HLS: A Packet Scheduler for Hierarchical Fairness

Natchanon Luangsomboon
Department of ECE
University of Toronto
nat.luangsomboon@mail.utoronto.ca

Jörg Liebeherr
Department of ECE
University of Toronto
jorg@ece.utoronto.ca

***Abstract*—Hierarchical link sharing addresses the demand for fine-grain traffic control at multiple levels of aggregation. At present, packet schedulers that can support hierarchical link sharing are not suitable for an implementation at line rates, whereas deployed schedulers perform poorly at distributing excess capacity to classes that need additional bandwidth. We present *HLS*, a packet scheduler that ensures a hierarchical max-min fair allocation of the link bandwidth. HLS supports minimum rate guarantees and isolation between classes. Since it is realized as a non-hierarchical round-robin scheduler, it is suitable to operate at high rates. We implement HLS in the Linux kernel and evaluate it with respect to achieved rate allocations and overhead. We compare the results with those obtained for CBQ and HTB, the existing scheduling algorithms in Linux for hierarchical link sharing. We show that the overhead of HLS is comparable to that of other classful packet schedulers.**

***Index Terms*—Packet scheduling, link sharing, fairness, Qdisc.**

I. Introduction

Packet scheduling plays a crucial role in the management of traffic flows, for prioritizing traffic, for flexible service differentiation, and for achieving performance metrics, such as flow completion times, throughput, and the tail of the delay distribution. This paper is concerned with packet scheduling methods that support traffic control at multiple aggregation levels. The need for such scheduling methods is largely driven by content providers that manage traffic within and between servers, clusters, and data centers. Increasingly, data centers rely on fine-grain traffic control at multiple levels of aggregation. The Google B4 inter data center network reports no less than five levels of traffic aggregation [1], [2]. Traffic control in support of a hierarchical distribution of available bandwidth is referred to as *hierarchical link sharing*.

As an example of link sharing, consider the hierarchy shown in Fig. 1. The top of the hierarchy, labeled as *root*, is a link with a fixed rate of 1000 (units are in Mbps). This bandwidth is to be divided between three traffic classes A, B, and C that each receive a minimum rate guarantee, as indicated in the figure. Traffic class A is further divided into classes $A1$ and $A2$, with guarantees of 100 and 200, respectively. Class B splits the bandwidth between $B1$ and $B2$ in the same fashion. Arriving packets are classified and mapped to *leaf classes*, which are the classes at the bottom level of the hierarchy.

This research is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

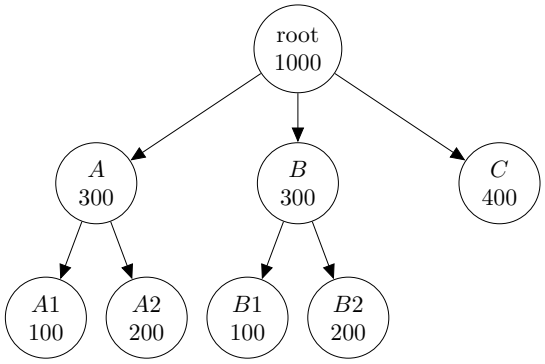


Fig. 1. Link sharing hierarchy.

Clearly, if the aggregate traffic from all leaf classes does not exceed the link capacity, every leaf class can obtain a rate equal to its arrival rate. Likewise, if the arrival rate of every leaf class exceeds its guaranteed rate, then each leaf class is limited to its guaranteed rate. The bandwidth allocation becomes less trivial when the aggregate arrival rate from all classes is larger than the link capacity, and some classes exceed their guaranteed rates, while others stay well below their guarantees. In this case, excess capacity left unused by some classes must be distributed equitably to classes that desire additional bandwidth.

Several packet scheduling algorithms that support class hierarchies with rate guarantees as shown in Fig. 1 are available, however, deployed or deployable algorithms show significant shortfalls while algorithms without such shortfalls are too complex to be deployable. This paper addresses this dichotomy by presenting a packet scheduler with provable link sharing properties and low computational complexity.

For a non-hierarchical setting, a bit-by-bit round-robin algorithm provides link sharing that satisfies a weighted version of max-min fairness [3]. However, bit-by-bit round robin assumes fluid flow traffic and is not implementable as a packet scheduler. Weighted-Fair-Queueing (WFQ) [4] has shown to have a strictly bounded deviation from the ideal bit-by-bit round robin [5]. The drawback of WFQ, which extends to some of its approximations [6], [7], is that it requires to maintain a priority queue that transmits packets in the order of assigned timestamps. This involves an $O(\log N)$ computation for each packet, where N is the number of backlogged packets. Deficit-Round-Robin (DRR) [8] is a packet-level round-robin scheduler for variable-sized packets, whose link sharing ability is inferior to WFQ, but with a simpler implementation with a

constant overhead for each packet. Due to the low complexity, Linux [9] and line-rate switches [10] generally realize link sharing with a round-robin scheduler, such as DRR.

For class hierarchies as in Fig. 1, Hierarchical Packet Fair Queueing (HPFQ) [11] achieves link sharing by employing a cascade of hierarchically organized WFQ schedulers. Packets at the head of the queue of backlogged leaf classes engage in a virtual tournament, with one round of the tournament for each level of the class hierarchy. The tournament starts at the bottom of the hierarchy. In each round, the packet with the smallest timestamp at one level proceeds to the next level. The winner of the tournament is selected for transmission. While HPFQ achieves almost ideal link sharing, it involves a considerable overhead and has not been considered for deployment.[1]

Attempts to extend DRR to a class hierarchy have so far not resulted in practical scheduling algorithms. In [13], the class hierarchy is mapped to a flat hierarchy by interleaving classes according to their guarantees. This results in good fairness properties, but rounds grow prohibitively large which may result in excessive delays between packet transmissions for some classes. Other efforts in this direction, e.g., [14], [15] make scheduling decisions in multiple stages, one per level in the class hierarchy, and thus inherit the drawbacks of HPFQ.

Class-based queuing (CBQ) [16] and Hierarchical Token Bucket (HTB) [17] are two packet schedulers for link sharing in class hierarchies that are actually deployed, even if the deployment is limited to Linux systems. CBQ provides minimum bandwidth guarantees to traffic classes and distributes excess capacity to backlogged classes. CBQ measures the transmission rate of each class to identify traffic classes that are allowed to transmit, which are then served by a variant of DRR. HTB tries to improve the efficiency of CBQ by metering the transmission rates of classes with token bucket filters. Classes that exceed their rate guarantee can 'borrow' bandwidth from classes further up in the class hierarchy. HTB schedules packets with a set of DRR schedulers, where only one DRR scheduler is active at a time. In addition to link sharing, HTB also enforces rate limits. HTB has become the primary tool for scheduling and shaping of hierarchically structured traffic flows in Linux servers [18]–[20].

CBQ and HTB implement rules that dictate when a class with need for additional bandwidth can transmit, however, with the rules it is not possible to determine (a priori) the allocated rates for a given traffic load. In contrast, the outcomes of schedulers such as HPFQ and hierarchical extensions of DRR schedulers satisfy a hierarchical version of max-min fairness, which ensures class guarantees as well as isolation between classes in the hierarchy.

Realizing hierarchical link sharing with round-robin schedulers is attractive, since it does not involve packet timestamps and priority queues, but has shown to be challenging. Extensions of DRR to class hierarchies has so far not resulted in a practical scheduling algorithm. On the other hand, CBQ and HTB systematically fail to isolate rate guarantees between classes in different branches of the class hierarchy. In particular, they allow classes to manipulate the rate allocation by reassigning rate guarantees in a subtree of the class hierarchy (see Subsec. V-C). Until now, there does not exist a round-robin packet scheduler for class hierarchies that can satisfy rate guarantees while isolating the allocations in different parts of the class hierarchy.

In this paper, we present *Hierarchical Link Sharing* (HLS), the first round-robin scheduler for hierarchical link sharing that ensures rate guarantees and isolation between classes, and that can run at high line rates. The rate allocation achieved by HLS satisfies a hierarchical version of max-min fairness. This allocation is strategy-proof, as defined in [21], in the sense that classes cannot improve their allocation through wrongful representation of their demand or the demand of their subclasses. HLS is a non-hierarchical variant of DRR with a time-variable quantum for each class. We have implemented HLS as a Linux kernel module [22]. We present experiments showing that HLS ensures rate guarantees for and isolation between classes, with an overhead that is comparable to other classful schedulers in the Linux kernel.

II. Hierarchical Max-Min Fairness

In a non-hierarchical setting, link sharing between classes can be achieved by fair queueing algorithms that approximate bit-by-bit round robin, resulting in a max-min fair rate allocation. Hierarchical max-min fairness results when (weighted) bit-by-bit round-robin approximations are applied to each group of siblings in the class hierarchy. Expressions that quantify the solution of this allocation exist for a non-hierarchical setting, but are not available for class hierarchies. In the following we quantify both the non-hierarchical and hierarchical notions of max-min fairness.

A. Terminology

We first introduce terminology needed to describe the relationships between classes in a class hierarchy. Fig. 2 depicts a class hierarchy as a rooted tree, where each node represents a class. The class at the top of the hierarchy, referred to as *root*, represents a network interface where the scheduling algorithm is active. Leaf nodes in the rooted tree represent *leaf classes*, which are shown as gray circles. All traffic arrivals are mapped to leaf classes. Nodes that are neither the root nor a leaf node represent *internal classes*. If $\mathcal{N}$ is the set of all classes, we denote by $\mathcal{L}$ and $\mathcal{I}$, respectively, the leaf classes and internal classes, with $\mathcal{N} = \mathcal{L} \cup \mathcal{I} \cup \{\mathrm{root}\}$.

For class i in the figure, the incoming edge connects to its parent class $p(i)$, and the outgoing edges connect to its child classes $\mathrm{child}(i)$. Other needed terms such as ancestors ($\mathrm{anc}(i)$), siblings ($\mathrm{sib}(i)$), descendants ($\mathrm{desc}(i)$), and leaf descendants ($\mathrm{ldesc}(i)$) are indicated by dashed boxes in Fig. 2.

In a class hierarchy, each class is associated with a weight or with a rate guarantee. A rate guarantee g_i of class $i \in \mathcal{I} \cup \{\mathrm{root}\}$ must satisfy the superadditive property $g_i \geq \sum_{j \in \mathrm{child}(i)} g_j$, with $g_{\mathrm{root}} = C$. An alternative specification

[1]The claim in [12] of realizing HPFQ by a hierarchy of PIFO queues is incorrect, as counterexamples are easily constructed when packet sizes are variable and class guarantees are not uniform.

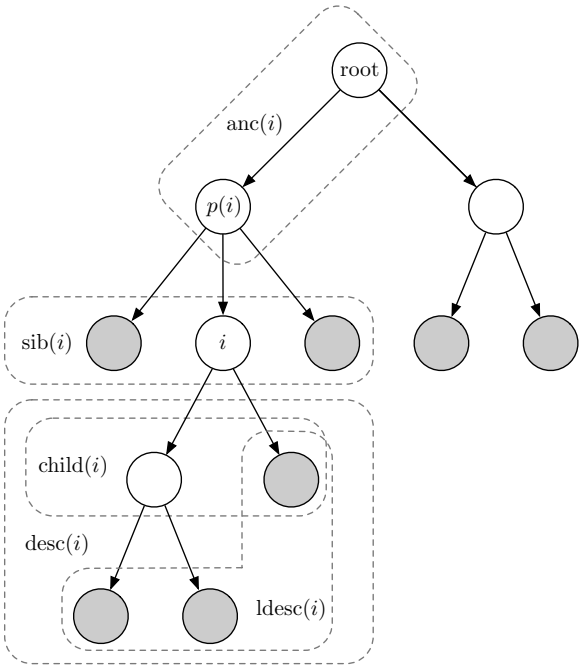


Fig. 2. Subsets in a link sharing hierarchy relative to class i.

of link sharing is based on *weights*, where $w_i > 0$ denotes the weight of class i. If three sibling classes 1, 2, and 3 have weights w_1, w_2, w_3, and all siblings are backlogged, the weights indicate that they will split the capacity made available to them as a group in the ratio $w_1 : w_2 : w_3$. Working with weights is often more convenient, since it allows to express link sharing as dividing available bandwidth locally between siblings. In contrast, rate guarantees appear as global quantities with constraints across all classes. We note that both concepts are equivalent, since guarantees can be viewed as weights, that is $w_i = g_i$ for each class i. Likewise, given the link capacity C and weights w_i, a rate guarantee of g_i can be computed from

$$g_i = C \cdot \prod_{\substack{j \in \mathrm{anc}(i) \cup \{i\} \\ j \neq \mathrm{root}}} \frac{w_j}{\sum_{k \in \mathrm{sib}(j)} w_k} .$$

In the following we will work with weights. By ensuring that $w_i \geq \sum_{j \in \mathrm{child}(i)} w_j$ for each class i, we can use 'weight' and 'rate guarantee' interchangeably.

B. Max-min fair allocation

We formulate rate allocations for traffic classes with fixed-rate traffic at a link with fixed capacity C. We define

r_i	Rate request of class i,
a_i	Rate allocation to class i ($a_i \leq r_i$),
w_i	Weight associated with class i.

Rate requests are not made explicitly, but are determined by traffic arrivals from a class at the link and the resulting backlog. In a max-min fair allocation without weights, if a class is allocated less than it requests, it receives at least as much as any other class. As a consequence, two classes that do not satisfy their demand have the same allocation. Also, if the total demand exceeds the capacity then the entire link capacity is allocated. When specifying a weight w_i for each class i, the weighted max-min fair allocation is defined by the following rules:

(R1) If $a_i < r_i$, then $\dfrac{a_i}{w_i} \geq \dfrac{a_j}{w_j}$ for each class $j \in \mathcal{N}$.

(R2) $\sum_{j \in \mathcal{N}} a_j = \min\big(\sum_{j \in \mathcal{N}} r_j, C\big)$.

Rule *(R1)* states that, if a class is not allocated its entire request, then its allocation in proportion to its weight is at least as large as the (also proportional) allocation of any other class. The second rule simply ensures that either all requests are satisfied or all resources are allocated.

A weighted max-min fair allocation creates a set $\mathcal{S}$ of *satisfied classes*, which receive their entire request ($a_i = r_i$). and a set $\mathcal{N} \setminus \mathcal{S}$ of *unsatisfied classes* with $a_i < r_i$. Rule *(R1)* implies that $\frac{a_i}{w_i} = \frac{a_j}{w_j}$ for any two unsatisfied classes. The allocation is strategy-proof since an unsatisfied class cannot increase its allocation by increasing or misrepresenting its request.

If there is at least one unsatisfied class i, we define the *fair share* f as

$$f = \frac{a_i}{w_i} ,$$

which results in the allocation $a_j = \min\{r_j , w_j f\}$.

Supposing that there exist unsatisfied classes, rule *(R2)* yields an expression for the fair share given by

$$f = \frac{C - \sum_{j \in \mathcal{S}} r_j}{\sum_{j \notin \mathcal{S}} w_j} . \tag{1}$$

The fair share is uniquely defined, as long as $\mathcal{S} \neq \emptyset$. Even though the expression for f is implicit, that is, f is defined in terms of $\mathcal{S}$, and $\mathcal{S}$ is defined in terms of f, the fair share can be algorithmically computed by a water filling algorithm.

C. Hierarchical max-min fair allocation

Next consider a class hierarchy as given in Fig. 2. The requests and allocations of internal classes and the root consist of the total requests and allocations, respectively, of their child classes. That is, for each $i \in \mathcal{I} \cup \{\mathrm{root}\}$,

$$r_i = \sum_{j \in \mathrm{child}(i)} r_j , \quad a_i = \sum_{j \in \mathrm{child}(i)} a_j . \tag{2}$$

With this notation, we can specify a max-min fair allocation for class hierarchies.

A *hierarchical weighted max-min fair* (*HMM fair*) allocation is defined by these two rules that hold for each $i \in \mathcal{L} \cup \mathcal{I}$.

(R1) If $a_i < r_i$, then $\dfrac{a_i}{w_i} \geq \dfrac{a_j}{w_j}$ for all $j \in \mathrm{sib}(i)$.

(R2) $\sum_{j \in \mathcal{L}} a_j = \min\big(\sum_{j \in \mathcal{L}} r_j, C\big)$.

The rules are analogous to those for max-min fairness without a hierarchy. In essence, each parent allocates the capacity available to it to its child classes using the max-min fairness principle. According to *(R1)*, if a class cannot satisfy its request, then its allocation relative to its weight is at least as large as the allocation of any of its siblings relative to the

weight of that sibling. The second rule makes sure that all available capacity is utilized. The allocation is strategy-proof for each group of siblings since it satisfies max-min fairness from Sec. II-B, and, therefore, is strategy-proof for the entire hierarchy. No class can obtain a larger allocation by increasing or misrepresenting its request.

If the rules for hierarchical max-min fairness are straightforward, the computation of the actual rate allocations is much less so. The reason is that the capacity available at an internal class depends on the requests of leaf classes in all branches of the hierarchy. In fact, to the best of our knowledge, an algorithm to compute an HMM rate allocation does not exist in the literature. The keys to such an algorithm are the following two observations, that can be obtained from (2) and rule *(R2)*:

(O1) If $a_i < r_i$, then $a_j < r_j$ for all $j \in \mathrm{anc}(i)$,
(O2) If $a_i = r_i$, then $a_j = r_j$ for all $j \in \mathrm{desc}(i)$.

To make observation *(O1)*, note that a class that receives a smaller rate than it requests will try to get more capacity from its parent, which, in turn, will seek to acquire capacity from its own parent, and so forth. Hence, if the request of a class is not satisfied, the resources of all its ancestors will be exhausted, leaving them unsatisfied as well. Observation *(O2)* follows since requests and allocations of an internal class consist of the aggregated requests and allocations of their child classes.

With this, we can present the results of an HMM allocation as specified by the following theorem. We provide a proof in [23].

Theorem 1. *Given a link with capacity C, and a class hierarchy where each class i has a request rate r_i and a weight $w_i > 0$. Define $a_{\mathrm{root}} = \min\{r_{\mathrm{root}}, C\}$.*
Then, the HMM fair allocation for each class $i \in \mathcal{I} \cup \mathcal{L}$ is

$$a_i = \begin{cases} \min\left(r_i, w_i f_{p(i)}\right), & \text{if } r_{p(i)} > a_{p(i)}, \\ r_i, & \text{otherwise}, \end{cases}$$

where the fair share f_k for each $k \in \mathcal{I} \cup \{\mathrm{root}\}$ is

$$f_k = \frac{a_k - \sum_{j \in \mathcal{S}_k} r_j}{\sum_{j \notin \mathcal{S}_k} w_j}, \tag{3}$$

and where $\mathcal{S}_k$ is defined as

$$\mathcal{S}_k = \{j \in \mathrm{child}(k) \mid r_j < w_j f_k\}.$$

The theorem lends itself to the development of an algorithm for computing the fair shares in (3), which is given in Algorithm 1. The algorithm starts at the top of the hierarchy and computes the fair share of the root. Then it proceeds to compute the fair share of children of the root and continues to traverse the class hierarchy in a top-down fashion (in no particular order) until a leaf class is reached. The function MaxMinFair computes the fair share from (1) for a set $\mathcal{N}$ of classes with requests and weights $r_i, w_i \geq 0$ for each class $i \in \mathcal{N}$, and a link capacity C. (MaxMinFair returns a fair share of infinity if $\sum_{j \in \mathcal{N}} r_i \leq C$, which yields the correct allocation $a_i = r_i$ for this case.) Algorithm 1 invokes this function for each set of siblings in the class hierarchy.

Algorithm 1: Computing fair shares for HMM fairness.

Input: Link capacity C, a set of $\mathcal{N}$ of classes with $r_i, w_i \geq 0$ for each class $i \in \mathcal{N}$, and a class hierarchy M supporting the operations in Fig. 2.
Output: Fair shares $\{f_i\}_{i \in \mathcal{I} \cup \mathrm{root}}$ from (3).

```
Function HMaxMinFair(N, M, {r_i}_{i∈N}, {w_i}_{i∈N}, C):
    foreach i ∈ I ∪ {root} do
        r_i ← Σ_{j∈ldesc(i)} r_j
    L ← {root}
    a_root ← min(r_root, C)
    do
        i ← Select an element from set L
        L ← L \ {i}
        if i ∉ L then
            f_i ← MaxMinFair(child(i), {r_j}_{j∈child(i)},
                             {w_j}_{j∈child(i)}, a_i)
            foreach j ∈ child(i) do
                a_j ← min(r_j, w_j f_i)
                L ← L ∪ {j}
    while L ≠ ∅
    return {f_i}_{i∈I∪{root}}
```

III. The HLS scheduler

We next present the Hierarchical Link Sharing (HLS) scheduler which allocates rates according to the principle of HMM fairness. We have implemented HLS as a Qdisc in the Linux kernel [22].

The design of HLS departs from that of HTB and CBQ, which both track the transmission rates of classes using moving averages in CBQ and token buckets in HTB. If a class requires additional bandwidth, both HTB and CBQ allow the class to borrow bandwidth from other classes in a greedy fashion. (HTB and CBQ descriptions prefer the term 'borrow,' but the bandwidth so acquired is never returned.) In contrast, HLS does not measure the transmission rates of classes. Instead it gives transmission quotas to classes such that HMM fairness is satisfied. Minimum rate guarantees follow as a consequence of achieving HMM fairness.

In HLS, each class i is associated with an integer weight $w_i > 0$, which can be set to the rate guarantee of the class (see Sec. II-A).

At its core, HLS is a non-hierarchical DRR scheduler with a time-variable quantum for each class, which we refer to as *quota*. Each round of the round robin visits each class that is designated as *active*, one by one, in an arbitrary order. A leaf class is active if it is backlogged at the start of a round. An internal class is active if at least one of its child classes is active. We distinguish two kinds of rounds, *main rounds* and *surplus rounds*, where each main round may be followed by one or more surplus rounds. The quota of a class is recomputed and assigned during a visit in a main round. If at the end of a main round some classes have unused quota, a surplus round is started, where the unused quotas are distributed to

active classes. An additional surplus round is started if after the completion of a surplus round there is still unused quota left.

Each active leaf class is visited once per round (main or surplus). The determination of the set of active classes is done at the start of a round. If a class becomes idle during a round, it remains idle until the end of that round, even if there is an arrival to that class in the middle of the round. We use $\mathcal{L}_{\rm ac}$ and $\mathcal{I}_{\rm ac}$, respectively, to denote the set of active leaf and internal classes in a round.

Each class i maintains a *balance*, denoted by B_i, which maintains the number of bytes that the class is allowed to transmit (if $i \in \mathcal{L}$) or that its leaf descendants are allowed to transmit (if $i \in \mathcal{I}$) in the current round. The initial setting is

$$B_i = \begin{cases} Q^*\,, & \text{if } i = \text{root}\,, \\ 0\,, & \text{otherwise}\,, \end{cases}$$

where $Q^* > 0$ denotes the total number of bytes from all classes that can be transmitted in a round. In Section IV, we address how to select Q^*. In a main round, the root distributes its balance across its child classes, who, in turn, distribute their balance to their own child classes, and so forth. The root and active internal classes also maintain a *residual*, denoted by R_i ($i \in \mathcal{I}_{\rm ac} \cup \{\text{root}\}$), which contains permits for the transmission of bytes that were collected from descendants in the previous round, with initial setting $R_i = 0$.

In each round, all active internal classes recompute the number of bytes that a child class with weight set to one can transmit in the round, which is referred to as the *fair quota* and denoted by F_i for class i. For a class $i \in \mathcal{I}_{\rm ac} \cup \{\text{root}\}$, the fair quota is defined as

$$F_i = \left\lfloor \frac{B_i}{w_i^{\rm ac}} \right\rfloor\,, \tag{4}$$

where

$$w_i^{\rm ac} = \sum_{k \in \text{child}(i) \cap (\mathcal{L}_{\rm ac} \cup \mathcal{I}_{\rm ac})} w_k$$

denotes the sum of the weights of the active child classes of class i. The rounding by the floor function avoids floating point operations in the Linux kernel. The root class recomputes $F_{\rm root}$ only in a main round and sets $F_{\rm root} = 0$ in all surplus rounds.

Before computing the fair quota, each class $i \in \mathcal{I}_{\rm ac} \cup \{\text{root}\}$ updates its balance and residual. For the root class the update is

$$B_{\rm root} = B_{\rm root} + R_{\rm root}\,, \quad R_{\rm root} = 0\,, \tag{5}$$

that is, the residual is added to the balance and then reset. For an active internal class, the update is

$$B_i = B_i + R_i + w_i F_{p(i)}\,,\ B_{p(i)} = B_{p(i)} - w_i F_{p(i)}\,,\ R_i = 0\,. \tag{6}$$

Here, the balance of class i is increased by $w_i F_{p(i)}$, and the balance of the parent $p(i)$ is reduced by the same amount. We refer to $w_i F_{p(i)}$ as the *quota* of class i. Also, the residual R_i is added to the balance and then reset. Since the quota of an internal class depends on the fair quota of the parent class, the update of balances and computations of the quota is performed in a top down fashion. Without the rounding in (4), every internal class would have a zero balance after the update. With rounding, the remaining balance of a class $i \in \mathcal{I}_{\rm ac} \cup \{\text{root}\}$ after the update of all its active child classes is bounded by $B_i < w_i^{\rm ac}$. Note that the unit of $w_i^{\rm ac}$ is in bytes, since it is the remainder of the integer division in (4).

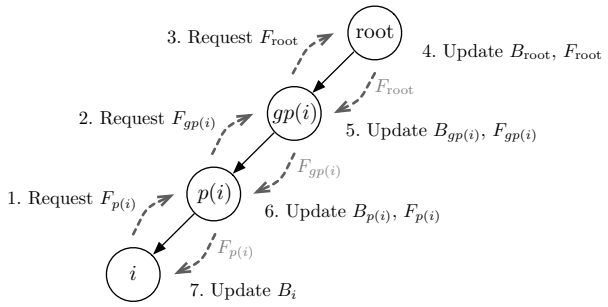


Fig. 3. Updating fair quotas and balances.

Before an active leaf class $i \in \mathcal{L}_{\rm ac}$ transmits in a round, it performs the update

$$B_i = B_i + w_i F_{p(i)}\,, \quad B_{p(i)} = B_{p(i)} - w_i F_{p(i)}\,. \tag{7}$$

In the HLS Qdisc implementation, the update of balances in (5)–(7) and the computation of fair quotas in (4) is initiated by the leaf classes, which is illustrated in Fig. 3. In the figure, node i represents an active leaf class. When this class is visited in the current round it requests the fair quota $F_{p(i)}$ from its parent. If the parent has not previously computed its quota in the current round, it sends a request for the fair quota $F_{gp(i)}$ to its own parent $gp(i)$ (we use $gp(i)$ to denote the grandparent of class i), and so forth. If the root is reached and the scheduler is in a main round, the balance $B_{\rm root}$ and the fair quota $F_{\rm root}$ is computed, and then $F_{\rm root}$ is passed to $gp(i)$. In a surplus round the root returns $F_{\rm root} = 0$. Next, the internal classes $gp(i)$ and $p(i)$ use the fair quotas from their respective parent to update their balances and compute their own fair quotas. In the last step, leaf class i updates its own balance. When the requests (steps 1–3 in Fig. 3) reach a class that already has computed its quota in the current round, no further upstream requests are made. In this fashion, each class $i \in \mathcal{I} \cup \{\text{root}\}$ updates its quota only once and updates its balance at most $1 + |\text{child}(i)|$ times per round.

When an active leaf class i is visited by the round robin, it updates its balance B_i according to (7) before transmitting packets. If the packet at the head of the queue has length L and $B_i \geq L$, the packet is transmitted, followed by the update

$$B_i = B_i - L\,, \quad B_{\rm root} = B_{\rm root} + L\,. \tag{8}$$

By increasing $B_{\rm root}$ for each transmitted packet, the root accrues a balance that will be distributed in the next main round. Class i can continue transmitting packets as long as it has a sufficient balance. If the packet at the head of the queue

has size L and $B_i < L$, the scheduler turns to the next class in the round robin. If a leaf class i is served and has no more packets to transmit, it becomes *idle* and returns its remaining balance to its parent. Say, class i becomes idle with a balance of B_i. Then it performs the update

$$R_{p(i)} = R_{p(i)} + B_i\,, \quad B_i = 0\,. \tag{9}$$

If a leaf class that has become idle during a visit of the round robin has returned its balance to its parent, and the parent has no other active child classes then the parent becomes idle, and performs itself the steps in (9).

Now we see the role of the residual. The residual R_i of an internal class i or the root collects the returned balances from child classes that became idle in the current round. The rationale for not adding the returned balance of an idle child class immediately to the balance of the parent is to prevent the returned balance from being used in the current round. Doing so would favor leaf classes that are visited later in the round robin. By adding the residual to the balance only at the start of a new round, we ensure that all descendants can obtain a portion of the unused balance.

HLS starts a new main round only if the sum of all quotas that has been distributed to classes has been used for transmissions. Note that a class does not use up its full quota only if it became idle in the current round. This results in the unused balance ('surplus') being added to the residual of the parent class. If this happens, the residuals accrued in a round will be distributed to descendant classes in subsequent *surplus rounds*. The condition to start a surplus round is that at least one internal class i satisfies $B_i + R_i \geq w_i^{\mathrm{ac}}$, meaning that the class computes a nonzero quota in (4) using its balance and residual. A surplus round operates just like a main round. First, all backlogged classes are marked as active followed by a complete round robin of active classes with the updates from (4)–(9). The only difference to a main round is that F_{root} is set to zero, meaning that no new quota is distributed from the root. If, at the end of a surplus round, there still exists an internal class j with $B_j + R_j \geq w_j^{\mathrm{ac}}$ another surplus round is started. This continues, until no internal class satisfies the condition, in which case a new main round is started.

With the updates of the balance counters in (5)–(9), the sum of balances and residuals of all classes satisfies the invariance

$$\sum_{i\in\mathcal{N}} B_i + \sum_{i\in I\cup\{\mathrm{root}\}} R_i \equiv Q^*\,.$$

Since balances are permits for transmission and the residuals are unused permits for transmission, maintaining the invariance ensures that the maximum amount of traffic transmitted in a round does not drift.

IV. Analysis

In this section we derive a sufficient condition for a lower bound on Q^*, which is the maximum number of bytes that can be transmitted in a round. We also investigate how well HLS approximates a hierarchical bit-by-bit round robin. In this section, we use $L_i^{\max}$ to denote the maximum packet size of class $i \in \mathcal{L}$.

A. Selection of Q^*

Since Q^* determines the duration of a round, it should not be selected too large, otherwise, the scheduler reacts too slowly to changes of the set of active leaf classes. At the same time, if Q^* is selected too small, the quotas that are passed down to leaf classes may not allow the transmission of any packet. If this happens, the scheduler is looping indefinitely through main rounds without any transmission. The following theorem, which is proven in the appendix, presents a lower bound on Q^*.

Theorem 2. *Setting*

$$Q^* = \sum_{i\in\mathcal{L}\cup\mathcal{I}} w_i + \sum_{i\in\mathcal{L}_{\mathrm{ac}}} L_i^{\max}$$

ensures that at least one packet can be transmitted in each main round.

Note that the first term is constant and that the second term depends on the set of active leaf classes.

We take advantage of the theorem in the Linux Qdisc implementation, where we adjust Q^* dynamically to the set of active leaf classes at the start of each round, using the residual R_{root}. When a class i becomes active we set $R_{\mathrm{root}} = R_{\mathrm{root}} + L_i^{\max}$, where $L_i^{\max}$ is set to the MTU of 1500 bytes. When a class i becomes idle, we set $R_{\mathrm{root}} = R_{\mathrm{root}} - L_i^{\max}$, which may result in $R_{\mathrm{root}} < 0$ and, after the update in (5), in $B_{\mathrm{root}} < 0$. In this case, we set the fair quota of the root to $F_{\mathrm{root}} = 0$ for the next round.

B. Fairness Analysis

To evaluate how well HLS realizes an HMM fair allocation for time-variable traffic, we use a fairness metric that measures the deviation from the allocation of an ideal hierarchical bit-by-bit round-robin scheduler. The fairness metric is defined as follows.

Definition 1. *A scheduling algorithm is HMM(α) fair if, for any two sibling classes i and j that are backlogged in an arbitrary time interval $[t_1, t_2]$, it holds that*

$$\left|\frac{D_i(t_1,t_2)}{w_i} - \frac{D_j(t_1,t_2)}{w_j}\right| \leq \alpha,$$

where $D_i(t_1, t_2)$ is the number of bytes that class i or its leaf descendants transmit in the interval $[t_1, t_2]$.

The left-hand side of the equation is the weighted difference between the number of bytes that two classes i and j transmit. Since the difference is zero in an ideal (fluid flow) scheduler, the bound α expresses how far a particular scheduling algorithm deviates from an ideal link sharing scheduler.

The next theorem provides the HMM fairness metric of HLS. The proof is provided in our technical report [23]. The theorem requires the following definitions:

$$\tilde{A}_i = \begin{cases} L_i^{\max} - 1, & i \in \mathcal{L}, \\ \sum_{j \in \mathrm{desc}(i)} w_j + \sum_{j \in \mathrm{ldesc}(i)} L_j^{\max}, & i \in \mathcal{I}, \end{cases}$$

$$\bar{A}_i = \tilde{A}_{\mathrm{rc}(i)} + w_{\mathrm{rc}(i)} \left(1 + \max_{j \in \mathrm{child(root)}} \frac{\tilde{A}_j}{w_j} \right), \ (i \in \mathcal{I} \cup \mathcal{L}),$$

with $rc(i)$ denoting the ancestor of class i that is a child class of root, given by

$$\mathrm{rc}(i) = \begin{cases} i, & i \in \mathrm{child(root)}, \\ \mathrm{rc(p}(i)), & \text{otherwise}, \end{cases}$$

With this notation, the fairness bound of HLS is given as follows.

Theorem 3. *HLS is HMM($\alpha^{(\mathrm{HLS})}$)-fair with*

$$\alpha^{(\mathrm{HLS})} = \max_{i \in \mathcal{N}, j \in \mathrm{sib}(i)} \left\{ \frac{\bar{A}_i}{w_i} + \frac{\bar{A}_j}{w_j} \right\}.$$

Let us compare the fairness bounds of HLS and with the bound derived for HDRR in [13], denoted by $\alpha^{(\mathrm{HDRR})}$ for the hierarchy in Fig. 1, where we set the weight of a class to its rate guarantee and set the maximum packet size of each leaf class i to $L_i^{\max} = 1500$ bytes. For HDRR, we also set its quantum Q to the maximum packet size of 1500 bytes. We obtain

$$\alpha^{(\mathrm{HLS})} = 103.5 \text{ bytes}, \qquad \alpha^{(\mathrm{HDRR})} = 1522.5 \text{ bytes}.$$

Here, HLS clearly has a better fairness metric. In general, due to the very different operations of HLS and HDRR, it is not feasible to show that HLS always has a better fairness metric. In fact, for deep hierarchies, the fairness metric of HDRR can be better than that of HLS.

In our technical report [23] we also analyze and compare a second performance metric, referred to as *transmission gap*, which bounds the elapsed time between two visits of the same class and the time until a newly backlogged class receives service in a round-robin scheduler.

V. Evaluation

We have implemented HLS as a kernel module in Linux kernel 4.15.0-101-generic [22]. A description of the implementation is available in [24]. Here we present measurement experiments of the HLS Qdisc in Linux and compare them with measurements of the existing link sharing schedulers in Linux, CBQ, and HTB. Other fair scheduling methods available in Linux, such as DRR, Fair Queueing (FQ), and Stochastic Fair Queueing (SFQ) are not considered in our experiments since they do not apply to class hierarchies. All experiments are conducted on the Emulab testbed at the University of Utah [25].

A. Experimental Setup

The topology of the experiments involves three Linux servers as shown in Fig. 4, designated as *traffic generator*, *scheduler*, and *traffic sink*. Each server is a Dell PowerEdge R430 with two 2.4 GHz 8-Core CPUs, 64 GB RAM, a dual-port/quad-port 1GbE PCI-Express NICs, and a dual-port/quad-port Intel X710 10GbE PCI-Express NICs. The servers run Ubuntu 18.04LTS. The traffic generator and the scheduler are connected by a 10 Gbps Ethernet link, and the scheduler and the traffic sink are connected by a 1 Gbps Ethernet link. Routing tables of all servers are set up statically so that all traffic is routed from the traffic generator to the traffic sink. The link sharing schedulers are configured at the egress of the 1 Gbps interface at the scheduler node. The traffic generator uses FIFO scheduling. With this setup we can saturate the outgoing link at the scheduler without overloading its CPUs.

In the first two experiments, the traffic generator sends UDP/IPv4 datagrams with a length of 1000 bytes, where destination port numbers are mapped to classes at the scheduler node. Our graphs plot the transmission rates of traffic classes using jumping windows with length 0.2 s. The rate at which the traffic generator sends packets is such that it ensures that each active leaf class is permanently backlogged at the egress of the scheduler node. The design of the experiments is similar to experiments in [11], [16], [17], [26] or, more recently, in [27]. By turning traffic sources and and off, the experiments show how quickly a scheduler reacts to changes of the traffic load as well as the resulting rate allocations.

B. Experiment 1: Validation of HMM fairness

In this experiment, which is a scaled version of an experiment in [16], [26], we show that HLS quickly converges to an HMM fair allocation when the set of active flows changes. The class hierarchy of the experiment is as shown in Fig. 1 for a 1 Gbps link, but with the following rate guarantees:

Class:	A	B	$A1$	$A2$	$B1$	$B2$
Rate guarantee:	700	300	300	400	100	200

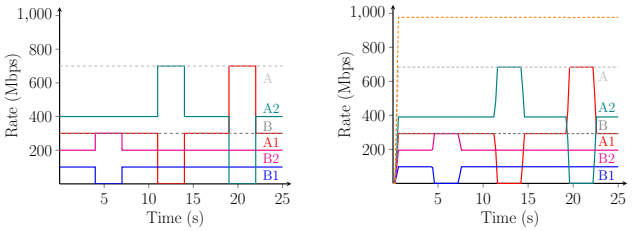
In the experiment, all leaf classes are initially active and transmit packets with a fixed packet size of 1000 bytes. At certain time intervals, one class becomes idle, in the following sequence:

Interval (in seconds):	$[4, 7]$	$[11, 14]$	$[19, 22]$
Inactive class:	$B1$	$A1$	$A2$

The throughput of the classes is shown in Fig. 5. The plot for each class is labeled. Dashed gray lines show the aggregate traffic of the internal classes A and B. The dashed line with label 'Total' indicates the aggregate traffic from all classes, which is at or close to the link capacity of 1 Gbps.

Fig. 5(a) shows the HMM fair allocations from Theorem 1. The measured rates for HLS in Fig. 5(b) show that HLS satisfies HMM fairness for all classes at all times. When all leaf classes are active, they each obtain their class guarantees. If one class drops out, the sibling class consumes the guarantee of its sibling.

Fig. 4. Network topology for experiments.



(a) HMM fair allocation. (b) HLS Qdisc (Linux).

Fig. 5. Experiment 1: HMM fairness.

C. Experiment 2: Isolating class guarantees

This experiment illustrate the need for isolating class guarantees, and the inability of the existing link sharing schedulers CBQ and HTB to realize isolation between classes. The experiment uses the class hierarchy from Fig. 1 (in Sec. I). In addition to the guarantees shown in the figure, we vary the guarantees of classes $A1, A2, B1, B2$ to evaluate three scenarios, labeled as 'L', 'M', 'H', which stands for low, medium, and high differences between the guarantees. The guarantees in the scenarios (in Mbps) are as follows:

Class:	$A1$	$A2$	$B1$	$B2$	A	B	C
Low 'L'	140	160	140	160	300	300	400
Medium 'M'	100	200	100	200	300	300	400
High 'H'	60	240	60	240	300	300	400

The 'M' scenario corresponds to the guarantees shown in Fig. 1. The guarantees of classes A, B, and C are the same in all three scenarios, and are as shown in Fig. 1.

In the experiment, three leaf classes $(A1, B2, C)$ generate traffic. In the middle of the experiment, in the interval $[10, 20]$ s, class C pauses transmissions. Since leaf classes $A1$ and $B2$ compete with each other at the level of their respective parent classes A and B, their allocation should be determined by the guarantees of the parents. If this is the case, $A1$ and $B2$ receive the same allocation in all three scenarios.

Fig. 6(a)–6(c) show the measured throughput of the HLS Qdisc. In all three scenarios, the throughput of active classes corresponds to the HMM fair allocation. When all three classes are active (in $[0, 10]$ s and $[20, 25]$ s) they split the allocation in the ratio $3 : 3 : 4$, according to the guarantees of classes A, B, C. When class C drops out, $A1$ and $B2$ split the capacity evenly, since A and B have the same guarantee.

The second row of graphs in Fig. 6 presents measurements of the HTB Qdisc. First note that, for all scenarios the minimum rate guarantees of internal and leaf classes are maintained at all times. When all three classes are active, they have the same allocation as HLS. However, when class C drops out, classes $A1$ and $B2$ do not split the freed up link capacity evenly. Instead, the throughput appears to depend on the guarantees of the active leaf classes $A1$ and $B2$. As seen in Figs. 6(e) and 6(f), by increasing the guarantee of class $B2$ and decreasing that of $A1$, the allocation becomes more lopsided.

The throughput in the scenarios under CBQ, depicted in the last row of graphs in Fig. 6, shows that CBQ allocates rates in a similar fashion as HTB. That is, sharing of bandwidth at the level of internal classes does not respect the guarantees of the internal classes. As with HTB, the allocation appears to be again determined by the guarantees of the leaf classes.

The observed link sharing of CBQ and HTB indicates a lack of isolation between classes in the hierarchy. Here, class B can manipulate its allocation by bundling its traffic in a single descendant class, at the cost of class A. The HMM fair allocation of HLS does not allow this to happen.

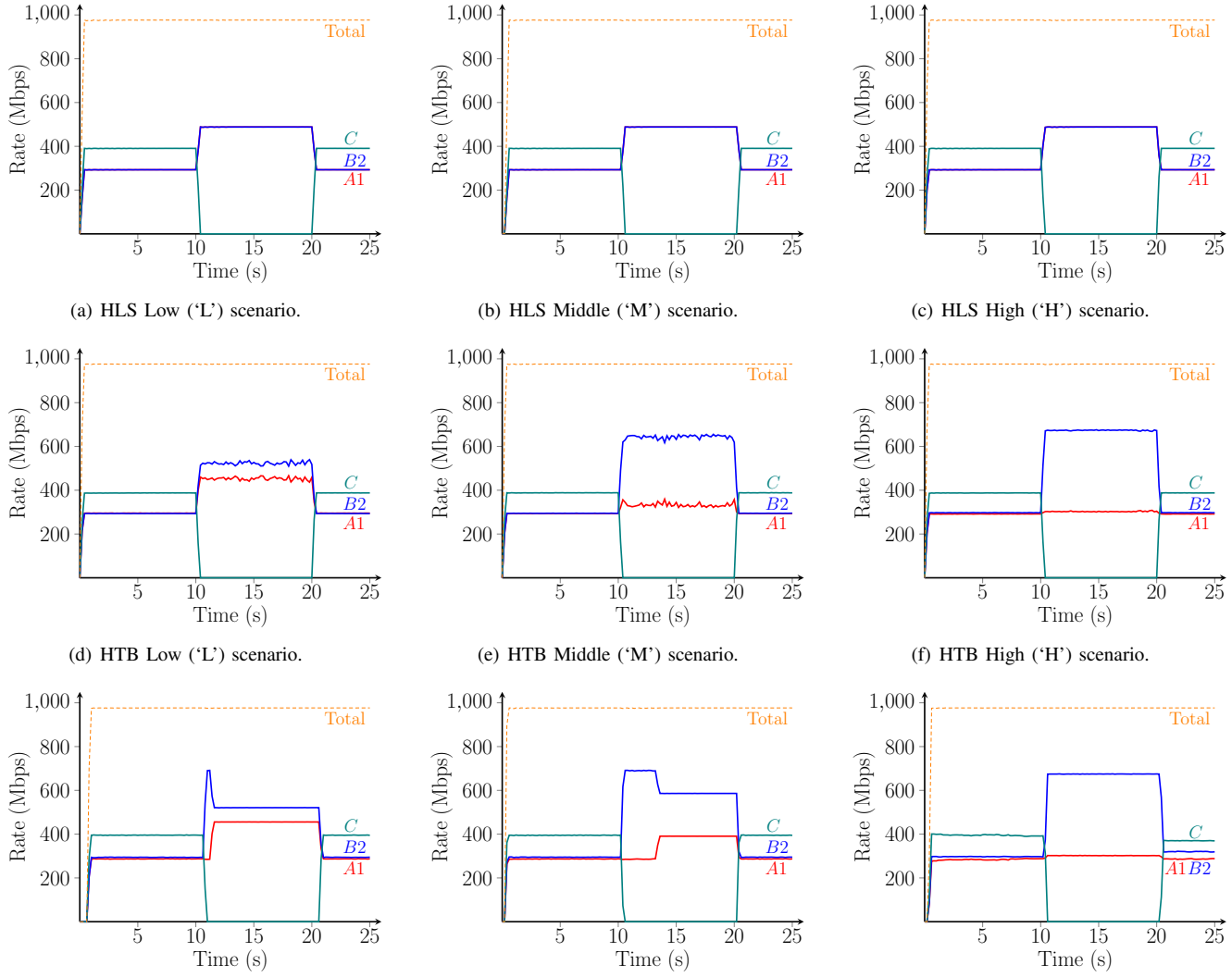
D. Experiment 3: Overhead

We next measure the processing overhead of HLS and compare it to that of CBQ and HTB. Since all schedulers are implemented as Linux Qdiscs, they share the performance limitations of the Qdisc framework, in particular, the single Qdisc lock. In order to move the bottleneck of the experimental setup to per-packet processing, we replace the 1 Gbps link in Fig. 4 between the scheduler and the traffic sink by a 10 Gbps link, and we send small packets. We verified that the servers that run the traffic generator and traffic sink are not bottlenecks in the experiment.

The experiment uses NetPerf TCP-RR [28], similar to an experiment in [19, Fig. 6]. TCP senders and receivers send 1-byte packets in each direction in a ping-pong fashion. One round of the ping-pong is called a transaction. The traffic from each TCP sender is mapped to a separate leaf class at the scheduler node (in Fig. 4). The performance metric is the total number of completed transactions per second.

We consider two class hierarchies: a full binary tree and a flat hierarchy. In a binary tree, with N leaf classes, the total number of classes, including the root, is $2N - 1$. In the flat hierarchy, the root class has N children which are all leaf classes. For HLS we set the weight of every class to one in both scenarios. For HTB and CBQ, the rate guarantees are divided evenly between the leaf classes. In addition to the hierarchical scheduling algorithms HLS, HTB, and CBQ, we also include measurements with FIFO scheduling. Since FIFO is a classless scheduler, outcomes are not sensitive to the class hierarchy.

Fig. 7(a) shows the transactions per second as a function of the number of leaf classes for the binary tree hierarchy. All schedulers show roughly the same performance. The number of transactions initially increases linearly with the leaf classes and saturates at around 300K transactions (100K $= 10^5$). Since HTB limits the number of levels in the class hierarchy, the binary tree hierarchy cannot be increased beyond 128 leaf classes. The fact that FIFO sometimes has worse results than the hierarchical schedulers indicates the degree of randomness in experiments that involve a large number of TCP flows.



(a) HLS Low ('L') scenario. (b) HLS Middle ('M') scenario. (c) HLS High ('H') scenario.

(d) HTB Low ('L') scenario. (e) HTB Middle ('M') scenario. (f) HTB High ('H') scenario.

(g) CBQ Low ('L') scenario. (h) CBQ Medium ('M') scenario. (i) CBQ High ('H') scenario.

Fig. 6. Experiment 2: HLS is HMM fair. With HTB and CBQ, class B can increase its allocation by bundling its guarantee and traffic in one child class.

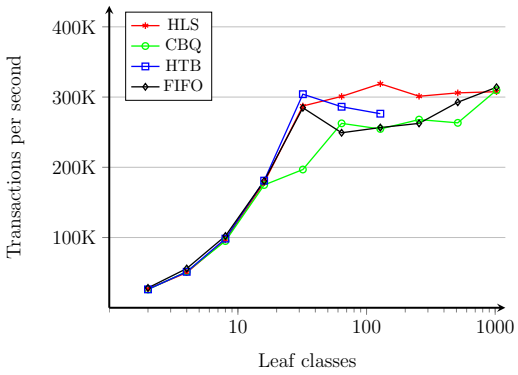
Fig. 7(b) depicts the results for the flat hierarchy. Here, the number of transactions initially increases and plateaus at around $300K$ transactions, similar to the binary tree scenario. After around 256 leaf classes, however, the performance of the classful schedulers declines. A comparison with FIFO points to a performance bottleneck that arises during packet classification, which impacts all classful schedulers in the same way. The classification compares the packet destination port to the port associated to each leaf class, and the number of comparisons grows linearly with the number of leaf classes.

The experiment leads us to conclude that the HLS Qdisc does not incur a performance penalty, compared to HTB and CBQ. In fact, since the schedulers perform similarly to FIFO, none of the hierarchical schedulers presents a bottleneck.

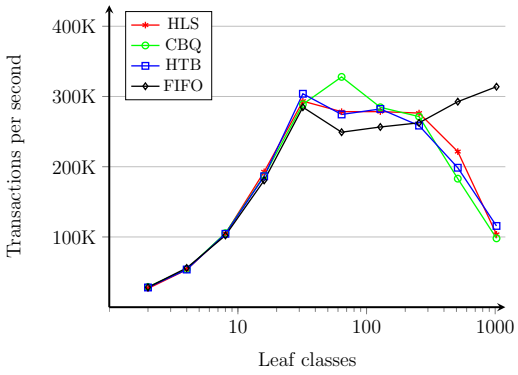
We emphasize that the outcome of this experiment is sensitive to the configuration of filters that map packets to traffic classes. In Fig. 7, the mapping of packets to classes is done progressively. Each internal node in the hierarchy has (two) filter expressions for mapping traffic to its child classes. Alternatively, the mapping can be performed at the root Qdisc for all leaf classes. If this is done, the results show a precipitous drop of completed transactions when the number of leaf classes exceeds 100.

VI. Related Work

There are several reasons for the recent surge of interest in shaping and scheduling algorithms. First, the increased flexibility of recent programmable packet switches has enabled customization of scheduling algorithms to application requirements [12], [29], [30]. Second, the Ultra-Reliable Low-Latency Communication service category in 5G networks, which guarantees latencies below 1 ms has led to standardization efforts by the IEEE (for Layer-2) and by the IETF (for Layer-3) for compatible protocol frameworks [31] and traffic control algorithms [32]. Third, an increased demand for fine-



(a) Binary tree class hierarchy.



(b) Flat class hierarchy.

Fig. 7. Experiment 3. Overhead.

grain control of traffic in data centers has created a need for advanced packet scheduling methods at servers [1], [2].

These efforts benefit from an intense period of research in the 1990s that created many of the scheduling and shaping algorithms in use today [4], [8], [16], [33]. Recent research on packet scheduling has put emphasis on generality, e.g., PIFO [12], UPS [34], and efficient implementations, for example, Carousel [19], Eiffel [20], Loom [27], CQ [35].

Most relevant to our paper is the claim in [12] of realizing HPFQ by a hierarchy of PIFO queues. However, the claim holds only when packets have a fixed size. For variable-sized packets and classes with different weights the arrival of a packet may require changing the relative order of packets in the PIFO buffers. By design, PIFO does not support reshuffling buffered packets.

BwE [18] performs a centralized rate allocation for hierarchically organized inter-data center traffic, which computes end-to-end max-min fair rate allocations for a network setting, which are enforced by HTB ceiling rates. Interestingly, in [18, Sec.9] it is argued that fair queueing is not suitable since 'weights are insufficient for delivering user guarantees.' By showing the equivalence of rate guarantees and weights in Sec. II-A, our paper invites a correction of the above statement.

We have not included HFSC [36], [37] in this paper, even though it is another hierarchical scheduler available in Linux. HFSC is hybrid scheduler that deterministically guarantees service curves and shares excess bandwidth with fairness objectives. As pointed out in [36] it is, in general, not possible to simultaneously guarantee the service curves of HFSC and its fairness criteria. Since HFSC resolves conflicting guarantees by giving priorities to service curves, the role of link sharing is limited. We also note that HFSC realizes so-called 'lower service curves' [38]. Rate guarantees of these service curves share a drawback with the VirtualClock scheduler [39], where a class that is served above its guaranteed rate for some time, may be later served at a rate below its guarantee. Differently, fair scheduling algorithms realize 'strict service curves' [38], which ensure rate guarantees for every time interval where a class is backlogged.

VII. Conclusions

We presented a round-robin scheduler for hierarchical link sharing that ensures rate guarantees and isolation between classes, and which is suitable for supporting high line rates. The presented HLS scheduler resolves shortcomings of deployed hierarchical link sharing algorithms when distributing excess capacity to traffic classes. The link sharing in HLS is strategy-proof in that a class that needs more bandwidth cannot increase its allocation by increasing its transmissions or misrepresenting the class hierarchy of its descendants. We have shown that the implementation of HLS does not create a performance bottleneck. The paper does not study how network interface card (NIC) offloading interferes with the HLS scheduler. Other future work is an extension of the Qdisc implementation of HLS to also enforce maximum (ceiling) rates.

Appendix

We can ensure that, in any main round, at least one packet can be transmitted by satisfying the condition

$$\sum_{i\in\mathcal{L}_{\mathrm{ac}}} B_i \geq \sum_{i\in\mathcal{L}_{\mathrm{ac}}} L_i^{\max}. \tag{10}$$

Then, by the pigeon hole principle, there is at least one active leaf class i with $B_i \geq L_i^{\max}$.

Consider the start of a round before any transmission in the round. Without changing the outcome, suppose all classes perform the updates of (5)–(7) at once. Then, we have $\sum_{i\in\mathcal{N}} B_i = Q^*$, and we can write (10) as

$$Q^* \geq \sum_{i\in\mathcal{I}\cup\{\mathrm{root}\}} B_i + \sum_{i\in\mathcal{L}_{\mathrm{ac}}} L_i^{\max}.$$

Recall that, after updating the balances of all classes, the remaining balance of an internal class or the root satisfies $B_i < w_i^{\mathrm{ac}}$, which we relax to $B_i < w_i^c$, where $w_i^c = \sum_{j\in\mathrm{child}(i)} w_i$ is the aggregate weight of all child classes. Summing up we obtain

$$\sum_{i\in\mathcal{I}\cup\{\mathrm{root}\}} B_i \leq \sum_{i\in\mathcal{I}\cup\{\mathrm{root}\}} w_i^c = \sum_{i\in\mathcal{I}\cup\mathcal{L}} w_i.$$

Hence, by setting Q^* as given in the theorem, we ensure that (10) is always satisfied, meaning that there is at least one packet transmission in each main round.

REFERENCES

[1] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *Proc. ACM Sigcomm*, 2013.

[2] M. Noormohammadpour and C. S. Raghavendra, "Datacenter traffic control: Understanding techniques and tradeoffs," *IEEE Communications Surveys Tutorials*, vol. 20, no. 2, pp. 1492–1525, 2018.

[3] D. Bertsekas and R. Gallager, *Data Networks*, 2nd ed. Prentice-Hall, 1992.

[4] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," in *Proc. ACM Sigcomm*, 1989.

[5] 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 Transactions on Networking*, vol. 1, no. 3, pp. 344–357, June 1993.

[6] S. J. Golestani, "Network delay analysis of a class of fair queueing algorithms," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 13, no. 6, pp. 1057–1070, 1995.

[7] P. Goyal, H. M. Vin, and H. Chen, "Start-time Fair Queueing: a Scheduling Algorithm for Integrated Services Packet Switching Networks," in *Proc. ACM Sigcomm*, 1996.

[8] M. Shreedhar and G. Varghese, "Efficient fair queueing using Deficit Round Robin," *IEEE/ACM Transactions on Networking*, vol. 4, no. 3, pp. 375–385, 1996.

[9] M. A. Brown. (2006) Traffic control howto (version 1.0.2). [Online]. Available: https://tldp.org/HOWTO/Traffic-Control-HOWTO/

[10] C. Systems. (2017) Modular qos configuration guide for cisco crs routers, ios xr release 6.2.x. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/routers/crs/software/crs-r6-2/qos/configuration/guide/b-qos-cg-crs-62x.pdf

[11] J. C. Bennett and H. Zhang, "Hierarchical packet fair queueing algorithms," *IEEE/ACM Transactions on Networking*, vol. 5, no. 5, pp. 675–689, 1997.

[12] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet scheduling at line rate," in *Proc. ACM Sigcomm*, 2016.

[13] D. Back, K. Pyun, S. Lee, J. Cho, and N. Kim, "A hierarchical deficit round-robin scheduling algorithm for a high level of fair service," in *International Symposium on Information Technology Convergence (ISITC 2007)*, 2007.

[14] S. S. Kanhere and H. Sethu, "Fair, efficient and low-latency packet scheduling using nested deficit round robin," in *Proc. IEEE Workshop on High Performance Switching and Routing*, 2001.

[15] M.-X. Chen and S.-H. Liu, "Hierarchical deficit round-robin packet scheduling algorithm," in *Advances in Intelligent Systems and Applications - Volume 1*. Springer, 2013, pp. 419–427.

[16] S. Floyd and V. Jacobson, "Link-sharing and resource management models for packet networks," *IEEE/ACM Transactions on Networking*, vol. 3, no. 4, pp. 365–386, Aug. 1995.

[17] M. Devera. (2003) Linux hierarchical token bucket. [Online]. Available: http://luxik.cdi.cz/~devik/qos/htb/

[18] A. Kumar *et al.*, "BwE: Flexible hierarchical bandwidth allocation for WAN distributed computing," in *Proc. ACM Sigcomm*, 2015.

[19] A. Saeed, N. Dukkipati, V. Valancius, V. T. Lam, C. Contavalli, and A. Vahdat, "Carousel: Scalable traffic shaping at end hosts," in *Proc. ACM Sigcomm*, 2017.

[20] A. Saeed, Y. Zhao, N. Dukkipati, E. Zegura, M. Ammar, K. Harras, and A. Vahdat, "Eiffel: Efficient and flexible software packet scheduling," in *Proc. NSDI*, 2019.

[21] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. NSDI*, 2011.

[22] N. Luangsomboon. (2021) Hierarchical link sharing (Qdisc). [Online]. Available: https://github.com/lantua/HLS

[23] N. Luangsomboon and J. Liebeherrr, "A round robin packet scheduler for hierarchical max-min fairness," *CoRR*, vol. abs/2108.09864, 2021. [Online]. Available: http://arxiv.org/abs/2108.09864

[24] N. Luangsomboon, "Fast packet scheduling for hierarchical fairness," Master's thesis, University of Toronto, Dept. of Electrical and Computer Engineering, 2021.

[25] B. White *et al.*, "An integrated experimental environment for distributed systems and networks," in *Proc. OSDI 2002*, 2002.

[26] S. Floyd and M. F. Speer. (1998) Experimental results for class-based queueing, draft paper. [Online]. Available: https://ee.lbl.gov/floyd/cbq/report.pdf

[27] B. Stephens, A. Akella, and M. Swift, "Loom: Flexible and efficient nic packet scheduling," in *Proc. NSDI*, 2019.

[28] HP Networking. (2015) Netperf 2.7.0. [Online]. Available: https://hewlettpackard.github.io/netperf/

[29] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," in *Proc. ACM Sigcomm*, 2014.

[30] A. Sivaraman, T. Mason, A. Panda, R. Netravali, and S. A. Kondaveeti, "Network architecture in the age of programmability," *ACM SIGCOMM Computer Communication Review*, vol. 50, no. 1, pp. 38–44, 2020.

[31] A. Nasrallah, A. S. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein, and H. E. Bakoury, "Ultra-low latency (ULL) networks: The IEEE TSN and IETF DetNet standards and related 5G ULL research," *IEEE Communications Surveys Tutorials*, vol. 21, no. 1, pp. 88–145, First Quarter 2019.

[32] J.-Y. LeBoudec, "A theory of traffic regulators for deterministic networks with application to interleaved regulators," *IEEE/ACM Trans. Netw.*, vol. 26, no. 6, pp. 2721–2733, 2018.

[33] H. Zhang, "Service Disciplines for Guaranteed Performance Service in Packet-Switching Networks," *Proceedings of the IEEE*, vol. 83, no. 10, pp. 1374–1399, October 1995.

[34] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal packet scheduling," in *Proc. NSDI*, 2016.

[35] N. K. Sharma, C. Zhao, M. Liu, P. G. Kannan, C. Kim, A. Krishnamurthy, and A. Sivaraman, "Programmable calendar queues for high-speed packet scheduling," in *Proc. NSDI*, 2020.

[36] I. Stoica, H. Zhang, and T. S. E. Ng, "A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services," in *Proc. ACM Sigcomm*, 1997.

[37] ——, "A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-97-154, 1997.

[38] J. Y. LeBoudec and P. Thiran, *Network Calculus*. Springer Verlag, Lecture Notes in Computer Science, LNCS 2050, 2001.

[39] L. Zhang, "Virtual clock: A new traffic control algorithm for packet switching networks," in *Proc. ACM Sigcomm*, 1990.