On the Fairness-Efficiency Tradeoff for Packet Processing with Multiple Resources

Wei Wang, Chen Feng, Baochun Li, and Ben Liang
Department of Electrical and Computer Engineering, University of Toronto
{weiwang, cfeng, bli, liang}@ece.utoronto.ca

ABSTRACT

Middleboxes are widely deployed in today's networks. They apply a variety of complex network functions to transform, filter, and optimize incoming traffic based on the payload of packets. These functions require the support of multiple types of resources, such as CPU and link bandwidth, for processing incoming packets. Hence, a multi-resource packet scheduling algorithm is needed to allow flows to share these resources fairly and efficiently. However, unlike traditional fair queueing where bandwidth is the only concern, we show in this paper that fairness and efficiency are conflicting objectives that cannot be achieved simultaneously in the presence of multiple resources. Ideally, a scheduling algorithm should allow network operators to flexibly specify their fairness and efficiency requirements, so as to meet the Quality of Service demands while keeping the system at a high utilization level. Yet, existing multi-resource scheduling algorithms focus on fairness only, and may lead to poor resource utilization. In this paper, we propose a new scheduling algorithm to achieve a flexible tradeoff between fairness and efficiency for packet processing, consuming both CPU and link bandwidth. Experimental results based on both real-world implementation and trace-driven simulation suggest that trading off a modest level of fairness can potentially improve the efficiency to the point where the system capacity is almost saturated.

Categories and Subject Descriptors

C.2.6 [**Computer-Communication Networks**]: Internetworking

General Terms

Scheduling Theory

Keywords

Fair Queueing; Middleboxes; Fairness-Efficiency Tradeoff

1. INTRODUCTION

Queueing algorithms determine the order in which packets in various independent flows are processed, and serve as a fundamental mechanism for allocating resources in a network appliance. Traditional queueing algorithms [1, 9, 21, 28] make scheduling decisions in network switches that simply forward packets to their next hops, and link bandwidth is the only resource being allocated.

In modern network appliances, *e.g.*, *middleboxes* [25, 27], link bandwidth is no longer the only resource shared by flows. In addition to packet forwarding, middleboxes perform a variety of critical network functions that require deep packet inspection based on the payload of packets, such as IP security encryption, WAN optimization, and intrusion detection. Performing these complex network functions requires the support of multiple types of resources, and may bottleneck on either CPU or link bandwidth [10, 13]. For example, flows that require basic forwarding may congest the link bandwidth [13], while those that require IP security encryption need more CPU processing time [10]. A queueing algorithm specifically designed for multiple resources is therefore needed for sharing these resources *fairly* and *efficiently*.

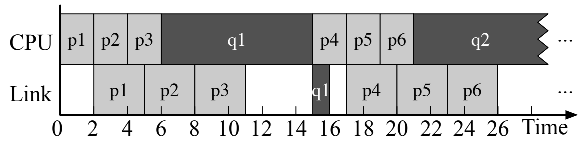
Fairness offers predictable service isolation among flows. It ensures that the service a flow receives (*i.e.*, number of packets processed per second) in an n-flow system is at least $1/n$ of that it achieves when the flow monopolizes all resources, independent of the behavior of other rogue flows. The notion of Dominant Resource Fairness (DRF) [14, 22] embodies this isolation property, with which each flow receives approximately the same processing time on its *dominant resource*, defined as the one that requires the most packet processing time [13].

Efficiency serves as another important metric measuring the resource utilization achieved by a queueing algorithm. High resource utilization naturally translates into high traffic throughput. This is of particular importance to enterprise networks, given the surging volume of traffic passing through middleboxes [27, 36].

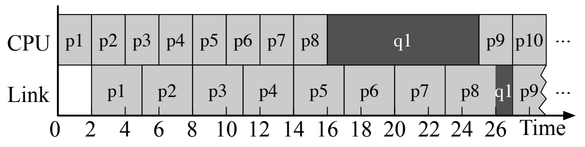
Both fairness and efficiency can be achieved at the same time in traditional single-resource fair queueing, where bandwidth is the only concern. As long as the schedule is *work conserving* [39], bandwidth utilization is 100% given a non-empty system. That leaves fairness as an independent objective to optimize.

However, in the presence of multiple resources, fairness is often a *conflicting objective* against efficiency. To see this, consider two schedules shown in Fig. 1 with two flows whose packets need CPU processing before transmission. Packets that finishes CPU processing are placed into a buffer in front of the output link. Each packet in Flow 1 has a *processing time vector* $\langle 2, 3 \rangle$, meaning that it requires 2 time units for CPU processing and 3 time units for transmission; each packet in Flow 2 has a processing time vector $\langle 9, 1 \rangle$. The dominant resource of Flow 1 is link bandwidth, as it takes more time to transmit a packet than processing it using CPU; similarly, the dominant resource of Flow 2 is CPU. To achieve DRF, the transmission time Flow 1 receives should be ap-

CoNEXT'14, December 2–5, 2014, Sydney, Australia.
Copyright 2014 ACM 978-1-4503-3279-8/14/12 ...$15.00.
http://dx.doi.org/10.1145/2674005.2675010.



(a) A packet schedule that is fair but inefficient.



(b) A packet schedule that is efficient but violates DRF.

Figure 1: An example showing the tradeoff between fairness and efficiency for multi-resource packet scheduling. Packets that finishes CPU processing are placed into a buffer in front of the output link. Flow 1 sends packets p1, p2, ..., each having a processing time vector $\langle 2, 3\rangle$; Flow 2 sends packets q1, q2, ..., each having a processing time vector $\langle 9, 1\rangle$. Schedule (a) achieves DRF but is inefficient; Schedule (b) is efficient but unfair.

proximately equal to the CPU processing time Flow 2 receives. In this sense, Flow 1 should schedule three packets whenever Flow 2 schedules one, so that each flow receives 9 time units to process its dominant resource, as shown in Fig. 1a. This schedule, though fair, leads to poor bandwidth utilization—the link is idle for 1/3 of the time. On the other hand, Fig. 1b shows a schedule that achieves 100% CPU and bandwidth utilization by serving eight packets of Flow 1 and one packet of Flow 2 alternately. The schedule, though efficient, violates DRF. While Flow 1 receives $24/25$ of the link bandwidth, Flow 2 receives only $9/25$ of the CPU time.

The fairness-efficiency tradeoff shown in the example above generally exists for multi-resource packet scheduling, but it has received little attention before. Existing multi-resource queueing algorithms focus solely on fairness [13, 32, 35]. However, for applications having a loose fairness requirement, trading off a modest degree of fairness for higher efficiency and higher throughput is well justified. In general, depending on the underlying applications, a network operator may weigh fairness and efficiency differently. Ideally, a multi-resource queueing algorithm should allow network operators to flexibly specify their tradeoff preference and implement the specified tradeoff by determining the "right" packet scheduling order.

However, designing such a queueing algorithm is non-trivial. It remains to be seen how *efficiency* can be quantitatively defined. Further, it remains open how the tradeoff requirement should be appropriately specified. But most importantly, given a specific tradeoff requirement, how can the scheduling decision be correctly made to implement it?

This paper represents the first attempt to address these challenges. We clarify the efficiency measure as the *schedule makespan*, which is the completion time of the last flow. We show that achieving a flexible tradeoff between fairness and efficiency is generally NP-hard. We hence limit our discussion to a typical scenario where CPU and link bandwidth are the two types of resources required for packet processing, which is usually the case in middleboxes. We show that the fairness-efficiency tradeoff can be strictly enforced by a GPS-like (Generalized Processor Sharing [9, 21]) fluid model, where packets are served in arbitrarily small increments on both resources. To implement the idealized fluid in the real world, we design a packet-by-packet tracking algorithm, using an approach similar to the virtual time implementation of Weighted Fair Queueing (WFQ) [9, 16, 21]. We have prototyped our tradeoff algorithm in the *Click* modular router [19]. Both our prototype implementation and trace-driven simulation show that a $15\% \sim 20\%$ fairness tradeoff is sufficient to achieve the optimal efficiency, leading to a nearly 20% improvement in bandwidth throughput with a significantly higher resource utilization.

2. FAIRNESS AND EFFICIENCY

Before discussing the tradeoff between fairness and efficiency, we shall first clarify how the notion of fairness is to be defined, and how efficiency is to be measured quantitatively. We model packet processing as going through a resource pipeline, where the first resource is consumed to process the packet first, followed by the second, and so on. A packet is not available for the downstream resource until the processing on the upstream resource finishes. For example, a packet cannot be transmitted (which consumes link bandwidth) before it has been processed by CPU.

2.1 Dominant Resource Fairness

Fairness is one of the primary design objectives for a queueing algorithm. A fair schedule offers service isolation among flows by allowing each flow to receive the throughput at least at the level when every resource is evenly allocated. The notion of Dominant Resource Fairness (DRF) embodies this isolation property by achieving the max-min fairness on the *dominant resources* of packets in their respective flows [13]. The dominant resource of a packet is defined as the one that requires the maximum packet processing time. In particular, let $\tau_r(p)$ be the time required to process packet p on resource r. The dominant resource of packet p is $r_p = \arg\max_r \tau_r(p)$. Given a packet schedule, let $D_i(t_1, t_2)$ be the time flow i receives to process the dominant resources of its packets in a backlogged period (t_1, t_2). The function $D_i(t_1, t_2)$ is referred to as the *dominant service* flow i receives in (t_1, t_2). A schedule is said to *strictly* implement DRF if for all flows i and j, and for any period (t_1, t_2) they backlog, we have

$$D_i(t_1, t_2) = D_j(t_1, t_2). \qquad (1)$$

In other words, a strict DRF schedule allows each flow to receive the same dominant service in any backlogged period.

However, because packets are scheduled as separate entities and are transmitted in sequence, strictly implementing DRF at all times may not be possible in practice. For this reason, a practical fair schedule only requires flows to receive *approximately the same* dominant services over time [13, 32, 35], as shown in the previous example of Fig. 1a.

2.2 The Efficiency Measure

In addition to fairness, efficiency is another important concern for a multi-resource scheduling algorithm, but has received no significant attention before. Even the definition of efficiency needs clarification.

Perhaps the most widely adopted efficiency measure is system throughput, whose conventional definition is the *rate of completions* [17], computed as the processed workload divided by the elapsed time (*e.g.*, bits per second). While this performance metric is well defined for single-resource systems, extending its definition to multiple types of resources leads to a *throughput vector*, where each component is the throughput of one type of resource (*e.g.*, 10 CPU instruction completions per second and 5 bits transmitted through the output link per second), and different throughput vectors may not be comparable.

Another possible efficiency measure is *resource utilization given non- empty system*, or simply *resource utilization* in the remainder of this paper.[1] However, in a middlebox, different resources may see different levels of utilization. The question is: how should the "system utilization" be properly defined? One possible definition is to add up the utilization rates of all resources. This definition implicitly assumes *exchangeable resources*, say, 1% CPU usage is equivalent to 1% bandwidth consumption, which may not be well justified in many circumstance, especially when one type of resource is scarce in the system and is valued more than the other.

In this paper, we measure efficiency with the *schedule makespan*. Given input flows with a finite number of packets, the makespan of a schedule is defined as the time elapsed from the arrival of the first packet to the time when all packets finish processing on all resources. One can also view makespan as the completion time of the last flow. Intuitively, given a finite traffic input, the shorter the makespan is, the faster the input traffic is processed, and the more efficient the schedule is.[2]

2.3 Tradeoff between Fairness and Efficiency

With the precise measure of efficiency, we are curious to know how much efficiency is sacrificed for fair queueing. To answer this question, we first generalize the definition of work conserving schedules from traditional single-resource fair queueing to multiple resources. In particular, we say a schedule is *work conserving* if at least one resource is fully utilized for packet processing when there is a backlogged flow. In other words, a work conserving schedule does not allow resources to be wasted in idle if they can be used to process a backlogged packet. Existing multi-resource fair queueing algorithms [13, 32, 35] use the goal of achieving work conservation as an indication of efficiency. However, in the theorem below, we observe that such an approach is ineffective.

Theorem 1. *Let m be the number of resource types concerned. Given any traffic input I, let $T^{\sigma}(I)$ be the makespan of a work conserving schedule σ, and $T^{*}(I)$ the minimum makespan of an optimal schedule. We have*

$$T^{\sigma}(I) \leq mT^{*}(I). \tag{2}$$

PROOF. Given a traffic input I, let the work conserving schedule σ consist of n_b *busy period*. A busy period is a time interval during which at least one type of resource is used for packet processing. When the system is empty and a new packet arrives, a new busy period starts. The busy period ends when the system becomes empty again. We consider the following two cases.

Case 1: $n_b = 1$. Let traffic input I consist of N packets, ordered based on their arrival times, where packet 1 arrives first. For packet i, let $\tau_r^{(i)}$ be its packet processing time on resource r. It is easy to check that the following inequality holds for the optimal schedule with the minimum makespan:

$$T^{*}(I) \geq \max_r \sum_{i=1}^{N} \tau_r^{(i)}. \tag{3}$$

On the other hand, for work conserving schedule σ, its makespan reaches the maximum when packet processing does not overlap in time, across all resources, *i.e.*,

$$T^{\sigma}(I) \leq \sum_{i=1}^{N} \sum_{r=1}^{m} \tau_r^{(i)} \tag{4}$$

This leads to the following inequalities:

$$T^{\sigma}(I) \leq \sum_{i=1}^{N} \sum_{r=1}^{m} \tau_r^{(i)} \leq \sum_{i=1}^{N} m \max_r \tau_r^{(i)} \leq mT^{*}(I). \tag{5}$$

Case 2: $n_b > 1$. Given traffic input I, let $I(t^{+})$ be the packets that arrive on or after time t. For schedule σ, let t_0 be the time when its second last busy period ($n_b - 1$) ends, and t_1 the time when the last busy period (n_b) starts. Because schedule σ is work conserving, no packet arrives between t_0 and t_1. We have

$$T^{\sigma}(I) = t_1 + T^{\sigma}(I(t_1^{+})), \tag{6}$$

and

$$T^{*}(I) = t_1 + T^{*}(I(t_1^{+})). \tag{7}$$

Note that given traffic input $I(t_1^{+})$, schedule σ consists of only one busy period. By the discussion of Case 1, we have

$$\begin{aligned} T^{\sigma}(I) &= t_1 + T^{\sigma}(I(t_1^{+})) \\ &\leq t_1 + mT^{*}(I(t_1^{+})) \\ &\leq mT^{*}(I), \end{aligned} \tag{8}$$

where the last inequality is derived from (7). □

We make the following three observations from Theorem 1. First, the tradeoff between fairness and efficiency is a *unique* challenge facing multi-resource scheduling. When the system consists of only one type of resource (*i.e.*, $m = 1$), work conservation is sufficient to achieve the minimum makespan, leaving fairness as the only concern. For this reason, efficiency has never been a problem for traditional single-resource fair queueing. Second, while work conservation also provides some efficiency guarantee for multi-resource scheduling, the more types of resources, the weaker the guarantee. Third, even with a small number of resource types, the efficiency loss could be quite significant. Since bandwidth throughput is inversely proportional to the schedule makespan, Theorem 1 implies that solely relying on work conservation may incur up to 50% loss of bandwidth throughput when there are two types of resources. While this is based on the worst case, as we shall see later in §6, our experiments confirm that a throughput loss of as much as 20% is introduced by the existing fair queueing algorithms. Trading off some degree of fairness for higher efficiency is therefore well justified, especially for applications with loose fairness requirements.

2.4 Challenges

Unfortunately, striking a desired balance between fairness and efficiency in a multi-resource system is technically non-trivial. Even minimizing the makespan without regard to fairness—a special case of fairness-efficiency tradeoff—is NP-hard. In particular, we note that minimizing the makespan of a packet schedule can be modeled as a *multi-stage flow shop* problem [6, 20, 23] studied in operations research, where the equivalent of a packet is a job, and the equivalent of a type of resource is a machine. However, flow shop scheduling is a notoriously hard problem, even in its *offline setting* where the entire input is known beforehand. Specifically, when all jobs (packets) are available at the very beginning, finding the mini-

[1]This definition is different from that of queueing theory, where the utilization is defined as the fraction of time a device is busy [17]. Under this definition, high utilization usually means a high congestion level with a large queue backlog and long delays [37], and is usually not desired.

[2]In general, makespan is not the only efficiency measure that one can define. For example, we can also measure efficiency with the average flow completion time. We choose makespan as the efficiency measure in this paper because it leads to tractable analysis. More importantly, makespan closely relates to "system utilization" and is conceptually easy to understand. The discussion of other possible efficiency measures is out of the scope of this paper.

mum makespan is strongly NP-hard when the number of machines (resources) is greater than two [12].

Given the hardness results above, in this paper, we limit our discussion to two types of resources, CPU and link bandwidth, as these are the two most concerned middlebox resources [13,25]. We note that even with two types of resources, minimizing the schedule makespan remains a hard problem. Because packets arrive dynamically over time, the problem resembles a 2-machine *online flow shop scheduling* problem where jobs (packets) do not reveal their information until they arrive. For this problem, only a limited amount of *negative* results is known [6,20,23,26,30]. Specifically, no online algorithm can ensure a makespan within a factor of 1.349 of the optimum *in all cases* [24]. We also notice that no existing work gives a concrete solution, even a heuristic algorithm, that jointly considers both makespan and fairness.

3. FAIRNESS, EFFICIENCY, AND THEIR TRADEOFF IN THE FLUID MODEL

The difficulty of makespan minimization is mainly introduced by the combinatorial nature of multi-resource scheduling. One approach to circumvent this problem is to consider a *fluid relaxation*, where packets are served in arbitrarily small increments on all resources. For each packet, this is equivalent to processing it *simultaneously* on all resources with the *same* progress, and head-of-line packets of backlogged flows can also be served *in parallel*, at (potentially) different processing rates. Such a parallel processing fluid model eliminates the need for discussing the scheduling orders of flows. Instead, it allows us to focus on the resource shares allocated to flows, hence relaxing a combinatorial optimization problem to a simpler dynamic resource allocation problem. While in general, optimally solving such a dynamic problem requires knowing future packet arrivals, we show in this section that, under some practical assumptions, a greedy algorithm gives an optimal *online* schedule with the minimum makespan. We can then strike a balance between efficiency and fairness by imposing some fairness constraints to the fluid schedule. We shall discuss later in §4 and §5 how this fluid schedule is implemented in practice with a packet-by-packet tracking algorithm at acceptable complexity.

3.1 Fluid Relaxation

In the fluid model, a flow is relaxed to a fluid where each of its packets is served simultaneously on all resources with the same progress. Packets of different flows are also served in parallel. The schedule needs to decide, at each time, the resource share allocated to each backlogged flow. In particular, let $\mathcal{B}^t$ be the set of flows that are backlogged at time t. Let $a_{i,r}^t$ be the fraction (share) of resource r allocated to flow i at time t. The fluid schedule determines, at each time t, the resource allocation $a_{i,r}^t$ for each backlogged flow i and each resource r.

Two constraints must be satisfied when making resource allocation decisions. First, we must ensure that no resource is allocated more than its total availability:

$$\sum_{i \in \mathcal{B}^t} a_{i,r}^t \leq 1, \quad r = 1, 2. \tag{9}$$

The second constraint ensures that a packet is processed at a *consistent* rate across resources. In particular, for a backlogged flow i and its head-of-line packet at time t, let $\tau_{i,r}^t$ be its packet processing time on resource r, and

$$r_i = \arg\max_r \tau_{i,r}^t \tag{10}$$

be its dominant resource. The processing rate that this packet receives on resource r is computed as the ratio between the resource share allocated and the processing time required: $a_{i,r}^t/\tau_{i,r}^t$. To ensure a consistent processing rate, we have

$$a_{i,r}^t/\tau_{i,r}^t = a_{i,r'}^t/\tau_{i,r'}^t, \quad \text{for all } r \text{ and } r'.$$

Substituting r_i into r' above, we see a *linear relation* between the allocation share of resource r and that of the dominant resource:

$$a_{i,r}^t = \frac{\tau_{i,r}^t}{\tau_{i,r_i}^t} a_{i,r_i}^t = \bar{\tau}_{i,r}^t d_i^t, \tag{11}$$

where

$$\bar{\tau}_{i,r}^t = \tau_{i,r}^t/\tau_{i,r_i}^t \tag{12}$$

is the *normalized packet processing time* on resource r, and

$$d_i^t = a_{i,r_i}^t \tag{13}$$

is the *dominant share* allocated to flow i at time t. Plugging (11) into (9), we combine the two constraints into one *feasibility constraint* of a fluid schedule:

$$\sum_{i \in \mathcal{B}^t} \bar{\tau}_{i,r}^t d_i^t \leq 1, \quad r = 1, 2. \tag{14}$$

Before we discuss the tradeoff between fairness and efficiency, we first consider two special cases, where either fairness or efficiency is the only objective to optimize in the fluid model. For ease of presentation, we drop the superscript t when time can be clearly inferred from the context. Table 1 summarizes the main notations used in the fluid model.

Table 1: Main notations used in the fluid model. The superscript t is dropped when time can be clearly inferred from the context.

Notation	**Explanation**
n	maximum number of flows that are concurrently backlogged
α	fairness knob specified by the network operator
$\mathcal{B}$ (or $\mathcal{B}^t$)	set of flows that are currently backlogged (at time t)
d_i (or d_i^t)	dominant share allocated to flow i (at time t)
$\bar{d}$ (or $\bar{d}^t$)	fair dominant share (at time t), given by (16)
$\tau_{i,r}$ (or $\tau_{i,r}^t$)	packet processing time on resource r required by the head-of-line packet of flow i (at time t)
$\bar{\tau}_{i,r}$ (or $\bar{\tau}_{i,r}^t$)	normalized $\tau_{i,r}$ (or $\bar{\tau}_{i,r}^t$), defined by (12)

3.2 Fluid Schedule with Perfect Fairness

We first consider the fairness objective. To achieve perfect DRF, the fluid schedule enforces strict max-min fairness on flows' dominant shares, under the feasibility constraint. Specifically, the fluid schedule solves the following *DRF allocation problem* [14, 22] at each time t:

$$\begin{aligned} \max_{d_i} \quad & \min_{i \in \mathcal{B}} d_i \\ \text{s.t.} \quad & \sum_{i \in \mathcal{B}} \bar{\tau}_{i,r} d_i \leq 1, \quad r = 1, 2. \end{aligned} \tag{15}$$

Let n be the number of backlogged flows. The optimal solution, denoted by $\bar{\mathbf{d}} = (\bar{d}_1, \ldots, \bar{d}_n)$, allocates each backlogged flow the same dominant share, *i.e.*,

$$\bar{d}_i = \bar{d} = 1/\max\left\{\sum_i \bar{\tau}_{i,1}, \sum_i \bar{\tau}_{i,2}\right\}. \tag{16}$$

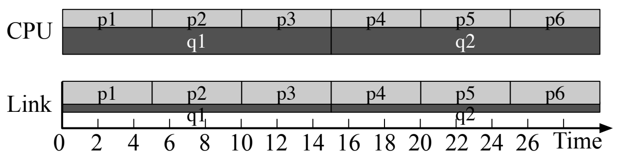


Figure 2: The DRGPS fluid that implements the perfect fairness in the example of Fig. 1. Flow 1 sends packets p1, p2, ..., and receives $\langle 3/5$ CPU, $1/15$ bandwidth$\rangle$; Flow 2 sends packets q1, q2, ..., and receives $\langle 3/5$ CPU, $1/15$ bandwidth$\rangle$. Only 2/3 of the link bandwidth is utilized.

In any backlogged periods, because flows are allocated the same dominant shares, they receive the same dominant services, achieving strict DRF at all times. The resulting fluid schedule is also known as DRGPS [33], a multi-resource generalization to the well-known GPS [9, 21].

Any discrete fair schedule is essentially a packet-by-packet approximation to DRGPS. For instance, applying DRGPS to the example of Fig. 1 leads to a fluid schedule shown in Fig. 2, where the normalized packet processing times of Flow 1 and Flow 2 are $\langle \bar{\tau}_{1,1}, \bar{\tau}_{1,2} \rangle = \langle 2/3, 1 \rangle$ and $\langle \bar{\tau}_{2,1}, \bar{\tau}_{2,2} \rangle = \langle 1, 1/9 \rangle$, respectively. By (16), both flows are allocated the same dominant share $\bar{d} = 3/5$. Specifically, Flow 1 receives $\langle 2/5$ CPU, $3/5$ bandwidth$\rangle$; Flow 2 receives $\langle 3/5$ CPU, $1/15$ bandwidth$\rangle$. In total, only 2/3 of the bandwidth is utilized, the same as the discrete fair schedule shown in Fig. 1a.

3.3 Fluid Schedule with Optimal Efficiency

We next discuss the efficiency objective. While there are some schedules proposed in the operations research literature that can achieve the minimum makspan for a flow shop problem, none of them applies in the context of packet scheduling: they either assume no packet arrivals (*e.g.*, [29]) or require full knowledge of future information (*e.g.*, [5]). We propose a simple *greedy* fluid schedule as follows.

For a given time instant, we define the system's *instantaneous dominant throughput* as the sum of the dominant share allocated, *i.e.*, $\sum_{i \in \mathcal{B}} d_i$. Intuitively, by maximizing $\sum_{i \in \mathcal{B}} d_i$ at all times, one would expect a high *average dominant throughput* $\sum_{i \in \mathcal{B}} D_i/T$, where T is the schedule makespan and D_i is the total dominant services (processing time) required by flow i. Given dominant workload $\sum_{i \in \mathcal{B}} D_i$, maximizing the average dominant throughput is equivalent to minimizing the schedule makespan T. Following this intuition, we propose a greedy fluid schedule that solves the following resource allocation problem to maximize the instantaneous dominant throughput at every time:

$$\begin{aligned} \max_{d_i \geq 0} \quad & \sum_{i \in \mathcal{B}} d_i \\ \text{s.t.} \quad & \sum_{i \in \mathcal{B}} \bar{\tau}_{i,r} d_i \leq 1, \quad r = 1, 2. \end{aligned} \tag{17}$$

In case that the optimal solution, denoted $\mathbf{d}^* = (d_1^*, \ldots, d_n^*)$, is not unique, the schedule chooses the one with the maximum overall utilization:

$$\max_{d_i^*} \sum_r \sum_{i \in \mathcal{B}} \bar{\tau}_{i,r} d_i^*. \tag{18}$$

In the example of Fig. 1, solving (17) allocates Flow 1 the dominant share $d_1^* = 9/25$ and Flow 2 the dominant share $d_2^* = 24/25$. It is easy to check that both CPU and link bandwidth are fully utilized.

Compared to those schedules proposed in the operations research literature, the greedy schedule defined by (17) is particularly attractive for packet scheduling due to the following three properties. First, it is an online algorithm without any *a priori* knowledge of future packet arrivals. Further, among all packets that are backlogged, only the information regarding head-of-line packets is required. This suggests that the schedule only needs to maintain a very simple per-flow state. Most importantly, the greedy schedule is more than a simple heuristic. Below we show that under some practical assumptions, greedily maximizing the dominant throughput gives the minimum makespan. Our analysis requires the following lemma, where we show that the schedule will not waste any resource in idle, unless all flows bottleneck on the same resource, in which case the other resource cannot be fully utilized anyway. The proof is deferred to our technical report [31] due to space constraint.

Lemma 1. *The fluid schedule defined by* (17) *fully utilizes both resources if there are two head-of-line packets with different dominant resource,* i.e.*, there exist two flows j and l, such that $\bar{\tau}_{j,1} = 1 > \bar{\tau}_{j,2}$ and $\bar{\tau}_{l,1} < \bar{\tau}_{l,2} = 1$.*

With Lemma 1, we analyze the makespan of the fluid schedule defined by (17). Following [13], we say a flow is *dominant-resource monotonic* if it does not change its dominant resource during backlogged periods. To make the analysis tractable, we assume that flows are dominant-resource monotonic. This is often true in practice as packets in the same flow usually undergo the same processing, and hence have the same dominant resource. The following lemma,whose proof can be found in [31], states the optimality of the fluid schedule in a static scenario without dynamic packet arrivals.

Lemma 2. *For dominant-resource monotonic flows, the fluid schedule defined by* (17) *gives the minimum makespan if all packets are available at the beginning.*

We now extend the results of Lemma 2 to an online case where packets dynamically arrive over time. The following theorem gives the optimality condition of the fluid schedule. The proof is deferred to [31].

Theorem 2. *For dominant-resource monotonic flows, the fluid schedule defined by* (17) *gives the minimum makespan among all schedules, if after the system has two flows with different dominant resources, whenever a new flow arrives, there exist two backlogged flows with different dominant resources.*

The optimality conditions required by Theorem 2 can be easily met in practice. Because the number of backlogged flows is usually large, it is almost true that we can always find two flows with different dominant resources. In fact, even in a very unfortunate case where all flows bottleneck on the same resource, the greedy fluid schedule does not deviate far away from the optimum: no matter what fluid schedule is used, the bottleneck resource is always fully utilized when the system is non-empty and hence has the same backlog, which is a dominant factor in determining the schedule makespan.

The significance of Theorem 2 is that it connects makespan, a measure defined in the *time domain*, to the instantaneous dominant throughput, a measure defined in the *space domain*. More importantly, it shows that minimizing the former is, in a practical sense, equivalent to maximizing the latter at all times, without the need to know future packet arrivals. We shall use this intuition to strike a balance between fairness and efficiency in the next subsection.

3.4 Tradeoff between Fairness and Efficiency

When both fairness and efficiency are considered, we express the tradeoff between the two conflicting objectives as a constrained optimization problem—minimizing makespan under some specified fairness requirements. Recall that when perfect fairness is enforced, all flows receive the same dominant share $\bar{d}$ computed by (16), *i.e.*, $d_i = \bar{d}$ for all i. When fairness is not a strict requirement, we introduce a *fairness knob* $\alpha \in [0, 1]$ to specify the fairness degradation. In particular, an allocation $\mathbf{d}$ is called *α-portion fair* if $d_i \geq \alpha\bar{d}$ for all backlogged flow i. In other words, each flow receives at least an α-portion of its fair dominant share $\bar{d}$. A fluid schedule is called α-portion fair if it achieves the α-portion fair allocation at all times.

By choosing different values for α, a network operator can precisely control the fairness degradation. As two extreme cases, setting $\alpha = 0$ means that fairness is not considered at all; setting $\alpha = 1$ means that perfect fairness must be enforced at all times.

Given the specified fairness knob α, the fluid schedule tries to minimize makespan under the corresponding α-portion fairness constraints. Since minimizing makespan is, in a practical sense, equivalent to maximizing the system's dominant throughput, we obtain a simple tradeoff heuristic that maximizes the dominant throughput, subject to the required α-portion fairness at every time t:

$$\begin{aligned} \max_{d_i} \quad & \sum_{i \in \mathcal{B}} d_i \\ \text{s.t.} \quad & \sum_{i \in \mathcal{B}} \bar{\tau}_{i,r} d_i \leq 1, \quad r = 1, 2, \\ & d_i \geq \alpha\bar{d}, \quad \forall i \in \mathcal{B}, \end{aligned} \tag{19}$$

where the fair share $\bar{d}$ is given by (16). We see that the fluid schedule captures both DRGPS and the greedy schedule defined by (17) as special cases with $\alpha = 1$ and 0, respectively.

Special Solution Structure. The tradeoff problem (19) has a *closed-form* solution, based on which the tradeoff schedule can be easily computed. We first allocate each flow its guaranteed portion of dominant share $\alpha\bar{d}$. We then denote

$$\tilde{d}_i = d_i - \alpha\bar{d} \tag{20}$$

as the *bonus dominant share* allocated to flow i. Substituting (20) into (19), we equivalently rewrite (19) as a problem of determining the bonus dominant share received by each flow:

$$\begin{aligned} \max_{\tilde{d}_i \geq 0} \quad & \sum_{i \in \mathcal{B}} \tilde{d}_i + |\mathcal{B}|\alpha\bar{d} \\ \text{s.t.} \quad & \sum_{i \in \mathcal{B}} \bar{\tau}_{i,r} \tilde{d}_i \leq \mu_r \quad r = 1, 2, \end{aligned} \tag{21}$$

where

$$\mu_r = 1 - \alpha\bar{d} \sum_{i \in \mathcal{B}} \bar{\tau}_{i,r}, \quad r = 1, 2, \tag{22}$$

and is the *remaining share* of resource r after each flow receives its guaranteed dominant share $\alpha\bar{d}$. Without loss of generality, we sort all the backlogged flows based on the processing demands on the two types of resources required by their head-of-line packets as follows:

$$\bar{\tau}_{1,1}/\bar{\tau}_{1,2} \geq \cdots \geq \bar{\tau}_{n,1}/\bar{\tau}_{n,2}. \tag{23}$$

The following theorem shows that at most two flows are awarded the bonus share at a time. Its proof is deferred to our technical report [31].

Theorem 3. *There exists an optimal solution $\tilde{\mathbf{d}}^*$ to (21) where $\tilde{d}_i^* = 0$ for all $2 \leq i \leq n - 1$. In particular, $\tilde{\mathbf{d}}^*$ is given in the following three cases:*

Case 1: $\mu_1/\mu_2 < \bar{\tau}_{n,1}/\bar{\tau}_{n,2}$. *In this case, resource 1 is fully utilized, with $\tilde{d}_n^* = \mu_1/\bar{\tau}_{n,1}$ and $\tilde{d}_i^* = 0$ for all $i < n$.*

Case 2: $\mu_1/\mu_2 > \bar{\tau}_{1,1}/\bar{\tau}_{1,2}$. *In this case, resource 2 is fully utilized, with $\tilde{d}_1^* = \mu_2/\bar{\tau}_{1,2}$ and $\tilde{d}_i^* = 0$ for all $i > 1$.*

Case 3: $\bar{\tau}_{n,1}/\bar{\tau}_{n,2} \leq \mu_1/\mu_2 \leq \bar{\tau}_{1,1}/\bar{\tau}_{1,2}$. *In this case, both resources are fully utilized, and we have*

$$\tilde{d}_i^* = \begin{cases} (\mu_1\bar{\tau}_{n,2} - \mu_2\bar{\tau}_{n,1})/(\bar{\tau}_{1,1}\bar{\tau}_{n,2} - \bar{\tau}_{1,2}\bar{\tau}_{n,1}), & i = 1; \\ (\mu_2\bar{\tau}_{1,1} - \mu_1\bar{\tau}_{1,2})/(\bar{\tau}_{1,1}\bar{\tau}_{n,2} - \bar{\tau}_{1,2}\bar{\tau}_{n,1}), & i = n; \\ 0, & o.w. \end{cases}$$

Once the optimal bonus dominant share has been determined as shown above, the optimal solution $\mathbf{d}^*$ to (19), which is the dominant share allocated to each flow, can be easily computed as the sum of the bonus share and the guaranteed share:

$$d_i^* = \tilde{d}_i^* + \alpha\bar{d}, \quad \text{for all } i. \tag{24}$$

We give an intuitive explanation of Theorem 3 as follows. The first two cases of Theorem 3 correspond to the scenario where after each flow receives its guaranteed share, the remaining amounts of the two types of resources are *unbalanced* and cannot be fully utilized simultaneously. In this case, the schedule awards the bonus share to the flow (either Flow 1 or Flow n) whose processing demands can better utilize the remaining resources. The third case covers the scenario where the remaining amounts of the two types of resources are *balanced*, and can be fully utilized when the system is non-empty. In this case, they are allocated to two flows (Flow 1 and Flow n) with *complementary* resource demands as their bonus shares.

Theorem 3 reveals an important structure, that *at most two* flows are allocated more dominant shares than others. We refer to these flows as the *favored flows* and all the others as the *regular flows*. We shall show in §5 that this structure leads to an efficient $O(\log n)$ implementation of the fluid schedule.

4. PACKET-BY-PACKET TRACKING

So far, all our discussions are based on an idealized fluid model. In practice, however, packets are processed as separate entities. In this section, we present a discrete tracking algorithm that implements the fluid schedule as a packet-by-packet schedule in practice. We show that the discrete schedule is asymptotically close to the fluid schedule, in terms of both fairness and efficiency. We start with a comparison between two typical tracking approaches.

4.1 Start-Time Tracking vs. Finish-Time Tracking

Two common tracking algorithms may be used to implement a fluid schedule in practice, *start-time tracking* and *finish-time tracking*. The former tracks the order of packet start times—among all packets that have already started in the fluid schedule, the one that starts the earliest is scheduled first. Finish-time tracking, on the other hand, assigns the highest scheduling priority to the packet that completes service the earliest in the fluid schedule. In traditional single-resource fair queueing, FQS [16] uses the former approach to track GPS, while WFQ [1,9,21] adopts the latter approach.

While both algorithms closely track the fluid schedule of fair queueing, only start-time tracking is well defined for the tradeoff schedule given by (19). This is due to the fact that, in the tradeoff schedule, future traffic arrivals may lead to a different allocation of packet processing rates and may subsequently change the packet finish times of current packets. As a result, determining the order of finish times requires future traffic arrival information and hence is unrealistic.[3] Start-time tracking avoids this problem as packets are scheduled only *after* they start in the fluid schedule.

For this reason, we use start-time tracking to implement the fluid schedule. We say a discrete schedule and a fluid schedule *correspond* to each other if the former tracks the latter by the packet start time. Specifically, we maintain the fluid schedule in the background. Whenever there is a scheduling opportunity, among all head-of-line packets that have already started in the fluid schedule, the one that starts the earliest is chosen. Below we show that this discrete schedule is *asymptotically close* to its corresponding fluid schedule.

4.2 Performance Analysis

To analyze the performance of start-time tracking, we introduce the following notations. Let $\tau^{\max}$ be the maximum packet processing time required by any packet on any resource. Let n be the maximum number of flows that are *concurrently* backlogged. Let T^{F} be the makespan of the fluid schedule, and T^{D} the makespan of its corresponding discrete schedule.Due to space constraints, all proofs are deferred to our technical report [31].

The following theorem bounds the difference between the makespan of the fluid schedule and its corresponding discrete schedule.

Theorem 4. *For the fluid schedule with $\alpha > 0$ and its corresponding discrete schedule, we have*

$$T^{\mathrm{D}} \leq T^{\mathrm{F}} + n\tau^{\max}. \qquad (25)$$

The error bound $n\tau^{\max}$ can be intuitively explained as the total packet processing time required by all n concurrent flows, each sending only one packet. In practice, the number of packets a flow sends is usually significantly larger than one. As a result, the traffic makespan is significantly larger than the error bound, *i.e.*, $T^{\mathrm{F}} \gg n\tau^{\max}$. Theorem 4 essentially indicates that in terms of makespan, the two schedules are asymptotically close to each other.

We next analyze the fairness performance of the discrete schedule by comparing the dominant services a flow receives under both schedules. In particular, let $D_i^{\mathrm{F}}(0,t)$ be the dominant services flow i receives in $(0,t)$ under the fluid schedule, and $D_i^{\mathrm{D}}(0,t)$ the dominant services flow i receives in $(0,t)$ under the corresponding discrete schedule. The following theorem shows that flows receive approximately the same dominant services under both schedules.

Theorem 5. *For the fluid schedule with $\alpha > 0$ and its corresponding discrete schedule, the following inequality holds for any flow i and any time t:*

$$D_i^{\mathrm{F}}(0,t) - 2(n-1)\tau^{\max} \leq D_i^{\mathrm{D}}(0,t) \leq D_i^{\mathrm{F}}(0,t) + \tau^{\max}. \qquad (26)$$

In other words, the difference between the dominant services a flow receives under the two corresponding schedules is bounded by a constant amount, irrespective of the time t. Over the long run, the discrete schedule achieves the same α-portion fairness as its corresponding fluid schedule. To summarize, start-time tracking retains both the efficiency and fairness properties of its corresponding fluid schedule in the asymptotic regime.

5. AN $O(\log n)$ IMPLEMENTATION

[3]This is not a problem of single-resource fair queueing as different flows are allocated the same processing rate, so that future traffic arrivals will not affect the order of finish times of current packets.

To implement the aforementioned start-time tracking algorithm, two modules are required: packet profiling and fluid scheduling. The former estimates the packet processing time on both CPU and link bandwidth; the latter maintains the fluid schedule as a reference system based on the packet profiling results. We show in this section that packet profiling can be quickly accomplished in $O(1)$ time using a simple approach proposed in [13]. The main challenge comes from the complexity of maintaining the fluid schedule, where direct implementation requires $O(n)$ time. Here, n is the number of backlogged flows. We give an $O(\log n)$ implementation based on an approach similar to virtual time. We shall show in §6 that the implementation can be easily prototyped in the Click modular router [19].

5.1 Packet Profiling

As pointed out by Ghodsi *et al.* [13], any multi-resource fair queueing algorithm, including our fluid schedule, requires knowledge of the packet processing time on each resource. Fortunately, as shown in [13], CPU processing time can be accurately estimated as a linear function of packet size. Specifically, for a packet of size l, the CPU processing time is estimated as $al+b$, where a and b are the coefficients depending on the type of packet processing (*e.g.*, IPsec). We have validated this linear model through an upfront experiment using Click [19]. For each type of packet processing, we measure the exact CPU processing time required by packets of different sizes. This allows us to determine the coefficients a and b. We fit such a linear model to the scheduler and use it to estimate the CPU processing time required by a packet. As for the packet transmission time, the estimation is simply the packet size divided by the outgoing bandwidth, which is known *a priori*.

5.2 Direct Implementation of Fluid Scheduling

Based on the packet profiling results, the fluid schedule is constructed and is maintained by the fluid scheduler. In particular, we need to determine the next packet that starts in the fluid schedule. This requires tracking the work progress of all n flows. Below we give a direct implementation that will be used later in our virtual time implementation.

For each flow i, we record d_i^*, which is the dominant share the flow receives in the fluid schedule at the current time and is computed by (24). We also record R_i, the *remaining dominant processing time* required by the head-of-line packet of the flow at the current time.

For flow i, its head-of-line packet will finish in R_i/d_i^* time if no event occurs then. An *event* is either a *packet departure* or a packet being the *new head-of-line* in the fluid schedule. Either of them may change the head-of-line packet of a flow, leading to different coefficients of the tradeoff problem (19). With d_i^* and R_i, we can accurately track the work progress of flow i in an *event-driven* basis. Specifically, upon the occurrence of an event, let Δt be the time elapsed since the last update. If $\Delta t < R_i/d_i^*$, meaning that the event occurs *before* the head-of-line packet finishes, we update $R_i \leftarrow R_i - d_i^*\Delta t$. If $\Delta t = R_i/d_i^*$, meaning that the event occurs *at the time* when the head-of-line packet finishes, we check if flow i has a next packet p to process. If it does, then packet p becomes the new head-of-line and should start in the fluid schedule. We update R_i as the dominant processing time required by p. Otherwise, we reset $R_i \leftarrow 0$, and flow i leaves the fluid system. We also recompute d_i^* after R_i is updated. (Note that it is impossible to have $\Delta t > R_i/d_i^*$.)

However, purely relying on the approach above to track the work progress of all n flows is highly inefficient. Whenever an event occurs, each flow must be updated individually, which requires at least $O(n)$ time per event and is too expensive. We next introduce a more efficient implementation that requires the above procedure for at most two flows.

5.3 Virtual Time Implementation of Fluid Scheduling

To avoid the high complexity required by the direct implementation above, we have noted, by Theorem 3, that *at most two* flows are favored and are allocated more dominant shares than others. Therefore, it suffices to maintain *at most three* dominant shares at a time—two for the favored flows and one for the other regular flows. For regular flows, we track their work progress using an approach similar to the *virtual time* implementation of GPS [9, 21]. Our intuition is that, by Theorem 3, all the regular flows are allocated the *same* dominant share, and their scheduling *resembles fair queueing*. For favored flows, since there are at most two of them, we track their work progress directly, using the direct implementation above. Our approach is detailed below.

5.3.1 *Identifying Favored and Regular Flows*

We first discuss how favored and regular flows can be quickly identified upon the occurrence of an event. By Theorem 3, it suffices to sort flows in order (23) and examine the three cases. Flows that receive the bonus share (*i.e.*, $\tilde{d}_i^* > 0$) are favored. Note that the entire computation requires only information regarding the head-of-line packet of the first and the last flow in order (23) ($\bar{\tau}_{1,r}$ and $\bar{\tau}_{n,r}$, the normalized dominant processing time). We store all the head-of-line packets in a double-ended priority queue maintained by a *min-max heap* for fast retrieval, where the packet order is defined by (23). This allows us to apply Theorem 3 and identify the favored and regular flows in $O(\log n)$ time.

5.3.2 *Tracking Favored Flows*

For favored flows, because there are at most two of them, we track their work progress using the direct implementation mentioned in §5.2, where we record d_i^* and R_i for each favored flow i. It is easy to see that the update complexity is dominated by the computation of d_i^*. As mentioned in the previous discussion, this can be done in $O(\log n)$ time by Theorem 3. Also, since there are at most two favored flows, the overall tracking complexity remains $O(\log n)$ per event.

5.3.3 *Tracking Regular Flows*

For regular flows, since they receive the same dominant share, their scheduling resembles fair queueing. We hence track their work progress using *virtual time* [1, 9, 21]. Specifically, we define virtual time $V(t)$ as a function of real time t evolving as follows:

$$\begin{aligned} &V(0) = 0, \\ &V'(t) = \alpha \bar{d}^t, \quad t > 0. \end{aligned} \tag{27}$$

Here, $\bar{d}^t$ is the fair dominant share computed by (16) at time t, and is fixed between two consecutive events; $\alpha\bar{d}^t$ is the dominant share each regular flow receives.[4] Thus, V can be interpreted as increasing at the marginal rate at which regular flows receive dominant services. Each regular flow i also maintains *virtual finish time* F_i, indicating the virtual time at which its head-of-line packet finishes in the fluid schedule. The virtual finish time F_i is updated as follows when flow i has a new head-of-line packet p at time t:

$$F_i = V(t) + \tau^*(p), \tag{28}$$

[4]We restore the superscript t here to emphasize that the fair dominant share computed by (16) may change over time.

where $\tau^*(p)$ is the dominant packet processing time required by p. Among all the regular flows, the one with the smallest F_i has its head-of-line packet finishing first in the fluid schedule. Unless some event occurs in between, at time t, the next packet departure for the regular flows would be in $t_N = (\min_i F_i - V(t))/\alpha\bar{d}$ time.

Using virtual time defined by (27), we can accurately track the work progress of regular flows in an event-driven basis. Specifically, upon the occurrence of an event at time t, let t_0 be the time of the last update, and $\Delta t = t - t_0$ the time elapsed since the last update. If $\Delta t < t_N$, meaning that the event occurs before the next packet departure of regular flows, we simply update the virtual time following (27):

$$V(t) = V(t_0) + \alpha\bar{d}\Delta t. \tag{29}$$

If $\Delta t = t_N$, then the event occurs at the time when a packet of a regular flow, say flow i, finishes in the fluid schedule. In addition to updating the virtual time, we check to see if flow i has a next packet p to process. If it does, meaning that the packet p should start in the fluid schedule, we update its virtual finish time F_i following (28). Otherwise, flow i departs the system. We also recompute $\bar{d}$ by (16).

The tracking complexity is dominated by the computation of the minimum virtual finish time, *i.e.*, $\min_i F_i$. By storing F_i's in a priority queue maintained by a heap, we see that the tracking complexity is $O(\log n)$ per event.

5.3.4 *Handling Identity Switching*

We note that the identity of a flow is not fixed: upon the occurrence of an event, a favored flow may switch to a regular flow, and vice versa. We show that such identity switching can also be easily handled in $O(\log n)$ time.

We first consider a favored flow i switching to a regular one at time t, which requires the computation of the virtual finish time F_i. Recall that we have recorded R_i, the remaining dominant processing time required by the head-of-line packet, for flow i as it is previously favored. By definition, the virtual finish time F_i can be simply computed as

$$F_i = V(t) + R_i. \tag{30}$$

Adding F_i to the heap takes at most $O(\log n)$ time.

We next consider a regular flow i switching to a favored one at time t, which requires the computation of R_i. Recall that we have recorded the virtual finish time F_i for flow i. By definition, the remaining dominant processing time required by its head-of-line packet is simply

$$R_i = F_i - V(t), \tag{31}$$

which is a dual of (30).

We also need to remove the virtual finish time, F_i, from the heap. To do so, we maintain an index for each regular flow, recording the location of its virtual finish time stored in the heap. Following this index, we can easily locate the position of F_i and delete it from the heap, followed by some standard "trickle-down" operations to preserve the heap property in $O(\log n)$ time.

To summarize, our approach maintains the fluid schedule by identifying favored and regular flows, tracking their work progress, and handling the potential identity switching. We show that any of these operations can be accomplished in $O(\log n)$ time. As a result, maintaining the fluid schedule takes $O(\log n)$ time per event.

5.4 Start-Time Tracking and Complexity

With the fluid schedule maintained as a reference system, the implementation of start-time tracking is straightforward. Whenever a packet starts in the fluid schedule, it is added to a FIFO queue.

Upon a scheduling opportunity, the scheduler polls the queue and retrieves a packet to schedule. This ensures that packets are scheduled in order of their start times in the fluid schedule. To minimize the update frequency, the scheduler lazily updates the fluid schedule only when the FIFO queue is empty.

We now analyze the scheduling complexity of the aforementioned implementation. The scheduling decisions are made by updating the fluid schedule in an event-driven basis. For each event, the update takes $O(\log n)$ time, where n is the number of backlogged flows. Note that there are only two types of events in the fluid schedule, new head-of-line and packet departure. Because a packet served in the fluid schedule triggers exactly these two events over the entire scheduling period, scheduling N packets triggers $2N$ updates in the fluid schedule, with the overall complexity $O(2N \log n)$. On average, the scheduling decision is made in $O(2 \log n)$ time per packet, the same order as that of DRFQ [13].

6. EVALUATION

We evaluate the tradeoff algorithm via both our prototype implementation and trace-driven simulation. We use a prototype implementation to investigate the detailed functioning of the algorithm, in a microscopic view. We then take a macroscopic view to evaluate the algorithm using trace-driven simulation, where flows dynamically join and depart the system.

6.1 Experimental Results

We have prototyped our tradeoff algorithm as a new scheduler in the Click modular router [19], based on the $O(\log n)$ implementation given in the previous section. The scheduler classifies packets to flows (based on the IP prefix and port number) and identifies the types of packet processing based on the port number specified by a *flow class table*. The scheduler also exposes an interface that allows the operator to dynamically configure the fairness knob α. Our implementation consists of roughly 1,000 lines of C++ code.

We run our Click implementation in user mode on a Dell PowerEdge server with an Intel Xeon 3.0 GHz processor and 1 Gbps Ethernet interface. To make fairness relevant, we throttle the outgoing bandwidth to 200 Mbps while keeping the inbound bandwidth as is. We also throttle the Click module to use only 20% CPU so that CPU could also be a bottleneck. We configure three packet processing modules in Click to emulate a multi-functioning middlebox: packet checking, statistical monitoring, and IPsec. The former two modules are bandwidth-bound, though statistical monitoring requires more CPU processing time than packet checking does. The IPsec module encrypts packets using AES (128-bit key length) and is CPU-bound. We configure another server as a traffic source, initiating 60 UDP flows each sending 2000 800-byte packets per second to the Click router. The first 20 flows pass through the packet checking module; the next 20 flows pass through the statistical monitoring module; and the last 20 flows pass through the IPsec module.

6.1.1 Fairness-Efficiency Tradeoff

We first evaluate the achieved tradeoff between schedule fairness and makespan. To fairly compare the makespan at different fairness levels, it is critical to ensure the same traffic input when running the algorithm with different values of fairness knob α. Therefore, we initially consider an idealized scenario where each flow queue has *infinite* capacity and never drops packets. Table 2 lists the observed makespans with various fairness requirements, in an experiment where each flow keeps sending packets for 10 seconds. We see that, as expected, trading off some level of fairness leads to a shorter makespan and higher efficiency. Furthermore, the marginal improvement of efficiency is decreasing. This suggests that one does not need to compromise too much fairness in order to achieve high efficiency. In our experiment, trading off 15% of fairness shortens the makespan by 15.3% from the strictly fair schedule ($\alpha = 1$), which is equivalent to a 18.1% bandwidth throughput enhancement and is near-optimal as seen in Table 2. Fig. 3 gives a detailed look into the achieved resource utilization over time, at four fairness levels. We see that strictly fair queueing ($\alpha = 1$) wastes 30% of CPU cycles, leaving the bandwidth as the bottleneck at the beginning. This situation remains until bandwidth-bound flows finish, at which time the bottleneck shifts to CPU. By relaxing fairness, CPU-bound flows receive more services, leading to a steady increase of CPU utilization up to 100%. Meanwhile, bandwidth-bound flows experience slightly longer completion times due to the fairness tradeoff.

We now verify the fairness guarantee. We run the scheduler at various fairness levels. At each level, for each flow, we measure its received dominant share *every second* for the first 20 seconds, during which all flows are backlogged. Fig. 4 shows the results, where each cross ("x") corresponds to the dominant share of a flow measured in one second. As expected, under strict fairness ($\alpha = 1$), all flows receive the same dominant share (around 2%). As α decreases, the fairness requirement relaxes. Some flows are hence favored and are allocated more dominant share, while others receive less. However, the minimum dominant share a flow receives is lower bounded by the α-portion of the fair share, shown as the solid line in Fig. 4. This shows that the algorithm is correctly operating at the desired fairness level.

We next extend the experiment to a more practical setup, where each flow queue has a limited capacity and drops packets when it is full. We set the queue size to 200 packets for each flow and repeat the previous experiments. In this case, comparing makespan

Table 2: Schedule makespan observed in Click at different fairness levels. The queue capacity is infinite.

Fairness knob α	Makespan (s)	Normalized Makespan
1.00	55.68	100.00%
0.95	52.50	94.28%
0.90	48.97	87.95%
0.85	47.17	84.72%
0.70	47.13	84.64%
0.60	47.07	84.54%
0.50	47.07	84.54%

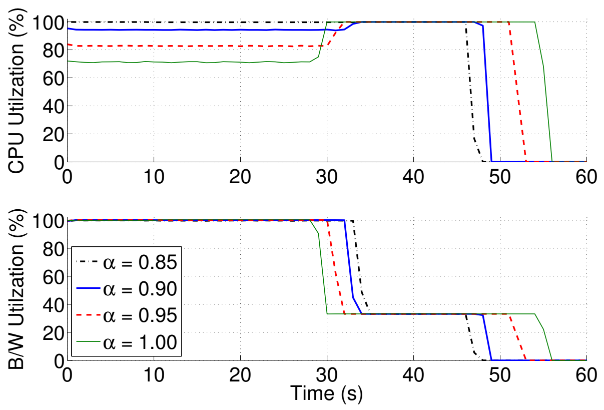


Figure 3: Overall resource utilization observed in Click. No packet drops.

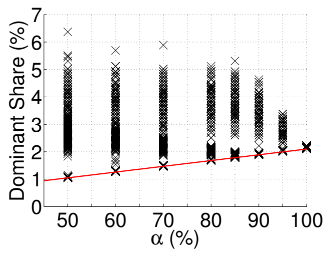
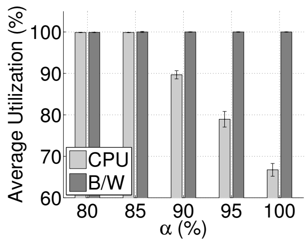
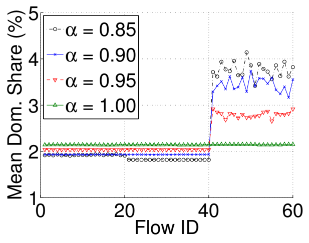


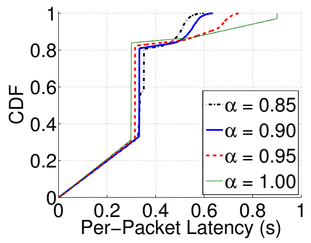
Figure 4: Dominant share each flow receives per second in Click. No packet drops. The strict fair share is 2%.



(a) Resource utilization.



(b) Dominant share.



(c) Per-packet latency

Figure 5: Average resource utilization and dominant share each flow receives in Click at different fairness levels. The queue capacity is 200 packets. The measurement of resource utilization and dominant share is conducted every second over the entire schedule.

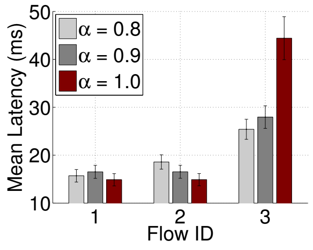
is inappropriate as the scheduler may drop different packets when running at different fairness level. We instead measure the resource utilization achieved *every second* over the entire scheduling period. Fig. 5a illustrates the average utilization of both CPU and bandwidth, where the error bar shows one standard deviation. Similar to the previous experiments, a fairness degradation of 15% is sufficient to achieve the optimal efficiency, enhancing the CPU utilization from 71% to 100%. Further trading off fairness is not well justified. As shown in Fig. 5b, the increased CPU throughput is mainly used to process those CPU-bound flows (Flows 41 to 60), doubling their dominant shares. Meanwhile, the dominant share received by all the other flows is at least 85% of the fair share, as promised by the algorithm. We also depict the per-packet latency CDF in Fig. 5c. We see that trading off fairness for efficiency significantly improves the tail latency, usually caused by flows that finish the last. On the other hand, flows whose shares have been traded off see slightly longer delays of their packets. Fortunately, these latency penalties are strictly bounded—thanks to the fairness guarantee—and are compensated by the significant latency improvement of favored flows.

We have also measured the scheduling overhead in the experiments. In particular, we configure the tradeoff scheduler for strict fair queueing by setting $\alpha = 1$. We then compare the incurred CPU overhead with that of MR^3 [32], a low complexity fair scheduler. Our measurement shows that the tradeoff scheduler introduces 1% CPU overhead compared with MR^3.

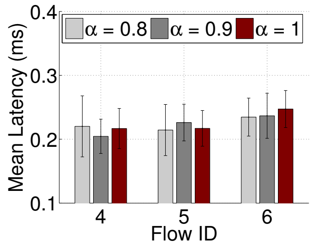
6.1.2 Service Isolation

We next examine the impact of fairness tradeoff on service isolation. We initiate 6 UDP flows sending 800-byte packets. Flows 1 to 3 are *elephant flows*, each sending 20,000 packets per second, and undergo the checking, monitoring, and IPsec modules, respectively. Flows 4 to 6 are *mice flows*, each sending 2 packets per second, and undergo the checking, monitoring, and IPsec modules, respectively. The queue capacity is set to 200 packets. Fig. 6 shows the per-packet latency of each flow at different fairness levels. We see that the tradeoff mainly affects those high-rate flows. For mice flows, even if they may receive less resource share when $\alpha < 1$, the guaranteed share is sufficient to accommodate their low-rate traffic. As a result, their packets are scheduled almost immediately upon arrival, with two orders of magnitude lower latency than the elephant flows.

We also compare our tradeoff scheduler against other fair queueing algorithms. In particular, we have implemented MR^3 [32] and GMR^3 [35] as two other round-robin $O(1)$ schedulers in Click, and conducted the same experiments mentioned above. We find that they achieve almost the same makespan and resource utilization as that of the tradeoff scheduler running with the strict fairness requirement ($\alpha = 1$). This should come with no surprises, as all the existing multi-resource fair queueing algorithms are essentially different approximations to the fluid schedule with perfect DRF. These results are omitted to avoid redundancy.



(a) Elephant flows.



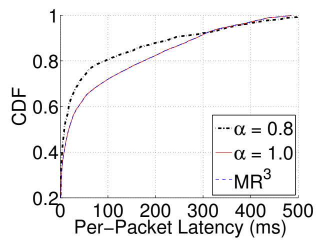
(b) Mice flows.

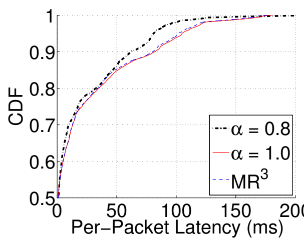
Figure 6: Mean per-packet latency of elephant (sending 20,000 pkts/s) and mice flows (sending 2 pkts/s) in Click. The error bar shows the standard deviation.

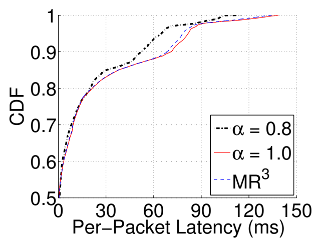
6.2 Trace-Driven Simulation

Next, we use trace-driven simulation to further evaluate the proposed algorithm from a macroscopic perspective. We have written a packet-level simulator consisting of 3,000 lines of C++ code and fed it with real-world traces [2] captured in a university switch. The traces are dominated by UDP packets. Based on the IP prefix and port number, we classify packets in the traces into nearly 3,000 flows and synthesize the input traffic by randomly assigning each flow to one of three middlebox modules: basic forwarding, statistical monitoring, and IPsec. The CPU processing time of each module follows a linear model based on the measurement results of [13]. The flow queue size is set to 200 packets, and the outgoing bandwidth is set to 200 Mbps. We linearly scale up the traffic by 5× to simulate a heavy load. Depending on the total resource consumption, the synthesized traffic is classified into the following three patterns: *CPU-bound* traffic where the CPU processing time exceeds 1.2× the transmission time, *bandwidth-bound* traffic where the transmission time exceeds 1.2× the CPU time, and *balanced* traffic otherwise.

Fig. 7 shows the mean utilization achieved at various fairness levels, where each data point is averaged over 10 runs under the corresponding traffic pattern. The error bar shows one standard deviation. We observe similar trends in all three patterns, that trading off fairness leads to higher utilization on both resources. Similar to our Click implementation results, we see that the marginal


(a) CPU-Bound.


(b) Balanced.


(c) Bandwidth-Bound.

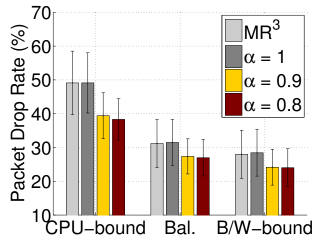
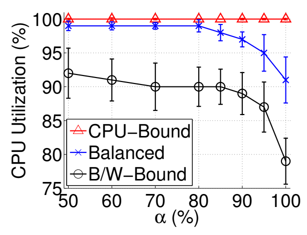

(d) Packet drop rate.

Figure 8: The improvement of per-packet latency and packet drop rate due to the fairness tradeoff.


(a) CPU utilization.

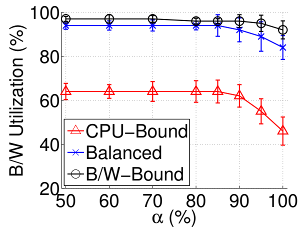
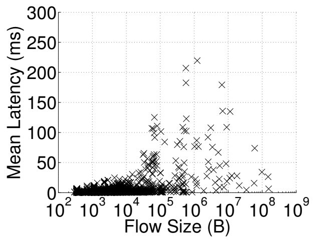

(b) Bandwidth utilization.

Figure 7: Resource utilization achieved at different fairness levels in the simulation, averaged over 10 runs.

improvement in utilization is decreasing, with the optimal utilization achieved by trading off more than 15% of fairness. Among all three patterns, traffic with balanced resource consumption has the least incentive to trade off fairness, as flows have complementary resource demands and can *dove-tail* one another [13]. In this case, fair queueing is sufficient to realize high efficiency. We have also simulated the other two multi-resource fair queueing algorithms, DRFQ [13] and MR^3 [32], and observed almost the same performance as that of the strictly fair queueing ($\alpha = 1$). We omit these results to avoid redundancy.

We examine in Fig. 8 the tradeoff impact on other measures relevant to efficiency. Specifically, we depict the per-packet latency CDF of three scheduling algorithms—tradeoff with $\alpha = 0.8$, complete fairness ($\alpha = 1.0$), and another round-robin fair scheduler called MR^3 [32]—in Figs. 8a, 8b, and 8c, for each of the three traffic patterns. In general, the enhanced resource utilization due to the fairness tradeoff translates into shorter latencies in all three traffic patterns. The improvements are mainly attributed to the shortened packet latency of favored flows. Furthermore, the packet drop rates are compared in Fig. 8d. We observe an average of 15% to 20% decrease in the packet drop rate under all three traffic patterns, suggesting that higher bandwidth throughput is achieved.

Finally, we investigate the tradeoff impact on service isolation in a dynamic environment. Ideally, we would like to see that compromising a small percentage of fairness will not affect the per-packet latency of mice flows as their guaranteed resource share, even when traded off, is sufficient to support their low packet rate. Fig. 9 confirms this isolation property with CPU-bound traffic, where we depict the mean packet latency for each flow in the first 20 seconds of the simulation, running with complete fairness and 80% of fairness, respectively. We see that compared to strictly fair queueing, trading off fairness has no impact on the latency of small flows, but it affects those medium and large ones: Some see shorter latency while others experience longer delay, depending on if they are favored flows or not. We have similar observations in the other two traffic patterns.


(a) $\alpha = 1$.

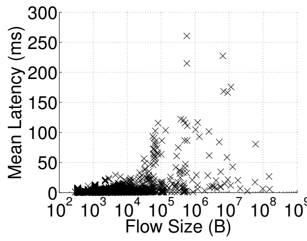

(b) $\alpha = 0.8$.

Figure 9: Per-packet latency against flow sizes in the first 20s of simulation, feeding CPU-bound traffic, with and without the fairness tradeoff.

7. RELATED WORK

Ghodsi *et al.* [13] identified the need of multi-resource fair queueing for deep packet inspection in middleboxes. They compared a set of queueing alternatives and proposed DRFQ, the first multi-resource fair queueing algorithm, that implements DRF in the time domain. Two follow-up queueing algorithms [32, 35] have also been proposed with lower scheduling complexity and bounded scheduling delay. All these works focus solely on fairness, and use the goal of achieving work conservation as the only indication of efficiency, similar to traditional single-resource fair queueing [9, 21].

However, as we have shown in this paper, unlike single-resource scheduling, there is a general tradeoff between fairness and efficiency when packet processing requires multiple types of resources. We have briefly mentioned this problem in our previous position paper [34], where we shared our visions on several possible directions that may lead to a concrete solution. We have materialized some of our visions in this paper by formally characterizing the tradeoff problem and proposing an implementable queueing algorithm to achieve a flexible balance between fairness and efficiency in the two-resource setting.

While the tradeoff problem has received little attention in the fair queueing literature, striking a balance between allocation fairness and efficiency has been a focus of many recent works in both networking and operations research. Specifically, Danna *et al.* [8] have presented an efficient bandwidth allocation algorithm to achieve a flexible tradeoff between fairness and throughput for traffic engineering. Joe-Wong *et al.* [18] have proposed a unifying framework with fairness and efficiency requirements specified by two parameters for a given multi-resource allocation problem. Discussions on the tradeoff between fairness and performance have also been given in the context of P2P networks [11, 38]. In the literature of operations research, Bertsimas *et al.* [3] have derived a tight bound to characterize the efficiency loss under proportional fairness and max-min fairness, respectively. They have later developed a more general framework to characterize the fairness-efficiency tradeoff in a family of "α-fair" welfare functions [4]. All these works focus on one-shot resource allocation in the space domain. In contrast, our focus in this paper is a packet scheduling problem where resources are shared in the time domain.

As explained in §2.4, our tradeoff problem captures the flow shop problem [5–7, 12, 15, 20, 23, 24, 30] as a special case when efficiency is the only concern. Our approach borrows the idea of WFQ [1, 9, 21], in relaxing a discrete packet flow to an idealized fluid and tracking the fluid schedule based on virtual time. However, for single-resource fair queueing, the main challenge is to design a packet-by-packet tracking algorithm, because the fluid schedule, GPS [1, 9, 21], is fairly straightforward and easy to compute. Our problem is more complex, requiring more careful modeling of the fluid schedule, packet-by-packet tracking with multiple resources, and tradeoff analysis.

8. CONCLUSION AND FUTURE WORK

Middleboxes perform complex network functions whose packet processing requires the support of multiple types of hardware resources. A multi-resource packet scheduling algorithm is therefore needed. Unlike traditional single-resource fair queueing, where bandwidth is the only concern, there exists a general tradeoff between fairness and efficiency in the presence of multiple resources. Ideally, we would like to achieve flexible tradeoff to meet QoS requirements while maintaining the system at a high resource utilization level. We show the difficulty of the general problem and limit our discussion to a common scenario where CPU and link bandwidth are the two resources required for packet processing. We propose an efficient scheduling algorithm by tracking an idealized fluid schedule. We show through both our Click implementation and trace-driven simulation that our algorithm achieves a flexible tradeoff between fairness and efficiency in various scenarios.

Despite the initial progress made by this paper, many challenges remain open. First, while optimized, the current implementation requires $O(\log n)$ time per packet scheduling, which may be a concern given a large number of flows. A simpler scheduler with lower complexity may be desired. Given the extensive techniques developed for low-complexity scheduler in the fair queueing literature, it would be interesting to see if and how these techniques extend to the multi-resource setting. Also, as shown by Theorem 1, in general, the more types of resources a system has, the more salient the fairness-efficiency tradeoff would be. For a system with more than two types of resources, we believe the intuition and technique developed in this paper may still be applied. We could define a similar fluid schedule that maximizes the dominant throughput under the specified fairness constraint, and use start-time tracking to implement it in practice.

9. ACKNOWLEDGEMENT

We thank our shepherd, Daniel Menasche, and the anonymous reviewers for their invaluable feedback. This work was partially funded by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

10. REFERENCES

[1] J. Bennett and H. Zhang. WF^{2}Q: Worst-case fair weighted fair queueing. In *Proc. IEEE INFOCOM*, 1996.

[2] T. Benson. Data set for IMC 2010 data center measurement. `http://pages.cs.wisc.edu/~tbenson/IMC_DATA/univ2_trace.tgz`, 2010.

[3] D. Bertsimas, V. F. Farias, and N. Trichakis. The price of fairness. *Oper. Res.*, 59(1):17–31, 2011.

[4] D. Bertsimas, V. F. Farias, and N. Trichakis. On the efficiency-fairness trade-off. *Management Sci.*, 58(12):2234–2250, 2012.

[5] D. Bertsimas and J. Sethuraman. From fluid relaxations to practical algorithms for job shop scheduling: the makespan objective. *Math. Program.*, 92(1):61–102, 2002.

[6] B. Chen, C. N. Potts, and G. J. Woeginger. A review of machine scheduling: Complexity, algorithms and approximability. In D.-Z. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization*, pages 1493–1641. Springer, 1999.

[7] J. G. Dai and G. Weiss. A fluid heuristic for minimizing makespan in job shops. *Oper. Res.*, 50(4):692–707, 2002.

[8] E. Danna, S. Mandal, and A. Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *Proc. IEEE INFOCOM*, 2012.

[9] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proc. ACM SIGCOMM*, 1989.

[10] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Predicting the resource consumption of network intrusion detection systems. In *RAID*, 2008.

[11] B. Fan, D.-M. Chiu, and J. C. Lui. The delicate tradeoffs in bittorrent-like file sharing protocol design. In *Proc. IEEE ICNP*, 2006.

[12] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Math. Oper. Res.*, 1(2):117–129, 1976.

[13] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. In *Proc. ACM SIGCOMM*, 2012.

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

[15] T. Gonzalez and S. Sahni. Flowshop and jobshop schedules: Complexity and approximation. *Oper. Res.*, 26(1):26–52, 1978.

[16] A. Greenberg and N. Madras. How fair is fair queuing. *J. ACM*, 39(3):568–598, 1992.

[17] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[18] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *Proc. IEEE INFOCOM*, 2012.

[19] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Trans. Comput. Sys.*, 18(3):263–297, 2000.

[20] J. Y. Leung. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, 2004.

[21] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Trans. Netw.*, 1(3):344–357, 1993.

[22] D. Parkes, A. Procaccia, and N. Shah. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. In *Proc. ACM EC*, 2012.

[23] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 2012.

[24] S. S. Seiden. A guessing game and randomized online algorithms. In *Proc. ACM STOC*, 2000.

[25] V. Sekar, N. Egi, S. Ratnasamy, M. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proc. USENIX NSDI*, 2012.

[26] J. Sgall. On-line scheduling. In A. Fiat and G. J. Woeginger, editors, *Online Algorithms*. Springer, 1998.

[27] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. ACM SIGCOMM*, 2012.

[28] M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round-robin. *IEEE/ACM Trans. Netw.*, 4(3):375–385, 1996.

[29] J. B. Sidney. The two-machine maximum flow time problem with series parallel precedence relations. *Oper. Res.*, 27(4):782–791, 1979.

[30] A. P. A. Vestjens. *On-line Machine Scheduling*. PhD thesis, Technische Universiteit Eindhoven, 1997.

[31] W. Wang, C. Feng, B. Li, and B. Liang. On the fairness-efficiency tradeoff for packet processing with multiple resources. Technical report, University of Toronto, 2014. `http://iqua.ece.toronto.edu/~bli/papers/tradeoff.pdf`.

[32] W. Wang, B. Li, and B. Liang. Multi-resource round robin: A low complexity packet scheduler with dominant resource fairness. In *Proc. IEEE ICNP*, 2013.

[33] W. Wang, B. Liang, and B. Li. Multi-resource generalized processor sharing for packet processing. In *Proc. ACM/IEEE IWQoS*, 2013.

[34] W. Wang, B. Liang, and B. Li. On fairness-efficiency tradeoffs for multi-resource packet processing. In *Proc. IEEE ICDCS Workshop on Data Center Performance (DCPerf)*, 2013.

[35] W. Wang, B. Liang, and B. Li. Low complexity multi-resource fair queueing with bounded delay. In *Proc. IEEE INFOCOM*, 2014.

[36] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In *Proc. SIGCOMM*, 2011.

[37] W. Whitt. Understanding the efficiency of multi-server service systems. *Management Sci.*, 38(5):708–723, 1992.

[38] B. Zhang, S. C. Borst, and M. I. Reiman. Optimal server scheduling in hybrid P2P networks. *Perform. Eval.*, 67(11), 2010.

[39] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. *Proc. IEEE*, 83(10):1374–1396, 1995.