PSPAT: software packet scheduling at hardware speed

Luigi Rizzo[1], Paolo Valente[2], Giuseppe Lettieri[1], Vincenzo Maffione[2]
[1]Univ. di Pisa, [2]Univ.di Modena e Reggio Emilia
rizzo.unipi@gmail.com. Work supported by H2020 project SSICLOPS.

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 scheduling mechanisms that support these rates and, at the same time, provide isolation and dependable service guarantees.

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, increasing performance and providing opportunities to parallelise operation.

We describe the operation of our system, discussion implementation and system issues, provide analytical bounds on the service guarantees of PSPAT, and validate the behaviour of its Linux implementation even at high link utilization comparing it with current hardware and software solutions. Our prototype can make over 15 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 CPU sockets with tens of CPU cores, and Network Interfaces (NICs) with an aggregate rate of 10..100 Gbit/s. Even at such data rates, handling bulk TCP traffic is doable: large frames (from 1500 up to 64 Kbyte segments with hardware segmentation offloading) imply relatively modest packet rates. On the contrary, Virtual Network Function (VNF) instances are challenging, as they often operate with very small packets and rates of 10+ Millions of packets per second (**pps**).

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 centralised 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 2.3, 5.5 and 5.6.

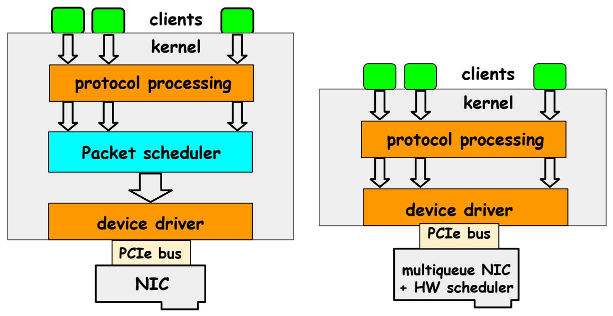

Figure 1: Common architectures for software and hardware packet schedulers in OSes.

STATE OF THE ART: OSes implement packet scheduling with one of the solutions shown in Figure 1. Software Packet Schedulers (left) are included in many OSes (TC [3] in Linux, dummynet [7] and ALTQ [9] in FreeBSD and other BSD OSes). Access to the Packet Scheduler, and subsequent transmissions, are completely serialized in these solutions, which can barely sustain 1..2 Mpps (see Section 5 and [7]).

Hardware packet scheduling (right) is possible on NICs with multiple transmit queues, offering 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, often with a capacity (bandwidth and transactions per second) barely matching that of the network link. As we show in Section 2.3, 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, shown in Figure 3 and called PSPAT. Two sets of MAILBOXes, implemented as lock free queues, 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 3.7 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 15 M *decisions per second* even under highly parallel workloads on a dual socket, 24 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 [27], WF^{2}Q+ [6], QFQ [8]) that provide isolation and provable, tight service guarantees with arbitrary workloads. These cannot be compared with: 1) queue management schemes such as FQ_CODEL [16], 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 [21], which, due to the complexity of the problem they address, cannot give guarantees at high link utilization. More details are given in Section 6.

TERMINOLOGY REMARKS. The term "scheduler" is often used ambiguously: sometimes it indicates a *Scheduling Algorithm* such as DRR [27] 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 2 gives some background on packet scheduling. The architecture and operation of PSPAT are presented in Section 3, which also discusses implementation details. Analytical bounds on service guarantees are computed in Section 4. Section 5 measures the performance of our prototypes and compares them with existing systems. Finally, Section 6 presents related work.

2 Motivation and background

Tenants of cloud platforms may have a legitimate need to generate traffic at rates of tens of millions of packets per second (pps): think of VNF instances implementing routers, firewalls, NATs, load balancers... . Handling high pps workloads in commodity operating systems is problematic even without any scheduling, and this has led to the development of OS bypass and network stack bypass techniques (see [24, 12]). Hardware packet schedulers can be fast, but come with their own set of problems (see Section 2.2.2), and are not necessarily on the path of the traffic. Software packet scheduling at high packet rates is a currently unsolved problem.

2.1 Scheduling Algorithms

A *Scheduling Algorithm* (**SA**) 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 giving priority to some flows over others; limiting the maximum rate of individual flows; 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 [20], 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)[5], 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.*

T-WFI matters as it measures the extra delay and jitter a packet may experience: we want it to have a small upper bound, 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 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 trade-offs. We have efficient *Weighted Fair Queueing* algorithms [5, 31] that match the T-WFI lower bound, with $O(\log N)$ cost per decision, where N in the number of flows. Fast variants of Weighted Fair Queueing, such as QFQ [8] and QFQ+[32] 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 [27], also known as Weighted Round Robin, WRR), have $O(1)$ time complexity but a poor $O(N)$ T-WFI.

Some numbers help appreciate the difference. With 1500 byte packets (1.2 μs at 10 Gbit/s), even for high weight flows, 25 busy flows on DRR cause at least 30 μs of latency (the worst case is much higher, see Section 4.2, also depending on the scheduling quantum used by DRR). In contrast, in a similar scenario, a good scheduler such as QFQ will normally give 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 offer 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.

2.2 Packet Schedulers

A Packet Scheduler (*PS*) is a set of three components: a set of *QUEUES* to store packets generated by CLIENTS and belonging to different flows; an *ARBITER* that selects the next packet to be transmitted using some Scheduling Algorithm (*SA*); a *DISPATCHER* that delivers the selected packet to the NIC (or the next stage in the network stack).

2.2.1 Software Packet Schedulers

Software Packet Schedulers are commonly implemented as a single unit in commodity OSes. CLIENTS (and interrupt handlers) contend to access the queues and the Scheduling Algorithm's data structures, and perform the functions of the ARBITER and DISPATCHER.

This architecture has two major performance issues. On the input side, there is serialization and high lock contention (with up to one competitor per core, so possibly tens of them) to pass packets to the SA. On the output side, the DISPATCHER is often a single thread even with multiqueue NICs, thus losing any potential parallelism. The resulting throughput barely reaches 1..2 Mpps (see Section 5 and [7]), but the blame goes on the design choices, not on the Scheduling Algorithm.

2.2.2 Hardware Packet Schedulers

Modern NICs support multiple independent transmit queues, and a limited choice of scheduling algorithms, so the entire Packet Scheduler can be moved to the NIC. This approach at first sight removes all bottlenecks of software Packet Schedulers: clients can operate in parallel on the different queues, and the NIC takes care of arbitration.

Nevertheless there are problems with this approach as well. One is a limitation in Scheduling Algorithms and number of queues supported by the hardware. The operating model is often "one queue per core", which does not support the (common) case where we would like to aggregate into one flow all traffic from a single tenant using

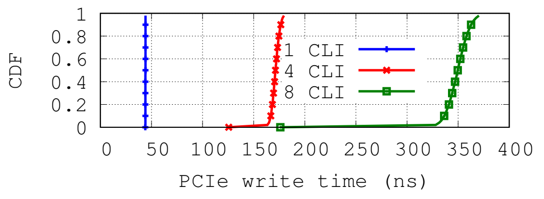

Figure 2: CDF of the duration of a PCIe write to an Intel X520 NIC, when the device reaches its capacity (approximately 20 Mreq/s). The CPU is stalled during this time.

multiple cores. A more serious problem is introduced by the presence of the PCIe bus, as described below.

2.3 PCIe limitations

A NIC connects to the system through a PCIe bus, whose capacity normally matches the link's bandwidth: typical dual port 10G and 40G cards use 8 PCIe lanes, with an aggregate bandwidth of 40..64 Gbit/s. NICs also support relatively low request rates for PCIe accesses from the CPU. The combination of these limits can severely impact performance. We give two examples below.

In our tests, an Intel XL710 dual-port 40 Gbit/s NIC, with 1500-byte frames, and both ports active, can sustain about 37.4 Gbit/s with one queue, down to 30.4 Gbit/s with 8 queues. Additional queues largely increase interrupts and bus traffic for a given data rate, and one should prefer the use of as few queues as possible. The latency test in Table 3 shows that even a few clients can cause bus congestion and introduce over 100 µs of delay while the link is well below "line rate".

Bus transaction rates are also problematic. PCIe writes from the CPU are absorbed by a write buffer that fills up rapidly if clients exceed the write rate supported by the device. An Intel X520 can sustain about 20 M PCIe writes per second, which the PCIe controller assigns to cores using round robin, irrespective of how the NIC's scheduler is programmed. Figure 2 shows how PCIe writes to a register of an Intel X520 NIC take a long time once the write buffer is full: a single client may stall for 50 ns, with 8 concurrent clients this time bumps to almost 400 ns.

As a result, greedy clients can saturate the PCIe resources and prevent other clients from even submitting traffic according to their fair share on the NIC's scheduler. Workarounds to this problem require additional buffer threads to perform write coalescing, causing additional latency and use of resources, and preventing a purely hardware solution.

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 Figure 3, 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, processes, threads;

C **cores** on a shared memory system;

C **Client Lists** CL_c, indicating clients recently active on each core;

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, described in more detail in Section 3.6. There are no constraints on the number of clients (M), cores (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).

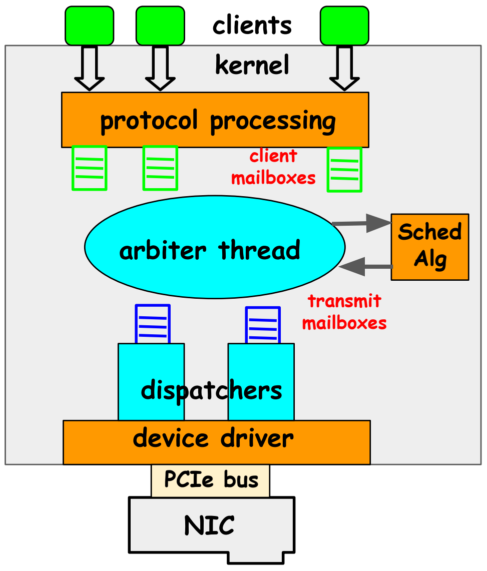

Figure 3: The architecture of PSPAT.

PSPAT operates as follows: 1) A client x, who wants to transmit and runs on core c, pushes packets into its client mailbox, CM_x, and then appends x to CL_c if not there already. 2) The ARBITER continuously runs function `do_scan()` in Figure 4, to grab new requests from the mailboxes of active clients on all cores, pass them to an SA, trim the Client Lists, and use a leaky bucket algorithm to release packets into the TX mailboxes. Finally, 3) dispatcher threads drain the TX mailboxes and invoke the device driver for the actual transmissions.

By reducing serialization to the bare minimum (only the ARBITER that runs the Scheduling Algorithm), and removing lock contention through the use of private, lock free mailboxes, PSPAT achieves very good scalability, and opens the door to performance enhancements through pipelining and batching.

Of course the devil is in the details, so the rest of this Section describes in depth the various components of PSPAT and discusses how they interact with the hardware.

3.1 Clients

Clients just push packets into their private mailbox to communicate with the ARBITER. 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 a new client has been scheduled on core *c and* that client has generated traffic since the previous 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 cores without creating data races (because the mailbox is private to the client) or correctness problems: even if, due to migration(s), a mailbox appears in multiple client lists, 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. Immediate feedback is not always possible, as local drops may be deferred because of the queueing policy (e.g. CODEL), or further schedulers or rate limiters. PSPAT offers immediate reporting when the client mailbox is full; depending on how flows are aggregated, there might be other reasons for drop that cannot be detected immediately.

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 core 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, see Section 2.3. 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 sending packets to the next stage is fast, as it is the case in frameworks like netmap [24] and DPDK [12], dispatching can be done directly by the ARBITER. Conversely, separate dispatchers can help improve throughput when packet transfers to the NIC or next stage is expensive.

3.4 The arbiter

The body of the ARBITER's code, function `do_scan()` in Figure 4, is structured in three blocks because of correctness and performance. We take a timestamp, `t_start`, before the first block that drains the mailboxes: this guarantees that any packets pushed in the SA have arrived before `t_start`, and this are valid candidates for transmission in the second block that implements the leaky bucket. Finally, the third block where we notify dispatchers gives the opportunity to exploit batching in the latter. Note that since packets are released to the TX mailboxes at or below link's rate, those mailboxes should never overflow unless the link goes down.

It is important that the ARBITER completes a round very quickly (to avoid adding too much latency) and does not spend a large fraction of its time stalled on memory reads (a real risk, as it has to scan $O(C)$ mailboxes updated by clients). C should be less than 100 even for a reasonably large system. The ARBITER accesses a small part of each mailbox, and does it frequently, so a mailbox 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 these values on a dual-socket E5-2640 system). It follows that 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.

```
int do_scan() {
  t_start = rdtsc();
  for (i=0; i < CORES; i++) {
    for (cli in CL[i]) {
      while ((pkt = extract(CM[cli])) != NULL) {
        SA.enqueue(pkt);
      }
      <trim CL[i] leaving last entry>
    }
  }
  while (link_idle < t_start) {
    pkt = SA.dequeue();
    if (pkt == NULL) {
      link_idle = t_start;
      return NO_TRAFFIC;
    }
    link_idle += pkt->len / bandwidth;
    i = pkt->tx_mbox_id;
    <enqueue pkt in TM[i]>
    i = pkt->client_id;
    <clear one entry in CM[i]>
  }
  for (i=0; i < TX_QUEUES; i++) {
    if (!empty(TM[i])) {
      <notify dispatcher[i]>
    }
  }
}
```

Figure 4: Simplified code for the ARBITER.

Section 3.6 describes how mailboxes can be implemented to reduce cache misses and amortise read stalls; we also rate limit accesses to each mailbox to once every 1-2 μs so that the ARBITER will never stall on memory reads for more than 10-20% of the time.

3.5 Avoiding busy wait

We would like to make the ARBITER sleep when it has no work to do. This happens i) while the current packet completes transmission, or ii) when all queues 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 queues 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 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 (20..50 µs), achieving performance in presence of traffic, and zero overhead when there is light or no traffic.

3.6 Mailboxes

Lock free queues often use Lamport's algorithm [17]: the producer (client C_x) updates the queue's current slot and insertion pointer, and the consumer (the ARBITER) reads both pieces of information. This scheme causes two cache misses and possibly requires memory barriers to ensure the correct ordering of reads.

PSPAT avoids using the producer's pointer (and the related barrier and read stall) through the technique presented in FastForward [10]: a special value in the slot (in our case, a NULL pointer) marks the insertion point, other values indicate that the slot is full and (implicitly) that the insertion pointer has advanced. Similarly, the consumer overwrites used slots with a NULL to pass its "release" pointer to the producer.

Having both producer and consumer write to the mailbox can generate write-write cache conflicts when a queue is nearly full or nearly empty, and both slots are in the same cache line. The former case is avoided by not using the full size of the queue. For the latter case, we make the consumer lag behind when clearing entries: when slot i is consumed, slot $i - \Delta$ is cleared, with Δ large enough to sit on a different cache line from slot i.

Read-write cache conflicts also occur when the producer updates multiple slots in the same cache line while it is accessed by the consumer. Also based on the measurements in Section 3.7, we address this problem by rate-limiting read accesses to the mailboxes.

3.7 Reader-writer speed and latency

PSPAT hits heavily on the memory system, so we need a clear picture of what happens, in terms of performance, with various concurrent access patterns and on system with multiple CPU sockets. We thus ran experiments on a dual socket Xeon E5-2640 system running at 2.5 GHz, with two threads, called W and R, running on different cores and accessing at configurable rate one or more shared variables V_i in different cache lines (few enough to fit in the L1 cache).

In a first experiment, W writes increasing 64-bit values to one or more shared variables, and R counts the number of reads and different values seen per unit of time. By varying the read and write rates, we can also derive the duration of a read stall (when the local cache is stale), and that of a write stall (when all write buffers are busy). On x86, due to its ordering guarantees, just two cache lines shared between R and W and written at high rates suffice to trigger write stalls.

In a second experiment, W and R run a request/response protocol, and V_i is incremented only when the R

	HT-HT	Core-Core	SKT-SKT
Read stall	10-15 ns	50 ns	130-220 ns
Write stall	–	15 ns	100 ns
Updates per second	75 M	20 M	5 M
Round trip latency	30 ns	130 ns	480 ns

Table 1: Cost of various memory operations on a dual socket Xeon E2-2640. See Section 3.7 for details.

confirms that has seen the previous update. In this case, we measure the number of round trip transactions per unit of time; its inverse is the latency incurred in passing information through a separate thread.

Table 1 reports the values measured, which depend on whether the two threads are on hyperthreads on the same core, cores on the same socket, or different sockets. The exact figures vary with different CPU types, but the important takeouts are the high cost difference between single and multi-socket platforms, awareness that even non atomic writes can stall, and that shared memory operations on multi socket systems (an important target platform for PSPAT) can be 4-5 times slower than on single socket systems.

To mitigate the interference between readers and writers, we rate limit read accesses to mailboxes: the slots near the insert and release points are cached on both sides, and refreshed at most once every Δ_A seconds on the ARBITER's side, and Δ_C seconds on the client's side. These two parameters are in the $1..5$ µs range.

4 T-WFI in PSPAT

As anticipated in Section 2, an important quality metric of a scheduler is the T-WFI (Definition 1). A larger T-WFI means increased jitter and delay in the communication, with obvious consequences. PSPAT does not define new Scheduling Algorithms, but uses one within the ARBITER, so the purpose of this Section is to determine the overall T-WFI of PSPAT given that of the underlying scheduling algorithm, T-WFI$_{\text{SA}}$

4.1 T-WFI analysis

The literature contains an evaluation of the T-WFI$_{\text{SA}}$ for several Scheduling Algorithms that we can use in PSPAT, see for example [26]. The analysis in [26] shows that the T-WFI of a complete Packet Scheduler is made of a first component, say T-WFI$_{\text{SA}}$, 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, as analysed in [26].)

In PSPAT, the second component depends on how we feed the Scheduler Algorithm and the NIC. We evaluate it here 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$_{\text{SA}}$, 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[1]. 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's first assume that DISPATCHERS operate with $\Delta_D = \Delta_A$ and are synchronised 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 synchronised, 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 $B\Delta_A$ 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). We omit the proof for brevity, but the number of Δ_D intervals to drain packets from the first block will be proportional to $k = \Delta_A B/L$, or the number of maximum sized packets in the block. 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

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

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

4.2 T-WFI examples

To put numbers into context: from [26] we know that

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

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

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, $L/B = 1.2\,\mu\text{s}$. Assuming weights ranging from 0.005 to 0.5, the client with the highest weight will have $\text{T-WFI}_{\text{QFQ}}^{(k)} = 12\,L/B = 14.4\,\mu\text{s}$ irrespective of N. For DRR, the dependency on N gives $\text{T-WFI}_{\text{DRR}}^{(k)} = 226\,L/B = 271.2\,\mu\text{s}$ 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.

5 Experimental results

We ran a number of experiments to evaluate the performance of PSPAT in terms of maximum throughput and latency distribution, and compare it with existing alternatives. The test scenarios have been chosen carefully and rigorously to emphasize the phenomena under investigation (cost of running the packet scheduler, scalability under load), reduce noise measurement (such as, effects of NIC's behaviour, or load on the receivers), run each solution in reasonable operating conditions, and make a fair comparison among the various alternatives.

5.1 Test environment

For our experiments we used two hardware platforms (called **I7** and **XEON2**, described below), two implementations of PSPAT, and several 10 G and 40 G NICs. Platform **I7** is a single-socket i7-3770K CPU at 3.5 GHz (4 cores, 8 threads), 1.33 GHz DDR3 memory, running Linux 4.7. We 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 8Gbit/s each). The NICs include a hardware DRR scheduler. Platform **XEON2** is a dual socket system with Xeon E5-2640 CPUs at 2.5 GHz (6 cores, 12 threads each), with 1.33 GHz DDR3 memory running Linux kernel 2.6.32-504. XEON2 does not have fast NICs so tests here are done on the loopback interface.

5.1.1 PSPAT versions

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. It is enabled with a `sysctl`, intercepts packets in `__dev_queue_xmit()` and after scheduling delivers them to the device through `dev_hard_start_xmit()`. The ARBITER uses the TC [3] subsystem from the Linux kernel as the Scheduling Algorithm. This gives a perfectly fair comparison with TC as we use exactly the same code and datapaths. On the other hand, the reuse of existing code brings in some unnecessary performance limitations, as discussed in Section 5.4.1.

The userspace version of PSPAT uses scheduler implementations taken from the dummynet [7] 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 [24] framework and the VALE [25] software switch. Since PSPAT already implements a rate limiter, it is not necessary to add another one (the HTB node) as in TC. Furthermore, we can use more efficient mailboxes, similar to those used by Hardware schedulers (see Section 5.4.1). Finally, clients also perform the role of DISPATCHERS, once the ARBITER has cleared packets for transmission.

Overall, the userspace PSPAT can be a lot more performant and helps explore performance improvements in Scheduling Algorithm implementation mailboxes, as well as support operation on platforms where we cannot operate in the kernel (as on our XEON2), or we do not want to (userspace I/O frameworks and protocol stacks).

5.2 Test configuration

For our tests, clients are traffic generators with configurable rate, packet and bursts size. They use either UDP sockets, or `pkt-gen`, a fast UDP source that uses the netmap API [24] but still goes through the standard Linux network stack, like the similarly named Linux kernel module [30, 19]. Usual care is taken to achieve reliable measurements (disable high C-states, lock CPU clock speed, pin threads to cores, properly configure interrupt moderation on the NICs, use busy wait instead of notifications in latency tests). Each single measurement is repeated 10 times, and arithmetic mean and standard deviation is computed over the 10 trials.

For our workload, running two clients on hyperthreads of the same physical core slows down both clients considerably (the pair delivers only 1.3 times the traffic of a single client). In the tests clients are allocated by first filling up cores, then CPU sockets, so that the offered load is monotonically increasing, although with different slope when going to even or odd number of clients. The ARBITER, if present, is allocated on a different core, and on XEON2 it is on the second socket, so we can measure the worst case cost of the interaction with clients.

Unless specified otherwise, in all experiments we have used 60 byte packets for speed measurements, and 1500 bytes ones for latency tests. 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; and PSPAT uses parameters $\Delta_A = 1000$ ns, while packet dispatch is implemented directly by the ARBITER.

On I7, traffic normally goes through a 40 Gbit/s NIC to a second port on the same machine. On XEON2, lacking suitable NICs, we run tests using the loopback interface (using different UDP ports to avoid contention on the destination socket).

5.3 Metrics

pps 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**, so its "decisions per second" and "pps" are the same.

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.

5.4 Throughput

Our first set of experiments measures the maximum throughput with multiple UDP senders in the following scenarios: i) no scheduler, ii) scheduled by TC, and iii) scheduled by PSPAT. The Scheduling Algorithm is QFQ (DRR is only marginally faster).

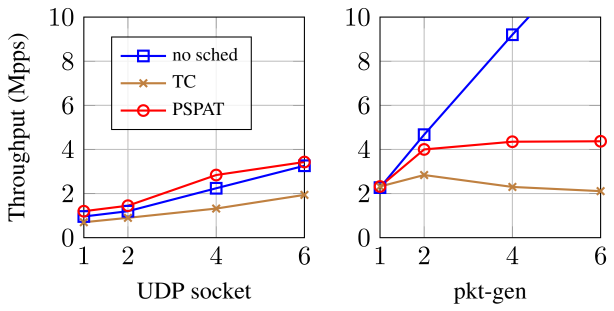

Figure 5: Throughput in pps, with different packet schedulers on I7.

Figure 5 shows the results on I7, where we use up to 6 clients (6 hyperthreads, 3 different cores) on a 40 Gbit/s NIC. On the left, clients use UDP sockets, peaking to 3.26 Mpps with 6 clients and no scheduler. TC causes a rate reduction between 25 and 40% even with just 6 clients. On the contrary, PSPAT actually slightly increases the throughput, because part of the clients's work is now done by the ARBITER (offloading work to other entities such as interrupt handlers and daemons is common practice in OSes). Note how there is no sign of saturation as the number of clients increases.

To exercise the system at higher loads we use `pkt-gen` as clients: these sources generate much higher packet rates, up to 13.5 Mpps with 6 clients. Figure 5, right shows how TC starts decreasing its rate as the number of clients increase, whereas PSPAT reaches twice the throughput and does not decline with up to 6 clients. Preliminary measurements suggest that the use of external dispatchers may raise the throughput to almost 7 Mpps.

5.4.1 Pushing the limit

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, Throughput (Mpps), PSPAT userspace DRR

	Clients			
Configuration	1	2	4	6
netmap pipes	15.00	17.00	36.00	35.90
scheduler only	25.00	37.00	36.00	35.90

Table 2: Throughput for userspace PSPAT on I7

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. As shown at the end of Table 2, on I7, and using netmap to send data (through a netmap pipe, as the NIC would not be fast enough for those data rates) PSPAT can schedule almost 40 Mpps (one packet per decision) with 6 clients and DRR. The bottleneck here is the ARBITER, as dispatching is done by the client themselves. Removing the transmission part, the peak rate remains the same, just a lower number of clients suffices to saturate the ARBITER.

5.4.2 Scalability

On XEON2 we can run the tests only with UDP sockets and on the loopback interface, but scale up to 20 clients. Results are in Figure 6 (note the logarithmic Y axis). The bottom curves, with UDP senders, show that the network stack scales reasonably well in absence of a scheduler, but with TC throughput saturates early on and severely as cores are added, down to just 0.32 Mpps with 20 clients, less than 10% of the maximum value. In contrast, PSPAT only loses a small amount of capacity even with a large number of clients.

The total throughput is limited by the performance of the network stack, not by the ARBITER. Replacing transmission with a no-op, we can measure the interaction of the clients with the ARBITER. As shown in the top curve, the ARBITER can make between 16 and 27 M decisions per second, without dropping as the number of clients increases. The large gap between 10 and 11 clients is because we start placing clients on the second socket, where also the ARBITER runs. Thus, clients above 10 have a faster path to the ARBITER and are able to achieve higher speeds (this is also the reason why on I7, which has a single socket, we see significantly higher speeds).

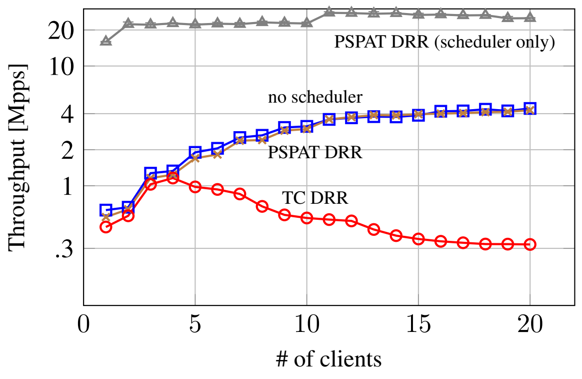

Figure 6: Throughput on XEON2 using UDP sockets and different configurations. PSPAT has almost no loss of performance with respect to the no scheduler case, whereas TC scales very poorly.

5.5 Rate allocation

We expect 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, no queues build up and the Scheduling Algorithm has no opportunity to make decisions, hence it may fail even on the rate allocation.

An experiment with TC showed exactly this behaviour. On I7 we configured two flows with weight 10 and 1, driving TC with short packets. With a link capacity up to 590 Mbit/s, approximately 1.2 Mpps, TC could keep up and rate allocation is nominal. Just 10% above that, no queueing occurs on the output because TC is too slow, and rates are allocated just at the speed requests come in (between 40 and 60% for each flow in our tests).

PSPAT avoids this problem because 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.

5.6 Latency distribution

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 (process interaction, buses, internal pipelines, notifications). We want to verify that in overload situations (on the CPU or other hardware) latency remains stable too.

Indeed, because latency is normally not as visible as throughput, this part of the experimentation was an interesting exercise in discovering unexpected bottlenecks in the system and poor parameter setting, and sometimes removing them.

For these experiments we set the first "TARGET" client to send at 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. We use a packet size of 1500 bytes to trigger early any potential CPU, memory or bus bottlenecks. Packets from the TARGET 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.

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

Baseline: The first three rows, with 1 client, show that in uncongested situation, one way latency is small distribution 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.

PCIe congestion: 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 TARGET 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. It is not that the bound is wrong, but the bottleneck is not the one modeled in the computation.

Slow scheduler: The last two rows with DRR, 5 clients and link set at 10 Gbit/s show the effect of a slow scheduler. PSPAT can to sustain the nominal packet rate (.823 Mpps, since preambles and framing are not accounted for), whereas TC is saturated, does not reach line rate, and causes a slightly higher latency as the service order depends on how clients win access to the scheduler.

5.6.1 Latency on loopback

We conclude the latency analysis comparing TC and the userspace PSPAT on a loopback interface. Transmission cost here is higher, and TC shows high latency when overloaded, as the client that holds the scheduler's lock tries desperately to drain incoming requests. We use DRR in these experiments.

Figure 7 shows some of our measurements. The curves labeled XEON TC and I7 TC show the latency vs link rate with TC. Theory says that latency for DRR is $\propto N/B$, a behaviour shown by all the curves in Figure 7 at low

Latency distributions in µs on I7

CLI	Notes	Percentile min	10	50	90	99
1	HW	5.7	5.8	6.0	6.1	6.4
1	TC	5.5	5.7	5.9	6.1	6.6
1	PSPAT	6.3	6.8	7.2	7.7	8.2
5	HW (PCIe congestion)	9.8	117.0	125.0	137.0	152.0
5	TC @ 10G .812 Mpps	6.6	8.5	12.6	16.6	18.6
5	PSPAT @ 10G .823 Mpps	6.4	7.3	9.0	11.1	12.2

Table 3: Latency 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.

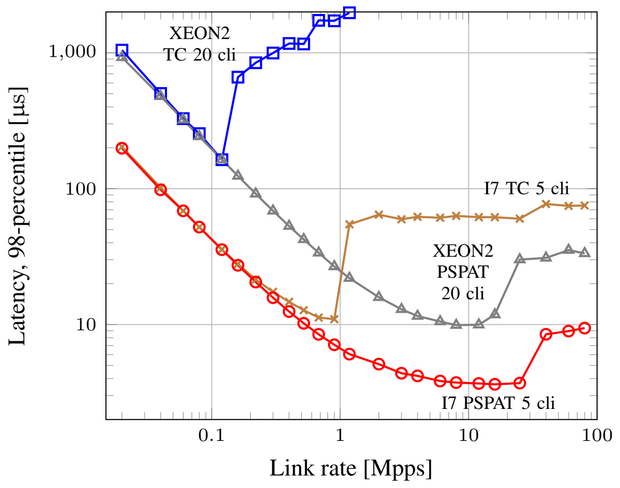

Figure 7: 98-percentile of the latency for TC and PSPAT at different link rates, operating on the loopback interface. Saturation is indicated by a sudden increase in the latency. Note the logarithmic scales on both axes.

link rates. On saturation, latency with TC goes up rapidly (much more on the older Linux version used in XEON2). The likely reason is that in TC one client acts as a dispatcher, and when the nominal rate exceeds the capacity of the scheduler, the dispatcher keeps trying to drain incoming requests until reaches a maximum loop count. We suspect the different settings in the two Linux versions are responsible for the radically different behaviour.

In contrast, in PSPAT dispatchers are separate so each is responsible for its own traffic, giving better isolation among clients thus also reducing the latency for "innocent" users.

6 Related work

Scheduling algorithms have been extensively studied in the 90's for their theoretical properties [20, 6, 5, 29, 28] and later for efficient implementations [31, 8, 32, 15, 33, 11, 18, 23]. Software packet schedulers such as TC [3], ALTQ [9] and dummynet [7] are available in most commodity operating systems.

The performance of host-only schedulers has not received much attention. Some data is reported in [7, 8], 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 [12, 24] network stacks were incapable to handle high packet rates.

Recent years have seen an increasing recourse to various heuristic solutions as in Hedera [1], partly motivated by more ambitious goals (such as, scheduling resources across an entire rack or data center, as in Fastpass [21]), 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 [14], or "Less is more" [2]. Senic [22] 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 [14, 2, 13] among others.

Scheduling network 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 [14] 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 [13], 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 [21], 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[2], and is several times more expensive than ours.

7 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 a variety of load configurations. An in-kernel version runs more than 2 times faster than TC, without slowing down with increased concurrency. We have much room for improving this version, from instantiating separate dispatcher nodes to improving the Scheduling Algorithm implementation (we currently hook into the implementations supplied by TC).

An optimised userspace version, even with 20 concurrent clients and a dual socket machine, prototype can handle over 15 million scheduling decisions per second without overloading, and twice that rate on a faster single core system. The maximum scheduling rate is almost 40 Mpps, and even under heavy overload latency remains stable.

[2]As clearly indicated by the authors, the bound given in the paper [21] only applies if link utilization is less than 50%

References

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI'10*. USENIX Association, 2010.

[2] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *NSDI'12*, pages 253–266, San Jose, CA, 2012. USENIX Association.

[3] W. Almesberger. Linux network traffic control–implementation overview. In *5th Annual Linux Expo*, number LCA-CONF-1999-012, pages 153–164, 1999.

[4] (Authors removed for double blind review). Mysched source code. *Please contact PC chairs to request the source code while the paper is under review*.

[5] J. C. R. Bennet and H. Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, 1997.

[6] J. C. Bennett and H. Zhang. Wf 2 q: worst-case fair weighted fair queueing. In *INFOCOM'96*, volume 1, pages 120–128. IEEE, 1996.

[7] M. Carbone and L. Rizzo. Dummynet revisited. *ACM SIGCOMM Computer Communication Review*, 40(2):12–20, 2010.

[8] F. Checconi, L. Rizzo, and P. Valente. QFQ: Efficient packet scheduling with tight guarantees. *IEEE/ACM Transactions on Networking*, 21(3):802–816, 2013.

[9] K. Cho. Managing traffic with altq. In *USENIX Annual Technical Conference, FREENIX Track*, pages 121–128, 1999.

[10] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism, a cache-optimized concurrent lock-free queue. *PPoPP'08*, 2008.

[11] C. Guo. SRR: An O(1) time complexity packet scheduler for flows in multi-service packet networks. *Proc. of ACM SIGCOMM 2001*, pages 211–222, August 2001.

[12] Intel. Intel data plane development kit. *http://edc.intel.com/Link.aspx?id=5378*, 2012.

[13] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable message latency in the cloud. In *ACM SIGCOMM 2015*, pages 435–448, London, UK, August 2015. ACM.

[14] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim. Eyeq: Practical network performance isolation at the edge. In *NSDI 13*, pages 297–311, Lombard, IL, 2013. USENIX Association.

[15] M. Karsten. Approximation of generalized processor sharing with stratified interleaved timer wheels. *IEEE/ACM Transactions on Networking*, 18(3):708–721, 2010.

[16] C. Kulatunga, N. Kuhn, G. Fairhurst, and D. Ros. Tackling bufferbloat in capacity-limited networks. In *Networks and Communications (EuCNC), 2015 European Conference on*, pages 381–385, June 2015.

[17] L. Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, Apr. 1983.

[18] L. Lenzini, E. Mingozzi, and G. Stea. Tradeoffs between low complexity, low latency, and fairness with deficit round-robin schedulers. *IEEE/ACM Transactions on Networking*, 12(4):681–693, 2004.

[19] R. Olsson. Pktgen the linux packet generator. In *Proceedings of the Linux Symposium, Ottawa, Canada*, volume 2, pages 11–24, 2005.

[20] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[21] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized Zero-Queue Datacenter Network. In *ACM SIGCOMM 2014*, Chicago, IL, August 2014.

[22] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. Senic: Scalable nic for end-host rate limiting. In *NSDI'14*, pages 475–488. USENIX Association, 2014.

[23] S. Ramabhadran and J. Pasquale. The stratified round robin scheduler: design, analysis and implementation. *IEEE/ACM Transactions on Networking*, 14(6):1362–1373, 2006.

[24] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC'12*, Boston, MA. USENIX Association, 2012.

[25] L. Rizzo and G. Lettieri. VALE, a switched ethernet for virtual machines. In *CoNEXT'12*, pages 61–72, Nice, France, 2012. ACM.

[26] L. Rizzo and P. Valente. On service guarantees of fair-queueing schedulers in real systems. *Computer Communications*, 67:34–44, 2015.

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

[28] S.J.Golestani. A self-clocked fair queueing scheme for broadband applications. *INFOCOM '94*, pages 636–646, June 1994.

[29] D. Stiliadis and A. Varma. A general methodology for designing efficient traffic scheduling and shaping algorithms. *INFOCOM '97*, pages 326–335, 1997.

[30] D. Turull, P. Sjödin, and R. Olsson. Pktgen: Measuring performance on high speed networks. *Computer Communications*, 82:39–48, 2016.

[31] P. Valente. Exact gps simulation and optimal fair scheduling with logarithmic complexity. *IEEE/ACM Transactions on Networking*, 15(6):1454–1466, 2007.

[32] P. Valente. Reducing the execution time of fair-queueing packet schedulers. *Computer Communications*, 47:16 – 33, 2014.

[33] X. Yuan and Z. Duan. Fair round-robin: A low complexity packet scheduler with proportional and worst-case fairness. *IEEE Trans. on Computers*, 58(3):365–379, 2009.