Universal Packet Scheduling

Radhika Mittal
UC Berkeley
radhika@eecs.berkeley.edu

Rachit Agarwal
UC Berkeley
ragarwal@berkeley.edu

Sylvia Ratnasamy
UC Berkeley
sylvia@eecs.berkeley.edu

Scott Shenker
UC Berkeley, ICSI
shenker@icsi.berkeley.edu

Abstract

In this paper we address a seemingly simple question: *Is there a universal packet scheduling algorithm?* More precisely, we analyze (both theoretically and empirically) whether there is a single packet scheduling algorithm that, at a network-wide level, can match the results of *any* given scheduling algorithm. We find that in general the answer is "no". However, we show theoretically that the classical Least Slack Time First (LSTF) scheduling algorithm comes closest to being universal and demonstrate empirically that LSTF can closely, though not perfectly, replay a wide range of scheduling algorithms in realistic network settings. We then evaluate whether LSTF can be used *in practice* to meet various network-wide objectives by looking at three popular performance metrics (mean FCT, tail packet delays, and fairness); we find that LSTF performs comparable to the state-of-the-art for each of them.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

General Terms

Algorithms, Design, Experimentation, Performance

1 Introduction

There is a large and active research literature on novel packet scheduling algorithms, from simple schemes such as priority scheduling [27], to complicated mechanisms to achieve fairness [12, 24, 28], to schemes that help reduce tail latency [11] or flow completion time [3], and this short list barely scratches the surface of past and current work. In this paper we do not add to this impressive collection of algorithms, but instead ask if there is a single *universal* packet scheduling algorithm that could obviate the need for new ones.

We can define a universal packet scheduling algorithm (hereafter UPS) in two ways, depending on our viewpoint on the problem. From a theoretical perspective, we call a packet scheduling algorithm *universal* if it can replay any *schedule* (the set of times at which packets arrive to and exit from the network) produced by any other scheduling algorithm. This is not of practical interest, since such schedules are not typically known in advance, but it offers a theoretically rigorous definition of universality that (as we shall see) helps illuminate its fundamental limits (i.e., which scheduling algorithms have the flexibility to serve as a UPS, and why).

From a more practical perspective, we say a packet scheduling algorithm is universal if it can achieve different desired performance objectives (such as fairness, reducing tail latency, minimizing flow completion times). In particular, we require that the UPS should match the performance of the best known scheduling algorithm for a given performance objective.

The notion of universality for packet scheduling might seem esoteric, but we think it helps clarify some basic questions. If there exists no UPS then we should *expect* to design new scheduling algorithms as performance objectives evolve. Moreover, this would make a strong argument for switches being equipped with programmable packet schedulers so that such algorithms could be more easily deployed (as argued in [29]; in fact, it was the eloquent argument in this paper that caused us to initially ask the question about universality).

However, if there is indeed a UPS, then it changes the lens through which we view the design and evaluation of packet scheduling algorithms: e.g., rather than asking whether a new scheduling algorithm meets a performance objective, we should ask whether it is easier/cheaper to implement/configure than the UPS (which could also meet that performance objective). Taken to the extreme, one might even argue that the existence of a (practical) UPS greatly diminishes the need for programmable *scheduling* hardware.[1] Thus, while the rest of the paper occasionally descends into scheduling minutiae,

[1] Note that the case for programmable hardware as made in recent work on P4 and the RMT switch [7, 8] remains: these systems target programmability in header parsing and in how a packet's processing pipeline is defined (i.e., how forwarding 'actions' are applied to a packet). The P4 language does not currently offer primitives for scheduling and, perhaps more importantly, the RMT switch does not implement a programmable packet scheduler; we hope our results can inform the discussion on whether and how P4/RMT might be extended to support programmable scheduling.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

HotNets '15 November 16–17 2015, Philadelphia, PA USA

Copyright 2015 ACM. ISBN 978-1-4503-4047-2/15/11 ...$15.00

DOI: http://dx.doi.org/10.1145/2834050.2834085.

the question we are asking has important practical (and intriguing theoretical) implications.

This paper starts from the theoretical perspective, defining a formal model of packet scheduling and our notion of replayability in §2. While we can prove that there is no UPS, we prove that Least Slack Time First (LSTF) [19] comes as close as any scheduling algorithm to achieving universality, and empirically (via simulation) find that LSTF can closely approximate the schedules of many packet scheduling algorithms. We then take a more practical perspective in §3, finding (via simulation) that LSTF is comparable to the state of the art in achieving various performance objectives. We discuss some related work in §4 and end with a discussion of open questions and future work in §5.

2 Theory: Replaying Schedules

2.1 Definitions and Overview

Network Model: We consider a network of store-and-forward routers connected by links. The input load to the network is a fixed set of packets $\{p \in P\}$, their arrival times $i(p)$ (*i.e.,* when they reach the ingress router), and the path $path(p)$ each packet takes from its ingress to its egress router. We assume no packet drops, so all packets eventually exit. Every router executes a nonpreemptive scheduling algorithm which need not be work-conserving or deterministic and may even involve oracles that know about future packet arrivals. Different routers in the network may use different scheduling logic. For each incoming load $\{(p, i(p), path(p))\}$, a collection of scheduling algorithms $\{A_\alpha\}$ (router α implements algorithm A_α) will produce a set of packet output times $\{o(p)\}$ (the time a packet p exits the network). We call the set $\{(path(p), i(p), o(p))\}$ a *schedule*.

Replaying a Schedule: Applying a different collection of scheduling algorithms $\{A'_\alpha\}$ to the same set of packets $\{(p, i(p), path(p))\}$ produces a new set of output times $\{o'(p)\}$. We say that $\{A'_\alpha\}$ *replays* $\{A_\alpha\}$ on this input if and only if $\forall p \in P$, $o'(p) \leq o(p)$.[2]

Universal Packet Scheduling Algorithm: We say a schedule $\{(path(p), i(p), o(p))\}$ is *viable* if there is at least one collection of scheduling algorithms that produces that schedule. We say that a scheduling algorithm is *universal* if it can replay *all* viable schedules. While we allowed significant generality in defining the scheduling algorithms that a UPS seeks to replay (demanding only that they be nonpreemptive), we insist that the UPS itself obey several practical constraints (although we allow it to be preemptive for theoretical analysis, but then quantitatively analyze the nonpreemptive version in §2.3):[3] We impose three practical constraints on a UPS:

(1) Uniformity and Determinism: A UPS must use the same deterministic scheduling logic at every router.

(2) Limited state used in scheduling decisions: We restrict a UPS to using only (i) packet headers, and (ii) static information about the network topology, link bandwidths, and propagation delays. It cannot rely on oracles or other external information. However, it can modify the header of a packet before forwarding it (resulting in *dynamic packet state* [32]).

(3) Limited state used in header initialization: We assume that the header for a packet p is initialized at its ingress node. The additional information available to the ingress for this initialization is limited to: (i) $o(p)$ from the original schedule and (ii) $path(p)$. Later, we extend the kinds of information the header initialization process can use, and find that this is a key determinant in whether one can find a UPS.

We make three observations about the above model. (i) It assumes greater capability at the edge than in the core, in keeping with common assumptions that the edge is capable of greater processing complexity, exploited by many architectural proposals [9, 25, 31]. (ii) When initializing a packet p's header, the ingress can only use the input time, output time and the path information for p itself, and must be *oblivious* [15] to the corresponding attributes for *other* packets in the network. (iii) The key source of impracticality in our model is the assumption that the output times $o(p)$ are known at the ingress. However, a different interpretation of $o(p)$ suggests a practical application of replayability (and thus our results): *if we assign $o(p)$ as the "desired" output time for every packet p, then the existence of a UPS tells us that if these goals are viable then the UPS will be able to meet them.*

2.2 Theoretical Results

For brevity, in this section we only summarize our key results. Interested readers can find detailed proofs in [21].

Existence of a UPS under omniscient initialization: Suppose we give the header-initialization process extensive information in the form of times $o(p, \alpha)$ which represent when p was forwarded by router α in the original schedule. We can then insert an n-dimensional vector in the header of every packet p, where the ith element contains $o(p, \alpha_i)$, α_i being the ith hop in $path(p)$. Every time a packet arrives at a router, the router can pop the value at the head of the vector in its header and use that as its priority (earlier values of output times get higher priority). This can perfectly replay any viable schedule, which is not surprising, as having such detailed knowledge of the internal scheduling of the network is tantamount to knowing the scheduling algorithm itself. For reasons discussed previously, our definition limited the information available to the output time from the network as a whole, not from each individual router; we call this *black-box* initialization.

[2] We allow the inequality because, if $o'(p) < o(p)$, one can delay the packet upon arrival at the egress node to ensure $o'(p) = o(p)$.

[3] The issue of preemption is somewhat complicated. Allowing the original scheduling algorithms to be preemptive allows packets to be fragmented, which then makes replay extremely difficult even in simple networks (with store-and-forward routers). However, disallowing preemption in the candidate UPS overly limits the flexibility and would again make replay impossible even in simple networks. Thus, we take the seemingly hypocritical but only theoretically tractable approach and disallow preemption in the original scheduling algorithms but allow preemption in the candidate UPS. In practice, when we care only about approximately replaying schedules, the distinction is of less importance, and we simulate LSTF in the nonpreemptive form.

Nonexistence of a UPS under black-box initialization: We can prove by counter-example that *there is no UPS* under the conditions stated in §2.1. The counter-example is available in [21] (and we provide some intuition later in this section). Given this result, we now ask *how close can we get to a UPS?*

Natural candidates for a near-UPS: Simple priority scheduling [4] can reproduce all viable schedules on a single router, so it would seem to be a natural candidate for a near-UPS. However, for multihop networks it may be important to make the scheduling of a packet dependent on what has happened to it earlier in its path. For this, we consider Least Slack Time First (LSTF) [19]. In LSTF, each packet p carries its slack value in the packet header, which is initialized to $slack(p) = (o(p) - i(p) - t_{min}(p, src(p), dest(p)))$ at the ingress [5] . The slack value, therefore, indicates the maximum queueing time that the packet could tolerate without violating the replay condition. Each router, then, schedules the packet which has the least remaining slack at the time when its last bit is transmitted. Before forwarding the packet, the router overwrites the slack value in the packet's header with its remaining slack (*i.e.,* the previous slack time minus how much time it waited in the queue before being transmitted). [6]

Key Results: Our analysis shows that the difficulty of replay is determined by the number of *congestion points*, where a *congestion point* is defined as a node where a packet is forced to "wait" during a given schedule. Our theorems show the following key results:

1. Priority scheduling can replay all viable schedules with no more than one congestion point per packet, and there are viable schedules with no more than two congestion points per packet that it cannot replay.

2. LSTF can replay all viable schedules with no more than two congestion points per packet, and there are viable schedules with no more than three congestion points per packet that it cannot replay.

3. There is no scheduling algorithm (obeying the aforementioned constraints on UPSs) that can replay *all* viable schedules with no more than three congestion points per packet, and the same holds for larger numbers of congestion points.

Main Takeaway: *LSTF is closer to being a UPS than simple priority scheduling, and no other candidate UPS can do better in terms of handling more congestion points.*

Intuition: The reason why LSTF is superior to priority scheduling is clear: by carrying information about previous delays in the packet header (in the form of the *remaining* slack value), LSTF can "make up for lost time" at later congestion points, whereas for priority scheduling packets with low priority might get repeatedly delayed (and thus miss their target output times). LSTF can always handle up to two congestion points per packet because, for this case, each congestion point is either the first or the last point where the packet waits; we can prove that any extra delay seen at the first congestion point during the replay can be naturally compensated for at the second. With three or more congestion points there is no way for LSTF (or any other packet scheduler) to know how to allocate the slack among them; one can create counter-examples where unless the scheduling algorithm makes precisely the right choice in the earlier congestion points, at least one packet will miss its target output time.

[4]Simple priority scheduling is where the ingress assigns priority values to the packets and the routers simply schedule packets based on these static priority values.

[5]where $src(p)$ is the ingress of p; $dest(p)$ is the egress of p; $t_{min}(p, \alpha, \beta)$ is the time p takes to go from router α to router β in an empty network.

[6]There are other ways to implement this algorithm, such as using additional state in the routers and having a static packet header as in Earliest Deadline First (EDF), but here we chose to use an approach with dynamic packet state.

Topology	Link Utilization	Scheduling Algorithm	Fraction of packets overdue Total	$> T$
I2 1Gbps-10Gbps	70%	Random	0.0021	0.0002
I2 1Gbps-10Gbps	10% 30% 50% 90%	Random	0.0007 0.0281 0.0221 0.0008	0.0 0.0017 0.0002 4×10^{-6}
I2 1Gbps-1Gbps I2 10Gbps-10Gbps	70%	Random	0.0204 0.0631	8×10^{-6} 0.0448
RocketFuel Datacenter	70%	Random	0.0246 0.0164	0.0063 0.0154
I2 1Gbps-10Gbps	70%	FIFO FQ SJF LIFO FQ/FIFO+	0.0143 0.0271 0.1833 0.1477 0.0152	0.0006 0.0002 0.0019 0.0067 0.0004

Table 1: LSTF Replayability Results across various scenarios. T represents the transmission time of the bottleneck link.

2.3 Empirical Results

The previous section clarified the theoretical limits on a *perfect* replay. Here we investigate, via ns-2 simulations [2], how well (a nonpreemptable version of) LSTF can *approximately* replay schedules in realistic networks.

Experiment Setup: *Default.* We use a simplified Internet-2 topology [1], identical to the one used in [22] (consisting of 10 routers and 16 links in the core). We connect each core router to 10 edge routers using 1Gbps links and each edge router is attached to an end host via a 10Gbps link.[7] The number of hops per packet is in the range of 4 to 7, excluding the end hosts. We refer to this topology as I2:1Gbps-10Gbps. Each end host generates UDP flows using a Poisson inter-arrival model, and our default scenario runs at 70% utilization. The flow sizes are picked from a heavy-tailed distribution [4, 5]. Since our focus is on packet scheduling, not dropping policies, we use large buffer sizes that ensure no packet drops.

Varying parameters. We tested a wide range of experimental scenarios and present results for a small subset here: (1) the default scenario with network utilization varied from 10-90% (2) the default scenario but with 1Gbps link between the endhosts and the edge routers (I2:1Gbps-1Gbps) and

[7]We use higher than usual access bandwidths for our default setup to increase the stress on the schedulers in the routers. We also present results for smaller access bandwidths, which have better replay performance.

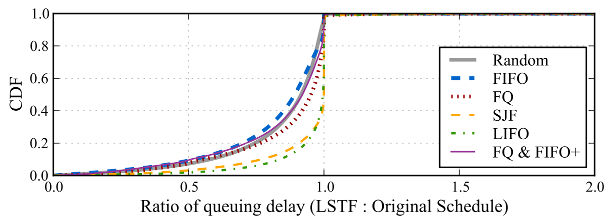

Figure 1: Ratio of queuing delay with varying packet scheduling algorithms, on the default Internet-2 topology at 70% utilization.

with 10Gbps links between the edge routers and the core (I2:10Gbps-10Gbps) and (3) the default scenario applied to two different topologies, a bigger Rocketfuel topology [30] (with 83 routers and 131 links in the core) and a full bisection bandwidth datacenter fat-tree topology from [3] (with 10Gbps links). Note that our other results were generally consistent with those presented here.

Scheduling algorithms. Our default case, which we expected to be hard to replay, uses completely arbitrary schedules produced by a *random* scheduler (which picks the packet to be scheduled randomly from the set of queued up packets). We also present results for more traditional packet scheduling algorithms: FIFO, LIFO, fair queuing [12], SJF (shortest job first using priorities), and a scenario where half of the routers run FIFO+ [11] and the other half run fair queuing.

Evaluation Metrics: We consider two metrics. First, we measure the fraction of packets that are overdue (i.e., which do not meet the original schedule's target). Second, to capture the *extent* to which packets fail to meet their targets, we measure the fraction of packets that are overdue by more than a threshold value T, where T is one transmission time on the bottleneck link ($\approx 12\mu s$ for 1Gbps). We pick this value of T both because it is sufficiently small that we can assume being overdue by this small amount is of negligible practical importance, and also because this is the order of violation we should expect given that our implementation of LSTF is non-preemptive.

Results: Table 1 shows the simulation results for LSTF replay for various scenarios, which we now discuss.

(1) Replayability. Consider the column showing the fraction of packets overdue. In all but three cases (we examine these shortly) over 97% of packets meet their target output times. In addition, the fraction of packets that did not arrive within T of their target output times is much smaller; e.g., even in the worst case of SJF scheduling (where 18.33% of packets failed to arrive by their target output times), only 0.19% of packets are overdue by more than T. Most setups perform substantially better: e.g., in our default setup with Random scheduling, only 0.21% of packets miss their targets and only 0.02% are overdue by more than T. Hence, we conclude that even without preemption LSTF achieves good (but not perfect) replayability under a wide range of scenarios.

(2) Effect of varying network utilization. The second row in Table 1 shows the effect of varying network utilization. We see that at 10% utilization, LSTF achieves exceptionally good replayability with a total of only 0.07% of packets overdue. Replayability deteriorates as utilization is increased to 30% but then (surprisingly) improves again as utilization increases. This improvement occurs because with increasing utilization, the amount of queuing (and thus the average slack across packets) in the original schedule also increases, providing more room for slack re-adjustments when packets wait longer at queues seen early in their paths during the replay. We observed this trend in all our experiments though the exact location of the "low point" varied across settings.

(3) Effect of varying link bandwidths. The third row shows the effect of changing the relative values of access *vs.* core links. We see that while decreasing access link bandwidth (I2:1Gbps-1Gbps) resulted in a much smaller fraction of packets being overdue by more than T (0.0008%), increasing the edge-to-core link bandwidth (I2:10Gbps-10Gbps) resulted in a significantly higher fraction (4.48%). For I2:1Gbps-1Gbps, packets are paced by the endhost link, resulting in few congestion points thus improving LSTF's replayability. In contrast, with I2:10Gbps-10Gbps, both the access and edge links have a higher bandwidth than most core links; hence packets (that are no longer paced at the endhosts or the edges) arrive at the core routers very close to one another and the effect of one packet being overdue *cascades* over to the following packets.

(4) Effect of varying topology. The fourth row in Table 1 shows our results using different topologies. LSTF performs well in both cases: only 2.46% (Rocketfuel) and 1.64% (datacenter) of packets fail replay. These numbers are still somewhat higher than our default case. The reason for this is similar to that for the I2:10Gbps-10Gbps topology – all links in the datacenter topology are set to 10Gbps, while half of the core links in the Rocketfuel topology are set to have bandwidths smaller than the access links.

(5) Varying Scheduling Algorithms. Row five in Table 1 shows LSTF's replay results for different scheduling algorithms. We see that LSTF performs well for FIFO, FQ, and even the combination of FIFO+ and FQ; with fewer than 0.06% of packets being overdue by more than T. SJF and LIFO fare worse with 18.33% and 14.77% of packets failing replay (although only 0.19% and 0.67% of packets are overdue by more than T respectively). The reason stems from two factors: (1) for these algorithms a larger fraction of packets have a very small slack value (as one might expect from the scheduling logic which produces a larger skew in the slack distribution), and (2) for these packets with small slack values, LSTF *without preemption* is often unable to "compensate" for misspent slack that occurred earlier in the path. To verify this intuition, we extended our simulator to support preemption and repeated our experiments: with preemption, the fraction of packets that failed replay dropped to 0.24% (from 18.33%) for SJF and to 0.25% (from 14.77%) for LIFO.

(6) End-to-end (Queuing) Delay. Our results so far evaluated LSTF in terms of measures that we introduced to test universality. We now evaluate LSTF using the more traditional metric of packet delay, focusing on the queueing delay a packet experiences. Figure 1 shows the CDF of the ratios of

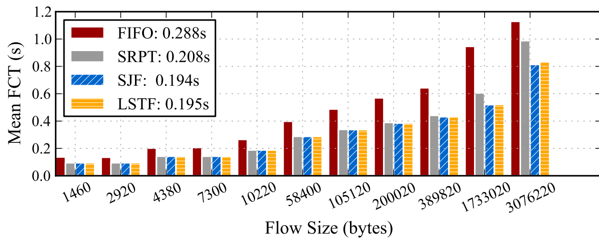

Figure 2: Mean FCT bucketed by flow size for the Internet2 topology with 70% utilization. The legend indicates the mean FCT across all flows.

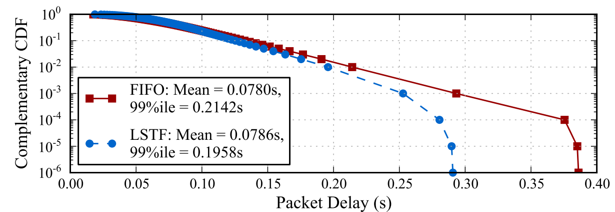

Figure 3: Tail packet delays for LSTF compared to FIFO. The mean and 99%ile packet delay values are indicated in the legend.

the queuing delay that a packet sees with LSTF to the queuing delay that it sees in the original schedule, for varying packet scheduling algorithms. We were surprised to see that most of the packets actually have a smaller queuing delay in the LSTF replay than in the original schedule. This is because LSTF eliminates "wasted waiting", in that it never makes packet A wait behind packet B if packet B is going to have significantly more waiting later in its path.

(7) Comparison with Priorities. To provide a point of comparison, we did a replay using simple priorities for our default scenario, where the priority for a packet p is set to $o(p)$ (which seemed most intuitive to us). As expected, the resulting replay performance is much worse than that with LSTF: 21% packets are overdue in total (vs 0.21% with LSTF), with 20.69% being overdue by more than T (vs 0.02% with LSTF).

Summary: We observe that, in almost all cases, less than 1% of the packets are overdue with LSTF by more than T. The replay performance initially degrades and then starts improving as the network utilization increases. The distribution of link speeds has a bigger influence on the replay results than the scale of the topology. Replay performance is better for scheduling algorithms that produce a smaller skew in the slack distribution. LSTF replay performance is significantly better than simple priorities replay performance, with the most intuitive priority assignment.

3 Practical: Achieving Various Objectives

In this section we look at how LSTF can be used *in practice* to meet three popular network-wide objectives: minimizing mean flow completion time, minimizing tail packet delays, and fairness. Instead of using the knowledge of a given previous schedule (as done in §2.3), we now use certain heuristics (described below) to assign the slacks.

For each objective, we first describe the slack initialization heuristic and then present some ns-2 simulation results on how LSTF performs relative to the state-of-the-art scheduling algorithm on the I2 1Gbps-10Gbps topology running at 70% average utilization.[8] The switches have finite buffers (packets with the highest slack are dropped when the buffer is full).

3.1 Mean Flow Completion Time

While there have been several proposals on how to minimize flow completion time (FCT) via the transport protocol [13,22], here we focus on scheduling's impact on FCT. In [3] it is shown that (i) Shortest Remaining Processing Time (SRPT) is close to optimal for minimizing the mean FCT and (ii) Shortest Job First (SJF) produces results similar to SRPT for realistic heavy-tailed distribution. Thus, these are the two algorithms we use as benchmarks.

Slack Initialization: The slack for a packet p is initialized as $slack(p) = fs(p) * D$, where $fs(p)$ is the size of the flow to which the packet p belongs and D is a value much larger than the delay seen by any packet in the network ($D = 1$ sec in our simulations).

Evaluation: We use TCP flows with a 5MB buffer in each router (which is equal to the average delay-bandwidth product for the Internet2 topology we are using). Figure 2 compares LSTF with FIFO, SJF and SRPT with *starvation prevention* as in [3] [9]. SJF has a slightly better performance than SRPT, both resulting in a significantly lower mean FCT than FIFO. LSTF's performance is nearly the same as SJF.

3.2 Tail Packet Delays

Clark et. al. [11] proposed the FIFO+ algorithm for minimizing the tail packet delays in multi-hop networks, where packets are prioritized at a router based on the amount of queuing delay they have seen at their previous hops.

Slack Initialization: All incoming packets are initialized with the same slack value (1 sec in our simulations). This makes LSTF identical to FIFO+.

Evaluation: We compare our LSTF policy (which, with the above slack initialization, is identical to FIFO+) with FIFO. We present our results using UDP flows, which ensures that the input load remains the same in both cases, allowing a fair comparison for the in-network packet-level behavior across the two scheduling policies. Figure 3 shows our results. With LSTF, packets that have traversed through more number of hops, and have therefore spent more slack in the network, get preference over shorter-RTT packets that have traversed through fewer hops. While this might produce a slight increase in the mean packet delay, it reduces the tail. This is in-line with the observations made in [11].

3.3 Fairness

Fairness is the most challenging objective to achieve with LSTF, but we show that it can achieve *asymptotic* fairness (i.e. eventual convergence to the fair-share rate).

[8] We have run our simulations in a wide variety of scenarios and find similar results to what we present here.

[9] The router always schedules the earliest arriving packet of the flow which contains the highest priority packet.

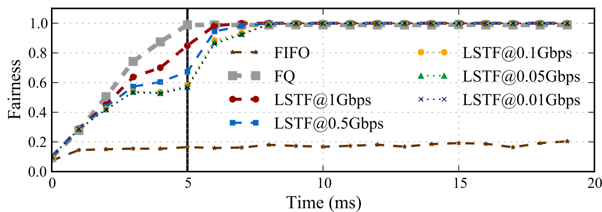

Figure 4: Fairness for long-lived flows on Internet2 topology. The legend indicates the value of r_{est} used for LSTF slack initialization.

Slack Initialization: Our approach is inspired from [33]. We assign $slack = 0$ to the first packet of the flow and the slack of any subsequent packet p_i is then initialized as:

$$slack(p_i) = max\Big(0, slack(p_{i-1}) + \frac{1}{r_{est}} - \big(i(p_i) - i(p_{i-1})\big)\Big)$$

where $i(p)$ is the arrival time of the packet p at the ingress and r_{est} is an estimate of the fair-share rate r^*. We show that the above heuristic leads to asymptotic fairness, for *any* value of r_{est} that is less than r^*, as long as all flows use the same value. A reasonable value of r_{est} can be estimated using some knowledge about the network topology and traffic matrices, though we leave a detailed exploration of this to future work. We can also extend the slack assignment heuristic to achieve weighted fairness by using different values of r_{est} for different flows, in proportion to the desired weights.

Evaluation: We evaluate the asymptotic fairness property by running our simulation on the Internet2 topology with 10Gbps edges (I2 10Gbps-10Gbps), such that all the congestion is happening at the core. However, we reduce the propagation delay, to make the experiment more scalable, while the buffer size is kept large so that the fairness is dominated by the scheduling policy. We start 90 long-lived TCP flows with a random jitter in the start times ranging from 0-5ms. The topology is such that the fair share rate of each flow on each link in the core network (which is shared by up to 13 flows) is around 1Gbps. We use different values for $r_{est} \leq$ 1Gbps for computing the initial slacks and compare our results with fair queuing (FQ). Figure 4 shows the fairness computed using Jain's Fairness Index [17], from the throughput each flow receives per millisecond. Since we use the throughput received by each of the 90 flows to compute the fairness index, it reaches 1 with FQ only at 5ms, after all the flows have started. We see that LSTF is able to converge to perfect fairness, even when r_{est} is 100X smaller than r^*, though it converges slightly sooner when r_{est} is closer to r^*.

4 Related Work

The real-time scheduling literature has studied optimality of scheduling algorithms[10] (in particular EDF and LSTF) for single and multiple processors [19, 20]. Liu and Layland [20] proved the optimality of EDF for a single processor in hard real-time systems. LSTF was then shown to be optimal for single-processor scheduling as well, while being more effective than EDF (though not optimal) for multi-processor scheduling [19]. In the context of networking, [10] provides theoretical results on emulating the schedules produced by a single output-queued switch using a combined input-output queued switch with a smaller speed-up of at most two. To the best of our knowledge, the optimality or universality of a scheduling algorithm for a network of inter-connected resources (in our case, switches) has never been studied before.

A recent paper [29] proposed programmable hardware in the dataplane for packet scheduling and queue management, in order to achieve various network objectives without the need for physically replacing the hardware. It uses simulation of three schemes (FQ, CoDel+FQ, CoDel+FIFO) competing on three different metrics to show that there is no "silver bullet" solution. As mentioned earlier, our work is inspired by the questions the authors raise; we adopt a broader view of scheduling in which packets can carry dynamic state leading to the results presented here.

5 Open Questions and Future Work

Theoretical Analysis: Our work leaves several theoretical questions unanswered, including the following. First, we showed existence of a UPS with omniscient header initialization, and nonexistence with black-box initialization. *What is the least information we can use in header initialization in order to achieve universality?* Second, we showed that the fraction of overdue packets is small and most are only overdue by a small amount during an LSTF replay. *Are there tractable bounds on both the number of overdue packets and/or their degree of lateness?* Finally, while we have a formal analysis of LSTF's ability to replay a given schedule, we do not yet have any formal model for the scope of LSTF in meeting various objectives in practice. *Can one describe the class of performance objectives that LSTF can meet?*

Real Implementation: We need to show the feasibility of implementing LSTF in hardware. However, we can prove that LSTF execution *at a particular router* is no more complex than the execution of fine-grained priorities, which can be carried out in almost constant time using specialized data-structures such as pipelined heap [6, 16].

Incorporating Feedback: Typically congestion control involves endhosts reacting to network feedback, which can be implicit (e.g., packet drops by Active Queue Management schemes [14, 23]) or explicit (e.g., ECN markings [26] or rate allocations schemes such as RCP [13] and XCP [18]). It is unclear whether it is necessary to incorporate such feedback mechanisms in our notion of universality, and if so how.

6 Acknowledgments

We are thankful to Prof. Satish Rao for his helpful tips regarding the theoretical aspects of this work. We would also like to thank Prof. Ion Stoica, Kay Ousterhout, Justine Sherry, Aurojit Panda and our anonymous HotNets reviewers for their thoughtful feedback. This material is based upon work supported by Intel Research and by the National Science Foundation under Grant No. 1117161, 1343947 and 1040838.

[10] where a scheduling algorithm is said to be optimal if it can (feasibly) schedule a set of tasks that can be scheduled by any other scheduling algorithm.

7 References

[1] Internet2. http://www.internet2.edu/.

[2] The Network Simulator NS-2. http://www.isi.edu/nsnam/ns/.

[3] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *Proc. ACM SIGCOMM*, 2013.

[4] M. Allman. Comments on bufferbloat. *ACM SIGCOMM Computer Communication Review*, 2013.

[5] T. Benson, A. Akella, and D. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. ACM IMC*, 2012.

[6] R. Bhagwan and B. Lin. Fast and Scalable Priority Queue Architecture for High-Speed Network Switches. In *Proc. IEEE Infocom*, 2000.

[7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 2014.

[8] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proc. ACM SIGCOMM*, 2013.

[9] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *Proc. ACM HotSDN*, 2012.

[10] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar. Matching output queueing with a combined input/output-queued switch. *IEEE Journal on Selected Areas in Communications*, 1999.

[11] D. D. Clark, S. Shenker, and L. Zhang. Supporting Real-time Applications in an Integrated Services Packet Network: Architecture and Mechanism. *ACM SIGCOMM Computer Communication Review*, 1992.

[12] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *ACM SIGCOMM Computer Communication Review*, 1989.

[13] N. Dukkipati and N. McKeown. Why Flow-Completion Time is the Right Metric for Congestion Control. *ACM SIGCOMM Computer Communication Review*, 2006.

[14] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. Netw.*, 1993.

[15] A. Gupta, M. T. Hajiaghayi, and H. Räcke. Oblivious Network Design. In *Proc. ACM-SIAM Symposium on Discrete Algorithm (SODA)*, 2006.

[16] A. Ioannou and M. G. H. Katevenis. Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-speed Networks. *IEEE/ACM Trans. Netw.*, 2007.

[17] R. Jain, D.-M. Chiu, and W. Hawe. A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems. *CoRR*, arXiv:cs/9809099, 1998.

[18] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *Proc. ACM SIGCOMM*, 2002.

[19] J. Y.-T. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica, Springer-Verlag New York Inc.*, 1989.

[20] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM (JACM)*, 1973.

[21] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker. Universal Packet Scheduling. *CoRR*, arXiv:1510.03551, 2015.

[22] R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Recursively Cautious Congestion Control. In *Proc. USENIX NSDI*, 2014.

[23] K. Nichols and V. Jacobson. Controlling Queue Delay. *ACM Queue*, 2012.

[24] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-node Case. *IEEE/ACM Trans. Netw.*, 1993.

[25] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker. Software-defined Internet Architecture: Decoupling Architecture from Infrastructure. In *Proc. ACM HotNets*, 2012.

[26] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, 2001.

[27] S. Blake and D. Black and M. Carlson and E. Davies and Z. Wang and W. Weiss. An Architecture for Differentiated Services. RFC 2475, 1998.

[28] M. Shreedhar and G. Varghese. Efficient Fair Queueing Using Deficit Round Robin. *ACM SIGCOMM Computer Communication Review*, 1995.

[29] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan. No Silver Bullet: Extending SDN to the Data Plane. In *Proc. ACM HotNets*, 2013.

[30] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. ACM SIGCOMM*, 2002.

[31] I. Stoica, S. Shenker, and H. Zhang. Core-stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks. In *Proc. ACM SIGCOMM*, 1998.

[32] I. Stoica and H. Zhang. Providing Guaranteed Services Without Per Flow Management. In *Proc. ACM SIGCOMM*, 1999.

[33] L. Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. *ACM SIGCOMM Computer Communication Review*, 1990.