Improving Software Middleboxes and Datacenter Task Schedulers

Hugo Sadok, Miguel Elias M. Campista, Luís Henrique M. K. Costa[*]

Universidade Federal do Rio de Janeiro – GTA/PEE/COPPE

{sadok, miguel, luish}@gta.ufrj.br

Abstract. *Shared systems have contributed to the popularity of many technologies. However, these systems often confront a common challenge: to ensure that resources are fairly divided without compromising utilization efficiency. In this master's thesis we look at this problem in two distinct systems—software middleboxes and datacenter task schedulers. We first present Sprayer, a system that uses packet spraying to load balance packets to cores in software middleboxes. Our design eliminates the imbalance problems of per-flow solutions and addresses the new challenges of handling shared flow states that come with packet spraying. Then, we present Stateful Dominant Resource Fairness (SDRF), a task scheduling policy for datacenters that looks at past allocations and enforces fairness in the long run. SDRF reduces users' waiting time on average and improves fairness by increasing the number of completed tasks for users with lower demands, with small impact on high-demand users.*

1. Motivation and Problem Statement

Over the last decades, shared systems have contributed to the popularity of many technologies. From Operating Systems to the Internet, they have all brought significant cost savings by allowing the underlying infrastructure to be shared. A common challenge in these systems is to ensure that resources are fairly divided without compromising utilization efficiency. This tradeoff between efficiency and fairness presents itself in a variety of ways and in different levels of system design. In this master's thesis we present ideas that improve both efficiency and fairness in two popular shared systems: software middleboxes and datacenter task schedulers. In the following subsections we describe the two problems we tackle.

1.1. Inefficient Use of Multiple Cores in Software Middleboxes

Today middleboxes are a primary component of both enterprise and Internet provider networks [Sekar et al. 2012]. Middleboxes allow network operators to deploy a wide range of network functions (NFs), such as Network Address Translators (NATs), firewalls, and load balancers. Yet, the cost and lack of flexibility of purpose-built hardware middleboxes are pushing operators to software running on commodity servers [Chiosi et al. 2012]. Moving to software, however, does not come for free. Software middleboxes have significant overhead and often need to use multiple CPU cores [Sun et al. 2017]—or even

[*] The content of this paper is partially adapted from our previously published works [Sadok et al. 2018a, Sadok et al. 2018b] as well as from the master's thesis [Barreto 2018]. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, CNPq, FAPERJ, and FAPESP grants #15/24494-8 and #15/24490-2.

multiple hosts [Kablan et al. 2017, Woo et al. 2018]—to achieve line rates. Moreover, the rapid increase of network link capacities only exacerbates this need.

When using multiple cores, middleboxes must determine which core to direct packets to. Today, this is done using Receive-Side Scaling (RSS). RSS is a feature of multi-queue network interface controllers (NICs) that directs packets to different cores using a hash of the five-tuple. Doing so, all packets from the same flow end up in the same core. The reasons for coupling packets from the same flow are twofold. First, processing same-flow packets sequentially avoids packet reordering. Second, having same-flow packets processed in the same core simplifies flow state handling. RSS, however, has significant shortcomings. It is inefficient, since it cannot use all the available cores when the number of concurrent flows is small—which happens frequently in real workloads [Barreto 2018, §3.1]. Moreover, since RSS directs flows to cores using a hash of the five-tuple, hash collisions cause asymmetry in flow distribution.[1] This results in unfairness even with a larger number of flows [Barreto 2018, §3.4]. In Section 2 we look at this problem and make a case for a natural alternative: that middleboxes should direct packets to cores at a finer granularity. We present a system that uses packet spraying to direct packets to cores in software middleboxes and addresses the new challenges of handling shared flow state that come with this new approach.

1.2. Long-Term Unfairness in Datacenter Task Schedulers

Modern datacenters are often shared by users with heterogeneous resource constraints [Reiss et al. 2012]. The amount of resources given to each user directly impacts the system performance from both fairness and efficiency standpoints [Joe-Wong et al. 2013]. In single-resource systems, max-min fairness is the most widely used and studied allocation policy. The main idea is to maximize the minimum allocation a user receives. It was originally proposed to ensure a fair share of link capacity for every flow in a network. Since then, max-min has been applied to a variety of individual resource types, including CPU, memory, and I/O [Ghodsi et al. 2011]. Nevertheless, datacenters need to allocate *multiple* resource types at the same time (such as CPU and memory) and max-min is unable to ensure fairness [Ghodsi et al. 2011].

In a datacenter environment, users often have heterogeneous demands and dynamic workloads [Reiss et al. 2012]. Different mechanisms have been proposed to address the multi-resource allocation, most notably, Dominant Resource Fairness (DRF) [Ghodsi et al. 2011]. DRF generalizes max-min to the multi-resource setting, by giving users an equal share of their mostly demanded resource—their *dominant resource*. Using this approach, DRF achieves several desirable properties. Despite the extensive literature on fair allocation, most allocation policies focus only on instantaneous, or short term, fairness, ensuring that users receive an equal share of the resources regardless of their past behaviors. DRF is no exception, it guarantees fairness only when users' demands remain constant. In practice, however, users' workloads are quite dynamic [Reiss et al. 2012] and ignoring this fact leads to sub-optimal allocations and unfairness in the long run. In Section 3 we propose a mechanism that extends DRF to consider past allocations. We show that this mechanism ensures fairness in the long run and reduces user's waiting time on average.

[1] Even when the number of cores is comparable to the number of flows, hash collisions happen with high probability due to the birthday problem.

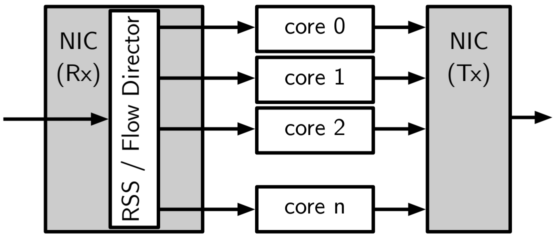

Figure 1. Software middlebox: the NIC can direct packets to cores using either RSS or Flow Director. Both send all packets from the same flow to the same core.

NF	State	Scope	Access Pattern	
			packet	flow
NAT, IPv4 to IPv6	Flow map	Per-flow	R	RW
	Pool of IPs/ports	Global	-	RW
Firewall	Connection context	Per-flow	R	RW
Load Balancer	Flow-server map	Per-flow	R	RW
	Pool of servers	Global	-	RW
	Statistics	Global	RW	-
Traffic Monitor	Connection context	Per-flow	-	RW
	Statistics	Global	RW	-
Redundancy Elimination	Packet cache	Global	RW	-
DPI	Automata	Per-flow	RW	-

Table 1. Example of state scope and access pattern of some popular stateful NFs. Most NFs only update flow state when connections start or finish.

2. Sprayer

To solve the imbalance problems caused by hashing flows to cores in software middleboxes, we take inspiration from a similar problem in a different domain: datacenter networks. Traditionally, datacenter networks use Equal Cost Multi-Path (ECMP) to direct packets to different paths. Like RSS, ECMP directs all packets from the same flow to the same path and, as such, has similar shortcomings. This observation has led recent works to consider load-balancing packets ignoring their flows. This approach, known as packet spraying, introduces reordering but, because datacenter networks have paths with low and very similar latencies, the amount of reordering is not enough to significantly harm TCP. In the first part of the master's thesis we propose Sprayer, a system that allows the development of network functions using packet spraying. There are two main challenges in the design of Sprayer: *spraying packets using existing NICs* and *handling flow states*.

2.1. Spraying Packets Using Existing NICs

At first glance, it may seem impossible to spray packets using existing commodity NICs, since they do not offer this functionality (see Figure 1). We circumvent this limitation using Flow Director, a functionality found in many commodity NICs designed to associate *specific* sets of flows to cores. We use Flow Director in an unconventional manner: instead of matching flows, we configure it such that packets are directed to cores using the checksum field of the TCP header. Since this field looks random, TCP packets are uniformly distributed across cores, regardless of their flows. Non-TCP packets fail to match any rules and fall back to traditional RSS. This avoids the potential problems packet reordering causes to some UDP applications (*e.g.*, VoIP).

2.2. Handling Flow States

When we send all the packets from the same TCP connection to the same core, we benefit from having partitionable flow states, which ensures that each core only has to keep state for its flows. Partitionable state is desirable, as it avoids the penalty of enforcing cache coherence, as well as the use of synchronization primitives. When we use packet spraying, packets from the same flow may go to different cores and this property no longer holds. What we observe, however, is that we get similar benefits if we instead provide *writing* partition. As long as we guarantee that each flow state can only be modified by a single core, we avoid the use of locks and significantly reduce cache invalidations.

To ensure writing partition, we depart from the observation that most NFs only change flow state when TCP connections start or finish. Table 1 shows the scope (per-flow or global state) and access pattern (read or write at every packet or flow) for some popular stateful NFs. To leverage this observation, Sprayer makes a distinction between *connection packets* and *regular packets*. Connection packets are those that have potential to modify TCP state (those flagged with `SYN`, `FIN`, or `RST`), while regular packets are all the others. Sprayer ensures writing partition for flow states by making sure that all connection packets from the same TCP connection are processed by the same core.[2]

2.3. Results

We implemented Sprayer on top of DPDK[3] and conducted experiments to understand how effective Sprayer is in comparison to RSS. Similarly to the datacenter observations, we find that the low difference in delay between packets processed in different cores is not enough to significantly impair TCP performance. Moreover, we observe that the overall TCP throughput remains consistent for both low and high number of concurrent flows. Therefore, for the typical number of concurrent flows found in real workloads, Sprayer greatly improves TCP throughput, compared to RSS. Furthermore, we show that Sprayer also improves fairness, even with a higher number of flows. For a more detailed description of the results, refer to the master's thesis [Barreto 2018, §3.4].

2.4. Related Work

There are multiple works that use packet spraying to improve both efficiency and fairness in datacenter networks (*e.g.*, [Handley et al. 2017]). Yet, Sprayer is the first to bring this concept to software middleboxes. Although the basic idea is similar, the implications are different. One of the challenges of using packet spraying in datacenters is to ensure that it keeps working in the presence of asymmetries caused by link failures. In middleboxes, this problem does not exist. Instead, flow state sharing is the main concern.

Many previous works have also investigated NF state so as to scale NFs to multiple hosts (*e.g.*, [Kablan et al. 2017, Woo et al. 2018]). Despite these solutions being orthogonal to our work, they have identified similar flow-state-access patterns as we did. Moreover, one of these solutions, StatelessNF [Kablan et al. 2017], moves all NF state (per-flow and global) to a remote server, which is an elegant approach to simplifying scalability and failure recovery. Although StatelessNF could potentially replace Sprayer's flow state abstractions, it requires non-commodity technology (InfiniBand). Moreover, accessing remote states increases latency and requires extra CPU cycles [Woo et al. 2018].

Some works have tried to improve middlebox efficiency when packets go through multiple NFs (NF chaining). Solutions such as NFP [Sun et al. 2017] exploit parallelism

[2] Note that the only NF on Table 1 that needs to update flow state for every packet is Deep Packet Inspection (DPI), which means that Sprayer cannot be used to implement it. Also note that some NFs need to update *global* state for every packet. This problem affects traditional flow-based approaches as well as Sprayer. Fortunately, for some types of global states, such as statistics, looser consistency is often tolerable, which helps to reduce its impact.

[3] Data Plane Development Kit: `https://www.dpdk.org/`.

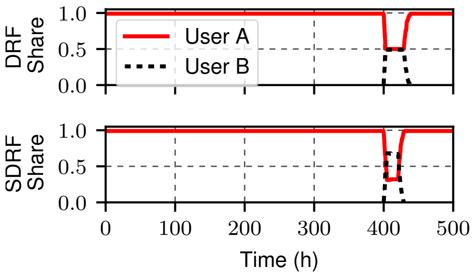

Figure 2. Share of dominant resource along time for two users when using DRF or SDRF.

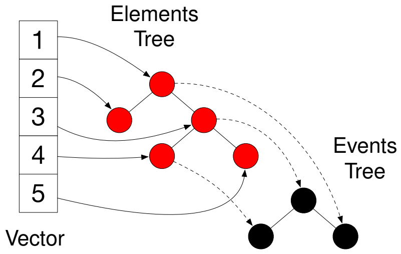

Figure 3. Illustration of a live tree with its data structures.

by processing the same packet in NFs located in different cores at the same time. These solutions, however, are specific to NF chaining and can only work for some configurations. Moreover, they require at least two inter-core transfers for every packet. Also related to NF chaining, NFVnice [Kulkarni et al. 2017] tries to improve fairness *among NFs* running on the same core, but makes no effort to improve fairness *among flows*.

3. Stateful Dominant Resource Fairness

In the second part of the master's thesis we introduce Stateful Dominant Resource Fairness (SDRF), an extension of the DRF mechanism that accounts for the past behavior of users and improves fairness in the long run. The key idea is to make users with lower average usage have priority over users with higher average usage. When scheduling tasks, SDRF ensures that users that only sporadically use the system have their tasks scheduled faster than users with continuous high usage. The intuition for SDRF is that when users use more resources than their rightful share of the system, they commit to use less in the future if another user needs. SDRF tracks users' commitments and ensures that whenever system resources are insufficient, commitments are honored.

To illustrate the benefits of considering the past in an allocation, consider an example with two users sharing a system. To simplify, assume both users have the same dominant resource (*e.g.*, CPU). User A is eager for resources and continuously submits a huge amount of tasks. In contrast, user B only uses the system sporadically. If we use DRF, whenever user B has a usage spike, both users get access to the same amount of resources—even though user B does not use the system as much as user A (see top of Figure 2). Alternatively, if we use SDRF, user A's commitment ensures that user B has access to a greater share of resources. Because of it, user B is able to complete her workload faster (see bottom of Figure 2). Also notice that, with SDRF, system's resources go back to user A sooner than if we were using DRF, which ends up causing very little impact in user A's workload. When we ensure long-term fairness, we are able to improve the allocation for users with lower demand with little impact on users with higher demand.

DRF's attractiveness stems from the properties it satisfies. We conduct a thorough evaluation of SDRF and show that it retains the fundamental properties of DRF. SDRF is strategyproof, as users cannot improve their allocation by lying to the mechanism. SDRF provides sharing incentives, as no user is better off if resources are equally partitioned. Moreover, SDRF is Pareto efficient, as no user can have her allocation improved without decreasing another user's allocation. The proof of all properties can be obtained in the master's thesis [Barreto 2018, §4.8].

3.1. Practical Considerations

Besides having desirable theoretical properties, a useful task scheduling policy must be efficiently implementable. In peak hours a scheduler may need to make hundreds of task placement decisions per second [Reiss et al. 2012]. While DRF can be efficiently implemented using a priority queue that determines which user has the highest allocation priority, when we consider the past, allocation priorities may change at any instant and the implementation cannot benefit from a priority queue. We mitigate this problem—being able to implement SDRF efficiently—introducing live tree, a data structure that keeps elements with predictable time-varying priorities sorted.

The key idea of a live tree is to focus on position-change events, instead of element priorities. When priorities follow a continuous function, elements change position whenever their priorities intersect. A live tree always has a current time associated with it and for this current time, it guarantees that elements are sorted. When the current time is updated, instead of updating every element priority, we see if any position-change event happened from the last update to the current time. Figure 3 depicts a live tree: it is composed of two red-black trees and an array. One tree is the *elements tree*, since it keeps elements sorted by priority, while the other is the *events tree*, since it tracks position-change events sorted by their time. The array is used for element lookup. In the master's thesis we describe live tree's operations in detail and provide their worst-case time complexity.

3.2. Results

To understand how SDRF performs under real workloads and how it compares to DRF, we implemented a discrete-event simulator and fed it with Google cluster traces.[4] These traces contain 30 million tasks (from either Google services or engineers) over a one-month period. Our results show that SDRF reduces the average time users wait for their tasks to be scheduled. Moreover, it increases the number of completed tasks for users with lower demands, with negligible impact on high-demand users. We also use the simulations to evaluate the performance of live tree, concluding that SDRF can be efficiently implemented in practice. For a more detailed description of the results, see the master's thesis [Barreto 2018, §4.5].

3.3. Related Work

Fair resource allocation is a prevalent research topic, both in the computer science and economics fields. Nonetheless, focus is often given to the single resource setting. Ghodsi *et al.* [Ghodsi et al. 2011] are the first to investigate the multi-resource setting under a shared computing perspective, proposing DRF. Joe-Wang *et al.* [Joe-Wong et al. 2013] extend the notion of fairness introduced by DRF to develop a framework that captures the fairness-efficiency tradeoff. Nevertheless, they assume a cooperative environment and as such do not evaluate strategyproofness. Another extension of DRF is proposed by Parkes *et al.* [Parkes et al. 2015] to account for users with different weights and zero demands. Even though the aforementioned works consider the multi-resource setting, they ignore the dynamic nature of users' demands.

[4] The source code for the discrete-event simulator as well as SDRF and Live Tree are open source and available at `https://github.com/hugombarreto/sdrf`.

Bonald and Roberts [Bonald and Roberts 2015] suggest Bottleneck Max Fairness (BMF), which also does not enforce strategyproofness, but improves resource utilization as compared to DRF. They consider dynamic demands in their analysis, arguing that for highly dynamic environments, such as networks, it is hard for users to manipulate the system. Even though the analysis of BMF considers dynamic demands, the allocation itself considers only short term usage, ignoring fairness in the long run. Kash *et al.* [Kash et al. 2014] investigate a dynamic setting where users arrive and never leave, however, they also assume that demands remain constant. Friedman *et al.* [Friedman et al. 2017] evaluate the scenario where multiple users arrive and leave the system. The focus, however, is on the fair division of resources as soon as the user arrives, limiting the number of task disruptions. There are also works that adapt DRF to packet processing [Ghodsi et al. 2012] and consider a recent past. Nevertheless, this is done to prevent limitations that arise when scheduling packets—in which resources must be shared in *time*—and not to ensure fairness and efficiency in the long run. Finally, others have focused on improving efficiency in the long run but not fairness [Grandl et al. 2016].

4. Impact

As a result of this master's thesis we have published 3 conference papers—including a publication at ACM HotNets, the major venue for discussing innovative ideas in Computer Networks—and presented a poster at USENIX NSDI. The list of works follows:

1. **H. Sadok**, M. E. M. Campista, L. H. M. K. Costa. "A Case for Spraying Packets in Software Middleboxes." In **ACM HotNets**, pp. 127–133, Nov. 2018. Qualis A1.
2. **H. Sadok**, M. E. M. Campista, L. H. M. K. Costa. "O Passado Também Importa: Um Mecanismo de Alocação Justa de Múltiplos Tipos de Recursos ao Longo do Tempo." In **SBRC**, May 2018. Qualis B2.
3. **H. Sadok**, M. E. M. Campista, L. H. M. K. Costa. "Um Mecanismo para Compartilhamento de Recursos em Nuvens Colaborativas Baseado na Credibilidade dos Usuários." In **SBRC**, pp. 458–471, May 2017. Qualis B2.
4. **H. Sadok**, M. E. M. Campista, L. H. M. K. Costa. "Per-Packet Load Balancing for Multi-Core Middleboxes." Poster in **USENIX NSDI**, Apr. 2018. Qualis A1.

Moreover, in the context of this master's thesis, we also co-authored the following journal paper:

5. R. S. Couto, **H. Sadok**, P. Cruz, F. F. Silva, T. Sciammarella, M. E. M. Campista, L. H. M. K. Costa, P. B. Velloso, M. G. Rubinstein. "Building an IaaS Cloud with Droplets: a Collaborative Experience with OpenStack." In **Journal of Network and Computer Applications**, vol. 117, pp. 59–71, Sep. 2018. Qualis A2 (Impact Factor: 3.991).

References

Barreto, H. F. S. S. M. (2018). Improving software middleboxes and datacenter task schedulers. Master's thesis, Universidade Federal do Rio de Janeiro.

Bonald, T. and Roberts, J. (2015). Multi-resource fairness: Objectives, algorithms and performance. In *ACM SIGMETRICS*.

Chiosi, M. et al. (2012). Network functions virtualisation: An introduction, benefits, enablers, challenges & call for action. Technical report, ETSI.

Friedman, E., Psomas, C.-A., and Vardi, S. (2017). Controlled dynamic fair division. In *ACM EC*.

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

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

Grandl, R., Chowdhury, M., Akella, A., and Ananthanarayanan, G. (2016). Altruistic scheduling in multi-resource clusters. In *USENIX OSDI*.

Handley, M., Raiciu, C., Agache, A., Voinescu, A., Moore, A. W., Antichi, G., and Wójcik, M. (2017). Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM*.

Joe-Wong, C., Sen, S., Lan, T., and Chiang, M. (2013). Multiresource allocation: Fairnes-efficiency tradeoffs in a unifying framework. *IEEE/ACM Trans. Netw.*, 21(6).

Kablan, M., Alsudais, A., Keller, E., and Le, F. (2017). Stateless network functions: Breaking the tight coupling of state and processing. In *USENIX NSDI*.

Kash, I., Procaccia, A. D., and Shah, N. (2014). No agent left behind: Dynamic fair division of multiple resources. *J. Artif. Intell. Res.*, 51(1):579–603.

Kulkarni, S. G., Zhang, W., Hwang, J., Rajagopalan, S., Ramakrishnan, K. K., Wood, T., Arumaithurai, M., and Fu, X. (2017). NFVnice: Dynamic backpressure and scheduling for NFV service chains. In *ACM SIGCOMM*.

Parkes, D. C., Procaccia, A. D., and Shah, N. (2015). Beyond dominant resource fairness. *ACM Trans. Econ. Comput.*, 3(1):3:1–3:22.

Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., and Kozuch, M. A. (2012). Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM SoCC*.

Sadok, H., Campista, M. E. M., and Costa, L. H. M. K. (2018a). A case for spraying packets in software middleboxes. In *ACM HotNets*.

Sadok, H., Campista, M. E. M., and Costa, L. H. M. K. (2018b). O passado também importa: Um mecanismo de alocação justa de múltiplos tipos de recursos ao longo do tempo. In *SBRC*.

Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K., and Shi, G. (2012). Design and implementation of a consolidated middlebox architecture. In *USENIX NSDI*.

Sun, C., Bi, J., Zheng, Z., Yu, H., and Hu, H. (2017). NFP: Enabling network function parallelism in NFV. In *ACM SIGCOMM*.

Woo, S., Sherry, J., Han, S., Moon, S., Ratnasamy, S., and Shenker, S. (2018). Elastic scaling of stateful network functions. In *USENIX NSDI*.