Multi-Resource Fair Queueing for Packet Processing

Ali Ghodsi
Vyas Sekar
Matei Zaharia
Ion Stoica

June 19, 2012

Acknowledgement

We thank Adam Oliner, Ganesh Ananthanarayanan, and Patrick Wendell for useful feedback on earlier drafts of this paper. This research is supported in part by NSF FIA Award #CNS-1038695, NSF CISE Expeditions award CCF-1139158, a Google PhD Fellowship, gifts from AmazonWeb Services, Google, SAP, Blue Goji, Cisco, Cloudera, Ericsson, General Electric, Hewlett Packard, Huawei, Intel, MarkLogic, Microsoft, NetApp, Oracle, Quanta, Splunk, VMware and by DARPA (contract #FA8650-11-C-7136).

Multi-Resource Fair Queueing for Packet Processing

Ali Ghodsi
alig@cs.berkeley.edu
U.C. Berkeley
KTH—Royal Institute of Technology

Vyas Sekar
vyas.sekar@gmail.com
Intel ISTC, Berkeley

Matei Zaharia
matei@cs.berkeley.edu
U.C. Berkeley

Ion Stoica
istoica@cs.berkeley.edu
U.C. Berkeley

June 19, 2012

Abstract

Middleboxes are ubiquitous in today's networks and perform a variety of important functions, including IDS, VPN, firewalling, and WAN optimization. These functions differ vastly in their requirements for hardware resources (*e.g.,* CPU cycles and memory bandwidth). Thus, depending on the functions they go through, different flows can consume different amounts of a middlebox's resources. While there is much literature on weighted fair sharing of link bandwidth to isolate flows, it is unclear how to schedule *multiple* resources in a middlebox to achieve similar guarantees. In this paper, we analyze several natural packet scheduling algorithms for multiple resources and show that they have undesirable properties. We propose a new algorithm, Dominant Resource Fair Queuing (DRFQ), that retains the attractive properties that fair sharing provides for one resource. In doing so, we generalize the concept of virtual time in classical fair queuing to multi-resource settings. The resulting algorithm is also applicable in other contexts where several resources need to be multiplexed in the time domain.

1 Introduction

Middleboxes today are omnipresent. Surveys show that the number of middleboxes in companies is on par with the number of routers and switches [28]. These middleboxes perform a variety of functions, ranging from firewalling and IDS to WAN optimization and HTTP caching. Moreover, the boundary between routers and middleboxes is blurring, with more middlebox functions being incorporated into hardware and software routers [2, 6, 1, 27].

Given that the volume of traffic through middleboxes is increasing [20, 32] and that middlebox processing functions are often expensive, it is important to schedule the hardware resources in these devices to provide predictable isolation across flows. While packet scheduling has been studied extensively in routers to allocate link bandwidth [24, 11, 29], middleboxes complicate the scheduling problem because they have *multiple* resources that can be congested. Different middlebox processing functions consume vastly different amounts of these resources. For example, intrusion detection functionality is often CPU-bound [14], software routers bottleneck on memory bandwidth when processing small packets [8], and forwarding of large packets with little processing can bottleneck on link bandwidth. Thus, depending on the processing needs of the flows going through it, a middlebox will need to make scheduling decisions across multiple resources. This becomes more important as middlebox resource diversity increases (*e.g.,* GPUs [30] and specialized hardware acceleration [23, 5]).

Traditionally, for a single resource, weighted fair sharing [11] ensures that flows are isolated from each other by making *share guarantees* on how much bandwidth each flow gets [24]. Furthermore, fair sharing is *strategy-proof*, in that flows cannot get better service by artificially inflating their resource consumption. Many algorithms, such as WFQ [11], GPS [24], DRR [29], and SFQ [18], have been proposed to approximate fair sharing through discrete packet scheduling decisions, but they all retain the properties of share guarantees and strategy-proofness. We would like a multi-resource scheduler to also provide these properties.

Share guarantees and strategy-proofness, while almost trivial for one resource, turn out to be non-trivial for multiple resources [16]. We first analyze two natural scheduling schemes and show that they lack these properties. The first scheme is to monitor the resource usage of the system, determine which resource is currently the bottleneck, and divide it fairly between the flows [15]. Unfortunately, this approach lacks both desired properties. First, it is not strategy-proof; a flow can manipulate the scheduler to get better service by artificially *increasing* the amount of resources its packets use. For example, a flow can use smaller packets, which increase the CPU usage of the middlebox, to shift the bottleneck resource from network bandwidth to CPU. We show that this can double the manipulative flow's throughput while hurting other flows. Second, when multiple resources can simultaneously be bottlenecked, this solution can lead to oscillations that substantially lower the total throughput and keep some flows below their guaranteed share.

A second natural scheme, which can happen by default in software router designs, is to perform fair sharing independently at each resource. For example, packets might first be processed by the CPU, which is shared via stride scheduling [31], and then go into an output link buffer served via fair queuing. Surprisingly, we show that even though fair sharing for a single resource is strategy-proof, composing per-resource fair schedulers this way is not.

Recently, a multi-resource allocation scheme that provides share guarantees and strategy-proofness, called Dominant Resource Fairness (DRF) [16], was proposed. We design a fair queueing algorithm for multiple resources that achieves DRF allocations.

The main challenge we address is that existing algorithms for DRF provide fair sharing in *space*; given a cluster with much larger number of servers than users, they decide how many resources each user should get on each server. In contrast, middleboxes require sharing in *time*; given a small number of resources (*e.g.,* NICs or CPUs) that can each process only one packet at a time, the scheduler must interleave packets to achieve the right resource shares over time. Achieving DRF allocations in time is challenging, especially doing so in a memoryless manner, *i.e.,* a flow should not be penalized for having had a high resource share in the past when fewer flows were active [24]. This memoryless property is key to guaranteeing that flows cannot be starved in a work-conserving system.

We design a new queuing algorithm called Dominant Resource Fair Queuing (DRFQ), which generalizes the concept of virtual time from classical fair queuing [11, 24] to multiple resources that are consumed at different rates over time. We evaluate DRFQ using a Click [22] implementation and simulations, and we show that it provides better isolation and throughput than existing schemes.

To summarize, our contributions in this work are three-fold:

1. We identify the problem of multi-resource *fair queueing*, which is a generalization of traditional single-resource fair queueing.
2. We provide the first analysis of two natural packet scheduling schemes—bottleneck fairness and per-resource fairness—and show that they suffer from problems including poor isolation, oscillations, and manipulation.
3. We propose the first multi-resource queuing algorithm that provides both share guarantees and strategy-proofness: Dominant Resource Fair Queuing (DRFQ). DRFQ implements DRF allocations in the time domain.

2 Motivation

Others have observed that middleboxes and software routers can bottleneck on any of CPU, memory bandwidth, and link bandwidth, depending on the processing requirements of the traffic. Dreger *et al.* report that CPU can be a bottleneck in the Bro intrusion detection system [14]. They demonstrated that, at times, the CPU can be overloaded to the extent that each second of incoming traffic requires 2.5 seconds of CPU processing. Argyraki *et al.* [8] found that memory bandwidth can be a bottleneck in software routers, especially when processing small packets. Finally, link bandwidth can clearly be a bottleneck for flows that need no processing. For example, many middleboxes let encrypted SSL flows pass through without processing.

To confirm and quantify these observations, we measured the resource footprints of several canonical middlebox applications implemented in Click [22]. We developed a trace generator that takes in real traces with full payloads [4] and analyzes the resource consumption of Click modules using the Intel(R) Performance Counter Monitor API [3]. Figure 1 shows the results for four applications. Each application's maximum resource consumption was normalized to 1. We see that the resource consumption varies across modules: basic forwarding uses a higher relative fraction of link bandwidth than of other resources, redundancy elimination bottlenecks on mem-

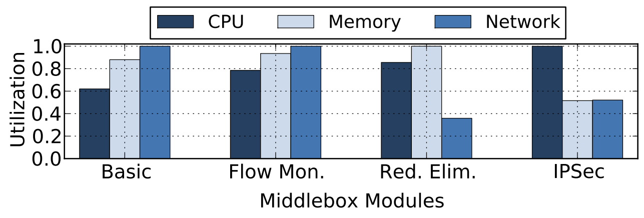


Figure 1: Normalized resource usage of four middlebox functions implemented in Click: basic forwarding, flow monitoring, redundancy elimination, and IPSec encryption.

ory bandwidth, and IPSec encryption is CPU-bound.

Many middleboxes already perform different functions for different traffic (*e.g.,* HTTP caching for some flows and basic forwarding for others), and future software-defined middlebox proposals suggest consolidating more functions onto the same device [28, 27]. Moreover, further functionality is being incorporated into hardware accelerators [30, 23, 5], increasing the resource diversity of middleboxes. Thus, packet schedulers for middleboxes will need to take into account flows' consumption across multiple resources.

Finally, we believe multi-resource scheduling to be important in other contexts too. One such example is multi-tenant scheduling in deep software stacks. For example, a distributed key-value store might be layered on top of a distributed file system, which in turn runs over the OS file system. Different layers in this stack can bottleneck on different resources, and it is desirable to isolate the resource consumption of different tenants' requests. Another example is virtual machine (VM) scheduling inside a hypervisor. Different VMs might consume different resources, so it is desirable to fairly multiplex their access to physical resources.

3 Background

Designing a packet scheduler for multiple resources turns out to be non-trivial due to several problems that do not occur with one resource [16]. In this section, we review these problems and provide background on the allocation scheme we ultimately build on, DRF. In addition, given that our goal is to design a packet queuing algorithm that achieves DRF, we cover background on fair queuing.

3.1 Challenges in Multi-Resource Scheduling

Previous work on DRF identifies several problems that can occur in multi-resource scheduling and shows that several simple scheduling schemes lack key properties [16].

Share Guarantee: The essential property of fair queuing is *isolation*. Fair queuing ensures that each of n flows can get a guaranteed $\frac{1}{n}$ fraction of a resource (*e.g.,* link

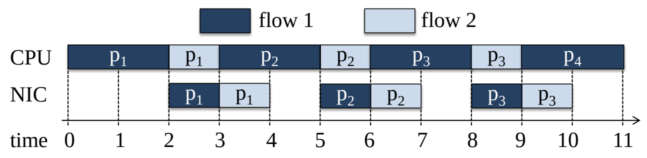


Figure 2: Performing fair sharing based on a single resource (NIC) fails to meet the share guarantee. In the steady-state period from time 2–11, flow 2 only gets a third of each resource.

bandwidth), regardless of the demand of other flows [24].[1] Weighted fair queuing generalizes this concept by assigning a weight w_i to each flow and guaranteeing that flow i can get at least $\frac{w_i}{\sum_{j \in W} w_j}$ of the sole resource, where W is the set of active flows.

We generalize this guarantee to multiple resources as follows:

Share Guarantee. A backlogged flow with weight w_i should get at least $\frac{w_i}{\sum_{j \in W} w_j}$ fraction of one of the resources it uses.

Surprisingly, this property is not met by some natural schedulers. As a strawman, consider a scheduler that only performs fair queueing based on one specific resource. This may lead to some flows receiving less than $\frac{1}{n}$ of all resources, where n is the total number of flows. As an example, assume that there are two resources, CPU and link bandwidth, and that each packet first goes through a module that uses the CPU, and thereafter is sent to the NIC. Assume we have two flows with *resource profiles* $\langle 2, 1 \rangle$ and $\langle 1, 1 \rangle$; that is, packets from flow 1 each take 2 time units to be processed by the CPU and 1 time unit to be sent on the link, while packets from flow 2 take 1 unit of both resources. If the system implements fair queuing based on only link bandwidth, it will alternate sending one packet from each flow, resulting in equal allocation of link bandwidth to the flows (both flows use one time unit of link bandwidth). However, since there is more overall demand for the CPU, the CPU will be fully utilized, while the network link will be underutilized at times. As a result (see Figure 2), the first flow receives $\frac{2}{3}$ and $\frac{1}{3}$ of the two resources, respectively. But the second flow only gets $\frac{1}{3}$ on both resources, violating the share guarantee.

Strategy-Proofness: The multi-resource setting is vulnerable to a new type of *manipulation*. Flows can manipulate the scheduler to receive better service by artificially inflating their demand for resources they do not need.[2]

For example, a flow might increase the CPU time required to process it by sending smaller packets. Depending on the scheduler, such manipulation can increase the flow's allocation across *all* resources. We later show that in several natural schedulers, greedy flows can as much as double their share at the cost of other flows.

[1]By "flow," we mean a set of packets defined by a subset of header fields. Administrators can choose which fields to use based on organizational policies, *e.g.,* to enforce weighted fair shares across users (based on IP addresses) or applications (based on ports).

[2]Note that there are manipulations that cannot be prevented at the scheduler level, such as a user using many different flows. Such cases must be mitigated through mechanisms orthogonal to the scheduler [10].

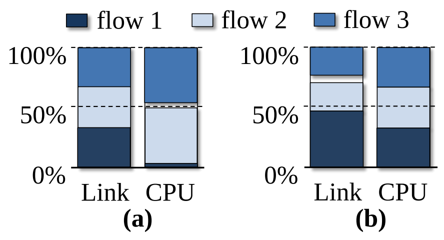


Figure 3: Bottleneck fairness can be manipulated by users. In (b), flow 1 increases its CPU usage per packet to shift the bottleneck to CPU, and thereby gets more bandwidth too.

These types of manipulations were not possible in single-resource settings, and therefore received no attention in past literature. It is important for multi-resource schedulers to be resistant to them, as a system vulnerable to manipulation can incentivize users to *waste* resources, ultimately leading to lower total goodput.

The following property discourages the above manipulations:

Strategy-proofness. A flow should not be able to finish faster by increasing the amount of resources required to process it.

As a concrete example, consider the scheduling scheme proposed by Egi *et al.* [15], in which the middlebox determines which resource is bottlenecked and divides that resource evenly between the flows. We refer to this approach as *bottleneck fairness*. Figure 3 shows how a flow can manipulate its share by wasting resources. In (a), there are three flows with resource profiles $\langle 10, 1\rangle$, $\langle 10, 14\rangle$ and $\langle 10, 14\rangle$ respectively. The bottleneck is the first resource (link bandwidth), so it is divided fairly, resulting in each flow getting one third of it. In (b), flow 1 increases its resource profile from $\langle 10, 1\rangle$ to $\langle 10, 7\rangle$. This shifts the bottleneck to the CPU, so the system starts to schedule packets to equalize the flows' CPU usage. However, this gives flow 1 a higher share of bandwidth as well, up from $\frac{1}{3}$ to almost $\frac{1}{2}$. In similar examples with more flows, flow 1 can almost double its share.

We believe the networking domain to be particularly prone to these types of manipulations, as peer-to-peer applications already employ various techniques to increase their resource share [26]. Such an application could, for instance, dynamically adapt outgoing packet sizes based on throughput gain, affecting the CPU consumption of congested middleboxes.

3.2 Dominant Resource Fairness (DRF)

The recently proposed DRF allocation policy [16] achieves both strategy-proofness and share guarantees.

DRF was designed for the datacenter environment, which we briefly recapitulate. In this setting, the equivalent of a flow is a job, and the equivalent of a packet is a job's task, executing on a single machine. DRF defines the *dominant resource* of a job to be the resource that it currently has the biggest share of. For example, if a

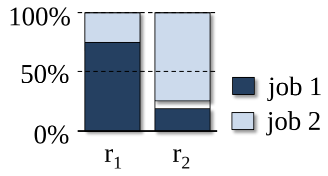


Figure 4: DRF allocation for jobs with resource profiles $\langle 4, 1\rangle$ and $\langle 1, 3\rangle$ in a system with equal amounts of both resources. Both jobs get $\frac{3}{4}$ of their dominant resource.

job has 20 CPUs and 10 GB of memory in a cluster with 100 CPUs and 40 GB of memory, the job's dominant resource is memory, as it is allocated $\frac{1}{4}$ of it (compared to $\frac{1}{5}$ for CPU). A job's *dominant share* is simply its share of its dominant resource, *e.g.,* $\frac{1}{4}$ in this example. Informally, DRF provides the allocation that "equalizes" the dominant shares of different users. More precisely, DRF is the max-min fair allocation of dominant shares.

Figure 4 shows an example, where two jobs run tasks with resource profiles $\langle 4\text{ CPUs}, 1\text{ GB}\rangle$ and $\langle 1\text{ CPU}, 3\text{ GB}\rangle$ in a cluster with 2000 CPUs and 2000 GB of memory. In this case, job 1's dominant resource is CPU, and job 2's dominant resource is memory. DRF allocates $\langle 1500\text{ CPUs}, 375\text{ GB}\rangle$ of resources to job 1 and $\langle 500\text{ CPUs}, 1500\text{ GB}\rangle$ to job 2. This equalizes job 1's and job 2's dominant shares while maximizing the allocations.

We have described the DRF *allocation*. Ghodsi *et al.* [16] provide a simple *algorithm* to achieve DRF allocations in space (*i.e.,* given a cluster of machines, compute which resources on which machines to assign to each user). We seek an algorithm that achieves DRF allocations in *time*, multiplexing resources across incoming packets. In Section 5, we describe this problem and provide a queuing algorithm for DRF. The algorithm builds on concepts from fair queuing, which we review next.

3.3 Fair Queuing in Routers

Fair Queuing (FQ) aims to implement max-min fair allocation of a single resource using a fluid-flow model, in which the link capacity is infinitesimally divided across the backlogged flows [11, 24]. In particular, FQ schedules packets in the order in which they would finish in the fluid-flow system.

Virtual clock [33] was one of the first schemes using a fluid-flow model. It, however, suffers from the problem that it can punish a flow that in the past got better service when fewer flows were active. Thus, it violates the following key property:

Memoryless scheduling. A flow's current share of resources should be independent of its share in the past.

In the absence of this memoryless property, flows may experience starvation. For example, with virtual clock, if one flow uses a link at full rate for one minute, and a second flow becomes active, then only the second flow is serviced for the next minute, until their virtual clocks equalize. Thus, the first flow starves for a minute.[3]

The concept of *virtual time* was proposed to address this pitfall [24]. Instead of measuring real time, virtual time measures the amount of work performed by the system. Informally, a virtual time unit is the time it takes to send *one* bit of a *unit*-weight flow in the fluid-flow system. Thus, it takes l virtual time units to send a packet of length l. Thus, virtual time progresses faster than real-time when fewer flows are active. In general, assuming a flow with weight w, it takes l/w virtual time units to send the packet in the fluid-flow system.

Virtual time turns out to be expensive to compute exactly, so a variety of algorithms have been proposed to implement FQ efficiently by approximating it [11, 24, 18, 29, 9]. One of the main algorithmic challenges we address in our work is to extend this concept to multiple resources that are consumed at different rates over time.

4 Analysis of Existing Policies

We initially explored two natural scheduling algorithms for middleboxes. The first solution, called *bottleneck fairness*, turns out to lack both strategy-proofness and the sharing guarantee. The second, called *per-resource fairness*, performs fair sharing independently at each resource. This would happen naturally in routers that queue packets as they pass between different resources and serve each queue via fair sharing. We initially pursued per-resource fairness but soon discovered that it is not strategy-proof.

4.1 Bottleneck Fairness

In early work on resource scheduling for software routers, Egi *et al.* [15] point out that most of the time, only one resource is congested. They therefore suggest that the system should dynamically determine which resource is congested and perform fair sharing on that resource. For example, a middlebox might place new packets from each flow into a separate queue and serve these queues based on the packets' estimated CPU usage if CPU is a bottleneck, their memory bandwidth usage if memory is a bottleneck, etc.

This approach has several disadvantages. First, it is not strategy-proof. As we showed in Section 3.1, a flow can nearly double its share by artificially increasing its resource consumption to shift the bottleneck.

Second, when neither resource is a clear bottleneck, bottleneck fairness can rapidly oscillate, affecting the throughput of all flows and keeping some flows below their share guarantee. This can happen readily in middleboxes where some flows require expensive processing and some do not. For example, consider a middlebox with two resources, CPU and link bandwidth, that applies IPsec encryption to flows within a corporate VPN but forwards other traffic to the Internet. Suppose that an external flow has a resource profile of $\langle 1, 6 \rangle$ (bottlenecking on bandwidth), while an internal flow has $\langle 7, 1 \rangle$. If both flows are backlogged, it is unclear which resource should be considered the bottleneck.

[3]A workaround for this problem would be for the first flow to never use more than half of the link capacity. However, this leads to inefficient resource utilization during the first minute.

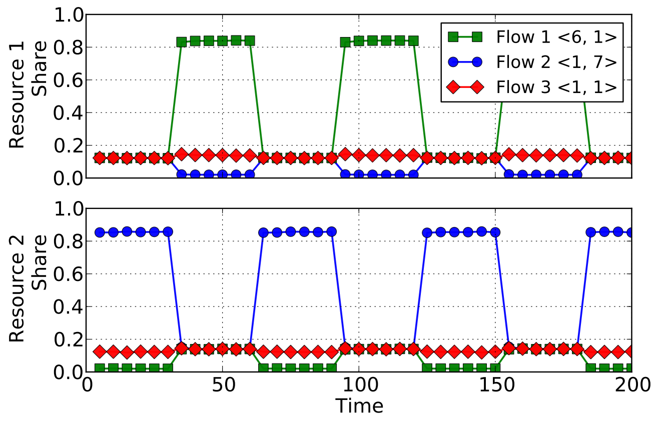


Figure 5: Example of oscillation in bottleneck fairness. Note that flow 3 stays below $\frac{1}{3}$ share of both resources.

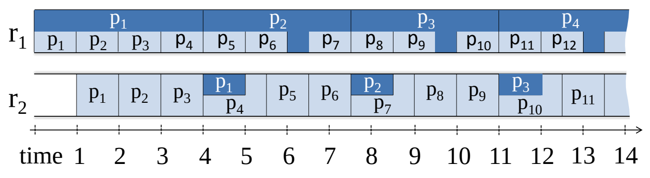
Indeed, assume the system decides that the first resource is the bottleneck and tries to divide it evenly between the flows. As a result, the first resource will process seven packets of flow 1 for every single packet of flow 2. Unfortunately, this will congest the second resource right away, since processing seven packets of flow 1 and one packet of flow 2 will generate a higher demand for resource 2 than resource 1, *i.e.,* $7\langle 1, 6\rangle + \langle 7, 1\rangle = \langle 14, 43\rangle$. Once resource 2 becomes the bottleneck, the system will try to divide this resource equally. As a result, resource 2 will process six packets of flow 2 for each packet of flow 1, which yields an overall demand of $\langle 1, 6\rangle + 6\langle 7, 1\rangle = \langle 43, 12\rangle$. This will now congest resource 1, and the process will repeat.

Such oscillation is a problem for TCP traffic, where fast changes in available bandwidth leads to bursts of losses and low throughput. However, bottleneck fairness also fails to meet share guarantees for non-TCP flows. For example, if we add a third flow with resource profile $\langle 1, 1\rangle$, bottleneck fairness always keeps its share of *both* resources below $\frac{1}{3}$, as shown in Figure 5. This is because there is no way, while scheduling based on one resource, to increase all the flows' share of that resource to $\frac{1}{3}$ before the other gets congested.

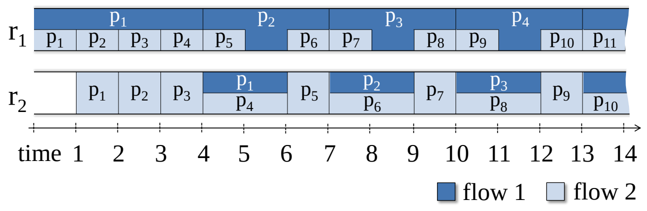
4.2 Per-Resource Fairness (PF)

A second intuitive approach is to perform fair sharing independently at each resource. For example, suppose that incoming packets pass through two resources: a CPU, which processes these packets and then an output link. Then one could first schedule packets to pass through the CPU in a way that equalizes flows' CPU shares, by performing fair queuing based on packets' processing times, and then place the packets into buffers in front of the output link that get served based on fair sharing of bandwidth.

Although this approach is simple, we found that it is not strategy-proof. For example, Figure 6(a) shows two flows with resource profiles $\langle 4, 1\rangle$ and $\langle 1, 2\rangle$ that share two resources. The labels of the packets show when each packet uses each resource.



(a) Allocation with resource profiles $\langle 4, 1\rangle$ and $\langle 1, 2\rangle$.



(b) Allocation with resource profiles $\langle 4, 2\rangle$ and $\langle 1, 2\rangle$.

Figure 6: Example of how flows can manipulate per-resource fairness. A shaded box shows the consumption of one packet on one resource. In (b), flow 1 increases per-packet resource use from $\langle 4, 1\rangle$ to $\langle 4, 2\rangle$ to get a higher share of resource 1 ($\frac{2}{3}$ as opposed to $\frac{4}{7}$).

For simplicity, we assume that the resources are perfectly divisible, so both flows can use a resource simultaneously. Furthermore, we assume that the second resource can start processing a packet only after the first one has finished it, and that there is only a 1-packet buffer for each flow between the resources. As shown in Figure 6(a), after the initial start, a periodic pattern with a length of 7 time units emerges. As a result, flow 1 gets resource shares $\langle \frac{4}{7}, \frac{1}{7}\rangle$, *i.e.,* it gets $\frac{4}{7}$ of the first resource and $\frac{1}{7}$ of the second resource. Meanwhile, flow 2 gets resource shares $\langle \frac{3}{7}, \frac{6}{7}\rangle$.

Suppose flow 1 artificially increases its resource consumption to $\langle 4, 2\rangle$. Then per-resource fair queuing gives the allocation in Figure 6(b), where flow 1's share is $\langle \frac{2}{3}, \frac{1}{3}\rangle$ and flow 2's share is $\langle \frac{1}{3}, \frac{2}{3}\rangle$. Flow 1 has thus increased its share of the first resource by 16%, while decreasing flow 2's share of this resource by 22%.

This behavior surprised us, because fair queuing for a single resource *is* strategy-proof. Intuitively, flow 1 "crowds out" flow 2 at the second resource, which is the primary resource that flow 2 needs, by increasing its share, and this causes the buffer for flow 2's packets between the two resources to be full more of the time. This leaves more time for flow 1 at the first resource.

We found that the amount by which flows can raise their share with per-resource fairness is as high as $2\times$, which provides substantial incentive for applications to manipulate the scheduler. We discuss an example in Section 8.2.1. We have also simulated other models of per-resource fairness, including ones with bigger buffers and ones that let multiple packets be processed in parallel (*e.g.,* as on a multicore CPU), but found that they give the same shares over time and can be manipulated in the same way.

Finally, from a practical viewpoint, per-resource fairness is hard to implement in systems where there is no buffer between the resources, *e.g.,* a system where scheduling decisions are only taken at an input queue, or a processing function that consumes CPU and memory bandwidth in parallel (while executing CPU instructions). Our proposal, DRFQ, is directly applicable in these settings.

5 Dominant Resource Fair Queuing

The goal of this section is to develop queuing mechanisms that multiplex packets to achieve DRF allocations.

Achieving DRF allocations in middleboxes turns out to be algorithmically more challenging than in the datacenter context. In datacenters, there are many more resources (machines, CPUs, etc) than active jobs, and one can simply divide the resources across the jobs according to their DRF allocations at a given time. In a packet system, this is not possible because the number of packets in service at a given time is usually much smaller than the number of backlogged flows. For example, on a communication link, at most one packet is transmitted at a time, and on a CPU, at most one packet is (typically) processed per core. Thus, the only way to achieve DRF allocation is to share the resources in time instead of space, *i.e.,* multiplex packets from different flows to achieve the DRF allocation over a longer time interval.

This challenge should come as no surprise to networking researchers, as scheduling a single resource (link bandwidth) in time was a research challenge receiving considerable attention for years. Efforts to address this challenge started with idealized models (*e.g.,* fluid flow [11, 24]), followed by a plethora of algorithms to accurately and efficiently approximate these models [29, 18, 9].

We begin by describing a unifying model that accounts for resource consumption across different resources.

5.1 Packet Processing Time

The mechanisms we develop generalize the fair queueing concepts of virtual start and finish times to multiple resources, such that these times can be used to schedule packets.

To do this, we first find a metric that lets us compare virtual times across resources. Defining the unit of virtual time as the time it takes to process one bit does not work for all resource types. For example, the CPU does not consume the same amount of time to process each bit; it usually takes longer to process a packet's header than its payload. Furthermore, packets with the same size, but belonging to different flows, may consume different amounts of resources based on the processing functions they go through. For example, a packet that gets handled by the IPSec encryption module will consume more CPU time than a packet that does not.

To circumvent these challenges, we introduce the concept of *packet processing time*. Denote the k:th packet of flow i as p_i^k. The processing time of the packet p_i^k at resource j, denoted $s_{i,j}^k$, is the time consumed by resource j to process packet p_i^k, normalized to the resource's processing capacity. Note that processing time is not always equal to packet service time.[4] Consider a CPU with four cores, and assume it takes a single core 10 μs to process a packet. Since the CPU can process four such

[4]Packet service time is the interval between (1) the time the packet starts being processed and (2) the time at which its processing ends.

packets in parallel, the normalized time consumed by the CPU to process the packet (*i.e.,* the processing time) is 2.5 μs. However, the packet's service time is 10 μs. In general, the processing time is the inverse of the throughput. In the above example, the CPU throughput is $400,000$ packets/sec.

We define a unit of virtual time as one μsec of processing time for the packet of a flow with weight one. Thus, by definition, the processing time, and by extension, the virtual time, do not depend on the resource type. Also, similarly to FQ, the unit of virtual time does not depend on number of flows backlogged. Here, we assume that the time consumed by a resource to process a packet does not depend on how many other packets, if any, it processes in parallel.

We return to processing time estimation in §5.7 and §7.1.

5.2 Dove-Tailing vs. Memoryless Scheduling

The multi-resource scenario introduces another challenge. Specifically, there is a trade-off between *dove-tailing* and *memoryless scheduling*, which we explain next.

Different packets from the same flow may have different processing time requirements, *e.g.,* a TCP SYN packet usually requires more processing time than later packets. Consider a flow that sends a total of 10 packets, alternating in processing time requirements $\langle 2, 1\rangle$ and $\langle 1, 2\rangle$, respectively. It is desirable that the system treats this flow the same as a flow that sends 5 packets, all with processing time $\langle 3, 3\rangle$. We refer to this as the *dove-tailing* requirement.[5]

Our dove-tailing requirement is a natural extension of fair queuing for one resource. Indeed, past research in network fair queuing attempted to normalize the processing time of packets of different length. For example, a flow with 5 packets of length 1 KB should be treated the same as a flow with 10 packets of length 0.5 KB.

At the same time, it is desirable for a queuing discipline to be *memoryless*; that is, a flow's current share of resources should not depend on its past share. Limiting memory is important to prevent starving flows when new flows enter the system, as discussed in Section 3.3.

Unfortunately, the memoryless and dove-tailing properties cannot both be *fully* achieved at the same time. Dove-tailing requires that a flow's relative overconsumption of a resource be compensated by its *past* relative underconsumption of a resource, *e.g.,* packets with profile $\langle 1, 2\rangle$ and $\langle 2, 1\rangle$. Thus, it requires the scheduler to have memory of past processing time given to a flow.

Memoryless and dove-tailing are at the extreme ends of a spectrum. We gradually develop DRFQ, starting with a simple algorithm that is fully memoryless, but does not provide dove-tailing. We thereafter extend that algorithm to provide full dove-tailing but without being memoryless. Finally, we show a final extension in which the amount of dove-tailing and memory is configurable. The latter algorithm is referred to

[5]Dove-tailing can occur in practice in two ways. First, if there is a buffer between two resources (*e.g.,* the CPU and a link), this will allow packets with complementary resource demands to overlap in time. Second, if two resources need to be consumed in parallel (*e.g.,* CPU cycles and memory bandwidth), there can still be dove-tailing from processing multiple packets in parallel (*e.g.,* multiple cores can be working on packets with complementary needs).

as DRFQ, as the former two are special cases. Before explaining the algorithms, we briefly review Start-time Fair Queuing (SFQ) [18], which our work builds on.

5.3 Review: Start-time Fair Queuing (SFQ)

SFQ builds on the notion of virtual time. Recall from Section 3.3 that a virtual time unit is the time it takes to send *one* bit of a *unit*-weight flow in the fluid-flow system that fair queuing approximates. Thus, it takes l virtual time units to send a packet of length l with weight 1, or, in general, l/w units to send a packet with weight w. Note that the virtual time to send a packet is always the same regardless of the number of flows; thus, virtual time progresses slower in real-time when more flows are active.

Let p_i^k be the k-th packet of flow i. Upon p_i^k's arrival, all virtual time based schedulers assign it a start and a finish time $S(p_i^k)$ and $F(p_i^k)$, respectively, such that

$$F(p_i^k) = S(p_i^k) + \frac{L(p_i^k)}{w_i}, \tag{1}$$

where $L(p_i^k)$ is the length of packet p_i^k in bits, and w_i is the weight of flow i. Intuitively, functions S and F approximate the virtual times when the packet would have been transmitted in the fluid flow system.

In turn, the virtual start time of the packet is:

$$S(p_i^k) = \max\left(A(p_i^k), F(p_i^{k-1})\right), \tag{2}$$

where $A(p_i^k)$ is the virtual arrival time of p_i^k. In particular, let a_i^k be the (real) arrival time of packet p_i^k. Then, the $A(p_i^k)$ is simply the virtual time at real time a_i^k, *i.e.,* $V(a_i^k)$.

Fair queueing algorithms usually differ in (1) how $V(t)$ (virtual time) is computed and (2) which packet gets scheduled next.

While there are many possibilities for both choices, SFQ proceeds by (1) assigning each packet a virtual time equal to the start time of the packet currently in service (that is, $V(t)$ is the start time of the packet in service at real time t) and (2) always scheduling the packet with the lowest virtual start time. We discuss why these choices are attractive in middleboxes in Section 5.7.

5.4 Memoryless DRFQ

In many workloads, packets within the same flow have similar resource requirements. For such workloads, a memoryless DRFQ scheduler closely approximates DRF allocations.

Assume a set of n flows that share a set of m resources j, $(1 \leq j \leq m)$, and assume flow i is given weight w_i, $(1 \leq i \leq n)$.

Throughout, we will use the notation introduced in Table 1.

Achieving a DRF allocation requires that two backlogged flows receive the same processing time on their respective dominant resources, *i.e.,* on the resources they respectively require the most processing time on. Given our unified model of processing time, we can achieve this by using the maximum processing time of each packet when

Notation	**Explanation**
p_i^k	k-th packet of flow i
a_i^k	arrival time of packet p_i^k
$s_{i,j}^k$	processing time of p_i^k at resource j
$S(p)$	virtual start time of packet p in system
$F(p)$	virtual finish time of packet p in system
$V(t)$	system virtual time at time t
$V(t,j)$	system virtual time at time t at resource j
$S(p,j)$	virtual start time of packet p at resource j
$F(p,j)$	virtual finish time of packet p at resource j

Table 1: Main notation used in the DRFQ algorithm.

computing the packet's virtual finish time, *i.e.,* using $\max_j\{s_{i,j}^k\} \times \frac{1}{w_i}$ for the k^{th} packet of flow i with weight w_i.

For each packet we record its virtual start time and virtual finish time as follows:

$$\begin{aligned} S(p_i^k) &= \max\left(V(a_i^k), F(p_i^{k-1})\right), \quad (3) \\ F(p_i^k) &= S(p_i^k) + \frac{\max_j\{s_{i,j}^k\}}{w_i}. \quad (4) \end{aligned}$$

Thus, the finish time is equal to the virtual start time plus the processing time on the dominant resource. For a non-backlogged flow, the start time is the virtual time at the packet's arrival. For a backlogged flow, the max operator in the equation ensures that a packet's virtual start time will be the virtual finish time of the previous packet in the flow.

Finally, we have to define the virtual time function, V. Computing the virtual time function exactly is generally expensive [29] and even more so for DRF allocations. We therefore compute it as follows:

$$V(t) = \begin{cases} \max_j\{S(p,j) | p \in P(t)\} & \text{if } P(t) \neq \emptyset \\ 0 & \text{if } P(t) = \emptyset \end{cases} \quad (5)$$

where $P(t)$ are the packets currently in service at time t. Hence, virtual time is the maximum start time of any packet p that is currently being serviced.

Note that in the case of a single link where there is at most one packet in service at a time, this reduces to setting the virtual time at time t to the start time of the packet being serviced at t. This is exactly the way virtual start time is computed in SFQ. While there are many other possible computations of the virtual time, such as the average between the start and the finish times of the packets in service, in this paper we only consider an SFQ-like computation. In Section 5.7 we discuss why an SFQ-like algorithm is particularly attractive for middleboxes.

Table 2 shows two flows with process times $\langle 4, 1\rangle$ and $\langle 1, 3\rangle$, respectively, on all their packets. The first flow is backlogged throughout the example, with packets arriving much faster than they can be processed. The second flow is backlogged in two

Flow 1 with processing times $\langle 4, 1\rangle$								
Packet	p_1^0	p_1^1	p_1^2	p_1^3	p_1^4	p_1^5	p_1^6	p_1^7
Real arrival time	0	1	2	3	4	5	6	7
Virt. start/finish	0/4	4/8	8/12	12/16	16/20	20/24	24/28	28/32

Flow 2 with processing times $\langle 1, 3\rangle$								
Packet	p_2^0	p_2^1	p_2^2	p_2^3	p_2^4	p_2^5	p_2^6	p_2^7
Real arrival time	0	1	2	3	10	11	12	13
Virt. start/finish	0/3	3/6	6/9	9/12	20/23	23/26	26/29	29/32

Scheduling Order								
Order	1	2	3	4	5	6	7	8
Packet	p_1^0	p_2^0	p_2^1	p_1^1	p_2^2	p_1^2	p_2^3	p_1^3
Order	9	10	11	12	13	14	15	16
Packet	p_1^4	p_1^5	p_2^4	p_2^5	p_1^6	p_2^6	p_1^7	p_2^7

Table 2: Basic example of how memoryless DRFQ works with two flows. The first flow is continuously backlogged. The second flow is backlogged in two bursts.

bursts (packets p_2^0 to p_2^3 and p_2^4 to p_2^7). In the time interval 0 to 3, both flows are backlogged, so virtual start times are simply equal to the previous packet's virtual finish time. At time 10, the second flow's second burst starts with p_2^4. Assume that the middlebox is then processing p_1^5, which has virtual start time 20. Thus, $V(a_2^4) = 20$, making the virtual start time 20, instead of the previous packet's virtual finish time 12. Thereafter, the inflow of packets from the two flows is again faster than the service time, leading to start times equal to the finish time of the previous packet.

Table 3 shows why dove-tailing fails with this memoryless DRFQ algorithm. One flow's packets have alternating process times $\langle 1, 2\rangle$ and $\langle 2, 1\rangle$, while the second flow's packets have process times $\langle 3, 3\rangle$. Both flows are continuously backlogged with higher inrate than service rate. With perfect dove-tailing, the virtual finish time of p_1^3 should be the same as that of p_2^1. Instead, flow 1's virtual times progress faster, making it receive poorer service.

5.5 Dove-tailing DRFQ

To provide dove-tailing, we modify the memoryless DRFQ mechanism to keep track of the start and finish times of packets on a per-resource basis. The memoryless algorithm scheduled the packet with the smallest start time. Since a packet will now have multiple start times (one per resource), we need to decide which of them to use when making scheduling decisions. Given that we want to schedule based on dominant resources, we will schedule the packet whose *maximum* per-resource start time is smallest across all flows. This is because the packet's maximum start time is its start time on its flow's dominant resource. If two packet's have the same start time, we lexicographically compare their next largest start times.

More formally, we will compute the start and finish times of a packet at each re-

Flow 1 alternating $\langle 1,2\rangle$ and $\langle 2,1\rangle$						
Packet	p_1^0	p_1^1	p_1^2	p_1^3	p_1^4	p_1^5
Real start time	0	1	2	3	4	5
Virtual start time	0	2	4	6	8	10
Virtual finish time	2	4	6	8	10	12
Flow 2 with processing times $\langle 3,3\rangle$						
Packet	p_2^0	p_2^1	p_2^2	p_2^3	p_2^4	p_2^5
Real start time	0	1	2	3	4	5
Virtual start time	0	3	6	9	12	15
Virtual finish time	3	6	9	12	15	18
Scheduling Order						
Order	1	2	3	4	5	6
Packet	p_1^0	p_2^0	p_1^1	p_2^1	p_1^2	p_2^2
Order	7	8	9	10	11	12
Packet	p_1^3	p_1^4	p_2^3	p_1^5	p_2^4	p_2^5

Table 3: Why dove-tailing fails with memoryless DRFQ.

source j as:

$$S(p_i^k, j) = \max\left(V(a_i^k, j), F(p_i^{k-1}, j)\right), \quad (6)$$

$$F(p_i^k, j) = S(p_i^k, j) + \frac{s_{i,j}^k}{w_i}. \quad (7)$$

As mentioned above, the scheduling decisions should be made based on the maximum start or finish times of the packets across all resources, *i.e.,* $S(p_i^k)$ and $F(p_i^k)$ where

$$S(p_i^k) = \max_j\{S(p_i^k, j)\}, \quad (8)$$

$$F(p_i^k) = \max_j\{F(p_i^k, j)\}. \quad (9)$$

In the rest of this section, we refer to $S(p_i^k)$ and $F(p_i^k)$ as simply the start and finish times of packet p_i^k.

Finally, we now track virtual time per resource, *i.e.,* $V(t, j)$ at time t for resource j. We compute this virtual time independently at each resource:

$$V(t,j) = \begin{cases} \max_{p \in P(t)}\{S(p,j)\} & \text{if } P(t) \neq \emptyset \\ 0 & \text{if } P(t) = \emptyset \end{cases} \quad (10)$$

Table 4 shows how dove-tailing DRFQ schedules the same set of incoming packets as the example given in Table 3. Now virtual start and finish times are provided for both resources, R_1 and R_2. When comparing the two scheduling orders it is evident that dove-tailing DRFQ improves the service given to the first flow. For example, p_1^3

Flow 1 alternating $\langle 1,2\rangle$ and $\langle 2,1\rangle$

Packet	p_1^0	p_1^1	p_1^2	p_1^3	p_1^4	p_1^5
Real start time	0	1	2	3	4	5
Virtual start time R_1	0	1	3	4	6	7
Virtual finish time R_1	1	3	4	6	7	9
Virtual start time R_2	0	2	3	5	6	8
Virtual finish time R_2	2	3	5	6	8	9

Flow 2 with processing times $\langle 3,3\rangle$

Packet	p_2^0	p_2^1	p_2^2	p_2^3	p_2^4	p_2^5
Real start time	0	1	2	3	4	5
Virtual start time R_1	0	3	6	9	12	15
Virtual finish time R_1	3	6	9	12	15	18
Virtual start time R_2	0	3	6	9	12	15
Virtual finish time R_2	3	6	9	12	15	18

Scheduling Order

Order	1	2	3	4	5	6
Packet	p_1^0	p_2^0	p_1^1	p_2^1	p_1^2	p_1^3
Order	7	8	9	10	11	12
Packet	p_2^2	p_1^4	p_1^5	p_2^3	p_2^4	p_2^5

Table 4: How dove-tailing DRFQ satisfies dove-tailing.

is now scheduled before p_2^2, rather than after as with memory-less DRFQ. Though real processing time and virtual start times are different, the virtual finish times clearly show that two packets of flow 1 "virtually" finish for every packet of flow 2. As start times are based on finish times of the previous packet, the schedule will reflect this ordering.

Table 5 shows how dove-tailing DRFQ is not memoryless. Flow 1 initially has processing time $\langle 2,1\rangle$ for packets p_1^0 through p_1^2. But packets p_1^3 through p_1^5 instead have processing time $\langle 0.2,1\rangle$. Flow 2's packets all have processing time $\langle 2,1\rangle$. As can be seen, once flow 1's processing time switches, it gets scheduled twice in a row (p_1^4 and p_1^5). This example can in be extended to have an arbitrary number of flow 1's packets scheduled consecutively, increasing flow 2's delay arbitrarily.

5.6 Δ–Bounded DRFQ

We have explored two algorithms that trade off sharply between memoryless scheduling and dove-tailing. We now provide an algorithm whose degree of memory and dove-tailing can be controlled through a parameter Δ.

Such customization is important because in practice it is not desirable to provide unlimited dove-tailing. If a flow alternates sending packets with processing times $\langle 1,2\rangle$ and $\langle 2,1\rangle$, then the system can buffer packets and multiplex resources so that in real time, a pair of such packets take time equivalent to a $\langle 3,3\rangle$ packet. Contrast this with the flow first sending a long burst of 1000 packets with processing time $\langle 1,2\rangle$ and

Flow 1 p_1^0–p_1^2 require $\langle 2, 1\rangle$, and p_1^3–p_1^5 require $\langle 0.2, 1\rangle$						
Packet	p_1^0	p_1^1	p_1^2	p_1^3	p_1^4	p_1^5
Real start time	0	1	2	3	4	5
Virtual start time R_1	0	2	4	6	6.2	6.4
Virtual finish time R_1	2	4	6	6.2	6.4	6.6
Virtual start time R_2	0	1	2	3	4	5
Virtual finish time R_2	1	2	3	4	5	6
Flow 2 with processing times $\langle 2, 1\rangle$						
Packet	p_2^0	p_2^1	p_2^2	p_2^3	p_2^4	p_2^5
Real start time	0	1	2	3	4	5
Virtual start time R_1	0	2	4	6	8	10
Virtual finish time R_1	2	4	6	8	10	12
Virtual start time R_2	0	1	2	3	4	5
Virtual finish time R_2	1	2	3	4	5	6
Scheduling Order						
Order	1	2	3	4	5	6
Packet	p_1^0	p_2^0	p_1^1	p_2^1	p_1^2	p_2^2
Order	7	8	9	10	11	12
Packet	p_1^3	p_2^3	p_1^4	p_1^5	p_2^4	p_2^5

Table 5: Example of dove-tailing DRF not being memoryless. As of packet p_1^3, flow 1's processing time switches from $\langle 2, 1\rangle$ to $\langle 0.1, 1\rangle$.

thereafter a long burst of 1000 packets with processing time $\langle 2, 1\rangle$. After the first burst, the system is completely done processing most of those packets, and the fact that the processing times of the two bursts dove-tail does not yield any time savings. Hence, it is desirable to bound the dove-tailing to match the length of buffers and have the system be memoryless beyond that limit.

Δ–Bounded DRFQ is similar to dove-tailing DRFQ (§5.5), except that the virtual start and finish are computed differently. We replace the virtual start time, Eq. (6), with:

$$S(p_i^k, j) = \max\left(V(a_i^k, j), B_1(p_i^{k-1}, j)\right) \quad (11)$$

$$B_1(p, j) = \max\left(F(p, j), \max_{j' \neq j}\{F(p, j')\} - \Delta\right) \quad (12)$$

Thus, the start time of a packet on each resource can never differ by more than Δ from the maximum finish time of its flow's previous packet on any resource. This allows each flow to "save" up to Δ processing time for dove-tailing.

We similarly update the virtual time function (Eq. 10) to achieve the same bounding effect:

$$V(t,j) = \begin{cases} \max_{p \in P(t)}\{B_2(p,j)\} & \text{if } P(t) \neq \emptyset \\ 0 & \text{if } P(t) = \emptyset \end{cases} \tag{13}$$

$$B_2(p,j) = \max\left(S(p,j), \max_{j' \neq j}\{S(p,j')\} - \Delta\right) \tag{14}$$

Dove-tailing DRFQ (§5.5) and memoryless DRFQ (§5.4) are thus special cases of Δ–bounded DRFQ. In particular, when $\Delta = \infty$, the functions B_1 and B_2 reduce to functions F and S as in the previous section, and Δ–bounded DRFQ becomes equivalent to dove-tailing DRFQ. Similarly, if $\Delta = 0$, then B_1 and B_2 reduce to the maximum per-resource start and finish time of the flow's previous packet, respectively. Thus, Δ–bounded DRFQ becomes memoryless DRFQ. For these reasons, we simply refer to Δ–bounded DRFQ as DRFQ in the rest of the paper.

5.7 Discussion

The main reason we chose an SFQ-like algorithm to approximate DRFQ is that SFQ does not need to know the processing times of the packets before scheduling them. This is desirable in middleboxes because the CPU and memory bandwidth costs of processing a packet may not be known until after it has passed through the system. For example, different packets may pass through different processing modules (*e.g.,* HTTP caching) based on their contents.

Like SFQ, DRFQ schedules packets based on their virtual start times. As shown in Eq. (13), the virtual start time of packet p_i^k depends only on the start times of the packets in service and on the finish time of the previous packet, $F(p_i^{k-1})$. This allows us to delay computing $S(p_i^k)$ until just after p_i^{k-1} has finished, at which point we can use the *measured values* of packet p_i^{k-1}'s processing times, $s_{i,j}^{k-1}$, to compute its virtual finish time.

Although the use of a SFQ-like algorithm allows us to defer computing the processing time of each packet until after it has been processed (*e.g.,* after we have seen which middlebox modules it went through), there is still a question of *how* to measure the consumption. Unfortunately, measuring the exact CPU and memory consumption of *each* packet (*e.g.,* using CPU counters [3]) is expensive. However, in our implementation, we found that we could estimate consumption quite accurately based on the packet size and the set of modules it flowed through. Indeed, linear models fit the resource consumption with $R^2 > 0.97$ for many processing functions. In addition, DRFQ is robust to misestimation—that is, flows' shares might differ from the true DRF allocation, but each flow will still get a reasonable share as long as the estimates are not far off. We discuss these issues further in Section 7.1.

6 DRFQ Properties

In this section, we discuss two key properties of Δ-bounded DRFQ. Lemma 6.1 bounds the unfairness between two backlogged flows over a given time interval. This bound is independent of the length of the interval. Lemma 6.2 bounds the delay of a packet that arrives when the flow is idle. These properties parallel the corresponding properties in SFQ [18].

A flow is called *dominant-resource monotonic* if, during any of its backlogged periods, its dominant resource does not change. A flow in which all packets have the same dominant resource is trivially a dominant-resource monotonic flow. In this section, $s^{\uparrow}_{i,r}$ denotes $\max_k s^k_{i,r}$.

Consider a dominant-resource monotonic flow i, and let r be the dominant share of i. Then the virtual start times of i's packets at resource r do not depend on Δ. This follows trivially from Eq. (11), as $B_1(p_i, r)$ is equal to $F(p_i, j)$ for any packet p_i of i. For this reason, the bound in the next lemma does not depend on Δ.

Theorem 6.1 *Consider two dominant-resource monotonic flows i and j, both backlogged during the interval $[t_1, t_2)$. Let $W_i(t_1, t_2)$ and $W_j(t_1, t_2)$ be the total processing times consumed by flows i and j, respectively, on their dominant resource during interval $[t_1, t_2)$. Then, we have*

$$\left|\frac{W_i(t_1,t_2)}{w_i} - \frac{W_j(t_1,t_2)}{w_j}\right| < \frac{s^{\uparrow}_{i,d_i}}{w_i} + \frac{s^{\uparrow}_{j,d_j}}{w_j}, \tag{15}$$

where $s^{\uparrow}_{q,d_q}$ represents the maximum processing time of a packet of flow q on its dominant resource d_q.

A proof similar to the one provided by SFQ [18] works. Here, we provide a shorter and more intuitive proof sketch.

Proof Sketch Assume parallel consumption of resources. Let $v_1 = V(t_1, d_i)$ and $v_2 = V(t_2, d_i)$ and consider a flow i, with dominant resource d_i, that is backlogged in that entire interval of time. We initially assume that $v_2 - v_1 > s^{\uparrow}_{i,d_i}/w_i$, such that i can at least have one packet with start time inside the interval. We establish a lower bound and thereafter an upper bound on the amount of processing time i receives in the interval. The maximum discrepancy between these two is then used to establish the theorem.

Consider the following lower bound on the amount of processing time flow i receives in the interval $[t_1, t_2]$ on its dominant resource d_i. The key insight is that the amount of processing time that flow i receives on its dominant resource is at least $v_2 - S(p^k_i, d_i)$, where packet p^k_i is i's first packet served in the interval $[t_1, t_2]$. This is because, by definition of V, virtual time can only progress to a greater value v_g when all packets with maximum starting time less than v_g have been served. Since i is backlogged, the start times of each of i's packets, after the k:th, is equal to the finish time of the previous packet. Thus, it receives processing time on its dominant resource equal to $\sum_{j=k}^{m} s^j_{i,d_i}$, where p^m_i is the last packet in the window, *i.e.,* $S(p^m_i, d_i) \in [v_1, v_2]$ and $F(p^m_i, d_i) \geq v_2$. Thus, $S(p^k_i, d_i)$ determines the total processing time received on d_i in the interval. Flow i's previous packet p^{k-1}_i must have a start time less than v_1. As i is backlogged, $S(p^k_i, d_i) \leq v_1 + \frac{s^{\uparrow}_{i,d_i}}{w_i}$. Thus, i will at least receive $(v_2 - v_1) - \frac{s^{\uparrow}_{i,d_i}}{w_i}$ processing time on d_i in the interval $[t_1, t_2]$.

Consider an upper bound on the amount of processing i receives on d_i in $[t_1, t_2]$. The processing time is maximized when the first packet k starts at $S(p_i^k, d_i) = v_1$ and the last packet p_i^m starts at the end of the interval, *i.e.,* $S(p_i^m, d_i) = v_2$. Since, p_i^m's processing time on d_i is at most $s^{\uparrow}_{i,d_i}$, it can receive processing time in the virtual time interval $[v_1, v_2 + \frac{S(p_i^m, d_i)}{w_i}]$. Thus, i receives at most $(v_2 - v_1) + \frac{s^{\uparrow}_{i,d_i}}{w_i}$ processing time on d_i in $[t_1, t_2]$.

The maximum discrepancy of processing times received by two backlogged flows i and j on their respective dominant resource in the interval $[t_1, t_2]$ is therefore $\frac{s^{uparrow}_{i,d_i}}{w_i} + \frac{s^{\uparrow}_{j,d_j}}{w_j}$, which proves the theorem for the special case.

Consider the case when $v_2 - v_1 \leq s^{\uparrow}_{q,d_q}/w_q$, for any flow $q \in \{i, j\}$. Then q might not get any service in $[t_1, t_2]$. The upper bound is that q has one packet that happens to have start time in the interval. Thus, the maximum discrepancy is $\max\limits_{q \in \{i,j\}} \left\{ \frac{s^{\uparrow}_{q,d_q}}{w_q} \right\}$, which proves the theorem for this case as well. □

The next result does not assume that flows are dominant-resource monotonic but assumes that each packet has a non-zero demand for every resource.

Theorem 6.2 *Assume packet p_i^k of flow i arrives at time t, and assume flow i is idle at time t. Assume all packets have non-zero demand on every resource. Let n be the total number of flows in the system. Then the maximum delay to start serving packet p_i^k, $D(p_i^k)$, is bounded above by*

$$D(p_i^k) \leq \max_r \left(\sum_{j=1, j \neq i}^{n} s^{\uparrow}_{j,r} \right), \tag{16}$$

where $s^{\uparrow}_{j,d_j}$ represents the maximum processing time of a packet of flow j on its dominant resource d_j.

Proof Sketch If no packet is being processed at time t, p_i^k will immediately be processed and the theorem trivially holds. Thus, assume a packet is currently being processed with maximum start time $v_t = \max_r v(t, r)$. Since i is assumed to not be backlogged, we have $S(p_i^k) = v_t$. Since processing happens with increasing start tags, all backlogged packets p must have $\max_r S(p, r) \geq v_t$. Since each flow has strictly non-zero processing time on each resource, at most one packet from each flow can have starting time v_t. Thus, at most $n - 1$ packets can have a starting time equal to that of p_i^k. Assuming parallel computation, each such packet of flow j requires $s^{\uparrow}_{j,r}$ processing time on resource r. Consequently, total delay is no greater the resource that finishes last when processing all those $n - 1$ packets. □

7 Implementation

We prototyped DRFQ in the Click modular router [22]. Our implementation adds a new DRFQ-Queue module to Click, consisting of roughly 600 lines of code. This module

Module	R^2 for CPU	R^2 for Memory
Basic Forwarding	0.921	0.994
Redundancy Elim.	0.997	0.978
IPSec Encryption	0.996	0.985
Stat. Monitoring	0.843	0.992

Table 6: R^2 values for fitting a linear model to estimate the CPU and memory bandwidth use of various modules.

takes as input a *class specification* file identifying the types of traffic that the middlebox processes (based on port numbers and IP prefixes) and a model for estimating the packet processing times for each middlebox function. We use a model to estimate the packets' processing times because measuring the exact CPU and memory usage of a single packet is expensive.

The main difference from a traditional Queue is that DRFQ-Queue maintains a per-flow buffer with a fixed capacity *Cap* per-flow and also tracks the last virtual completion time for each flow. As each packet arrives, it is assigned a virtual start time and is added to the buffer corresponding to its flow. If the specific buffer is full, the packet is dropped, but the virtual completion time for the flow is not incremented. On every call to dequeue a packet, DRFQ-Queue looks at the head of each per-flow queue and returns the packet with the lowest virtual start time.

To decide when to dequeue packets to downstream modules, we use a separate token bucket for each resource; this ensures that we do not oversaturate any particular resource. On each dequeue, we pull out a number of tokens from each resource corresponding to the packet's estimated processing time on that resource. In addition, we periodically check the true utilization rate of each resource, and we scale down the rate at which we dequeue packets if we find that we have been estimating processing times incorrectly. This ensures that we do not overload the hardware.

7.1 Estimating Packets' Resource Usage

To implement any multi-resource scheduler, one needs to know the consumption of each packet for each resource. This was simple when the only resource scheduled was link bandwidth, because the size of each packet is known. Consumption of other resources, such as CPU and memory bandwidth, is harder to capture at a fine granularity. Although CPU counters [3] can provide this data, querying the counters for each packet adds overhead. Fortunately, it is possible to estimate the consumption accurately. We used a two-step approach similar to the one taken in [15]: (i) Determine which modules each packet passed through. Fortunately, with DRFQ it is necessary to know this only *after* the packet has been processed (*c.f.,* §5.7), (ii) Use a model of each module's resource consumption as a function of packet size to estimate the packet's consumption.

For the second step, we show that modules' resource consumption can be estimated accurately using a simple linear model based on packet size. Specifically, for a given module m and resource r, we find parameters $\alpha_{m,r}$ and $\beta_{m,r}$ such that the resource

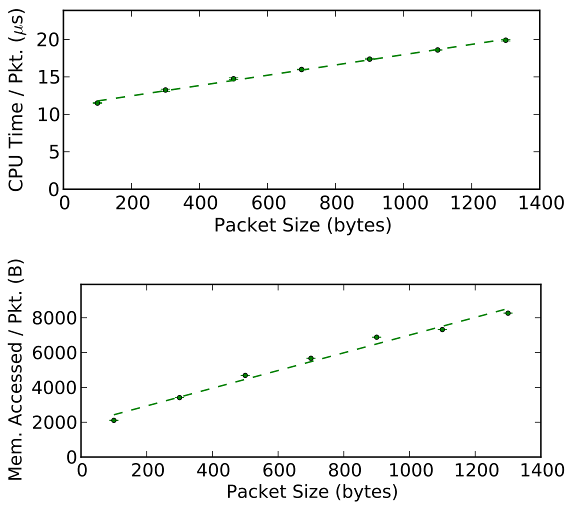


Figure 7: Per-packet CPU and memory b/w consumption of the redundancy elimination. Results are averaged over five runs that measure the consumption of processing 10,000 packets each. Error bars show max and min values. As seen, for both memory and CPU, linear models fit well with $R^2 > 0.97$.

consumption of a packet of size x is $\alpha_{m,r}x + \beta_{m,r}$. We have fit such linear models to four Click modules and show that they predict consumption well, fitting with $R^2 \geq 0.97$ in most cases. For example, Figure 7 shows the CPU and memory bandwidth consumption of a redundancy elimination module. We list the R^2 values for other modules in Table 6. The only cases where R^2 is lower than 0.97 are for the CPU consumptions of basic forwarding and statistical monitoring, which are nearly constant but have jumps at certain packet sizes, as shown in Figure 8. We believe this to be due to CPU caching effects.

Further refinements can be made based on the function of the module. For example, if a module takes more time to process the first few packets of a flow (for some one-time work), one can use a separate linear model for them.

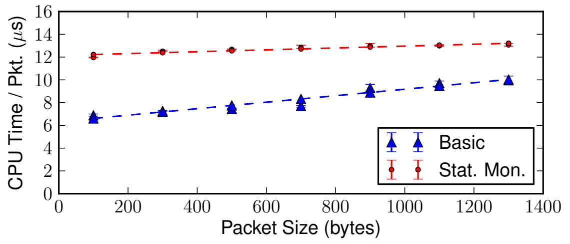


Figure 8: CPU usage vs. packet size for basic forwarding and statistical monitoring.

Finally, if the estimation is wrong, DRFQ will still run, but flows' shares may be off by the ratio to which processing times have been misestimated. One can also imagine dynamically recomputing each flow's usage, but we chose not to explore estimation further in this paper as it is orthogonal to our main focus of defining a suitable allocation policy.

8 Evaluation

We evaluated DRFQ using both our Click implementation and packet-level simulations. We use Click to show the basic functioning of the algorithm and simulations to compare it in more detail against other schedulers. Our workload is mostly dominant-resource monotonic, so we used the $\Delta = 0$ configuration by default, unless otherwise stated.

8.1 Implementation Results

We ran a Click-based multi-function middlebox in usermode on an Intel(R) Xeon(R) CPU 2.8GHz X5560 machine with a 1Gbps Ethernet link. We connected this machine to a traffic generator that uses Click to send packets from multiple flows. We configured the middlebox to apply three different processing functions to these flows based on their port number: basic forwarding, per-flow statistical monitoring, and IPSec encryption. Because our machine only had one 1 Gbps link, we throttled its outgoing bandwidth to 200 Mbps to emulate a congested link, and throttled the fraction of CPU time that the DRFQ module is allowed to use for processing to 20% so that the CPU can also be a bottleneck at this rate.

8.1.1 Dynamic Allocation

We begin by generating three flows that each send 25,000 1300-byte UDP packets per second to exceed the total outgoing bandwidth capacity. We configured the flows such that: (i) Flow 1 only undergoes basic forwarding, which is link bandwidth bound, (ii) Flow 2 undergoes IPSec, which is CPU-bound, (iii) Flow 3 requires statistical monitoring, which is bandwidth-bound but uses slightly more CPU than basic forwarding.

Figure 9 shows the resource shares of the flows over time (measured using timing instrumentation we added in Click), as we start and end them at different points. We see that Flow 1 initially has a complete share of the network, but only 20% of the CPU since it only requires lightweight processing. When Flow 2 arrives, Flow 1's CPU and network share expectedly decreases. Note, however that Flow 1's network share is more than 2$\times$ higher that Flow 2 because Flow 2 has a different dominant resource, namely CPU. Also, the two flows' resource demands dovetail, and their dominant shares are equalized. Finally when Flow 3 arrives, we observe that the network shares of Flow 1 and Flow 3 are equalized (as the dominant resource is link bandwidth for both), and Flow 2's share decreases further to equalize the dominant shares.

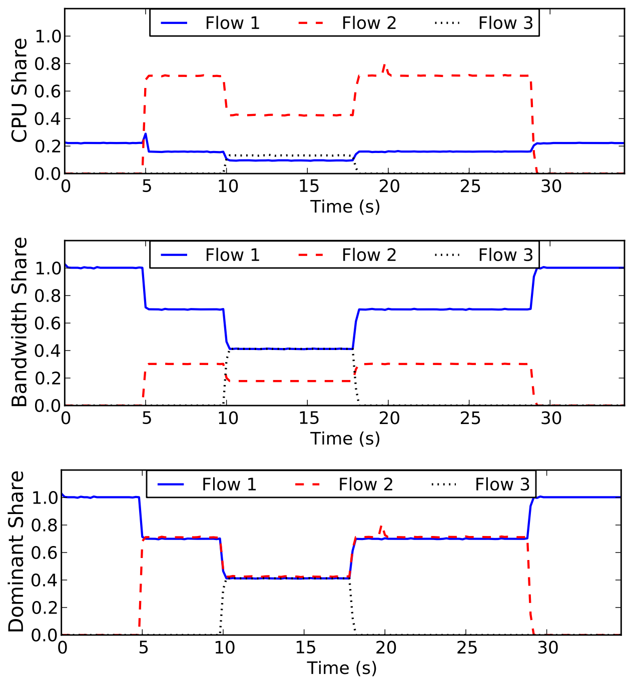


Figure 9: Shares of three competing flows arriving at Click at different times. Flow 1, 2, and 3 respectively undergo basic forwarding, IPSec, and statistical monitoring.

8.1.2 Isolation of Small Flows

Next, we extend the above setup to analyze the impact of DRFQ on short flows. As before, Flow 1 and Flow 2 require basic and IPSec processing respectively, and they are set to send 40,000 packets/second each to exceed the outgoing bandwidth. We then add two new flows, Flow 3 and Flow 4, both using only basic processing, but sending packets at a much lower rate of 1 packet/second and 0.5 packets/second, respectively. Ideally, we want these low-rate flows to have no backlog and not be impacted by the larger queues from the high-rate flows. Figure 10 confirms that this happens in practice, showing the steady-state latency of the four flows: both low-rate flows see more than an order of magnitude lower per-packet latency than the larger ones. We also notice that the high-rate IPSec flow has a higher latency than the high-rate basic flow because it has a smaller bandwidth share but the same queue size.

8.1.3 Comparison with Per-Resource Fairness

We also implemented per-resource fairness [15] in Click to test whether the oscillations that occur when two resources are in demand affect performance. For these experiments, we used TCP flows, and added 20 ms of network latency to get realistic behavior for wide-area flows. We made per-resource fairness check for a new bottleneck every

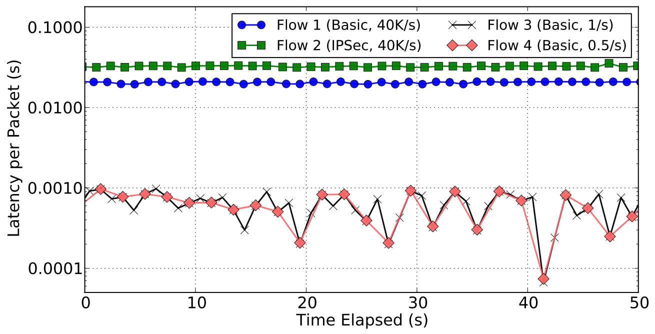


Figure 10: Latencies of DRFQ scheduling in Click when two bottlenecked flows (sending 40,000 packets/s each) and two low-rate flows (sending 0.5-1 packets/s) compete.

Scenario	Flow 1 (BW-bound)	Flow 2 (CPU-bound)
Running alone	191 Mbps	33 Mbps
Per-res. fairness	75 Mbps	32 Mbps
DRFQ	160 Mbps	28 Mbps

Table 7: Throughput of bandwidth and CPU intensive flows alone and under per-resource fairness and DRFQ.

300 ms. We ran two TCP flows for 30 seconds each: one that only undergoes basic processing, and one that undergoes CPU-intensive redundancy elimination as well.

Table 7 shows the throughputs of both flows running separately (one at a time), together under per-resource fairness, and together under DRFQ. With per-resource fairness, the oscillations in available bandwidth for flow 2 cause it to lose packets, back off, and get less than half the share it had running alone (*i.e.,* less than its share guarantee). This does not happen for the second flow because its rate is smaller so its queue in the middlebox does not overflow. In contrast, however, DRFQ provides a high throughput for both flows, letting both use about 83% of the bandwidth it would have alone because their demands dove-tail.

8.2 Simulation Results

We compare DRFQ with the alternative solutions using per-packet simulations. The results are based on a discrete-event simulator that assumes resources are being used serially. It implements different queuing principles, including DRFQ, by looking at an input queue of packets and selecting which flow's packet should get processed next. It uses Poisson arrivals and normally distributed resource consumption. The packet processing times have means according to each flow's provided resource profile and standard deviation set to a tenth of the mean.

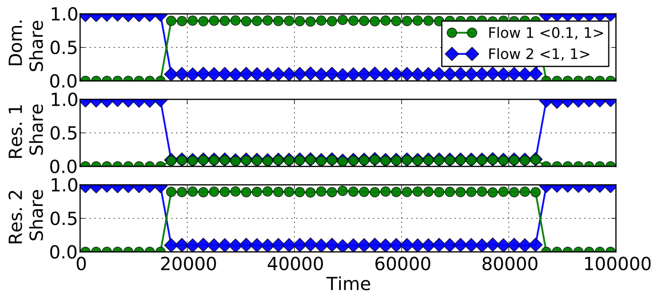


Figure 11: Fair queuing applied to only the first resource violates the share guarantee for flow 2.

8.2.1 Comparison With Alternative Schedulers

Single-resource Fair Queuing: The first approach we test applies fair queuing on just one resource (*e.g.,* link bandwidth). This is the allocation that would result if traditional weighted fair queuing were used, ignoring the multi-resource consumption of packets. Figure 11 shows the simulation of a scenario in which one flow uses an equal amount of two resources, *i.e.,* $\langle 1, 1\rangle$. Another flow, with profile $\langle 0.1, 1\rangle$, starts and ends at times 15,000 and 85,000, respectively. Fair queuing is only applied to the first resource. We see that the share guarantee is violated; the $\langle 1, 1\rangle$ flow gets only 10% of each resource when the other flow is active.

Bottleneck Fairness: To investigate how Bottleneck Fairness behaves when multiple resources are bottlenecked, we let one flow use equal amounts of two resources, $\langle 1, 1\rangle$. Then we let two flows have resource profiles $\langle 1, 0.1\rangle$ and $\langle 0.1, 1\rangle$. Bottleneck queuing was configured to dynamically switch to the current bottleneck (every 20,000 time units). As can be seen in Figure 12, oscillations occur when the bottleneck shifts (*c.f.,* §4.1). As a result, the first flow only gets 10% of either resource, far less than its ideal share guarantee of $\frac{1}{3}$.

Per-Resource Fairness: Figure 13 investigates how a flow can manipulate per-resource fair queuing by changing its demands (*e.g.,* by changing packet sizes) to receive better service. It simulates a scenario with ten flows. The first flow has resource profile $\langle 20, 1\rangle$, whereas the last nine have $\langle 10, 11\rangle$. At time 25,000 the first flow artificially changes its demand to roughly $\langle 20, 11\rangle$, leading it to double its share under per-resource fair queuing. Meanwhile, the same change under DRFQ has no effect on the shares.

8.2.2 Isolation Under DRFQ

Next, we investigate packet delays under DRFQ. Figure 14 shows two different flows. The first is constantly backlogged to the level that it overflows all buffers and suffers from packet drops. The second flow periodically sends single packets, spread far apart in time. For both flows we measure the queuing delay for every packet and plot the mean and standard deviation. The x-axis shows the same simulation for various buffer sizes. As the buffer size is increased, the delay on the backlogged flow increases, as

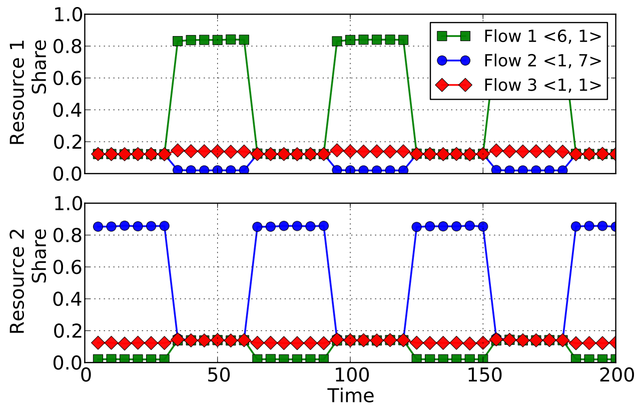


Figure 12: Bottleneck Queuing [15] leading to heavy oscillations, which in turn almost starve a third flow.

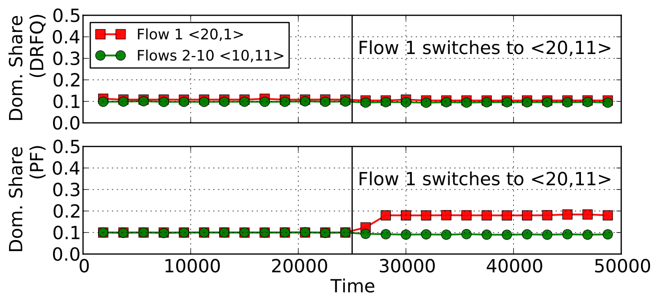


Figure 13: A flow manipulating PF to double its share.

the incoming rate of packets is much higher than what the system can handle. The periodic flow, however, is unaffected by the backlogged flow and receives constant delay, irrespective of the buffer length.

8.3 Overhead

To evaluate the overhead of our Click implementation of DRFQ, we used the aforementioned trace generator to create a synthetic 350 MB workload from actual traces [4]. We ran the workload through two applications: flow monitor and intrusion detection system (IDS). For each application, we measured the overhead with and without DRFQ. For flow monitor the overhead was 4%, whereas for IDS it was 2%. While this is already low, we believe the overhead can be further reduced. First, DRFQ requires per-flow queues that are currently implemented in software. Many software routers and middleboxes already have support for in-hardware queues. Second, the overhead can be reduced using fair queueing per-class or per-aggregate basis, rather than per-flow basis.

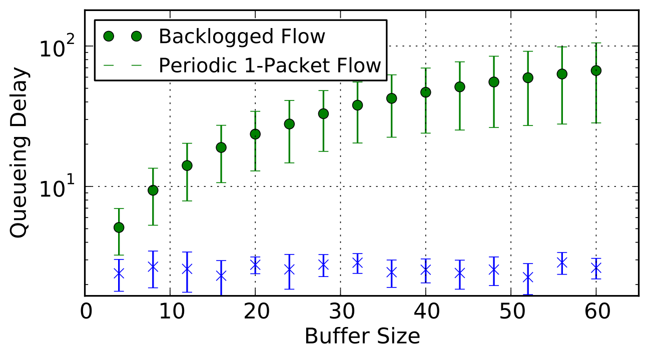


Figure 14: Per-packet delay of a backlogged flow compared to a periodic single-packet flow under DRFQ.

9 Related Work

Our work builds on WFQ [11, 24] as it, in similarity with many GPS approximations [18, 9, 17], uses the notion of virtual time. In particular, we approximate virtual time using start times as in SFQ [18], as it helps us avoid knowing in advance what middlebox modules a packet will traverse. As our evaluation shows, naively performing fair queuing on a single resource provides poor isolation for flows, violating the share-guarantee. Our attempt to extend WFQ by doing per-resource fair queuing (§4.2) turned out to violate strategy-proofness. Thus, DRFQ generalizes WFQ to multiple resources while providing isolation and strategy-proofness.

In the context of middleboxes, Egi *et al.* [15] proposed bottleneck fairness for software routers. We share their motivation for multi-resource fairness. However, we showed (§4.1) that their mechanism can not only provide poor isolation, but it can lead to heavy oscillations that severely degrade system performance. Dreger *et al.* [13] suggest measuring resource consumption of modules in NIDS and shutting off modules that overconsume resources. This approach is infeasible as some modules must run at all times, *e.g.,* a VPN module. Moreover, shutting down modules does not provide isolation between flows. With our approach, the flows that overconsume resources will fill buffers, eventually leading to modules not processing them, but each flow is sure to at least get its share guarantee of service.

In the context of active networks, Alexander *et al.* [7] propose a scheduling architecture called RCANE. This approach is akin to Per-Resource Fairness and therefore violates strategy-proofness.

Multi-resource fairness has been investigated in the context of micro-economic theory. Ghodsi *et al.* [16] provide an overview and compare with the method preferred by economists, Competitive Equilibrium from Equal Incomes (CEEI). They show that CEEI is not strategy-proof and has several other undesirable properties. Dolev *et al.* [12] proposed an alternative to DRF. It too fails to be strategy-proof, and is also computationally expensive to compute.

Our focus in this paper has been on achieving DRF allocations in the time domain. Others have analyzed how DRF allocations can be computed [19] and extended [21, 25].

10 Conclusion

Middleboxes apply complex processing functions to an increasing volume of traffic. Their performance characteristics are different from traditional routers; different processing functions have different demands across multiple resources, including CPU, memory bandwidth, and link bandwidth. Traditional single resource fair queuing schedulers therefore provide poor isolation guarantees between flows. Worse, in systems with multiple resources, flows can shift their demand to manipulate schedulers to get better service, thereby wasting resources. We have analyzed two schemes that are natural in the middlebox setting—bottleneck fairness and per-resource fairness—and shown that they have undesirable properties. In light of this, we have designed a new algorithm, DRFQ, for multi-resource fair queueing. We show through a Click implementation and extensive simulations that, unlike other approaches, our solution does not suffer from oscillations, provides flow isolation, and is strategy-proof. For future research directions, we believe DRFQ is applicable in many other multi-resource fair queueing contexts, such as VM scheduling in hypervisors.

11 Acknowledgements

We thank Adam Oliner, Ganesh Ananthanarayanan, and Patrick Wendell for useful feedback on earlier drafts of this paper. This research is supported in part by NSF FIA Award #CNS-1038695, NSF CISE Expeditions award CCF-1139158, a Google PhD Fellowship, gifts from Amazon Web Services, Google, SAP, Blue Goji, Cisco, Cloudera, Ericsson, General Electric, Hewlett Packard, Huawei, Intel, MarkLogic, Microsoft, NetApp, Oracle, Quanta, Splunk, VMware and by DARPA (contract #FA8650-11-C-7136).

References

[1] Crossbeam network consolidation. `http://bit.ly/qlotDK`.

[2] F5 Networks products. `http://www.f5.com/products/big-ip/`.

[3] Intel performance counter monitor. `http://software.intel.com/en-us/articles/intel-performance-counter-monitor/`.

[4] M57 network traffic traces. `https://domex.nps.edu/corp/scenarios/2009-m57/net/`.

[5] Palo alto networks. `http://www.paloaltonetworks.com/`.

[6] Vyatta Software Middlebox. `http://www.vyatta.com`.

[7] D. S. Alexander, P. B. Menage, A. D. Keromytis, W. A. Arbaugh, K. G. Anagnostakis, and J. M. Smith. The Price of Safety in An Active Network. *JCN*, 3(1):4–18, March 2001.

[8] K. Argyraki, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Understanding the packet forwarding capability of general-purpose processors. Technical Report IRB-TR-08-44, Intel Research Berkeley, May 2008.

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

[10] B. Briscoe, A. Jacquet, C. D. Cairano-Gilfedder, A. Salvatori, A. Soppera, and M. Koyabe. Policing congestion response in an internetwork using re-feedback. In *Proc. SIGCOMM*, 2005.

[11] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM '89*, pages 1–12, New York, NY, USA, 1989. ACM.

[12] D. Dolev, D. G. Feitelson, J. Y. Halpern, R. Kupferman, and N. Linial. No justified complaints: On fair sharing of multiple resources. *CoRR*, 2011.

[13] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *ACM CCS*, pages 2–11. ACM Press, 2004.

[14] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Predicting the resource consumption of network intrusion detection systems. In *RAID*, pages 135–154, 2008.

[15] N. Egi, A. Greenhalgh, M. Handley, G. Iannaccone, M. Manesh, L. Mathy, and S. Ratnasamy. Improved forwarding architecture and resource management for multi-core software routers. In *NPC*, pages 117–124, 2009.

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

[17] S. J. Golestani. A self-clocked fair queueing scheme for broadband applications. In *INFOCOM*, pages 636–646, 1994.

[18] P. Goyal, H. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. *ACM Trans. on Networking*, 5(5):690–704, Oct. 1997.

[19] A. Gutman and N. Nisan. Fair Allocation Without Trade. In *AAMAS*, June 2012.

[20] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *Proc. IMC*, 2011.

[21] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang. Multi-resource allocation: Fairness-efficiency tradeoffs in a unifying framework. In *INFOCOM*, pages 1206–1214, 2012.

[22] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18, August 2000.

[23] M. Kounavis, X. Kang, K. Grewal, M. Eszenyi, S. Gueron, and D. Durham. Encrypting the Internet. In *Proc. SIGCOMM*, 2010.

[24] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control - the single node case. *ACM Transactions on Networking*, 1(3):344–357, June 1993.

[25] D. C. Parkes, A. D. Procaccia, and N. Shah. Beyond Dominant Resource Fairness: Extensions, Limitations, and Indivisibilities. In *ACM EC*, 2012.

[26] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in bittorrent. In *NSDI'07*, 2007.

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

[28] V. Sekar, S. Ratnasamy, M. Reiter, N. Egi, and G. Shi. The Middlebox Manifesto: Enabling Innovation in Middlebox Deployments. In *HotNets 2011*, Oct. 2011.

[29] M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round robin. *ACM Transactions on Networking*, 4(3):375–385, 1996.

[30] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Signature Matching in Network Processing Using SIMD/GPU Architectures. In *International Symposium on Performance Analysis of Systems and Software*, 2009.

[31] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional Share Resource Management*. PhD thesis, MIT, Laboratory of Computer Science, Sept. 1995. MIT/LCS/TR-667.

[32] Z. Wang, Z. Qian, Q. Xu, Z. M. Mao, and M. Zhang. An Untold Story of Middleboxes in Cellular Networks. In *SIGCOMM*, 2011.

[33] L. Zhang. Virtual clock: a new traffic control algorithm for packet switching networks. *SIGCOMM CCR*, 20:19–29, August 1990.