Approximation of Generalized Processor Sharing with Interleaved Stratified Timer Wheels - Extended Version

Martin Karsten
David R. Cheriton School of Computer Science
University of Waterloo
mkarsten@cs.uwaterloo.ca
Technical Report CS-2009-24, University of Waterloo

***Abstract*—This paper presents *Interleaved Stratified Timer Wheels* as a novel priority queue data structure for traffic shaping and scheduling in packet-switched networks. The data structure is used to construct an efficient packet approximation of General Processor Sharing (GPS). This scheduler is the first of its kind by combining all desirable properties without any residual catch. In contrast to previous work, the scheduler presented here has constant and near-optimal delay and fairness properties, *and* can be implemented with $O(1)$ algorithmic complexity, *and* has a low absolute execution overhead. The paper presents the priority queue data structure and the basic scheduling algorithm, along with several versions with different cost-performance trade-offs. A generalized analytical model for rate-controlled rounded timestamp schedulers is developed and used to assess the scheduling properties of the different scheduler versions. Some illustrative simulation results are presented to reaffirm those properties.**

***Index Terms*—Communication systems, Computer network performance, Packet scheduling, Data structures, Algorithms**

I. Introduction

Packet scheduling algorithms are a cornerstone for the future development of packet-switched networks as ubiquitous communication infrastructure, integrating a wide range of network technologies and offering a wide variety of application services. Packet scheduling algorithms increase the level of control over packet transmissions and permit the support of different service policies. There are many application areas for packet scheduling, ranging from detailed quality of service guarantees for individual application flows [1] to service assurances for aggregates [2]. In each of these scenarios, a more precise scheduler translates into more efficient resource usage in relation to the "quality" of the service guarantees. Because of the significant complexity and execution cost of packet schedulers, the architectural sweet spot of network and capacity planning has been in configurations with very simple schedulers, so far. A feasible packet scheduler with perfect or near-perfect service properties has been elusive. Clearly, the availability of such a packet scheduler will go a long way towards establishing accurate traffic control as a basic building block for packet-switched communication networks.

General Processor Sharing (GPS) has been introduced in [3] as a conceptual scheduler with many desirable properties. In very basic terms, GPS scheduling works by assigning fractions of the overall forwarding capacity to flows. These fractions are termed *weights* and the GPS scheduler guarantees fluid service in proportion to a flow's weight compared to the sum of all other active flows' weights. In reality, packets are atomic units and thus a packet scheduler always deviates somewhat from perfect GPS service. A large variety of GPS emulation algorithms have been proposed, but no algorithm exists so far that combines very close GPS approximation with constant algorithmic complexity *and* low execution overhead.

This paper presents a novel data structure called *Interleaved Stratified Timer Wheels* (ISTW) and appropriate access operations. This design enables the construction of a set of novel packet schedulers with effectively constant complexity, and constant fairness and delay characteristics in all relevant dimensions. The ISTW data structure is used as a compact and efficient priority queue that enables the virtual traffic shaping necessary for achieving these characteristics. Constant complexity in this context is defined as amortized execution cost over a certain amount of input traffic. Amortized cost can be translated into extra buffering and as such, additional delay. In contrast to previous work [4] however, the amortization period for the schedulers presented here is tightly limited and practical. In particular, the worst-case execution cost for all per-packet operations is proportional to the size of the corresponding packet. In an earlier version of this work [5], the SI-WF^{2}Q scheduler has been proposed and analyzed, which requires an additional small but nontrivial amortization buffer to achieve constant execution complexity. The schedulers presented here overcome this limitation. In short, the contributions of this paper are:

- We describe the basic ISTW data structure, and summarize and slightly improve the previous SI-WF^{2}Q proposal. Based on this work, we generalize and improve the analytical model, and demonstrate its applicability.
- We present a new packet scheduler termed *K Packet Scheduler* that avoids the search problem of SI-WF^{2}Q in exchange for only a minor increase of the worst-case fairness bound. This scheduler is thus the first of its kind by combining a near-optimal packet approximation of GPS with constant and low execution overhead.
- We also present a simpler variant of the K Packet Scheduler that further reduces the memory footprint and

complexity in exchange for slightly higher error terms.

The rest of the paper is organized as follows. In Section II, we review previous work and its relation to the approach presented here. In Section III, we introduce and analyze a general model for rate-controlled timestamp schedulers that operate on rounded deadlines. This is followed by the specification of a new priority queue data structure in Section IV and resulting packet scheduler designs and their assessment in Section V. We present canonical simulation results to verify and illustrate the new schedulers' service properties in Section VI and discuss further interesting details in Section VII, before finishing the paper with a conclusion in Section VIII. This paper is an extended version of [6].

II. Background and Related Work

A. Generalized Processor Sharing

Generalized Processor Sharing (GPS) [3] is a conceptual scheduling discipline defined for a set of flows $i \in \mathcal{F}$, such that each flow i is allocated a weight ϕ_i. At each time, assuming a fixed server capacity $\mathcal{C}$ and a set of backlogged flows $\mathcal{B}$, the service for each backlogged flow $i \in \mathcal{B}$ is guaranteed to be

$$\frac{\phi_i}{\sum_{j\in\mathcal{B}} \phi_j}\mathcal{C}.$$

In other words, GPS always shares the available link capacity in perfect proportion to the flows' weights. Because GPS cannot be implemented in reality, there is a class of schedulers that attempt to approximate GPS as good as possible. These schedulers are primarily assessed through quality metrics that describe their deviation from perfect GPS service. On the other hand, the respective algorithmic complexity determines the practical feasibility of each scheduler.

The main quality metrics of a GPS approximation packet scheduler are the *delay bound*, especially in a form that can be used to determine an end-to-end delay bound as shown in several analytical frameworks [2], [7], [8], [9], as well as two fairness measures. *Relative fairness* (introduced in [10]) denotes the capability of a scheduler to distribute excess capacity between different sessions in proportion to their allocated service rates. *Worst-case fairness* (introduced in [11] and refined in [12]) expresses the maximum deviation from perfect GPS scheduling. While the delay bound only characterizes how far the actual service for a session can be *behind* the ideal GPS scheduler during a busy period, worst-case fairness essentially provides an integrated bound on how far *ahead* or *behind* the actual service can be. Fairness thus also describes the burst characteristics of the service allocation in relation to the ideal smooth service of GPS. The key metric for describing the quality of these service characteristics as well as the computational complexity of a packet scheduler is the asymptotic relation between the respective characteristic and the number of flows in the system, which is described as constant, logarithmic, or linear. For example, constant delay describes the property that the delay bound is independent of the number of flows. Any real-world packet-oriented scheduler needs to operate on packets as atomic service units and thus cannot avoid a certain service deviation from the fluid GPS model.

Existing GPS emulation algorithms can be classified as timestamp schedulers, round-robin schedulers, and hybrid versions. Different schedulers provide different combinations of the aforementioned scheduling quality characteristics, while none of the existing proposals is optimal in all quality dimensions and also of low complexity and execution overhead. There is a class of fundamentally different schedulers, termed *Service Curve Schedulers* [13], [14], which can provide delay bounds independently of throughput guarantees. These schedulers are inherently more complex than GPS emulation schedulers, so it seems hopeless to think about efficient implementation before solving the GPS emulation problem. Hence, although the techniques presented here may be applicable to such schedulers, details are out of scope for this paper.

B. Timestamp Schedulers

Timestamp schedulers approximate GPS behaviour by simulating the virtual system time in the equivalent GPS system. The respective start and/or finish times of packets in the reference GPS system are used to decide the order in which packets receive service. By the same token, the simulated virtual time is used as a starting point for newly arriving flows, which is necessary to achieve at least some bound on unfairness and burstiness. This challenge is well-known since the earliest proposals for rate-proportional packet scheduling [15], [16]. Note that the order of packets within a flow is not affected by the scheduler operation. Therefore, only the first packet in each flow's queue needs to be considered for scheduling and the term "flow timestamp" is often used when referring to the packet timestamp at the head of a flow's queue.

C. Worst-case Fair Weighted Fair Queueing (WF^{2}Q)

An optimal packet-based approximation of GPS is given by *Worst-case Fair Weighted Fair Queueing* (WF^{2}Q) [12]. The deviations from GPS scheduling are bound by strictly rate-dependent values for the respective flow (or two flows in the case of relative fairness) with provably optimal coefficients. In particular, all scheduling errors are independent of the number of flows in the system. Earlier attempts at approximating GPS, such as proposed in [3], [10], [15], [17], incur a potentially linear deviation from GPS scheduling, in terms of either fairness or delay behaviour. In fact, it turns out that when considering only packet start times for scheduling, the startup delay cannot be limited effectively and the delay bound depends on the number of flows in the system. When scheduling packets only by increasing finish times, packet bursts and unfairness can occur, also bound only by the number of flows.

Conceptually, the WF^{2}Q algorithm works by combining both criteria, start and finish time. In a first shaping step all eligible packets are selected, that is all packets with a start time not later than the current system virtual time. From all these packets, the one with the smallest finish time is sent next. This packet selection policy, termed *Start-eligible Earliest Finish-time First* (SEFF) ensures tight bounds for all quality indices. Stiliadis and Varma [18] independently arrive at the same conclusion that the combination of traffic shaping and finish-time service results in optimal scheduling characteristics. While [18] only describes a very simplified implementation in the context of fixed-size packet networks (ATM in this case), the original WF^{2}Q proposal [12] is not at all concerned with algorithmic complexity or execution overhead.

D. WF^{2}Q Approximation

There are two proposals for implementing an approximation to WF^{2}Q with lower complexity: WF^{2}Q+ [19] and *Leap Forward Virtual Clock* (LFVC) [4]. The work by Stiliadis and Varma [18] contains a similar concept, but does not elaborate on all details. In general, all SEFF-based algorithms contain three parts that are relevant for their execution overhead and complexity:

- Flows are sorted according to timestamps.
- The SEFF policy requires consideration of both the start and the finish timestamp for the scheduling decision.
- The virtual time of the GPS reference system needs to be simulated or approximated.

Sorting and priority queues are among the best-studied problems in Computer Science. Without further restrictions, maintaining a sorted container has $O(\log N)$ complexity in the number of elements. In the context of GPS emulation schedulers, however, it can be a very acceptable trade-off to use rounded time values (and incur some additional scheduling error) in exchange for a finite universe of sorting values, which enables more efficient solutions to the sorting problem. For example, the van Emde Boas priority queue [20] has $O(\log\log N)$ access complexity for insertion, removal and finding the lowest value. Similarly, a timer wheel [21] could operate in $O(1)$ for insertion or removal and, in combination with hierarchical bitmaps and a priority encoder of width K, with $O(\log_K N)$ complexity for searching the lowest value (see Section 5 in [22] for a brief discussion). In both cases, a finite time horizon must be assumed, which translates into a maximum specifiable inter-arrival time of packets. Since the maximum packet inter-arrival time is part of the lower bound on the startup delay, one can assume that there is an upper limit to this value and it will not change in future networks. Then, if the desired delay precision is also fixed in terms of wall-clock time, one could argue that these algorithms operate with constant complexity in all relevant dimensions. The original LFVC work [4] presents generalized proofs for rounded timestamps, but it requires uniform routing of all timestamps, which in turn prohibits a constant-time implementation.

Unfortunately, the SEFF policy makes the above considerations somewhat irrelevant. It basically requires keeping flows in a two-dimensional container where they are sorted by both their start *and* finish times. This approach has been chosen for WF^{2}Q+, but inevitably requires a tree-based data structure and consequently $O(\log N)$ access complexity. Alternatively, flows can be kept in two one-dimensional containers, as proposed for LFVC, and exploit the lower access complexity discussed above. However, this two-container solution requires the transfer of all newly eligible flows from one container to the other in between the processing of two consecutive packets. Since all flows may end up with the same or very close start times, this number cannot be limited and effectively results in $O(N)$ worst-case complexity. While the transfer cost could be amortized over the number of packets transferred, this amortization would require $O(N)$ buffer space between scheduler and output link and result in a corresponding scheduling error.

E. Virtual Time

Traditionally, the precise simulation of GPS virtual time has been considered as being an operation with linear complexity, since it needs to keep track of all changes in the set of flows backlogged in the GPS reference system. A recent proposal [23] allows for the exact simulation of GPS virtual time with $O(\log N)$ algorithmic complexity in the number of flows. Independent analysis shows $O(\log N)$ to be a lower bound [24]. Both WF^{2}Q+ and LFVC (along with other algorithms) use a simpler approximation of GPS virtual time. Basically, the approximated virtual time progresses with real time during actual service, that is, it is incremented by the duration of the current packet at each scheduling step. If however the smallest start time of all backlogged flows (which is readily available in WF^{2}Q+ and LFVC) is larger than the current virtual time, the virtual time jumps forward to this minimum start time.

This approximation of GPS virtual time has some negative effects on the scheduling quality of WF^{2}Q+ and LFVC [23]. Under certain circumstances, the approximation of virtual time could lead to unfairness and burstiness being linear in the number of flows. This observation is not a contradiction of the findings for WF^{2}Q+ and LFVC, but results from different assumptions. Normally, GPS emulation schedulers are considered in a context where some kind of delay guarantees are sought. Such delay guarantees can only be given if the sum of rates of all flows does not exceed the link capacity. In terms of the GPS definition, this denotes a situation where the sum of weights is less than or equal to 1. In this case, all previous results about WF^{2}Q+ and LFVC hold, and burstiness is independent of the number of flows. If this restriction is removed, then the observations reported in [23] become an issue. However, it is somewhat questionable whether any application scenario requires the support for sum of weights exceeding 1. Certainly, all scenarios that aim at providing some form of delay guarantee do not qualify. Furthermore, the maximum spread between the highest and lowest service rate is typically limited. For such a scenario, it is possible to find a better scheduling solution with a small and constant execution overhead, as demonstrated in this work.

There is another seeming contradiction between the results reported by Xu and Lipton [25] and the results presented here. This discrepancy is explained by the computational model by Xu and Lipton, which does not allow for the floor or ceiling function to be used. The lower bounds are critically linked to this assumption. In contrast, our algorithm is based on rounded timestamps using the floor function and provides a superior combination of scheduling properties and algorithmic complexity.

F. Low Complexity Implementation

A number of techniques for the efficient implementation of timestamp schedulers are presented and discussed by Stephens et al. [26]. For the particular case of fixed packet sizes in ATM networks, the article presents an implementation of WF^{2}Q+ with constant execution overhead. In the case of variable packet sizes, a different solution is presented, which technically can be regarded as having $O(1)$ complexity in the number of flows, but there are shortcomings. The scheduler implementation uses stratification in the virtual service time of packets to reduce the complexity of the one-container solution proposed for WF^{2}Q+ [19]. This results in a number of stratified groups which is logarithmic to the ratio of the maximum over the minimum supported service rate. For example, if the system were to support service rates between 16 Kbit/s and 40 Gbit/s at packet sizes ranging from 64 to 1500 bytes, this would result in 26 stratified groups. The algorithm then specifies that between each scheduling step, it is necessary to inspect the start and finish times of the front flow in each group to determine the next one to receive service under the SEFF policy. This sequence of comparisons is a nontrivial and costly operation and is hardly possible within the strict timing bounds of high-speed links. Instead, the paper refers to hardware-based timestamp sorting. In other words, this sequence of comparisons introduces too high an absolute constant overhead per scheduling step. Also, the proposal requires additional sorting, since flows change their group association. Finally, the theoretical treatment of rounded timestamps is not as comprehensive as the model and analysis presented in this paper. However, the work by Stephens et al. provides very interesting insight into the implementation details of a packet scheduler at high line rates [26].

The practical implementation of WF^{2}Q+ by Rouskas and Dwekat [27] relies on the fact that certain packet sizes dominate the overall traffic. The existence of other packet sizes is mentioned, but not addressed thoroughly and no quantitative results are given for this case. Also, it seems that the scheduler can only support a fixed set of flow rates, although the details of this are not completely discussed. As such, the proposal presented there is not as general as ours. Furthermore, the algorithm depends on the ability to sort timestamps in constant time in hardware, which is a much stronger requirement than a priority encoder.

G. Round Robin and Hybrid Schedulers

Round robin schedulers take a fundamentally different aim at emulating GPS scheduling. Instead of timestamp computations and sorting, service slots are assigned in some modified round robin fashion. This dramatically reduces the algorithmic complexity of such schedulers, but most early proposals suffer from rather large error terms in their fairness and delay properties. A more recent example, *Smoothed Round Robin* (SRR) [28], uses a fixed weight matrix to achieve very low computational complexity. Its relative fairness only depends on the order of the weight matrix and is thus independent of the number of flows. However, the weight matrix cannot be changed easily and the delay and worst-case fairness of SRR is linear in the number of flows. The G-3 scheduler [29] is an extension of the SRR scheduler [28] to overcome the restrictions of a fixed weight matrix. Recent proposals, such *Stratified Round Robin* (STRR) [30], *Fair Round Robin* (FRR) [31], and *Group Round Robin* (GRR) [32], use flow stratification along service rates and a two-level scheduling hierarchy to solve the problem of dynamically adding and removing flows. The VWQGRR scheduler [33] proposes a new grouping strategy for the GRR scheduler [32] to improve delay bounds, but does not change the fundamental properties.

The critical parameter determining the trade-off between quality and complexity of a round robin scheduler is the size of the quantum used for each scheduling round. Any round robin scheduler has an error term of $\frac{a\times\text{MTU}}{\text{flow rate}}$ due to the minimum quantum. Such a scheduling error poses a problem for low-rate flows with small packet sizes, such as voice. Further, if the quantum is chosen too small, this can lead to multiple processing steps without output (*slip processing*) and thus break $O(1)$ complexity. On the other hand, a larger quantum results in larger error terms.

STRR uses a large quantum and has perfect $O(1)$ complexity. However, the algorithm's general delay bound and worst-case fairness is linear in the number of flows. FRR does not specify a particular quantum, but any quantum that can avoid linear scheduling errors inevitably leads to slip processing in the round robin loop. This effectively breaks $O(1)$ complexity. GRR is a general technique for hierarchical scheduling using round robin as intra-group scheduler. The quantum problem is approached differently than in other hierarchical round robin schedulers. GRR allocates a large quantum to each group, but interleaves group service according to the inter-group scheduler. While this results in the best scheduling properties of all round robin schedulers, it poses the risk of breaking $O(1)$ complexity through slip processing: A flow that becomes non-backlogged cannot be removed from the round robin list immediately. Instead, if it is still non-backlogged during the next round, it is considered departed and effectively removed from the list. However, the number of departed flows may be arbitrarily large in any round and therefore, may result in $O(N)$ processing steps between the processing of two consecutive packets. Derived proposals such as VWQGRR and G-3 improve the respective basic versions, but still suffer from the same general quantum-related problems.

H. Hierarchical Scheduling

Hierarchical scheduling allows for increased control over link sharing and resource allocation. Some of the shortcomings of the early fair-queueing schedulers become very apparent in the context of hierarchical packet scheduling [19]. Hierarchical rate-based scheduling invariably increases the delay bounds for leaf classes and as such, service curve schedulers such as SCED [13] and HFSC [14] are more suitable to achieve both link sharing and tight delay goals at the same time, albeit with increased complexity. For the purpose of this work, we do not argue in favour or against the usefulness of hierarchical scheduling. However, as noted in [19], hierarchical configurations can be regarded as an excellent litmus test for the fairness characteristics of a packet scheduling algorithm. It is strictly for this reason that the simulation experiments use hierarchical scheduling.

III. Model and Analysis

We present a model for a *General Rate-controlled Packet Service* (GRPS) scheduler that uses some form of timestamp rounding to reduce algorithmic complexity and analyze the scheduling properties of the general model. The basic model and Lemmas 1-4 are defined entirely in terms of byte time, independent of the link speed or real time. Therefore, the model applies to fixed and variable bitrate links.

A. System Model

The system model is a generalization of the analytical model used for SI-WF^{2}Q [5], which in turn is based on Bennett and Zhang's work on WF^{2}Q+ [19] as well as Suri et al.'s work on LFVC [4].

A GRPS scheduler system is comprised of a set of flows $i \in \mathcal{F}$. Each flow i is allocated a positive rate fraction r_i with

$$\sum_{i\in\mathcal{F}} r_i \le 1. \tag{1}$$

Flows are assigned start and finish time labels according to the first packet p in their respective queue and the current system virtual time V:

$$S_i = S_i^p = \begin{cases} \max(V, F_i^{p-1}) & \text{arrival to empty queue} \\ F_i^{p-1} & \text{otherwise} \end{cases} \tag{2}$$

$$F_i = F_i^p = \quad S_i^p + \frac{l_i^p}{r_i} \quad \text{for next packet with length } l_i^p \tag{3}$$

We assume the existence of per-flow rounding functions $h_i^S(x)$ for start times and $h_i^F(x)$ for finish times, such that the rounding error is limited. In particular, start times are rounded down and finish times are rounded up. It has been observed before that rounded timestamps work well in this case [4], [26]. The rounding functions are characterized by maximum rounding errors $\overline{S}_i$ and $\overline{F}_i$ as

$$\hat{S}_i = h_i^S(S_i) \text{ with } S_i \ge \hat{S}_i \ge S_i - \overline{S}_i, \text{ and} \tag{4}$$

$$\hat{F}_i = h_i^F(F_i) \text{ with } F_i \le \hat{F}_i \le F_i + \overline{F}_i. \tag{5}$$

We denote with $\mathcal{B}$ the set of backlogged flows. A *busy period* is defined as a period where $\mathcal{B}$ is continuously not empty. In contrast to WFQ or WF^{2}Q, but similar to WF^{2}Q+ or LFVC, virtual time progresses with the real link speed, but it jumps forward when necessary, so that it never falls behind the smallest rounded start time. After serving packet p with size l_p, virtual time increases as

$$V = \max(V + l_p, \min_{i\in\mathcal{B}}(\hat{S}_i)). \tag{6}$$

Backlogged flows are separated into *blocked* and *active* flows, depending on their respective start times in relation to the system virtual time. The following requirements must be satisfied by the scheduling algorithm:

Requirement 1: Flow i blocked $\Rightarrow V \le S_i$.

Requirement 2: Flow i active $\Rightarrow V \ge \hat{S}_i$.

The flow selection policy can be considered as *rounded* SEFF, i.e., among the active flows, the one with the smallest rounded finish time $\hat{F}_i$ is chosen for service. We show that based on these definitions and assumptions, the terms $\overline{S}_i$ and $\overline{F}_i$ are sufficient to characterize the scheduling properties of any specific GRPS instance in relation to WF^{2}Q.

B. Analysis

Our analysis closely follows the proof structure from previous work [5]. However, the analysis is more general and applies to a whole class of schedulers. We have fixed a number of minor inaccuracies and improve the relative fairness bound given before [5].

Lemma 1: For a set of flows $\mathcal{G}$ with $\forall i \in \mathcal{G} : S_i \ge V$, the following inequality holds for all $V' \ge V$:

$$\sum_{i\in\mathcal{G}} l_i + \sum_{i\in\mathcal{G}} (V' - F_i) r_i \le V' - V \tag{7}$$

with l_i being the size of the first packet in flow i's queue.

Proof: By definition, $l_i = (F_i - S_i)r_i$ for all flows and thus, $l_i \le (F_i - V)r_i$ for flows with $S_i \ge V$. Therefore,

$$\sum_{i\in\mathcal{G}} l_i \le \sum_{i\in\mathcal{G}} (F_i - V) r_i \tag{8}$$

$$= \sum_{i\in\mathcal{G}} (V' - V) r_i - \sum_{i\in\mathcal{G}} (V' - F_i) r_i \tag{9}$$

$$\le (V' - V) - \sum_{i\in\mathcal{G}} (V' - F_i) r_i \tag{10}$$

which directly leads to the lemma. The first step uses $F_i - V = V' - V - (V' - F_i)$ and the second step uses $\sum_{i\in\mathcal{G}} r_i \le 1$, based on (1). ■

Lemma 2: At any virtual time V, the *Backlog Inequality* holds for all virtual time values V' with $L + V' \ge V$:

$$\sum_{i:\hat{F}_i \le V'} l_i + \sum_{i:\hat{F}_i \le V'} (V' - F_i) r_i \le L + V' - V \tag{11}$$

with l_i being the size of the first packet in flow i's queue and L being the maximum transmission unit (MTU). If a flow j is not backlogged, $l_j = 0$ and $F_j \ge V$.

Proof: The proof is by induction over those events that change variables. The base case is trivial.

Packet enqueue: Denote the size of new packet p with l_i^p and the finish time after enqueue with F_i^p. If $\hat{F}_i^p > V'$, nothing changes and the lemma holds. Otherwise, since $\hat{F}_i^p \le V'$ after the arrival of the packet, $\hat{F}_i \le V'$ before the packet arrival and the flow is already part of the set. In this case, the first term on the left side of (11) is incremented by l_i^p, but since the flow's finish time F_i is incremented by the virtual packet time $\frac{l_i^p}{r_i}$, the second term left is reduced by l_i^p.

Virtual time jump: After a virtual time jump as in (6), all flows in the system have $S_i \ge \hat{S}_i \ge V$ and Lemma 1 applies for all $V' \ge V$. For V' in $[V - L, V[$, the additional L term in (11) absorbs any decrement on the right hand side. Therefore, the lemma holds.

Packet service: Assume flow j receives service. Denote the size of the current packet for service with l_j, flow rate with

r_j, and rounded finish time with $\hat{F}_j$. We have to distinguish two cases, depending on V' and $\hat{F}_j$.

Case 1: If $V' \geq \hat{F}_j$, then the first term on the left side of (11) is decremented by l_j. Since V is incremented by l_j, the right side of (11) is decremented by the same amount. Therefore, the lemma holds. For the next packet in flow j's queue, the case 'Packet enqueue' applies.

Case 2: If $V' < \hat{F}_j$, then all flows i with $\hat{F}_i \leq V'$ have $\hat{F}_i < \hat{F}_j$. All of these flows must have been blocked, otherwise the current packet would not have been chosen. Therefore, Requirement 1 applies and $S_i \geq V$ holds for all flows i with $\hat{F}_i \leq V'$ before the packet service. Assume the virtual time before service is V_1 and after service V_2. Lemma 1 applies then for all $V' \geq V_1$, i.e.

$$\sum_{i:\hat{F}_i \leq V'} l_i + \sum_{i:\hat{F}_i \leq V'} (V' - F_i) r_i \leq V' - V_1. \tag{12}$$

Because $V_2 = V_1 + l_j$ and we assume $L + V' \geq V_2$ after service, we only need to consider V' with $L + V' \geq V_1 + l_j$ before service. Therefore

$$V' - V_1 \leq (L - l_j) + V' - (V_2 - l_j) = L + V' - V_2 \tag{13}$$

and the lemma holds after service. ■

The *Service Time* lemma establishes that a packet is served when the virtual time reaches its rounded finish time with an error term of at most one maximum packet size L in addition to the rounding error.

Lemma 3: For any backlogged flow i

$$V \leq F_i + \overline{F}_i + L. \tag{14}$$

Proof: We prove $V \leq \hat{F}_i + L$, which directly implies the lemma. The proof is by contradiction. The only event that could lead to a violation of the assumption is serving a packet during a busy period. Assume that at V_1 the lemma holds. A packet p with rounded finish time $\hat{F}_1$ and length l_p is served and afterwards at V_2, there is a packet q with finish time F_2, such that $\hat{F}_2 + L < V_2$. Denote with S_1 and S_2 the corresponding start times. We need to distinguish three cases.

Case 1: Packet q is active at V_1. Then, $\hat{F}_2 \geq \hat{F}_1$ (both packets were eligible at V_1 and p was chosen). Applying Lemma 2 with $V = V_1$ and $V' = \hat{F}_2$ results in

$$\sum_{i:\hat{F}_i \leq \hat{F}_2} l_i + \sum_{i:\hat{F}_i \leq \hat{F}_2} (\hat{F}_2 - F_i) r_i \leq L + \hat{F}_2 - V_1 \tag{15}$$

Because $F_i \leq \hat{F}_i$, the second term on the left side of the inequality is non-negative and therefore

$$l_p \leq \sum_{i:\hat{F}_i \leq \hat{F}_2} l_i \leq L + \hat{F}_2 - V_1 \tag{16}$$

$$V_2 - V_1 \leq L + \hat{F}_2 - V_1 \tag{17}$$

$$V_2 \leq \hat{F}_2 + L \tag{18}$$

The step from (16) to (17) uses $V_1 + l_p = V_2$.

Case 2: Packet q is not active at V_1, but becomes active between V_1 and V_2. Then, $\hat{S}_2 \geq V_1$. Virtual time advances by at most L and therefore:

$$\hat{F}_2 \geq \hat{S}_2 \geq V_1 \geq V_2 - L \tag{19}$$

Case 3: Packet q is not active after service to p, therefore V_2 is reached by a virtual time jump before q can be served. In this case:

$$\hat{F}_2 \geq \hat{S}_2 \geq V_2 \geq V_2 - L \tag{20}$$

This concludes the proof. ■

Finally, we need to establish a bound between virtual time V and real time R, assuming a fixed link capacity.

Lemma 4: Let I be the last time when the system was idle and let J be the amount of virtual time "jumping" that has been done during the current busy period. Then, whenever a packet has completed service:

$$V + I - J = R. \tag{21}$$

Proof: At the beginning of a busy period, $I = R$ and $V = J = 0$. Whenever a packet is chosen for service and transmitted, V and R increase by the same amount. When virtual time jumps forward, V and J increase by the same amount. ■

Using the above lemmas, we can now prove the main theorems describing the service characteristics of GRPS schedulers in general.

Theorem 1: (End-to-End Delay) GRPS is a *Guaranteed Rate* (GR) scheduler [7] with an error term β_i for flow i as

$$\beta_i \leq L + \overline{F}_i \tag{22}$$

Proof: Let p_i^j and l_i^j denote the jth packet of flow i and its size. Let $A_R(p_i^j)$ denote the real-time arrival time of packet p_i^j. The guaranteed rate clock values are defined as is [7]:

$$GRC(p_i^j) = \max(A_R(p_i^j), GRC(p_i^{j-1})) + \frac{l_i^j}{r_i} \tag{23}$$

with $GRC(p_i^0) = 0$.

Denote with F_i^j the finish time of the jth packet. Let $I(p_i^j)$ and $J(p_i^j)$, respectively, be the values of I and J from Lemma 4 when service of p_i^j is completed. Let $I'(p_i^j)$ and $J'(p_i^j)$, respectively, be the values of I and J when p_i^j arrives at the head of its queue. We first prove that during a busy period

$$F_i^j + I(p_i^j) - J(p_i^j) \leq GRC(p_i^j) \tag{24}$$

We prove (24) by induction on j. The base case is trivial. The virtual time of a packet arrival is denoted by $A(p_i^j)$.

Inductive Step: Assume (24) holds for $j - 1$. We need to distinguish two cases.

Case 1: $A(p_i^j) > F_i^{j-1}$. Then

$$F_i^j = A(p_i^j) + \frac{l_i^j}{r_i}. \tag{25}$$

Using Lemma 4 we can characterize the packet arrival time as

$$A(p_i^j) + I'(p_i^j) - J'(p_i^j) \leq A_R(p_i^j) \tag{26}$$

Adding $\frac{l_i^j}{r_i}$ results in

$$A(p_i^j) + \frac{l_i^j}{r_i} + I'(p_i^j) - J'(p_i^j) \leq A_R(p_i^j) + \frac{l_i^j}{r_i}. \tag{27}$$

Rearranging the terms using (25), and replacing A_R by a trivial maximum expansion, results in

$$F_i^j + I'(p_i^j) - J'(p_i^j) \leq \max(A_R(p_i^j), GRC(p_i^{j-1})) + \frac{l_i^j}{r_i}. \quad (28)$$

The right hand side shows the definition of GRC. Since we are considering a busy period, $I'(p_i^j) = I(p_i^j)$ and $J'(p_i^j) \leq J(p_i^j)$. Therefore

$$F_i^j + I(p_i^j) - J(p_i^j) \leq GRC(p_i^j), \quad (29)$$

which establishes (24) for Case 1.

Case 2: $A(p_i^j) \leq F_i^{j-1}$. In this case, $F_i^j = F_i^{j-1} + \frac{l_i^j}{r_i}$. Through the induction hypothesis, we obtain

$$F_i^{j-1} + I(p_i^{j-1}) - J(p_i^{j-1}) \leq GRC(p_i^{j-1}) \quad (30)$$

Adding $\frac{l_i^j}{r_i}$ and replacing GRC by a trivial maximum expansion results in

$$F_i^j + I(p_i^{j-1}) - J(p_i^{j-1}) \leq \max(A_R(p_i^j), GRC(p_i^{j-1})) + \frac{l_i^j}{r_i}. \quad (31)$$

The right hand side shows the definition of GRC. Since we are considering a busy period, $I(p_i^{j-1}) = I(p_i^j)$ and $J(p_i^{j-1}) \leq J(p_i^j)$. Therefore

$$F_i^j + I(p_i^j) - J(p_i^j) \leq GRC(p_i^j) \quad (32)$$

which establishes (24) for Case 2.

We can now prove the theorem. A packet p_i^j is served no later than $F_i^j + \overline{F}_i + L$ (Lemma 3). At the end of transmission, the real time equals $F_i^j + \overline{F}_i + L + I(p_i^j) - J(p_i^j)$ by Lemma 4. By (24), this is bound by $GRC(p_i^j) + \overline{F}_i + L$. ■

Theorem 2: (Relative Fairness) The relative fairness [10] of GRPS between any two flows i and j is bound by $\theta_{i,j}$ with

$$\theta_{i,j} \leq 2L + \sum_{x=i,j} \left(\frac{L_x}{r_x} + \overline{S}_x + \overline{F}_x\right). \quad (33)$$

with L_x being the maximum packet size for flow x.

Proof: We can determine the earliest and latest possible finish times for a packet from flow i that receives service at a virtual time V as follows:

$$F \geq F^{\min} = V - \overline{F}_i - L \quad (34)$$

$$F \leq F^{\max} = V + \overline{S}_i + \frac{L_i}{r_i} \quad (35)$$

$F^{\min}$ follows from Lemma 3 and $F^{\max}$ follows from Requirement 2. Consequently, for any interval $[V_1, V_2]$ during which a flow is backlogged its maximum service is bound by $V_2 - V_1 + F^{\max} - F^{\min}$, while the minimum service is bound by $V_2 - V_1 + F^{\min} - F^{\max}$. The maximum deviation between two backlogged flows i and j can be computed by subtracting both terms from each other:

$$(F_i^{\max} - F_i^{\min}) - (F_j^{\min} - F_j^{\max}) \quad (36)$$

Inserting (34) and (35) into (36) gives the lemma. ■

Theorem 3: (Worst-case Fairness) Let $Q_i(p)$ be the backlog of an arbitrary flow i immediately after packet p arrives. The time δ to clear this backlog is bound as

$$\delta_i \leq \frac{Q_i}{r_i} + L + \frac{L_i}{r_i} + \overline{S}_i + \overline{F}_i. \quad (37)$$

Proof: Consider a packet p for flow i. Suppose that immediately after p arrives, the packet at the head of flow i's queue is p_i^j. Further assume that there are $m \geq 0$ packets in the queue in front of p, meaning $p = p_i^{j+m}$. Let R_1 denote the real arrival time and R_2 the real time when p's service is complete. Let V_1, V_2 denote the virtual times corresponding to the real times R_1 and R_2, and F_1, F_2 the flow's finish times at R_1 and R_2, respectively. (35) guarantees the following inequality:

$$F_1 \leq V_1 + \frac{L_i}{r_i} + \overline{S}_i \quad (38)$$

or

$$-V_1 \leq -F_1 + \frac{L_i}{r_i} + \overline{S}_i. \quad (39)$$

Lemma 3 gives the bound $V_2 \leq F_2 + \overline{F}_i + L$. Subtracting V_1 from V_2, i.e. adding (39) to the bound for V_2, results in

$$V_2 - V_1 \leq F_2 - F_1 + L + \frac{L_i}{r_i} + \overline{S}_i + \overline{F}_i. \quad (40)$$

Since flow i is backlogged during the interval $[F_1, F_2]$, we get $F_2 = F_1 + \sum_{n=1}^{m} \frac{l_i^{j+n}}{r_i}$, which can be inserted into (40):

$$V_2 - V_1 \leq \sum_{n=1}^{m} \frac{l_i^{j+n}}{r_i} + L + \frac{L_i}{r_i} + \overline{S}_i + \overline{F}_i. \quad (41)$$

The queue size $Q_i(p)$ satisfies

$$\sum_{n=1}^{m} \frac{l_i^{j+n}}{r_i} \leq Q_i(p). \quad (42)$$

Plugging this into (41) yields

$$V_2 - V_1 \leq \frac{Q_i(p)}{r_i} + L + \frac{L_i}{r_i} + \overline{S}_i + \overline{F}_i. \quad (43)$$

Using the bounds between virtual time and real time from Lemma 4, we can obtain a lower bound for R_1 and an upper bound for R_2:

$$V_1 + I - J_1 \leq R_1 \text{ and } V_2 + I - J_2 \geq R_2 \quad (44)$$

with $J_2 \geq J_1$. Therefore

$$\begin{aligned} R_2 - R_1 &\leq V_2 + I - J_2 - (V_1 + I - J_1) && (45)\\ &= V_2 - V_1 - (J_2 - J_1) && (46)\\ &\leq \frac{Q_i(p)}{r_i} + L + \frac{L_i}{r_i} + \overline{S}_i + \overline{F}_i - (J_2 - J_1). && (47) \end{aligned}$$

Since $J_2 - J_1 \geq 0$, this concludes the proof. ■

Note that the relative fairness bound between two flows is tighter than reported earlier [5]. It is the sum of error terms of the absolute fairness bounds of both flows. This is consistent with intuition and previous findings [34]. In comparison to the properties of WF^{2}Q [19], the additional error terms can be directly related to the timestamp rounding errors $\overline{S}_i$ and $\overline{F}_i$.

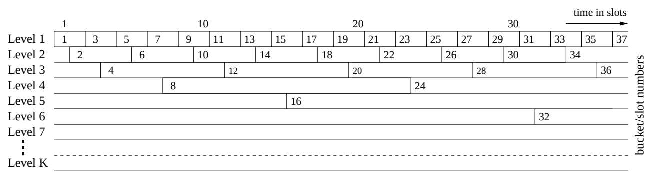

Fig. 1. Interleaved Stratified Timer Wheels (ISTW)

IV. Interleaved Stratified Timer Wheels

The key challenge resulting from the SEFF policy (cf. Section II-A) is that the next flow for service is chosen from the set of backlogged flows based on two criteria at the same time: the smallest finish time among those flows with a start time less or equal the scheduler's current virtual time. For WF^{2}Q+ [19], it is proposed to use a tree-based data structure to accommodate both criteria, while LFVC [4] has introduced the idea of two containers to hold blocked vs. active flows. Any tree-based data structure can only be maintained with logarithmic complexity, while the two-container solution suffers from a transfer problem, if the system virtual time (6) crosses the start time threshold of many flows in one step.

We propose *Interleaved Stratified Timer Wheels* (ISTW) as a priority queue that supports efficient searching for the nearest future event and controlled scanning for past events, if necessary. For rate-proportional scheduling, one ISTW container is used to sort finish times and search for the next flow to service. To facilitate the SEFF policy, a second ISTW instance keeps flows sorted by start times. Besides searching for the next start time, if necessary for (6), ISTW supports scanning for past start times with a meaningful bound for lateness, but without the need to transfer many flows at the same time.

An ISTW container is a collection of timer wheels where time is measured in fixed base time slots and the bucket width is doubled for each level. The top-most wheel in Level 1 has a bucket width of two base time slots. Furthermore, the timer wheels are interleaved, such that buckets are never aligned with each other across levels. This is shown in Figure 1. Each bucket can be identified by the number of the first time slot that it covers, so that the corresponding rounded slot or bucket number for a slot j in level $k, k \geq 1$, can be computed as

$$h_k(j) = 2^k \left\lfloor \frac{j - 2^{k-1}}{2^k} \right\rfloor + 2^{k-1}. \tag{48}$$

We observe that the rounding error is limited by

$$j < h_k(j) + 2^k. \tag{49}$$

The *find-first-set* (ffs) operation can be used to determine the level of a given bucket number. The ffs operation finds the position of the least significant bit in a word. It can be implemented in software at logarithmic cost of the word length [35]. Further, a priority encoder can be implemented in hardware at a very low cycle cost. For example, recent Intel processors implement the ffs operation (termed BFS) at 1-3 clock cycles at Gigahertz clock rates, depending on the architecture [36], while the IXP network processor provides a one-cycle ffs instruction [37].

```
function ISTW::insert( slot , elem ) {
  k = ffs(slot);
  levelCount[k-1] += 1;
  setbit(levelbits , k);
  getBucket(slot).push_back(elem);
}
```

Fig. 2. ISTW Insert Operation

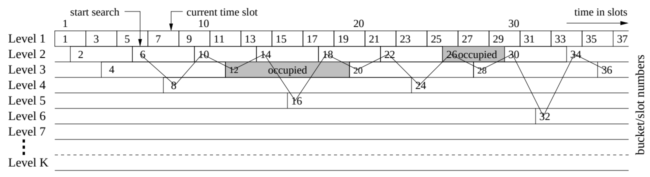

Fig. 3. Zigzag Search in ISTW

An ISTW container maintains a bitmask `levelbits` to indicate whether any bucket in a particular level is occupied. The `insert` operation is shown in Figure 2. It assumes a function `getBucket` that retrieves the bucket corresponding to a given rounded time slot number.

Searching for the next occupied slot in an ISTW container works by first locating the smallest occupied level number in the ISTW container by using the ffs operation on `levelbits` and then performing a linear search of buckets. Because the timer wheels are interleaved, iterating through bucket numbers with a step width of half the bucket size of the smallest occupied level results in a search pattern where the smallest occupied level is searched linearly, but every second search step considers a bucket from a bigger numbered level. This search pattern ensures that any bigger-level bucket along the way is visited as well. In Figure 3, the complete search path is illustrated, although in this example the search would terminate at Bucket 12. Assuming a function `roundSlot` that implements h_k from (48), the `search` function is described by the pseudo-code in Figure 4. It is easy to see that the worst-case execution complexity of this linear search is proportional to the number of buckets between the current time and the first element in the smallest occupied level.

We define the *current bucket* for each level as the bucket that overlaps with the current time slot. The `scan` operation for an ISTW container retrieves an element from the smallest occupied level that has an element in its current bucket. This is done by maintaining a bitmask `frontbits` that indicates the occupancy of the current buckets for all levels. The pseudo-code is shown in Figure 5.

Similar data structures for sorting and searching have been used for other schedulers, such as SRR [28], STRR [30], or GRR [32], all of which are discussed in Section II. However, since all these proposals follow the round robin approach, they do not consider a comprehensive combination of searching and scanning on both start and finish times. In particular, interleaving the timer wheels is an essential feature in ISTW for maintaining `frontbits`, as discussed in SectionV-B.

```
function ISTW::search( slot ) {
  if (!levelbits) return NULL;
  k = ffs(levelbits);
  slot = roundSlot(slot,k);
  step = (1 << (k-1));
  while ( getBucket(slot).empty() ) {
    slot += step;
    k = ffs(slot);
  }
  levelCount[k-1] -= 1;
  if ( levelCount[k-1] == 0 )
    clearbit(levelbits,k);
  return getBucket(slot).pop_front();
}
```

Fig. 4. ISTW Search Operation

```
function ISTW::scan( slot ) {
  k = ffs(slot);
  if ( !getBucket(slot).empty() )
    setbit(frontbits,k);
  if ( !frontbits )
    return search(slot);
  k = ffs(frontbits);
  if ( getBucket(slot).size() == 1 )
    clearbit(frontbits,k);
  levelCount[k-1] -= 1;
  if ( levelCount[k-1] == 0 )
    clearbit(levelbits,k);
  return getBucket(slot).pop_front();
}
```

Fig. 5. ISTW Scan Operation

```
function enqueue( pkt ) {
  flow = classify(pkt);
  flow.Q.push_back( pkt );
  if ( flow.Q.size() == 1 ) {
    if ( flow.S < V ) flow.S = V;
    else if (!Active.levelbits)
      V = flow.S;
    insertFlow( flow );
  }
}
```

Fig. 6. Scheduler Enqueue Operation

```
function dequeue() {
  Vf = Vs = V/g;
  if ( mainQ.empty() ) }
    if ( !Active.levelbits ) return;
    flow = Active.search(Vs);
  } else {
    flow = mainQ.pop_front();
  }
  pkt = flow.Q.pop_front();
  transmit( pkt ); // in the background
  V += pkt.size;
  flow.S = flow.F;
  if (!flow.Q.empty()) insertFlow(flow);
  transfer();
  for ( ; Vf < V/g; Vf += 1 ) {
    mainQ.append( Active.getBucket(Vf) );
  }
}
```

Fig. 7. Scheduler Dequeue Operation

V. Scheduler Design

A. Basic Operations

SI-WF^{2}Q, first described in [5], uses the ISTW data structure to implement a two-container solution for rate-based scheduling. We first show the pseudo-code for the `enqueue` and `dequeue` operations in here, before discussing its execution complexity and analyzing its scheduling properties using the general model from Section III. The two containers are termed `Active` and `Blocked` with obvious semantics. We assume a function `insertFlow` that computes the finish time `F` according to (3), rounded timestamps `Sx` and `Fx`, and decides whether to insert a new or returning flow into the `Blocked` or the `Active` container. `V` denotes the system virtual time.

The `enqueue` operation, shown in Figure 6 reconciles the flow's start time with the system virtual time, before inserting a new flow. If a flow's start time is "old", Line 6 implements the `max` function from (2). Otherwise, if the system is empty (Line 5), the virtual time is immediately set to the flow's start time according to (6), which effectively resets the system.

The `dequeue` operation, shown in Figure 7 chooses the next flow for service and updates state variables as needed. Flows are not necessarily taken from the `Active` container directly, but instead for each increase in virtual time, the `Active` container is scanned for flows that have an "expired" finish time and these flows are moved to a central service queue `mainQ`. This is a safe operation, since any packets that arrive later will have their finish times set to a value greater than the system virtual time. Only if scanning comes up empty, i.e., the next available finish time is greater or equal than the current virtual time, the `Active` container is searched. This procedure is further discussed in the next section.

The `transfer` operation, shown in Figure 8 implements the transfer of flows that become eligible during the next packet's transmission, as well as a potential jump in virtual time corresponding to (6). Note that virtual time only jumps, if the `Active` container is empty, therefore it does not interfere with the scanning loop at the end of `dequeue`.

These operations using two ISTW containers collectively implement the system model from Section III. The time period represented by one slot is denoted by the constant `g`. To analyze the scheduling properties, compliance with Requirements 1 and 2 needs to be shown. However, we first discuss the execution complexity of the operations.

```
function transfer() {
  while ( Vs < V/g && !Blocked.empty() ) {
    if ( !Active.levelbits &&
         !Blocked.frontbits ) {
      tmp = Blocked.search( Vs );
      Vs = tmp.Sx;
    } else {
      tmp = Blocked.scan( Vs );
      Vs += 1;
    }
    if (tmp) Active.insert( tmp.Fx, tmp );
  }
  if ( Vs > V/g ) V = Vs * g;
}
```

Fig. 8. Scheduler Transfer Operation

B. Execution Complexity

The scheduler uses stratified rounded timestamps in combination with the ISTW data structure to keep the execution complexity low. Stratification groups "similar" flows based on their service rates and potentially other characteristics. The execution complexity of rate-based packet schedulers is typically characterized in relation to the number of flows in the system, which implicitly assumes that per-packet operations have a small constant execution overhead.

However, in certain existing schedulers, the per-packet operations do not have truly constant per-packet overhead, but occasionally need to execute loops with the worst-case number of iterations being on the order of the number of flows in the system. Since these occurrences are rare enough, their cost is amortized over time and termed *amortized constant overhead*. However, given the tight timing requirements with which packets have to be released to a high-speed output link, it is not possible to ensure that the output link is always fully utilized when the runtime of the per-packet operations cannot be tightly controlled. Examples of schedulers with this property are LFVC and those round robin schedulers that are subject to slip processing, as discussed in Section II.

In this work, we use a slightly different notion of "constant complexity", which works well for packet schedulers. A packet scheduler typically needs to be designed for a worst-case traffic mix of only minimum-sized packets. If the cost of processing a packet is proportional to the size of the packet, then the increased cost for a larger packet is directly offset by the fact that a larger packet keeps the link busy for a longer time. Thereby, only the amortized cost for processing a packet is constant, but in contrast to long-term amortization as discussed above, the amortization period is only one packet. We denote this as *packet-amortized constant complexity*. In particular, for the schedulers presented below we propose to set the length of a base time slot in the ISTW container to the minimum supported packet size. We then only need to consider the loops for assessing the execution complexity.

The flow transfer loop in `transfer` executes in proportion to the increase in virtual time caused by the packet that has just been chosen for service and thus, clearly satisfies the requirements of packet-amortized constant complexity. This operation is the essential motivation for interleaving the timer wheels in ISTW. Only because of interleaving it is possible to maintain `frontbits` smoothly, i.e., for each slot exactly one bucket in one level needs to be checked.

The overhead of the linear search in an ISTW container depends on the distance in buckets between the starting point and the timestamp found (cf. Section IV). In case of the `Active` container, Equations (34) and (35) limit the range of rounded finish times at any virtual time V to $[V-L, V+\overline{S}_i+\overline{F}_i+\frac{L_i}{r_i}]$ for any flow i. Assuming that $\overline{S}_i$ and $\overline{F}_i$ are constant and not significantly larger than $\frac{L_i}{r_i}$, this directly translates into packet-amortized constant overhead, because the search cost is immediately amortized by transmitting the packet. The only caveat is that the search has to start at $V-L$, and L may be relatively large compared to the other parameters, especially when comparing to small packets from high rate flows. Therefore, the `dequeue` operation is enhanced by scanning the `Active` container for "expired" finish times that are smaller than the current virtual time. The overhead of the scanning loop at the end of `dequeue` is proportional to the size of the packet being transmitted. This guarantees that a search in `Active`, if necessary, can start at the virtual time.

Similar reasoning can be used for the `Blocked` container. The minimum start time is given by Requirement 1 and the maximum start time follows from (35) by assuming that a flow is reinserted into the blocked container immediately after packet service. Thereby, the range of rounded start times in `Blocked` is limited to $[V-\overline{S}_i, V+\frac{L_i}{r_i}]$. The part $[V-\overline{S}_i, V]$ is covered by scanning and transfer, so that `Blocked` only needs to be searched, if the next start time is greater or equal than the current virtual time. However, there is a key difference to the finish time search discussed above. The linear overhead of the search (from the $\frac{L_i}{r_i}$ component) corresponds to the *previous* packet of flow i, rather than the next one. The previous packet that causes a search might already be transmitted and thus, search overhead and packet transmission cannot be directly related as before. However, the search overhead per flow is limited to the equivalent of $\frac{L_i}{r_i}$ in buckets and clearly, multiple flows in the same level only increase the density of timestamps and do not add up to a higher search overhead. Since the bucket size is proportional to $\frac{1}{r_i}$, an overall amortization buffer of $nL_{\max}$, for n levels and an MTU size of $L_{\max}$, is sufficient to absorb this worst-case effect of the start search overhead.

C. Scheduling Properties

Requirement 2 is satisfied, because the `dequeue` operation only transfers flows with a rounded start time greater or equal the virtual time through `scan`, or sets the virtual time to a flow's start time when using `search`. We denote the time slot period with λ and show that Requirement 1 is satisfied, if the following two conditions are met: For any flow i in level k

$$r_i > \frac{1}{2^k}, \text{ and} \tag{50}$$

$$\overline{S}_i \geq 2^k\lambda \tag{51}$$

We show that during any time period without virtual time jumps, each bucket in `Blocked` is cleared out, before the virtual time reaches the next bucket in the same level. This establishes Requirement 1. A bucket in `Blocked` in level k covers 2^k time slots. Each flow only occurs once in each bucket of its level in `Blocked`, because if the flow's start time after service falls in the same bucket as before, the flow stays in `Active`. Also, each bucket stays as the front bucket (indexed by `frontbits`) for a duration of 2^k time slots. Every slot in the `Blocked` container marks the beginning of new bucket, which may hold several flows. For a bucket in level k, the system has 2^k time slots to clear the bucket. If we denote the set of flows that are stored in bucket x with E_x, the number of elements in the set with $\parallel E_x \parallel$, the set of flows in level k with u_k, and the set of flows in levels $1..k$ with U_k, we can show by induction over k that for any period of 2^k slots, the following equation holds.

$$\sum_{x=v}^{v+2^k-1} \parallel E_x \parallel \leq 2^k \sum_{i\in U_k} r_i + \parallel E_z \parallel \text{ with } z > k. \tag{52}$$

The base case ($k = 1$) is trivial. One of two slots belongs to level 1 and might contain one flow i with $0.5 < r_i \leq 1$.

(Inductive Step) Assume (52) holds for k. Then:

$$\sum_{x=v}^{v+2^{k+1}-1} \parallel E_x \parallel \tag{53}$$

$$= \sum_{x=v}^{v+2^k-1} \parallel E_x \parallel + \sum_{x=v+2^k}^{v+2^{k+1}-1} \parallel E_x \parallel \tag{54}$$

$$\leq 2^k \sum_{i\in U_k} r_i + \parallel E_{z_1} \parallel + 2^k \sum_{i\in U_k} r_i + \parallel E_{z_2} \parallel \tag{55}$$

$$= 2^{k+1} \sum_{i\in U_k} r_i + \parallel E_{z_1} \parallel + \parallel E_{z_2} \parallel \text{ with } z_1, z_2 > k \tag{56}$$

$$\leq 2^{k+1} \sum_{i\in U_{k+1}} r_i + \parallel E_{z_2} \parallel \text{ with } z_2 > k \tag{57}$$

The first step splits up the time period of length 2^{k+1} into two periods of length 2^k to prepare the inductive step. The second step applies the induction hypothesis. The next step rearranges the terms and the last step uses the following consideration: One of the buckets E_{z_1} and E_{z_2} belongs to level $k+1$. We assume without loss of generality E_{z_1}. Condition (50) implies that $\parallel E_{z_1} \parallel < 2^{k+1} \sum_{i\in u_{k+1}} r_i$. Further, $U_{k+1} = U_k \cup u_{k+1}$, and therefore, (56) can be transformed into (57), which confirms the hypothesis. In combination with (51), this establishes Requirement 1.

D. SI-WF^{2}Q

SI-WF^{2}Q [5] implements the basic scheduler design with the following rounding functions. Its scheduling error terms are very small, but constant complexity requires an amortization buffer as discussed in Section V-B. Since it has been analyzed in [5], we omit the details here.

$$h_i^S(S) = h_k(\frac{S}{\lambda} - 2^k)\lambda \text{ with } k = \lfloor \log_2(\frac{1}{r_i}) \rfloor + 1 \tag{58}$$

$$h_i^F(F) = h_k(\frac{S}{\lambda} + 2^k)\lambda \text{ with } k = \lfloor \log_2(\frac{1}{r_i}) \rfloor + 1 \tag{59}$$

E. The K Packet Scheduler (KPS)

As discussed in Section V-B, the start time search in the basic scheduler design incurs an execution overhead proportional to $\frac{L_i}{r_i}$. The key idea to reduce the impact of the $\frac{L_i}{r_i}$ term is to stratify not only based on a flow's service rate as in SI-WF^{2}Q, but to also include the packet length to even out the virtual duration of each packet in a stratified level. In this new proposal the `Blocked` container determines a flow's stratified service level based on the maximum virtual packet size, rather than just the service rate. The rounding function for start times in the `Blocked` container is defined as

$$h_i^S(S) = h_{k'}(\frac{S}{\lambda} - 2^{k'})\lambda \text{ with } k' = \lfloor \log_2(\frac{L_i}{\lambda r_i}) \rfloor + 1 \tag{60}$$

where L_i is the maximum packet size of flow i. The rounding function for finish times is identical to (59). We observe for start times that

$$\frac{L_i}{r_i} \leq 2^{k'}\lambda < \frac{2L_i}{r_i}. \tag{61}$$

Intuitively, the new rounding function in (60) addresses the start search problem by demoting flows with larger packets to lower levels. This effectively reduces the number of buckets to search per level to one, regardless of the actual packet size, and thus avoids any linear search. On the downside, it also slightly increases the fairness bounds. We summarize the characteristics of KPS in the following theorem.

Theorem 4: KPS is a GRPS scheduler with

$$2^{k'}\lambda \leq \overline{S}_i < 2^{k'+1}\lambda < \frac{4L_i}{r_i} \tag{62}$$

$$\overline{F}_i < 2^k\lambda < \frac{2\lambda}{r_i} \tag{63}$$

$$\hat{S}_i \leq V + 2^{k'}\lambda \tag{64}$$

Proof: Equation (49) in combination with (60) and (59) direct leads to (62) and (63), respectively. Since (50) is satisfied by (60) and (59), and because the left part of (62) is identical to (51), KPS is a GRPS scheduler.

For (64), we observe that the highest possible start time of a flow in the `Blocked` container is immediately after its service. Requirement 2 states that for an active flow i, $\hat{S}_i < V$. Since (61) shows that service increases $\hat{S}_i$ by at most $2^{k'}\lambda$, (64) follows directly. ■

Therefore, (62) and (63) characterize the scheduling properties. Finally, (64) shows that rounded start times are stored at most one bucket behind the current bucket. Therefore, the start time search only incurs a small constant overhead.

Compared to SI-WF^{2}Q, KPS has slightly increased error terms for relative and absolute fairness. However, most importantly, all flow-rate dependent error terms only depend on each flow's respective packet sizes and all error terms have only small coefficients. This is the typical litmus test for a high-quality GPS approximation. However, strictly speaking, KPS still has packet-amortized constant overhead, since the `transfer` function must be executed while a packet is being transmitted and its overhead is proportional to the packet size.

F. Simple KPS

In this simpler variant of the scheduler, we apply the new stratification method to the `Active` container, as well. This also reduces the `Active` container to significantly fewer buckets, which reduces the overall memory footprint and execution overhead. The start container remains the same as in KPS. The rounding function for start times is identical to (60) and for finish times it is

$$h_i^F(F) = h_k(\frac{F}{\lambda} + 2^k)\lambda \text{ with } k = \lfloor \log_2(\frac{L_i}{\lambda r_i}) \rfloor + 1 \quad (65)$$

Theorem 5: Simple KPS is a GRPS scheduler with (62) and (64), as well as

$$\overline{F}_i < 2^k \lambda < \frac{2L_i}{r_i} \quad (66)$$

$$\hat{F}_i < V + 4 \cdot 2^k \lambda \quad (67)$$

Proof: The considerations are the same as for KPS, except for (67): Equation (35) states that $F_i \leq V + \overline{S}_i + \frac{L_i}{r_i}$. Since $\overline{S}_i < 2^{k+1}\lambda$ from (62), $\frac{L_i}{r_i} \leq 2^k\lambda$ from (61), and $\hat{F}_i \leq F_i + 2^k\lambda$ from (5) and (66), (67) follows directly. ■

While the delay error term for Simple KPS is worse than for KPS, all flow-rate dependent error terms still only depend on each flow's packet size and all error terms have only small coefficients. The asymptotic execution overhead is the same as for KPS, but (67) shows that the absolute memory footprint and search overhead for `Active` is significantly reduced, since `Active` can now be implemented with only five buckets in each level, regardless of packet sizes.

VI. SIMULATIONS

We use a modified simulation setup from [12] to illustrate the worst-case fairness of the schedulers proposed in this paper in comparison to WF^{2}Q+ and a pure finish-time scheduler. A hierarchical scheduling setup is an ideal test to expose worst-cast fairness characteristics of a scheduler. We use WF^{2}Q+ as a benchmark with known excellent scheduling properties, as well as SPFQ [22] as a scheduler that lacks proper mechanisms to guarantee low worst-case fairness.

The simulation setup uses a dumbbell topology with a single bottleneck and multiple sender and receiver nodes. The bottleneck link is configured with a hierarchy of service classes as shown in Figure 9. The figure also shows the actual sending rate per service class. For each leaf class, there is one sender sending at the specified rate. The CS senders send at a constant bitrate, while the PS sources send Poisson traffic with an on-period of 75ms and an off-period of 25ms. The BE-1 service class is always backlogged and the RT-1 sender sends CBR traffic with a pattern of 75ms on-period and 25ms off period. All packet sizes are set to 8000 bytes. This represents the scenario “with correlated background traffic” from [12], since it provides the most challenging environment.

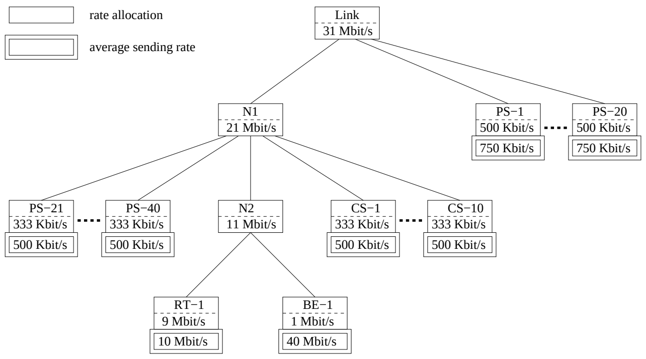

Fig. 9. Scheduling Hierarchy

As mentioned before, hierarchical scheduling is an excellent litmus test to verify and illustrate the worst-case fairness properties of a packet scheduler. In contrast to the experiments in [19], we only compare to the non-shaped *Starting Potential Fair Queueing* (SPFQ) scheduler from [22]. Its worst-case fairness properties are comparable to the WFQ, SCFQ, and SFQ schedulers chosen in [19].

The relevant observation parameters for these experiments are the queueing delay variations experienced by the RT-1 traffic during the experiment. The results are shown in Figure 10 as the worst-case delay measurement over intervals of 50ms. For illustrative purposes, the SI-WF^{2}Q delay is shown excluding the additional amortization buffer. Similar to the original results in [19], a non-SEFF scheduler causes significant delay variations for the real-time traffic class. The superior worst-case fairness of WF^{2}Q+ results in significantly reduced delay fluctuations, independent of the type of cross traffic. Omitted here, but shown in [5], SI-WF^{2}Q provides basically the same service as WF^{2}Q+. As predicted by the analysis, KPS and Simple KPS deliver a worst-case fairness that is almost as good as that of WF^{2}Q+ and SI-WF^{2}Q, despite their drastically reduced execution overhead.

VII. DISCUSSION

This work introduces the concept of packet-amortized constant complexity. Execution complexity is not defined in relation to an arbitrarily sized unit of work. Instead, as long as the overhead is strictly proportional to the size of the unit of work (a packet in this case), it is considered constant. Note that the basic complexity consideration is independent of whether the actual execution speed on any particular platform is fast enough to achieve line speed.

SI-WF^{2}Q and the other schedulers proposed in this paper offer superior properties compared to all other known packet schedulers. Their main benefits are:

- The KPS scheduling algorithm can be executed with true packet-amortized constant complexity. This is the key improvement over SI-WF^{2}Q and WF^{2}Q+. For Simple KPS, the execution overhead is even smaller.
- The absolute execution overhead and memory footprint is small. At the core of the algorithm, the `search` and `scan` operations only access ISTW bitmaps frequently.

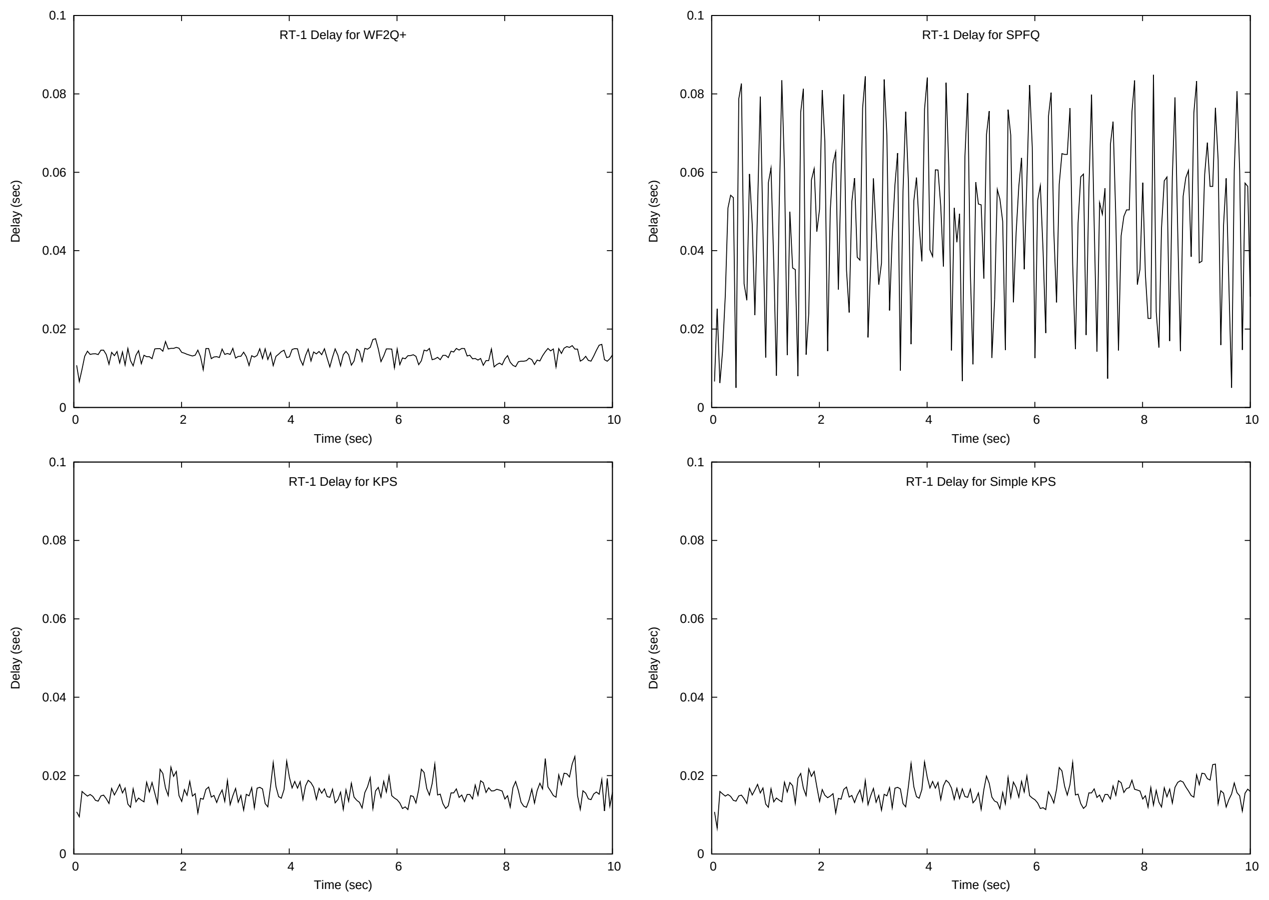

Fig. 10. RT-1 Delay with Different Schedulers

- Interleaving and stratification of timer wheels allows for controlled but timely processing of timers. This is the key improvement over LFVC.
- The schedulers have constant small scheduling errors, only depending on per-flow characteristics and constants depending on the link speed. In particular, there is no MTU error term of $L_{\max}/r_i$, as in round robin schedulers.

In contrast to optimal GPS approximations, such as WF^{2}Q [12] or the improved version in [23], the main limitations of the schedulers considered here are given below. However, these are practical conditions for most realistic configurations.

- A minimum packet size and service rate are required.
- A maximum packet size must be specified for each flow in KPS and Simple KPS.
- A priority encoder of width $\log_2(\frac{\text{link speed}}{\text{minimum rate}})$ bits is needed. For example, 32 bits can support service rates from 1 Kbit/s to 4 Tbit/s. If not supported by hardware, it can be implemented in software at cost $O(\log_2(\text{width}))$.
- The sum of relative rates must be less than 1.
- Timestamp rounding introduces small extra error terms.
- Packet-amortized constant complexity results in line-speed processing, only if the underlying hardware can support line-speed forwarding at minimum packet sizes.

Table I presents a summary of the schedulers discussed throughout this paper. For each type of scheduler, the table lists the dominant error term component (without coefficients) across its delay and fairness properties, the average execution overhead, as well as the necessary time period for amortizing the worst-case execution overhead, if applicable. The execution overhead does not necessarily follow from the respective original proposal, but is determined using the best available techniques for virtual time maintenance [23], as well as van Emde Boas priority queues [20] with a finite number of rounded time slots, if applicable. In the table, N refers to the number of flows, n to the number of stratification levels, K to the number of time slots in the time horizon, and C to the link speed. All other variables are the same as in the mathematical model. In the following sections, we briefly discuss additional interesting aspects about the analytical model and schedulers.

A. Analytical Model

The analytical model can be used to quickly assess a large variety of different schedulers. For example, LFVC [4] is a rate-controlled rounded timestamp proposal. Using its rounding scheme to determine the model parameters $\overline{S}_i$ and $\overline{F}_i$, the scheduler properties follow directly. For finish-time only schedulers such as WFQ [15] or SPFQ [22], Case 2 in Lemma 2 does not apply and the fairness properties cannot be assessed, but the delay properties follow directly by setting $\overline{F}_i = 0$ or the appropriate rounded finish time.

TABLE I
SUMMARY OF RATE PROPORTIONAL SCHEDULERS

Type	Worst-Case Scheduling Error	Execution Overhead	Amortization Period
GPS [11]	0	∞	n/a
VC [16]	∞	$O(logN)$ or $O(loglogK)$	n/a
WFQ [15]	$N \cdot L_{\max}$	$O(logN)$	n/a
SCFQ [10]	$N \cdot L_{\max}$	$O(logN)$ or $O(loglogK)$	n/a
SFQ [17]	$N \cdot L_{\max}$	$O(logN)$ or $O(loglogK)$	n/a
SPFQ [22]	$N \cdot L_{\max}$	$O(logN)$ or $O(loglogK)$	n/a
RR Variants (cf. Section II-G)	$N \cdot L_{\max}$	$O(1)$	n/a
WF^{2}Q [12]	$L_{\max} + L_i/r_i$	$O(logN)$	n/a
WF^{2}Q+ [19]	$L_{\max} + L_i/r_i$	$O(logN)$	n/a
LFVC [4]	$L_{\max} + L_i/r_i$	$O(logN)$ or $O(loglogK)$	$N \cdot L_{\max}/C$
SI-WF^{2}Q	$L_{\max} + L_i/r_i$	$O(1)$	$n \cdot L_{\max}/C$
KPS	$L_{\max} + L_i/r_i$	$O(1)$	L_i/C
S-KPS	$L_{\max} + L_i/r_i$	$O(1)$	L_i/C

N: number of flows, n: number of levels, K: number of time slots, C: link speed

B. Implementation Details

The schedulers proposed in this paper use a flow's maximum packet size to determine the appropriate stratification level. We do not consider this a major restriction, since QoS service models such as Integrated Services [1] require the specification of this parameter anyway. At runtime, flows simply stay in the `Active` container after service of a smaller packet. Smaller packets are still subject to the delay and fairness error terms based on the maximum packet size, but most timing-critical applications will operate with mostly fixed packet sizes anyway.

Assuming that the minimum packet size (λ resp. `g`) is a power of 2, the data path implementation of all proposed schedulers is extremely simple and only requires the management of linked lists, addition and shift operations, as well as a single multiplication to determine the virtual packet size. In addition, only the ffs operation is required to implement efficient search between stratified levels. The computation of the logarithm to determine the stratified level only happens at flow setup time and can be done on the control processor. Therefore, a *find last bit set* (fls) operation is not needed.

The KPS scheduler and its simpler variant have a significantly smaller memory footprint than SI-WF^{2}Q, because one or both containers are reduced in size. This should help with efficient implementation, since most of the relevant information can be stored in fast on-chip memory.

We have implemented a preliminary version of the KPS scheduler on an Intel IXP network processor to verify its feasibility and found that the absolute execution overhead seems small enough to support very high line rates [38]. However, a more thorough study of implementation aspects, along with a prototype implementation and lab experiments is needed to completely assess the implementation characteristics of any of the schedulers discussed in this paper.

C. CPU Scheduling?

The execution overhead of SI-WF^{2}Q and KPS is proportional to the size of the packet being transmitted and philosophically relies on the assumption that network line cards are designed to sustain a traffic mix consisting of only minimum size packets. Therefore, it is not clear whether it would be a suitable choice for CPU scheduling where the scheduler itself competes with the scheduled workload. The Simple KPS variant eliminates most of this bottleneck and might be a candidate for CPU scheduling. However, there are still operations that are proportional to the current unit of work.

VIII. CONCLUSIONS

We present a general analytical model to assess the scheduling properties of GPS approximation schedulers that operate on rounded timestamps. It can be used to quickly assess the scheduling properties of candidate schedulers. We also present proposals for different variations of the same basic scheduler design that illustrate the design choices for trading off implementation complexity and overhead with scheduling quality. The proposed schedulers are analyzed using the general model. We illustrate the results by simulations. The KPS and Simple KPS schedulers introduced here approximate GPS with near-optimal service properties and can be implemented with constant execution overhead. In future work, we will attempt to produce realistic implementations of these algorithms.

However, a packet scheduler is only one component in a very complex network architecture. The overall trade-offs of different architecture proposals and service models with respect to the viability of business models, which are mainly shaped by application demand, remain fundamentally unclear. Nevertheless, the availability of an efficient and sophisticated packet scheduler hopefully opens new avenues for the design and operation of packet-switched communication networks.

ACKNOWLEDGEMENTS

This work is supported by the Natural Sciences and Engineering Research Council of Canada. Early feedback from S. Keshav has helped shaping its direction. Paolo Valente has pointed out problems in previous versions of the paper and provided very useful feedback. The anonymous reviewers have also given valuable feedback to improve the presentation of the paper.

REFERENCES

[1] S. Shenker, C. Partridge, and R. Guerin, "RFC 2212 - Specification of Guaranteed Service," Sep. 1997.

[2] J. C. R. Bennett, K. Benson, A. Charny, and J.-Y. L. William F. Courtney, "Delay Jitter Bounds and Packet Scale Rate Guarantee for Expedited Forwarding," *IEEE/ACM Transactions on Networking*, vol. 10, no. 4, pp. 529–540, Aug. 2002.

[3] 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, Jun. 1993.

[4] S. Suri, G. Varghese, and G. Chandranmenon, "Leap Forward Virtual Clock: A New Fair Queuing Scheme with Guaranteed Delays and Throughput Fairness," in *Proceedings of INFOCOM 1997*. IEEE, Apr. 1997, pp. 557–565, technical report with full proofs available at http://citeseer.ist.psu.edu/74110.html.

[5] M. Karsten, "SI-WF^{2}Q: WF^{2}Q Approximation with Small Constant Execution Overhead," in *Proceedings of INFOCOM 2006*. IEEE, Apr. 2006.

[6] ——, "Approximation of Generalized Processor Sharing with Interleaved Stratified Timer Wheels," to appear in ACM/IEEE Transactions on Networking.

[7] P. Goyal and H. M. Vin, "Generalized Guaranteed Rate Scheduling Algorithms: A Framework," *IEEE/ACM Transactions on Networking*, vol. 5, no. 4, pp. 561–571, Aug. 1997.

[8] A. K. Parekh and R. G. Gallager, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Multiple Node Case," *IEEE/ACM Transactions on Networking*, vol. 2, no. 2, pp. 137–150, Apr. 1994.

[9] D. Stiliadis and A. Varma, "Latency-Rate Servers: A General Model for Analysis of Traffic Scheduling Algorithms," *IEEE/ACM Transactions on Networking*, vol. 6, no. 5, pp. 611–624, Oct. 1998.

[10] S. J. Golestani, "A Self-Clocked Fair Queuing Scheme for Broadband Applications," in *Proceedings of INFOCOM 1994*. IEEE, Jun. 1994, pp. 636–646.

[11] A. K. Parekh, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks," Ph.D. dissertation, Massachusetts Institute of Technology, Feb. 1992.

[12] J. C. R. Bennett and H. Zhang, "WF^{2}Q: Worst-case Fair Weighted Fair Queueing," in *Proceedings of INFOCOM 1996*. IEEE, Mar. 1996, pp. 120–128.

[13] H. Sariowan, R. L. Cruz, and G. Polyzos, "SCED: A Generalized Scheduling Policy for Guaranteed Quality of Service," *IEEE/ACM Transactions on Networking*, vol. 7, no. 5, pp. 669–684, Oct. 1999.

[14] I. Stoica, H. Zhang, and T. S. E. Ng, "Hierarchical Fair Service Curve Algorithm for Link-Sharing, Real-Time and Priority Service," *ACM Computer Communication Review*, vol. 27, no. 4, pp. 249–262, Oct. 1997, Proceedings of SIGCOMM 1997.

[15] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," *ACM Computer Communication Review*, vol. 19, no. 4, pp. 1–12, Sep. 1989, Proceedings of SIGCOMM 1989.

[16] L. Zhang, "Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks," *ACM Computer Communication Review*, vol. 20, no. 4, pp. 19–29, Sep. 1990, Proceedings of SIGCOMM 1990.

[17] P. Goyal, H. M. Vin, and H. Cheng, "Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks," *IEEE/ACM Transactions on Networking*, vol. 5, no. 5, pp. 690–704, Oct. 1997.

[18] D. Stiliadis and A. Varma, "A General Methodology for Designing Efficient Traffic Scheduling and Shaping Algorithms," in *Proceedings of INFOCOM 1997*. IEEE, Apr. 1997, pp. 326–335.

[19] J. C. R. Bennett and H. Zhang, "Hierarchical Packet Fair Queueing Algorithms," *IEEE/ACM Transactions on Networking*, vol. 5, no. 5, pp. 675–689, Oct. 1997.

[20] P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and Implementation of an Efficient Priority Queue," *Mathematical Systems Theory*, vol. 10, pp. 99–127, 1977.

[21] G. Varghese and A. Lauck, "Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility," *Operating Systems Review Special Issue: Proceedings of the Eleventh Symposium on Operating Systems Principles, Austin, TX, USA*, vol. 21, no. 5, pp. 25–38, Nov. 1987.

[22] D. Stiliadis and A. Varma, "Efficient Fair-Queueing Algorithms for Packet-Switched Networks," *IEEE/ACM Transactions on Networking*, vol. 6, no. 2, pp. 175–185, Apr. 1998.

[23] P. Valente, "Exact GPS Simulation with Logarithmic Complexity, and its Application to an Optimally Fair Scheduler," *ACM Computer Communication Review*, vol. 34, no. 4, pp. 269–280, Oct. 2004, Proceedings of SIGCOMM 2004.

[24] Q. Zhao and J. Xu, "On the Computational Complexity of Maintaining GPS Clock in Packet Scheduling," in *Proceedings of INFOCOM 2004*. IEEE, Mar. 2004.

[25] J. Xu and R. J. Lipton, "On Fundamental Tradeoffs between Delay Bounds and Computational Complexity in Packet Scheduling Algorithms," *IEEE/ACM Transactions on Networking*, vol. 13, no. 1, pp. 15–28, Feb. 2005.

[26] D. C. Stephens, J. C. Bennett, and H. Zhang, "Implementing Scheduling Algorithms in High-Speed Networks," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1145–1158, Jun. 1999.

[27] G. N. Rouskas and Z. Dwekat, "A Practical and Efficient Implementation of WF^{2}Q+," in *Proceedings of ICC 2007*. IEEE, 2007.

[28] C. Guo, "SRR: An O(1) Time Complexity Packet Scheduler for Flows in Multi-Service Packet Networks," *IEEE/ACM Transactions on Networking*, vol. 12, no. 6, pp. 1144–1155, Dec. 2004.

[29] ——, "G-3: An O(1) Time Complexity Packet Scheduler That Provides Bounded End-to-End Delay," in *Proceedings of INFOCOM 2007*. IEEE, 2007.

[30] S. Ramabhadran and J. Pasquale, "Stratified Round Robin: A Low Complexity Packet Scheduler with Bandwidth Fairness and Bounded Delay," *ACM Computer Communication Review*, vol. 33, no. 4, pp. 239–250, Oct. 2003, Proceedings of SIGCOMM 2003.

[31] X. Yuan and Z. Duan, "FRR: a Proportional and Worst-Case Fair Round Robin Scheduler," in *Proceedings of INFOCOM 2005*. IEEE, Apr. 2005.

[32] B. Caprita, J. Nieh, and W. C. Chan, "Group Round Robin: Improving the Fairness and Complexity of Packet Scheduling," in *Proceedings of the 2005 Symposium on Architecture for Networking and Communications Systems*. ACM Press, Oct. 2005, pp. 29–40.

[33] L. Zhong, J. Xu, and X. Wang, "VWQGRR: a Novel Packet Scheduer," in *Proceedings of ICN 2007*. IEEE, 2007.

[34] Y. Zhou and H. Sethu, "On the Relationship Between Absolute and Relative Fairness Bounds," *IEEE Communication Letters*, vol. 6, no. 1, pp. 37–39, Jan. 2002.

[35] "The Aggregate Magic Algorithms." [Online]. Available: http://aggregate.org/MAGIC

[36] Intel Corp., "Intel 64 and IA-32 Architectures Optimization Reference Manual," Dec. 2008, Order Number: 248966-017. [Online]. Available: http://www.intel.com/products/processor/manuals/

[37] ——, "Intel IXP2805 Network Processor - Programmer's Reference Manual," Apr. 2006, Order Number: 310015, Revision: 007US.

[38] M. Groves, "Concurrent Implementation of Packet Processing Algorithms on Network Processors," Master's thesis, University of Waterloo, 2006. [Online]. Available: http://hdl.handle.net/10012/2937