A Fast and Practical Software Packet Scheduling Architecture

Luigi Rizzo, Paolo Valente, Giuseppe Lettieri, Vincenzo Maffione
Dipartimento di Ingegneria dell'Informazione
Università di Pisa
{rizzo}@iet.unipi.it – draft 20160511

Abstract

Dynamic resource scheduling is key to achieve dependable service guarantees, allocate spare capacity and protect systems against misuse. For network traffic in a cloud environment, packet scheduling is often done in software, a task made hard by the extremely high frequency of decisions (10+ million packets per second may flow on 10–40–100 Gbit/s links) and the number of concurrent sources (systems with 24–48 cores are now common).

No currently available solution *simultaneously* supports high decision rates, scales to many concurrent clients, and has provable, small deviation from ideal allocation *at high link utilization.* The pieces to make the above possible do exist, though, from efficient schedulers with tight analytical service guarantees to fast packet I/O frameworks.

In this paper we fill the gap and propose an architecture, called MYSCHED[1], to run software packet schedulers efficiently even in a high speed, highly concurrent environment. We achieve this result by separating the scheduling decision from the actual packet transmission, so that the latter can be performed in parallel by clients. We provide analytical bounds on the service guarantees of MYSCHED even at high link utilization, and present an accurate discussion of implementation issues. Our prototype can make over 20 million scheduling decisions per second even with tens of concurrent clients running on a multi-core, multi-socket system, while adding less than 2 µs to the communication delay.

1 Introduction

Allocating and accounting for spare capacity is the foundation of cloud environments, where multiple tenants share resources managed by the cloud provider. Dynamic resource scheduling in an Operating System (OS) ensures that CPU, disk, and network capacity are assigned to tenants as specified by contracts and configuration, giving performance isolation and predictable services, and allowing an effective use of resources. In this paper we focus on packet scheduling in a server that hosts cloud clients: Virtual Machines (VMs), OS containers, or any other mechanism to manage and account for resources.

Packet scheduling requires millions of **decisions per second (dps)**, matching the packets per second (**pps**) rates of modern network interfaces. Tens of clients, running concurrently different CPU cores, may *each* issue requests at such rates, so a scheduler must be able to handle concurrency and extremely high rates, provide good isolation among clients, and be robust to overload.

Terminology remark: the word "scheduler" is often used to indicate a specific "scheduling algorithm", such as DRR [26] or WF^{2}Q+ [6], which is just one of the components needed to implement a full scheduling system. In this paper we call "packet scheduler" or simply "scheduler" the entire system that i) receives packets from one or more clients, ii) decides on the fate and order of service of packets, and iii) dispatches packets to the next hop in the communication path.

OSes implement scheduling in one of the ways shown in Figure 1. The solution on the left, where the scheduler is a software module in the OS's network stack, is used in many systems including Linux with TC [3], and FreeBSD and other BSD OSes with dummynet [7] and ALTQ [9]. As discussed in Section 2, these software schedulers can

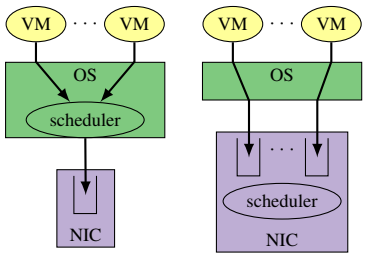

Figure 1: The path between VMs or clients and the network interface.

[1] name will be changed soon

only handle 1..2 Mpps. Replacing the scheduler with traffic shapers provides an easy way to increase the rate, but harms link utilization and guarantees.

In the solution on the right, each client has a separate transmit queue in the NIC, which implements scheduling in hardware. Removing a potentially slow piece of software from the path is only apparently an advantage: contention between clients moves to the PCIe bus, far slower than memory; NICs often have underpowered controllers (see for example the various limitations listed in [21, Sec.7.1 and Sec.8]); so hardware schedulers are not necessarily faster than software ones.

In this paper we propose a different architecture, called MYSCHED and illustrated in Figure 2: we separate the *scheduling decision*, which still remains sequential, and is made by an *ARBITER* block, from the *actual delivery of the packet to the NIC*, which is done concurrently by the clients[2] after the arbiter has granted permission. Clients never contend with each other, they just talk to the arbiter through lock free queues. The arbiter can use any scheduling algorithm, and it is the only entity interacting with it. Packets are released by the arbiter at a rate just below the link's capacity, thus preventing congestion on the link and on the PCIe bus. As a result MYSCHED scales extremely well, and permits reuse of existing and well studied scheduling algorithms and code. Further, we can establish analytical bounds on the service guarantees of MYSCHED reusing the same analysis done for the underlying scheduling algorithm.

Even with the split of decision and transmissions, building and tuning a fast, robust and scalable implementation of MYSCHED requires low level solutions that behave well on modern multi core, multi socket systems. We thus measured, and discuss in the paper, the effect of heavy memory contention on such systems, and use the results to drive our design decisions.

Our contributions then include: i) the design and discussion of MYSCHED: ii) solutions to achieve high performance and scalability; iii) a theoretical analysis of service guarantees; and iv) the implementation and performance analysis pf MYSCHED. Our prototype can deliver over 20 M decisions per second even under highly parallel workloads on a dual socket, 24 thread system, and degrades very gracefully under overload.

The code for MYSCHED is publicly available at [4].

[2]provided, of course, the availability of a multiqueue NIC; this is a standard feature on modern, fast NICs which are the target of this work.

1.1 Scope

MYSCHED belongs to a class of schedulers (such as DRR [26], WF^{2}Q+ [5], QFQ [8]) that provide tight and provable service guarantees even at high loads. 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 [20], which, due to the complexity of the problem they address, cannot give guarantees at high link utilization. More details are given in Section 6.

1.2 Paper structure

Section 2 gives some background on packet scheduling. Section 3 describes the architecture and operation of MYSCHED, followed by a discussion of implementation details. Analytical bounds on service guarantees are computed in Section 4. Finally, Section 5 reports the performance of our prototype and compares it with existing systems, and Section 6 presents related work.

2 Background

The theory. Packet scheduling refers to the task of passing packets belonging to different "flows" (defined in any meaningful way, e.g., by source client, network address or physical port ID) to a downstream link, in an order that satisfies a given requirement. Examples are giving priority to some flows over others, limiting the maximum rate of individual flows, or dividing the total capacity of the link proportionally to "weights" assigned to flows with pending transmission requests ("backlogged" flows). A scheduler 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 can serve multiple flows in parallel; this is called a "fluid system". Real links, however, are forced to serve one packet at a time [19], 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 the

Definition 1 (T-WFI) *maximum absolute value of the difference between the completion time of a packet of the flow in i) the scheduling 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.*

Bandwidth limiting or even proportional sharing are simple to achieve over coarse time intervals, but the task becomes harder and harder as we want to remain close to the fluid system on short intervals. It can be proven that the minimum T-WFI of a packet scheduler equals the transmission time of one maximum sized packet ("MSS" in network terminology). Efficient *Weighted Fair Queueing* algorithms that match this lower bound have been designed [5, 29], with scheduling decisions that have $O(\log N)$ cost in the number of flows, N.

The practice. Fast variants of Weighted Fair Queueing, such as QFQ [8] and QFQ+[30] achieve $O(1)$ time complexity per decision, at the price of a small constant increase of the T-WFI. Implementations of QFQ and QFQ+ are included in commodity OSes such as Linux and FreeBSD. Their runtime cost is comparable to that of simpler, but much less precise, schedulers such as Deficit Round Robin (DRR) [26], whose T-WFI is linear in the number of flows. All of the above implementations can make a scheduling decision in 20..50 ns on modern CPUs.

A scheduler has three components: a set of *QUEUES* to store incoming packets for the 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). Many scheduling algorithms do not use packet arrival times to make their decisions, but only rely on the state of the queues when the link is idle. Thus, in principle, nothing needs to be done on packet arrivals (other than enqueue them), and the arbiter can keep scanning the queue[3] and, every time the link is idle, run the scheduling algorithm, to assign the link to one of the backlogged flows. As an optimization, in a software scheduler the arbiter may go to sleep when queues are empty and be woken up on the next packet arrival.

Schedulers are commonly implemented as a single unit in commodity OSes, with a single thread in charge of running the arbiter and then dispatch traffic downstream. The latter is an expensive operation, so this solution does not scale with multiple concurrent clients. As shown in Section 5, schedulers in Linux or FreeBSD only reach an aggregate throughput (in packets per second, which is the relevant metric) around 1..2 Mpps[4].

[3] this is exactly how hardware schedulers in NICs work

[4] Specific OS or hardware versions may be twice as fast, but regardless of the exact value, we are targeting (and have achieved) a ten-fold speedup with our design.

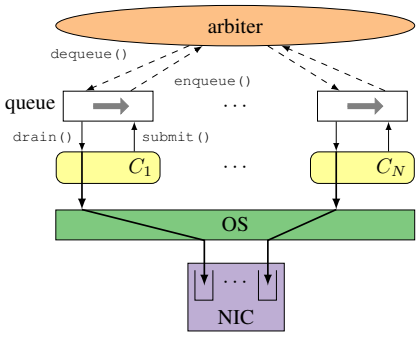

Figure 2: MYSCHED architecture.

3 MYSCHED architecture

In our architecture, MYSCHED, we split the components of the scheduler so that we can operate the arbiter in parallel with the dispatching of traffic. The overall architecture is shown in Figure 2, and includes N clients (either VMs or processes running on the host), and an arbiter unit running as a separate thread.

Each client shares a private queue with the arbiter. When packets are available for transmission, clients insert the requests in their queue through function `submit()` in Figure 3 (function `kick_arbiter()` is a a NO-OP and will be discussed in Section 3.1). Function `drain()` is used to transmit packets for which the arbiter has granted permission; it can be called opportunistically by each client, or by other threads that run periodically or on explicit notifications from the arbiter (this is very similar to the operation of an interrupt handler or NAPI thread).

The arbiter runs function `do_scan()`, continuously or every few microseconds (see Section 3.3), to collect new requests and make scheduling decisions. `do_scan()` emulates a link with a bandwidth B less than or equal to the capacity of the NIC, and uses the variable `link_idle` to track the time when the emulated link will be available for new transmissions. Every time the link is idle, `do_scan()` uses a work conserving scheduling algorithm SA (such as DRR or QFQ or others), to select a new packet to be transmitted, marks it as ready in its queue, and advances `link_idle` by the transmission time of the packet.

MYSCHED differs from conventional architectures in two ways, both crucial for performance and scalability:

- the arbiter only makes scheduling decisions. The delivery of packets to the NIC, which can be up to 10..20 times more expensive, is left to the clients,

```
1  int submit(pkt) {
2    i = pkt->queue_id;
3    cur = q[i].tail;
4    next = (cur + 1) % QUEUE_SIZE;
5    if (slots[next] != EMPTY) {
6      return ENOSPACE;
7    }
8    slots[cur] = pkt;
9    kick_arbiter(); /* possibly no-op */
10   return SUCCESS;
11 }
12
13 void drain(i) { /* drain marked packets */
14   slots = q[i].slots;
15   cur = q[i].client_head; /* next packet to send */
16   sched_head = q[i].head; /* set by the arbiter */
17   while (cur != sched_head) {
18     <transmit packet in slots[cur]>
19     slots[cur] = EMPTY; /* release the slot */
20     cur = (cur + 1) % QUEUE_SIZE;
21   }
22   q[i].client_head = cur;
23 }
```

```
1  int do_scan() {
2    now = rdtsc();
3    for (i=0; i < N; i++) {
4      while ((pkt = <new pkt in q[i]>) != NULL) {
5        SA.enqueue(pkt);
6      }
7    }
8    while (link_idle < now) {
9      pkt = SA.dequeue();
10     if (pkt == NULL) {
11       link_idle = now;
12       return NO_TRAFFIC;
13     }
14     i = pkt->queue_id;
15     link_idle += pkt->len / bandwidth;
16     q[i].head = (q[i].head + 1) % QUEUE_SIZE;
17   }
18   return MORE_TRAFFIC;
19 }
20
21 void arbiter() {
22   for (;;) {
23     do_scan(); /* ignore return value */
24   }
25 }
```

Figure 3: Client and arbiter functions.

```
1  void arbiter() {
2    arb_sleep = FALSE;
3    lock(arb_lock);
4    for (;;) {
5      if (do_scan() == MORE_TRAFFIC) {
6        /* link busy, sleep for a while */
7        sleep_till(link_idle);
8        continue;
9      }
10     arb_sleep = TRUE;
11     barrier();
12     if (do_scan() == NO_TRAFFIC) {
13       if (lock_owned(arb_lock)) {
14         unlock(arb_lock);
15       }
16       wait_event(&arb_sleep);
17     }
18     arb_sleep = FALSE;
19   }
20 }
21
22 /* 1st version: only wake up the arbiter */
23 int kick_arbiter() {
24   barrier();
25   if (arb_sleep == TRUE) {
26     send_event(&arb_sleep);
27   }
28 }
29
30 /* 2nd version: wake up arbiter and do some work */
31 int kick_arbiter() {
32   do {
33     ret = trylock(arb_lock);
34   } while (ret == ACQUIRED || arb_sleep == FALSE);
35
36   if (ret != ACQUIRED) {
37     return;
38   }
39   final = rdtsc() + some_interval;
40   do {
41     ret = do_scan();
42   } while (ret == NO_TRAFFIC || last_idle < final) {
43   /* notify and pass lock to the arbiter */
44   if (ret == NO_TRAFFIC) {
45     unlock(arb_lock);
46   } else {
47     send_event(&arb_sleep);
48   }
49 }
```

Figure 4: Algorithm changes to avoid busy wait

which run in parallel and have the opportunity to perform packet processing *after* the scheduling decision has been made;

- clients cannot interfere with each other. Each client only interacts with the arbiter through a private, lock-free queue. In conventional schedulers, instead, clients all contend to access the scheduling algorithm, which leads to scalability issues.

3.1 Avoiding busy wait

Readers may be rightfully concerned by the need to constantly run the arbiter even in the absence of traffic. The code in Figure 4 addresses this problem, sending the arbiter to sleep when it has no work to do. This can happen in two circumstances: while i) the current packet completes transmission, or ii) all queues are empty.

The first case is easy to handle: the arbiter can just block 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 to block. The following mechanism captures these events avoiding races: the arbiter informs clients of its upcoming blockage through variable `arb_sleep`, and then double checks for new requests before actually blocking. The full code for the modified arbiter is function `arbiter()` in Figure 4.

Clients now must run function `kick_arbiter()` at the end of `submit()` to check the state of the arbiter and act accordingly. This can be done in two ways, shown in Figure 4. The first one in line 22 only sends a notification to the arbiter (e.g., through an `eventfd`) if it might be sleeping (a memory barrier is necessary to ensure the correct ordering of operations).

The second solution, in line 30, runs the entire scheduler in the client when the link is idle (thus behaving as a conventional scheduling architecture), and reverts to the parallel implementation under higher load. This is achieved by having the client try to acquire the lock `arb_lock` until successful, or until the arbiter is found running. If the lock is acquired, the client runs `do_scan()` one or a few times, before releasing the lock or notifying the arbiter.

In the interest of performance, the arbiter should not go to sleep too frequently: `usleep()` has a minimum duration in the order of 10 µs, below which it is imprecise or ineffective; `unlock()` and `send_event()` can easily cost $2..5$ µs on multisocket systems[5], and acquiring a contended lock can be similarly expensive. As a consequence, when the arbiter becomes active, it should stay awake for at least $20..50$ µs to reduce the frequency of expensive operations in the clients.

3.2 Scalability

We now discuss implementation details to achieve MYSCHED's main goal, namely high performance and scalability. The rest of this Section is only relevant at extremely high rates, in the order of millions of requests per second. We assume that the operating system can provide separate transmit paths to the various clients. This is a realistic assumption with modern multiqueue NICs.

The two main concerns on MYSCHED's scalability thus are i) the need to periodically scan *all N queues* to collect traffic, and ii) memory contention in accessing the lock-free queues. Ultimately, both are impacted by the characteristics of the memory subsystem in the host.

3.2.1 Scanning queues

The absence of explicit notifications of new packets requires a scan of all N queues in `do_scan()`. N is relatively small (up to 50..100), and the region of the queue being scanned, for idle clients, is likely to be in the L1 or L2 cache of the arbiter, resulting in access times in the order of 2..4 ns each (we have measured these values on a dual-socket E5-2640 system). On the same platform a cache miss (which is unavoidable with active clients, even if we had explicit notifications) can easily cost in excess of 200 ns. If follows that, for practical values of N, the cost of scanning idle queues is comparable to a single cache miss. Additionally, scanning all queues in a tight loop permits read requests to be posted in parallel, thus amortizing the cost of cache misses, which gives a significant advantage even with 2-3 misses handled in parallel.

3.2.2 Memory contention on queues

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

MYSCHED follows the design presented in FastForward [10]: the arbiter (consumer) never accesses the producer's insertion pointer, and instead detects new insertions from the content of the slot in the queue: a special value (in our case, a packet length of 0) marks an empty slot, other values indicates that the slot is full and (implicitly) that the insertion pointer advances. This saves one read stall and an even more expensive memory barrier.

FastForward uses the same technique also to pass the consumer's pointer to the producer: the consumer updates queue slots with the "empty" value, notifying the producer and preparing the slot for new insertions. However that approach would cause write-write conflicts on the queue between producer and consumer, and likely on the same cache line as queues are always almost full or almost empty. Hence, in MYSCHED a client looks at the arbiter's (consumer) pointer to determine which packets have been marked, and the client itself then clears the marked slots.

Cache conflicts also occur when the producer updates multiple queue slots in the same cache line while it is accessed by the consumer. FastForward addresses this problem by slowing down the consumer, so that the two parties operate on different cache lines. We cannot use this approach because we may not have subsequent requests, and latency matters. We instead make the arbiter take snapshots of a few cache lines of data around the current insert position in the queue, and read from the local snapshot until it finds an empty slot.

3.3 Reader-writer speed and latency

Even with the techniques discussed so far, MYSCHED will incur in cache misses when exchanging information

[5] single socket systems are 2-4 times faster, but we target machines with high core counts.

between clients and the arbiter. To determine how we should deal with these misses, we need to measure, under heavy load, the costs of reading and writing shared memory, the amount of information that can be passed around, and the latency incurred.

To this purpose, we have run experiments on a dual socket Xeon E5-2640 system running at 2.5 GHz, with two threads, called W and R, on different cores accessing one or more shared variables (few enough to fit in L1 cache). Both threads can issue read or write requests at different rates, up to one per clock cycle if the memory coherency protocol does not stall them. In one experiment W writes a stream of increasing 64-bit values to one or more shared variables V_i, and R counts how many reads it can complete and how many different values it sees per unit of time. In a second experiment, W and R run a request/response protocol, so that the value written to V_i is increased only when the R confirms that has seen the previous update. In this case, we measure the number of round trip transactions per unit of time.

We found the following:

writes can be fast. Writes to a single memory location can be posted up to once per clock cycle. A reader on the same *single* memory location on a different core does not slow down writes;

readers may slow down writers. When accessing *multiple* cache lines, a busy reader on a different socket may significantly slow down a writer to the same cache lines. We measured writes times of as much as 100 ns with W and R on different sockets, versus 15 ns when W and R are on the same socket;

reads misses are expensive. Cache lines can be invalidated by writes, causing a miss on subsequent reads. We measured the following read times on a miss: hyperthreads on the same core: 10-15 ns; cores on the same socket: 50 ns; cores on different sockets: 130-220 ns. When threads run on different sockets, we found that if memory is local to the reader, misses are much longer than if the memory is remote;

updates are infrequent. Not all values written are visible to other cores. Cache updates are triggered by reads, but following an update (which in our experiment is indicated by a different value being returned), a core sees the same value for some time even though the writer keeps updating the same location. We found: hyperthreads on the same core: 75 M updates/s; cores on the same socket: 20 M updates/s; cores different sockets: 4-5 M updates/s. Between updates, the same value is seen for a duration comparable to the read miss latency.

latency is high. Partially related to the previous two items, latency in passing information from one core to another is typically high. Latency is at least the sum of the invalidation and reload times. For a complete request-response exchange, we found: hyperthreads on the same core: 30 ns; cores on the same socket: 130 ns; cores different sockets: 480 ns.

All the above figures vary significantly with different CPU types, but multi socket systems (an important target for MYSCHED) can be 4-5 times slower than single socket systems in dealing with shared memory. While memory access latencies are well known, interference between readers and writers is less known and yet it may have a significant impact on a system like MYSCHED, where parties communicate through updates to shared memory and lack explicit notifications. We thus need to design the algorithms in the clients and arbiter to minimize the effect of the behaviour of the memory subsystem. The resulting tradeoff between latency and performance will be a parameter of our design space.

We can easily mitigate the interference between readers and writers by rate limiting accesses to shared memory locations on both reader and writer sides. Empirically, we found that spacing batches of reads and writes by $1..2\,\mu s$ makes it possible to handle groups of 50..100 shared variables with an amortized read/write cost of 20 ns, even when threads are on different sockets and all variables are updated on each round. Reading unmodified variables will only cost the 2..4 ns measured in Section 3.2.1. We implement this spacing by rate limiting reads on shared data: the arbiter reads each queue at most once every Δ_A seconds, and each client checks the arbiter's head pointer at most every Δ_C seconds. These two parameters are in the $1..5\,\mu s$ range.

Unfortunately we cannot completely avoid read stalls: CPUs provide various `PREFETCH` instructions to issue read requests ahead of time, but they are only advisory and can be overridden by cache invalidations. Our queue access scheme would work perfectly well if the CPU had instructions to read stale values from the cache without paying for the miss penalty: in fact, slots and queue pointers are updated in a detectable way (slots alternate between EMPTY and non-empty values; pointers always advance) and cannot wrap around without a feedback from the other party. However, no popular CPUs provides such "non blocking, possibly stale read" instruction.

As a workaround, the rate limiting and batching of reads also turns out to be useful for this. By letting reads be posted in parallel, on the arbiter, we can amortize the effect of cache misses.

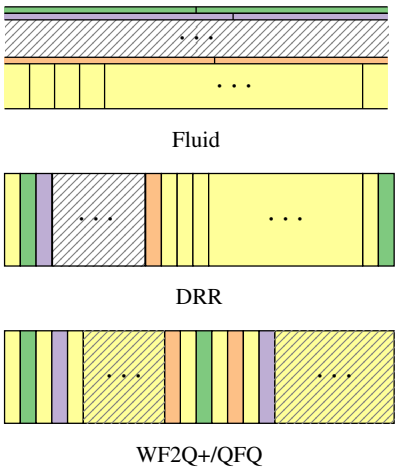

Figure 5: Transmission order in fluid and packet schedulers. Different service orders may achieve long term bandwidth allocation, but some have a lot more burstiness.

4 Service guarantees

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, which affects very badly the performance of request-response exchanges.

As an example, Figure 5 shows packet transmission in a fluid scheduler and in two cases (representing the DRR and WF2Q scheduling algorithms), with radically different behaviours: in DRR, the T-WFI is proportional to the number of flows; in WF2Q, QFQ and other algorithms the T-WFI is bounded by a constant independent of the number of flows. In general, achieving a good approximation on shorter intervals (hence keeping the T-WFI small) requires more expensive computations in the scheduling algorithm, hence there is a tradeoff between T-WFI and runtime costs.

MYSCHED per se does not define new scheduling algorithms, but uses them within the arbiter, so the purpose of this Section is to determine the overall T-WFI of MYSCHED given the T-WFI$_{\text{SA}}$ of the underlying scheduling algorithm SA.

4.1 T-WFI in MYSCHED

T-WFI has been computed in the literature for several work conserving scheduling algorithms that we can use in MYSCHED, see for example [25]. The analysis in [25] shows that the T-WFI of a 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 other artifacts outside the scheduling algorithm (commonly, the presence of a FIFO in the communication device, which is the one analysed in [25].) In MYSCHED, artifacts are different, so goal here is to determine the overall T-WFI of MYSCHED given the base T-WFI$_{\text{SA}}$ and the way we drive the scheduler algorithm and the link.

In this analysis we make the following (realistic) assumptions, to ensure that the system is well configured: 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 client runs `client_drain()` every Δ_C seconds.
As for how multiple queues are served by the NIC, round-robin is a realistic expectation, as it is trivial to implement in hardware and prevents starvation.

Under these assumptions, the arbiter may see incoming packets, hence enqueue them into the scheduler and dequeue them from the scheduler, 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 (such as the one we consider) where decisions are made only when the link is idle.

We call a "block" the amount of traffic scheduled, i.e., marked, in every interval Δ_A. This can amount to at most $B \cdot \Delta_A$ bits, plus one maximum sized packet L[6]. 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 marks up to one block of data at once, and clients send marked packets to the link in parallel, the order of transmission may be different from the one decided by the arbiter. Let's first assume that clients operate with $\Delta_C = \Delta_A$ and are synchronised with the arbiter. This adds a delay term Δ_C 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_C.

When $\Delta_C \neq \Delta_A$ and/or they are not synchronised, a further complication occurs, as the link may receive at once up to $B \cdot (\Delta_C + \Delta_A)$ bits, more than the capacity of the link before the next round of transmissions from the clients. The excess block $B\Delta_A$ that remains at the end of the round will in turn be reordered together with the content of the next round (which this time is within the link's capacity). We omit the proof for brevity, but the number of Δ_C 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. Fortunately, for practical

[6]The arbiter marks all packets that start transmission within the current interval Δ_A, so the last one may complete within the next interval.

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 $k = 1$.

Putting all pieces together, we have

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

While the above completes the theoretical evaluation, moving from theory to practice, we can consider what happens if clients or the arbiter occasionally miss their deadline Δ_C and Δ_A for dispatching or marking packets. For the arbiter there is little we can do other than accept an increase of the T-WFI for all traffic due to a larger Δ_A (and k). For clients, we can enforce clients that miss their deadline Δ_C to resubmit their requests[7] so that well behaved clients will not be affected.

4.2 T-WFI examples

To put number in context: from [25] we know that

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

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

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 $2\Delta_A + 2\Delta_C$ (between 2 and 4 µs) introduced by MYSCHED is small or negligible.

5 Experimental results

We ran a number of experiments to evaluate the performance of MYSCHED, in terms of both maximum achievable throughput and service guarantees, and compare it with existing alternatives. The test scenarios have been chosen carefully and rigorously to emphasize the phenomena under investigation (cost of the 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 Scope again

In addition to the considerations in Section 1.1, we have a few more remarks related to experimental conditions.

Userspace vs. kernel. Running MYSCHED in user space does not affect the validity of our results; clients and arbiter would use the same code and interact exactly in the same way if all the components were in the kernel. In a VM hosting platform, each VMs has dedicated kernel threads that perform I/O (specifically network I/O, as in the case of `vhost-net`) on behalf of the VM, and would be perfect candidates to run the client code in Figure 3.

Absolute performance. We know that different CPU models and OS versions may differ in performance by a large factor (in fact, one of our test platforms is twice as fast as the other). More than reporting the best absolute performance, our goal is to point out certain architectural features of the systems under analysis (typically, scalability and behaviour under load) that exist across the board; specific CPU or OS versions do not influence our conclusions, especially because our system performs one order of magnitude better than existing solutions.

5.2 Test environment

Our main test platform, which we call XEON2, is a dual socket system with two Xeon E5-2640 CPUs at 2.5 GHz (6 cores, 12 threads each), with 1.33 GHz DDR3 memory running CENTOS with Linux kernel 2.6.32-504. Additional experiments have been run on another system called I7, which is a single-socket i7 CPU at 3.5 GHz (4 cores, 8 threads) running Linux 4.5 (this version supports the netmap framework). Additional software used for experiments includes the TC [3] scheduling framework; the netmap [23] framework and VALE [24] software switch for high speed I/O; and scheduling algorithm implementations are taken from the dummynet [7] link emulator (they match the performance of those used in TC).

The prototype implementation of MYSCHED runs entirely as user space threads, and includes synthetic clients to generate traffic with specific profiles (duration, rate, packet size and burst size). The arbiter is implemented as described in Figure 3, except that shared variables are read at most every Δ_A and Δ_C seconds. Parameters of

[7] the arbiter grants permission to transmit a certain amount of data, irrespective of which packets they are. Thus, it is feasible to "resubmit" requests without causing reordering. We omit details for simplicity.

the scheduling algorithm, queue size, output bandwidth are specified through the command line.

In all cases the experiments involve running one or more clients on different cores, each generating traffic with configurable rate, packet size and burst size. Clients issue requests through UDP sockets, or directly to MYSCHED, and in the latter case they complete the transmission using either UDP sockets, or netmap ports to reach higher I/O speeds.

With our workload, running two clients on hyperthreads of the same physical core slows down both processes considerably (the pair delivers only 1.2..1.3 times the throughput of a single client). Clients are thus allocated by first filling up cores, then CPU sockets (starting from the first one), so that the offered load is monotonically increasing, although with different gaps when going to even or odd number of clients. To minimize interference, the arbiter, if present, is allocated on a different core, and on XEON2 it is on the second socket, so that we can measure the worst case cost of the interaction with clients.

Operating parameters. Unless specified otherwise, in all experiments we have used 60 byte packets (including MAC headers) to minimise the OS overhead in copying data; the DRR scheduling algorithm (the fastest available; QFQ would add another 20-30 ns per packet, which is a measurable difference with MYSCHED) with a quantum size of 1 packet, queue and bandwidth large enough not to act a bottleneck; clients have the same weight, and send as fast as possible; and MYSCHED uses parameters $\Delta_A = 1000$ ns, $\Delta_C = 1000$ ns.

In all experiments, traffic goes through the loopback interface (for UDP) or VALE switches (for netmap), using different destination ports to avoid contention. 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.

Each single measurement is repeated 10 times, and arithmetic mean and standard deviation is computed over the 10 trials.

5.3 Metrics

pps vs decisions per second: for packet processing systems, the load has little dependency on the packet size, so the metrics of interest for throughput are normally expressed in "packets per second" (pps). On high speed links, the transmission time for short packets is so small (down to 67 ns on a 10 Gbit/s port, and 16 ns on a 40 Gbit/s port) that it makes no sense to schedule individual packets, let alone the fact that the scheduling algorithm may not be fast enough. Many systems thus may aggregate packets (if available) in batches of up to several MSS worth of data, and make a single scheduling decision for the entire batch. Our MYSCHED prototype implements this option, but here **we only report results with aggregation disabled**, so the number of "decisions per second" equals the "pps' figure.

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 and cache misses, which ultimately leads to variable processing times in the system. We thus look at a related metric, namely the latency distribution in one-way communication between a client and a receiver.

5.4 Maximum 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 MYSCHED. With link and client speeds set as high as possible, any throughput limitation is only due to the performance of the scheduler or UDP sockets.

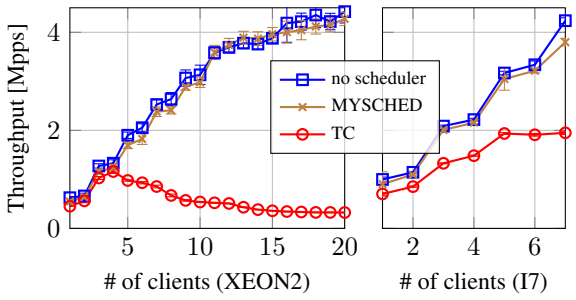

Figure 6: Maximum throughput in packets per second, with UDP and different schedulers on XEON2 and I7. Error bars show two standard deviations from the mean.

Figure 6 shows the results of the measurements on XEON2 (up to 20 clients) and I7 (up to 7 clients). With no scheduler, the aggregate throughput reaches about 4.5 Mpps, (I7 has about twice the speed of XEON2 for the same number of cores). There are no particular sign of saturation, but a clear change of slope as we start using cores on the second socket on XEON2.

Enabling TC significantly impacts throughput and scalability on both platforms. Up to 4 clients we lose about 30%; above that, throughput quickly flattens or (in the case of XEON2) declines, down to 0.32 Mpps with 20 clients. Conversely, MYSCHED not only keeps the throughput close to the no-scheduler case, but also avoids

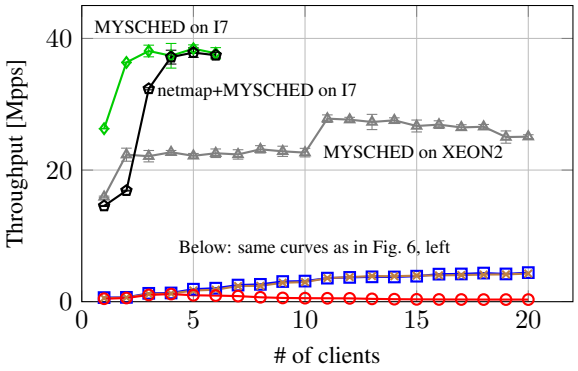

Figure 7: Throughput of MYSCHED with netmap and different configurations. Error bars show two standard deviations from the mean. The curves at the bottom are the same as in Figure 6.

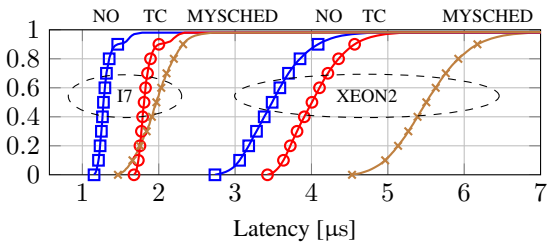

Figure 8: Latency distributions of UDP one way delay with no interfering nodes, for I7 and XEON2.

saturation: the two curves remain extremely close up to the end of the range.

MYSCHED can actually deliver much higher packet rates, as shown in Figure 7. Here we run experiments on system I7 (up to 6 clients, due to fewer cores available), with clients using MYSCHED as a scheduler and netmap to send data. The results are shown by the curve labeled netmap+MYSCHED on the top left of the figure: we see that MYSCHED can schedule almost 40 Mpps (one packet per decision).

Given the different nature and speed of the two test platforms, we ran two more experiments, where clients submit requests to MYSCHED but replace transmissions with a no-op. On I7 (curve on the top left), we are very close to the netmap+MYSCHED case. On XEON2, the maximum decision rate is significantly lower (between 16 and 27 Mpps), but still 5 times higher than when we do UDP send. 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).

Comparison with other solutions. With almost 40 M decisions per second on I7, MYSCHED exceeds by a large factor the performance of other solutions (in fact, it also exceeds the ability of current NICs to generate traffic). As a reference, Fastpass [20] reports, for the arbiter alone, the use of 8 cores to schedule 2.2 Tbit/s with a request size of 10 MTU, or 15 Kbytes. This equals to approximately 20 M decisions per second. Conversely, MYSCHED runs the entire interaction with clients (not just the scheduling algorithm) at twice the speed and using just one core.

5.5 Latency distribution

For the experiments on latency distributions, we set the first "TARGET" client to send at 4 Kpps, (thus using a small fraction of the link's bandwidth,) but it has a weight 50 times higher than all other "interfering" clients, which instead generate traffic as fast as possible. In these experiments we use $\Delta_C = \Delta_A = 500$ns, and only UDP because of the low data rate on the TARGET.

Packets from the TARGET client are marked with a TSC timestamp when they are submitted to the scheduler; the TSC is read again on the receiver[8] to compute the one way latency. We can use the TSC as sender and receiver are on the same system with invariant and synchronous TSCs across sockets.

The baseline (no interfering clients) for latency distributions is shown in Figure 8: the three curves on the left are measured on I7 with, from left to right, no scheduler, TC and MYSCHED. The three curves on the right show the same configurations on XEON2. TC adds approximately 300..600 ns over the base case, while for MYSCHED the extra latency is 0.5..2 μs, entirely due to the round trip time for shared memory exchanges with the arbiter (on the other socket for XEON2 experiments) and the two intervals Δ_A and Δ_C.

When interfering clients are present, latency is proportional to the amount of traffic scheduled before that of the target client, divided by the link's bandwidth. The actual value and distribution, depending on the scheduling algorithm, (see Section 4), vary between a constant (in the case of WF2Q+/QFQ), and a linear term in the number of interfering flows for weaker algorithms such as DRR (the one used in these tests). Regardless, latency should decrease as the link's bandwidth increases, as long as the scheduler is not overloaded. At that point, the scheduler will not be able to to keep up with requests, resulting in a latency increase (and possibly, allocation mismatches; but

[8] the first receiver now does read packets, and runs on a separate core using non-blocking I/O to avoid sleeping and the inherent latency jitter.

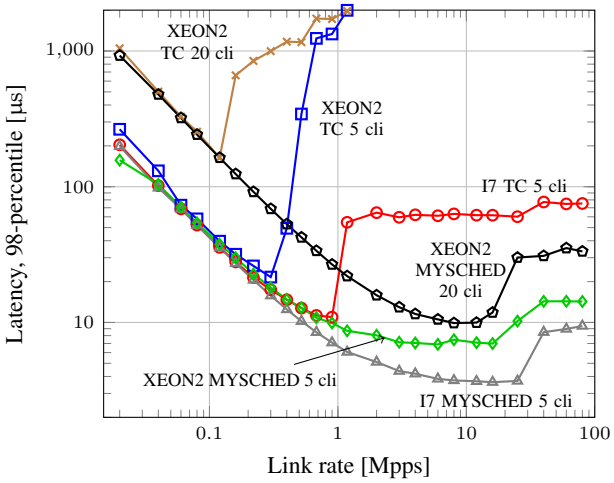

Figure 9: 98-percentile of the latency for TC and MYSCHED at different link rates (90-percentile curves are very similar). Saturation is indicated by a sudden increase in the latency. Note the logarithmic scales on both axes.

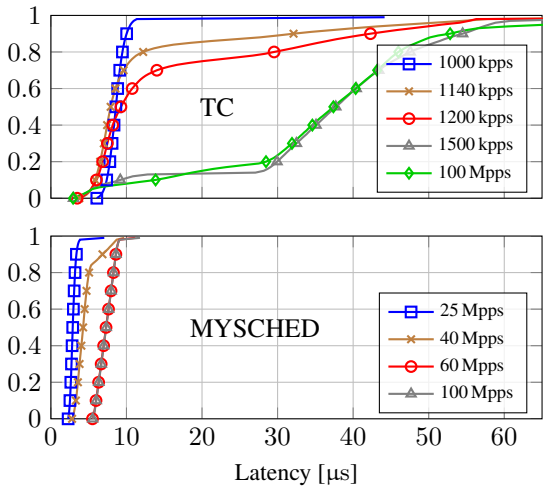

Figure 10: Distribution of latencies around overload for the two schedulers running on I7 with 5 clients. The thresholds are very different (1.1 Mpps for TC, over 25 Mpps for MYSCHED), and yet MYSCHED has a much lower latency under heavy overload.

we only measure latency here).

To determine the overload threshold, we ran experiments with different link speeds (we use 'pps' as the parameter as it is a more significant quantity, unrelated to packet size), and plot the 98-percentile of the distributions (higher percentiles would just reflect artifacts of the OS, such as interfering interrupts.)

The results are shown in Figure 9 (very crowded due to space limitations). With TC, the threshold varies between 200 Kpps (on XEON2 with 20 clients), 500 Kpps (on XEON2 with 5 clients) and 1.1 Mpps (on I7 with 5 clients). All these values are much lower than the maximum throughput measured in Figure 6. Beyond the threshold TC misses its service guarantees, and latency grows significantly, by over 5 times on I7, and by almost two orders of magnitude on XEON2 (presumably an artifact of an older version of TC so we do not claim this to be a property of TC).

MYSCHED on the same hardware/OS behaves much better, with overload triggering above 15 Mpps even in the worst case (XEON2 with 20 clients), and over 25 Mpps on I7 with 5 clients. Especially interesting is the fact that the maximum latency, even under severe overload, stays between 10 and 30 μs depending on the platform, or five times lower than TC in all cases.

We further explore the latency distributions at selected rates around the overload threshold. In Figure 10 we show the latency distribution for the two schedulers around the overload threshold, in both cases running on I7 and with 5 clients (the most favourable of our test scenarios for TC). The figure shows clearly how even in this case TC degrades significantly under a modest load (about 1 Mpps), while MYSCHED remains below 10 μs at all loads up to 100 Mpps. Even in the worst scenario (XEON2 with 20 clients, not shown due to space limitations), MYSCHED'latency stays below 30 μs. When overloaded, MYSCHED has to scan all N queues at each scheduling decision; while there is a linear term, our mechanism to rate limit queue accesses and cache misses are very effective, so there constants are very small and the latency remains bounded and small.

In fairness, MYSCHED achieves such good performance because it uses an extra core to run the arbiter. However, we can use the solution in Section 3.1 to switch to client-based operation under light load, and pay the extra cost only at high packet rates when the additional core is put to good use.

6 Related work

Scheduling algorithms have been extensively studied in the 90's for their theoretical properties [19, 6, 5, 28, 27] and later for efficient implementations [29, 8, 30, 15, 31, 11, 18, 22]. 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 re-

ceived 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, 23] network stacks were uncapable 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 [20]), 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 [21] 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 [20], which has some high level similarity with MYSCHED. 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 MYSCHED, and cannot give strict service guarantees[9] and is several times more expensive than ours.

7 Conclusions

We have presented MYSCHED, a framework for building high performance, highly scalable packet schedulers that decouple scheduling decisions (inherently sequential) from actual packet transmissions, which can occur in parallel on modern hardware. MYSCHED is designed to eliminate interference between clients by having a scheduling thread handle requests from all clients asynchronously. Finally, the design of our framework allows us to make a worst case analysis of its service guarantees. The analysis shows that MYSCHED adds only a couple of microseconds of latency to that of the base scheduler.

We have implemented MYSCHED and evaluated its performance on single and dual socket systems and a variety of load configurations. Even with Even with 20 concurrent clients and a dual socket machine, our 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 stays below 10 μs. All these figures are over 10 times higher than those available with the scheduling frameworks in existing operating systems.

For the high packet rates it supports, MYSCHED is certainly a 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,

[9] As clearly indicated by the authors, the bound given in the paper [20] only applies if link utilization is less than 50%

e.g. instances of Virtual Network Functions, may generate traffic with extremely high packet rates. Our prototype is very easy to integrate with hypervisors, or to run within the kernel – also because the code for the scheduling algorithms comes directly from dummynet, and is already being used in the FreeBSD, Linux and Windows kernels. The code for MYSCHED is entirely BSD licensed and available at [4].

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] 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.

[20] 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.

[21] 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.

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

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

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

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

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

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

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

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

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

[31] 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.