A Switch-Based Approach to Throughput Collapse and Starvation in Data Centers

Alexander Shpiner*, Isaac Keslassy

Department of Electrical Engineering, Technion - Israel Institute of Technology, Haifa 32000, Israel

Gabi Bracha, Eyal Dagan, Ofer Iny, Eyal Soha

Broadcom, Yakum 60972, Israel.

Abstract

Data center switches need to satisfy stringent low-delay and high-capacity requirements. To do so, they rely on small switch buffers. However, in case of congestion, data center switches may suffer from throughput collapse for short TCP flows as well as temporary starvation for long TCP flows.

In this paper, we introduce a lightweight hash-based algorithm called HCF (Hashed Credits Fair) to solve these problems at the switch level while being transparent to the end users. We show that it can be readily implemented in data center switches with $O(1)$ complexity and negligible overhead. We illustrate using simulations how HCF mitigates the throughput collapse of short flows. We also show how HCF reduces unfairness and starvation for long-lived TCP flows as well as for short TCP flows, yet maximizes the utilization on the congested link. Last, HCF also prevents packet reordering.

Key words: TCP, Data centers, Switching.

1. Introduction

1.1. Motivation

The recent emergence of the *data center switch market* has required switch vendors to answer new stringent requirements and rethink their switch architectures. This is because switch vendors of both Ethernet switches and Internet routers want to enter the data center switch market, ideally by using variants of their next-generation switch architecture. However, while Ethernet switches typically require extremely low delay with reasonable capacity, and Internet backbone routers require extremely high capacity with reasonable delay, data center switches require *both* extremely low delay and extremely high capacity. As a result, the data center switch stringent requirements on delay and capacity are now becoming generalized requirements on most high-end next-generation switch designs as well.

Data center switch vendors need to address two crucial problems that result from the low-delay requirements. First, they need to address the *TCP incast problem, i.e., the throughput collapse of short flows* [1, 2, 3, 4, 5]. The throughput of TCP-based applications drastically reduces when multiple senders communicate with a single receiver in high-capacity low-delay networks. Fast and highly bursty data transmissions overfill the switch buffers, causing intense packet loss that leads to TCP timeouts as modeled in [6]. These timeouts last hundreds of milliseconds on a network whose round-trip-time (RTT) is hundreds of microseconds, i.e. three orders of magnitude lower. Protocols that have some form of synchronization requirement, such as filesystem reads and writes or highly parallel data-intensive queries found in large memory-cached clusters, keep waiting for timed-out connections to finish before issuing new requests. These timeouts and the resulting delay can reduce application throughput by up to 90% [2, 5].

The second crucial problem is the *starvation of long TCP flows*. As we will further show in this paper, during congestion, long TCP flows can be temporarily starved during *tens of seconds*, even though the total switch throughput stays high. These high delays are unacceptable for latency-sensitive data center applications. For instance, an algorithmic trading application needs a guarantee that its information will most probably not be delayed beyond a millisecond, or even a few microseconds. Small delays also play a crucial factor in switch benchmarks. For instance, a recent benchmark study favored two switches over a third because they essentially achieved lower delays for large Ethernet frames, reaching 750 ns vs. 3.4 μs [7].

The *throughput collapse of short flows* and the *starvation of long flows* actually have shared reasons, stemming from the high-capacity and low-delay requirements of data center switches. The high-capacity requirement, needed to deal with a large number of flows, would require significant buffer sizes to prevent a high loss rate for the TCP flows, as in Internet high-

*Corresponding author (phone: (972) 4-829 5738, fax: (972) 4-829 5757).
The conference version of this paper received the *best paper award* at IWQoS'10, Beijing, China, June 2010. This work was partly supported by European Research Council Starting Grant No. 210389 and by the Hasso Plattner Institute Research School.
Email addresses: shalex@tx.technion.ac.il (Alexander Shpiner), isaac@ee.technion.ac.il (Isaac Keslassy), gbracha@broadcom.com (Gabi Bracha), eyald@broadcom.com (Eyal Dagan), oferi@broadcom.com (Ofer Iny), eyals@broadcom.com (Eyal Soha)

speed links [8, 9, 10, 11]. However, the low-delay requirement imposes the use of small switch buffer sizes, and as a result typically incurs a large loss rate and many timeouts for TCP flows in case of congestion. These timeouts cause *unfairness* among flows, resulting in high delay variability.

This paper is about reducing the short-flow throughput collapse and long-flow starvation by reducing the delay unfairness among flows. To do so, we want to use a *lightweight switch-based mechanism* that does not significantly impact the switch architecture or require any meaningful additional buffering. In contrast to previous papers on TCP incast [1, 2, 3, 4, 5], we assume that it is forbidden to change the TCP protocol implementations both at the sources and destinations. Thus, the switch-based mechanism should be transparent to the end hosts.

1.2. Contributions

In this paper, we introduce a simple switch-based algorithm to reduce the TCP unfairness in data centers. To our knowledge, this paper is unique in that it is the first to point out the fundamental starvation properties of long TCP flows in data center networks, even when there is no throughput collapse. It is also the first to suggest a specific switch-based mechanism to address this starvation.

We propose to use the HCF (Hashed Credits Fair) algorithm, a novel lightweight algorithm for data center switches that is transparent to TCP-based applications. HCF relies on hashing to aggregate flows into bins, and therefore needs to maintain only a limited number of bins credits using a few bits of information. In addition, HCF regularly updates the hashing functions to prevent persistent hash collisions between the same flows. Last, HCF combines the credit hashing mechanism with a queueing mechanism that provides a higher priority to flow aggregates that have not been recently served.

We demonstrate that HCF prevents reordering by using a special updating mechanism. We also prove that HCF has an $O(1)$ time complexity, and that it requires significantly less resources than competing fairness algorithms. In addition, we explain why HCF does not require any change at the end stations, and in particular does not require any change to the standard TCP protocol.

Later, through simulations, we show the short-flow throughput collapse in the TCP incast problem. We also point out the long-flow starvation problem. We show how HCF can help solve these problems by providing an increased fairness among flows.

The rest of the work is organized as follows. We first survey the related work in Section 2. Then we present our proposed algorithm in Section 3. Next, we show alternative implementation parameters in Section 4. Last, we present simulation results in Section 5, before concluding.

2. Related Work

Preventing unfairness among TCP flows is a well-known problem, and therefore we describe below several solutions from the literature. We also explain why they are not adapted to data center networks.

2.1. TCP Incast Solutions

To solve the TCP incast problem, former papers typically suggest changing the TCP protocol in data centers, e.g., by reducing the value of the minimal retransmission time-out (RTO) from 200 ms to 1 ms or less [1, 2, 5]. Additional suggested changes affect the application itself, such as increasing the request sizes, limiting the number of servers, throttling data transfers, using global scheduling , or controlling the sending rate from the receiver side [12, 3]. However, all these solutions require changing the protocols at the end users, and therefore cannot be implemented at the switch level in a transparent way. These solutions can be combined with our switch-level algorithm to achieve even better performance.

2.2. Increasing the Buffer Size

While the rule-of-thumb for buffer sizes in the Internet is typically *too large* [11], in data center networks it is actually *too small*. A possible solution is to increase the buffer size. This is suggested in [8], which mentions the unfairness problem of TCP flows in oversubscribed links and proposes to increase the buffer size proportionally to the total number of active TCP flows as a solution. This would of course require off-chip memory, and therefore can cause a major hardware change with significant issues of power consumption, chip in/out pin SERDES implementation, buffer area and cost, as well as a high buffer latency.

2.3. ACK Buffering

An alternative solution that would require less buffering would be to *buffer only ACK packets* instead of buffering all packets. In other words, the buffer mechanism would be changed to sort between ACKs and data packets, and send ACKs to a special larger buffer. Since ACKs are significantly smaller than data packets, the solution would require much less buffering than the previous solution while keeping the same total RTT. Again, within a switch, this typically requires off-chip memory, and therefore causes a major hardware change with the same issues as mentioned in the previous solution. In addition, note that these large ACK buffers, as well as alternative ACK-based solutions [13, 14, 15], would be hard to implement when some of the TCP connections are duplex TCP connections, in which the ACK returns within the reverse data packet.They might also not fit when the reverse path is not necessarily the same as the forward path, and therefore does not necessarily meet the same points of congestion.

2.4. Queue Control and Fairness Algorithms

There are many fairness and AQM algorithms in the literature to provide increased fairness among flows, such as WFQ (Weighted Fair Queueing) [16] or DRR (Deficit Round Robin) [17], GPS (Generalized Processor Sharing) [18], SFQ (Stochastic Fair Queueing) [19], RED (Random Early Detection) [20], FRED [21], BLUE [22] or SFB (Stochastic Fair Blue) [23].

However, most of the fairness algorithms rely on buffer sizes that typically need to grow with the number of active flows.

To prevent such detrimental scaling effects, some of the fairness algorithms use hashing to aggregate flows, thus providing an approximate fairness, and most importantly, reducing complexity. However, while they fit Internet requirements, these algorithms do not work well with the shallow buffers in data centers, because in order to provide fairness, they need to divide these small buffers among the many flow aggregates. Thus, they can barely keep some buffering for each flow aggregate. Moreover, they cannot deal with bursts from a same flow, since the buffer per aggregate is not large enough. In addition, to avoid reordering, the hash functions often cannot change, thus incurring permanent hash conflicts between flows.

More particularly, algorithms that are based on per-flow queueing, like GPS, FQ and WFQ, typically need to adapt the buffer size to the number of active flows, and therefore can hardly scale given a small buffer size. Algorithms that react according to the queue length, like RED, are not suitable to the highly-congested and relatively shallow buffers, where the queue length changes aggressively between full and empty. Algorithms that are based on link utilization and packet drops, like BLUE, do not fit to the datacenter networks, because the link utilization also changes aggressively and packet loss can result in high cost due to the long retransmission timer. It is also possible to consider a combination of SFQ with DRR to achieve a potential solution, due to the achieved scalability of the hash function of SFQ and the fairness mechanism in DRR. However, due to the shallow buffers and low round-trip-times in the datacenter networks, its fairness mechanism cannot prevent a small number of flows from dominating the buffer, as later shown in simulations.

Therefore, our objective will be to find an even simpler algorithm that does not need large buffers in order to work properly, does not incur reordering, and does not cause permanent hash conflicts.

2.5. Limiting the congestion window using ECN

Explicit congestion notification (ECN) [24] allows end-to-end notification of network congestion without dropping packets. Constantly setting the ECN congestion indication at the switch level could force flows to restrict the congestion window size to 1. However, such a solution would force endpoints to use ECN. In addition, solutions based on ECN indications, as presented in [25], do not solve the Incast problem in highly congested networks with relatively small buffers, where the average congestion window of a flow is close to 1. Continual ECN indications with a shallow buffer force the TCP sender to reset the retransmit timer (RTO) if the sending rate must be decreased below the congestion window of 1 MSS [24].

2.6. TCP Proxy

Another possible solution is to maintain a TCP proxy at each switch [26]. In this solution, each switch maintains all TCP connections and manages them on behalf of the end users. For instance, a switch between a client and a sever would tell the client that it is the server, and concurrently tell the server that it is the client, thus managing two TCP connections at once. It is then easier for the proxy to maintain a list of allowed flows with their respective credits, and manage all the flows going through it. However, such a solution obviously requires a prohibitive complexity and a full change of the switch hardware, and is therefore not appropriate for data centers.

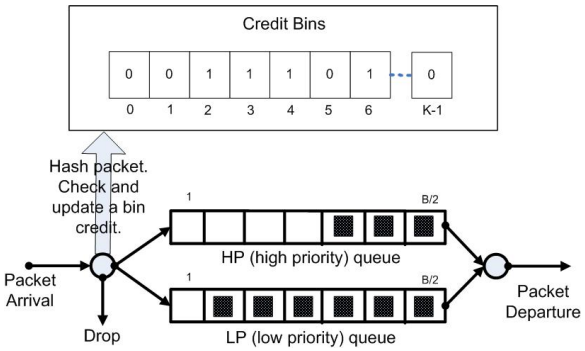


Figure 1: Architecture of the HCF algorithm.

3. Hashed Credits Fair (HCF) Algorithm

In this section we introduce the Hashed Credits Fair (HCF) algorithm. The algorithm is implemented in the intermediate switches, and its objective is to provide increased fairness among flows using a lightweight implementation.

3.1. HCF Overview

The HCF algorithm sorts arriving packets into two queues: the High-Priority (HP) and the Low-Priority (LP) queues. The HP queue receives packets of flows that *have recently been under-served,*, while the LP queue receives the other packets. The HP queue is always served first.

HCF needs to know which flows have recently been under-served, while avoiding memory-expensive per-flow crediting mechanisms. To do so, flows are hashed into bins, and credits are attributed per bin. Through the number of bins, the HCF algorithm trades off memory size and fairness. Moreover, the hash function is often updated to avoid persistent hash collisions between the same flows.

3.2. HCF Algorithm

Figure 1 illustrates the switch architecture for the HCF algorithm. For simplicity, we assume that all packets have the same length.The architecture relies on a buffer of size B and on K credit counters.

The buffer is divided into the HP and LP queues. For instance, assuming that the total buffer size is B, the sizes of the HP and LP queues can be $B/2$ each. Other buffer divisions are possible, as discussed in Section 4, together with additional implementation alternatives.

Time is divided into *priority periods*. The period length is dynamic and determined by the HP queue size: whenever the HP queue becomes empty, the priority period ends. Then, the next priority period lasts until at least one HP packet has been serviced and the HP queue is empty again.

The objective of the priority periods is to reset the credit counters and to keep changing the hash function. Each priority period uses a unique hashing function, possibly chosen among a predetermined set of functions. The inputs of the hash function are the source/destination IP addresses and ports, and its output is the index of the hashed bin among the K possible indices.

The algorithm takes the following steps as packets arrive to the queue.

Initialization — At the start of each priority period,

1. Reset the number of credits $c(k)$ of each bin k, with $0 \leq k \leq K-1$, to $c(k) = c_0$ credits.
2. Pick a new hash function f.

Packet arrival — At the arrival of each packet p,

1. Apply the hash function f on the packet's IP header and obtain the corresponding bin $k = f(p)$ of the packet.
2. Check the number of credits in bin k and the queue sizes. If the bin has credits and the HP queue is not full, packet p joins the HP queue. Otherwise, if the LP queue is not full, p joins the LP queue. If both queues are full, p is simply dropped.
3. If p joins the HP queue, decrement the number of credits in its bin: $c(k) \leftarrow c(k)-1$. (With variable-size packets, credits can be based on the packet length instead. For simplicity, we restrict explanations to fixed-size packets.)
4. If p joins the LP queue, set the number of credits in its bin: $c(k) \leftarrow 0$. [1] (This ensures packet order preservation within a flow.)

Packet departure — When the output line can service the queues,

1. Give priority to the HP queue: if it is not empty, read the head-of-line packet in the HP queue. Else, read from the LP queue.
2. If the HP queue was serviced, check its number of packets. If the HP queue becomes empty, the priority period ends. Re-enter the initialization step.

Queue Swapping — When the HP queue is empties,

1. Swap the header pointers of the HP queue and the LP queue.
2. Continue to Initialization.

Algorithm 1 describes the detailed pseudo-code for the HCF algorithm. It shows that the HCF algorithm holds in a few lines of code, and relies on three functions that respectively implement the initialization of the priority period, the packet arrivals, and the packet departures.

As shown in the simulation results of Section 5, a high packet arrival rate typically causes the LP queue to rarely get empty, thus practically ensuring a near-100% utilization of the bottleneck link. In addition, the credit mechanism significantly reduces the unfairness among flows and the flow starvation.

[1] This condition was not included in the journal version. We would like to thank Ahmad Omary for this helpful remark.

Algorithm 1 Hashed Credits Fair (HCF)

```
init(){
 ∀k : c(k) ← c_0;
 *f() ← createHashFunction(time);
}

arrive(p){
 k ← f(p.srcIP, p.destIP, p.srcPort, p.destPort);
if c(k) > 0 and HP.full = false then
   HP.enqueue(p);
   c(k) ← c(k) − 1;
else if LP.full = false then
   LP.enqueue(p);
   c(k) ← 0;
else
   drop(p);
end if
}

transmit(){
if HP.empty = false then
   HP.dequeue();
   if HP.empty = true then
      queuesSwap();
      init();
   end if
else
   LP.dequeue();
end if
}

queuesSwap(){
temp = HP.head;
HP.head = LP.head;
LP.head = temp;
}
```

3.3. HCF Complexity

The objective of the HCF algorithm is to be very lightweight and easily implementable, so as to fit data center switches without additional hardware requirements.

The above pseudo-code shows that both the packet arrival and the packet departure functions have an $O(1)$ time complexity. Their main functions are queueuing/dequeueing packets, and checking/updating bin credits. Therefore, assuming that the initialization of the credit bins in the registers to fixed predetermined values is $O(1)$, the HCF algorithm runs in an $O(1)$ time complexity.

On the other hand, when the initialization process occurs, the bin credits are initialized, and therefore the theoretical time complexity is $O(K)$. Therefore, the naive worst-case theoretical algorithm complexity is $O(K)$.

However, in practice, the hardware complexity is $O(1)$, in the sense that for a small K, the credit array can be stored in the internal registers, and thus easily initialized in a packet time. This

makes the HCF algorithm readily implementable in hardware.

In addition, the required memory space overhead for the management of the algorithm is the memory of the bin array that holds the amount of credits. Since the maximum credit of each bin is c_0 and there are K bins, the memory needed is $K \cdot \lceil \log(c_0 + 1) \rceil$. For instance, it could easily be implemented in hardware using $K = 32$ counters of 8 bits each, thus only holding 32 bytes. In addition, we need to manage two queues instead of one, and potentially store the timestamp of the priority period, thus adding a negligible overhead. Therefore, the total overhead of the HCF is essentially negligible compared to the memory size needed for the queue (e.g., 32 bytes are negligible relative to an Ethernet packet of 1500 bytes).

In addition, note that the priority period of HCF adjusts dynamically to the traffic through the size of the HP queue. It does not need to be predetermined.

3.4. Queue Swapping to Prevent Packet Reordering

The queue swapping procedure is added to the HCF algorithm in order to prevent packet reordering. For instance, consider two packets p_1 and p_2 from the same flow successively arriving to the switch. It might be that the flow does not have credits left, and therefore p_1 is stored in the LP queue. Then, the HP queue might get empty, thus generating a new priority period with bin credits re-initialized to c_0. If there is no swapping, when p_2 arrives, it might therefore use a credit and be stored in the HP queue, which has higher priority. Thus, p_2 would depart earlier than p_1, and the packets would be reordered.

As described above, a simple solution to this packet reordering problem is to *swap* the HP and LP queues at each new priority period, while the credits are initialized, by redefining the LP queue as the HP queue, and vice versa. In the other words, when each priority period ends, the HP queue is empty. In the next priority period the previous LP queue becomes the next HP queue and the previous HP queue becomes the next LP queue. Hence, all the packets that were left in the previous LP queue at the end of the priority period, will be serviced first at the next priority period. The following theorem demonstrates that this solution ensures that non-dropped packets are not reordered.

Theorem 1. *HCF with queue swapping prevents reordering among packets of a same flow that leave the switch.*

Proof 1. *First, packets that leave the switch have not been dropped, and therefore we can restrict the proof to packets currently in the queues HP and LP.*

Consider the set $\mathcal{S}_F$ of all packets in the queues from a given flow F, and define the following total order relation $\leq$ on any two packets $\{x, y\} \in {\mathcal{S}_F}^2$: $x \leq y$ iff (x and y are in the same queue and x is ahead of y) OR (x is in HP and y is in LP). Then the relation clearly satisfies anti-symmetry, transitivity and totality over $\mathcal{S}_F$.

Assume that two packets p_1 and p_2 of the same flow arrive in that order but leave the queues reordered: we want to show that this is impossible.

Let's first show that $p_1 \leq p_2$. Clearly, if they were reordered in the switch, at some point, both p_1 and p_2 were in the switch queues. There are three possible cases. First, if both p_1 and p_2 are in the same queue, since the HP and LP queues are FIFO, then necessarily p_1 is ahead of p_2 and $p_1 \leq p_2$. Otherwise, if p_1 is in HP and p_2 is in LP, $p_1 \leq p_2$ as well by definition. In the last case, if p_1 is in LP, it means that the flow does not have credits left. If p_2 arrives in the same priority period, it will hash into the same bin with no credits left, and therefore cannot be stored in HP. Else, if it arrives after the priority period of p_1, then p_1 is now in HP by swapping, and as shown above necessarily $p_1 \leq p_2$ again. Therefore, in all cases, $p_1 \leq p_2$.

Let's now show that whenever $p_1 \leq p_2$, p_1 necessarily leaves before p_2. First, whenever first defined in some priority period, the relation $p_1 \leq p_2$ is kept during the entire priority period (as long as no packet has departed): if p_1 is ahead of p_2 in the same queue, it stays ahead, and likewise, if p_1 is in HP and p_2 is in LP, they stay in their respective queues. In addition, by definition, p_2 cannot leave before p_1 during this priority period: either p_1 is ahead of p_2 in the same FIFO queue, or p_1 is in HP and p_2 is in LP, which cannot be serviced before HP gets emptied. Therefore, to be reordered, neither p_1 not p_2 can leave during the first priority period in which the relation $p_1 \leq p_2$ is defined.

Last, after the priority period ends, by definition, the HP queue is empty. Therefore both p_1 and p_2 were necessarily in the LP queue, and p_1 was necessarily ahead of p_2. Since the LP queue (which becomes the HP queue) is FIFO, p_2 cannot leave before p_1, hence there cannot be reordering.

4. Implementation Alternatives

There are several possible implementation alternatives for the HCF algorithm, involving a broad span of tradeoffs between cost and performance. While we detail below several of these alternatives that often introduce additional parameters, we have attempted to reduce the number of parameters needed in the main HCF variant described above, since we only need to define the number of bins K (in addition to the queue size B, which is needed even in droptail). The objective of reducing the number of parameters is to enable both an easier implementation and a range-free scalability of the algorithm.

4.1. Bloom Counter

Instead of using a single hash function, it is possible to combine several hash functions by using a *Bloom counter* equivalent, where the counter is successively decremented from c_0 to 0 [27, 28].

For instance, a possible implementation using Bloom counters would work as follows. Each arriving packet is mapped to several bins. If at least one bin has a remaining credit, the packet is stored in HP, and all of its corresponding positive hashed credits are decremented. Else it is stored in LP. For instance, if a flow is hashed to credit values $\{0, 2, 3\}$, they are updated to $\{0, 1, 2\}$ and the packet is stored in LP.

In particular, when $c_0 = 1$, this implementation reduces to a simple *Bloom filter* (more precisely, it shows the complementary of the Bloom filter bit values). For instance, if a flow is

hashed to credit values $\{0, 1, 1\}$, they are updated to $\{0, 0, 0\}$. Therefore, it is a generalization of the HCF algorithm for any number of hash functions, and reduces to HCF when using a single hash function.

The goal of the Bloom filter is to represent *set membership* while minimizing the false positive error. In our case, it represents whether a flow belongs to the set of flows that have already used a credit, while minimizing the probability that a flow is wrongly tagged as having already used a credit. The ideal number of hash functions κ in such a Bloom filter is provided by

$$\kappa \approx \frac{K}{N} \cdot \log 2,$$

where K is the number of bins and N the number of flows to represent [27, 28]. Since we want a small number of bins, we often use $K \ll N$, and therefore there is no point in using more than one hash function. Simulations with $K = 20$ bins and $N \approx 400$ flows confirmed that Bloom filters with $\kappa \geq 2$ hash functions did not improve the performance of HCF.

4.2. Number of Bins

The number of hashed bins K has a significant influence on the performance of the system. On the one hand, an HCF switch with a single bin is similar to a FIFO-based switch. On the other hand, a large number of bins minimizes hash collisions, and therefore maximizing the fairness among flows. However, a large number of bins also consumes slightly more system resources, and in particular takes more memory and increases the implementation complexity. The simulation results in Section 5.3 provide a performance comparison using different numbers of bins. It appears that $K = 20$ bins are often enough both for fairness and starvation.

4.3. Priority Period

In the presented algorithm we use a dynamic priority period, i.e. the period length dynamically changes depending on the queue occupancy. To simplify the implementation, a *fixed* period length can be used, so that the credits are initialized and a new hash function is determined at predetermined periodic times. However, while a fixed priority period might be simpler to implement and avoid initializing credits too often, it requires tuning the period parameter. In addition, simulations in Section 5.3 show that a fixed priority period has a negative impact on the system performance. Further, the fixed priority period violates the packets ordering, thus, is not recommended for using.

4.4. Buffer Division between HP and LP Queues

We have presented an equal division of the buffer between the HP and LP queues, each of size $B/2$. However, different divisions are possible. A large LP queue will induce a larger link utilization. On the other hand, a large HP queue favors longer priority periods and a better fairness.

Note that when using the queue swapping presented above to prevent reordering, it is natural to use an equal partition as well, since the queue sizes are being swapped at each new priority period.

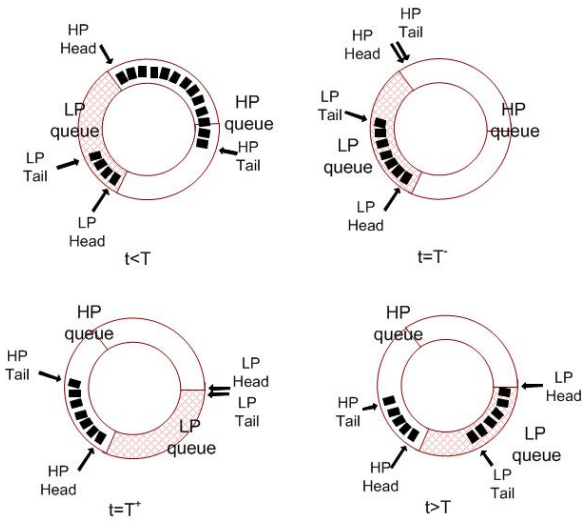


Figure 2: Architecture of buffer division into three thirds.

Furthermore, it is also possible to use a *dynamic division* of the buffer. When a packet arrives and the sum of the queue sizes is less then B, the packet joins the suitable queue. If the buffer is full, i.e. the sum of the queue sizes is equal to B, and the packet is destined to the HP queue, then the last packet from the LP queue is dropped, and the arrived packet is stored in the HP queue. This solution improves the performance, although it involves a more complicated management.

4.5. Buffer Division to Three Thirds with Queues Swapping

A possible issue when using queue swapping is that the HP queue might already be full at the start of the period that follows the queue swapping. Therefore, it might need to drop the newly-arriving high-priority packets. Such an issue occurs when the LP queue was full before the swap, because the LP queue becomes the new HP queue in the swap.

A possible solution to overcome this problem is to keep space for those newly arriving packets. The buffer is divided into three equal parts. At every point of time, two of the three thirds belong to the HP queue and one third belongs to the LP queue. The architecture and the algorithm are shown on Figure 2.

By definition of the dynamic priority period, at the end of the priority period, at some time $t = T^-$, the HP queue is empty. Soon after, at the beginning of the next priority period, at time $t = T^+$, the HP queue uses again two thirds of the queue, but now one of the two thirds is the one that was formerly used by the LP queue. The third remaining part, which is also empty, becomes the new LP queue. Thus, there will be buffer space for new arriving high-priority and the low-priority packets.

The next theorem emphasizes the following result: at each priority period, at least $B/3$ of the high-priority packets and $B/3$ of the low-priority packets are guaranteed to enter to the buffer.

Theorem 2. *Assuming the buffer division above, for any buffer size B (divisible by 3), HCF guarantees buffer space for at least*

B/3 HP packets and B/3 LP packets in the start of each priority period.

Proof 2. *Assume that a priority period ends at time $t = T^-$, and the new priority period begins at time $t = T^+$. According to the definition of the dynamic priority period, the HP queue is empty at time $t = T^-$ and the length $LP(T^-)$ of the LP queue at time $t = T^+$ satisfies $0 \leq |LP(T^-)| \leq B/3$. Thus, in the worst case there are $B/3$ packets in the LP queue.*

When the queue swapping occurs, the LP queue becomes the new HP queue. Thus, the length $HP(T^+)$ of the HP queue at time $t = T^+$ satisfies $0 \leq |HP(T^+)| \leq B/3$ and the LP queue is empty.

The total size of the HP queue is $2B/3$, therefore there is a buffer space for at least $2B/3 - |B(T^+)| \geq 2B/3 - B/3 = B/3$ packets in the HP queue. In addition, the LP queue is empty, and it can contain up to $B/3$ packets, therefore there is a buffer space for $B/3$ packets in the LP queue as well. □

4.6. Hash Function

A hash function is used to map packets to their corresponding bins. There are many possible alternative hash functions that can be used [19]. Since the input size and the output ranges for the hash function are fixed, the hash function can be easily implemented. In addition, in order to implement a different hash function for each priority period, it is possible to use an XOR of the packet header with the local time-stamp of the priority period start. The algorithm is fully distributed, and therefore there are no synchronization issues between different ports.

5. Simulation Results

In this section, we run simulations of a congested link using an NS2 simulator [29]. We assume an N-to-1 topology with a shared congested link, similar to the topology presented in [3], with N sources sending N flows through the congested link to a shared destination. We further assume that the congested link corresponds to an output of an output-queued switch.

We successively simulate the effects of HCF on on *long-flow starvation* and on *short-flow throughput collapse*. We then analyze the impact of changing the HCF parameters, and finally check mixes of short and long flows.

We attempt to provide some intuition for all these simulations. Yet, note that the reasons behind data center fairness problems such as the TCP incast are often hard to model because of the complex interactions involved [1, 2, 5].

5.1. Long-Flow Starvation

5.1.1. Settings and Metrics

In the long-lived flow simulations, we run $N = 400$ long TCP New Reno flows [30] with $RTT = 100\mu s$. We also run UDP packets with average arrival rate of 5% of the switch output link capacity. The capacity C of the switch congested output link is 100 Mbps, with other links running at a much higher capacity so as to have the switch output link form a single bottleneck in the network. The total buffer size B in the switch output is 20 packets, with a uniform packet size L of 1500 bytes. Therefore, in the HCF switch, the buffer is divided equally between the HP and LP queues with 10 packets per queue. The HCF algorithm relies on a dynamic priority period, the flows are hashed in the simulation using a real hash function into $K = 20$ bins, and each bin receives an initial credit of $c_0 = 1$ credit unit. Each simulation is run for 3 minutes, and simulation results are measured during a final *measuring time* of $T = 10$ seconds.

In addition, we compare HCF with the RED-ECN-based and hashed DRR-based switches. We use the ns2 implementations of the RED and DRR queues. For both algorithms, we chose all the default parameters. The parameters for the RED-ECN queue are set up to turn on the ECN bit, and never drop a packet if the queue is not full, with a minimal and a maximal thresholds to zero and to the buffer size, respectively. The reason for this parameter tuning is that in this case we are able to restrict the flows most of the time. As a consequence, we can also reach higher fairness, which is our key objective. The DRR-based queue uses a hash function, similarly to HCF. The DRR queue mechanism hashes the flows into 20 bins using a hash function, in the same way as HCF also uses $K = 20$ bins. Therefore, in all simulations we chose the DRR queue mechanism to hash the flows into the same number of bins as the HCF's hash function. Thus we obtain the same hash function collision probability in both algorithms, enabling us to get closer to an apples-to-apples comparison.

In the simulations we measure two performance indicators that can reflect on the extreme unfairness among flows in the data center network: the *unfairness* and the *starvation percentage*.

In an ideally-fair system, all flows would be able to send the same amount of traffic during the measured time of $T = 10$ seconds i.e. 208.333 packets. This intuitively suggests to define the unfairness based on the deviations between the number of packets sent by each flow and the average across all flows. Therefore, we define the *unfairness* as the variance of the cumulative number of packets sent by each flow during some time T.

A significant unfairness often causes long periods of temporary starvation for flows. We want to characterize such temporary starvation, and will simply define *starvation percentage* as the percentage of all flows that have not sent any packet during the measuring time T.

In addition, we also consider the *throughput* and *utilization* of the bottleneck link. The bottleneck link *throughput* is defined as the rate of all packets transmitted through the bottleneck link during time T, and the bottleneck link *utilization* is defined as the ratio of its throughput by the bottleneck link capacity. The goal is to keep the bottleneck link utilization close to 100%, thus maximizing the throughput. In all simulations, we found that the bottleneck link utilization was extremely close to 100%. The utilization was 99.72%, 99.93%, 99.94% and 99.93% for FIFO, DRR, RED-ECN and HCF switches, respectively. This is due to the relatively high arrival rate of the packets to the switch, causing the queue to become empty extremely rarely. Therefore, there was no point plotting it.

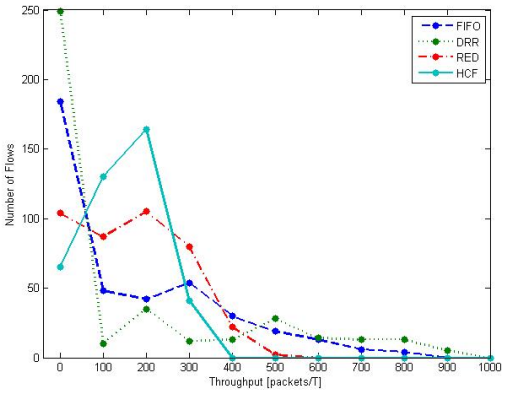


Figure 3: PDF of per-flow throughput in number of received packets during the period of 10 seconds.

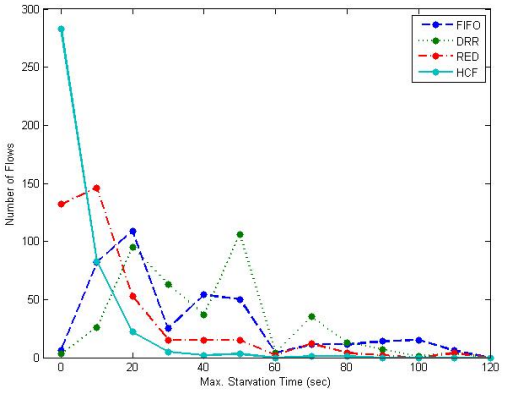


Figure 4: PDF of the longest flow starvation times per flow during the period of 10 seconds.

5.1.2. Throughput Distribution and Starvation Time

Figure 3 plots the distribution of the *per-flow throughput*. For the FIFO-based switch, we can see that most of the flows have a low throughput during time T, while several flows have a high throughput. Therefore, the distribution is extremely unfair, because a few flows send significantly more packets than the others.

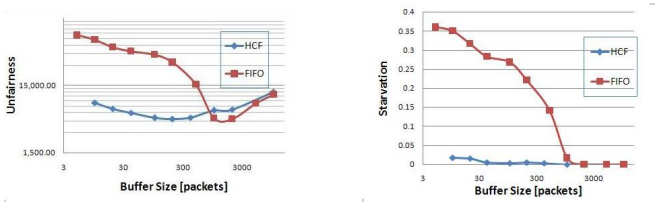
On the other hand, the distribution of the HCF algorithm is more concentrated around its mean, thus displaying a *lower unfairness*. This lower unfairness is reflected by the lower distribution variance: while the FIFO, DRR and RED switches have a computed variance of $5.55 \cdot 10^4$, $9.41 \cdot 10^4$ and $1.79 \cdot 10^4$ respectively, the HCF switch has the lowest variance of $6.74 \cdot 10^3$.

Likewise, we can also see that fewer flows are significantly impacted by sending nearly packets (65 flows for HCF are in the first histogram bin vs. 185, 250 and 105 flows for FIFO, DRR and RED, respectively), thus reflecting as well on the *lower starvation*. More precisely, 31% of flows in FIFO and DRR are fully starved during this period of 10 seconds, 8% of flows in RED, while only 0.025% of flows in HCF are fully starved.

Figure 4 illustrates the impact of HCF on starvation time. For each TCP flow, we measure the longest starvation time, i.e. the longest inter-ACK time, over a 3-minute simulation. We then plot the distribution of this longest starvation time.

Given a simple *FIFO*-based switch with droptail queueing and output-queued switching, the plot shows that several flows have a starvation time that exceeds a minute, and that the starvation time of most flows exceeds 20 seconds. On the other hand, when using *HCF* in the switch, it can be seen that most starvation times are under 20 seconds, thus potentially having a significant impact on application performance (even though the performance might of course still not be acceptable for the applications that are most latency-sensitive).

The distribution of the DRR algorithm is close and sometimes worse that of the FIFO queue. The reason is that DRR does not reserve buffer space for the flows that have not been served recently, as in the LP queue in HCF, and therefore the served flows tend to fully occupy the small buffer. The RED-based switch performs worse than HCF, because it is less efficient in a congested scenario where the average window size of a flow is smaller than 1.



(a) Unfairness. (b) Starvation Percentage.

Figure 5: Influence of the buffer size. Comparison between an HCF switch and a FIFO-based switch.

5.1.3. Buffer Size

Figure 5 shows the influence of the buffer size on the unfairness and starvation. It is obtained by changing the buffer size B in the baseline defined above.

We can see that for low buffer sizes, the unfairness and starvation with HCF are significantly lower than with FIFO. This is the case we are most interested in, since data center buffers are often shallow and only consist of a few packets. This simulation illustrates how buffers with a few packets are enough with HCF to provide low starvation, while FIFO needs buffers of some 1000 packets. In fact, this can impact the switch architecture: given a packet size of 1500B, this scaled-down example with 400 flows would need at least 1.5MB of buffering per output. Therefore, a real-life switch with some 40000 flows might require some 100 times more buffering, thus not being able to store all packets in the internal memory, and requiring some external memory with significant hardware changes.

In addition, for higher buffer sizes, the unfairness and starvation are about similar. In particular, the unfairness is a bit lower with FIFO and starts increasing with large buffer sizes. A possible explanation is that large buffer sizes favor larger window sizes, and therefore a larger traffic burstiness, thus increasing the variance of the per-flow instantaneous throughput.

5.2. Short-Flow Throughput Collapse

5.2.1. Typical Data Center

We run simulations of the TCP incast problem with short-flow throughput collapse using the parameters of the data center incast scenario from [2]. We use 1 Gbps links, a 32 kB switch buffer, a 0.1 ms round-trip transmission time and data blocks of 1 MB. As in the typical incast scenario we assume that the 1 MB data blocks are equally distributed over the N storage servers and are transmitted to the requester over N TCP flows. Therefore, each flow sends $1/N$ MB of data.We simulate TCP-Reno flows with packets of size 1KB. The HCF switch hashes flows into 16 bins with 1 credit per bin, and uses a dynamic priority period. The buffer is divided equally between the HP and the LP queues with 16 packets per queue. The *block goodput* of the short TCP flows is *defined* as the size of the transmitted data (block size) divided by the longest finish time (latency) of the flow, as in [1, 2, 3, 4, 5]. We measure *block goodput* as a function of the number of flows (servers).

Again, we compare HCF with the RED-ECN-based and hashed-DRR-based switches. As we stated before, we used the default ns2 parameters for the RED-ECN queue, seting the minimal and the maximal thresholds to zero and to equal to the buffer size, respectively. The DRR queue mechanism's hash function hashes the flows into 16 bins.

Figure 6(a) compares FIFO, DRR, RED-ECN and HCF. HCF outperforms the other algorithms over most flow numbers, which are roughly divided into three regions:

- *Goodput collapse* — below 25 servers (flows), the size of each flow is long enough, so the influence of HCF on fairness is noticeable.
- *Goodput preservation* — between 25-100 servers, the flow lengths are too small to notice the influence of HCF on fairness.
- *Goodput recovery* — above 100 servers, the large number of flows causes increased congestion in both FIFO and HCF, but the HCF credit mechanism balances the packet drops across the flows, thus balancing the timeouts as well.

Therefore, HCF works better in two cases: with few flows per block, and with increased congestion.

Note that the HCF algorithm works at the switch level. Therefore it can be combined with the solutions surveyed in Section 2.1, which need to change the application and/or transport stacks, in order to achieve even better performance. The HCF algorithm mitigates the TCP incast due to the fair scheduling of the flows in the shallow switch buffer. The TCP incast problem arises, when TCP flows suffer from a varying number of retransmission time-out (RTO) events. With the traditional FIFO-based switches, while several flows accomplish their transmission with no RTO event at all, other flows suffer from many RTO events. The flows that do not suffer from the RTO increase their congestion windows faster and occupy the buffer space in the switch, thus preventing the loss-suffered flows from injecting new packets to the buffer. This unfairness of the number of RTO events per flow has a crucial effect on the total transmission time of a data block, because it depends on the finishing time of the latest flow. The HCF algorithm balances the RTO events between the flows by preserving buffer space for new flows in the HP queue that have previously suffered losses. Therefore it tends to equalize the finishing times of all flows and reduces the finishing time of the latest flow.

5.2.2. Next-generation Data Center

Next, we analyze next-generation data centers with parameters from [2], where the round-trip times are decreased from 100 μs to 20 μs, the transmission link capacity is increased from 1 Gbps to 10 Gbps, and the block size is increased from 1 MB to 80 MB. Figure 6(b) shows the comparison of FIFO, DRR, RED and HCF. As in the previous case, HCF outperforms the other algorithms over the whole range.

5.2.3. Fixed-sized Flows

Next, we keep the length of each flow constant to 10 KB (i.e. each flow sends the same amount of data regardless to N). We simulate a data center with typical parameters of a 1 GB link capacity and a 100 μs round-trip time. We compare the performance of a FIFO, DRR and RED-ECN switches with an HCF switch, which hashes flows into 128 bins.

Figure 6(c) compares FIFO, DRR, RED-ECN and HCF. We see that HCF outperforms the other algorithms only for a large number of flows. For a small number of flows, there is not enough congestion, so the advantages of HCF are limited; and at the same time, the HCF buffer space is not fully utilized, because HCF drops more packets then FIFO (to save space for HP packets).

5.2.4. Starvation Time

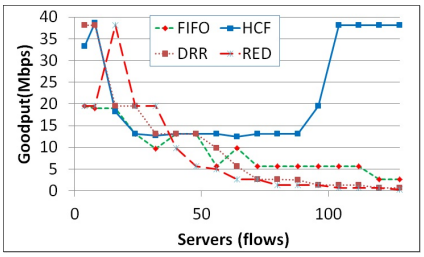
We define starvation time as the time between two packet arrivals (at the destination). The maximal starvation time is affected by the retransmission exponential back-off value, which grows with the number of consecutive RTO events.

Figures 7(a), 7(b) and 7(c) show the maximal observed starvation time at the FIFO-queue, DRR-queue, RED-queue and the HCF-queue based switches as a function of the number of servers. In most cases, the maximal starvation time in the HCF-queue based switch is smaller than in the FIFO, DRR and RED-ECN cases, which provides some basis for the better goodput.

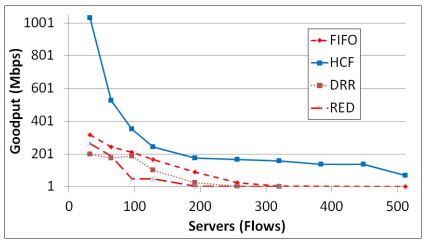
In addition, Figures 8(a), 8(b) and 8(c) show the average observed starvation time and Figures 9(a), 9(b) and 9(c) show its standard deviation between the flows. Again, in most cases, the average starvation time in the HCF-based switch is smaller than in the FIFO, DRR and RED-ECN cases.

5.3. Analysis of HCF switch parameters

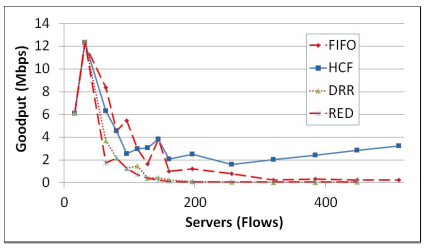
We now want to analyze how the parameters of the HCF algorithm impact its performance. These parameters were detailed in Section 4. We use a *baseline* for the set of simulation settings, and for each simulation vary a single parameter in this baseline. The *baseline* settings follow those defined in Section 5.1.



(a) Comparison of goodput at a Typical Data Center using FIFO, DRR, RED and HCF switches with fixed block size.

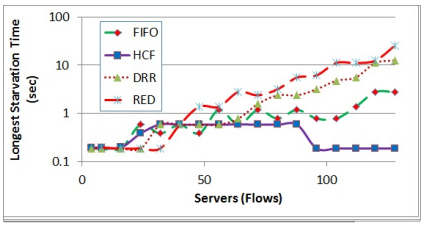


(b) Comparison of goodput at a Next-Generation Data Center using FIFO, DRR, RED and HCF switches with fixed block size.

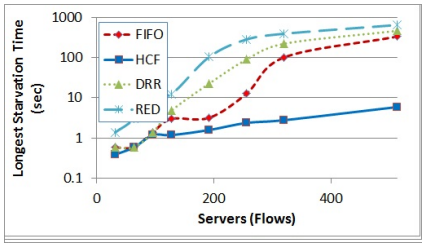


(c) Comparison of goodput at a Typical Data Center using FIFO, DRR, RED and HCF switches with fixed flow size.

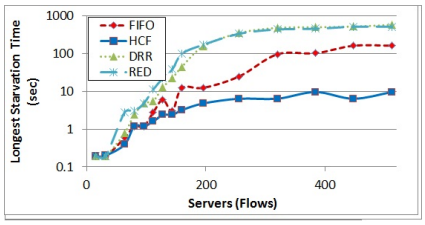
Figure 6: Comparison of goodput using FIFO, DRR, RED and HCF switches under TCP incast scenario.



(a) Comparison of Maximal Starvation Time at a Typical Data Center at FIFO, DRR, RED and HCF switches with fixed block size.

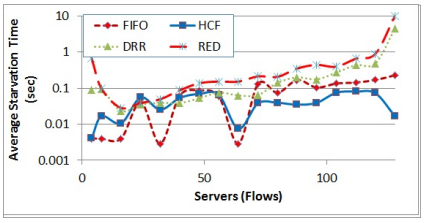


(b) Comparison of Maximal Starvation Time at a Next-Generation Data Center at FIFO, DRR, RED and HCF switches with fixed block size.

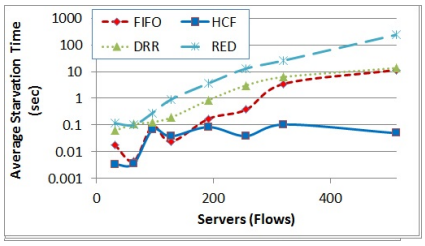


(c) Comparison of Maximal Starvation Time at a Typical Data Center at FIFO, DRR, RED and HCF switches with fixed flow size.

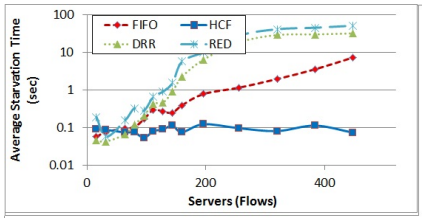
Figure 7: Comparison of Maximal Starvation Time using FIFO, DRR, RED and HCF switches under the TCP incast scenario.



(a) Comparison of Average Starvation Time at a Typical Data Center at FIFO, DRR, RED and HCF switches with fixed block size.

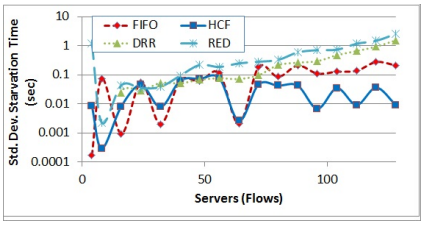


(b) Comparison of Average Starvation Time at a Next-Generation Data Center at FIFO, DRR, RED and HCF switches with fixed block size.

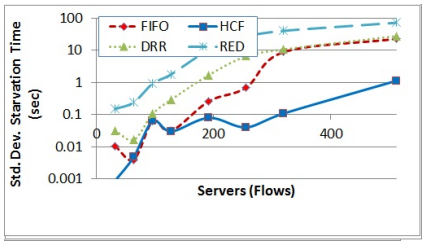


(c) Comparison of Average Starvation Time at a Typical Data Center at FIFO, DRR, RED and HCF switches with fixed flow size.

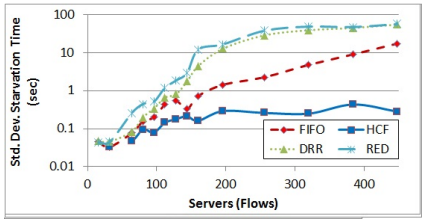
Figure 8: Comparison of Average Starvation Time using FIFO, DRR, RED and HCF switches under the TCP incast scenario.



(a) Comparison of Standard Deviation of a Starvation Times at a Typical Data Center at FIFO, DRR, RED and HCF switches with fixed block size.

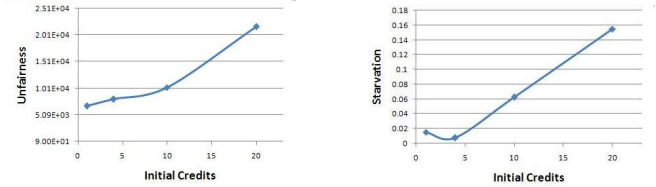


(b) Comparison of Standard Deviation of a Starvation Times at a Next-Generation Data Center at FIFO, DRR, RED and HCF switches with fixed block size.



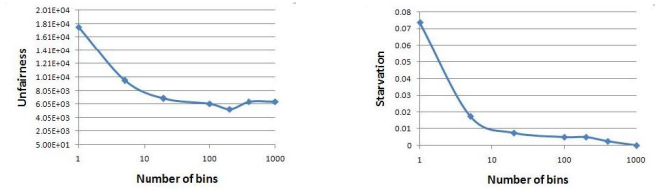
(c) Comparison of Standard Deviation of a Starvation Times at a Typical Data Center at FIFO, DRR, RED and HCF switches with fixed flow size.

Figure 9: Comparison of Standard Deviation of a Starvation Times using FIFO, DRR, RED and HCF switches under the TCP incast scenario.



(a) Unfairness. (b) Starvation Percentage.

Figure 10: Influence of the initial number of credits per bin.



(a) Unfairness. (b) Starvation Percentage.

Figure 11: Influence of the number of bins.

5.3.1. Initial Number of Credits per Bin

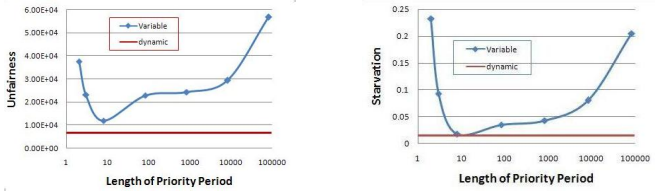
Figure 10 shows the influence of the number of initial credits c_0 on the unfairness and starvation. It can be seen that a large number of initial credits decreases the performance of the system; an intuitive explanation is that an HCF switch with an infinite number of credits becomes a FIFO switch, and therefore loses the benefits of the credits. In addition, smaller credits enable smaller priority periods, and therefore a fast renewal of the hash function, thus reducing the odds of persistent hash collisions between any two flows.

5.3.2. Number of Bins

Figure 11 shows the influence of the number of hash credit bins K on the unfairness and starvation. Of course, a smaller number of bins increases unfairness and starvation, and therefore decreases the performance of the system, because it increases hash collisions between flows. The plots show that a few dozen bins seem enough to provide a reasonable fairness and starvation. Given $c_0 = 1$, i.e. a single bit per bin, this means that only a few bytes stored in the register are needed for the HCF bins.

5.3.3. Priority Period Length

Figure 12 shows the influence of the priority period on the unfairness and starvation. The red solid line shows the value for the dynamic priority period. It is compared with different values of a fixed priority period. Simulations show that the dynamic priority period fares better than *any* of the fixed period values. Therefore, there is no need to fix the priority period value of HCF. It is an interesting result, in the sense that the system can learn to dynamically regulate itself better than any fixed regulation.



(a) Unfairness. (b) Starvation Percentage.

Figure 12: Influence of the fixed priority period. The red solid line shows the performance with a dynamic priority period.

5.4. Short TCP Flows over Long Flows

We have previously analyzed separately the performance of long-lived TCP flows and the short-lived TCP flows. We now want to analyze the performance of mixes of short and long flows, and in particular the influence of long flows on short flows. To do so, we change the simulation network to include only 100 long-lived TCP flows, and use a buffer size of $B = 10$ packets. We then add several short TCP flows, and check the total transfer latency time for each short TCP flow. We use three types of short flows: 10-, 30-, and 60-packet flows, and generate 100 flows of each type. The total simulation time is 5 minutes, and the start times of the short TCP flows are spread uniformly in the first half of the simulation time (i.e., the first 2.5 minutes).

Figure 13 compares the performance of the FIFO switch and the HCF switch under the same network parameters. It plots the CDF of the transfer latency when measured over the 100 flows of each type. For instance, a CDF value of 95% at a latency value of 25 seconds in the second plot indicates that 95% of the short flows of the second type (i.e., with 30 packets to send) finished transmission at most 25 seconds after they started.

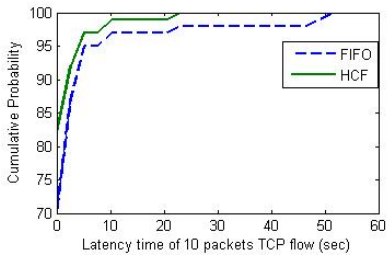
The plots show that HCF switches often provide a lower transfer latency than the FIFO switch. More significantly, they have lower odds of having long transfer latencies. This is especially meaningful for highly parallel applications, such as scientific computing and parallel database accesses, which start several flows in parallel and are dependent on the highest transfer latency among all of these.

6. Discussion

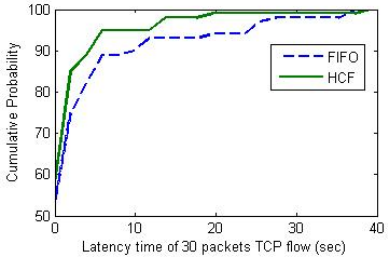
We now briefly discuss the correctness and generality of the assumptions made in this paper.

TCP Problems — We mentioned the incast, latency and the starvation problems in the Introduction. There are several other essential problems that need to be solved in order to use TCP in data centers, which we leave for further study. For instance: how to best load-balance TCP flows across multiple paths to enable adaptive routing; how to provide guaranteed delays for some of the flows; how to avoid congestion spreading in low-load networks; how to adapt to wireless and optical data center networks; and how to efficiently carry RDMA traffic [31, 32, 33, 34].

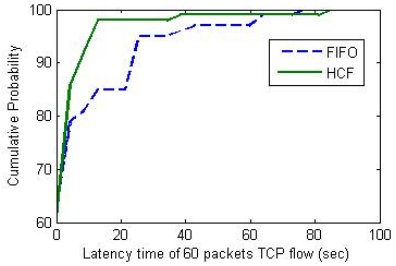
Single Bottleneck — We have assumed the existence of a main (or single) bottleneck link. This assumption relies on the



(a) CDF of Transfer Latency for TCP Flows of 10 packets.



(b) CDF of Transfer Latency for TCP Flows of 30 packets.



(c) CDF of Transfer Latency for TCP Flows of 60 packets.

Figure 13: CDF of transfer latency for short TCP flows of different sizes.

observation that TCP flows having more than one bottleneck actually depend on the most congested one [11]. Thus, the assumption seems realistic enough. However, we also assumed that congestion only affects packets, not ACKs. This assumption is too restrictive, and the HCF algorithm will need to be applied on the ACKs as well.

RTTs — In simulations, we did not consider RTT variations among flows. This assumption should have a limited impact, because the average propagation delay can often be neglect in front of the average queueing delay in data centers, as illustrated with the data center parameters in the Introduction.

Goodput vs. Throughput — We referred in our performance criteria for long TCP flows to the throughput, although the end user may be interested instead in the goodput, which is defined by the rate of *new* packets transmitted through the bottleneck link. However, we have seen in our simulations that the goodput was always within 5% of the throughput, therefore we neglected the difference between the two indicators.

Switch Model — We have modeled the switch as an output-queued switch. Real switches are input-queued (IQ) or combined-input-output-queued (CIOQ), making them harder to analyze. In addition, the datacenter switches are usually implemented with a speedup large enough that the switch scheduling effects are negligible [7]. Therefore, we neglect the influences of the switch scheduling algorithms, which might add other aspects of starvation and unfairness [35].

7. Conclusion

In this paper, we analyzed both the TCP-incast short-flow throughput-collapse problem, and the long-flow starvation problem. We presented the significant unfairness problem of TCP flows in data centers. To address it, we introduced HCF, a novel lightweight data center switch algorithm that is transparent to TCP-based applications. HCF combines a hashing-based credit allocation algorithm with a queueing mechanism that provides a higher priority to flows with credits. We further showed that HCF runs in an $O(1)$ time complexity, that it requires significantly less resources than competing fairness algorithms, that it does not incur reordering, and that it is transparent to the end-station users. Last, we illustrated through simulations that HCF can dramatically reduce unfairness and starvation for long TCP flows in data centers, as well as increase the goodput of short TCP flows that suffer from TCP incast problem.

It should be noted that while the HCF algorithm was especially studied in the framework of data centers, it can be readily adapted to provide fairness in other types of networks, since it uses a simple and compact structure that can be easily generalized.

References

[1] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, and G. A. Gibson, "A (in)cast of thousands: scaling datacenter TCP to kiloservers and gigabits," Technical Report CMU-PDL-09-101, Carnegie Mellon, Feb. 2009.

[2] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson and B. Mueller, "Safe and effective fine-grained TCP retransmissions for datacenter communication," *ACM SIGCOMM'09*, Aug. 17-21, 2009.

[3] E. Krevat, V. Vasudevan, A. Phanishayee, D. G. Andersen, G. R. Ganger, G. A. Gibson and S. Seshan, "On application-level approaches to avoiding TCP throughput collapse in cluster-based storage systems," *Supercomputing'07*, Nov. 10-16, 2007.

[4] Y. Chen, R. Griffith, J. Liu, R. H. Katz and A. D. Joseph, "Understanding TCP incast throughput collapse in datacenter networks," *WREN'09*, Aug. 21, 2009.

[5] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson and S. Seshan, "Measurement and analysis of TCP throughput collapse in cluster-based storage systems," *FAST '08*, Feb. 2008.

[6] J. Zhang, F. Ren and C. Lin, "Modeling and understanding TCP incast in data center networks," *IEEE INFOCOM '11*, Apr. 2011.

[7] David Newman, "Latency and jitter: cut-through design pays off for Arista, Blade", *Network World*, `http://www.networkworld.com/reviews/2010/011810-ethernet-switch-test-latency.html`

[8] R. Morris, "TCP behaviour with many flows," *IEEE International Conference on Network Protocols (ICNP'97)*, Oct. 1997.

[9] R. Morris, "Scalable TCP congestion control," *IEEE Infocom'00*, Tel Aviv, Israel, Mar. 2000.

[10] C. Villamizar and C. Song, "High performance tcp in ansnet", *ACM Computer Communications* Review, Vol. 24, No. 5, pp. 45-60, 1994.

[11] G. Appenzeller, I. Keslassy and N. McKeown, "Sizing Router Buffers, *ACM SIGCOMM*, 2004.

[12] H. Wu, Z. Feng, C. Guo and Y. Zhang, "ICTCP: Incast Congestion Control for TCP in data center networks," *ACM Co-NEXT '10*.

[13] Y. Sun, C. C. Lee, R. Berry and A. H. Haddad, "An application of the control theoretic modeling for a scalable TCP ACK pacer", *AACC*, 2004.

[14] J. Wu, Y. Shi, P. Zhang, S. Cheng and J. Ma, "ACK delay control for improving TCP throughput over satellite links", *Networks*, Vol. 28, pp. 303-312, 1999.

[15] M. Barbera, A. Lombardo, C. Panarello and G. Schembra, "Active window management: an efficient gateway mechanism for TCP traffic control", *ICC*, 2007.
[16] D. Stiliadis and A. Varma, "Latency-rate servers: a general model for analysis of traffic scheduling algorithms," *IEEE/ACM Transactions on Networking*, Oct. 1998.
[17] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin", *IEEE/ACM Transactions on Networking*, Vol. 4, No. 3, pp. 375–385, Jun 1996.
[18] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control inintegrated services networks: the multiple node case," *IEEE/ACM transactions on networking*, 1994.
[19] P. E. McKenney, "Stochastic fairness queueing", *IEEE INFOCOM '90*, Vol. 2, pp. 733-740, Jun. 1990.
[20] S. Floyd and V. Jacobson, "Random early detection (RED) gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, Vol. 1, No. 4, pp. 397-413, Aug. 1993.
[21] D. Lin and R. Morris, "Dynamics of random early detection," In Proc. of *ACM SIGCOMM*, September 1997.
[22] W. Feng, D. Kandlur, D. Saha, and K. Shin, " Blue: a new class of active queue management algorithms," In *UM CSE-TR-387-99*, Apr. 1999.
[23] W. Feng, D. D. Kandlur, D. Saha and K.G. Shin, "Stochastic fair blue: a queue management algorithm for enforcing fairness," *IEEE Infocom '01*, vol. 3, pp. 1520-1529, Apr. 2001.
[24] K. K. Ramakrishnan, S. Floyd and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP," *RFC 3168*, Sept. 2001
[25] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Datacenter TCP (DCTCP)," *ACM SIGCOMM '10*, Aug.-Sept. 2010.
[26] I. Maki, G. Hasegawa, M. Murata and T. Murase, "Throughput analysis of TCP proxy mechanism", *ATNAC'04*, Sept. 2004.
[27] O. Rottenstreich, Y. Kanizo and I. Keslassy, "The variable-increment counting bloom filter," *IEEE Infocom '12*.
[28] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," *ACM SIGCOMM'02* , Vol. 32, No. 4, pp. 323-336, 2002 .
[29] The network simulator 2, `http://www.isi.edu/nsnam/ns`.
[30] The NewReno modification to TCP's fast recovery algorithm, RFC 3782, `http://www.ietf.org/rfc/rfc3782.txt`.
[31] K. Kant, "Towards a virtualized data center transport protocol," *I*EEE Infocom Workshop on High-Speed Networks, Phoenix, AZ, April 2008.
[32] C. Minkenberg and M. Gusat, "Flow congestion management for 10G Ethernet," *Interconnection Network Architectures: On-Chip, Multi-Chip (INA-OCMC'09)*, Jan. 2009.
[33] R. Luijten, "Data center interconnects – trends for IB and Ethernet," *Presentation*, IBM Zurich Research Lab, Switzerland, 2006.
[34] Y. Birk and V. Zdornov, "Improving communication-phase completion times in HPC clusters through congestion mitigation," *SYSTOR'09*, May 2009.
[35] A. Shpiner and I. Keslassy, "Modeling the interactions of congestion control and switch scheduling," *IEEE IWQoS '09*, Jul. 2009.