Sharing CPUs via endpoint congestion control

Laura Vasilescu
laura.vasilescu@cs.pub.ro

Vladimir Olteanu
vladimir.olteanu@cs.pub.ro

Costin Raiciu
costin.raiciu@cs.pub.ro

University Politehnica of Bucharest
Splaiul Independentei 313a
Bucharest, Romania

ABSTRACT

Software network processing relies on dedicated cores and hardware isolation to ensure appropriate throughput guarantees. Such isolation comes at the expense of low utilization in the average case, and severely restricts the number of network processing functions one can execute on a host.

In this paper we propose that multiple processing functions should simply share a CPU core, turning the CPU into a special type of "link". We use multiple NIC receive queues and the FastClick suite to test the feasibility of this approach. We find that, as expected, per core throughput decreases when more processes are contending; however the decrease is not dramatic: around 10% drop with 10 processes, and 50% in the worst case where the processing is very cheap (bridging). We also find that the processor is not shared fairly when the different functions have different per packet costs. Finally, we implement and test in simulation a solution that enables efficient CPU sharing by sending congestion signals proportional to per-packet cost for each flow. This enables endpoint congestion control (e.g. TCP) to react appropriately and share the CPU fairly.

CCS CONCEPTS

• **Networks → Middle boxes / network appliances**; **Network resources allocation**; *Network protocol design*;

KEYWORDS

CPU sharing, congestion control

ACM Reference format:
Laura Vasilescu, Vladimir Olteanu, and Costin Raiciu. 2017. Sharing CPUs via endpoint congestion control. In *Proceedings of KBNets '17, Los Angeles, CA, USA, August 21, 2017,* 6 pages.
https://doi.org/10.1145/3098583.3098589

1 INTRODUCTION

Modern networks are more than just bit pipes; they also offer processing and sometimes even storage. Traffic no longer just flows through, it is also processed in the network by "middleboxes" including firewalls, WAN optimizers, tunnel endpoints or application-level caches. As network processing in ASICs or bespoke hardware is difficult to upgrade and scale, a recent push towards running network processing as software on commodity hardware, called NFV, has already moved from research into deployment: major operators are running virtualized network functions on small data centers deployed throughout their networks. This trend is only set to continue, with datacenters being pushed close to the clients at the mobile edge (a trend called Mobile Edge Computing). In data centers, operators rely on commodity processing to implement various functionality (e.g. firewalling, NATs, load balancing).

The shift towards in-network processing stems, to some extent, from the popularity of cloud computing, and it comes as no wonder that the current mindset towards optimizing new networks is similar to optimizing clouds: the general formulation is that a user wanting service will specify the amount of bandwidth and processing it needs for its traffic, and a centralized controller will solve an optimization problem to decide where to place the computing and how to route the traffic [2]. We observe that this approach has a few drawbacks: a) it is very difficult to characterize traffic demand and processing demand ahead of time, b) provisioning decisions are made on subsets of the path (e.g. the ISP network), and thus do not take into account the end-to-end properties of traffic, and finally c) one needs to reserve resources for peak load, leading to wasted resources. Putting this in context, we can conclude that modern multi-resource networks are allocated using the equivalent of circuit switching in the old telephone networks. While this type of allocation offers great isolation properties, it is also very inefficient.

The starting point of this paper is the observation that all resources, including processors, main memory and storage, are used, most often, to produce, consume or to transfer data, and can be viewed as special types of network "links" with different speeds.

We want to explore what would happen if we do away with cloud-style reservations for in-network processing: is it possible to adopt the best-effort per-packet service in the context of sharing other resources beyond bandwidth? In other words, we would like to rely on endpoint congestion control to share network processing resources in the same way it shares network capacity.

We rely on experimentation to see whether traditional OS process scheduling is sufficient to share network processing. Our results show that sharing CPUs via traditional OS mechanisms is suboptimal, as it leads to inefficient and unfair resource allocations in many situations. Next, we propose changes to the buffering discipline used by the network processes that remedy the problems identified in our experiments. We evaluate these changes in simulation, showing that they achieve both good fairness and resource allocation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
KBNets '17, August 21, 2017, Los Angeles, CA, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5053-2/17/08…$15.00
https://doi.org/10.1145/3098583.3098589

2 BACKGROUND AND APPROACH

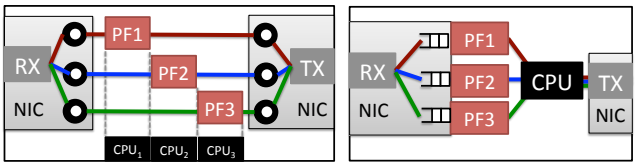
A wide body of existing research has focused on achieving high packet processing performance on commodity hardware, typically by bypassing the kernel: packets move from the NIC directly to the process and back out on the NIC [7, 12] as shown in Fig. 1.a. This is accomplished by mapping the send and receive packet rings in application memory; the kernel is only used as a control plane to set-up the direct datapath between apps and the NIC. To check for available packets apps either rely on polling (with DPDK [7]) or use syscalls (netmap [12]) which effectively signal NIC interrupts to apps. By default, a single process gets exclusive access to the whole NIC, but it is also possible to have many applications share the NIC by using hardware multiplexing: modern NICs have tens/hundreds of send/receive queue pairs, each of which can be exclusively assigned to a different process. Incoming traffic can be directed to the various queues by using hashing (called Receive Side Scaling, or RSS), or via explicit filters (e.g. Flow Director in Intel NICs). In virtualized environments, kernel bypass is used in conjunction with hypervisor bypass: the NIC rings are mapped directly in the guests' memory by using hardware virtualization (called SR-IOV).

Existing work has shown that it is possible to process packets for multiple 10Gbps NICs on commodity hardware for packet forwarding [5, 12], load balancing [3, 9], NATs [10], caches [8], etc. However, all these works consider processing in isolation, on a dedicated core or a machine. The main assumption is that the same processing is applied to all traffic coming from an interface and, in this context, existing solutions are up to the job.

Almost all current work assumes CPU resources are reserved for network processing, and wide-area provisioning even reserves bandwidth between the different processing sites. Network processing, however, must be able to cope with smaller traffic volumes belonging to multiple tenants. Consider a mobile edge cloud deployment, close to a cellular base station: every mobile user or even application could have a personalized firewall and proxy wanting to run on the edge cloud. As the number of users is on the order of a few thousand per cell, how should one provision the computing resources at the mobile edge? Today's approach would be to reserve processing for every single user's firewall, but that is obviously wasteful, since most users are silent most of the time and have intermittent traffic bursts where the peak speed could be in the tens or hundreds of Mbps.

Packet-switching, coupled with endpoint congestion control, is a much better way to share Internet links. Packet switching enables resource pooling [13]: the total capacity on a path can be used by any flow traversing that path, and endpoint congestion control adapts the rates to network conditions, ensuring that flows get their fair share when they cross bottleneck links.

Would it be possible to resource pool CPUs for flows that require network processing in the same way that we pool bandwidth? To enable resource pooling, we can simply schedule multiple processing functions on the same CPU core, and have the flows share the CPU in the same way they share bandwidth: on a best effort basis. This simple idea is shown in Figure 1.b: the three processes are scheduled on the same CPU, each being given a receive ring by the NIC and taking turns at processing packets. When a process is not running, its packets will be buffered in the NIC ring buffers; once



a) Reserving cores for each function b) Pooling network processing

Figure 1: Network processing on commodity hardware: moving from reserved cores to resource pooling.

the process is scheduled, it will process the packets from its ring until either its scheduling quantum expires or there are no more packets to process. If the process cannot keep up with the incoming packet rate, the ring buffers will fill up, leading to packet drops.

Running multiple processing functions on the same CPU is easy enough to do, and it will enable memory isolation between the processing functions via the OS or hypervisor mechanisms. The interesting remaining question is whether it will enable the efficient type of bandwidth resource pooling prevalent in the Internet today. Network links have the following properties that enable efficient sharing via endpoint congestion control:

- Constant throughput: regardless of the number of flows, a (wired) link's total throughput is constant.
- Proportional end-to-end congestion notification - all flows sharing a bottleneck link perceive the same average loss rate and this enables endpoint congestion control to ensure high utilization and fairness at the same time.

If the properties hold for CPU sharing, then sharing will be as simple as Fig. 1. In the next section, we set out to discover the answer experimentally.

3 PROCESSOR POOLING IN NETWORKS

In our experiments we rely on FastClick [1], a version of the Click modular router [6]. Click is a well known packet processing framework that has hundreds of pre-compiled elements that can be combined in directed packet-processing graphs called *configurations* to implement a wide range of network processing functionality [6]. FastClick [1] solves the performance problems of Click and enables simple, user-mode deployment: it runs on top of netmap [12], automatically assigns queue-pairs to Click processes, sets CPU affinity and has batching support to improve performance.

We use a simple Click configuration that simply forwards packets from input to output and changes the MAC addresses. We run multiple copies of this configuration as separate FastClick processes that are assigned to run on the same CPU core, and each process is assigned a separate NIC queue-pair. To simulate more complex processing functionality, we also add an element to the forwarding configuration that executes a fixed number of cycles per packet. We generate packets from a sender using the `pkt-gen` tool from netmap, ensuring that load is split equally among the Click forwarding processes. We measure per-process and total throughput.

3.1 Throughput

In all our experiments, we generate 128B packets at 10Gbps (8.22 million packets per second) and forward them using a variable

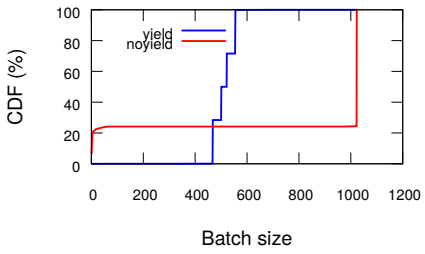


Figure 2: Batch sizes used by Click (5 processes per core, 1024 packets per receive ring).

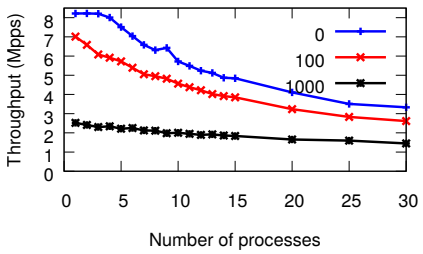


Figure 3: Total throughput vs. # Click processes per core (plain forwarding, batch size = 1024 packets)

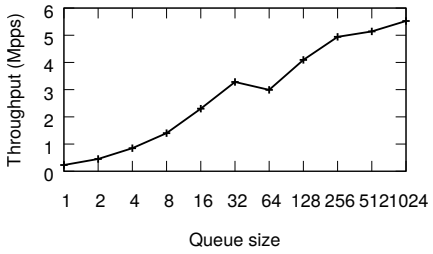


Figure 4: Total throughput as we vary the batch size (plain forwarding, ten processes)

number of processes, all sharing the same CPU core. We use fairly large buffer rings (1024 packets) per process, and similar FastClick batches. In our initial experiments, we found that FastClick is able to sustain line rate for 128B packets when a single process is used, but the throughput dropped quickly when we added more processes: with 5 processes, total throughput was a mere 5.1Mpps. We expected a throughput reduction due to context switching overheads, but this was rather extreme.

To reduce the effects of context switching, we first used the round-robin scheduler and modified the scheduling quantum, trying values of 1ms, 10ms and 100ms. Surprisingly, performance was the same regardless of the quantum, and a closer look at scheduler statistics shows that the number of context switches was more or less the same for all quanta, instead of being inversely correlated. This implies that our processes were never preempted by the OS scheduler. To understand why, we show the main loop executed by each Click process below:

```
while(1){
   poll(...);//read packet batch from NIC
   process_batch(...);//process packets
   ioctl(TXSYNC,...);//send packets out
}
```

A process will be scheduled when `poll` has packets ready; at that point the process will read and process all the packets available in its receive ring. The `process_batch` call can handle at least 8200 packets per ms (as shown by the single process results), and we expect that a batch of 1024 packets will take around 100μs; after this, the packets are placed on the (most likely empty) send ring and the OS is notified.

Next, the process will invoke `poll` again; this call is unlikely to block since in the meantime some more packets have arrived in the process's ring; these packets will be further processed, another smaller batch retrieved, and so on, until there are no packets in the receive ring. At this point the process will block. This effect is clearly shown in Figure 2 where a CDF of actual batch sizes is shown for an experiment with 5 processes sharing the same core: more than 90% of batches contain ten or fewer packets, while the remaining ones have 1000 packets or more.

A simple fix is to have each process yield the processor after it processes the first batch. We implemented it by adding a `sched_yield` call before `poll` and reran the experiment in Figure 2. The batch size distribution is a Gaussian distribution centred around 500 packets. The throughput achieved using `yield` is around 7.5 Mpps, while the throughput without using `yield` is 5.1 Mpps.

With yield, we reran the throughput experiments, varying the number of processes used and the cost per packet. The results are shown in Figure 3, for the case where processing is plain forwarding (line 0), or when an additional number of cycles is spent per packet (100 or 1000). With plain forwarding, when we have 30 processes the total throughput is 3.5Mpps (or 4.3Gbps), a 60% reduction. This is because a large number of processes fighting for the cache means there is little or no data in the cache when a process is re-scheduled. Note that the maximum decrease is reached at 25 contending processes, and thus adding further processes does not further decrease total throughput.

The overheads strongly depend on how costly the packets are to process. As packets become more expensive, it takes more time to process a batch of packets, reducing the number of context switches and their relative overheads. When 1000 cycles are spent per packet, the overhead of running with 30 processes instead of a single one is 40% (from 2.5Mpps to 1.5Mpps). Even higher per-packet costs further reduce overheads: with 10000 cycles per packet, throughput drops by 16% from 288Kpps (1 process) to 239Kpps (30 processes).

Effect of buffer sizing. Batching is one of the main techniques used to amortize the costs of system calls: the netmap paper [12] shows that batches of at least 10 packets per syscall are needed to amortize syscall overheads for the packet generator. In our case, we control batches by setting the appropriate number of packets in the NIC rings and FastClick. To understand the effect of buffer sizing, in our next experiment we ran plain forwarding (no extra cycles per packet) with different sized ring buffers, varying the number of FastClick processes as above. The results show that higher queue sizes and, implicitly, FastClick batches result in better total throughput: using ten processes and 128 packet buffers results in a throughput of 3.7Mpps; with 1024 buffers the throughput is 5.5Mpps, a 50% improvement.

The results so far show that CPU sharing is fairly efficient for network processing: while total throughput is reduced by 50% in the worst case we have tested, in practice we expect the reduction to be around 10%-20%. Even with a 50% drop in throughput, the potential savings due to pooling are massive: we only need two CPU cores to guarantee 10Gbps forwarding performance for 30 tenants; with reservations we would need 30 cores.

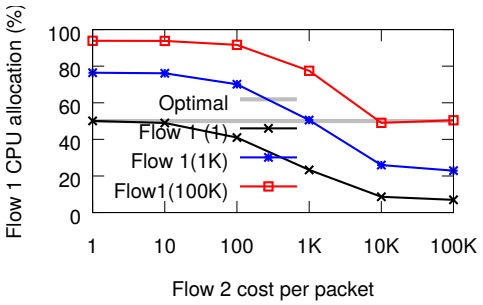


Figure 5: Throughput distribution for two competing processes processing flows with different per-packet costs.

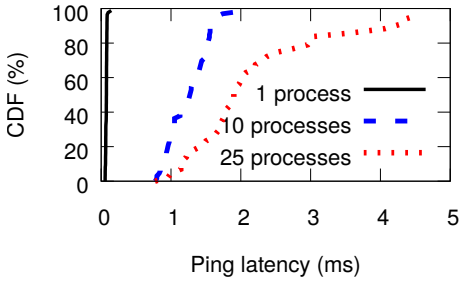


Figure 6: Ping latency measurements with different numbers of processes contending for the CPU.

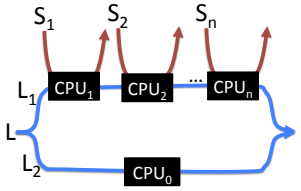


Figure 7: Line topology: a multipath flow competes with many short flows on one of its paths, and is alone on the other path.

3.2 Sharing behaviour

Our experiments so far use non-responsive flows where the packet rates are exactly the same. These flows are then mapped to different receive rings by the NIC and processed by separate processes, where the per-packet cost could vary between processes. If the flows were sharing a normal link (whether drop-tail or RED-based) instead of a CPU, they would experience the same loss rate and achieve similar rates. We experimentally measured how the CPU is shared in practice. Our experiments, shown in Fig. 5, highlight three CPU sharing outcomes that depend on how costly the per-packet processing is. All of these outcomes are ill-suited for resource pooling.

Packet-level fairness for cheap packets. When context switches are triggered by blocking on I/O rather than the OS scheduler, the emerging behaviour is that of *packet-level fairness* among the different processes contending for the same core; this is the case for all experiments discussed previously. This is intuitive, since each process will run through a similarly sized batch (1024 packets or less) and then lose the CPU.

Packet-rate fairness does not imply fair sharing of the CPU. When the processes spend different number of cycles per packet, the CPU allocation is unfair: for instance, a process that forwards 1000 cycle packets competes with another process that does simple forwarding, each process forwards 1.25Mpps and the expensive process gets around 80% of CPU. This is clearly shown in Figure 5 when there is a difference between the costs of the different flows.

CPU fairness for expensive packets. When the time taken to process a batch is more than 1ms (the scheduler quantum), the sharing is dictated by the OS scheduler and each process gets roughly the same share of the CPU. In Figure 5, note how a flow spending 10K cycles per packet gets the same CPU time as one spending 100K cycles per packet.

Unfair allocation for mixed scenarios. When some processes exceed their quantum and some don't, the CPU is given preferentially to the more expensive processes. For instance, when a plain forwarding process competes with one that fully uses its quantum, the forwarding will only get 10% of the CPU; the fair outcome would be 50% of the CPU.

In summary, the OS scheduler does a poor job of sharing the CPU among network-processing tasks.

3.3 Latency

The next question is to what extent resource pooling the CPU increases packet-level latency. The scenario we examine is when we have P CPU-hungry network processes (such as the ones we have examined in our experiments until now) and one idle process that must simply forward ping traffic. In the worst case where each process uses its 1ms scheduling quantum, the ping latency could be inflated by upto P ms—is this what happens in practice?

We ran one process that forwards only ICMP ping packets, measuring the end-to-end latency as reported by ping in three experiments: when the ping forwarder runs alone, and when it shares the core with other 10 or 25 forwarding processes (as above). The results, shown in Figure 6, show that ping latency is around 120 μs on an idle core. When competing with 10 forwarding processes the median latency jumps to 1.3ms, and the tail to 2ms. With 25 processes, the median is 1.8ms and tail is 4.5ms. This is significantly better than the worst case 25ms, but still significant. To reduce latencies further, we could run smaller queues, which in turn would hurt total throughput. Within a single machine, however, it should be possible to schedule lighter processes on one core and busy processes on other cores, and using differently sized queues for the two categories.

3.4 Multiple bottlenecks

Real networks are much more complex than the dumbbell topology we've used so far (with the CPU as the bottleneck). In such networks, per-resource fair-sharing can lead to inefficient use of network capacity; this is what we want to explore in this section. We resort to simulation in the high-speed *htsim* simulator from [11]. An interesting yet simple topology to study is the one in Figure 7, where a multipath flow called L has two subflows taking different paths. The first subflow, L_1, traverses n bottlenecks (where n is configurable), sharing each bottleneck with a single other short flow (S_i). The second subflow, L_2, traverses a single bottleneck that it uses exclusively.

For simplicity, we assume all flows have packets that are equally costly for each bottleneck, and that one CPU can process a flow at 100Mbps. We are interested in how throughput is distributed among the short flows and the multipath flow, and what the total network throughput is. To emulate OS scheduling, we have implemented in

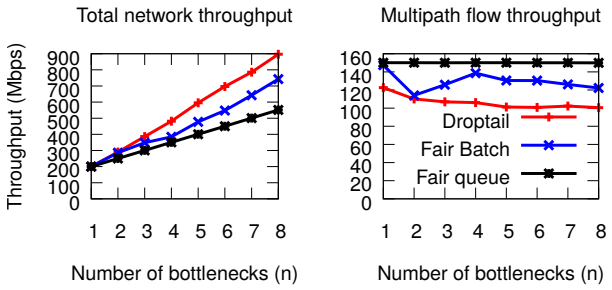


Figure 8: Fair queueing hurts total network throughput and overall fairness.

htsim a fair queueing mechanism that can process a configurable number of packets per batch; when we set the maximum batch size to one, we have a perfect fair queue implementation; when we set it to 1024 we mimick the behaviour seen in our experiments above.

We vary n, the number of CPUs and their associated short flows, and measure total throughput obtained by the Multipath TCP long flow, as well as the throughput of the short flows. Multipath TCP pushes traffic away from congested links: in our example, a perfect MPTCP congestion controller will measure a strictly higher loss rate on its L_1 subflow, and will therefore move traffic away from the top link and fill the bottom link instead, regardless of n [14].

When each CPU is simulated using a standard drop-tail link, the results shown in Figure 8 confirm our expectations (with small deviations from theory when n is small): the multipath flow pushes most of its traffic on the bottom path, and total network throughput is within 2% of the theoretical optimal. However, when we simulate the CPU as a simple fair-queueing link, subflow L_1 gets 50Mbps regardless of n, reducing the speed of the n short flows to $50/n$ Mbps each and driving total network throughput to half the optimal. In other words, giving 150Mbps to the MPTCP connection is done by wasting 50Mbps · n capacity; the larger n is, the larger the wasted throughput. This happens because the MPTCP subflow will only get signals from the first fair queue (when it overflows), and not the rest. Finally, we ran an experiment where we use batching in the fair queue, mimicking OS packet processing. The outcome is slightly better than per-packet fair queueing, but the top subflow still gets around 25Mbps regardless of n, with the appropriate drop in total network throughput.

To summarize, fair queueing is inadequate for wide area network sharing because it can severely decrease total network throughput; drop-tail behaviour is desirable for this particular topology.

4 TOWARDS A SOLUTION

We now explore potential solutions to enable appropriate sharing of CPUs. A good solution must behave like a droptail queue, sending congestion signals proportionally to all participating flows, but it must send more signals to more expensive flows to enable fair sharing of the bottleneck resource.

When sharing a bottleneck link, all TCP senders will converge to the same CWND as they measure the same loss rate: on average, $CWND = \sqrt{\frac{2}{p}}$, where p is the loss rate. Consider now two flows F_1 and F_2 sharing a bottleneck link and having the same round trip time, and F_2's packets are twice the size of F_1's. The two flows will get the same congestion signals and will have the same average congestion window, giving F_2 twice the throughput of F_1.

To enable fair-sharing of the bottleneck link, F_2's window must be half of F_1's, so F_2 must see a loss rate four times higher than F_2. A variant of RED called RED_4 does exactly this [4]: it sends more congestion signals to flows that have higher packet sizes. RED_4 computes a mark rate p for its cheapest flow based on the buffer usage, and marks other flows packets with probability $\frac{MSS_i^2}{MSS_{min}^2} \cdot p$.

We can use RED_4 in our case with a slight change: instead of using the packet size in the formula above, we use the average time it takes to process a packet instead. We have implemented RED_4 in *htsim* and measured how much throughput F_1 achieves as a function of F_2's packet size, showing results in Fig. 9. First, notice how per-packet droptail queues allocate the bottleneck unfairly; our experiments in the previous section have shown that fair queueing behaves equally bad, giving the expensive flow 2 a larger share. RED_4, on the other hand, behaves really well, being within 10% of the optimal allocation. Finally, notice that F_1 starts to get slightly more capacity when F_2 is more costly: this is because our simulator does not implement ECN, and our congestion signals are packet drops. When F_2 has really large packets, it starts experiencing high loss rates that trigger timeouts, further reducing its rate.

The sharing behaviour of RED_4 is really encouraging; RED_4 also behaves correctly in the line topology from Figure 8. However, RED_4 assumes a single queue is shared by all flows, and retrofitting this architecture to the OS would be very inefficient.

Instead, we propose RED_4 MQ, a variation of RED_4 that uses multiple per-flow queues instead of a single queue, and that can be easily implemented on top of the architecture in Fig. 1. RED_4 MQ applies the same RED_4 algorithm on packet enqueue, but uses the sum of all queue sizes to decide the drop probability instead of the size of the single queue in the original algorithm. We have implemented it in *htsim* and use a default batch size of 1024, as used in our testbed experiments in section 3.

Fig. 9 shows how RED_4 MQ shares a bottleneck among two competing flows with different per-packet costs: RED_4 MQ performs very well, similar to the single queue RED_4 algorithm. In the line topology (see Fig. 8), RED_4 MQ performs within 10% of droptail queues, which is nearly optimal.

Finally, to understand why RED_4 MQ behaves so well, we studied its sensitivity to the batch size in the fairness and line topologies. Figure 11 shows the performance of RED_4 MQ compared to optimal as a function of batch size. The results show that batching is crucial to achieving the correct behaviour; intuitively, when we have small batches, the emergent behaviour of the algorithm is dominated by fair-queueing, as all flows will be serviced round-robin and slowly converge to the same window. When batching is used, on the other hand, the RTT in the line topology increases significantly for expensive flows, reducing their effective rate, and burst losses are much more likely when n is large.

Practical considerations. Implementing RED_4 MQ is our future work: it requires some minor modifications to netmap (kernel code)

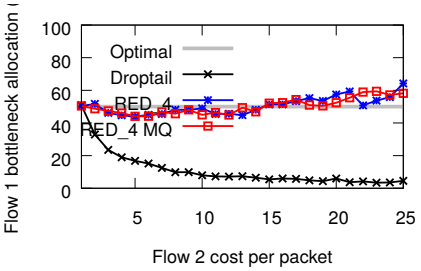


Figure 9: Throughput distribution among two competing flows with different packet sizes.

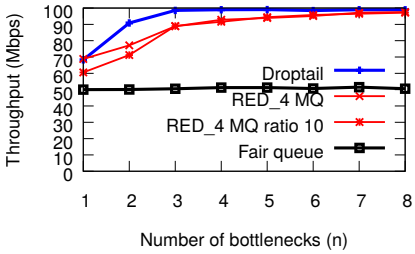


Figure 10: RED_4 MQ performs close to optimal in the line topology, while fair-queueing hurts total throughput.

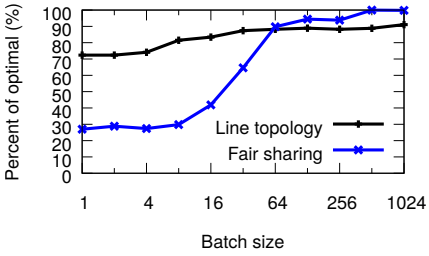


Figure 11: Effect of batch size on the performance of RED_4 MQ: larger batches give better performance.

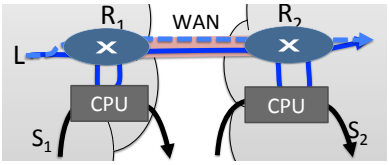


Figure 12: To compress or not? Using CPU pooling and Multipath TCP to dynamically select the most efficient way to transport data between two datacenters

to measure the total number of packets in all receive rings and average packet processing times per process. Understanding in more depth the emergent behaviour of RED_4 MQ requires careful theoretical analysis (e.g. using fluid models). Using these insights, we intend to look beyond RED_4 MQ for alternatives that do not rely on RED, an AQM notoriously difficult to configure correctly.

Our work has so far assumed that traffic passes through a middlebox (e.g. a firewall); however certain middleboxes terminate TCP connections (e.g. a proxy). In such cases, when the proxy is overloaded TCP flow control will throttle the senders. Sharing CPUs correctly in this case requires further thought.

5 IMPLICATIONS

Our work shows it should be possible for multiple network processing functions to share a CPU efficiently. Furthermore, our RED_4 MQ active queue mechanism provides appropriate congestion signals to the endpoints such that CPUs can be shared via endpoint congestion control, just like networks. CPU resource allocation becomes now a part of network routing: the network decides the links and the CPUs a certain flow should take, and the flow modifies its rate according to the congestion signals it receives. This will enable operators to load balance traffic across its network links and network processing load across its processors.

The benefits of CPU processing are however much higher than pooling bandwidth or processing in isolation: they also allow cross resource pooling, i.e. trading off between bandwidth and CPU utilization. As an example, consider the task of sending web crawl data between the two datacenters shown in Figure 12: one can send the data as is on the busy WAN link, or compress the data at the source, transfer and decompress it at the destination datacenter. It is almost impossible to decide ahead of time what the best strategy is: if the CPUs are idle and the data is compressible (e.g. text), compression is obviously superior; if the CPUs are busy or the data compresses poorly, it is better to send directly without compressing. Ideally, one should dynamically load balance across the two paths as done by the Multipath TCP flow L: one subflow sends directly over the WAN link, while the other crosses via the CPUs to compress and decompress the data. The loss rate on the two subflows will tell the Multipath TCP controller how to send most traffic: when the CPU is congested, the loss rate on the bottom subflow will be higher and most traffic will move to the top subflow if the WAN link is uncongested; conversely, when the WAN link is congested, more traffic will be sent via the bottom subflow and thus be compressed.

ACKNOWLEDGEMENTS

This work was performed within the Superfluidity project (project no. 671566) funded by the European Commission under its H2020 framework.

REFERENCES

[1] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In *ANCS 2015*.
[2] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz. Near optimal placement of virtual network functions. In *INFOCOM 2015*.
[3] Daniel E. Eisenbud et al. Maglev: A fast and reliable software network load balancer. In *NSDI 2016*.
[4] S. de Cnodder, O. Elloumi, and K. Pauwels. Red behavior with different packet sizes. In *ISCC 2000*.
[5] M. Dobrescu, N. Egi, K. Argyraki, B. G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *SOSP*, 2009.
[6] E. Kohler et al. The Click modular router. *ACM Trans. Computer Systems*, 18(1), 2000.
[7] Intel Corporation. DPDK: Data Plane Development Kit. http://dpdk.org/.
[8] S. Kuenzer, A. Ivanov, F. Manco, J. Mendes, Y. Volchkov, F. Schmidt, K. Yasukata, M. Honda, and F. Huici. Unikernels Everywhere: The Case for Elastic CDNs. In *VEE 2017*.
[9] V. Olteanu and C. Raiciu. Datacenter scale load balancing for multipath transport. In *HotMiddlebox 2016*.
[10] V. A. Olteanu, F. Huici, and C. Raiciu. Lost in network address translation: Lessons from scaling the world's simplest middlebox. In *HotMiddlebox 2015*.
[11] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath tcp. In *SIGCOMM 2011*.
[12] L. Rizzo. netmap: A novel framework for fast packet i/o. In *Proc. USENIX Annual Technical Conference*, 2012.
[13] D. Wischik, M. Handley, and M. B. Braun. The resource pooling principle. *SIGCOMM CCR October 2008*.
[14] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *NSDI 2011*.