a "current" slot k. By the simple greedy algorithm described after Theorem 2.2, there exists a throughput optimal scheduling policy π so that $S(\pi) - Q(\pi)$ is schedulable. Let π' be any other throughput optimal scheduling

policy. Since $S(\pi')-Q(\pi') \equiv S(\pi)-Q(\pi)$ and $S(\pi)-Q(\pi)$ is schedulable, it must be that $|S(\pi')-Q(\pi')| \geq |S(\pi)-Q(\pi)|$. The proof of Theorem 2.2 is complete.

The approach used in the proofs of Theorems 2.1 and 2.2 is frequently used in the remainder of this paper: the approach uses an induction on time to prove a set of necessary and sufficient conditions that must be satisfied, in each time slot, to achieve optimality.

3.2 Proof of Theorem 2.3. Theorem 2.3 is proved by induction. The following results are useful preliminaries.

LEMMA 3.9 *Let A_1, A_2, C_1, and C_2 be sets of packets with laxities such that $A_1 \cap A_2 = C_1 \cap C_2 = \emptyset$, $C_1 \succeq A_1$, and $C_2 \succeq A_2$. Then $C_1 \cup C_2 \succeq A_1 \cup A_2$.*

PROOF. The proof is similar to the proof of Lemma 3.4. □

LEMMA 3.10 *Let A be a set of packets with laxities and $C \subset A$, such that $\Phi(A) \cap C \neq \emptyset$. Then $\Phi(C) \subset \Phi(A)$.*

PROOF. Let p be the smallest packet in C, and q a packet in $\Phi(C)$. By Lemma 3.7, $T(C-p) \equiv T(C-q)$. Thus, by Lemma 3.4, $T(A-p) \equiv T(A-q)$. Further, $p \in \Phi(A)$, so, by part (ii) of Lemma 3.6, $q \in \Phi(A)$. □

LEMMA 3.11 *Let A and C be two sets of packets, such that $\Phi(A \cup C) \subset A$. Then $\Phi(A \cup C) = \Phi(A)$.*

PROOF. By Lemma 3.10, $\Phi(A) \subset \Phi(A \cup C)$, and $\Phi(\Phi(A \cup C)) \subset \Phi(A)$. By definition, $\Phi(\Phi(A \cup C)) = \Phi(A \cup C)$. So $\Phi(A \cup C) \subset \Phi(A) \subset \Phi(A \cup C)$, implying the lemma. □

Given a binary vector u of length at most M, and a set A of packets with laxities, denote by A^u the set of packets p in A such that u is a prefix of *p.c.* The following preliminary results are used in later proofs to analyze the properties of policy D_{lex} under the assumption that all packets in any arrival sequence have first bit equal to 0.

LEMMA 3.12 *If all packets have first bit equal to 0, then for any set of packets with laxities A, for any k, such that $1 \leq k \leq M$, and for any binary vector u of dimension k, if $\Gamma^k(A)$ contains a packet p, with u being a prefix of p.c, then $\Gamma^k(A) = \Phi(A^u)$.*

PROOF. Clearly, $\Gamma^1(A) = \Phi(A)$, since all packets have first bit equal to 0. Therefore Lemma 3.12 is true in case $k = 1$. For the sake of argument by induction, let $1 \leq k \leq M-1$ and suppose Lemma 3.12 is true for k. Then there exists a binary vector u of dimension k such that $\Gamma^k(A) = \Phi(A^u)$. If all packets in $\Phi(A^u)$ have kth bit equal to 1, then $\Gamma^{k+1}(A) = \Gamma^k(A)$; thus, $\Gamma^{k+1}(A) = \Phi(A^u)$, which in turn, by Lemma 3.11, is also equal to $\Phi(A^{u1})$, where $u1$ is the binary vector obtained by appending 1 to u. On the other hand, if $\Phi(A^u)$ contains a packet with $k+1$th bit equal to 0, then $\Gamma^{k+1}(A)$ is equal to $\Phi(\Phi(A^u)^{u0})$, where $u0$ is obtained by appending 0 to u. However, by Lemma 3.10, $\Phi(A^{u0}) \subset \Phi(A^u)$, which implies that $\Phi(A^{u0}) \subset \Phi(A^u)^{u_0}$, which, in turn, implies, by Lemma 3.11, that $\Phi(\Phi(A^u)^{u0}) = \Phi(A^{u0})$. Hence, $\Gamma^{k+1}(A) = \Phi(A^{u_0})$. This verifies Lemma 3.12 for $k+1$, and the induction argument is complete. □

The following result is a corollary to Lemma 3.12. Let A be a set of packets with laxities, let $\mathcal{C}(A)$ be the set of packets $p \in A$, such that $p \in \Phi(A^u)$, for every prefix u of *p.c*, and let $\mathcal{C}^*(A)$ be the set of packets of highest priority in $\mathcal{C}(A)$. The following result relates $\mathcal{C}^*(A)$ to $\Gamma(A)$.

COROLLARY 3.3 *If all packets have first bit equal to 0, then for any set A of packets with laxities, $\Gamma(A) = \mathcal{C}^*(A)$.*

PROOF. By Lemma 3.12, if $p \in \Gamma(A)$, then for any prefix u of *p.c*, $p \in \Phi(A^u)$. Therefore, $\Gamma(A) \subset \mathcal{C}(A)$. Now, let q be a packet with priority higher than that of p, and let i be the dimension of $p.c \cap q.c$. Then, by Lemma 3.12, $q \notin \Gamma^{i+1}(A)$, so $q \notin \Phi(A^{p.c \cap q.c})$, so $q \notin \mathcal{C}(A)$. □

LEMMA 3.13 *Let A and C be two sets of packets with laxities, such that packets in A and C have first bit equal to 0 and such that for any class identifier u, $A^u \equiv C^u$. Then, for any n, such that $1 \le n \le M$, packets in $\Gamma^n(A)$ and $\Gamma^n(C)$ have equal nth bit. Therefore, packets in $\Gamma(A)$ and $\Gamma(C)$ have the same class.*

PROOF. Argument by induction on n is used. The base case is established first: packets in $\Gamma^1(A)$ and $\Gamma^1(C)$ have equal first bit. This is trivial since all packets have first bit equal to 0.

For the induction step, let $1 \le n < M$ and suppose the packets in $\Gamma^n(A)$ and $\Gamma^n(C)$ have equal jth bits for $1 \le j \le n$. If will be shown that $\Gamma^{n+1}(A)$ and $\Gamma^{n+1}(C)$ have equal jth bits for $1 \le j \le n+1$. Since $\Gamma^{n+1}(A) \subset \Gamma^n(A)$ and $\Gamma^{n+1}(C) \subset \Gamma^n(C)$, packets in $\Gamma^{n+1}(A)$ and $\Gamma^{n+1}(C)$ have equal jth bits for $1 \le j \le n$, so it must only be shown that they have equal $n+1$st bits. Further, the $n+1$st bits of the packets in $\Gamma^{n+1}(A)$ are all the same, and the $n+1$st bits of the packets in $\Gamma^{n+1}(C)$ are all the same. Hence, it suffices to show that if $\Gamma^{n+1}(A)$ contains a packet with $n+1$st bit equal to zero, then $\Gamma^{n+1}(C)$ contains a packet with $n+1$st bit equal to zero.

Suppose $p \in \Gamma^{n+1}(A)$ with $n+1$st bit equal to zero. Let w be the length n prefix of the packets in $\Gamma^n(A)$. Then by Lemma 3.12, $\Gamma^n(A) = \Phi(A^w)$ and $\Gamma^n(C) = \Phi(C^w)$. Let q be a packet in $\Phi(C^{p.c})$. Since q has the same class as p, $q.c$ has $n+1$st bit equal to 0. Let v be any class identifier, such that $w0$ is a prefix of v. Lemma 3.3 implies that $T(C^v) \equiv T(A^v)$. Further, if $v = p.c$, part (i) of Lemma 3.6 implies that $T(C^v - q) \succeq T(A^v - p)$. Taking the union over all such v and applying Lemma 3.9 yields $T(C^{w0} - q) \succeq T(A^{w0} - p)$. In turn, by part (ii) of Lemma 3.6, this implies that $q \in \Phi(C^{w0})$. But $\Phi(C^{w0}) = \Gamma^{n+1}(C)$, so $\Gamma^{n+1}(C)$ contains a packet with $n+1$st bit equal to 0. The induction step is complete, and the lemma is proved. □

The proof of Theorem 2.3 is now discussed. First, Theorem 2.3 is proved under the assumption that all packets arriving in the system have first bit equal to 0. In this case, whatever the scheduling policy, $\tilde{S}$ is equal to S, in any time slot. The result is formally stated in the following proposition, and is proved next, but first the notion of lex-optimality is extended to the case when arrival sequences are limited to a class of arrival sequences.

Let $\mathbf{A}$ be a particular class of arrival sequences. A scheduling policy is lex-optimal relative to $\mathbf{A}$ if it satisfies the properties obtained by replacing in the definition of lex-optimality the expression "whatever the arrival sequence" by "whatever the arrival sequence in $\mathbf{A}$". Let $\mathbf{A_0}$ be the set of arrival sequences $\mathcal{A}$, such that any packet arriving in $\mathcal{A}$ has first bit equal to 0.

PROPOSITION 3.2 *A no-early-dropping scheduling policy is lex-optimal relative to $\mathbf{A_0}$ if and only if the policy is on-line, and schedules a packet in $\Gamma(S)$ whenever $S \neq \emptyset$.*

The proof is by induction on M. If M is equal to 1, Proposition 3.2 is the same as Theorem 2.1, and is, therefore, true. As an induction hypothesis, let Proposition 3.2 be true if the class identifiers have dimension smaller than M, for some fixed $M \ge 2$. In this case the induction hypothesis implies that a scheduling policy is order $M-1$ lex-optimal relative to $\mathbf{A_0}$ if and only if the policy schedules a packet in $\Gamma^{M-1}(S)$ whenever $S \neq \emptyset$. This is true, since for any n, the definition of order n lex-optimality and the definition of Γ^n depend on only the first n bits of the class identifiers.

Now, let π be a no-early-dropping scheduling policy that is order $M-1$ lex-optimal relative to $\mathbf{A_0}$ and schedules a packet in $\Gamma(S)$ whenever S is not empty. It is now established that π is lex-optimal relative to $\mathbf{A_0}$. This is first established, assuming that π is *class monotone*, i.e., whenever π schedules a packet, the packet is the *smallest* of its class. The smallest packet of A, a set of packets with laxities, is the packet that arrived first among packets with the smallest laxity in A— for this it is assumed that even packets that arrive in the same time slot do not arrive simultaneously. The following result justifies the attention given to class monotone scheduling policies.

LEMMA 3.14 *Let π be an on-line no-early-dropping scheduling policy. There exists an on-line, class monotone no-early-dropping scheduling policy π' that schedules a packet of the same class as π whenever π schedules a packet.*

PROOF. The class monotone scheduling policy π' is defined as follows: whenever policy π schedules a packet p, policy π' schedules the smallest packet in p's class; whenever policy π does not schedule a packet, neither does policy π'. The existence of π' and, therefore, the validity of Lemma 3.14 follow easily from Lemma 3.2, and an induction on time: in each time slot, for any class identifier u, $S(\pi')^u \succeq S(\pi)^u$. □

Lemma 3.14 implies that, whatever the set of arrival sequences $\mathbf{A}$, first, if a no-early-dropping scheduling policy lex-optimal relative to $\mathbf{A}$ exists, then a class monotone no-early-dropping scheduling policy lex-optimal relative to $\mathbf{A}$ exists, and second, a class monotone no-early-dropping scheduling policy that is order $M-1$ lex-optimal relative to $\mathbf{A}$ and maximizes the throughput of packets with the Mth bit equal to 0, over all class monotone no-early-dropping scheduling policies order $M-1$ lex-optimal relative to $\mathbf{A}$, is lex-optimal relative to $\mathbf{A}$.

Let π_c be the class monotone no-early-dropping scheduling policy that schedules a packet in $\Gamma^M(S)$ whenever S is not empty. As discussed above, by the induction hypothesis, π_c is order $M-1$ lex-optimal relative to $\mathbf{A_0}$. It is now established that π_c is lex-optimal relative to $\mathbf{A_0}$.

The following notation is useful. Given a set of packets with laxities A, a suffix of A is any subset of A that consists of the i largest packets in A, for some i, such that $0 \le i \le |A|$. Also, in the remainder of this paper, for any binary vector u, $u0$ is obtained by appending a 0 to u, and $u1$ by appending a 1 to u.

Let π be any class monotone no-early-dropping scheduling policy that is order $M-1$ lex-optimal relative to $\mathbf{A_0}$, and let u be a binary vector of dimension $M-1$, such that $u[1]=0$. It is now established by induction on time that: (a) $S(\pi_c)^u \equiv S(\pi)^u$; (b) $S(\pi_c)^{u0}$ is a suffix of $S(\pi)^{u0}$; and (c) $S(\pi_c)^{u1} \succeq S(\pi)^{u1}$. The induction hypothesis is the following: in some time slot, results (a), (b), and (c) are true.

Result (a) in the next time slot follows from Lemma 3.12: both policies π and π_c schedule a packet in $\Phi(S^u)$, whenever either policy schedules a packet in S^u. Results (b) and (c) are now considered.

First, π_c and π are class monotone no-early-dropping scheduling policies, thus, in any time slot, for any class identifier v, either $S(\pi_c)^v$ is a suffix of $S(\pi)^v$, or $S(\pi)^v$ is a suffix of $S(\pi_c)^v$. Therefore, for result (b), clearly, the analysis reduces to the case where, in addition to the assumptions in effect, $S(\pi_c)^{u0} = S(\pi)^{u0}$, and policy π schedules a packet p of class $u0$. It must be shown that policy π_c also schedules p in the current time slot. This is now undertaken.

Policies π and π_c are order $M-1$ lex-optimal relative to $\mathbf{A_0}$; therefore, policy π_c necessarily schedules a packet in $S(\pi_c)^u$. Further, by Lemma 3.12, policy π_c schedules a packet in $\Phi(S(\pi_c)^u)$. It simply remains to show that $\Phi(S(\pi_c)^u)$ contains a packet of class $u0$. Let q be any packet in $\Phi(S(\pi_c)^{u0})$. By Lemma 3.2, $T(S(\pi_c)^{u0} - p) \succeq T(S(\pi)^{u0} - q)$. Further, by Result (c) of the induction assumption and Lemma 3.3, $T(S(\pi_c)^{u1}) \succeq T(S(\pi)^{u1})$; hence, by Lemma 3.4, $T(S(\pi_c)^u - p) \succeq T(S(\pi)^u - q)$, which, by result (a) and part (ii) of Lemma 3.6, implies that $p \in \Phi(S(\pi_c)^u)$; therefore, $\Phi(S(\pi_c)^u)$ contains a packet of class $u0$. Result (b) thus holds in the next time slot.

Result (c) is now considered. If π_c does not schedule a packet of class $u1$, then Result (c) in the next time slot follows from Lemma 3.3. However, by Result (b), if $S(\pi)^{u0} = S(\pi_c)^{u0}$ and π schedules a packet of class $u0$, then so does π_c. Thus, the analysis of result (c) reduces to the case where, in addition to the assumptions in effect, $S(\pi_c)^{u0}$ is a *strict* suffix of $S(\pi)^{u0}$ (the sets are not equal), $S(\pi_c)^{u1}$ is a suffix of $S(\pi)^{u1}$, and policy π_c schedules a packet p of class $u1$.

It is now shown that $T(S(\pi_c)^{u1} - p) \succeq T(S(\pi)^{u1})$. In the current case $S(\pi_c)^u \equiv S(\pi)^u$, and the former set is a strict subset of the latter. Therefore, it must be true that packets in $S(\pi_c)^u$ cannot *all* be scheduled, starting in the *next* time slot, which by Lemma 3.1, implies that packets in $\Phi(S(\pi_c)^u)$ cannot *all* be scheduled, starting in the next time slot. Further, since policy π_c schedules a packet of class $u1$, $\Phi(S(\pi_c)^u) \subset S(\pi_c)^{u1}$; hence, packets in $S(\pi_c)^{u1}$ cannot *all* be scheduled, starting in the next time slot. Now, let τ be a set that $T(S(\pi_c)^{u1})$ can cover; necessarily, $|\tau| \le |S(\pi_c)^{u1}| - 1$, so there exists $q' \in S(\pi_c)^{u1}$, such that $T(S(\pi_c)^{u1} - q')$ can cover τ. Thus, by Lemma 3.2, $T(S(\pi_c)^{u1} - p)$ can cover τ. Thus $T(S(\pi_c)^{u1} - p) \succeq T(S(\pi)^{u1})$, which means (c) is true in the next time slot. This completes the proof of results (a), (b), and (c) by induction on time.

Consequently, any packet of class $u0$ that is scheduled by policy π in some time slot, is scheduled by policy π_c in the same time slot or earlier: $S(\pi_c)^{u0}$ is a suffix of $S(\pi)^{u0}$. Therefore, policy π_c is lex-optimal.

The proof of the "if" part of Proposition 3.2 follows easily by induction on time: by Lemmas 3.4, 3.6, 3.12, and 3.13, for any no-early-dropping scheduling policy π that schedules a packet in $\Gamma(S)$ whenever S is not empty, for any class identifier u, and in any time slot, the following holds: $S(\pi_c)^u \equiv S(\pi)^u$. Therefore, policies π and π_c schedule a packet of the same class whenever either policy schedules a packet. Since π_c is lex-optimal, so is π. This completes the proof of the "if" part of Proposition 3.2 for M. It remains to establish the "only if" part of Proposition 3.2 for M. This proof is undertaken.

Let π be an order $M-1$ lex-optimal no-early-dropping scheduling policy, such that, in some time slot, t say, π schedules a packet p that is not in $\Gamma(S)$. Necessarily, $p.c$ ends with a 0, $p \in \Phi(S^u)$, but $p \notin \Phi(S^{u0})$, where u is the prefix of $p.c$ of dimension $M-1$. Thus, the vector $u0$ is equal to $p.c$.

Let l be the largest laxity in $\Phi(S^u)$, let C be the set of packets in S^{u0} that have laxity smaller than or equal to l, and let q be any packet in $\Phi(S^{u0})$. Since $S^{u0} \cap \Phi(S^u) \neq \emptyset$, Lemma 3.10 implies that $\Phi(S^{u0}) \subset \Phi(S)$, so $\Phi(S^{u0}) \subset C$. Hence by Lemma 3.11, $\Phi(S^{u0}) = \Phi(C)$. Also $\Phi(S^{u0}) = \Gamma(S)$. So $p \notin \Phi(C)$, and, by Lemma 3.6, $T(C-p)$ cannot cover $T(C-q)$. Let τ be a set that $T(C-q)$ can cover, but that $T(C-p)$ cannot cover.

Now, let the subsequent arrival sequence consist of two distinct kinds of packets: *real* and *imaginary* packets. It is assumed that, by some method, which does not change packets' attributes, such as deadlines and class, it is possible to "tag" the packets as real or imaginary packets. Such a tagging has no bearing on the current proof, but is useful for later arguments.

There are no real arrivals after time slot $t+l-1$. In time slot $t+1$, l real packets with laxity $l-1$ and class $u1$ arrive, and a real packet with laxity 1 and class $u0$ arrives in each time slot, among time slots $t+1$ through $t+l-1$, that is not referenced by an element of τ (relative to current time slot t). Further, in each of the time slots after time slot t, an arbitrary number of imaginary packets with arbitrary deadlines and class $01\ldots1$ (M bits) may arrive. The real arrivals all have deadline smaller than or equal to $t+l$.

Now, let D be the set of all packets in S that have priority higher than $u0$, or are of class $u0$ with laxity larger than l. The goal is to show that whatever the order $M-1$ lex-optimal scheduling policy, starting with S, the policy does not schedule any packet in D in the following $l-1$ time slots.

On the one hand, if D contains packets of class $u0$, then, by Lemma 3.1, $T^l(D^{u0}) \sim D^{u0}$, and $T^l(D^{u0})$ is schedulable. On the other hand, if there is a packet q' in D with priority higher than $u0$, let v' be the binary vector $q'.c \cap u$. By Lemma 3.12, $S^{v'0} \cap \Phi(S^{v'}) = \emptyset$; thus, $S^{v'0} \subset S^{v'} - \Phi(S^{v'})$. Further, by Lemma 3.1, $T^{l'}(S^{v'} - \Phi(S^{v'})) \sim S^{v'} - \Phi(S^{v'})$, and $T^{l'}(S^{v'} - \Phi(S^{v'}))$ is schedulable, where l' is the largest laxity in $\Phi(S^{v'})$. Further, by Lemma 3.10, $\Phi(S^u) \subset \Phi(S^{v'})$; therefore, $l' \geq l$. On either hand it follows that for any packet g in D, there exists a prefix w of $u0$, a set of packets E, and an integer j , such that $g \in E$, $E \subset S^w$, $T^j(E) \sim E$, $T^j(E)$ is schedulable, and $j \geq l$. It is now shown that, with the arrival sequence described earlier, $g \notin \Phi(S^w)$ in the following $l-1$ time slots. Therefore, by Lemma 3.12, $g \notin \Gamma^{M-1}(S)$ in these time slots, and cannot be scheduled by an order $M-1$ lex-optimal scheduling policy. This follows from the result below.

LEMMA 3.15 *Let A be a set of packets with laxities, such that, for some $j \geq 1$, $T^j(A) \sim A$ and $T^j(A)$ is schedulable. Then, for any nonempty set of packets with laxities F, such that all packets in F have laxity smaller than or equal to j, $\Phi(A \cup F)$ is a subset of F.*

PROOF. All packets in A are needed to cover $\{j, \ldots, j+|A|-1\}$ with packets in $T(A \cup F)$, which, by part (i) of Lemma 3.6, implies the result. □

By Corollary 3.3, clearly, the packets of class $u0$ that arrive in the subsequent time slots are in set $\Gamma^{M-1}(S)$ for these time slots, and can be scheduled in these time slots by an order $M-1$ lex-optimal scheduling policy. Consequently, let π' be an order $M-1$ lex-optimal scheduling policy that schedules packet q in the current time slot, and the new arrivals of class $u0$ in time slots, among the following $l-1$ time slots, that are not referenced by elements in τ. Since l packets of class $u1$ and laxity $l-1$ are added in the time slot following the current, any packet of class u that has deadline smaller than or equal to $t+l$ is in set $\mathcal{C}(S)$ in the following $l-1$ time slots. Therefore, since no packet of higher priority is in that set, packets of class u are in set $\Gamma^{M-1}(S)$ in these time slots. Therefore, π' can be extended in the time slots referenced by τ as follows: in each of these time slots, π' schedules any packet in $\Phi(C')$, where C' is

the set of packets of C remaining in $S(\pi')$. Clearly, π' is order $M-1$ lex-optimal, and schedules packets of class $u0$ in the current time slot and the following $l-1$ time slots.

Policy π, however, schedules at least one packet of class $u0$ too few in the subsequent $l-1$ time slots. Therefore, policy π is not lex-optimal. This completes the proof of the "only if" part of Proposition 3.2 for M. This completes the proof of Proposition 3.2 by induction on M.

It remains to prove Theorem 2.3 in the general setting. It is no more assumed that all class identifiers start with a 0. The idea of the proof is that the arrival sequences can be modified to use Proposition 3.2. Any arrival sequence is modified as follows. First, the class identifier of each packet in the arrival sequence is expanded from M bits to $M+1$ bits by adding a 0 at the beginning of the class identifier. Second, the arrival sequence is "stuffed" by adding imaginary packets to the modified arrival sequence. Henceforth, the packets originally in the arrival sequence are called real packets. The imaginary packets are of class $01\ldots1$ ($M+1$ bits), and are added as follows. Let I_d be the no-early-dropping scheduling policy that never schedules any packet, and, in each time slot, let m^* denote the maximum laxity in $S(I_d)$ under the original arrival sequence; $m^*=0$ if $S(I_d)=\emptyset$. Then, in each time slot, m^* imaginary packets with laxity m^* are added to the arrivals for the slot. This defines the stuffed version of the arrival sequence.

An arrival sequence is said to be stuffed if the sequence is the stuffed version of some arrival sequence. It is now established that Theorem 2.3 holds when arrival sequences are restricted to stuffed sequences. This is formally stated in the following proposition.

PROPOSITION 3.3 *A no-early-dropping scheduling policy is lex-optimal relative to the set of stuffed arrival sequences if and only if the policy is on-line, and schedules a packet in* $\Gamma(S)$ *whenever* $S\neq\emptyset$.

Proposition 3.3 is now established. Clearly, all packets of stuffed sequences have first bit equal to 0, and, further, in each time slot, enough imaginary packets are added to the real packets so that any nonidling scheduling policy is throughput optimal relative to stuffed sequences, i.e., in any time slot, $\Gamma^1(S)=S$. Therefore, an on-line scheduling policy is order 1 lex-optimal relative to stuffed sequences if and only if the policy schedules a packet in S whenever $S\neq\emptyset$, and $\Gamma^1(S)=S$. This means that Proposition 3.3 is valid if $M=1$. As an induction hypothesis, it is assumed that Proposition 3.3 is valid for $M-1$, i.e., if arrival sequences are stuffed, an on-line no-early-dropping scheduling policy is order $M-1$ lex-optimal if and only if the policy schedules a packet in $\Gamma^{M-1}(S)$ whenever $S\neq\emptyset$. Then it remains to show that the policy is lex-optimal if and only if the policy schedules a packet in $\Gamma(S)$ whenever $S\neq\emptyset$.

The "if" part is discussed first. The argument used for the "if" part of the induction step of Proposition 3.2 can be repeated here to yield the desired result. Nowhere in that argument is there a condition on the arrival sequences, other than the assumption that class identifiers start with a 0, which holds for stuffed sequences. Therefore, the "if" part of Proposition 3.3 holds for M.

The "only if" part is now discussed. The proof is similar to the one done for the "only if" part of the induction step of Proposition 3.2, since, in the latter proof, adding imaginary packets was allowed in the construction of the sequence used to show that an order $M-1$ policy is not lex-optimal whenever the policy does not schedule a packet in $\Gamma(S)$: the number of imaginary packets added, and their deadlines were arbitrary; thus these packets could be added in such a way that the subsequent sequence be "stuffed". From this, it follows that Proposition 3.3 is valid for M, and, therefore, valid for any value of M. This completes the proof of Proposition 3.3.

The proof of Theorem 2.3 is now presented. Given a scheduling policy π, under the original arrival sequences, let the *extension* of π be the scheduling policy π', under the stuffed arrival sequences, that schedules the same packet as π whenever π schedules a real packet that has a 0 bit; otherwise, π' schedules any packet in $S(\pi')$ if $S(\pi')\neq\emptyset$. For clarity, from this point on, the notation S is used to denote the buffer of a policy in the original model; the buffer of an extension policy is denoted $P(S)$.

Clearly, a scheduling policy, under the original arrivals, is lex-optimal if and only if the policy has an extension that is lex-optimal for stuffed arrival sequences. This comes from the definition of lex-optimality. It is shown next that whenever $\tilde{S}\neq\emptyset$, $\Gamma(S)=\Gamma(P(S))$, where $\Gamma(P(S))$ is computed with the extended class identifiers. Let S be such that $\tilde{S}\neq\emptyset$, and let m be the position of the first 0 bit

of the highest priority packet in S. Clearly, for any i, such that $1 \le i \le m-1$, $\Gamma^{i+1}(P(S)) = S$, and $\Gamma^{m+1}(P(S)) = \Phi(P(S)^{(m+1)})$, which is equal to $\Gamma^m(S)$. Since, $\Gamma(P(S))$ is obtained from $\Gamma^{m+1}(P(S))$ by the same steps used to obtain $\Gamma(S)$ from $\Gamma^n(S)$, it follows that $\Gamma(P(S)) = \Gamma(S)$. Further, if $\tilde{S} = \emptyset$, then $\Gamma(P(S)) = \Phi(P(S)) = P(S)$.

Consider a scheduling policy π applied to the original arrival sequence. If π schedules a packet in $\Gamma(S)$ whenever $\tilde{S} \neq \emptyset$, then π's extension is lex-optimal relative to stuffed sequences, so π is also lex-optimal. On the other hand, if in some slot with $\tilde{S} \neq \emptyset$ π does not schedule a packet in $\Gamma(S)$, then π's extension does not schedule a packet in $\Gamma(P(S))$, so π's extension is not lex-optimal. Thus, neither is π lex-optimal. This completes the proof of Theorem 2.3.

4. Verification of optimality of algorithms. The main purpose of this section is to prove Theorem 2.5, that Algorithm D_{lex} is indeed lex-optimal and requires no more buffer space than any on-line throughput optimal policy. We begin by establishing Theorem 2.4, that D_s is throughput optimal.

4.1 Proof of Theorem 2.4. It is first shown that, at the beginning of each time slot and, in each time slot, just before a call to procedure Tick, the following properties of B hold.

Property 1. The virtual laxities of packets in B are distinct and ordered; the closer to the head of B the smaller the virtual laxity.

Property 2. For any packet p in B, $p.v \le p.l$, and there are $p.l - p.v + 1$ packets in B with virtual laxities consecutive from $p.v$ to $p.l$.

Property 3. For packet $b_k \in B$, $b_k.v \ge k$.

Since Properties 1 through 3 are valid at the beginning time slot 1, it suffices to show that these properties are preserved by procedure Insert and by procedure Tick. This is done in two steps.

First, let Properties 1 through 3 be valid just before a call to procedure Insert, and let there be a new arrival p for insertion into B. Let B' be the array obtained by insertion of pck at position 0 of B at the end of the compete phase of procedure Insert. The packets of B' are also denoted $b_0, \ldots, b_L$, from the head of B' to the tail. The array B' thus obtained does not serve any practical purpose as to the implementation of Algorithm D_s; however, it is used as a logical tool to simplify the discussion.

It is now shown that Properties 1 through 3 hold with B replaced by B'. For clarity, the properties are referred to as Properties $1'$, $2'$ and $3'$ when B' is concerned. The result is trivial if B is empty prior to the insertion of the new arrival p. In this case $b_0.v$ in B' is equal to $b_0.l$. For the general case, it is worth discussing the dynamics of the compete phase of procedure Insert.

At the beginning of the compete phase, $pck = p$, $pck.v = pck.l = p.l$, and B is divided into three sections, B_1, B_2, and B_3; where a section is a possibly empty set of packets with contiguous positions in B. Section B_3 consists of packets in B that have virtual laxity larger than $pck.l$. Section B_2 is empty if B does not contain any packet with virtual laxity equal to $pck.l$; otherwise, taking b as the packet in B such that $b.v = pck.l$, section B_2 consists of packets b' in B, such that virtual laxities of packets in B, between b' and b, are consecutive and increasing from $b'.v$ to $pck.l$. Section B_1 consists of the packets in B that are neither in B_2 nor in B_3.

The compete phase runs as follows. The variable pos decreases from L, the number of packets in B, to 0 during the execution of the *while* loop in the compete phase. The execution of this loop may be divided into three stages as follows. In the first stage, from $pos = L$ down to pos equal to the position of the first packet of B_3, $pck.v < b_{pos}.v$; thus, pck is promoted over b_{pos}, without swap or update in virtual laxity: *pck slides by* b_{pos}. In the second stage, from pos equal to the position of the last packet of B_2 down to the first, $pck.v = b_{pos}.v$; thus, pck and b_{pos} may be swapped if desired, and the virtual laxity of the packet promoted is decreased by 1 during the promotion. In the third stage, when pos reaches the position of the last packet of B_1, either $pck = p$ and $pck.v = l$, or $pck.v = b_{pos+1}.v - 1$; in either case $pck.v > b_{pos}.v$. Thus, b_{pos} is promoted without any update in virtual laxity. Henceforth, for any value of pos, pck and b_{pos} are swapped without update in virtual laxities. Thus, the packets of B_1 are simply shifted by one position towards the head of B. Their order and virtual laxities remain the same.

Figure 7 illustrates the execution of procedure Insert for a new arrival. In the figure the packets' attributes are represented by two numbers: the top number is each packet's virtual laxity, while the

bottom number is the laxity. Thus, the new arrival, which is the initial value of *pck*, has laxity 5 and virtual laxity 5. For the example, B_3 consists of packets b_5, b_6, and b_7, B_2 consists of packets b_3 and b_4, and B_1 consists of packets b_1 and b_2. As discussed above, the compete phase is divided into three stages. In the first stage, *pck* slides by the packets of B_3, i.e., *pck* is promoted over b_7, b_6 and b_5, without any update in virtual laxity, and, further, while doing so, *pck* remains equal to the new arrival. In the second stage, which is determined by values of *pos* equal to the positions of packets of B_2, $pck.v = b_{pos}.v$, the packets may be swapped, and the virtual laxity of the promoted packet, in each iteration of the *while* loop, is decreased by 1. In the example shown, when *pck* reaches b_4 and b_3, *pck* is promoted over b_4. In the third stage, *pck* and b_{pos} are swapped, without update in virtual laxity. In this stage, $pck.v$ is always larger than $b_{pos}.v$; therefore the packets of B_1 are simply shifted by one position, without changing their virtual laxities. Part (c) of Figure 7 presents the intermediate buffer B', while part (c) presents B after the squeeze phase. No packet is dropped in the example. Figure 8 shows an example for which Insert

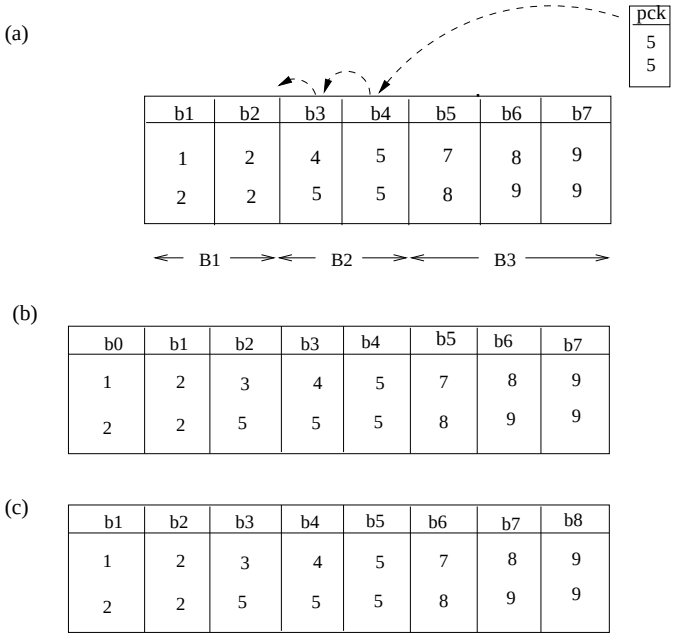

Figure 7: Illustration of procedure Insert in a case without dropping.

drops a packet. Properties $1'$ through $3'$ follow from the discussion above: B' is the concatenation of B_1, B'_2, and B_3, where B'_2 is obtained from B_2 through the second stage described above.

It is now shown that Properties $1'$ through $3'$ imply that Properties 1 through 3 hold on the array B that is obtained at the end of the squeeze phase. Clearly, from the description of Algorithm D_s, two results need to be established: first, if $b_0.v = 0$, then a packet is dropped during the *insertion* phase, and, second, if packet b_k is dropped, then, for any q preceding b_k in B', $q.v < q.l$. The first result follows from the following claim.

CLAIM 4.1 *A packet is dropped during the squeeze phase if and only if* $b_0.v = 0$.

PROOF. The "if" part is considered first. It is assumed that $b_0.v = 0$. Let b_j be the last packet in B' such that the virtual laxities of packets from b_0 to b_j are consecutive. Then, by Property $2'$, $b_j.v = b_j.l = j$, so a packet is dropped that precedes or is equal to b_j. The "if" part is proved.

The "only if" part is now considered. It is assumed that packet b_k is dropped during the squeeze phase. Then $k \le b_k.v \le b_k.l = k$. Since the virtual laxities are increasing, the virtual laxities of packets from b_0 to b_k necessarily range from 0 to k; so $b_0.v = 0$. □

The second result follows from the proof of the "only if" part of Claim 4.1. According to that proof, if packet b_k is dropped, then, for any b_j preceding or equal to b_k in B', $b_j.v = j$. Thus, $b_j.v < b_j.l$, since b_j precedes b_k. This shows that Properties 1 through 3 are preserved by procedure Insert.

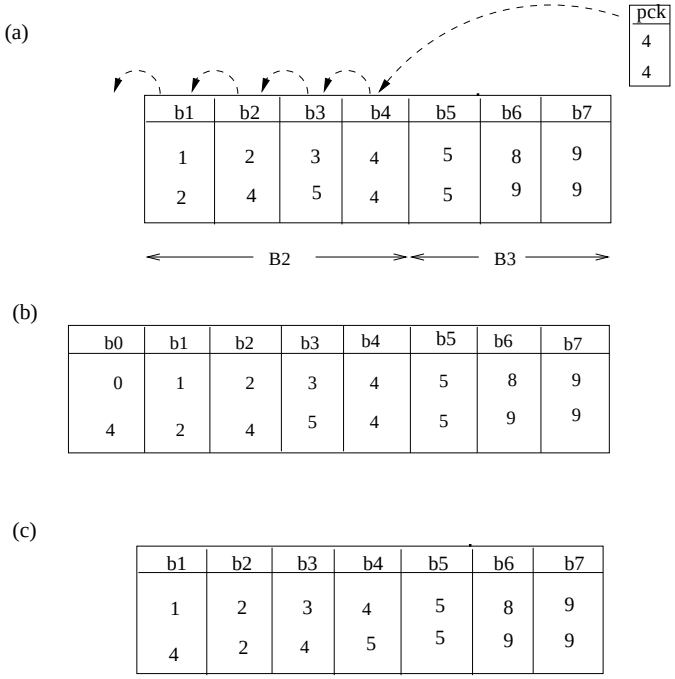

Figure 8: Illustration of procedure Insert in a case with dropping

Procedure Tick also preserves Properties 1 through 3. During procedure Tick, the packet at the head of B is scheduled, and the laxities and virtual laxities are decreased to reflect the passing of time. Properties 1 through 3 depend only on the difference between the virtual laxity and position of packets in B, on the difference between the virtual laxity and laxity of packets in B, and on the packets that are after each packet of B. Thus, procedure Insert and procedure Tick preserve Properties 1 through 3. This completes the proof of the validity of Properties 1 through 3 at the beginning of each time slot and just before each call to procedure Tick.

It is now shown that Properties 1 through 3 imply that policy D_s is throughput optimal. The goal is to show that policy D_s satisfies the "if" part of Theorem 2.2. Two parts are considered: first, it is established that each scheduling decision made by procedure Tick, is consistent with throughput optimality, i.e., $b_1 \in \Phi(B)$ whenever B satisfies Properties 1 through 3. Second, it is established that the dropping in procedure Insert is consistent with the requirements of Theorem 2.2. The first part is considered.

Let b_j be the last packet in B such that the virtual laxities of packets from b_1 to b_j are consecutive. Then, $b_j.v = b_j.l$, and all packets preceding b_j, if any, have laxity less than or equal to $b_j.l$. Now, let b_i be the first packet such that $b_i.v = b_i.l$, and the laxities of packets preceding b_i, if any, are less than or equal to $b_i.l$. It is now shown that $b_i.l$ is the smallest laxity of B with minimum excess index, from which it follows immediately that $\Phi(B) = \{b_1, \ldots, b_i\}$; in particular, $b_1 \in \Phi(B)$.

Clearly, the virtual laxities and laxities of packets after b_i, if any, are larger than $b_i.v$, and, thus, larger than $b_i.l$, Thus, the excess index of $b_i.l$ is equal to $b_1.v - 1$. Now, let b_n be a packet, such that $b_n.l < b_i.l$. Let b_m be the packet of B such that $b_m.v = b_n.l$. Clearly, b_m precedes b_i, and there must be a packet that precedes or is equal to b_m with laxity larger than $b_n.l$; otherwise, $b_m.v$ would be equal to $b_m.l$, and all packets before b_m would have laxity less than or equal to $b_m.l$, which contradicts the definition of b_i. Thus, the excess index of $b_n.l$ is larger than or equal to $b_m.v - m + 1$, which is equal to $b_1.v$, and, therefore, larger than the excess index of $b_i.l$.

Finally, for any laxity l in B, the excess index of l is larger than or equal to $b_{k_l}.l - k_l$, where b_{k_l} is the packet with virtual laxity l; and, further, $b_{k_l}.l - k_l$ is larger than or equal to $b_1 - 1$. Thus, $b_i.l$ is the largest laxity in $\Phi(B)$, as was to be shown. Thus, the scheduling in policy D_s is consistent with throughput optimality. It remains to consider the second part, that the dropping does not preclude future throughput optimality. This is now done.

Let packet b_k be dropped during the insertion of a new arrival p. By the proof of Claim 4.1, there are $b_j.l + 1$ packets in B' with laxity less than or equal to $b_j.l$, where b_j is the last packet such that the virtual laxities are consecutive between b_0 and b_j. Thus, B' is not schedulable. Now, after the dropping, the array B obtained is schedulable: the laxity of each packet is larger than or equal to the packet's position in B; thus, these packets can *all* be scheduled in order, starting from b_1. Also, $|B|$ is equal to $|B'| - 1$. Thus, B is a schedulable subset of B' with maximum cardinality; thus, $B \equiv B'$, so, by Theorem 2.2, dropping packet b_k does not matter. This completes the proof of Theorem 2.4.

4.2 Proof of Theorem 2.5. This proof is divided into two main parts: first, important properties of the data structure of Algorithm D_{lex} are established; second, these properties are used to determine the properties of the scheduling policy D_{lex}. The properties of the data structure are now considered. Throughout this proof class identifiers all start with a 0, unless otherwise stated.

Let p be a packet in the data structure of Algorithm D_{lex}: p is either packet *pck* or a packet in B. The *alternative* virtual laxity vector of p, denoted $p.\tilde{v}$, is obtained by replacing any ∞ in $p.v$ by the last finite coordinate of $p.v$ preceding the ∞. For example, if p is of class 01101, with $p.v = (5\infty\infty6\infty)$, then $p.\tilde{v} = (55566)$.

The properties are now stated; these properties hold on the array B that is observed, in any time slot, just before a call to procedures M-Insert and M-Tick.

Property 1. For any packet p in B, the coordinates of $p.\tilde{v}$ are nondecreasing, starting from $p.\tilde{v}[1]$.

Property 2. Let u be a prefix of some class identifier, and let i be u's dimension. Then the ith alternative virtual laxity of packets in B^u are increasing with position. (Recall that B^u is the set of packets $p \in B$ such that u is a prefix of $p.c$.)

Property 3. Let u be a prefix of some class identifier, let i be u's dimension, and let p be any packet in B^u. Then, $p.\tilde{v}[i] \leq p.l$, and there are $p.l - p.\tilde{v}[i] + 1$ packets in B^u with ith alternative virtual laxities consecutive from $p.\tilde{v}[i]$ to $p.l$.

Property 4 For any packet b_k in B, $b_k.\tilde{v}[1] \geq k$.

The proof of Properties 1 through 4 is done by induction on time. First, it is shown that if these properties hold on B before a call to procedure M-Insert for a new arrival p, then, just after the compete phase, the same properties hold, with B replaced by B' in the statement of the properties, where B' is the array obtained by inserting *pck* at position 0 of B. For clarity, the properties are referred to as Properties 1′ through 4′ when B' is concerned. Then, it is established that Properties 1′ through 4′ imply that Properties 1 through 4 hold on the array B obtained after the squeeze phase of procedure M-Insert.

Properties 1 through 4 and Properties 1′ through 4′ can be readily verified for the example shown in Figure 9, which illustrates execution of the compete phase of a call to procedure M-Insert in case $M = 7$. Part (a) shows B and a new packet to be inserted. The numbers in the bottom row indicated the laxities of the packets in B, while the other numbers are the laxities of the packets. The new arrival has laxity 6 and class 0000111. Part (b) presents the array B' that is obtained prior to the squeeze phase, in which packet b_{12} will be dropped. The dashed lines in part (b) as well as part (c) of Figure 9 will be discussed later in the paper.

The first result is considered. Clearly, from the description of Algorithm D_{lex}, Properties 1′ and 4′ are true. It remains to show that Properties 2′ and 3′ also hold. For this, let p be the new arrival that is to be inserted by procedure M-Insert.

Let u be any prefix of some superclass, let i be u's dimension. If u is a prefix of $p.c$, the subsets B_1^u, B_2^u and B_3^u of B^u are defined as follows. Set B_3^u is the set of packets in B^u with ith alternative virtual laxity larger than $p.l$. If B^u does not contain any packet b_0 with ith alternative virtual laxity equal to $p.l$, then $B_2^u = \emptyset$; otherwise, B_2^u is the set of packets $b \in B^u$, such that the ith alternative virtual laxities of packets in B^u, from b to b_0, are increasing and consecutive from $b.\tilde{v}[i]$ to $p.l$. Finally, $B_1^u = B^u - (B_2^u \cup B_3^u)$. These definitions are analogous to the definitions of B_1, B_2 and B_3 for B of Algorithm D_s.

The goal now is to prove the *projection property* of the compete phase of Algorithm D_{lex}, which is stated below. *Projection property* Let u be a prefix of some class identifier, and let i be u's dimension. During the execution of a call to procedure M-Insert for the new arrival p, if u is not a prefix of $p.c$, then,

(a)

b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11	b12	b13	b14	b15	b16	pck
1	2	3	4	5	6	7	8	9	10	11	12	13	14	16	17	6
1	2	3	4	5	6	7	8	9	10	11	12	13	14	17	–	6
5	6	–	–	–	–	–	–	–	–	–	–	–	–	17	–	6
–	–	3	4	5	6	7	8	9	10	11	12	13	–	17	–	–
–	–	4	5	–	–	–	–	–	–	–	–	–	–	17	17	–
–	–	4	5	5	6	7	8	9	10	11	12	13	–	17	17	–
–	–	4	5	6	7	8	–	–	–	–	–	–	14	17	17	6
6	6	5	5	6	8	8	13	10	11	12	12	13	14	17	17	

(b)

b0	b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11	b12	b13	b14	b15	b16
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16	17
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	17	–
4	5	6	–	–	–	–	–	–	–	–	–	–	–	–	17	–
6	–	–	3	4	5	6	7	8	9	10	11	12	13	–	17	–
–	–	–	4	5	–	–	–	–	–	–	–	–	–	–	17	17
–	–	–	4	5	5	6	7	8	9	10	11	12	13	–	17	17
–	–	–	4	5	6	7	8	–	–	–	–	–	–	14	17	17
6	6	6	5	5	6	8	8	13	10	11	12	12	13	14	17	17

(c)

b0+	b1+	b2+	b3+	b4+	b5+	b6+	b7+	b8+	b9+	b10+
0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	10
–	–	–	–	–	–	–	–	–	–	–
0	1	2	3	4	5	6	7	8	9	10
1	2	–	–	–	–	–	–	–	–	–
1	2	2	3	4	5	6	7	8	9	10
1	2	3	4	5	–	–	–	–	–	–
2	2	3	5	5	10	7	8	9	9	10

Figure 9: Execution of the compete phase of M-Insert, prior to dropping.

the compete phase does not change the order and ith alternative virtual laxities of packets in B^u. On the other hand, if u is a prefix of $p.c$, and *only* the order and ith alternative virtual laxities of packets in $B^u \cup \{pck\}$ are considered, the following is observed: first, pck is promoted over all packets in B^u_3, if any, and, further, $pck = p$ and $pck.\tilde{v}[i] = p.l$ after each such promotion. Second, if $B^u_2 \neq \emptyset$, the first time pck reaches a packet of B^u_2 in B, which must be the last packet in B^u_2, then $pck = p$ and $pck.v = p.l$; further, when pck reaches any packet b in B^u_2, u is a prefix of $pck.c$, and the coordinates of $b.\tilde{v}$ and $pck.\tilde{v}$, from the first to the ith, are equal. Third, the order and ith alternative virtual laxities of packets in B^u_1 are not changed, and, after the compete phase, these packets precede any other packet in $B^u \cup \{p\}$.

The projection property implies that the compete phase of Algorithm D_{lex} is identical to the compete phase of Algorithm D_s when the latter is run with the new arrival p on the array B'' obtained as follows: packets of B^u, from the first to the last in B, are inserted in consecutive positions of B'', starting from position 1, and the ith alternative virtual laxities of these packets are taken as their virtual laxities in B''. In particular, if u is not a prefix of $p.c$, there is no arrival for the compete phase of Algorithm D_s run on B'', and, if u is a prefix of $p.c$, the packet promoted in case of optional swapping, if any, is determined by the virtual laxity vectors of the competing packets.

Figure 10 presents the execution of a call to procedure M-Insert. For this example, $M = 2$. The laxity of each packet in Figure 10 is given by the bottom number; the other numbers represent the virtual laxities. The new arrival has laxity 8 and class 00; therefore, the virtual laxities of the packets are initially equal to 8. The projection property is illustrated by the example. If only the first alternative virtual

laxities are considered, then the new arrival slides by packets with first alternative virtual laxity larger than 8: these are packets $b_6, \ldots, b_{10}$. Then, for values of *pos*, from 5 t0 2, the first virtual laxity of the packet promoted is decreased by 1: these coordinates are consecutive from 8 down to 5. Finally, b_1 is simply shifted by one position towards the head of B. The squeeze phase for this example is not illustrated: it consists in simply adding *pck* as the head of the array in part (c); no dropping is necessary.

The property also holds for the second alternative virtual laxities. Packets b_1, b_6, b_7, and b_{10}, which are not from the class of the arrival, keep the same second alternative virtual laxities at the end of the compete phase. In parts (a) and (b), the new arrival *pck* slides by packets b_5, b_8 and b_9, which have second virtual laxity larger than 8. On the other hand, the swapping is optional for packets b_3, and b_4, which have second virtual laxity consecutive from 7 to 8. In this example we elected to swap when b_4 is reached. Finally, b_2 is promoted over *pck*, since b_2 has second virtual laxity equal to 5.

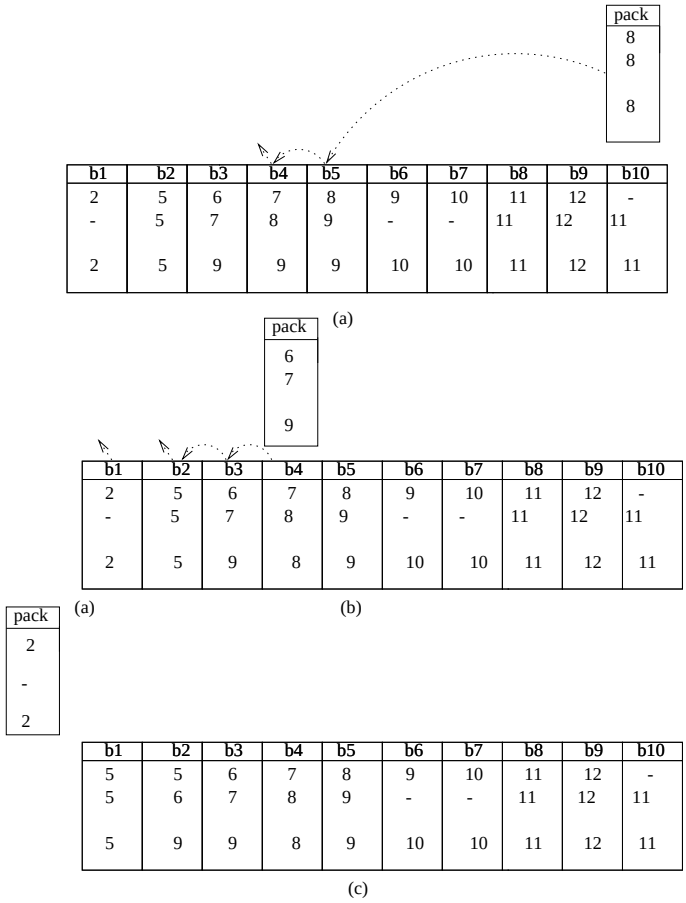

Figure 10: Execution of the compete phase of procedure M-Insert.

The projection property is now established by induction. If u is a prefix of some class and has dimension 1, then u is equal to 0, since all class identifiers start with a 0. Thus, the projection property for u is true, since Algorithm D_{lex} step-by-step follows D_s, when all class identifiers start with a 0, with the exception that the packet promoted in cases that allow an optional swapping for Algorithm D_s is determined by coordinates, other than the first, of the virtual laxity vectors.

As the induction hypothesis, let there be an integer j, such that $2 \leq j \leq M$, and such that for any v of dimension smaller than j, the projection property holds for v. Let u be a dimension j prefix of some class.

First, it is assumed that u is not a prefix of *p.c.* Then it must be shown that the ith alternative virtual laxities and order of packets in B^u are not affected by the compete phase of procedure M-Insert.

Since the jth alternative virtual laxity of a packet in B^u can only be changed if, at some point, during the compete phase, the packet has virtual laxities, from the first to the jth, equal to that of the packet with which it is competing, which implies that the latter packet is also a packet from B^u. However, by Property 2, the jth virtual laxities of the packets in B^u are distinct. Hence the jth alternative virtual laxity of packets in B^u are not changed during the compete phase.

Thus, only the order of the packets in B^u need to be considered. For this, let v be the vector $p.c \cap u$, and let j' be v's dimension. Then $j' < j$. If $j' + 1 < j$, let w be u's prefix of dimension $j' + 1$. Then, by the induction hypothesis, the order and the $j' + 1$th alternative virtual laxities of packets in B^w are not affected by the compete phase, therefore, the order and jth alternative virtual laxities of packets in B^v are not affected. Thus, it is assumed that $j' = j - 1$.

Clearly, the order or jth alternative virtual laxities of packets in B^u cannot be affected unless the order or $(j-1)$th alternative virtual laxities of *these* packets in B^v are affected. Further, the latter can only be affected for packets in B^v_2. Furthermore, when *pck* reaches a packet b in $B^u \cap B^v_2$, if u is a prefix of $pck.c$, then, by the induction hypothesis, the coordinates of $pck.\tilde{v}$ and $b.\tilde{v}$, from the first to the $(j-1)$th, are equal, However, since the alternative virtual laxities of packets in B^u do not change during the compete phase, by Property 2, $pck.\tilde{v}[j] > b.\tilde{v}[j]$. Thus, *pck* and b are swapped, without update of the jth virtual laxity. Therefore, packets in B^u are not reordered, and the projection property holds for u.

It remains to show that the projection property holds if u is a prefix of $p.c$, which is now assumed. Three results need to be established: (a) *pck* slides by all packets in B^u_3, if any, and remains equal to p while doing so; (b) when *pos* reaches the position of the last packet of B^u_2, if any, *pck* is still equal to p; further, when *pos* reaches the position of any packet in B^u_2, u is a prefix of $pck.c$, and $pck.\tilde{v}[j] = b_{pos}.\tilde{v}[j]$; finally, (c) the packets in B^u_1, if any, are simply shifted towards the head of B. For this, let v be the prefix of u of dimension $j-1$.

By definition, if $u[j] = 1$, $B^u_a = B^v_a$, for any $a \in \{1, 2, 3\}$, and so the jth alternative virtual laxities of packets in $B^u \cup \{p\}$ are *always* equal to the $j-1$th alternative virtual laxities of these packets. Therefore, results (a), (b) and (c) follow from the induction assumption on j. It is now assumed that $u[j] = 0$. The following result is useful.

Claim 4.2 *(i)* $B^v_3 \subset B^u_3$*; (ii)* $B^u_2 \subset B^v_2$*; (iii)* $B^v_1 \subset B^u_1$.

Proof. Part (i) follows from Property 2. Part (ii) is now discussed. If $B^u_2 = \emptyset$, the result is trivial. Otherwise, let g be the first packet in B^u_2, and g' the last packet in B^u_2. Since, $g'.\tilde{v}[j] = p.l$, then, by Property 2, $g'.\tilde{v}[j-1] \leq p.l$, by Property 3, the $(j-1)$th alternative virtual laxities of packets in B^v, from g', are consecutive from $g'.\tilde{v}[j-1]$ up to some $a \geq p.l$.

It remains to show that the $(j-1)$th alternative virtual laxities of packets of B^v_2, from the first packet of B^u_2 to the last, are consecutive. For this let b and b' be two adjacent packets of B^v in B (there could be packets of $B - B^v$ between b and b'), such that b precedes b' and $b'.\tilde{v}[j-1] > b.\tilde{v}[j-1] + 1$. Then, by Property 3, $b.\tilde{v}[j-1] = b.\tilde{v}[j] = b.l$, and packets of B^v preceding or equal to b, if any, have laxity smaller than or equal to $b.l$, and, therefore, jth alternative virtual laxity smaller than or equal to $b.l$. By Property 2, packets of B^v, from b to the tail of B, have jth virtual laxity larger than or equal to $b'.\tilde{v}[j-1]$. This completes the proof of part (ii). Part (iii) follows from Part (ii). □

Result (a) is now established. By the induction assumption on j, *pck* slides by packets in $B^v_3 \cap B^u$. Further, by Claim 4.2, the remaining packets of B^u_3, if any, are also in B^v_2. Thus, for result (a), it remains to consider these latter packets.

By definition, all packets in $B^v - (B^u_1 \cup B^u_2)$ have jth virtual laxity larger than $p.l$, possibly infinite. Further, by the induction hypothesis, *pck* is still equal to p and $pck.v = p.l$ when *pos* reaches the position of the last packet of B^v_2. Thus, by Claim 4.2, *pck* also slides by the packets in $B^v - (B^u_1 \cup B^u_2)$, and remains equal to p while doing so. Therefore, result (a) holds. Result (b) is now discussed.

By the discussion in the previous paragraph, when *pos* reaches the position of the last packet of B^u_2, $pck = p$, and $pck.\tilde{v}[j] = p.l$. Thus, from the induction hypothesis, and Claim 4.2, the alternative virtual laxities of *pck* and b_{pos}, from the first to the jth are equal: either packet can be promoted without violating Algorithm D_s. The jth alternative virtual laxity of the packet promoted is decreased by 1. Further, from this point, *pck* is promoted over packets in $B^v_2 - B^u_2$, since these packets have infinite jth

virtual laxity. Moreover, by Property 2, when *pck* reaches the next packet of B_2^u, if any, either *pck* or that packet can be promoted without violating Algorithm D_s, at the level of the jh alternative virtual laxities. The jth alternative virtual laxity of the packet promoted is decreased by 1. The packet promoted may be determined by other coordinates of the virtual laxity vectors. This process continues until the first packet of B_2^u is reached. This shows that Result (b) holds.

Result (c) is now discussed. By Claim 4.2, only values of *pos* equal to the positions of packets in $B_2^v \cap B_1^u$ need to be considered. Let k be the position, in B, of the last packet of $B_2^v \cap B_1^u$. By the induction hypothesis, and from results (a) and (b), when *pos* reaches position k, *pck* is in B^v, and either $pck.v[j] = \infty$, or *pck* is also in B^u and $pck.v[j] > b_k.v[j]$. Thus, *pck* is swapped with b_k without affecting the jth virtual laxity of b_k. Then, after position k, *pck* is promoted over packets of B_2^v that have infinite jth virtual laxity, until the next packet of B_2^v with finite virtual laxity, if any, is reached. Then, by Property 2, the packets are swapped without update to the jth virtual laxity. The process continues until the first packet of $B_2^v \cap B^u$ is reached. Thus, Result (c) holds. Therefore, the projection property of the compete phase of Algorithm D_{lex} holds for u, and, therefore, holds in general.

The projection property of the compete phase of Algorithm D_{lex} clearly implies that Properties $2'$ and $3'$ are true for B'. Therefore, Properties $1'$ through $4'$ are true for B'.

Finally, it must be shown that Properties $1'$ through $4'$ of B' imply Properties 1 through 4 for B before a call to procedure Tick. For this, it must be shown that: first, if $b_1.v[1] = 0$, then a packet is dropped during the insertion, and, second, when the packet is dropped the increase in some virtual laxities does not affect Property 1. The first result follows from the projection property for $u = (0)$ and Claim 4.1. The second result follows from the claim below.

For packet b_i of B', let n_i be the largest integer such that $b_i.\tilde{v}[n_i] = b_i.\tilde{v}[1]$. The dotted lines in part (b) of Figure 9 underline the n_ith virtual laxity for packet b_i for each i. Also, let $b_{r(i)}$ be the first packet of B' that satisfies the following: $b_{r(i)}$ follows or is equal to b_i, $b_{r(i)}.\tilde{v}[1] = b_{r(i)}.l$, and packets between b_i and $b_{r(i)}$ have laxities smaller than or equal to $b_{r(i)}.l$.

Claim 4.3 *Let b_k be the packet dropped when the squeeze phase of procedure M-Insert is run on B', and let b_i be any packet preceding or equal to b_k, in B'. Then, n_i is the dimension of $b_i.c \cap b_k.c$; b_k precedes or is equal to $b_{r(i)}$; packets from b_i to $b_{r(i)}$ are in $B'^{b_i.c \cap b_k.c}$; and, finally, for j, such that $i \le j \le r(i)$, $n_j \ge n_i$.*

Proof. First, it is established in the proof of Claim 4.1 that $b_k.\tilde{v}[1] = b_k.l = k$, and that b_k is the first packet that has equal virtual laxity and laxity. Thus all alternative virtual laxities of b_k are equal to k, and b_k precedes or is equal to $b_{r(i)}$.

Second, the first alternative virtual laxities of packets from b_i to $b_{r(i)}$ are consecutive: from Property $3'$, b_{i^*}, the last packet, such that the first alternative virtual laxities are consecutive from b_i to b_{i^*} has equal first alternative virtual laxity and laxity, and all packets between b_i and b_{i^*} have laxity smaller than or equal to $b_{i^*}.l$.

Third, let u be a prefix of $b_i.c \cap b_k.c$, and m be u's dimension. It is now established that all packets from b_i to $b_{r(i)}$ are in B'^u. If $m = 1$, then Claim 4.3 is trivial. Thus, it is assumed that $m \ge 2$, and, for an argument by induction, that all packets between b_i and $b_{r(i)}$ are in $B'^{u'}$, where u' is the dimension $m-1$ prefix of u. The result is shown below using an argument by contradiction in two cases.

In both cases, the $(m-1)$th alternative virtual laxities of packets between b_i and $b_{r(i)}$ inclusive are consecutive. This comes easily from an argument similar to the one presented in the proof of part (ii) of Claim 4.2.

For the first case, let there be a packet, between b_i and b_k, that is not in B'^u, and let b_j be the last such packet, last from the head of B'. If $b_j.\tilde{v}[m-1] = b_j.\tilde{v}[m]$, then $b_j.\tilde{v}[m] = b_j.l$, since the next possible mth alternative virtual laxity of a packet in $B'^{b_j.c}$ is at least $k+1$; therefore, $b_j.\tilde{v}[1] = b_j.l = j$, which means that b_j should have been dropped before b_k. On the other hand, if $b_j.\tilde{v}[m] \ne b_j.\tilde{v}[m-1]$, then the mth and $m-1$th alternative virtual laxities of b_n, the last packet of B'^u preceding b_j, are equal, and $b_n.\tilde{v}[m] = b_n.l$, since the next possible mth alternative virtual laxity of a packet in B'^u is at least $b_n.\tilde{v}[m-1] + 2$; therefore, $b_n.\tilde{v}[1] = b_n.l = n$, which leads to a contradiction, since b_n precedes b_k.

For the second case, let there be a packet, between b_k and $b_{r(i)}$, that is not in B'^u, and let b_j be the first such packet. Then, $b_{j-1}.\tilde{v}[m] = b_{j-1}.\tilde{v}[m-1] = j-1$, since all packets between b_k and b_{j-1} are in B'^u, and all alternative virtual laxities of b_k are equal to k. Also, $b_j.\tilde{v}[m-1] = j$. Thus, there cannot be a packet in B'^u after b_{j-1} that has mth alternative virtual laxity equal to j. Thus, all packets of B'^u preceding b_{j-1} have laxity smaller than or equal to $j-1$, and $b_{j-1}.l = j-1$. Further, from the first case, all packets between b_i and b_{j-1} are in B'^u. Then this means that $b_{r(i)}$ must precede or be equal to b_{j-1}, which is a contradiction. This completes the proof that all packets from b_i to $b_{r(i)}$ are in B'^u, which implies that $n_i \geq m$.

It remains to show that $n_i \leq m$. For the sake of an argument by contradiction, suppose $n_i > m$. Then $m < M$. Let v be the dimension $m+1$ prefix of $b_i.c$, and let b_j be the last packet of B'^v, between b_i and b_k inclusive, such that $b_j.\tilde{v}[m+1] = b_j.\tilde{v}[m]$. Then, necessarily, b_j precedes b_k, since $b_k \notin B^v$; further, $b_j.\tilde{v}[m+1] = b_j.l$, since the next possible $(m+1)$th alternative virtual laxity of a packet in B'^v is at least $b_j.\tilde{v}[m+1]+2$. Therefore, $b_j.l = j$, which is a contradiction. □

Claim 4.3 implies that if b_i precedes b_k, the packet dropped, then it is precisely the n_i first coordinates of $b_i.v$ that are decreased by the squeeze phase of M-Insert. Hence Property 1, and indeed all four properties, are preserved by M-Insert.

Finally, it is clear that the scheduling, and the passing of time do not affect Properties 1 through 4: the packet at the head of B is scheduled, and the update in the virtual laxities reflect the passing of time. This completes proof of Properties 1 through 4 by induction on time. The remainder of this subsection relates Properties 1 through 4, and Properties $1'$ through $4'$ to the properties of policy D_{lex}. Towards this goal the following proposition is proved.

PROPOSITION 4.1 *Policy D_{lex} is lex-optimal relative to $\mathbf{A_0}$.*

The proof of Proposition 4.1 uses the following results. It is assumed throughout that class identifiers start with a 0.

CLAIM 4.4 *Just before a call to M-Tick, $b_1 \in \Gamma(B)$.*

PROOF. Clearly, for any prefix u of $b_1.c$ of dimension i say, Properties 1 through 4 imply that packets in B^u, with their ith alternative virtual laxities, satisfy the same properties as those satisfied by the packets, and virtual laxities in Algorithm D_s. Therefore, from what was established for Algorithm D_s, $b_1 \in \Phi(B^u)$. Thus, $b_1 \in \mathcal{C}(B)$. It is now shown that b_1 has the highest priority in $\mathcal{C}(B)$.

Let b_j be the first packet in B of some class with priority higher than that of b_1, let u be the vector $b_1.c \cap b_j.c$, let r be u's dimension, and let b_n be the last packet of B^{u1} that precedes b_j. The fact $b_n.c[r+1] = 1$ implies that $b_n.\tilde{v}[r+1] = b_n.\tilde{v}[r]$. The next possible $r+1$th alternative virtual laxity of a packet in B^v is at least $b_n.l+2$, so that $b_n.\tilde{v}[r] = b_n.l$ and all packets in B^v preceding or equal to b_n have laxity smaller than or equal to $b_n.l$. Therefore, as shown in the proof of Theorem 2.4, $\Phi(B^u)$ is included in or equal to the set of the packets of B^u between b_1 and b_n. Thus, b_j and all packets of the same class as b_j are not in $\Phi(B^u)$, which implies that these packets are not in $\mathcal{C}(B)$. Hence, $b_1 \in \Gamma(B)$. □

The following results are used to deal with the dropping performed in Algorithm D_{lex}. So far it was assumed that scheduling policies start in time slot 1 with an initially empty buffer; however, in this section it is convenient to consider a scheduling policy that starts in some time slot t with an initial set of packets A. In such a case, additional packets may arrive at the beginning of time slot t, before any packet is dropped or scheduled in time slot t. The definition of lex-optimality can be naturally extended to lex-optimality starting with A in time slot t.

LEMMA 4.1 *Let A be a set of packet with laxities such that all packets in A start with a 0, let C be a subset of A and π a scheduling policy starting with A lex-optimal relative to $\mathbf{A_0}$ such that whatever the arrival sequence π does not schedule any packet in C. Let π' be any scheduling policy starting with A such that π schedules a packet in $\Gamma(S(\pi') - C)$ whenever $S(\pi') - C \neq \emptyset$. Then π' is lex-optimal relative to $\mathbf{A_0}$.*

PROOF. Since π is lex-optimal, it must be that $A - C \succeq A$. Therefore, by Theorem 2.2, π' is throughput optimal relative to $\mathbf{A_0}$. For the sake of an argument by induction, let π' be order $M-1$ lex-optimal relative to $\mathbf{A_0}$.

By Theorem 2.3, π' is lex-optimal relative to $\mathbf{A_0}$, starting with $A-C$, which implies that π' maximizes the number of packets with Mth bit equal to 0 scheduled over all policies that are order $M-1$ lex-optimal relative to $\mathbf{A_0}$, starting with $A-C$. By the induction hypothesis, π and π' schedule packet with the same $M-1$ first bits whenever either policy schedules a packet; therefore π is order $M-1$ lex-optimal relative to $\mathbf{A_0}$, starting with $A-C$, since π does not schedule any packet in C. Consequently, π' schedules at least as many packets with Mth bit equal to zero as does π.

Since π is lex-optimal relative to $\mathbf{A_0}$, starting with A, π' is also lex-optimal relative to $\mathbf{A_0}$. □

LEMMA 4.2 *Let A and C be two sets of packets with laxities such that, whatever the class vector u, $A^u \equiv C^u$. Let π be a lex-optimal no-early-dropping scheduling policy starting with A, and let π' be a lex-optimal no-early-dropping scheduling policy starting with C. Then, whatever the subsequent arrival sequence, which is taken identical for π and π', in each time slot in which either policy schedules a packet the policies schedule packets of the same class.*

PROOF. By Lemma 3.13 it suffices to prove the following statement for any time slot: for every class vector u, $S(\pi)^u \equiv S(\pi')^u$. The statement is clearly true for the first time slot. For the sake of argument by induction, suppose the statement holds in some time slot. By Proposition 3.2, if $S(\pi) \neq \emptyset$ (equivalently, if $S(\pi') \neq \emptyset$), π schedules a packet in $\Gamma(S(\pi))$ and π' schedules a packet in $\Gamma(S(\pi'))$. Further, by Lemma 3.13, if either policy schedules a packet, the policies schedule packets of the same class, denoted by p for policy π and p' for policy π'. Furthermore, Lemma 3.12 implies that $p \in \Phi(S(\pi)^{p.c})$ and $p' \in \Phi(S(\pi')^{p.c})$. Therefore, by Lemmas 3.3 and 3.6, whatever the class vector u, $T(S(\pi)-p)^u \equiv T(S(\pi')-p')^u$. This implies that the statement holds in the next time slot. This completes the induction argument, proving the statement for all slots. □

The proof of Proposition 4.1 is now undertaken, using an induction on the number of calls to M-Insert. The induction hypothesis is as follows: for some $m \geq 1$, there exists a scheduling policy π, lex-optimal relative to $\mathbf{A_0}$, such that, whatever the arrival sequence, π does not schedule any packet among the packets dropped by D_{lex} in the first m calls to M-Insert, if any.

The induction hypothesis is vacuously true for $m = 0$. The validity of the hypothesis for $m+1$ is now considered, assuming the validity of the hypothesis for m. It suffices to restrict attention to sequences such that the $(m+1)$th call to M-Insert leads to the dropping of a packet. The focus is turned to the array B' from which a packet, b_k say, is dropped during the squeeze phase of the $(m+1)$th call to procedure M-Insert. Let n be the time slot in which the $m+1$th call to M-Insert is completed.

For any $j \in \{0 \ldots k-1\}$, let C_j be the set of packets of B', from b_j to $b_{r(j)}$. By Claim 4.3, $b_k \in C_j$, from which it follows easily that $C_0 \subset C_1 \ldots \subset C_{k-1}$. Also, $C_j \subset B'^{u_j}$, where u_j is the dimension n_j prefix of $b_j.c$, and $T^j(C_j)$ is unschedulable, since C_j contains $r(j)-j+1$ packets, each with laxity in $\{j+1, \ldots, r(j)\}$. Finally, if $0 \leq j < j' < k-1$ then $n_j \leq n_{j'}$. Moreover if $n_j < n_{j'}$ then b_j has strictly higher priority than $b_{j'}$, because by the definition of n_j, $b_j.c[n_j+1] = 1$.

Let k^* be the position of the first packet of class $b_k.c$ in B'. The proof of the induction step is completed by establishing two results. The first result is that if $k^* \geq 1$, then π does not schedule any packet of class $b_k.c$ in time slots n through $n+k^*-1$. The second result, proved using Lemma 4.2, is that π can be extended after time slot $n+k^*-1$ so that π does not schedule b_k.

The first result is now considered. As a starting point it is established that for any $j \in \{0, \ldots, k-1\}$ and any set of packets with laxities A, either $b_j \in \Gamma^{n_j}(T^j(C_j) \cup A)$ or $T^j(C_j) \cap \Gamma^{n_j}(T^j(C_j) \cup A) = \emptyset$.

Since $T^j(C_j) \subset B'^{u_j}$, and $T^j(C_j)$ is unschedulable, by Lemma 3.1, for any prefix v of u_j, $\Phi(T^j(C_j)) \subset \Phi((T^j(C_j) \cup A)^v)$. It is clear that if the virtual laxities and laxities of packets in C_j are decreased by j, and the packets are ordered into an array, array B_j say, from position 0 to position $r(j)-j$, then Claim 4.3 is true for B_j and b_k would still be dropped if the squeeze phase of procedure M-Insert were run on B_j. Part (c) of Figure 9 presents B_3 for an example. Therefore, by the projection property of Algorithm D_{lex} and the proof of Theorem 2.4, $b_j \in \Phi(T^j(C_j))$. Thus, for any prefix v of u_j, $b_j \in \Phi((T^j(C_j) \cup A)^v)$. Therefore, by Corollary 3.3, either $b_j \in \Gamma^{n_j}(T^j(C_j) \cup A)$ or packets in $\Gamma^{n_j}(T^j(C_j) \cup A)$ have priority

higher than that of b_j.

Let k_0 be the minimum $j \in \{0, \ldots, k-1\}$ such that $n_j > n_0$. The result of the previous paragraph implies that any order n_0+1 lex-optimal (relative to $\mathbf{A_0}$) scheduling policy π' starting with B' schedules packets with priority higher than that of b_{k_0} in time slots n through $n+k_0-1$, whatever the arrival sequence. Therefore, the policy π, which by the induction hypothesis is lex-optimal relative to $\mathbf{A_0}$, does not schedule any packet in C_{k_0} in time slots n through $n+k_0-1$.

If $k_0 < k^*$ the same reasoning can be applied for time slots $n+k_0$ through $n+k_1-1$, where k_1 is the minimum $j \in \{k_0, \ldots, k-1\}$ such that $n_j > n_{k_0}$, to show that π does not schedule packets in C_{k_1} in any of those time slots. Continue in this way until k^* is reached, to show that π does not schedule packets in C_k^* in time slots n through $n+k^*-1$. This proves the first result.

The second result is now considered. For j, such that $k^* \le j < k$, $T^j(C_j - b_k)$ is schedulable since the packets in $T^j(C_j - b_k)$ can be scheduled according to their order in B. Thus, $T^j(C_j - b_k) \equiv T^j(C_j)$ and, for any class vector u and set of packets with laxities A, $(A \cup T_j(C_j - b_k))^u \equiv (A \cup T_j(C_j))^u$, since packets in C_j have the same class. Thus, by Lemma 4.2, π can be extended from time slot $n+k^*$ onward in a lex-optimal fashion as follows: in each slot $n+j$ with $k^* \le j < k$ it schedules either b_j or a packet of strictly higher priority. Thus packet b_k is not scheduled by its deadline. In slot $n+k$ and any later slot, any packet from $\gamma(S)$ is scheduled whenever $\gamma(S) \neq \emptyset$. This proves the second result.

The induction hypothesis is thus proved for all m. This implies that there exists a scheduling policy π, lex-optimal relative to $\mathbf{A_0}$, such that π does not schedule any packet dropped by policy D_{lex}, whatever the arrival sequence. Therefore, by Claim 4.4 and Lemma 4.1, policy D_{lex} is lex-optimal relative to $\mathbf{A_0}$. This proves Proposition 4.1.

The proof of Theorem 2.5 is now given. First, by the proof of Proposition 4.1, whatever the arrival sequence in which all class identifiers start with a 0, in each time slot, policy D_{lex} schedules the same packet as a no-early-dropping scheduling policy π that schedules a packet in $\Gamma(S)$ whenever $S \neq \emptyset$. Therefore, by Proposition 3.3, policy D_{lex} is lex-optimal, even if it is assumed that all arrival sequences are stuffed.

Second, let $\mathcal{A}$ be any arrival sequence, and let $\mathcal{A}'$ be the stuffed version of $\mathcal{A}$. Let Algorithm D_{lex} be run on the arrival sequence $\mathcal{A}'$ as follows: in each time slot, first, procedure M-Insert is called to insert the imaginary arrivals from $\mathcal{A}'$, if any, then the real arrivals, if any, are considered. This does not change the properties of policy D_{lex}, since the order in which the new arrivals are considered in Algorithm D_{lex} is assumed arbitrary.

However, running Algorithm D_{lex} on the stuffed sequences, as described above, yields the following result: in each time slot, before insertion of the real arrivals and after the insertion of the imaginary arrivals, because of the nature of the stuffing, the first coordinates of the virtual laxities of packets in B are consecutive from 1 to m^*. This implies that the ordering for the real packets, and the dropping of these packets, during the calls to procedure M-Insert for each of these packets, is simply determined by the coordinates of the virtual laxities of these packets from the second to the Mth.

Thus, policy D_{lex} on $\mathcal{A}'$ is an extension of policy D_{lex} on $\mathcal{A}$. The former is lex-optimal, from earlier discussion; therefore, so is the latter. Theorem 2.5 is proved.

5. Performance of lex-optimal scheduling policies. The results of several simulations are presented in this section to illustrate the interactions among packets with different classes and deadlines under a lex-optimal scheduling policy. There are eight classes in the simulations ($M = 3$). The arrivals for a given class are Bernoulli type, so the arrival rate for a class is also the probability a packet of the class is generated in a time slot. A maximum laxity is defined for each class, and upon arrival the laxity of a packet is selected uniformly over the integers from 1 to the maximum laxity.

In the scenario of Figure 11, all eight classes have maximum laxity 10 and arrival rate α. The common arrival rate α is varied from 0 to 1 and the number of packets of each class scheduled over an interval of 10,000 time slots is indicated in the figure. The mean load is 8α per time slot. The curves are well ordered, with the higher priority classes having the higher throughput. For fixed α, the throughput does not vary linearly with priority. For example, for $\alpha = 0.25$, over 90% of the throughput is given to the four highest priority classes. Classes 000 and 001 have somewhat higher throughput than class 010, and

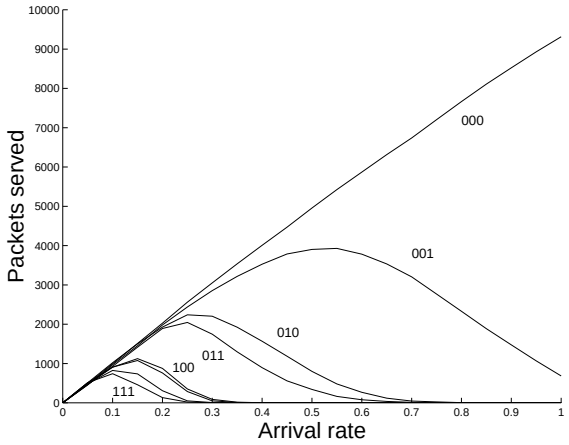

Figure 11: Maximum laxity 10 and varying common arrival rate.

significantly higher throughput than class 0111. For $\alpha = 0.5$, nearly 90% of the throughput is given to the two highest priority classes, with class 001 having roughly three-fourths of the throughput of class 000. The curves for the four highest priority classes and the curves for the four lowest priority classes each form a cluster. This is most pronounced when $0.2 < \alpha < 0.4$. Also, the curves of two classes differing only in their last priority bit tend to be the closest to each other.

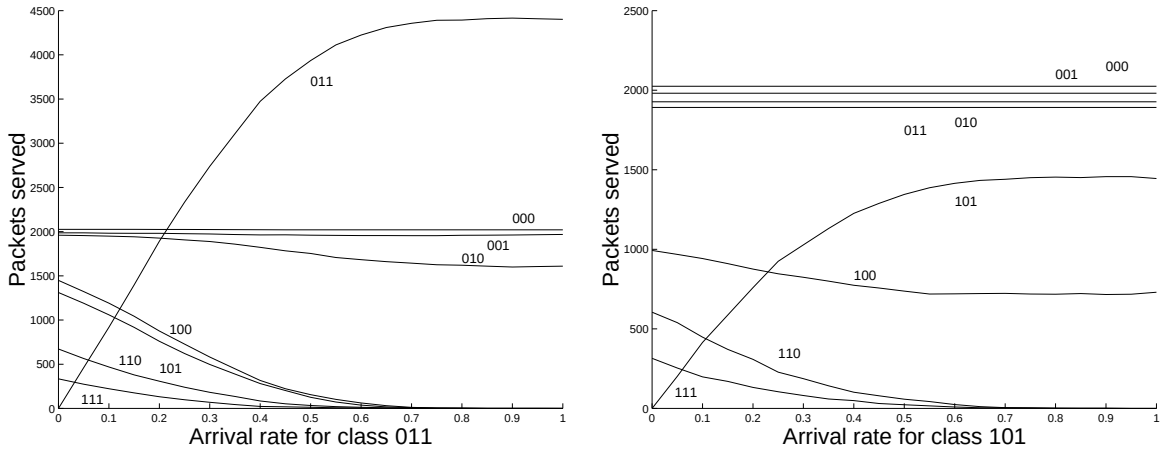

Figure 12: Maximum laxity 10, variable arrival rate for one class, arrival rate 0.20 for other classes.

The first plot in Figure 12 shows the throughputs in case the maximum laxity for all classes is 10, and the arrival rate of class 011 is varied, with the arrival rates of the other classes fixed at 0.20. Since 011 begins with a 0, packets of class 011 can impact the throughput of any other class of packets. As the rate of class 011 approaches 1, the impact on classes 000 and 001 is minimal, while the throughput for class 010 is reduced to about 25% of its value when class 011 traffic is not present, even though class 011 is lower priority than class 010. The second plot in Figure 12 is the same except it is the rate of class 101 that is varied. However the response is substantially different. The throughputs for the first four classes, classes 000 through 011, are not affected at all by the presence of class 100 packets or any packets of lower priority. The class 100 packets, which have a common first zero priority bit with class 101, are markedly affected by the increase in the arrival rate of class 101 packets. The throughput of class 100 drops to about three fourths of its value in the absence of class 101 packets. Even for arrival rate 1, only 14% of the class 101 packets are scheduled, while 35% of the nex-higher-priority class 100 packets are scheduled.

The first plot in Figure 13 shows the throughputs in case the arrival rate for all classes is 0.20, and the maximum laxity of class 011 is varied, with the maximum laxities of the other classes fixed at 10. Most of the variations in throughput are rather minor and do not stand out strongly in the figure. However, as

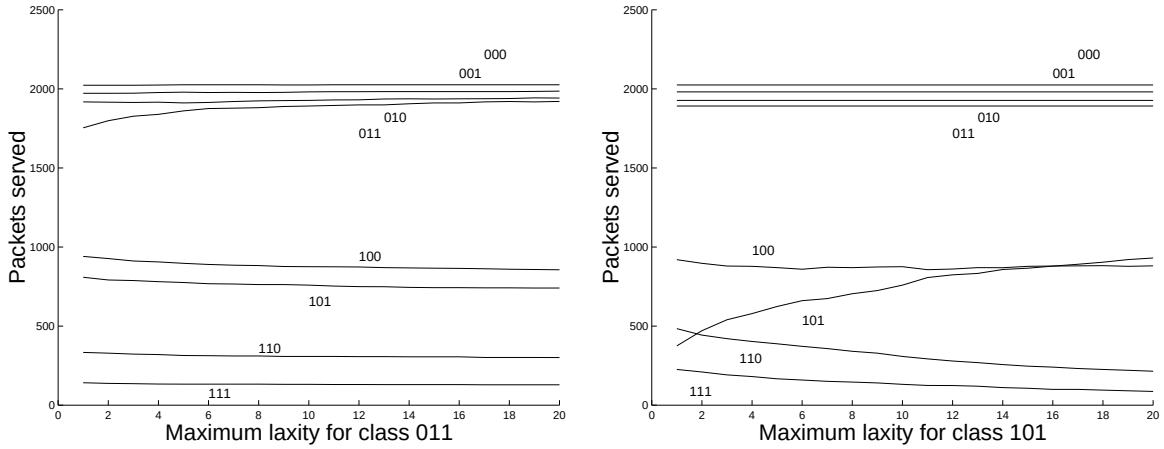

Figure 13: Common arrival rate 0.20, variable maximum laxity for one class, maximum laxity 10 for other classes.

the maximum laxity for class 011 increases, the throughputs for all four highest priority classes increase, while the throughputs of the four lowest priority classes decrease slightly. The increase in the throughput of the highest priority packets is easily explained by the fact these classes have a first bit equal to 0 as does class 011, and are therefore given more opportunities by the increase in the laxity of packets of class 011. On the other hand, packets of the four highest priority classes and those of the four lowest priority classes are scheduled in a strict priority fashion, since classes in each set of classes have a different first bit; thus, the higher the scheduling opportunities for the four highest priority classes the lower those of the four lowest priority classes. The second plot in Figure 13 is the same except it is the maximum laxity of class 101 that is varied. As expected, the throughput for the four highest priority classes is not affected at all by the variation in the laxities of class 101 packets. The throughput for class 101 increases markedly with the increase in laxity, with most of the increase coming at the expense of the two lowest priority classes.

Comparing Figures 12 and 13, we observe that varying the arrival rate of one class has a greater influence than varying the maximum laxity, but the changes are qualitatively similar. The interactions between the traffic and throughput of the classes respect the partition of classes presented in Figure 2, showing how lex-optimality combines aspects of pure-priority scheduling and nested priority scheduling.

6. Conclusion. This paper presents a new optimality criteria and an associated scheduling Algorithm for on-line multiclass scheduling of unit length packets with hard deadlines by a single server in slotted time. It was shown that a no-regret subset of the set of available packets exists for the optimality criteria. Some of the key elements of the proofs are the use of an ordering among sets of packets with laxities, concentration on the case that all arrivals have first class bit 0, strong properties of the data structure maintained by Algorithm D_{lex}, and heavy use of argument by induction.

The simulations performed on lex-optimal scheduling policies suggest that these policies are interesting for practical implementations in which the goal is to mix priorities and maximum throughput in a flexible manner, without the need to assign numerical values to the packets.

Acknowledgments. We thank the reviewers for suggestions regarding the presentation. This work was supported in part by the National Science Foundation under grant NSF ANR 99-80544.

References

[1] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha, *On-line scheduling in the presence of overload*, IEEE Symp. on Foundations of Comp. Science, 1991, pp. 100–110.

[2] G.C. Buttazo, *Hard real-time computing systems: Predictable scheduling algorithms and applications*, Kluwer Publishers, Norwell, MA, 1977.

[3] R. Chipalkatti, J. F. Kurose, and D. Towsley, *Scheduling policies for real-time and non-real-time traffic in a statistical multiplexer*, Proc. IEEE INFOCOM, 1989, pp. 774–783.

[4] M.R. Garey, D. S. Johnson, B. B. Simons, and R. E. Tarjan, *Scheduling unit-time tasks with arbitrary release times and deadlines*, SIAM J. Comput. **10** (1981), 259–269.

[5] B. Hajek, *On the competitiveness of on-line scheduling of unit-length packets with hard deadlines in slotted time*, Proc. 2001 Conf. on Information Sciences and Systems, Johns Hopkins, 2001, pp. 434–438.

[6] P. Hruschka and L. Jackson, *Real-time systems: Investigating industrial practice*, Wiley Series in Software Based Systems, England, 1993.

[7] J. J. Hunter, *Mathematical techniques of applied probability, volume 2, discrete time models: Techniques and applications*, Academic Press, New York, 1983.

[8] E.L. Lawler, *Combinatorial optimization: Networks and matroids*, Rinehart and Winston, New York, 1976.

[9] T.L. Ling and N. Shroff, *Scheduling real-time traffic in ATM networks*, Proc. IEEE INFOCOM, 1996, pp. 198–205.

[10] C.L. Liu and J. W. Layland, *Scheduling algorithms for multiprogramming in a hard-real-time environment*, Journal of the ACM **20** (1973), 46–61.

[11] C. D. Locke, *Best-effort decision making for real-time scheduling*, Ph.D. thesis, Computer Science Department,Canergie-Mellon University, 1986.

[12] A.K. Mok, *Towards mechanization of real-time system design*, Foundations of Real-Time Computing: Formal Specifications and Methods (A.M. van Tilborg and G. M. Koob, eds.), Kluwer, Norwell, MA, 1991.

[13] C.H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: Algorithms and complexity*, Prentice Hall, 1982.

[14] J.M. Peha and F. A. Tobagi, *Evaluating scheduling algorithms for traffic with heterogeneous performance objectives*, Proc. IEEE GLOBECOM, 1990, pp. 460–465.

[15] L.L. Peterson and B. S. Davie, *Computer networks*, 3rd ed., Morgan Kaufmann, San Francisco, 2003.

[16] S. Pingali and J. F. Kurose, *On scheduling two classes of real-time traffic with identical deadlines*, Proc. IEEE GLOBECOM, 1991, pp. 460–465.

[17] J.A. Stankovic and K. Ramamritham (eds.), *Hard real-time systems*, The Computer Society, Washington, D.C., 1988.

[18] A.M. van Tilborg and G. M. Koob (eds.), *Foundations of real-time computing:scheduling and resource management*, Kluwer, Norwell, MA, 1991.