PSPAT

PSPAT: Packet Scheduling with PArallel Transmit

Luigi Rizzo, Paolo Valente, Giuseppe Lettieri, Vincenzo Maffione*
*Università di Pisa, *Univ. di Modena e Reggio Emilia*
http://info.iet.unipi.it/ luigi/research.html

Work supported by H2020 project SSICLOPS

Paper at http://info.iet.unipi.it/ luigi/papers/20160511-mysched-preprint.pdf

Problem statement

- Network links way too fast
- NICs and operating systems are catching up
- VMs are catching up, too
- Software Packet scheduling not there yet

Why do we care about **software** schedulers ?

- first block in the network path for VMs

Scheduling

Contract between client and resource manager:

"If you behave, I'll give you some guarantee"

- conditions must be under control of individual clients
  - ``*if other clients behave*'' would be a bad conditions
- guarantees are a binding promise for the scheduler
  - we want a solution with known theoretical service guarantees

Scope and limitations

Theory gives us several algorithms with tradeoff between performance and guarantees

- **DRR** (deficit round robin): 20 ns/decision, *O*(N) delay
- **WF2Q+** (Weighted Fair Queueing): *O*(log N) time, *O*(1) delay
- **QFQ/QFQ+** (Quick Fair Queueing): 40-50 ns/decision, *O*(1) delay

Operating conditions:

- request rates up to 1..10 M/s
- response times < 1us

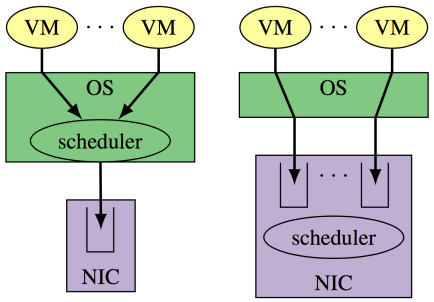
Traditional scheduler placement

SW: Both **decision** and **dispatch** are serialized.

- unnecessary serialization
- prevents scalability

HW: the NIC takes care of scheduling

- limited choice of algorithms
- the bus is still a point of contention.



Cutting corners

Things we do when we are too slow:

- trivial schedulers (FIFO, DRR: fast but poor delay guarantees)
- active queue management (RED, CODEL: rely on **everyone** behaving)
- bounded number of queues: rely on quiet neighbours

and we give up on guarantees.

Wrong approach!

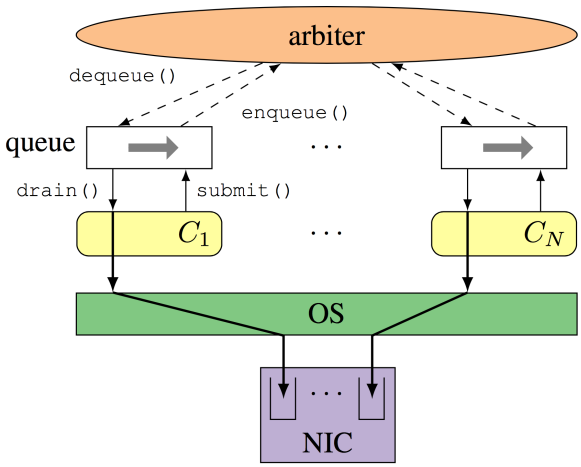
- the algorithms are good enough
- we need to remove the unnecessary tx serialisation

PSPAT key idea

Decouple scheduling and transmisison

- decision must be sequential, transmission does not need to
- scheduler (arbiter) only indicates **WHEN** a packet can be transmitted
- clients then transmit in parallel, can enjoy an uncongested path to the wire.
- can still do a worst case analysis

PSPAT scheduler placement



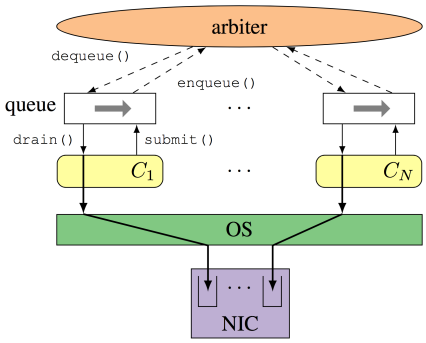
PSPAT scheduler placement

Operation:

- clients submit packets to the queue
- arbiter marks those ready for transmission
- clients do the transmit

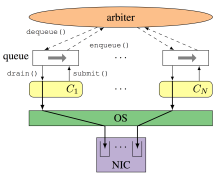
Pros and cons

- client and arbiter can operate lock free
- no extra queueing
- arbiter has to scan through all queues



PSPAT pseudocode: Client Submit

Clients just put packets in the queue

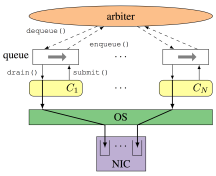



```
int submit(pkt) {
  i = pkt->queue_id;
  cur = q[i].tail;
  next = (cur + 1) % QUEUE_SIZE;
  if (slots[next] != EMPTY) {
    return ENOSPACE;
  }
  slots[cur] = pkt;
  kick_arbiter(); /* only if arbiter can sleep */
  return SUCCESS;
}
```

- no barrier needed, as in **FastForward**
- synchronization only through current slot

PSPAT pseudocode: Client Transmit

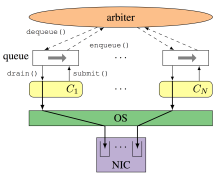
Client drains marked packets and reset queue slots




```
void drain(i) { /* drain marked packets */
  slots = q[i].slots;
  cur = q[i].client_head; /* next packet to send */
  sched_head = q[i].head; /* set by the arbiter */
  while (cur != sched_head) {
    <transmit packet in slots[cur]>
    slots[cur] = EMPTY; /* release the slot */
    cur = (cur + 1) % QUEUE_SIZE;
  }
  q[i].client_head = cur;
}
```

- again no barrier, sync only on consumer pointer
- queue is only written to by the client

PSPAT pseudocode: Arbiter scan

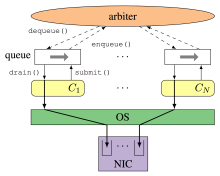



```
int do_scan() {
   t0 = rdtsc(); /* take a timestamp */
   /*-- first pass, scan queues --*/
   for (i=0; i < N; i++) {
      while ((pkt = <new pkt in q[i]>))
         SA.enqueue(pkt);
   }
   /*-- 2nd pass, mark packets due for tx --*/
   while (link_idle < t0) {
      pkt = SA.dequeue();
      if (pkt == NULL) {
         link_idle = t0;
         return NO_TRAFFIC;
      }
      i = pkt->queue_id;
      q[i].head = (q[i].head + 1) % QUEUE_SIZE; /* mark! */
      link_idle += tx_time(pkt->len, bandwidth);
   }
   return MORE_TRAFFIC;
}
```

PSPAT additional code features

Several tricks reduce cache bouncing and misses

- lock free queue with various slot and pointer caching tricks
- queue slots only updated by the client
- arbiter can skip non-empty queues on scans
- idle queues will likely be cached on the arbiter
- rate limit (memory) access to queues



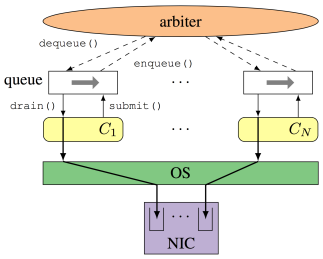
Performance analysis

Measure throughput and latency of different configurations:

- UDP, no scheduler
- UDP, with TC
- UDP, with PSPAT
- just PSPAT
- fast I/O (netmap) with PSPAT

Special interest in performance at high loads

- hopefully no livelock or latency explosion



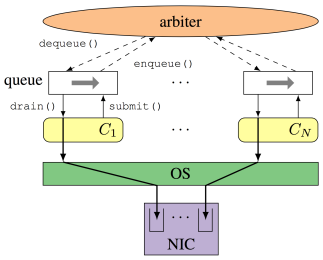
Measurement setup

Traffic generators

- one thread per client, pinned to a core, programmable rate
- tests over loopback interface
- 2 output mechanism: sockets or netmap
- 3 scheduler architectures: none, TC, PSPAT

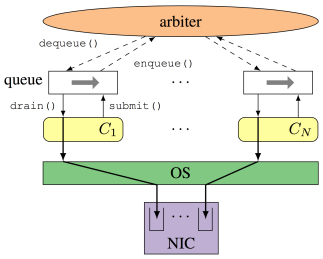
Two different platforms

- single socket I7, linux 4.5, 4 cores/8 threads
- dual socket E5, linux 2.6.32, 12 cores/24 threads

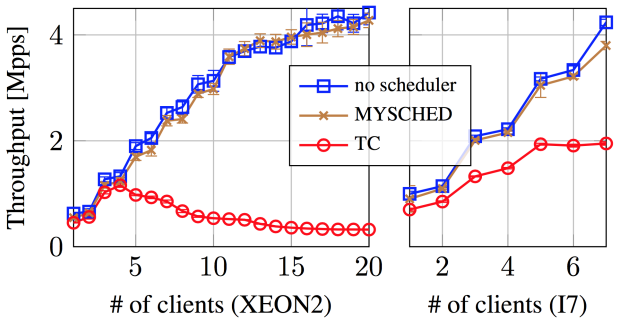


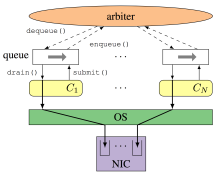
Throughput measurements

- Clients send as fast as possible
- variable number of clients
- schedulers use DRR
- TC and PSPAT rates higher than scheduler's capacity
- measurements in PPS as that is the relevant metric

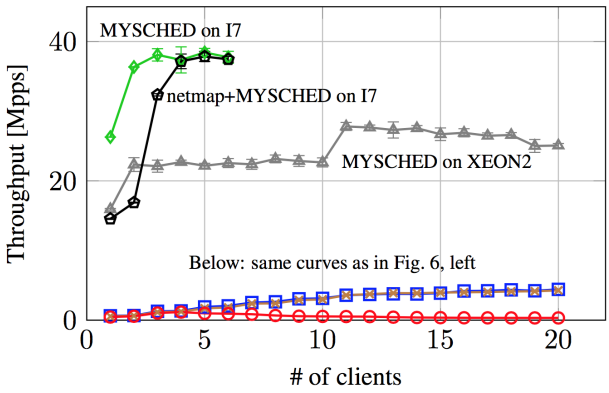


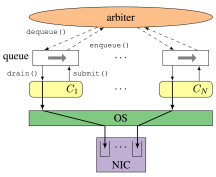
Throughput with regular UDP





Throughput with fast I/O



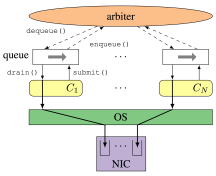


One way latency measurements

Experiments with different link rates and number of clients

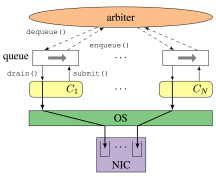
- one client has **weight=100**, sends at half the reserved bandwidth
- other clients have **weight=1**, send as fast as possible
- only UDP sockets

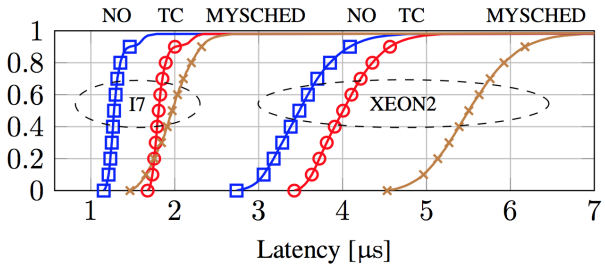
Theory says latency is proportional to MSS/RATE



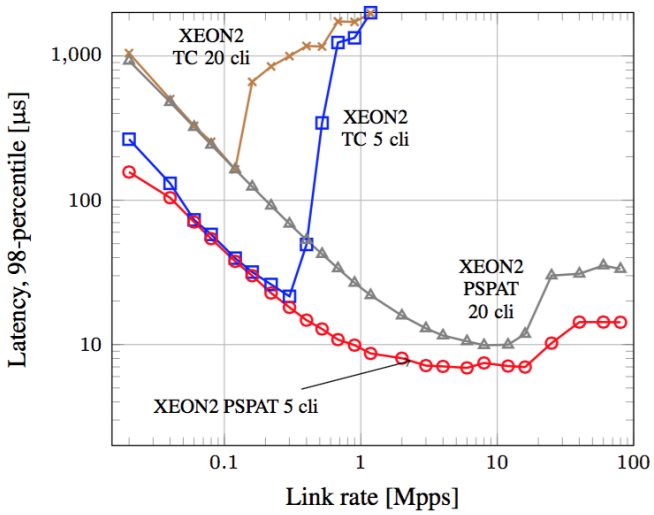
Baseline latency distribution: only high weight client

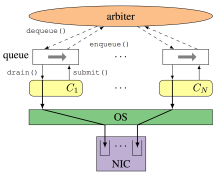
Link always idle here, so only syscall + communication overhead



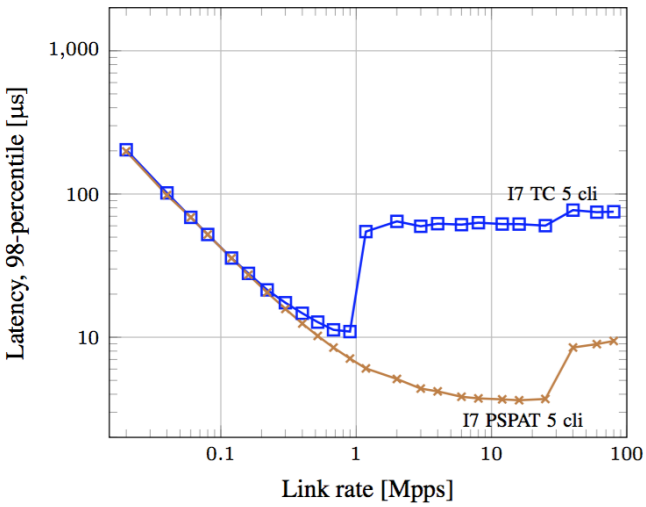


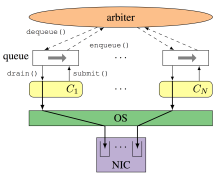
Latency versus rate, Xeon + linux 2.6.32



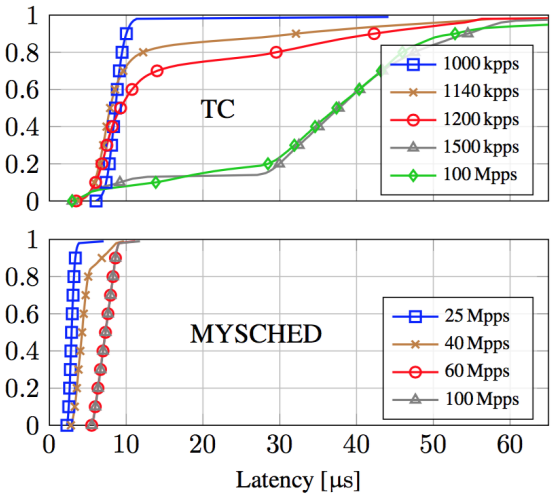


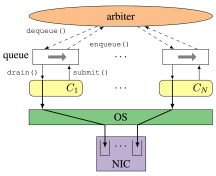
Latency versus rate, I7 + linux 4.5





Latency under overload (I7)





Summary

- Throughput and latency tests very encouraging
- potentially 5-10x higher throughput
- very small latency increase at low load
- much better latency under high load
- can fall back to NAPI-like behaviour at low load

http://info.iet.unipi.it/ luigi/research.html
http://info.iet.unipi.it/ luigi/papers/20160511-mysched-preprint.pdf