Computer Communications 120 (2018) 32–45

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

PSPAT: Software packet scheduling at hardware speed

Luigi Rizzo[1,a], Paolo Valente[b], Giuseppe Lettieri[*,a], Vincenzo Maffione[a]

[a] Dipartimento di Ingegneria dell'Informazione, Università di Pisa, Italy
[b] Dipartimento di Science Fisiche, Informatiche e Matematiche, Università degli Studi di Modena e Reggio Emilia, Italy

ARTICLE INFO

Keywords:
Packet scheduling
Lock-free queues
Multi-core
Multi-socket

ABSTRACT

Tenants in a cloud environment run services, such as Virtual Network Function instantiations, that may legitimately generate millions of packets per second. The hosting platform, hence, needs robust packet scheduling mechanisms that support these rates and, at the same time, provide isolation and dependable service guarantees under all load conditions.

Current hardware or software packet scheduling solutions fail to meet all these requirements, most commonly lacking on either performance or guarantees.

In this paper we propose an architecture, called PSPAT to build efficient *and robust* software packet schedulers suitable to high speed, highly concurrent environments. PSPAT decouples clients, scheduler and device driver through lock-free mailboxes, thus removing lock contention, providing opportunities to parallelize operation, and achieving high and dependable performance even under overload.

We describe the operation of our system, discuss implementation and system issues, provide analytical bounds on the service guarantees of PSPAT, and validate the behavior of its Linux implementation even at high link utilization, comparing it with current hardware and software solutions. Our prototype can make over 28 million scheduling decisions per second, and keep latency low, even with tens of concurrent clients running on a multi-core, multi-socket system.

1. Introduction

Allocating and accounting for available capacity is the foundation of cloud environments, where multiple tenants share resources managed by the cloud provider. Dynamic resource scheduling in the Operating System (OS) ensures that CPU, disk, and network capacity are assigned to tenants as specified by contracts and configuration. It is fundamental that the platform guarantees isolation and predictable performance *even when overloaded*.

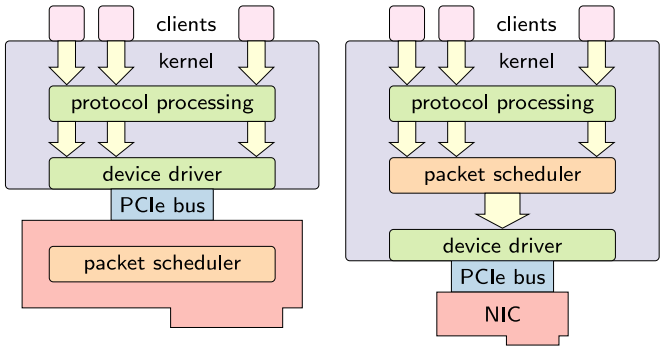
Problem and use case. In this work we focus on packet scheduling for very high packet rates and large number of concurrent clients. This is an increasingly common scenario in servers that host cloud clients: Virtual Machines (VMs), OS containers, or any other mechanism to manage and account for resources. Current hosts feature multiple processor *sockets* with tens of CPUs, and Network Interfaces (NICs) with an aggregate rate of 10–100 Gbit/s, potentially resulting in rates of 10+ Millions of packets per second (pps). Those rates, and the fact that traffic source are highly concurrent, make packet scheduling extremely challenging.

The challenge. Scheduling the link's capacity in a fair and robust way almost always requires to look at the global state of the system. This translates in some centralized data structure/decision point that is very expensive to implement in a high rate, highly concurrent environment. A Packet Scheduler that cannot sustain the link's rate not only reduces communication speed, but may easily fail to achieve the desired bandwidth allocation or delay bounds, sometimes by a large factor. We give several such examples in Sections 2.2, 6.5 and 6.6.

State of the art. OSes implement packet scheduling in software or in hardware. Software packet schedulers are included in many OSes (TC [1] in Linux, dummynet [2] and ALTQ [3] in FreeBSD and other BSD OSes), generally using the architecture shown in Fig. 1(b). Access to the Packet Scheduler, and subsequent transmissions, are completely serialized in these solutions, which can barely sustain 1–2 Mpps (see Section 6 and [2]). Hardware packet scheduling is possible on NICs with multiple transmit queues, which, as shown in Fig. 1(a), can offer each client a private, *apparently* uncontended path down to the NIC. But even this solution has one serialization point *before the scheduler*, namely the PCIe bus and its controller on the NIC's side. Those hardware resources are normally configured and/or designed to barely match the nominal link speeds in standard benchmarks, thus

* Corresponding author.
E-mail addresses: rizzo@iet.unipi.it (L. Rizzo), g.lettieri@iet.unipi.it (G. Lettieri).
[1] Present address: Google, United States.

https://doi.org/10.1016/j.comcom.2018.02.018
Received 18 January 2017; Received in revised form 26 November 2017; Accepted 19 February 2018
Available online 23 February 2018



(a) Hardware Packet Scheduler using a multiqueue NIC.

(b) Software Packet Scheduler.

Fig. 1. Common architectures for software and hardware packet schedulers in OSes.

suffering significant degradation in other conditions. As an example, several 10G and 40G NICs do not even support full line rate with minimal-sized ethernet frames, or present significant performance loss when buffers are not aligned to cache-line boundaries. In Section 2.2, we show an example of how bus contention can prevent clients from even issuing requests to the NIC at a sufficient rate.

Our contribution. In this paper we propose a different architecture for software packet schedulers, called PSPAT and shown in Fig. 4. PSPAT uses two sets of mailboxes, implemented as lock free queues, to decouple clients, the Scheduling Algorithm, and the actual delivery of packets to the NIC. This allows maximum parallelism among these activities, removes locking, and permits a flexible distribution of work in the system. It is critical that the mailboxes are memory friendly, otherwise we would have just replaced locks with a different form of contention. Section 4 discusses this problem and our solution, based on careful and rate-limited memory accesses.

PSPAT scales very well, and permits reuse of existing scheduler algorithms' implementations. Our prototype can deliver over 28 M *decisions per second* even under highly parallel workloads on a dual socket, 40 thread system. This is several times faster than existing solutions, and is achieved without affecting latency. Besides great performance, we can establish analytical bounds on the service guarantees of PSPAT reusing the same analysis done for the underlying Scheduling Algorithm.

Our contributions include: (i) the design and performance evaluation of PSPAT; (ii) a theoretical analysis of service guarantees; and (iii) the implementation of PSPAT, publicly available at [4].

Scope. PSPAT supports a class of Scheduling Algorithms (such as DRR [5], WF^{2}Q+ [6], QFQ [7]) that provide isolation and provable, tight service guarantees with arbitrary workloads. These cannot be compared with: (1) queue management schemes such as FQ_CODEL [8], or OS features such as PFIFO_FAST or multiqueue NICs, often incorrectly called "schedulers", that do not provide reasonable isolation or service guarantees; (2) heuristic solutions based on collections of shapers reconfigured on coarse timescales, that also cannot guarantee fair short term rate allocation; or (3) more ambitious systems that try to do resource allocation for an entire rack or datacenter [9], which, due to the complexity of the problem they address, cannot give guarantees at high link utilization. More details are given in Section 7.

Terminology remarks. The term "scheduler" is often used ambiguously: sometimes it indicates a *Scheduling Algorithm* such as DRR [5] or WF^{2}Q + [6]; sometimes it refers to the entire *Packet Scheduler*, i.e. the whole system that (i) receives packets from clients, (ii) uses a Scheduling Algorithm to compute the order of service of packets, and (iii) dispatches packets to the next hop in the communication path. For clarity, in this paper we avoid using the term "scheduler" alone.

Paper structure. Section 2gives some background on packet scheduling. The architecture and operation of PSPAT are presented in Section 3, which also discusses implementation details. Section 4 describes the mailbox data structure used by PSPAT. Analytical bounds on service guarantees are computed in Section 5. Section 6 measures the performance of our prototypes and compares them with existing systems. Finally, Section 7 presents related work.

2. Motivation and background

In this paper, we target systems that need to handle highly concurrent workloads of tens of millions of packets per second. These workloads are not just a theoretical possibility. High end hosts with tens of CPUs are standard in datacenters; a commodity 40 Gbit/s NIC has a peak packet rate of 59.5 Mpps; and tenants of cloud platforms may legitimately generate such high packet rates when implementing routers, firewalls, NATs, load balancers.... Contention in the use of a shared resource (the network interface) thus demands for a packet scheduler that supports the packet rates sustainable by the link. In this section we discuss some possible architectures to implement the scheduler, and highlight features of the NIC, the memory systems and Scheduling Algorithm that will be useful to design the efficient packet scheduler presented in Section 3.

2.1. Packet schedulers

A Packet Scheduler (PS) can be modeled as in Fig. 2. One or more *Queues* store packets generated by *clients* and belonging to different flows; an *Arbiter* selects the next packet to be transmitted using some Scheduling Algorithm (*SA*); a *Dispatcher* delivers the selected packet to the physical link (or the next stage in the network stack). Depending on the desired speed of operation and other constraints, the scheduling functions (Arbiter and Dispatchers) can be embedded in the NIC (*hardware scheduler*), or implemented in software as part of the operating system.

2.2. Hardware packet schedulers (and their limitations)

Modern NICs normally support multiple transmit queues to allow lock free access to different CPUs, and sometimes offer a limited choice of Scheduling Algorithms to decide the service order of queues. The entire packet scheduler thus can be offloaded to the NIC, as illustrated in Fig. 1(a). Unfortunately this architecture is inherently unable to protect the NIC against excessive requests from users, because the scheduler operates *after* the PCIe bus and its controller on the NIC's side. Both these resources can be a bottleneck, and all CPUs have equal

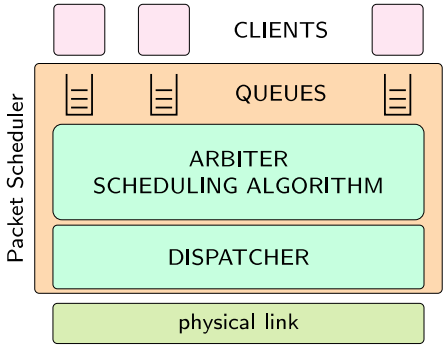


Fig. 2. High level structure of a packet scheduler. Queues receive requests from clients, an Arbiter runs the Scheduling Algorithm, one or more dispatchers move data to the physical link. Components can be implemented in software, hardware or a combination of the two.

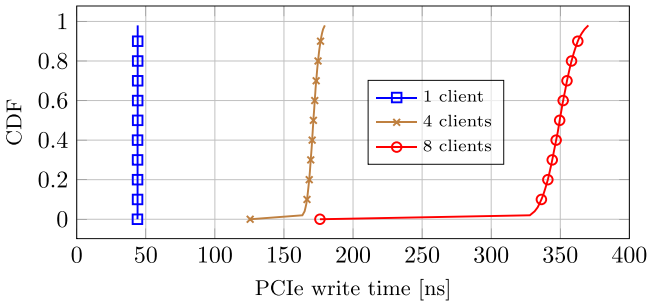


Fig. 3. CDF of the duration of a PCIe write to an Intel X520 NIC, where 1, 4 or 8 clients continuously access the same device register. The CPU stalls waiting for the access to complete. In all tests the device reaches its capacity (approximately 20 Mreq/s). Each CDF is obtained from 100,000 samples.

Table 1
Cost of locking contention for a dual Xeon processor machine, 10 CPUs each, 40 hyperthreads in total.

Users	1	2	3	4	8	20	40
Time [μs]	0.10	0.27	1.36	2.10	5.70	32.0	64.0
Total mops/s	10.0	7.4	2.2	1.9	1.4	0.61	0.55

bus access irrespective of the desired link bandwidth's distribution. It follows that one or more greedy CPUs can issue PCIe write requests at a rate that prevents others from achieving the desired link sharing.

To show this problem, we ran an experiment in which one or more CPUs concurrently issue PCIe write requests (an operation required to send packets to a queue), and we measure the distribution of PCIe write latencies, as shown in Fig. 3. Writes go through the processor's write buffer, so that at low rates they are nearly instantaneous. However, when the write buffer fills up they take as long as the NIC's controller takes to process them. In our experiment, an Intel X520 NIC (2×10 Gbit/s) takes slightly less than 50 ns per write, as shown by the curve for 1 CPU. As multiple CPUs issue writes, the total rate remains the same and each CPU has limited guaranteed access to the link. With 8 active CPUs, the median write time is 350 ns, and the time grows linearly with the number of CPUs. Thus we cannot guarantee each CPU more than 1/*N*th of the NIC's capacity.

2.3. Software packet schedulers

The above limitations (or the absence of scheduling support in the NIC) can be overcome by software packet schedulers, which normally sit between protocol processing and the device driver as shown in Fig. 1(b). Implementations in commodity OSes generally make clients (and interrupt handlers) execute the Arbiter and Dispatcher by gaining exclusive access to the queues and the Scheduling Algorithm's data structures for each packet inserted or extracted. Arbitration occurs before sending requests to the NIC so the bus cannot be saturated anymore. Unfortunately the performance of such an architecture under concurrent workloads is generally limited to 1–2 Mpps (see Section 6 and [2]), mostly because of heavy lock contention. As an example, Table 1 shows the cost of a 100 ns critical section protected by a spinlock as the number of concurrent users grows; this experiment mimics the behavior of multiple clients accessing a packet scheduler. We can see that the per-client cost (first row) grows much more than linearly, making the cost of the SA (10–50 ns) almost negligible. The total throughput (last row) also decreases significantly with the number of users.[2]

[2] Recent versions of Linux optimize lock access in the scheduler dequeue phase giving a speedup of roughly a factor of 2.

2.4. Memory communication costs

The key idea of PSPAT, described later in Section 3, is to replace the expensive per-NIC global lock with N shared memory lock-free mailboxes and a thread that polls them. The performance of our solution depends on how efficiently clients and the polling thread can communicate through these mailboxes.

To characterize the behavior of such shared memory communication we run some tests on a dual socket Xeon E5-2640, 2.5 GHz system. Two threads, a writer and a reader, access one or more shared 64-bit variables V_i stored in different cache lines (few enough to fit in the L1 cache).

For brevity, we report the results from an experiment of interest: the writer keeps incrementing one or more V_i variables, and the reader counts the number of reads and of different values seen over time. In this experiment the reader alternates bursts of very quick reads (when the cache line is believed to be unchanged) with individual slow reads while the cache line is stale and updated. Writes normally go through a write buffer, but they may stall too if the write buffer cannot run at sufficient speed due to memory contention. The rate of visible updates and the duration of read/write stalls[3] depend on the "memory distance" between reader and writer, and Table 2 shows these parameters in various settings. It is particularly important to note that communication between sockets is (relatively) slow, so protocols should be designed in a way to amortize the latency and long stalls.

2.5. Scheduling Algorithms

A key component of a packet scheduler is the *Scheduling Algorithm* (SA), which is in charge of sorting packets belonging to different "flows" (defined in any meaningful way, e.g., by client, network address or physical port), so that the resulting service order satisfies a given requirement. Examples include (i) giving *priority* to some flows over others; (ii) applying *rate limiting* to individual flows; (iii) proportional share scheduling, i.e., dividing the total capacity of the link proportionally to "weights" assigned to flows with pending transmission requests ("backlogged" flows). An SA is called "work conserving" if it never keeps the link idle while there are backlogged flows.

Perfect proportional sharing can be achieved by an ideal, infinitely divisible link that serves multiple flows in parallel; this is called a "fluid system". Physical links, however, are forced to serve one packet at a time [10], so there will be a difference in the transmission completion times between the ideal fluid system and a real one, adding latency and jitter to the communication. A useful measure of this difference is the Time Worst-case Fair Index (T-WFI)[11], defined as follows:

Definition 1 (*T-WFI*). The maximum absolute value of the difference between the completion time of a packet of the flow in (i) the real system, and (ii) an ideal system, if the backlog of the flows are the same in both systems at the arrival time of the packet.

Each SA is characterized by a certain T-WFI metric, which depends on the inner working of the specific algorithm. T-WFI matters as it measures the extra delay and jitter a packet may experience: we would like to have a small upper bound for it, possibly independent on the number N of flows. For any work-conserving packet system, the T-WFI has a lower bound of one maximum sized segment (*MSS*), proven trivially as follows. Say a packet A for a flow with a very high weight arrives just a moment after packet B for a low weight flow. In a fluid system, upon its arrival, A will use almost entirely the link's capacity; in a packet system, A will have to wait until B has completed.

The theory. Some systems replace Scheduling Algorithms with bandwidth limiting, or heuristics that give proportional sharing only

[3] See Appendix A for details on how stall times are measured.

Table 2
Cost of read or write memory operations on a dual socket Xeon E2-2640. Reader and writer can run on two hyperthreads of the same physical core (HT–HT), on two different physical cores (CPU–CPU) or on different processor sockets (SKT–SKT).

Measured parameter	HT–HT	CPU–CPU	SKT–SKT
Different values per second seen by reader	75 M	20 M	5 M
Read stall latency	10–15 ns	50 ns	130–220 ns
Write stall latency	–	15 ns	100 ns

over coarse time intervals (e.g., milliseconds). These solutions are trivial but not interesting, because the large and variable delay they introduce disrupts applications. As the acceptable delays (hence, T-WFI) become shorter, the problem becomes challenging and we enter into a territory of cost/performance tradeoffs. We have efficient *Weighted Fair Queueing* algorithms [11,12] that match the T-WFI lower bound, with $O(\log N)$ cost per decision, where N is the number of flows. Fast variants of Weighted Fair Queueing, such as QFQ [7] and QFQ+ [13] achieve $O(1)$ time complexity per decision, at the price of a small *constant* increase of the T-WFI, which remains within $O(1)$ of the ideal value. On the other end of the spectrum, algorithms such as DRR (Deficit Round Robin [5], also known as Weighted Round Robin, WRR), have $O(1)$ time complexity but a poor $O(N)$ T-WFI.

Some numbers can help appreciate the difference among Scheduling Algorithms. With 1500 byte packets (1.2 μs at 10 Gbit/s), a DRR algorithm with just 25 busy flows will cause at least 30 μs of latency even for high weight flows (the worst case is much higher, see Section 5.2, also depending on the scheduling quantum used by DRR). In contrast, in a similar scenario, with a good algorithm such as QFQ, a high weight flow will suffer 2–3 μs of extra latency irrespective of the number of flows.

The practice. Implementations of QFQ and QFQ+ are included in commodity OSes such as Linux and FreeBSD. Their runtime cost is comparable to that of simpler schedulers such as DRR, which offers much worse T-WFI. All of the above implementations can make a scheduling decision in 20–50 ns on modern CPUs. Because of its simplicity, DRR/WRR is widely used in hardware schedulers on popular high speed NICs.

3. PSPAT architecture

Having identified the problems in existing packet schedulers, we now discuss how our design, PSPAT, addresses and solves them. We split the components of the Packet Scheduler as shown in Fig. 4, so that we can operate clients, Arbiter and Dispatcher(s) in parallel. The components of our system are:

M clients each with its own *Client Mailbox*, CM_i. Clients can be VMs, containers, processes, threads;
C CPUs on a shared memory system (each hyperthread counts as one CPU);
C Client Lists CL_c, indicating clients recently active on each CPU;
1 Arbiter thread, running a Scheduling Algorithm;
N flows in which traffic is aggregated by the Arbiter;
T Transmit queues on the Network Interface (NIC),
0–T TX mailboxes TM_i, and an equivalent number of Dispatcher threads feeding the NIC.

Mailboxes and client lists are implemented as lock free, single-producer single-consumer queues. There are no constraints on the number of clients (M), CPUs (C), flows (N), and transmit queues (T), or on how traffic from different clients is aggregated into flows. In particular, M and N can be very large (thousands).

PSPAT operates as follows:

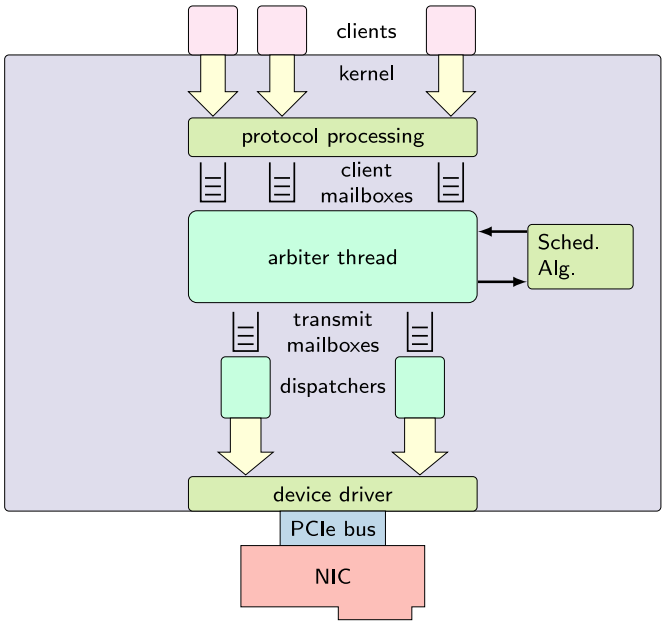


Fig. 4. The architecture of PSPAT. A dedicated Arbiter thread runs the Scheduling Algorithm, communicating with clients and transmitter threads (Dispatchers) by means of lock-free queues.

1) A client x, who wants to transmit while running on CPU c, pushes packets into its Client Mailbox, CM_x, and then appends x to CL_c only if x is not already the last entry in CL_c;
2) the Arbiter continuously runs function `do_scan()`, in Fig. 5, to grab new requests from the mailboxes of active clients on all CPUs, pass them to an SA, trim the Client Lists, and use a leaky bucket algorithm to release packets to the TX mailboxes;
3) finally, Dispatcher threads drain the TX mailboxes and invoke the device driver for the actual transmission.

By reducing serialization to the bare minimum (only the Arbiter runs the Scheduling Algorithm), and removing lock contention through the use of private, lock free mailboxes, PSPAT can achieve very good scalability, opening the door to performance enhancements through pipelining and batching. Note that the leaky bucket rate limiting also protects the PCIe bus from saturation. Of course the devil is in the details, so the rest of this section and Section 4 describe in depth the various components of PSPAT and discusses how they interact with the hardware.

3.1. Clients

Each client pushes packets into its private mailbox (CM) to communicate with the Arbiter. A client that transmits many packets from CPU C (before being preempted) will append itself into CL_c at most on the first transmission. For all the subsequent transmissions it will quickly verify that the last entry in CL_c already contains itself an do nothing. As a result, under high load the average cost of a client updating CL_c is approximately the cost of reading from the L1 cache (as the append operation is amortized over many packets). The overall cost of clients updating the CL_c is then proportional to the average context switch rate, which usually does not exceed 1000–2000 context switches per second per CPU. The cost of a context switch is anyway at least an order of magnitude higher of the cost of updating a CL (even in the worst case of a dual socket CPU); should the context switch rate be too high, the system performance will suffer because of the CPU scheduling overhead, irrespective of the Packet Scheduling architecture. Note in particular that the cost of updating the CLs is not proportional on the

```
1  void do_scan() {
2    t_now = time();
3    for (i=0; i < CPUs; i++) {
4      for (cli in CL[i]) {
5        while ((pkt = extract(CM[cli])) != NULL) {
6          SA.enqueue(pkt);
7        }
8      }
9      <trim CL[i] leaving last entry>
10   }
11   while (link_idle < t_now) {
12     pkt = SA.dequeue();
13     if (pkt == NULL) {
14       link_idle = t_now;
15       return; /* no traffic */
16     }
17     link_idle += pkt->len / bandwidth;
18     i = pkt->tx_mbox_id;
19     <enqueue pkt in TM[i]>
20     i = pkt->client_id;
21     <clear one entry in CM[i]>
22   }
23   for (i=0; i < TX_QUEUES; i++) {
24     if (!empty(TM[i])) {
25       <notify dispatcher[i]>
26     }
27   }
28 }
```

Fig. 5. Simplified code for the Arbiter. The first loop pushes packets to the SA. The second loop invokes the SA and emulates the link. The last loop submits the scheduled packets to the Dispatchers.

number of clients M.

The lack of notifications requires the Arbiter to scan all mailboxes that may contain new requests since the previous scan. Client Lists let us keep the number of mailboxes to scan within $O(C)$. A Client List CL_c contains more than one entry only if new clients have been scheduled on CPU c *and* have generated traffic since the previous Arbiter scan. This is a very rare event, as the time to schedule a thread is comparable to the duration of a scan (see Section 3.4).

Clients may migrate to other CPUs without creating data races (because the mailbox is private to the client) or correctness problems: even if a mailbox appears in multiple Client Lists as a result of migration(s), its owner will not get any unfair advantage.

Backpressure. Conventional scheduler architectures often return an immediate error if a packet is dropped locally; this information provides backpressure to the sender (e.g., TCP) so that it can react. A drop may happen because of underlying queues are full or because of SA specific policies (e.g., CODEL may drop to keep the delay under a threshold). PSPAT offers immediate reporting when the Client Mailbox is full. Policy-related drops cannot be reported immediately, because they happen in the SA, which does not run in the context of the client thread. The Arbiter thread can however set a backpressure flag in the Client Mailbox, so that the drop can be reported as soon as the client attempts the next transmission.

3.2. Flows

How traffic is assembled into flows (which are then subject to scheduling) depends on how the scheduler is configured at runtime. Some settings may split traffic from one client into different flows, others may aggregate traffic from multiple clients into the same flow. PSPAT is totally agnostic to this decision, as it delegates flow assembly to the Scheduling Algorithm. Our architecture also guarantees that traffic from the same CPU is never reordered.

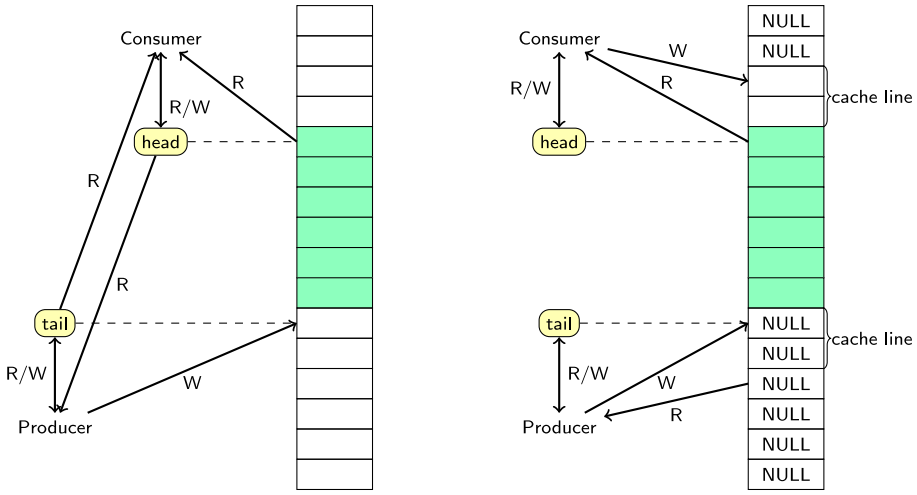
3.3. Dispatchers

NICs implement multiple queues to give each client an independent, uncontended I/O path, but on the NIC's side, more queues can be detrimental to performance, as explained in Section 2.2. PSPAT provides separate I/O paths through Client Mailboxes, which have much better scalability in size and speed, and lets us keep the number of Dispatchers, T, as small as it suffices to transfer the nominal workload to the next stage (typically a NIC). If transmitting packets to the next stage is fast, as it is the case in frameworks like netmap [14] and DPDK [15], dispatching can be done directly by the Arbiter thread. Conversely, if transmission is expensive, the overall throughput can be improved by means of separate Dispatcher threads running in parallel to Arbiter.

3.4. The Arbiter

The body of the Arbiter's code, function `do_scan()`, in Fig. 5, is structured in three blocks. The first block takes a timestamp `t_now`, and drains the CMs, submitting the packets to the SA. All packets arrived before `t_now`, are guaranteed to be in the scheduler, and the number of packets extracted from each CM is limited by the mailbox's size. For each CPU, this block also trims the associated CL to keep only the last element, which is likely to reference the client that is currently running on the CPU; in this way we clear most entries in the CL and at the same time we save the current client from adding itself to the CL for each packet it wants to transmit. Similar to what happens for clients (Section 3.1), the cost of trimming is proportional to the overall context switch rate. In the second block, packets are extracted from the SA and pushed to the TX mailboxes in conformity with the link's rate up to time `t_now`,; CM entries are released accordingly. This guarantees correctness of the scheduling order (all packets arrived before `t_now`, are in the scheduler) and prevents TX mailboxes from overflowing. Finally, the third block notifies the Dispatchers, so that TX mailboxes can be efficiently drained in batches [16].

It is important that the Arbiter completes a round very quickly, to avoid adding too much latency, and that it does not spend a large fraction of its time stalled on memory reads; the latter is a real risk, as it has to scan $O(C)$ CMs updated by clients. C should not be more than 100 even for a reasonably large system. The Arbiter accesses a small part of each CM, and does it frequently, so a CM that needs no service (either empty, or completely full), is likely to be in the L1 or L2 cache of the Arbiter. This results in access times in the order of 2–4 ns each (we have measured such values on a dual-socket E5-2640 system). It follows that



(a) Lamport-based. (b) Improved FastForward.

Fig. 6. Lockless mailboxes implementation. Entries containing a valid value are colored. Memory sharing between producer and consumer affects performance. Obviously, array entries need to be shared. The classical (Lamport) scheme on the left also has additional memory sharing on the head and tail pointers, whereas the scheme on the right uses special values to indicate the validity of an entry, so head and tail are not shared.

a full scan of idle clients collectively takes in the order of 200 ns, comparable to the cost of a single cache miss in the architectures we target (see Table 2). Section 4 describes how mailboxes can be implemented to reduce cache misses and amortize read or write stalls.

3.5. Avoiding busy wait

PSPAT has been designed to be deployed on highly loaded systems, where the Arbiter thread has (almost) always some work to do. In this situation it can busy wait to minimize latency and maximize throughput; this strategy is also convenient for energy efficiency as sleeping or blocking for very short time intervals (1–20 μs) can be unfeasible or add unacceptable per-packet overhead, as explained in [17, Section 4.3].

However, if PSPAT is deployed on systems where the load can be low for long periods, we would like to make the Arbiter sleep when there is no work to do. This happens (i) while the current packet completes transmission, or (ii) when all the Client Mailboxes (CMs) are empty. In the first case, the Arbiter can just `sleep()` until the (known) time when the link will be idle again. The case of empty CMs is trickier because new requests may arrive at any time, including while the Arbiter is deciding whether to block, and we need a notification from clients to restart. Such notifications are expensive, so they should be avoided if possible. This is a common problem in multicore OSes and normally handled as follows: (1) the Arbiter spins for some short interval T_w when idle, in case new requests come soon; (2) when the Arbiter is asleep, clients run the Arbiter's functions themselves if there is no contention with other clients, and activate the Arbiter's thread otherwise. Combining these two strategies, we guarantee at most one contention period and one notification every T_w seconds. The value for T_w can be chosen by experimentation, in such a way to achieve performance in presence of traffic, and very low overhead when there is light or no traffic.

To put number in context, if the Arbiter has no more requests to process and no traffic arrives in the next millisecond (or more), spinning for at most 20–50 μs before deciding to block does not impact CPU consumption in practice. On the other hand, if some traffic does come while the Arbiter is spinning we save the (high) cost of a block-notify-wakeup sequence, which impacts both Arbiter and clients. Since the notification can easily cost 2–5 μs for the clients (on a multi-socket machine), if the Arbiter spins at least for 20–50 μs then the client will not spend on notifications more than 10% of its CPU time. In particular, this strategy avoids pathological situations where there is a notification every 1–2 packets under moderate or high load.

In summary, if power consumption or CPU sharing require to avoid busy wait, this can be done without too much harm to performance or latency. An experimental evaluation of the impact on PSPAT, however, is not in the focus of this paper. Our previous work [17–19] covers this space, analyzing similar high-rate packet processing systems, including evaluation of sleeping and notification costs, and impact of overall throughput, latency and energy efficiency.

4. Mailboxes

In this section we motivate and describe our mailbox implementation, which is crucial to PSPAT high-throughput operation. Mailboxes are implemented as circular queues of a fixed number of *slots*, accessed concurrently by only two threads: a *producer*, inserting in the queue `tail`, and a *consumer*, extracting from the queue `head`. Each queue slot can accommodate a single memory pointer. In our case, Client Mailboxes and TX mailboxes contain pointers to packet descriptors (`sk_buff`'s in Linux). Client Lists are also implemented as mailboxes: they contain pointers to Client Mailboxes.

Locking operations in multi-core/multi-socket systems are very expensive, therefore insertion and extraction operations in shared memory queues are usually implemented in a lock-free fashion.

The common way to achieve this is by using Lamport's algorithm [20] (see Fig. 6(a)). In this algorithm, the `tail` and `head` indices are shared among the producer and the consumer. On insertion, the producer compares `head` and `tail` to make sure that there is room available, then it first updates the queue's `tail` slot, and then (circularly) increments `tail`. On extraction, the consumer compares `head` and `tail` to make sure that the queue contains some new message, then it first extract the message in the `head` slot, and then (circularly) updates `head`.

There are a couple of issues with this algorithm in modern, multi-core/multi-socket systems, where the producer and consumer are meant to run on separate CPUs/hyperthreads. First, the algorithm correctness relies on the preservation of the ordering between two writes (e.g., writes into the `tail` slot and into the `tail` index, during insertion). On some architectures, this may require the interposition of costly memory barriers. Second, the sharing of the `tail` and `head` indices between the producer and the consumer may cause write and read operations to wait for the remote caches to update or deliver their cache lines. When these wait times cannot be hidden by the CPU internal pipeline, they cause throughput-limiting *stalls* in the execution (see Section 2.4).

4.1. PSPAT's fast mailboxes

PSPAT avoids the sharing of the `tail` and `head` indices through the technique presented in FastForward [21] (see Fig. 6(b)):

- empty slots contain a special value (e.g. a NULL pointer);
- `tail` and `head` are private to the producer and consumer, respectively;
- the producer waits for an empty `tail` slot, before writing a new message to it and advancing the index;
- the consumer waits for a non-empty `head` slot to read a new messages, mark the slot as empty and advance the index.

The main advantage of this technique is that no memory barriers are needed between queue and pointer updates, because producer and consumer only look at the queue. Moreover, cache conflicts between the producer and the consumer are now limited to two situations: nearly-full and nearly-empty queues, when the two threads are likely accessing the same cache line, thus causing lots of read and write stalls due to invalidation/update messages going back and forth from their caches.

Conflicts in the nearly-full case are easily avoided if the producer refrains from adding messages to a new cache line until it is completely free; this can be ascertained by looking at the first slot in the cache line further down the queue, and checking it's NULL. In Fig. 6, this is illustrated by having the producer read the slot one cache line ahead of the `tail` one.

Nearly-empty queue states, however, cannot be avoided: the consumer cannot wait until the producer has filled a cache line, since the producer may have nothing more to transmit for an arbitrary long period of time. To avoid write–write conflicts between the consumer releasing slots and the producer adding new messages, we let the consumer release the slots only when it has consumed a full cache line (see the write arrow from the consumer in Fig. 6).

There remains the problem of read–write cache conflicts occurring when the producer updates multiple slots in the same cache line while it is accessed by the consumer. We address this problem by rate-limiting read accesses to the mailboxes. For example, the Arbiter rate-limits its own accesses to each Client Mailboxes to one read every 1–2 μs. Since write stalls cost in the order of 100 ns (see Table 2), each client will not stall for more than 5–10% of its time.

5. T-WFI in PSPAT

An important quality metric of a Scheduling Algorithm is the T-WFI (Definition 1). A larger T-WFI is associated to more jitter and delay, with obvious consequences on network performance. PSPAT does not define new algorithms, but reuses existing ones within the Arbiter. The purpose of this section is therefore to determine the overall T-WFI of PSPAT given that of the underlying Scheduling Algorithm, T-WFI$_{SA}$.

5.1. T-WFI analysis

The literature contains an evaluation of the T-WFI$_{SA}$ for several Scheduling Algorithms that we can use in PSPAT. For example, the analysis in [22] shows that the T-WFI of a complete Packet Scheduler is made of a first component, say T-WFI$_{SA}$, accounting for intrinsic inaccuracies of the Scheduling Algorithm, plus a component due to external artifacts (such as the presence of a FIFO in the communication device.)

In PSPAT, the second component depends on how we feed the Scheduler Algorithm and the NIC. Here we evaluate this under the following assumptions: (i) the Arbiter, each client, the NIC and the link must all be able to handle B bits/s; (ii) the Arbiter calls `do_scan()`, to make a round of decisions every Δ_A seconds; (iii) each Dispatcher processes the Transmit Mailbox every Δ_D seconds; (iv) the NIC serves its queues using round-robin (trivial to implement in hardware and avoids starvation).

Under these assumptions, the Arbiter may see incoming packets and pass them to the SA with a delay Δ_A from their arrival. This quantity just adds to T-WFI$_{SA}$, without causing any additional scheduling error, at least in Scheduling Algorithms where decisions are made only when the link is idle.

We call a "block" the amount of traffic released to the Transmit Mailboxes in every interval Δ_A. This can amount to at most $B\cdot\Delta_A$ bits, plus one maximum sized packet L.[4] The quantity exceeding $B\cdot\Delta_A$ is subtracted from the budget available for the next interval Δ_A, so the extra traffic does not accumulate on subsequent intervals.

Since the Arbiter releases up to one block of data at once, and Dispatchers send those packets to the NIC in parallel, the order of transmission may be different from the one decided by the Arbiter. Let us first assume that Dispatchers operate with $\Delta_D = \Delta_A$ and are synchronized with the Arbiter. This adds a delay term Δ_D to the service of packets, and also a potential reordering within the block, which amounts (in time) to the size of the block itself, i.e. Δ_D.

When $\Delta_D \neq \Delta_A$ and/or Dispatchers are not synchronized, a further complication occurs, as the link may receive at once up to $B\cdot(\Delta_D + \Delta_A)$ bits, more than the capacity of the link, before the next round of transmissions. The excess block $B\cdot\Delta_A$ that remains at the end of the round will in turn be reordered together with the block from the next round (which this time is within the link's capacity). The number of Δ_D intervals to drain packets from the excess block will be proportional to $k = \Delta_A B/L$, or the number of maximum sized packets in the block. We report the proof of this property in the appendix, to not interrupt the flow. For practical purposes, Δ_A is 1–2 μs, and even on a 40 Gbit/s interface the value of k is less than 5. On a 10 Gbit/s and lower, for all practical purposes we can assume $k = 1$.

In conclusion, putting all pieces together, we have

$$\text{T-WFI} = \text{T-WFI}_{\text{SA}} + \Delta_A + (2 + k)\Delta_D. \tag{1}$$

5.2. T-WFI examples

To put numbers into context: from [22] we know that

$$\text{T-WFI}^{(k)}_{\text{QFQ}} = 6\frac{L_k}{\phi_k B} + \frac{L - L_k}{B}, \tag{2}$$

$$\text{T-WFI}^{(k)}_{\text{DRR}} = \left(\frac{1}{\phi_{min}} + \frac{1}{\phi_k} + N - 1\right)\frac{L}{B}. \tag{3}$$

The T-WFI depends on the weight of each client. In the equations, N is the number of clients, and L_k is the maximum packet size for client k. ϕ_k is the weight of client k, $0 < \phi_k < 1$ and $\sum_{k=1}^{N}\phi_k = 1$, ϕ_{min} is the minimum weight among all clients.

In practice, QFQ has a T-WFI of about $6/\phi_k$ times the maximum packet transmission time (L/B), whereas for DRR the multiplying factor has a large term $1/\phi_{min}$ plus a linear term in the number of clients. For a 10 Gbit/s link and $L = 1500$ bytes, $L/B = 1.2\mu$s. Assuming weights ranging from 0.005 to 0.5, the client with the highest weight will have $\text{T-WFI}^{(k)}_{\text{QFQ}} = 12$, $L/B = 14.4\mu s$ irrespective of N. For DRR, the dependency on N gives $\text{T-WFI}^{(k)}_{\text{DRR}} = 226$, $L/B = 271.2\mu s$ for 25 clients, and 301 L/B, or 361.2 μs for 100 clients. In comparison, the additional term $2\Delta_A + 2\Delta_D$ (between 2 and 4 μs) introduced by PSPAT is small or negligible.

6. Experimental evaluation

We now evaluate PSPAT architecture and compare its performance

[4] The Arbiter releases all packets that start transmission in the current interval, so the last one may complete after the end of the interval.

with existing alternatives. As described in Sections 3 and 4, PSPAT uses convenient data structures for fast operation and to limit or avoid contention between the clients. Our goal here is therefore to check that an implementation of PSPAT behaves well in the following worst-case conditions:

- Under high input load, e.g. 10 Mpps or more;
- As the number of clients grows to fill up the available CPU cores in the system;
- A combination of the two cases above.

In particular we would like to verify that as the input load and/or number of clients increase, the following behavior holds:

- The throughput increases until a saturation point, which is the maximum load that the Arbiter can sustain in the given configuration. It is not acceptable that the throughput drops significantly with increasing load, as this is a sign that the Packet Scheduler is being slowed down by some kind of congestion.
- The latency distribution degrades gracefully and as predicted by the T-WFI of the PS (Section 5). It is not acceptable that the latency figure grows unbounded because of uncontrolled congestion.
- The PS behavior complies with the Scheduling Algorithm configuration (e.g., rate allocation to the different flows, priorities). The PS may fail to invoke the SA in the expected way because of contention on the PS data structures.

Note that our experiments use proper Scheduling Algorithms (e.g. QFQ, DRR), which need to access some global state to make decisions. In particular we leave out queue management schemes such as Linux `fq_codel` or `pfifo_fast`, as they make local decisions and so they are not real schedulers. With local decisions the clients can run the queue management scheme in parallel on different cores, so that our centralized architecture is not necessary.

6.1. Two PSPAT implementations

In order to better evaluate the architecture of PSPAT, we have built two versions of PSPAT, one in kernel, one in userspace. The in-kernel PSPAT is a completely transparent replacement of the Linux packet scheduler. Once enabled with a `sysctl`, packets are intercepted in `_dev_queue_xmit()` and put in the Client Mailboxes. After being scheduled, the packets are delivered to the network device through `dev_hard_start_xmit()`. The Arbiter is implemented as a kernel thread which uses the TC [1] subsystem from the Linux kernel (also known as the “qdisc” layer) as the Scheduling Algorithm. This gives a perfectly fair comparison with TC as we use exactly the same code and datapaths for the SA. On the other hand, the reuse of existing code brings in some unnecessary performance limitations, as discussed below.

The userspace version of PSPAT uses scheduler implementations taken from the dummynet [2] link emulator, and optimized for running in a single thread. It supports multiple network I/O mechanisms through UDP sockets, BPF sockets, or high speed ports provided by the netmap [14] framework and the VALE [23] software switch. Since PSPAT already implements a rate limiter, it is not necessary to add another one (the HTB node) as in TC. The in-kernel version of PSPAT requires two memory accesses (and cache misses) per packet: one to fetch the `skbuf` from the CM, one to fetch the packet size and metadata from the `skbuf`. On top of this, classification through TC consumes some extra time. In contrast, hardware schedulers can make their decisions using pre-classified traffic (one client = one flow), and the packet length is readily available in the transmit ring. The userspace version of PSPAT permits an evaluation in conditions similar to those of hardware schedulers. Each client produces a different flow, so the only information needed for scheduling is the packet length, which is 2 bytes instead of the 8 bytes of an `skbuf`, and is available in the CM without an extra pointer dereference. Communication through the mailbox is thus a lot more efficient. Finally, clients also perform the role of Dispatchers, once the Arbiter has cleared packets for transmission.

Overall, the userspace PSPAT can be a lot faster and therefore useful to explore performance improvements in the implementation of Scheduling Algorithms and the Arbiter architecture. Moreover, it supports operation on platforms where we cannot replace the running kernel, or we do not want to (e.g., userspace I/O frameworks and protocol stacks).

6.2. Testbed description

For our experiments we use two hardware platforms called I7 and XEON40, and several 10 G and 40 G NICs. Platform I7 is a single-socket i7-3770K CPU at 3.5 GHz (4 physical cores, 8 threads), 1.33 GHz DDR3 memory, and runs Linux 4.13. On I7 we can use dual port Intel NICs: the X520 (10 Gbit/s, 8 PCIe-v2 lanes at 5 Gbit/s each) and the XL710 (40 Gbit/s, 8 PCIe-v3 lanes at 8 Gbit/s each), both including a hardware DRR scheduler. Our tests use one or two NIC ports, connected back-to-back. Since we have physical access and complete control on I7, we are able to replace the Linux kernel and use the in-kernel PSPAT implementation, in addition to the userspace one (see Section 6.1).

Platform XEON40 is a dual socket system with Xeon E5-2640 CPUs v4 at 2.4 GHz (max turbo 3.4 GHz), with 10 cores (20 threads) for each socket, 2.133 GHz DDR4 memory, and running Linux kernel 3.10. XEON40 does not have fast NICs so tests here are done on the loopback interface, using different UDP ports to avoid contention on the destination socket. Moreover, on XEON40 we do not have the freedom to change the kernel and so we are only able to test the userspace PSPAT implementation.

6.3. Experiment methodology and configuration

In all the experiments, clients are traffic generators with configurable rate, packet size and burst size. With the in-kernel PSPAT, clients are either processes sending on UDP sockets, or `pkt-gen` transmitters. The `pkt-gen` program is a fast UDP packet generator that uses the netmap API [14]; although netmap normally bypasses the whole Linux kernel stack, we use a configuration (*emulated* netmap mode) that lets the packets go through the standard Linux network device (qdisc) layer, like the similarly named Linux kernel module [24,25]. In the userspace implementation clients are instead simple generator threads that directly write to the Client Mailboxes.

Usual care is taken to achieve reliable and meaningful measurements:

- Disable high C-states, to avoid the high and unpredictable latency of CPU wake ups.
- Lock CPU clock speed to the maximum value, to avoid dynamic frequency adjustments.
- Pin threads to CPU cores to limit interferences due to process scheduling. The Arbiter thread, if present, is allocated on a separate core, with the other hyperthread normally left idle to avoid interference. The assignment strategy of client threads is experiment specific.
- Configure interrupt moderation on the NICs to vendor-recommended values, that allow for interrupt batching with low latency cost.
- For latency tests, on the receiver side, use of busy wait instead of notifications. This removes the unrelated receiver wake-up latency from the measurement.
- For throughput tests, receivers bind UDP sockets or netmap ports, but normally do not read from them to minimize the receive side processing costs and focus the measurements on the transmit side.
- Repeat each single measurement 10 times, computing the arithmetic

mean and standard deviation. The standard deviation is not shown in the plots when it is smaller than 5% of the average.

Unless specified otherwise, in all experiments the system is configured for the worst case or to stress the phenomena under investigation:

- Using minimum-sized packets (60 bytes) for throughput measurements allows to stress the systems under the highest possible load.
- Using MTU-sized packets (1500 bytes) for latency tests cause the highest T-WFI, and so the highest latency.
- The Scheduling Algorithm (DRR or QFQ) uses a quantum of one packet, and queue size and bandwidth large enough not to act as a bottleneck.
- Clients have the same weight, and send as fast as possible.
- PSPAT uses $\Delta_A = 1000$ ns, as explained in Section 3.4
- On XEON40 the Arbiter thread runs on the second socket, while clients are allocated starting from the first socket; in this way we can measure the worst case cost of the interaction with clients.

6.3.1. Packets per second vs. decisions per second

For packet processing systems, the load has little dependency on the packet size, so the metric of interest for throughput is normally "packets per second" (pps). When the packet transmission time is very short (say, below 500 ns), there is no measurable advantage in scheduling individual packets, and it may be preferable to make a single decision on multiple packets for each flow, (say, up to 500–1000 ns worth of data, if available). If one performs this aggregation, the "pps" figure may be deceiving, and instead one should report the number of "decisions per second". The userspace version of PSPAT supports aggregation, but here *we only report results with aggregation disabled* (i.e. the worst case), so its "decisions per second" and "pps" are the same.

6.3.2. T-WFI vs. latency distribution

The T-WFI cannot be measured directly unless we can identify the worst case scenario. Furthermore, the theoretical analysis abstracts from real world phenomena such as lock contention, cache misses, congestion, which ultimately lead to variable processing times and latency. We thus look at a related metric, namely the latency distribution in one-way communication between a client and a receiver.

6.4. Throughput experiments

Our first set of experiments measures the maximum throughput achievable with multiple UDP senders in the following scenarios: (i) no scheduling (baseline); (ii) scheduling done by TC; and (iii) scheduling done by PSPAT. In order to measure the maximum throughput, we configured the Scheduling Algorithm with a very high link rate (i.e. 500 Gbit). In this way the link emulation never acts as a limitation, and we are able to see the bottlenecks in the network stack, the packet scheduler or the clients.

Fig. 7 shows the results on I7 with the in-kernel PSPAT and a 40 Gbit/s NIC, using up to 100 UDP clients distributed over the first 6 hyperthreads (the first three 3 physical cores). Clients are allocated on the hyperthreads in a round-robin fashion, filling the two hyperthreads of a physical core before using the next physical core. With no schedulers, clients can work in parallel on the 8 NIC queues, and throughput increases monotonically until the 6 hyperthreads are fully utilized, which happens around 10 clients. Note that the slope of the curve for 1–6 clients alternates between two values, because of the order used to allocate the clients. This happens because running two greedy UDP clients on the two hyperthreads of a same physical core slows down both clients considerably: the pair delivers only 1.3 times the traffic delivered by a single client if the other hyperthread is idle. With more than 10 clients the system saturates almost perfectly at 5.7 Mpps, with only minor performance degradation due to increased memory pressure and process scheduling overhead. When using the Linux TC packet

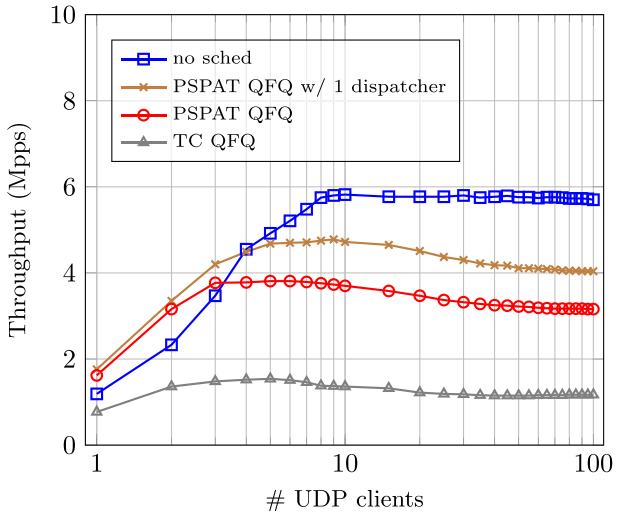


Fig. 7. Throughput in pps on I7 using UDP socket senders. For PSPAT experiments, the in-kernel version is used.

scheduler (configured with QFQ), throughput is much lower in general (up to 80% lower than the baseline), and starts degrading after 5 clients, where it peaks at 1.5 Mpps. The huge difference w.r.t. the baseline shows that the Linux packet scheduler is the bottleneck. With many clients TC saturates at 1.18 Mpps. The throughput degradation in this case is also due to increased contention on the scheduler lock. The other two curves report the performance of PSPAT configured with QFQ; in the first one transmission is carried out directly by the Arbiter, while in the second one transmission is done by a separate Dispatcher thread, which is pinned to one of the 6 hyperthreads used for the clients. In both cases, for less than 4 clients the throughput of PSPAT is even higher than the baseline, because part of the work (scheduling and transmission) is now performed in parallel by the Arbiter or the Dispatcher thread, instead of being performed by the clients. Note that offloading work to other entities such as interrupt handlers and worker threads is common practice in OSes. For more clients, the throughput reaches a peak and then slightly degrades to the saturation point. While peak and saturation numbers are different when a Dispatcher is used (4.7 vs. 3.8 Mpps for the peak, 4 vs. 3.2 Mpps for the saturation), the behavior of the curve is very similar. Our profiling analysis shows that the sensible throughput degradation is to be attributed mainly to the implementation of the Scheduling Algorithm: we observed that the Arbiter thread spends an increasing portion of its time in the SA enqueue/dequeue routines as the number of clients increases. To a lesser extent, increasing the number of clients implies more work for the Arbiter, which must handle more queues.

To exercise the system at higher input loads we use `pkt-gen` as clients: these sources generate much higher packet rates, up to 16 Mpps with 8 clients. Fig. 8 shows how with no scheduler the system achieves perfect scaling, with each client transmitting on a separate NIC queue. The alternating slope is not visible here because a single pkt-gen client does not utilize the 100% of its hyperthread; the two hyperthreads on the same core can therefore alternate efficiently in using the shared CPU resources, without measurable interference. The curves for TC and PSPAT in Fig. 8 show a behavior that is very similar to the one of Fig. 7, with both packet schedulers saturating already with two clients, as pkt-gen clients are more greedy than UDP ones. Throughput for TC starts to drop earlier (with 3 clients), slowly declining down to 1.5Mpps as more clients are added to fill all the available NIC queues and hyperthreads. Up to 6 clients, PSPAT throughput slightly declines because of increased inefficiency of the Scheduling Algorithm, as explained earlier. If pkt-gen clients are placed on the same physical core (with 7 clients) and the same hyperthread of the Arbiter thread (with 8 clients), throughput drops significantly because the clients are aggressively stealing CPU

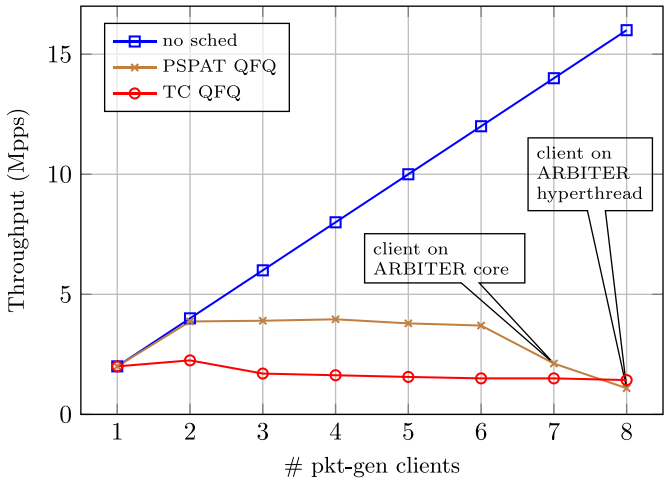


Fig. 8. Throughput in pps on I7 using netmap pkt-gen senders. For PSPAT experiments, the in-kernel version is used. A wrong configuration of PSPAT (i.e., not reserving a physical core for the Arbiter thread) causes a significant throughput drop in the worst case, as can be seen in the PSPAT curve for 7 and 8 clients.

cycles (or the CPU internal pipeline) to the Arbiter while the Arbiter is the bottleneck. Despite being a worst-case, this is not the right way to deploy PSPAT, which requires a physical core to be dedicated to the Arbiter thread, but we present this result here for completeness. Note that in all the correct configurations (up to 6 clients) PSPAT is still twice as performant as TC.

Finally, Fig. 9 shows the results of our throughput experiments on I7 with the userspace PSPAT configured with DRR. The three curves on the bottom correspond to the same experiments reported in Fig. 7, except for the experiment with the Dispatcher, which does not apply here because the transmission is performed by the client themselves. For the baseline and TC experiments, the only differences with the previous experiments are that (i) the UDP clients are implemented as threads in the same process that (optionally) runs the userspace Arbiter thread; and (ii) DRR is used in place of QFQ. This explains the minor performance differences between the two curves across Figs. 7 and9, which are otherwise the result of the same experiment. The alternating slope is visible also here for both the baseline and PSPAT curves. The most notable difference between kernel and userspace PSPAT is that a sensible UDP throughput degradation is not visible with the userspace version, which saturates almost perfectly at 4 Mpps. This confirms that

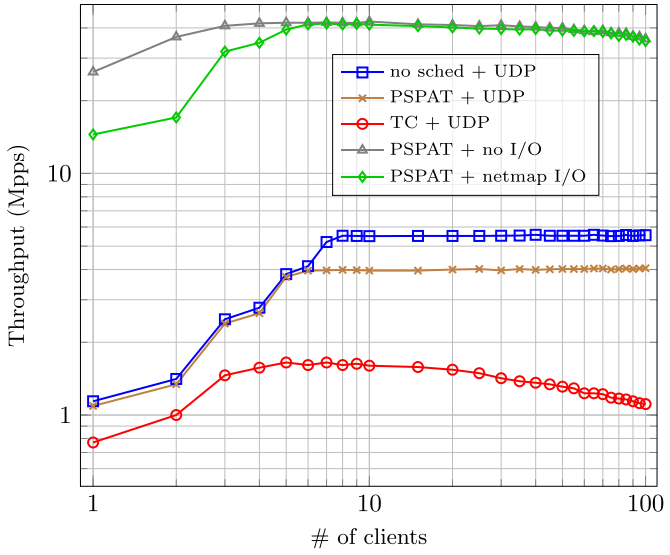


Fig. 9. Throughput for userspace PSPAT on I7.

an optimized implementation of the PSPAT architecture (together with a scalable implementation of the Scheduling Algorithms) is able to behave ideally under high load and/or with many clients, and that the throughput drop of the kernel PSPAT is not a limitation of our architecture.

As the userspace implementation is sufficiently fast, we perform two additional tests to push the PSPAT architecture to its limits. The corresponding curves are reported at the top of Fig. 9. In the first experiment, described by the lower curve, each client transmits the packets to a port of a separate netmap VALE [23] switch instead of sending to an UDP socket. This makes the I/O very fast for two reasons: first, the cost of transmitting a single packet is lower because the OS network stack is completely bypassed; second, and more importantly, the Arbiter releases scheduled packets in batch, so that the client can send the whole batch with a single system call. Note that we use VALE ports because they are faster than our NICs in terms of pps. The throughput curve shows a behavior that is very similar to the UDP + PSPAT curve but is more than 10 times higher for any number of clients, peaking at about 40 Mpps with 5 clients or more. In this situation the Arbiter is the bottleneck, and a minor performance degradation is measurable as the number of clients grows (with 100 clients the throughput is 15% less than the peak). Once again, this degradation is due to the higher number of queues that the Arbiter thread needs to scan, and it is only visible because of the extreme packet rates enabled by the netmap fast I/O. Replacing the fast I/O with a no-op does not lead to interesting differences. The throughput is essentially the same for 5 clients or more, while for less than 5 clients the alternating slope is replaced by a more linear trend, since the two clients on the same hyperthread have less work to do and therefore interfere less with each other. The same behavior in saturation just confirms that the Arbiter thread is the bottleneck for both experiments.

6.4.1. Scalability with dual-socket machines

On XEON40 we can run the throughput tests only with UDP clients sending to the loopback interface (see Section 6.2), and userspace PSPAT. These experiments are interesting mainly because XEON40 is a dual-socket machine, and we can see how the NUMA memory system affects PSPAT performance. Results are reported in Fig. 10, using a logarithmic scale for both *X* and *Y* axes. The bottom curves show that the network stack scales reasonably well in absence of a scheduler, with the usual alternating slope resulting from the client allocation strategy. However, when TC is configured with a Scheduling Algorithm (DRR),

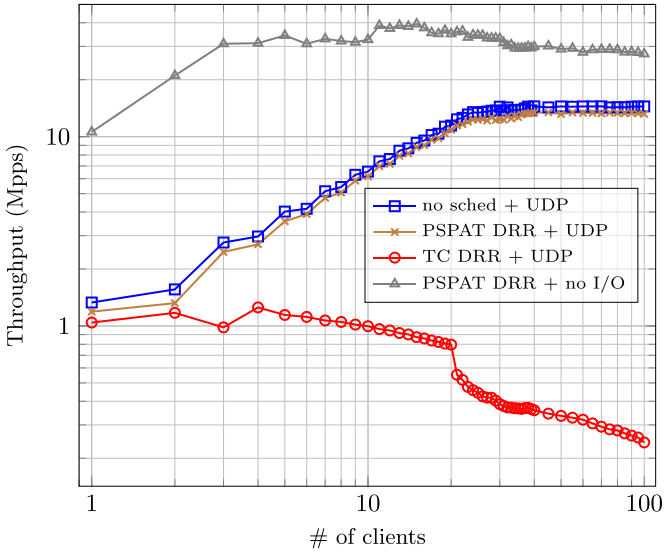


Fig. 10. Throughput on XEON40 using UDP sockets and different configurations. The userspace PSPAT has almost no loss of performance with respect to the no scheduler case, whereas TC scales very poorly.

throughput saturates early on and severely as clients are added, down to just 0.25 Mpps with 100 clients (less than 20% of the maximum value). Note that the throughput drop due to the congestion of the TC subsystem becomes more evident for more than 20 clients, when new clients are placed to cores on the second socket. This dramatically increases the cost of locking as contenders are placed across two different sockets.

In contrast, the userspace PSPAT only loses a small amount of capacity even with a large number of clients. No sensible throughput drop happens because of (greedy) clients being placed across different sockets. Similarly to I7, the total throughput is limited by the network stack, not by the Arbiter. By replacing transmission with a no-op, we can isolate the interaction of the clients with the Arbiter. As shown in the top curve, the Arbiter can make between 30 and 40 M decisions per second. The bump at 11 appears to be caused by an operating regime with a better usage of the internal memory busses in the Xeon socket where the clients are running. This suggests that the throughput in the top curve is limited by the clients rather than by the Arbiter.

6.5. Stability of rate allocation

It is expected that a correctly configured Packet Scheduler, driven below its maximum speed, can guarantee rates as configured and according to the properties of the Scheduling Algorithm. When the Packet Scheduler cannot sustain the link's speed in terms of packet rate, however, no queues build up and the Scheduling Algorithm has no opportunity to make decisions. This situation can lead to incorrect rate allocation.

An experiment with TC on I7 shows exactly this behavior. Once TC is configured with a given link capacity and two flows with weight 10 and 1, we start sending 60-bytes packets at maximum speed from both flows. With a link capacity up to 590 Mbit/s (corresponding to approximately 1.2 Mpps) TC can keep up and rate allocation is nominal. Just 10% above that, no queueing occurs in the SA (because TC cannot keep up with the packet rate), and the bandwidth is shared only depending on how requests interleave; in our experiment, the first flow gets 40% of the effective link bandwidth, while the second flow gets the remaining 60%.

PSPAT avoids this problem by design: in each round of `do_scan()`, it first collects outstanding requests before making scheduling decisions, thus allowing queues to build up and creating input for the decision, independently on the input load.

6.6. Measurement of latency distributions

Our final evaluation looks at latency distributions. The actual latency introduced by the Packet Scheduler depends in part on the Scheduling Algorithm and its configuration, and in part on other causes, such as process interaction, buses, internal pipelines and notifications. As stated at the beginning of Section 6, we want to verify that latency remains stable even when the system is overloaded.

For these experiments we set the first client (the "VICTIM") to send at an arbitrary low rate (4 Kpps), thus using a small fraction of the link's bandwidth, but with a weight 50 times higher than all other "interfering" clients (which generate traffic as fast as possible). Using a packet size of 1500 bytes also helps to trigger early any potential CPU, memory or bus bottlenecks. Packets from the VICTIM client are marked with a TSC timestamp when they are submitted to the packet scheduler; the TSC is read again on the receiver (on the same system) to compute the one way latency. The receiving NIC uses netmap to sustain the incoming traffic rate with no losses. The VICTIM client and the receiver are pinned to the two hyperthreads of the same physical core; the interfering clients run the remaining two (for PSPAT tests) or three physical cores (for TC tests) to avoid interferences due to CPU scheduling.

Table 3
Latency (μs) introduced by the scheduler in various configurations, when sending 1500 byte frames on a 40 Gbit/s NIC. The huge latency with HW scheduler is not a mistake: the PCIe bus is saturated and cannot cope with the load. All tests are run on I7 using the in-kernel PSPAT implementation.

Clients	Configuration	Min	Percentile				
			10	50	90	98	99
1	HW	5.7	5.8	6.0	6.1	6.4	6.4
1	TC	5.5	5.7	5.9	6.1	6.3	6.6
1	PSPAT	6.3	6.8	7.2	7.7	8.0	8.2
5	HW (PCIe congestion)	9.8	117.0	125.0	137.0	147.0	152.0
5	TC @ 10G 0.812 Mpps	6.6	8.5	12.6	16.6	18.0	18.6
5	PSPAT @ 10G 0.823 Mpps	6.4	7.3	9.0	11.1	11.9	12.2

Table 3 reports the one way latency on I7 and a 40 Gbit/s port in a number of configurations that we comment here.

The first three rows, with the VICTIM client alone, show the one way latency in the uncongested situation (baseline): latency is small and its distribution is mostly flat in all three cases. PSPAT adds an extra 1–2 μs because of the rate-limited reads and the time it takes to scan all queues.

The next row shows what happens when there is congestion on the PCIe bus. Here 5 clients drive the NIC's queues at full speed, but the bus can only sustain (as determined from other measurements) a little less than 35 Gbit/s, so less than line rate. The resulting congestion on the bus prevents the VICTIM client from being served as it should, and the latency jumps well over 100 μs, much beyond even the analytical bound for the T-WFI. The bound is not wrong: the bus bottleneck is just not modeled as part of the T-WFI.

The last two rows with DRR, 5 clients and link set at 10 Gbit/s show the effect of a slow scheduler. The in-kernel PSPAT can sustain the nominal packet rate (0.823 Mpps, since preambles and framing are not accounted for), whereas TC is saturated, runs a little below the line rate, and causes a slightly higher latency as the service order depends on how clients win access to the scheduler lock.

6.6.1. Latency on loopback

We conclude the latency analysis comparing TC and the userspace PSPAT on a loopback interface, using up to 80 UDP clients and DRR configured with variable link rate. The test configuration is similar to the one described in Section 6.6; however, here we use minimum sized packets to reach higher packet rates, so that we can achieve maximum interference and stress congestion on the scheduler.

The curves in Fig. 11 show some of our measurements, describing how latency varies with increasing link rate. Theory says that latency for DRR is proportional to the number of flows and inversely proportional to the link bandwidth. This behavior is shown by all the curves in Fig. 11 for link rates below the saturation threshold. On saturation, the Packet Scheduler gets overloaded, acting as a bottleneck for the packet rate. Note that link rate is reported in terms of pps (for 60-bytes packets) rather than bps, in order to show the saturation threshold in terms of packet rate. This is important because the cost for the Packet Scheduler to process a packet does not depend on the packet size.

On XEON40, the behavior of TC and PSPAT is similar until about 150 Kpps. Beyond this value TC saturates because the large number of clients starts to congest the scheduler lock, which becomes very expensive to acquire (see Table 1), specially on a dual-socket machine. In this situation the VICTIM client competes for the lock with the interfering ones, irrespective of its higher priority, resulting in higher latencies. In contrast, PSPAT saturates only at 20 Mpps, which is 2 orders of magnitudes higher. Because of PSPAT architecture, clients don't compete with each other, but talk with the scheduler by means of

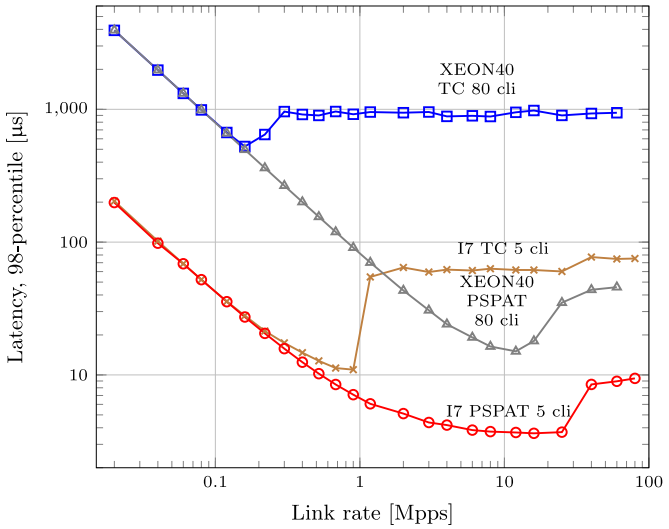


Fig. 11. 98-percentile of the latency for TC and userspace PSPAT at different link rates, operating on the loopback interface. Saturation is indicated by a sudden increase in the latency. Note the logarithmic scales on both axes.

dedicated queues; this gives better isolation among clients, reducing the latency for "well-behaved" users.

The curves labeled with I7 TC and I7 PSPAT show the result of the same experiment on I7, with only 5 clients. Both curves show lower latency because there are less flows, in accordance to the DRR properties. The saturation threshold is higher for both TC (1 Mpps) and PSPAT (35 Mpps) because of the faster CPU and because I7 is a single socket machine, which means cheaper memory interactions (see Table 2).

7. Related work

Scheduling Algorithms have been extensively studied in the 90s for their theoretical properties [6,10,11,26,27], and later for efficient implementations [7,12,13,28–32]. Software packet schedulers such as TC [1], ALTQ [3] and dummynet [2] are available in most commodity operating systems.

The performance of host-only schedulers has not received much attention. Some data is reported in [2,7], but otherwise the majority of experimental analysis uses bulk TCP traffic (often with large segments and hardware supported TSO) or ping-pong tests, and in both cases packet rates are not too high. Part of the reason is also that, until recently [14,15] network stacks were incapable to handle high packet rates.

Recent years have seen an increasing recourse to various heuristic solutions as in Hedera [33], partly motivated by more ambitious goals (such as, scheduling resources across an entire rack or data center, as in Fastpass [9]), and partly because of the presumed low performance of existing software solution (which, as we demonstrated, were erroneously blaming Scheduling Algorithms rather than heavyweight network stacks). Also, the increasing importance of distributed computation and the impact of latency and tail latency on many such tasks has shifted the interest from maximum utilization to latency reduction.

As part of this trend, numerous recent proposals started using rate limiters, such as EyeQ [34], or "Less is more" [35]. Senic [36] shows how large numbers of rate limiters can be implemented in hardware. By (re)configuring rate limiters (more on this later) one can keep traffic rates under control thus achieving some of the advantages of scheduling without the complexity of the algorithms. Running links below nominal capacity is also a common strategy to reduce congestion hence latency, and is used in [34,35,37] among others.

Schedulingnetwork resources for an entire cluster or datacenter is a challenging problem that has often been addressed by monitoring traffic on individual nodes, and exchanging feedback between the various node to, eventually, reconfigure rate limiters at the sources. Unavoidably, such solutions act on coarse timescales (a few milliseconds at best) and lack any theoretical analysis of performance bounds. As an example in this category, EyeQ [34] proposes an architecture where rate meters at the destinations periodically communicate suitable rates for the sources, tracking active sources and their weights. The information is used to program per-destination pacers on the sources, thus reducing the load for the scheduler(s). The control loop (at the receiver) compares the receive rate with allocations, and adjusts them every 200 μs, with a feedback that according to the authors converges in approximately 30 iterations. From these numbers and graphs in the paper, we can infer that EyeQ has a response time of several milliseconds, adds a round trip latency of over 300 μs, and does not support rates higher than 1 Mpps. Another example in this category, Silo [37], uses network calculus to derive formulas for the admission of new clients, then uses padding frames to implement fine grained traffic shaping in a standard NIC.

Another approach to cluster-level scheduling is Fastpass [9], which has some high level similarity with PSPAT. In Fastpass, requests for packet transmissions are first passed to a global, external scheduler that replies with the exact time at which the packet should be transmitted. Fastpass addresses a significantly harder problem than ours, namely, to reduce queueing on the entire network in a datacenter, as opposed to a single link. As a result, it must use a centralized scheduler for an entire group of machines, which knows the topology, capacity and state of the network, as well as the weights/reserved bandwidth for the various flows. Due to the computational complexity of the problem, the scheduler in Fastpass must use heuristics that are more expensive than PSPAT, cannot give strict service guarantees,[5] and is several times more expensive than ours.

8. Conclusions

We have presented PSPAT, a scalable, high performance packet scheduler that decouples clients, Scheduling Algorithm and transmissions using lock free mailboxes. This maximises parallelism in the system, and permits good scalability and very high throughput without penalising latency.

We have implemented PSPAT and evaluated its performance on single and dual socket systems and in a variety of load configurations. An in-kernel implementations runs more than 2 times faster than TC, without slowing down with increased concurrency. We have much room for improving this version, from instantiating more Dispatcher threads to improving the Scheduling Algorithm implementation (we currently hook into the implementations supplied by TC).

An optimized userspace PSPAT implementation can handle over 28 million scheduling decisions per second without overloading, even with 100 concurrent clients on a dual socket machine, and even faster on a single-socket system. The maximum scheduling rate is almost 40 Mpps, and latency remains stable even under heavy load and a large number of clients.

Considering the high packet rates it supports, PSPAT is a good candidate for use in firewalls or software routers managing access to high speed links. Given the high scalability, it also very well suited for use on cloud hosting platforms, where resource allocation for potentially non cooperating clients is a necessity, and certain clients, e.g. instances of Virtual Network Functions, may generate traffic with extremely high packet rates.

[5] As clearly indicated by the authors, the bound given in the paper [9] only applies if link utilization is less than 50%.

Acknowledgments

This paper has received funding from the European Union's Horizon 2020 research and innovation programme 2014–2018 under grant agreement No. 644866. This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

Appendix A. Deriving the cost of memory stalls

The cost of memory stalls is heavily dependent on the CPU architecture (x86 in our case), memory coherency protocol and CPU model. The key observation to estimate the duration of read stalls for a system like PSPAT is that a memory load operation can complete very quickly if the variable is already in a valid cache line, or incur in a larger latency (stall) when the cache line has been invalidated by a write operation done by another CPU. Similarly, a write operation can complete very quickly if the variable is in a valid cache line owned *exclusively* by the writing CPU. Or it can stall if the CPU write buffer is full, and another CPU has recently read the same cache line and turned the cache line in *shared* mode; in the latter case the write stalls until the update is propagated to the other CPU. Due to x86 strong memory ordering guarantees, writing alternatively to just two cache lines shared between R and W, at high rates, is enough to keep the CPU write buffer full.

Let us T_{RF} and T_{WF} be the average cost of read and write operations, respectively, when they are "fast". Similarly, T_{RS} and T_{WS} are the average cost of read and write operations when they are slow, that is when they stall. To compute T_{RF} and T_{RS} we let R run at maximum speed (i.e. continuously reading from V_i), and let W write at a given rate, using a different rate in each run. For each run we count the total number of reads N_R, the number of different values seen N_{RS} and the total duration D_R of the test as seen by R. Since for each different value seen we pay a slow read we can write

$$T_{RS}N_{RS} + T_{RF}(N_R - N_{RS}) = D_R \tag{A.1}$$

Since we have two unknowns (T_{RF} and T_{RS}), two runs would be enough to compute them. In practice we can collect more points and use interpolation. To obtain a well-conditioned set of equations we can use the same duration for all the runs (so that all the D_R will have similar values), and double the writer rate at each run. Note that if the writer rate is zero, N_{RS} is zero and we can immediately compute T_{RF}.

We can compute T_{WF} and T_{WS} in a similar way. We let W run at maximum speed (i.e. continuously writing to V_i), and let R read at a given rate in each run. We still count the number of different values seen by R, but we also count the total number of writes done by W (N_W) and the total duration D_W as seen by W. Since for each different value seen by R we pay a slow write, we have

$$T_{WS}N_{RS} + T_{WF}(N_W - N_{RS}) = D_W \tag{A.2}$$

and compute T_{WF} and T_{WS} by interpolation, using different reader rates and the same duration across multiple runs.

Appendix B. Proof from Section 5

We report here the proof of the following property, used in Section 5: the number of Δ_D intervals to drain packets from the excess block will be proportional to $k = \Delta_A B/L$, or the number of maximum sized packets in the block.

The excess block is drained at the minimum possible rate if clients succeed in keeping the maximum possible number of queues, say Q, constantly non-empty from when the round (that leaves the excess block unfinished) terminates. In fact, in this case the packets in the excess block are served at the minimum possible rate, B/Q, by the round-robin scheduler in the NIC. Then it will take about $\frac{Q}{B}B\Delta_A = Q\Delta_A$ time units to consume the block. As for Q, the number of packets for different queues that clients can dispatch in their $B \cdot \Delta_D$ budget, and thus the maximum number of different queues that clients can keep full, is proportional to the size $B \cdot \Delta_D$ of the budget, and inversely proportional to the size L of the packets. The total time to consume the excess block is then proportional to $\frac{B\Delta_D}{L}\Delta_A = k\Delta_D$.

References

[1] W. Almesberger, Linux network traffic control – implementation overview, Proceedings of the Fifth Annual Linux Expo, No. LCA-CONF-1999-012, (1999), pp. 153–164.
[2] M. Carbone, L. Rizzo, Dummynet revisited, ACM SIGCOMM Comput. Commun. Rev. 40 (2) (2010) 12–20.
[3] K. Cho, Managing traffic with ALTQ, Proceedings of the USENIX Annual Technical Conference, FREENIX Track, (1999), pp. 121–128.
[4] G. Lettieri, V. Maffione, L. Rizzo, PSPAT source code, 2018. https://doi.org/10.5281/zenodo.1163365.
[5] M. Shreedhar, G. Varghese, Efficient fair queuing using deficit round-robin, IEEE/ACM Trans. Network. 4 (3) (1996) 375–385.
[6] J.C. Bennett, H. Zhang, WF^{2}Q: worst-case fair weighted fair queueing, Proceedings of the 1996 INFOCOM'96, 1 IEEE, 1996, pp. 120–128.
[7] F. Checconi, L. Rizzo, P. Valente, QFQ: efficient packet scheduling with tight guarantees, IEEE/ACM Trans. Network. 21 (3) (2013) 802–816, http://dx.doi.org/10.1109/TNET.2012.2215881.
[8] C. Kulatunga, N. Kuhn, G. Fairhurst, D. Ros, Tackling bufferbloat in capacity-limited networks, Proceedings of the 2015 European Conference on Networks and Communications (EuCNC), (2015), pp. 381–385. 10.1109/EuCNC.2015.7194103
[9] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, H. Fugal, Fastpass: a centralized zero-queue datacenter network, Proceedings of the 2014 ACM SIGCOMM 2014, Chicago, IL, 2014.
[10] A.K. Parekh, R.G. Gallager, A generalized processor sharing approach to flow control in integrated services networks: the single-node case, IEEE/ACM Trans. Network. 1 (3) (1993) 344–357.
[11] J.C.R. Bennet, H. Zhang, Hierarchical packet fair queueing algorithms, IEEE/ACM Trans. Network. 5 (5) (1997) 675–689.
[12] P. Valente, Exact GPS simulation and optimal fair scheduling with logarithmic complexity, IEEE/ACM Trans. Network. 15 (6) (2007) 1454–1466, http://dx.doi.org/10.1109/TNET.2007.897967.
[13] P. Valente, Reducing the execution time of fair-queueing packet schedulers, Comput. Commun. 47 (2014) 16–33, http://dx.doi.org/10.1016/j.comcom.2014.04.009. http://www.sciencedirect.com/science/article/pii/S0140366414001455
[14] L. Rizzo, Netmap: a novel framework for fast packet I/O, Proceedings of the 2012 USENIX ATC'12, USENIX Association, Boston, MA, 2012.
[15] Intel, Intel Data Plane Development Kit, 2012. http://edc.intel.com/Link.aspx?id=5378.
[16] J. Corbet, Bulk Network Packet Transmission, 2014. https://lwn.net/Articles/615238/.
[17] G. Lettieri, V. Maffione, L. Rizzo, A study of I/O performance of virtual machines, Comput. J. (2017) 1–24, http://dx.doi.org/10.1093/comjnl/bxx092.
[18] L. Rizzo, S. Garzarella, G. Lettieri, V. Maffione, A study of speed mismatches between communicating virtual machines, Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems, ANCS '16, ACM, New York, NY, USA, 2016, pp. 61–67. 10.1145/2881025.2881037
[19] L. Rizzo, G. Lettieri, V. Maffione, Speeding up packet I/O in virtual machines, Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 47–58. http://dl.acm.org/citation.cfm?id=2537857.2537864
[20] L. Lamport, Specifying concurrent program modules, ACM Trans. Program. Lang. Syst. 5 (2) (1983) 190–222, http://dx.doi.org/10.1145/69624.357207. http://doi.acm.org/10.1145/69624.357207
[21] J. Giacomoni, T. Moseley, M. Vachharajani, Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. in: Proceedings of the Thirteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08).
[22] L. Rizzo, P. Valente, On service guarantees of fair-queueing schedulers in real systems, Comput. Commun. 67 (2015) 34–44.
[23] L. Rizzo, G. Lettieri, VALE, a switched ethernet for virtual machines, Proceedings of

the Eighth International Conference on Emerging Networking Experiments and Technologies, CoNEXT'12, ACM, Nice, France, 2012, pp. 61–72. 10.1145/2413176.2413185
[24] D. Turull, P. Sjödin, R. Olsson, Pktgen: measuring performance on high speed networks, Comput. Commun. 82 (2016) 39–48.
[25] R. Olsson, Pktgen the linux packet generator, Proceedings of the 2005 Linux Symposium, 2 Ottawa, Canada, 2005, pp. 11–24.
[26] D. Stiliadis, A. Varma, A general methodology for designing efficient traffic scheduling and shaping algorithms, Proceedings of the Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution (INFOCOM '97), (1997), pp. 326–335. 10.1109/INFCOM.1997.635151
[27] S.J. Golestani, A self-clocked fair queueing scheme for broadband applications, Proceedings of the Thirteenth IEEE Networking for Global Communications (INFOCOM '94), (1994), pp. 636–646.
[28] M. Karsten, Approximation of generalized processor sharing with stratified interleaved timer wheels, IEEE/ACM Trans. Network. 18 (3) (2010) 708–721, http://dx.doi.org/10.1109/TNET.2009.2033059.
[29] X. Yuan, Z. Duan, Fair round-robin: a low complexity packet scheduler with proportional and worst-case fairness, IEEE Trans. Comput. 58 (3) (2009) 365–379.
[30] C. Guo, SRR: An O(1) time complexity packet scheduler for flows in multi-service packet networks, IEEE/ACM Trans. Network. 12 (6) (2004) 1144–1155.
[31] L. Lenzini, E. Mingozzi, G. Stea, Tradeoffs between low complexity, low latency, and fairness with deficit round-robin schedulers, IEEE/ACM Trans. Network. 12 (4) (2004) 681–693, http://dx.doi.org/10.1109/TNET.2004.833131.
[32] S. Ramabhadran, J. Pasquale, The stratified round robin scheduler: design, analysis and implementation, IEEE/ACM Trans. Network. 14(6). 10.1109/TNET.2006.886287.
[33] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat, Hedera: dynamic flow scheduling for data center networks. Proceedings of the Tenth USENIX Conference on Networked Systems Design and Implementation (NSDI'10), USENIX Association, 2010.
[34] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, C. Kim, EyeQ: Practical Network Performance Isolation at the Edge, Proceedings of the Seventh USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), USENIX Association, Lombard, IL, 2013, pp. 297–311. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/jeyakumar
[35] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, M. Yasuda, Less is more: trading a little bandwidth for ultra-low latency in the data center, Proceedings of the 2012 USENIX Symposium on Networked Systems Design and Implementation (NSDI'12), USENIX Association, San Jose, CA, 2012, pp. 253–266. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/alizadeh
[36] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, A. Vahdat, SENIC: Scalable NIC for End-host Rate Limiting, Proceedings of the 2014 USENIX Symposium on Networked Systems Design and Implementation (NSDI'14), USENIX Association, 2014, pp. 475–488. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/senic-scalable-nic-end-host-rate-limiting-0
[37] K. Jang, J. Sherry, H. Ballani, T. Moncaster, Silo: predictable message latency in the cloud, Proceedings of the ACM SIGCOMM 2015, ACM, London, UK, 2015, pp. 435–448. 10.1145/2829988.2787479