An Experimental Study of New and Known Online Packet Buffering Algorithms

Susanne Albers* Tobias Jacobs†

Abstract

We present the first experimental study of online packet buffering algorithms for network switches. The design and analysis of such strategies has received considerable research attention in the theory community recently. We consider a basic scenario in which m queues of size B have to be maintained so as to maximize the packet throughput. A *Greedy* strategy, which always serves the most populated queue, achieves a competitive ratio of only 2. Therefore, various online algorithms with improved competitive factors were developed in the literature.

In this paper we first develop a new online algorithm, called *HSFOD*, which is especially designed to perform well under real-world conditions. We prove that its competitive ratio is equal to 2.

The major part of this paper is devoted to the experimental study in which we have implemented all the proposed algorithms, including *HSFOD*, and tested them on packet traces from benchmark libraries. We have evaluated the experimentally observed competitivess, the running times, memory requirements and actual packet throughput of the strategies. The tests were performed for varying values of m and B as well as varying switch speeds. The extensive experiments demonstrate that despite a relatively high theoretical competitive ratio, heuristic and greedy-like strategies are the methods of choice in a practical environment. In particular, *HSFOD* has the best experimentally observed competitiveness.

*Department of Computer Science, University of Freiburg, Georges Köhler Allee 79, 79110 Freiburg, Germany. salbers@informatik.uni-freiburg.de Work supported by the Deutsche Forschungsgemeinschaft, projects AL 464/4-1 and 4-2.

†Department of Computer Science, University of Freiburg, Georges Köhler Allee 79, 79110 Freiburg, Germany. jacobs@informatik.uni-freiburg.de

1 Introduction

Over the past five years the algorithms community has witnessed tremendous research interest in packet buffering algorithms, see e.g. [1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 16, 19, 20, 21, 22, 23, 25] for a selection of the work. Given a network router or switch that is equipped with packet buffers of limited capacity, the general goal is to design strategies for serving these buffers so as to maximize the total packet throughput. While packet buffering policies have been investigated in the applied computer science and, in particular, networking communities for many years, only seminal papers by Aiello et al. [2] and Kesselman et al. [19] have initiated theoretical and algorithmic studies. These studies aim at analyzing existing algorithms and at designing new strategies with a provably good performance.

Obviously, packet buffering is an online problem in that data packets arrive over time and, at any time, future packet arrivals are unknown. Results from queueing theory cannot be applied directly as network traffic exhibits so-called *self-similar* properties, cf. [14, 29]. Therefore, algorithmic research resorts to competitive analysis [27], comparing an online algorithm A to an optimal offline algorithm OPT that knows the entire packet arrival sequence in advance. Algorithm A is called c-competitive if, for all packet arrival sequences, the throughput achieved by A is at least $1/c$ times that of OPT. In the above-mentioned algorithmic body of work, various packet buffering problems were investigated. The following natural questions arise: Do the competitive analyses give meaningful results? Are the proposed new algorithms interesting from a practical point of view? Does optimizing the worst-case behaviour also improve the practical performance? So far, these issues were not addressed.

In this paper we present the first experimental study of online packet buffering algorithms. We consider a scenario that is very basic and has been investigated the most among the proposed models, see [3, 6, 7, 19, 25]. Specifically, we are given m packet buffers, each of which is associated with an input port of a switch. Each buffer is organized as a queue and can simultaneously store up to B data packets. The capacity B is also referred to as the *size* of the buffer. Time is assumed to be discrete. Each time step consists of two phases, namely a *packet arrival phase* and a *packet transmission phase*. At any time, in the packet arrival phase, new packets may arrive at the buffers. Let b_i be the number of packets currently stored in buffer i, and let a_i be the number of newly arriving packets at that buffer. If $a_i + b_i \leq B$, then all new packets can be accepted; otherwise $a_i + b_i - B$ packets must be dropped. Furthermore, at any time, in the packet transmission phase, an algorithm can select one non-empty buffer and transfer the packet at the head of that queue to the output port. We assume w.l.o.g. that the packet arrival phase precedes the transmission phase. The goal is to maximize the throughput, i.e. the total number of transferred packets.

The scenario we study here arises, for instance, in input-queued (IQ) switches which represent the dominant switch architecture today. In an IQ switch with m input and m output ports packets that arrive at input i and have to be routed to output j are buffered in a virtual output queue Q_{ij}. In each time step, for any output j, one data packet from queues Q_{ij}, $1 \leq i \leq m$, can be sent to that output. A small IQ switch with three input and three output ports is depicted in Figure 1. In our problem formulation the m buffers correspond to queues Q_{ij}, $1 \leq i \leq m$, for any fixed j. We emphasize that we consider all packets to be equally important, i.e. all of them have the same value. Most current networks, in particular IP networks, treat packets from different data streams equally in intermediate switches.

Known algorithms: The most simple and natural packet buffering algorithm is the *Greedy* policy: At any time serve the queue currently storing the largest number of packets. Unfortunately, *Greedy* has essentially the worst possible competitive ratio. It is easy to show [3, 7] that any *work conserving* algorithm, which at any time serves an arbitrary non-empty buffer, is 2-competitive. Obviously, *Greedy* belongs to the class of work conserving strategies. It was shown in [3] that the competitive ratio of *Greedy* is not smaller than $2 - 1/B$, no matter how ties are broken. Thus *Greedy* has a competitiveness of exactly 2, for arbitrary buffer sizes. The first deterministic algorithm that achieved a competitive ratio below 2 was devised in [3]. The proposed *Semi Greedy* algorithm deviates from standard *Greedy* when the buffer occupancy is low and has a competitive performance of $17/9 \approx 1.89$. The deterministic

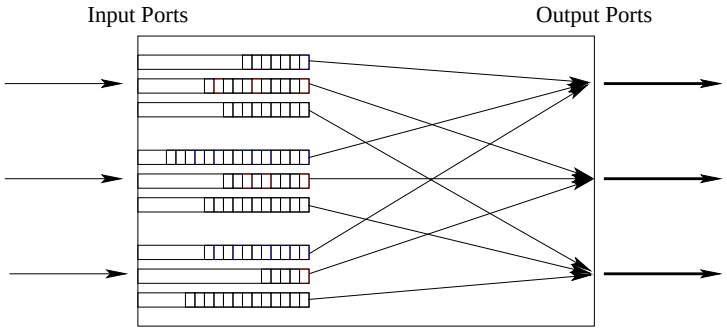

Figure 1: An IQ switch.

strategy with the smallest competitive ratio known is the *Waterlevel* algorithm [6] with a competitiveness of $\frac{e}{e-1}(1+\frac{\lfloor H_m+1\rfloor}{B})$, where H_m is the m-th Harmonic number. This ratio is optimal, for large B, as no deterministic algorithm can have a competitive ratio smaller than $e/(e-1) \approx 1.58$, see [3]. As for randomized strategies, a *Random Schedule* algorithm [7] achieves a competitive ratio of $e/(e-1)$ while a *Random Permutation* algorithm is 1.5-competitive [25]. These performance ratios hold against oblivious adversaries and are close to the best lower bound of 1.46, see [3]. The five algorithms just mentioned comprise all online strategies known in the literature for our packet buffering problem. As for the offline problem, a polynomial time algorithm computing optimal solutions was given in [3].

Contributions of this paper: We first introduce a new online packet buffering algorithm called *HSFOD*. It is based on the idea to estimate the packet arrival rate for each port. In each time step the algorithm transmits a packet from a non-empty queue that, according to these arrival rates, encounters packet loss earliest in the future assuming buffers would not be served anymore. This new strategy is presented in Section 2. We prove that it achieves a competitive ratio of 2.

The major part of this paper is devoted to an extensive experimental study of the packet buffering problem under consideration. The main purpose of our experiments is to determine the experimentally observed competitiveness of all the proposed online algorithms and to establish a relative performance ranking among the strategies. As the name suggests, the experimentally observed competitiveness is the ratio of the throughput of an online algorithm to that of an optimal solution as it shows in experimental tests. Additionally, we wish to evaluate the running times and memory requirements of the algorithms as some of the strategies are quite involved and need auxiliary data structures. Finally, we are interested in the actual throughput in terms of the total number of successfully transmitted packets.

In order to get realistic and meaningful results, we have tested the algorithms on real-world traces. We selected traces from the Internet Traffic Archive [18], which is a moderated trace repository sponsored by ACM SIGCOMM. In our experiments we have studied varying port numbers m as well as varying buffers sizes B. Furthermore, we have investigated the influence of varying the *speed* of a switch, i.e. the frequency with which it can forward packets. We have adjusted this parameter relative to the given data traces. For instance, a speed of value 1 indicates that the *average* packet arrival frequency is equal to the frequency with which packets can be transmitted.

In Section 3 we present a concise description of the five previously known online buffering algorithms as well as the optimal offline strategy. For all the proposed strategies, including *HSFOD*, we describe how the given pseudo-code was indeed implemented and discuss runtime issues as well as extra space requirements of the strategies. We implemented the data model and the algorithms using the Java programming language. The test environment is described in Section 4. A detailed presentation of the results follows in Section 5. One of the most important findings is that the experimentally observed competitiveness is much lower than the theoretical bounds. Typically, the online algorithms are at most 3%

worse than an optimal offline algorithm. In fact, *HSFOD* shows the best performance, having a gap of less and 0.1%. We remark here that *HSFOD* was designed after we had implemented and evaluated the previously known algorithms. Hence it can be viewed as a result of an algorithm engineering process. Furthermore, the theoretical competitive ratios are no proper indication of how the algorithms perform in practice. The randomized algorithms, despite their low theoretical competitiveness, do not perform better than the deterministic ones. From a practical point of view *Greedy*, *HSFOD* and *Semi Greedy* are the algorithms of choice. Section 6 summarizes the main results of our study.

2 The new algorithm *HSFOD*

The new strategy we introduce estimates future packet arrival rates by keeping track of past arrival patterns. Based on these arrival rates, the algorithm mimics the optimal offline algorithm *SFOD* [3] which at each point of time transmits a packet from a non-empty queue that would overflow earliest in the future if no queues were served.

Algorithm HSFOD: For each input port, the algorithm maintains a weighted moving average estimating the packet arrival rate at that port. These estimates are updated after each packet arrival phase. For $i \leq i \leq m$, let $r_i(t)$ be the rate at port i at time t. Then

$$r_i(t) = \alpha \cdot r_i(t-1) + (1-\alpha) \cdot a_i(t),$$

where $a_i(t)$ is the number of packets that have just arrived at port i, and $\alpha \in (0,1)$ is some fixed constant. We set $r_i(0) = 0$ initially. The overflow time for each port i is calculated as $t_i^{\text{ov}}(t) = (B - b_i(t))/r_i(t)$, where $b_i(t)$ is the number of packets currently stored in buffer i. Note that $r_i(t)$ and $t_i^{\text{ov}}(t)$ are allowed to take fractional values. At any time the algorithm serves the buffer that has the smallest overflow time; ties may be broken arbitrarily.

Theorem 1 *The competitive ratio of* HSFOD *is exactly equal to 2.*

Proof. As *HSFOD* is work conserving, i.e. always transmits a packet when there is a non-empty buffer, it has a competitive ratio of 2 (see [3]). The following packet arrival pattern shows the tightness of that bound. The packets arrive in $m-1$ phases. In phase $i = 1, \ldots, m-1$ there arrive B packets at port $i+1$ and a huge amount of $A_i \gg B$ packets at port i, such that *HSFOD* transmits only packets from the latter buffer in the next B time units. It is easy to observe that such an A_i exists for any $0 < \alpha < 1$. No further packets arrive until buffer i has been completely emptied by *HSFOD*, which also marks the end of phase i. The adversary algorithm ADV transmits only packets from buffer $i+1$, which is thus empty by the end of the phase. After $m-1$ phases, *HSFOD* stores a total of exactly B packets in its buffers, while ADV stores $(m-1) \cdot B$ packets. So after $(m-1) \cdot B$ additional time steps without packet arrival, *HSFOD* has transmitted a total number of $m \cdot B$ packets, while ADV has been able to transmit $(2m-2) \cdot B$ packets. The ratio converges to 2 for large m. □

HSFOD depends on a parameter α, where $0 < \alpha < 1$, that weights past and current packet arrivals and hence determines the length of the *HSFOD*'s memory. For larger values of α, the long-term packet arrival rate has a higher weight than short-term changes. Good buffering policies make the greatest difference when the speed of a switch takes values around 1, see Section 5 that reports on the experimental results. In this case the average data rate at any port is $1/m$, which means that a packet arrives only about every mth time step. Since the short-term arrival rate fluctuates more (from 0 to 1, from 1 to 0) than the long-term rate, the short-term arrival rate does not provide a realistic estimate of the overflow time. Thus, reasonable values of α will be close to 1.

Figure 15 in the Appnedix shows the experimentally observed competitiveness of *HSFOD* for values of α between 0.9 and 1. The plot shows the results for the data set that will be the representative trace

throughout Section 5. We observe that all ratios are nearly 1. The best results are achieved for values of α with $0.99 \leq \alpha \leq 1$. This interval is considered in more detail in Figure 16 in the Appendix. Here values of α with $0.995 \leq \alpha \leq 0.997$ lead to the lowest ratios. Hence we set α to 0.997 in our experiments.

3 The implemented algorithms

In this section we present the packet buffering algorithms we have evaluated experimentally, i.e. all the previously known algorithms in addition to *HSFOD*. We describe some implementation issues, optimizing running time and memory requirements of the strategies. We note that our experiments of course represent an algorithmic simulation rather than a switch hardware realization of the buffering strategies.

As we consider a scenario where all packets have the same value, unless otherwise stated, the algorithms apply a greedy admission policy: At any time t and for any of the m buffers, whenever new data packets arrive, an algorithm accepts as many packets as possible subject to the constraint that a buffer can only store up to B packets simultaneously. Thus the algorithms we present here only specify which buffer to serve in each time step. We will use the terms *buffer* and *queue* interchangeably and use q_i to refer to the i-th buffer/queue. Let the *load* of a queue be the number of packets currently stored in it.

3.1 Deterministic online algorithms

We first state the *Greedy* and *Semi Greedy* policies.

Algorithm Greedy: In each time step serve the queue currently having the maximum load; ties may be broken arbitrarily.

In our implementation of *Greedy*, we break ties by choosing the buffer with the smallest index. Furthermore, using a standard heap data structure, we determine the most populated queues in worst case time $O(\log m)$.

Algorithm Semi Greedy: In each time step execute the first of the following three rules that applies to the current buffer configuration. (1) If there is a queue buffering more than $\lfloor B/2 \rfloor$ packets, serve the queue currently having the maximum load. (2) If there is a queue the hitherto maximum load of which is less than B, then among these queues serve the one currently having the maximum load. (3) Serve the queue currently having the maximum load. In each of the three rules, ties are broken by choosing the queue with the smallest index. Furthermore, whenever all queues become empty, the hitherto maximum load is reset to 0 for all queues.

In our implementation we use two priority queues based on standard heaps. The first one stores the load of all the queues. The second one stores the load of those queues whose hitherto maximum load is less than B. With the help of these auxiliary data structures, we can determine in $O(\log m)$ worst case time which queue to serve.

We next give a condensed presentation of the *Waterlevel* strategy. In the original paper [6] the description was more general. *Waterlevel* is quite involved and consists of a cascade of four algorithms that simulate each other. At the bottom level there is a fractional *Waterlevel* algorithm, denoted by *FWL*, that allows us to process fractional amounts of packets. *FWL* is based on the fact that a packet switching schedule can be viewed as a matching that maps any time step t to the packet p transmitted during that step. For any packet arrival sequence σ, consider the following bipartite graph $G_\sigma = (U, V, E)$ in which vertex sets U and V represent time steps and packets, respectively. If T is the last point in time at which packets arrive in σ, then packets may be transmitted up to time $T + mB$. Thus, for any time t, $1 \leq t \leq T + mB$, set U contains a vertex u_t. For any packet p that ever arrives, V contains a vertex v_p. Let P_t^i be the set of the last B packets that arrive at queue q_i until (and including) time t and let $P_t = \cup_{i=1}^m P_t^i$. The set of edges is defined as $E = \{(u_t, v_p) \mid p \in P_t\}$.

In a standard matching, each edge is either part of the matching or not, i.e. for any edge $(u_t, v_p) \in E$ we can define a variable x_t^p that takes the value 1 if the edge is part of the matching and 0 otherwise. In a fractional matching we relax this constraint and allow $x_t^p \in [0,1]$. Intuitively, x_t^p is the extent to which packet p is transmitted during time t. Of course, at any time t, a total extent of at most 1 can be transmitted, i.e. $\sum_{p \in P_t} x_t^p \leq 1$, and over all time steps any packet p can be transmitted at most once, i.e. $\sum_{t=1}^{T+mB} x_t^p \leq 1$.

The graph G_σ evolves over time. At any time t, $1 \leq t \leq T + mB$, a new node u_t is added to U and new packet nodes, depending on the packet arrivals, may be added to V. A switching algorithm has to construct an online matching, mapping time steps (fractionally) to data packets that have arrived to far. The idea of the *Waterlevel* algorithm is to serve the available data packets as evenly as possible. For any time t and any packet p, let $s_t^p = \sum_{t'<t} x_t^p$ be the extent to which p has been served to far. The fractional *Waterlevel* algorithm *FWL* works as follows.

Algorithm FWL: At any time t, for any packet $p \in P_t$ match an extent of $x_t^p = \max\{h - s_t^p, 0\}$, where h is the maximum number such that $\sum_{p \in P_t} x_t^p \leq 1$.

The goal of the following steps is to discretize *FWL*. This is done by admitting only full, integral packets to the buffers and by transmitting only full, integral packets. In order to guarantee the same throughput as *FWL* one employs slightly larger buffers.

Algorithm FWL': Work with queues of size $B+1$ and run a simulation of *FWL* on queues of size B. At any time t, in the packet arrival phase, accept as many packets as possible subject to the constraint that only complete packets may be accepted. In the transmission phase, at any time t, transmit a total amount of X_t^i from queue q_i, where X_t^i is the total amount transferred by *FWL* from queue q_i. If $\sum_{i=1}^m X_t^i < 1$, then transmit an amount of $1 - \sum_{i=1}^m X_t^i$ from arbitrary non-empty queues as long as there are such.

While the last algorithm discretized the arrival step, the next one discretizes the transmission step.

Algorithm D(FWL'): Work with queues of size $B + 1 + \lfloor H_m \rfloor$. Run a simulation of *FWL'* with queues of size $B+1$. At any time t and for any queue q_i, let S_i be the total number of packets transmitted from queue q_i by *D(FWL') before* time t and let S_i' be the total amount of packets from queue q_i transmitted by *FWL'* up to (and including) time t. Transmit a packet from the queue for which the residual service extent $S_i' - S_i$ is largest.

In a last step we take care of the large buffer sizes.

Algorithm Waterlevel: Work with queues of size B. Run a simulation of *D(FWL')*. In each time step, accept a packet if *D(FWL')* accepts it and the corresponding queue is not full. Transmit packets as *D(FWL')* if the corresponding queue is not empty.

Obviously, *Waterlevel* is expensive with respect to both running time and space. At first sight it may seem that each of the simulated algorithms *FWL*, *FWL'* and *D(FWL')* needs an extra space of $\Theta(mB)$. However, this does not hold true. For each simulated algorithm it suffices to just keep track of the current load in each queue.

We next describe an efficient implementation of *FWL*. At any time t and for any packet $p \in P_t$, the algorithm has to determine the extent to which p is served. To this end, the service extents s_t^p are crucial. Note that P_t contains at most mB packets, namely the last B packets that have arrived at each of the m queues. In our implementation we maintain a doubly-linked list L of all the s_t^p values, $p \in P_t$, sorted in increasing order. For each entry in the list we store a vector of length m indicating how many packets in q_i, $1 \leq i \leq m$, currently take that value. The values in L together with the total number of packets taking a certain value give rise to a waterlevel profile depicted in Figure 2. Each level of the profile represents a value in L. The width of a level corresponds to the total number of packets $p \in P_t$ having a service extent s_t^p equal to that level. In each time step at which new data packets arrive, we have to update L. This is done by first adding a waterlevel of height $s = 0$ at the head of L, storing for each queue q_i the number n_i of newly arrived packets. If, for queue q_i, the previous load l_i plus n_i exceeds B, then we

have to discard the oldest $n_i' = l_i + n_i - B$ packets from q_i. This corresponds to a proper update of P_t. Algorithm *FWL* ensures that packets residing longer in q_i have larger service extents. Thus, starting at the tail of L, we discard for any q_i the oldest n_i' packets. This is done by simply decreasing the number of packets from q_i that contribute to a waterlevel until a total number of n_i' has been discarded.

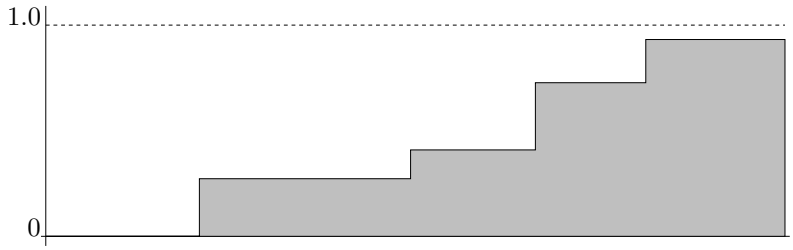

Figure 2: The waterlevel profile

The computation of the x_t^p values amounts to filling water of volume 1 into the waterlevel profile. More specifically, we repeatedly have to find out which adjacent waterlevels to merge. Each merge operation can be performed in $O(m)$ time. Simultaneously, while raising waterlevels, we keep track of the extents X_t^i to which packets from q_i are being served.

Algorithm HSFOD: The algorithm was described in Section 2.

After each packet arrival phase, for each port, the arrival rate has to be updated and the expected overflow time has to be computed. Unlike in the implementation of the greedy-like strategies, a use of priority queues is not sensible here.

3.2 Randomized algorithms

We first present the algorithm *Random Schedule*. In addition to the m packet queues the algorithm maintains m auxiliary queues, each of size B, which are initially empty. Over time the auxiliary queues will contain real numbers from the range $(0, 1)$, which serve as priorities. These priorities may be labeled as either marked or unmarked. In the following, $q_1, \ldots, q_m$ will refer to the original packet queues and $Q_1, \ldots, Q_m$ to the auxiliary queues.

Algorithm Random Schedule: At any time execute the following two steps.

1. In the packet arrival phase, for any new packet admitted to a queue q_i, choose a real number uniformly at random from $(0, 1)$ and append it to Q_i. If Q_i was full prior to this operation, then first delete the element at the head of Q_i. The newly inserted number is labeled unmarked.
2. In the transmission phase, check if the $Q_1, \ldots, Q_m$ store unmarked numbers. If so, let Q_i be the queue storing the largest unmarked number; ties may be broken arbitrarily. Change the label of that number to marked and transmit a data packet from queue q_i. Otherwise, if there are no unmarked numbers, transmit a packet from an arbitrary non-empty queue.

We remark that *Random Schedule* uses a considerable amount of $\Theta(mB)$ extra space to store the auxiliary queues $Q_1, \ldots, Q_m$. Additionally, in our implementation we maintain a priority queue based on standard heaps that stores the unmarked numbers from $Q_1, \ldots, Q_m$. Whenever a new data packet is admitted to a packet buffer q_i, we have to insert a number into the priority queue. This operation may be preceded by a *delete* operation if a number first has to be removed from the head of Q_i. Executing a *deletemax* operation, we can determine which of the packet buffers to serve. All the operations may take up to $O(\log(mB))$ time. We remark that numbers deleted from the heads of the $Q_1, \ldots, Q_m$ may be unmarked. Therefore, we explicitly have to maintain $Q_1, \ldots, Q_m$ and it is not sufficient to just store the priority queue of unmarked numbers. In our experiments we have also tested a priority

queue implementation based on Fibonacci heaps. In this case the running time of *Random Schedule* decreases by about 20% but the memory requirements increase by a factor of 10 due to the complex pointer structure of Fibonacci heaps. As we will see in Section 5, even with standard heaps *Random Schedule* has extremely high extra memory requirements, and we therefore did not use Fibonacci heaps in our tests.

The second randomized switching algorithm known is called *Random Permutation*. The basic approach of the algorithm is to reduce the packet switching problem with m buffers of size B to one with mB buffers of size 1. To this end, a packet buffer q_i of size B is associated with a set $Q_i = \{q_{i,0}, \ldots, q_{i,B-1}\}$ of B buffers of size 1. A packet arrival sequence σ for the problem with size B buffers is transformed into a sequence $\tilde{\sigma}$ for unit-size buffers by applying a Round Robin policy. More specifically, the j-th packet ever arriving at q_i is mapped to $q_{i,j \bmod B}$ in Q_i. *Random Permutation* at any time runs a simulation of the following algorithm *SimRP* for $m' = mB$ buffers of size 1.

Algorithm SimRP(m'): The algorithm is specified for m' buffers for size 1. Initially, choose a permutation π uniformly at random from the permutations on $\{1, \ldots, m'\}$. In each step transmit the packet from the non-empty queue whose index occurs first in π.

The algorithm for buffers of arbitrary size then works as follows.

Algorithm Random Permutation: Given a packet arrival sequence σ that arrives online, run a simulation of *SimRP(mB)* on $\tilde{\sigma}$. At any time, if *SimRP(mB)* serves a buffer from Q_i, transmit a packet from q_i. If the buffers of *SimRP(mB)* are all empty, transmit a packet from an arbitrary non-empty queue if there is one.

Obviously, the algorithm needs a large amount of $\Theta(mB)$ extra space to run *SimRP(mB)*. Using a priority queue that stores non-empty buffers of capacity 1, we can determine in $O(\log(mB))$ time which queue to serve.

3.3 An optimal offline algorithm

In order to compare the performance of the online algorithms to that of an optimal solution, we implemented the algorithm *SFOD* [3], which was proven to be an optimal offline strategy. As *SFOD* is just used for comparison, we only state the algorithm without discussing details of the implementation.

Algorithm SFOD: At any time serve the non-empty buffer that encounters packet loss earliest in the future assuming buffers would not be served anymore; ties may be broken arbitrarily. If there is no such buffer, serve an arbitrary non-empty queue.

4 The test environment

We have tested the online packet buffering algorithms on real-world traces from the Internet Traffic Archive [18], a moderated trace repository maintained by ACM SIGCOMM. We have performed extensive tests with seven traces whose characteristics are summarized in Table 1. A first set of four traces monitors wide-area traffic between Digital Equipment Corporation (DEC) and the rest of the world. A second set of three traces monitors wide-area traffic between the Lawrence Berkeley Laboratory (LBL) and the rest of the world. Only the TCP traffic was considered. Despite being a number of years old, these traces still represent standard benchmarks when marking experimental tests and are recommended for such studies, see e.g. the text book by Krishnamurthy and Rexford [24] or [28]. The traces were gathered in over a time horizon of one to two hours and consist of 1.3 to 3.8 million data packets each. In the various traces the information relevant to us is, for any data packet, the arrival time and the sending host address. For the sake of anonymity, the latter addresses were renumbered in the original traces.

As indicated in the introduction the main goal of our experiments is to determine the experimentally observed competitiveness of the online switching algorithms and to establish a relative performance

Name	Date	# Packets	Place
DEC-PKT-1	08.03.1995 22.00–23.00	2.1 mio	DEC
DEC-PKT-2	09.03.1995 02.00–03.00	2.6 mio	DEC
DEC-PKT-3	09.03.1995 10.00–11.00	2.8 mio	DEC
DEC-PKT-4	08.03.1995 14.00–15.00	3.8 mio	DEC
LBL-PKT-4	21.01.1994 14.00–15.00	1.3 mio	LBL
LBL-PKT-5	28.01.1994 14.00–15.00	1.3 mio	LBL
LBL-TCP-3	20.01.1994 14.10–16.10	1.8 mio	LBL

Table 1: Packet traces

ranking among the strategies. Furthermore, we are interested in the algorithms' running times and memory requirements. As for the running time of a strategy, we evaluated the *average time* it takes the algorithm to determine which queue to serve (total running time summed over all time steps/#time steps). This time is easy to determine using timers provided by the Java library. As for extra space requirements, we have evaluated, for any of the algorithms, the maximum amount of memory needed by auxiliary data structures employed by that algorithms. This analysis was performed using the Java class `ObjectOutputStream`. Finally in our tests, we have evaluated the actual throughput in terms of the number of data packets transferred.

In our experiments we have studied varying port numbers m as well as varying buffers sizes B. In order to be able to investigate varying values of m, we have to map sending host addresses (e.g. about 3000 in DEC-PKT-1) to port number numbers in the range $\{0, \ldots, m-1\}$. We chose a mapping that maps each sending host address to a port number chosen uniformly at random from $\{0, \ldots, m-1\}$. We would like to point out that such a mapping does not lead to balanced traffic at the ports as some hosts generate a large number of packets. In our traces, under the random mapping, we observe highly non-uniform packet arrival patterns where 10 to 15% of the ports receive ten times as many packets as each of the other ports. This is consistent with the fact that web traffic with respect to packets' source (and destination) addresses is distributed non-uniformly, exhibiting essentially a power-law structure [13, 28]. Typically, 10% of the hosts account for 90% of the traffic. However, this fact does not allow a direct conclusion on the distribution of sending hosts among the input ports of a switch. This distribution strongly depends on the network topology. So, alternatively, a power-law governed assignment of hosts to ports is not more reasonable than our uniform distribution.

Another important parameter in the experimental tests is the speed of the switch, i.e. how fast the switch can transfer packets. Here we consider speed values relative to the data volume of a given trace. For a trace data set D, let $f_D = (\#\text{packets in } D)/(\text{length of time horizon of } D)$ be the average packet arrival rate in D. Speed s indicates that the switch forwards data packets with frequency sf_D. Thus, intuitively, a speed 1 switch can forward the data exactly as fast as it arrives on the average. If the speed is low, inevitably, buffers tend to be highly populated. If the speed is high, buffers are only lightly loaded. In summary, each of our experiments is specified by the following parameters: (a) switching algorithm A; (b) trace data set D; (c) number m of buffers; (d) buffer size B; (e) speed s.

5 Experimental results

We have done extensive tests with all the network traces mentioned in Section 4. A first, very positive finding is that the results are consistent for all the traces. The phenomena reported in this section, unless otherwise stated, have occurred for all the data sets. Due to space limitations, in this paper we only present the plots for trace DEC-PKT-1. A zip-file containing the plots for all the traces can be downloaded at `http://www.informatik.uni-freiburg.de/~jacobs/`. In the following

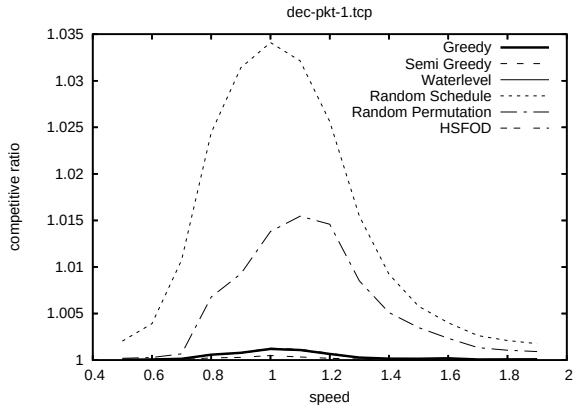

Figure 3: Competitive ratio, $m = 30$ and $B = 100$

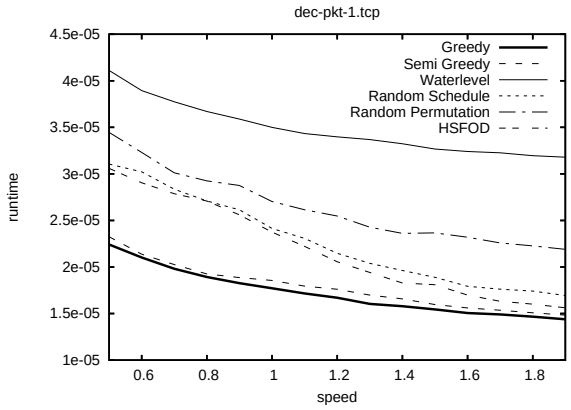

Figure 4: Running time, $m = 30$ and $B = 100$

subsections we report on the competitiveness, running time and memory requirements of the algorithms as parameters m, B and s vary. It turned out that a variation of the speed s gives the most interesting results and we therefore start with a description of this issue.

5.1 Varying the speed s

Figure 3 depicts the experimentally observed competitiveness for varying s. We consider in this presentation a basic setting with $m = 30$ and $B = 100$. These parameters are chosen relative to the size of DEC-PKT-1, which consists of 2.1 million packets. More precisely, we wish to simulate the algorithms for sufficiently large m and time steps with considerable packet traffic. Furthermore, switch simulations in the literature usually also work with $m = 8$ to $m = 32$ ports, see e.g. [26, 30]. Our basic setting of m and B is not critical. As we will see, the observed phenomena occur for other parameter settings (smaller/larger m and smaller/larger B) as well.

An important result of our study is that the experimentally observed competitiveness of all the algorithms ranges between 1.0 and 1.035 and hence is considerably lower than the theoretical bounds. This is not surprising because competitive analysis is a strong worst-case performance measure. It is astonishing, though, that the gap is so high. Remarkably, *Greedy*, *Semi Greedy* and *Waterlevel* have an experimental competitiveness that is always below 1.002, i.e. they are never 0.2% worse than an optimal solution. *HSFOD* exhibits an even better competitiveness of less than 1.001 for all values of s. Furthermore, interestingly, the curves for *Greedy*, *Semi Greedy* and *Waterlevel* are almost identical and indistinguishable in the plot. The three algorithms have essentially the same performance: For instance, the difference in the number of transferred packets is less than 1000 when the total throughput of each of the three strategies is about 2 million packets. All the algorithms have the highest ratios for values of s around 1. On other traces, the peak sometimes occurs at $s \approx 1.1$. Thus, the worst case occurs when the average packet arrival rate is equal to the rate with which the switch can forward packets and packet scheduling decisions matter. For small and large values of s, the experimental competitiveness tends to 1. This is due to the fact that buffers tend to be either heavily populated (small s) or lightly populated (large s) and all the algorithms transfer essentially an optimum number of packets. Another important result is that the theoretical and experimentally observed competitive ratios are unrelated. In particular, in the experiments the randomized strategies, which have low theoretical competitive ratios, do perform considerably worse than the deterministic algorithms.

Figure 4 shows the running times of the algorithms, i.e. the average time in seconds it takes an algorithm to perform 1 time step (update auxiliary data structures to account for incoming packets and determine the queue to be served). We evaluate the running times for varying s because the buffer occu-

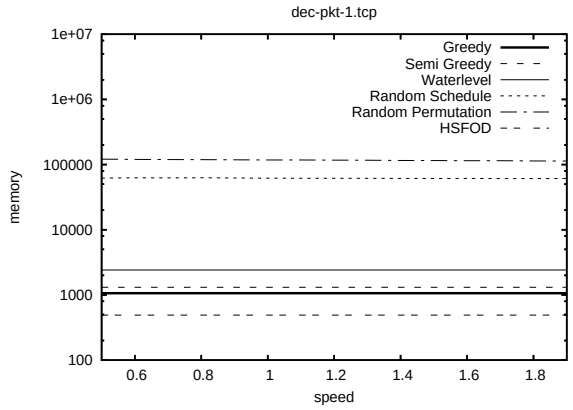

Figure 5: Memory, $m = 30$ and $B = 100$

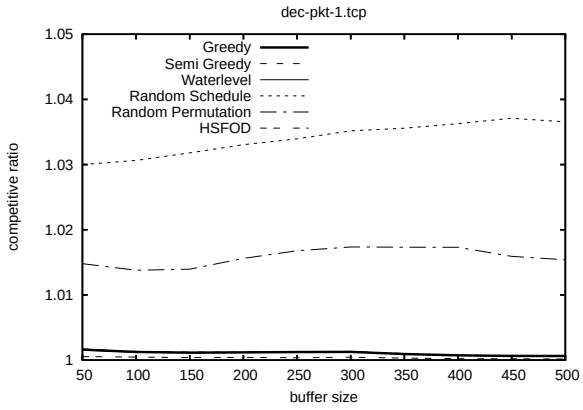

Figure 6: Competitive ratio, $m = 30$ and $s = 1.0$

pancy depends on s and the latter occupancy can affect the running time. Uniformly over all algorithms we observe decreasing running times for increasing values of s. The reason is that, for large s, buffers tend to be empty and the algorithms need less time to handle one time step. *Greedy* and *Semi Greedy* are the fastest algorithms, *Semi-Greedy* being only slightly slower than *Greedy*. *HSFOD*, *Random Schedule*, *Random Permutation* and *Waterlevel* have considerably higher running times. *Waterlevel* is the slowest strategy with running times that are more than twice as high as that of *Greedy*. A shown in Figure 4, the algorithms need 20 to 40 milliseconds to perform one time step. These times would be lower in a switch hardware implementation; our runtime tests just represent a comparative study of the algorithms.

Figure 5 reports on the memory requirements of the algorithms, measured in bytes. Recall that we monitored the maximum total memory required used by auxiliary data structures. The memory requirements are stable for varying s. Nevertheless we depict them in a plot to allow better comparision with the results of the following sections where memory requirements vary as B and m vary. We emphasize here that our plots for the memory requirements are drawn using a *logarithmic scale* as the amount of extra memory needed differs vastly among the strategies. As to be expected, *HSFOD*, *Greedy* and *Semi Greedy* have small requirements. *HSFOD* uses no more than 500 bytes, while *Greedy* and *Semi Greedy*, using priority queues, allocate 1000 to 1300 bytes. *Waterlevel* has space requirements that are twice as high. Huge amounts of extra space (80.000 to 100.000 bytes) are required by *Random Schedule* and *Random Permutation*. Recall that these algorithms need space for auxiliary queues and mB unit-size buffers.

5.2 Varying the buffer size

In a next set of experiments we study the effect of varying the buffer size B. We investigate this effect for the critical speed $s = 1.0$ where the observed competitive ratios are highest. Figure 6 shows that the buffer size has essentially no effect on the competitiveness; only the randomized strategies show a slight fluctuation. This supports our statement that our initial setting $m = 30$ and $B = 100$ plays no particular role. Again, *HSFOD* outperforms all the other algorithms and the performance of *Greedy*, *Semi Greedy* and *Waterlevel* is almost identical. In Figure 7 we observe that the running times, too, are stable. The only exception is *Random Schedule*. The maintenance of its auxiliary queues takes more time as B increases. As for the required space (cf. Figure 8), as was to be expected, the deterministic strategies have fixed demands as the size of the auxiliary data structures depends only on m. *Random Schedule* and *Random Permutation* experience a linear increase as the auxiliary data structures depend on mB. The increase is about 20 bytes per additional buffer cell. Recall that Figure 8 is drawn on a logarithmic scale.

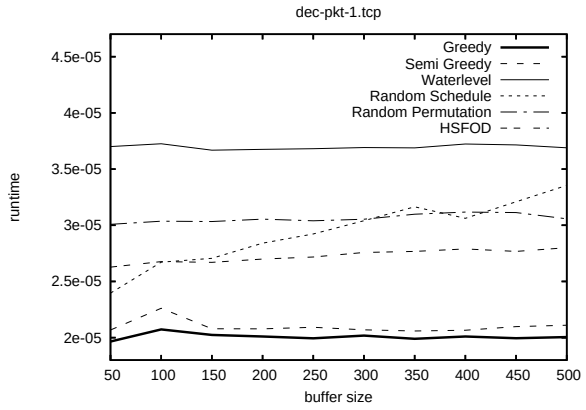

Figure 7: Running time, $m = 30$ and $s = 1.0$

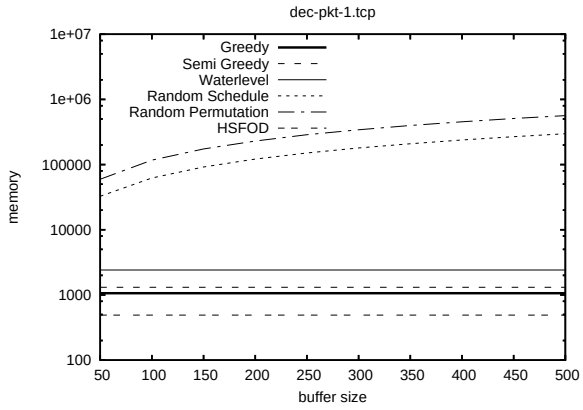

Figure 8: Memory, $m = 30$ and $s = 1.0$

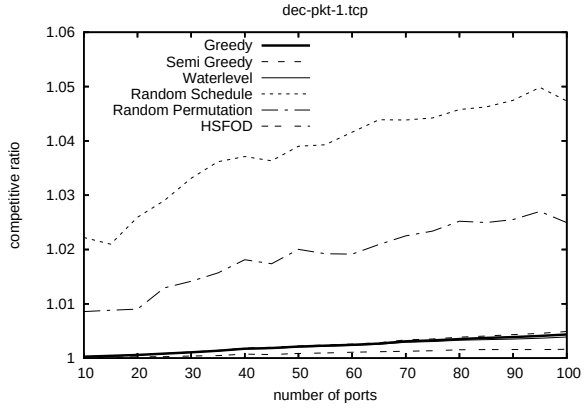

Figure 9: Competitive ratio, $mB = 3000$ and $s = 1.0$

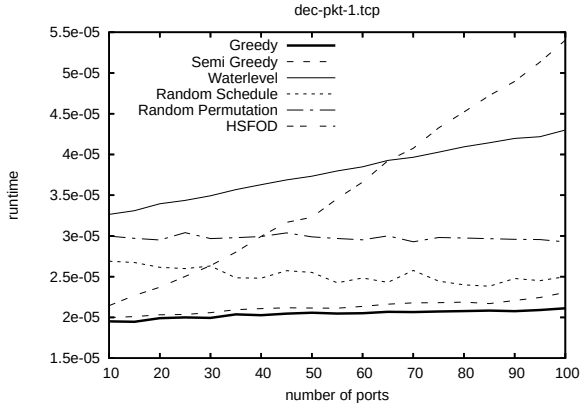

Figure 10: Running time, $mB = 3000$ and $s = 1.0$

5.3 Varying the number of ports

Next we analyze the effect of varying the number m of ports, focusing again on the most critical speed $s = 1.0$. We consider a fixed product of mB, which is equal to $mB = 3000$ for our initial parameters. The reason is the following: Varying m while fixing B would investigate the effect to giving a switch more total buffer space, an issue that was already studied in Section 5.2.

Interestingly, the algorithms perform well in our new scenario. All deterministic algorithms show a very slight increase in experimental competitiveness, see Figure 9. The increase is more pronounced in the case of *Random Schedule* and *Random Permutation*. The general increase in competitiveness is due to the fact that for a larger number of ports, online algorithms have a higher chance of serving the "wrong" port. Figure 10 reveals a weakness of *HSFOD*; its running time increases linearly with m. The same holds for *Waterlevel*, although the gradient is smaller here. For all the other strategies the running times are stable. As for the memory requirements (see Figure 11) as was to be expected, the deterministic algorithms have slightly increasing demands (about 4 bytes per additional port). The demands are fixed for *Random Schedule* and *Waterlevel* as the sizes of the auxiliary data structures are linear in mB.

5.4 The absolute throughput

Finally, we analyze the actual throughput of the algorithms, i.e. the total number of successfully transferred data packets. Our analyses also include the optimal offline algorithm SFOD. Recall that the

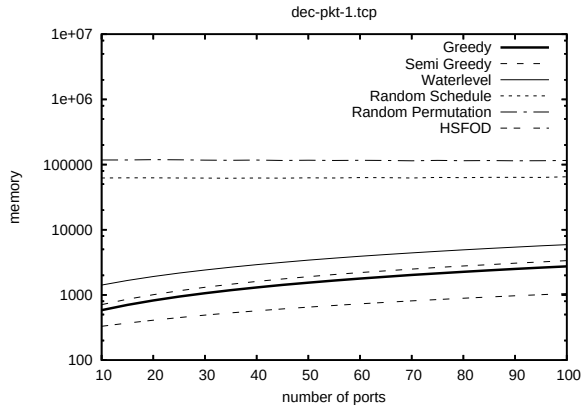

Figure 11: Memory, $mB = 3000$ and $s = 1.0$

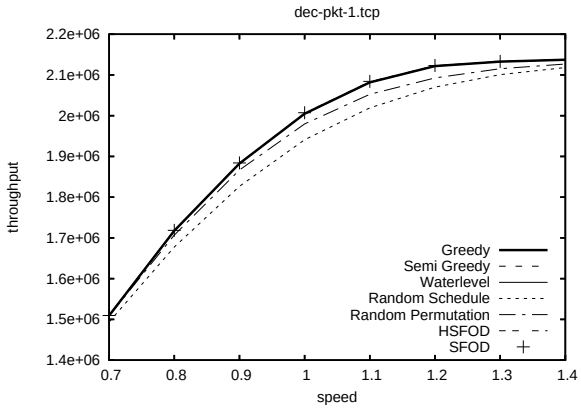

Figure 12: Throughput, $m = 30$ and $B = 100$

DEC-PKT-1 trace consists of 2.1 million data packets, which then represents the maximum throughput possible. Figure 12 depicts the throughput as s varies. We observe an almost linear increase as s increases, leading to the maximum possible throughput at $s = 1.2$. At our critical speed $s = 1$ we vary again B and m, cf. Figures 13 and 14. As B increases, the throughput improves. Interestingly, the gradient is almost the same for all the algorithms. Increasing the number m of ports while fixing the total amount of memory available in the switch, *SFOD*, *HSFOD*, *Greedy*, *Semi Greedy* and *Waterlevel* experience almost no performance loss. On the other hand, *Random Permutation* and *Random Schedule* experience a loss in throughput. The gradient is almost the same for the latter two algorithms.

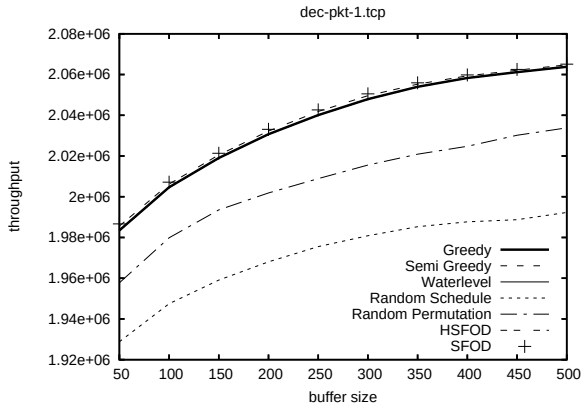

Figure 13: Throughput, $m = 30$ and $s = 1.0$

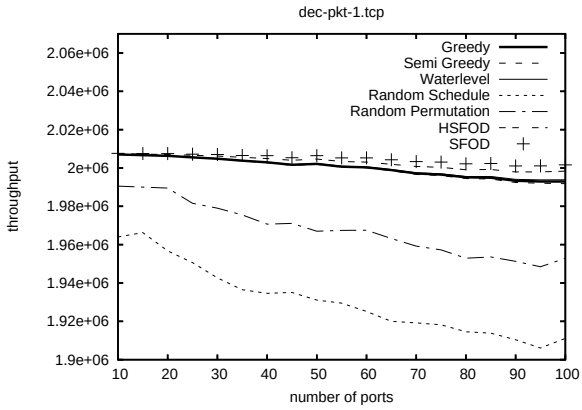

Figure 14: Throughput, $mB = 3.000$ and $s = 1.0$

6 Summary and conclusions

We summarize the most important findings of our experimental study.

- *Greedy:* Excellent experimental competitiveness (below 1.002 in all test); nearly optimal for small and large s as well as for large B. Very low, stable running time. Very low memory requirements.
- *HSFOD:* The best experimental competitiveness, closing more than half of the gap between *Greedy* and the optimal offline algorithm. The running time is however high for large values of m. Very low memory requirements.
- *Semi Greedy:* Excellent experimental competitiveness equal to that of *Greedy*. Low running time that is slightly higher than that of *Greedy*, but the difference is marginal. Very low memory requirements

that are a bit larger than that of *Greedy*.

- *Waterlevel:* Excellent experimental competitiveness equal to that of *Greedy* and *Semi Greedy*. However, the running time is typically more than twice as high; the gap is increasing for large m. Memory requirements, too, are more than twice as high.
- *Random Schedule:* Worst experimental competitiveness among all the algorithms. Running time is significantly higher than that of *Greedy* and *Semi Greedy*. Memory requirements are huge compared to that of the deterministic algorithms.
- *Random Permutation:* High experimental competitiveness. High running time and huge memory requirements.

We conclude that, from a practical point of view, *HSFOD* is the algorithm of choice for switches with a small number of ports. For larger m, *Greedy* is the best algorithm if computation time is limited. The employment of another algorithm is only advisable if a worst case performance must be *guaranteed*. In this case we recommend to apply *Semi Greedy* as it achieves the same experimental performance as *Greedy* and its running time is only marginally higher.

Our tests also show that the experimentally observed competitive ratios of the packet buffering algorithms are considerably smaller than the theoretical bounds. Most of the strategies perform within 3% of an optimal offline solution. The same phenomenon also occurs in other online problems such as paging or scheduling [4, 17, 15, 31]. In our opinion, this gap is no weakness of competitive analysis as competitive analysis is a strong worst-case performance and sequences causing the worst-case ratio usually do not occur in practice. A second general finding of our tests is that the relative performance of the algorithms is unrelated with respect to the theoretical and experimentally observed competitive ratios, i.e. algorithms with a small theoretical competitiveness do not perform better in practice. This is somewhat disappointing. Apparently, the randomized strategies are tailored to specific worst-case input sequences but do not respond well to typical inputs.

References

[1] G. Aggarwal, R. Motwani, D. Shah and An Zhu. Switch scheduling via randomized edge coloring *Proc. 4th Annual IEEE Symp. on Foundations of Computer Science*, 502–511, 2003.

[2] W. Aiello, Y. Mansour, S. Rajagopolan and A. Rosén. Competitive queue policies for differentiated services. *Proc. INFOCOM*, 431–440, 2000.

[3] S. Albers and M. Schmidt. On the performance of greedy algorithms in packet buffering. *Proc. 36th ACM Symp. on Theory of Computing*, 35–44, 2004.

[4] S. Albers, B. Schröder. An experimental study of snline scheduling algorithms. *ACM Journal of Experimental Algorithms*, 7(3), 2002.

[5] N. Andelman, Y. Mansour and A. Zhu. Competitive queueing policies in QoS switches. *Proc. 14th ACM-SIAM Symp. on Discrete Algorithms*, 761–770, 2003.

[6] Y. Azar and A. Litichevskey: Maximizing throughput in multi-queue switches. *Proc. 12th Annual European Symp. on Algorithms (ESA)*, Springer LNCS 3221, 53–64, 2004.

[7] Y. Azar and Y. Richter. Management of multi-queue switches in QoS networks. *Proc. 35th ACM Symp. on Theory of Computing*, 82–89, 2003.

[8] Y. Azar and Y. Richter. An improved algorithm for CIOQ switches. *Proc. 12th Annual European Symp. on Algorithms (ESA)*, Springer LNCS 3221, 65–76, 2004.

[9] A. Aziz, A. Prakash and V. Ramachandran. A new optimal scheduler for switch-memory-switch routers. *Proc. 15th ACM Symp. on Parallelism in Algorithms and Architectures*, 343–352, 2003.

[10] N. Bansal, L. Fleischer, T. Kimbrel, M. Mahdian, B. Schieber and M. Sviridenko. Further improvements in competitive guarantees for QoS buffering. *Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP)*, Springer LNCS 3142, 196–207, 2004.

[11] A. Bar-Noy, A. Freund, S. Landa and J. Naor. Competitive on-line switching policies. *Proc. 13th ACM-SIAM Symp. on Discrete Algorithms*, 525–534, 2002.

[12] M. Chrobak, W. Jawor, J. Sgall and T. Tichy. Improved online algorithms for buffer management in QoS Switches. *Proc. 12th Annual European Symp. on Algorithms (ESA)*, Springer LNCS 3221, 204–215, 2004.

[13] I. Elhanany, D. Chiou, V. Tabatabaee, R. Noro and A. Poursepanj. The network processing forum switch fabric benchmark specifications: An overview. *IEEE Network*, 5–9, 2005.

[14] P. Embrechts and M. Maejima. *Selfsimilar Processes*. Princeton University Press, 2002.

[15] R.L. Graham. Bounds for certain multi-processing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[16] E.L. Hahne, A. Kesselman and Y. Mansour. Competitive buffer management for shared-memory switches. *Proc. 13th ACM Symp. on Parallel Algorithms and Architectures*, 53–58, 2001.

[17] A. Borodin, S. Irani, P. Raghavan and B. Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50:244–258, 1995.

[18] The Internet traffic archive `http://ita.ee.lbl.gov`.

[19] A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber and M. Sviridenko. Buffer overflow management in QoS switches. *Proc. 31st ACM Symp. on Theory of Computing*, 520–529, 2001.

[20] A. Kesselman and Y. Mansour. Loss-bounded analysis for differentiated services. *Proc. 12th ACM-SIAM Symp. on Discrete Algorithms*, 591–600, 2001.

[21] A. Kesselman, Y. Mansour and R. van Stee. Improved competitive guarantees for QoS buffering. *Algorithmica*, 43(1-2):63–80, 2005.

[22] A. Kesselman and A. Rosén. Scheduling policies for CIOQ switches. *Proc. 15th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, 353–361, 2003.

[23] H. Koga. Balanced scheduling towards loss-free packet queueing and delay fairness. *Proc. 12th Annual International Symp. on Algorithms and Computation*, Springer LNCS Vol. 2223, 61–73, 2001.

[24] B. Krishnamurthy and J. Rexford. *Web Protocols and Practice*. Addison-Wesley, 2001.

[25] M. Schmidt. Packet buffering: Randomization beats deterministic algorithms. *Proc. 22nd Annual Symp. on Theoretical Aspects of Computer Science (STACS)*, Springer LNCS 3404, 293–304, 2005.

[26] S. Sukhtankar, D. Hecht and W. Rosen. A novel switch architecture for high-performance computing and signal processing networks. *Proc. 3rd IEEE International Symposium on Network Computing and Applications*, 215–222, 2004.

[27] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. of the ACM*, 28:202-208, 1985.

[28] C. Williamson. Internet traffic measurements. *IEEE Internet Computing*, 5:70–74, 2001.

[29] W. Willinger, M.S. Taqqu and A. Erramilli. A bibiographical guide to self-similar traffic and performance modeling for modern high-speed networks. In F.P. Kelly, S. Zachary and I. Ziedins (eds.), *Stochastic Networks Theory and Applications*, Oxford Science Press, 339–366, 1996.

[30] M. Yang and S.Q. Zheng. An efficient scheduling algorithm for CIOQ switches with space-division multiplexing expansion. *Proc. 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE INFOCOM)*, 2003.

[31] N.E. Young: The k-server dual and loose competitiveness for paging. *Algorithmica*, 11:525–541, 1994.

Appendix

The experimentally observed competitiveness of *HSFOD* for varying α. The results are shown for trace DEC-PKT-1

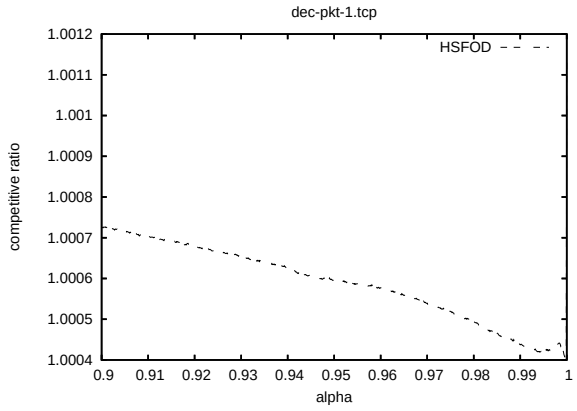

Figure 15: Competitive ratio, $s = 1$, $m = 30$, $B = 100$

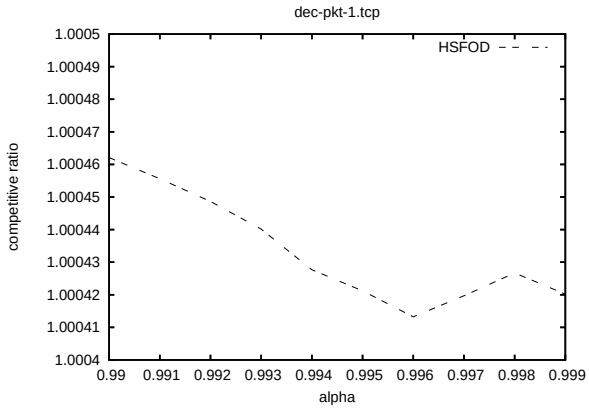

Figure 16: Competitive ratio, $s = 1$, $m = 30$, $B = 100$