Measurement Based Fair Queuing for Allocating Bandwidth to Virtual Machines

Khoa To, Jitendra Padhye, George Varghese, Daniel Firestone
Microsoft

ABSTRACT

We wish to allocate outgoing bandwidth at a server among customer VMs. The allocation for each VM is proportional to the bandwidth purchased for that VM by the customer, and any idle bandwidth is also proportionally redistributed. This is the classical fair queuing problem. However, most solutions [9, 5, 4] to the classical fair queuing problem assume tight feedback between transmitter and scheduler, and cheap scheduler invocation on every transmission. Since these assumptions are false in Virtual Switches, we propose MBFQ (Measurement Based Fair Queuing) with two levels of scheduling: a microscheduler that operates cheaply and paces VM transmissions, and a macroscheduler that periodically redistributes tokens to microschedulers based on the measured bandwidth of VMs. We show that MBFQ allows a VM to obtain its allocated bandwidth in three scheduling intervals, and that idle bandwidth is reclaimed within five periods. An implementation of MBFQ is available in Windows Server 2016 Technical Preview.

1. INTRODUCTION

We revisit the age-old problem of weighted fair allocation of bandwidth among competing "flows". Why do we need a new solution for such a well-studied problem? The reason is the new context and requirements imposed by *software implementation* in a *virtualized cloud* environment. This new context both *demands* and *allows* a simple and CPU-efficient solution.

Fair bandwidth allocation has typically been studied in the context of a router. It was assumed that one had to deal with thousands, if not millions of flows. It was also assumed (perhaps unjustifiably) that bandwidth had to be apportioned at a fine granularity approaching ideal processor sharing. A tight coupling between the packet transmission engine and the packet scheduler was also assumed, since both were implemented in switch hardware. For example, the DRR [9] implementation assumes that the scheduler is woken up on every packet departure. Further, the "CPU utilization" was important only in the sense that the switch hardware had to capable of handling the expected workload because the CPU (switch hardware) was dedicated to scheduling.

Our context is quite different. We want to build a packet scheduler for Windows Hyper-V Virtual Switch that ensures that VMs hosted on a single server share outgoing bandwidth in proportion to specified weights. Customized versions of this Vswitch powers Azure, and many large private corporate clouds. Our context implies that:

We cannot assume any hardware support: We must support legacy deployments, and so cannot assume NIC hardware support for packet scheduling. Instead, it must be be done by the CPU.

Coordination costs are high: Coordination between a software Vswitch and a hardware NIC is very expensive at Gigabit speeds, requiring optimizations like Large Send Offload. Transmission from a VM to the NIC should ideally bypass a coordinating scheduler to minimize context switches, and avoid access to shared state to minimize lock overhead. Further, to scale to 40 Gbps, the scheduler should be distributed across cores instead of being single-threaded.

CPU cycles are precious: For a cloud provider, any CPU cycles saved from say packet scheduling can be resold for profit [3]. This requires us to depart from the state-of-the art software solutions such as QFQ [5] that requires a CPU to do significant processing *on every sent packet*.

Fine-grained guarantees are unnecessary: Public and private cloud providers typically host less than 100 VMs per physical server, and provide only coarse per-VM bandwidth guarantees determined by the pricing tier. A granularity of 1 Mbps typically suffices. Fine-grained, per-flow bandwidth is neither used nor demanded by the customers, since application-layer issues often have far more impact on throughput.

We *do*, however, want our scheduler to be as work conserving as possible, and we *do* want to allocate any spare bandwidth roughly proportionally among backlogged VMs.

Thus, we designed a new packet scheduling algorithm, called Measurement-based Fair Queuing (MBFQ) to provide roughly proportional bandwidth sharing with minimum CPU overhead. We achieve this by refactoring scheduling

HotMIddlebox, August 22-26, 2016, Florianopolis , Brazil
 ISBN 978-1-4503-4424-1/16/08…$15.00
DOI: http://dx.doi.org/10.1145/2940147.2940153

into two parts: a *microscheduler* that controls the send rate of a VM using a token bucket, and a *macroscheduler* that periodically allocates tokens to each microscheduler based on VM bandwidth usage. Coordination costs are small and limited to the macroscheduler that runs once every scheduling period T (say 100 milliseconds). Each microscheduler can run in a separate thread/core. The split scheduler can thus scale to high overall transmit bandwidth with very little CPU overhead.

Our design tradeoff is that when demand changes, it takes a few macroscheduler periods (T) before the system converges to fair allocations, whether in allowing a previously idle VM to recapture its allocated bandwidth or to redistribute bandwidth when a VM's demand falls. T must be high not just to reduce CPU overhead but also to stably measure the bandwidth demand of a VM in the face of bursts.

Since the NIC does not provide per-packet feedback, the macroscheduler further ensures that the sum of bandwidths allocated to the VMs does not exceed the NIC bandwidth. To minimize bandwidth "wastage", the macroscheduler gradually increases/decreases the allocated bandwidth to a VM based on its measured demand in the last period.

The rest of the paper is organized as follows. We first describe our operating context in more detail (§2). Then, we describe the MBFQ algorithm (§3), and its implementation (§4). Finally, we evaluate the algorithm in (§5).

2. MODEL

We seek to allocate the outbound link bandwidth among client VMs, to ensure:

1. Congestion Freedom: The sum of the allocated bandwidths does not exceed the outbound link bandwidth.

2. Strong Fair Allocation: The bandwidth is divided among clients in proportion to their weights, not exceeding their demand. Leftover bandwidth is divided proportionally.

The classical fair queuing model (Figure 1(a)) used by algorithms such as DRR [9] and WF2Q [4] make two implicit assumptions. First, after packet transmission, the link alerts the hardware scheduler who chooses the next packet to transmit. Second, the scheduler is able to keep up with link speed. Both assumptions are reasonable for hardware routers.

However, in our model (Figure 1(b)) the feedback loop between link and scheduler is batched. Modern NICs use mechanisms like Large Send Offload (LSO) to minimize overhead, so they can only generate one send-complete notification for a group of packets. Without per-packet feedback, a DRR software implementation will receive transmit completion notifications in bursts. Figure 2 shows the distribution of transmit-complete notifications for a random 1000 contiguous samples, for three different traffic patterns. The figure shows that the size of each notification can be on the order of 500 kilobytes, and they can be microseconds (10000 ticks) apart. If we run DRR under such conditions, it would cause transmit jitter and packet drops; upper-layer throttling by the TCP stack will further exacerbate the issue. Figure 2 also shows that the distribution depends on the traffic pattern. Thus, the scheduler cannot simply be tuned to respond to a certain distribution.

Implementations of weighted fair queuing algorithms like QFQ [5] solve this problem by using technologies such as DPDK [1] or NetMap [7], that allow them to bypass the NIC batching. This is not feasible in our scenario. For fine-grained scheduling, they also require more CPU cycles than we can afford to spend.

We must also consider another subtle issue with classical fair queuing model. The software entity that schedules packets from the queues must ensure that packets are en-queued to the NIC's transmit buffer in the same order that they are de-queued from the VSwitch queues. Without preserving the ordering, queues on different CPUs would compete, post qos-scheduler, for the NIC's transmit buffer, leading to loss of fairness at the NIC level.

There are two options to achieve this. First, the software could use a handle to the NIC's transmit buffer. This requires strong coordination between the software module and the NIC's driver. This is not desirable for a software implementation on top of a hardware abstraction layer (e.g. NDIS in Windows), in a general-purpose OS that must work across many NIC vendors. A second alternative is to make the software entity single-threaded so that each packet is processed sequentially through the entire software stack from VSwitch to NIC using a single processor. This is not scalable at multi-gigabit speeds. Additionally, having to process packets on a single thread forces packets that were processed on different processors to be then processed on a different single processor, leading to cache misses which increase latency.

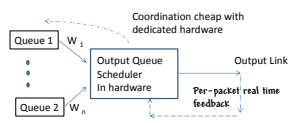
Therefore, our implementation needs to be able to schedule packets across multiple processors. This means we cannot satisfy the second implicit assumption that classical fair queuing models made (namely the scheduler can keep up with link speed) if the scheduler is invoked on a per-packet basis. Signaling events across processors on a per-packet basis is prohibitively expensive. For example, in Figure 2, transmission completes can occur on the order of nanoseconds. Signaling a scheduler at that granularity would require the CPU to spend most of its time doing the signaling, leaving very few cycles for packet processing. It typically takes at least one dedicated core to saturate a 40Gbps link.

Our solution is to divide the scheduler into two entities as shown in Figure 1(c). The macroscheduler runs only every T seconds and hence can run on a single thread. The microschedulers, by contrast, run on every packet based on tokens allocated by the microscheduler.

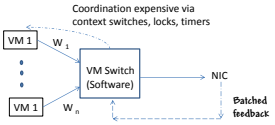
While this model reduces overhead, it has obvious drawbacks because allocations can only occur in units of T seconds. Thus, our evaluation will focus on the worst case time for a VM to ramp up to its guaranteed bandwidth, and its counterpart: the minimum amount of time needed to redistribute "unused" bandwidth from a VM that is not fully using its allocated bandwidth to other active VMs.

3. MBFQ ALGORITHM

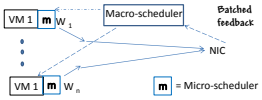
Table 1 shows the notation we use in this section. The



(a) Standard Fair queuing model



(b) VM Fair Queuing model



(c) Macro-Micro scheduler model

Figure 1: Scheduling models

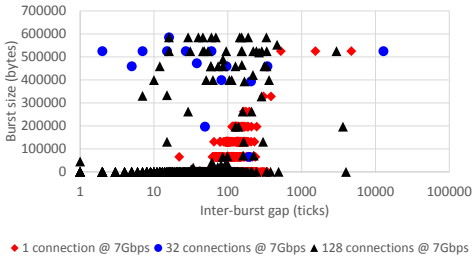


Figure 2: Burstiness of packet transmit-completes

C	The link capacity to share among VMs
$AvailBw_{All}$	Bandwidth available, initialized to C.
W_{All}	Total weight of VMs that need bandwidth.
$VmCount_{All}$	Number of VMs that need bandwidth.
SR_i	The measured send rate of the VM.
AR_i	The current allocated rate for the VM.
TR_i	The target rate for the VM, based on its SR_i.
RU_i	Consecutive iterations VM needs bandwidth.
MG_i	The VM's minimum bandwidth guarantee.
W_i	The VM's weight, the ratio of MG_i to C.
NR_i	New rate being allocated to the VM.
$BelowTR_i$	Whether NR_i is less than TR_i.

Table 1: MBFQ notation

MBFQ algorithm runs every T seconds. Rates are computed in two phases. First, we compute an ideal target rate (TR) for each VM, by comparing its measured send rate (SR) to the allocated rate (AR). TR is a proxy for what the VM would desire if it were not subject to constraints like link bandwidth, and sharing with other VMs. In the second phase, we adjust the TR to ensure that the link is not oversubscribed and rates are allocated according to weights. These adjusted target rates then become the allocated rates.

Phase 1: Computing target rates: The pseudocode for phase 1 is shown in Figure 3. We calculate a new target rate for every VM, based on its recently-measured send rate (SR). Intuitively, if a VM is not fully using its allocated rate, we should offer the residual bandwidth to other VMs. On the other hand, if the VM is indeed fully using its allocated rate, it may be able to use additional bandwidth.

We assume that a VM is fully utilizing is allocated bandwidth if $SR \geq 0.95 * AR$ (line 9). The 5% margin allows for small, unavoidable variations in the sending rate. To increase the bandwidth allocated to such a VM we need to strike a balance between two competing goals. First, we must minimize "waste"– the VM may not be able to use the additional bandwidth that we give it. Second, customers must be able

```
1:  VmCount_All ← 0
2:  W_All ← 0
3:  AvailBw_All ← C
4:  for (each VM sharing the link capacity) do
5:      SR_i ← AverageSendRateInLastTSeconds
6:      if (AR_i = Disabled) then
7:          TR_i ← 1.1 × SR_i
8:      else if (SR_i < 0.85 × AR_i for the last 500 msec)
    then
9:          TR_i ← 1.1 × SR_i
10:         RU_i ← max(0, RU_i − 1)
11:     else if (SR_i > 0.95 × AR_i) then
12:         RU_i ← min(3, RU_i + 1)
13:         if (RU_i = 1) then
14:             TR_i ← min(1.2 × AR_i, AR_i + 0.1 × C)
15:         else if (RU_i = 2) then
16:             TR_i ← min(1.5 × AR_i, AR_i + 0.1 × C)
17:         else if (RU_i = 3) then
18:             TR_i ← max(2 × AR_i, MG_i)
19:     else
20:         TR_i ← AR_i
21:         RU_i ← max(0, RU_i − 1)
22:
23:     TR_i ← max(TR_i, 10Mbps)
24:     NR_i ← min(TR_i, MG_i)
25:
26:     if (NR_i < TR_i) then
27:         BelowTR_i ← true
28:         W_All ← W_All + W_i
29:         VmCount_All ← VmCount_All + 1
30:     else
31:         BelowTR_i ← false
32:
33:     AvailBw_All ← AvailBw_All − NR_i
```

Figure 3: MBFQ Phase 1

to quickly ramp up to the bandwidth that they have paid for, if they are backlogged.

Our approach works as follows (lines 10 - 16). If, for a given VM, $SR \geq 0.95 * AR$, we set $TR = 1.2 * AR$. If the VM qualifies again in the immediate next round, we set $TR = 1.5 * AR$, and if it qualifies again, we set $TR = \max(2 * AR, MG)$ – i.e. if needed, we let it go to full minimum guaranteed bandwidth (or higher). Thus, we are conservative in the first two rounds, but a customer with substantial pending demand is guaranteed to reach the minimum guaranteed bandwidth in three time intervals or less. This staged allocation increment is a balance between granting the VM its bandwidth guarantee as quickly as possible, and minimizing unused bandwidth (i.e. maximizing work-conserving property).

If, on the other hand, the VM is using less than 85% of its

34: **while** ($AvailBw_{All} > 0$ and $VmCount_{All} \neq 0$) **do**
35: **for** (each VM sharing the network adapter) **do**
36: **if** ($BelowTR_i = true$) **then**
37: $FairShare_i \leftarrow AvailBw_{All} \times W_i \div W_{All}$
38: $NR_i \leftarrow NR_i + FairShare_i$
39:
40: **if** ($NR_i \geq TR_i$) **then**
41: $AvailBw_{All} \leftarrow AvailBw_{All} + (NR_i - TR_i)$
42: $NR_i = TR_i$
43: $BelowTR_i \leftarrow false$
44: $W_{All} \leftarrow W_{All} - W_i$
45: $VmCount_{All} \leftarrow VmCount_{All} - 1$

Figure 4: MBFQ Phase 2

allocated bandwidth, i.e. $SR \leq 0.85 * AR$ for five consecutive intervals (i.e. 500 milliseconds), we reduce the bandwidth allocated to this VM by setting $TR = 1.1 * SR$.

Phase 2: Preventing Congestion and Enforcing Fair Sharing: The pseudocode for phase 2 is shown in Figure 4. Our goal is to adjust the allocated rates so that the link is not oversubscribed, and any leftover bandwidth is allocated in proportion to the VMs' weights.

We start by initializing each VM's allocated rate (AR) to be the minimum of its target rate (TR) and its guaranteed rate (MG) (line 24 in Figure 3) . This guarantees congestion freedom since the sum of the MG does not exceed the link bandwidth. But it can leave bandwidth on the table.

We distribute this "remaining" bandwidth among "needy" VMs (those whose target rates are more than their guaranteed rates), in proportion to their weights. This process must be iterative, since we may end up allocating a VM more than its target rate calculated in the first phase. Since the target rate is our proxy for what the VM would desire in an ideal world without sharing, we remove this bandwidth and iterate again. Note also that the process is guaranteed to terminate since at least one VM will be removed from the needy list in each iteration. In practice, the loop terminates within a few nanoseconds, even for 100s of VMs.

4. IMPLEMENTATION

In this section, we briefly discuss the implementation of key components of MBFQ in Windows Server 2016.

Binary location: MBFQ is implemented in Windows as a Hyper-V Virtual Switch Extension, which is an NDIS lightweight filter (LWF) driver [2].

Like other LWF drivers, MBFQ uses a standard NDIS API to receive outgoing packets from the upper stack and sends conformed packets to the lower stack through another standard NDIS API. The MBFQ implementation is completely abstracted from the application above and the hardware below. The only information from hardware that MBFQ receives is from a standard NDIS API for receiving link state status (NDIS_STATUS_LINK_STATE) to determine the link capacity of the underlying network adapter.

The macroscheduler: MBFQ uses a timer to invoke the macroscheduler every 100 milliseconds. if the network adapter utilization is below 80% the macroscheduler deactivates the microschedulers (if they have not already been deactivated), and immediately returns. Thus, when the link utilization is low, which is the common case in data centers [8, 6], outgoing traffic is not throttled, and MBFQ uses minimal CPU. If the link utilization is above 80%, the macroscheduler computes and distributes the transmit rates to the microschedulers as discussed earlier.

The microschedulers: The microschedulers are implemented as token buckets, whose rates are periodically adjusted by the macroscheduler. Each microscheduler can schedule packets across multiple processors, using per-processor sub-queues that share a common token bucket. This provides a fully distributed weighted fair queuing implementation where packets can remain on the same processor from the application layer to the NIC's transmit buffer.

5. EXPERIMENTAL EVALUATION

We have evaluated MBFQ extensively. Due to lack of space, we only present selected microbenchmarks in this paper. Our microbenchmarking testbed consists of two Dell PowerEdge R410 servers. Both servers have four 2.26GHz cores, with hyper-threading enabled, which gives it 8 logical processors. Each machine has 8GB of RAM and 10G NIC. Both machines run a flavor of Windows Server OS with Hyper-V enabled for network virtualization. This configuration emulates our common use case: VMs communicating with other VMs in the same data center. [1]

5.1 Picking a Timer Period for Measurement

Question: The macrosheduler runs every T time units. What is the right value of T?

Motivation: As discussed earlier, it can take up to three iterations (thus $3 * T$) for the macroscheduler to allocate a VM with its minimum guaranteed bandwidth. Therefore, we don't want T to be too large. On the other hand, recall that we measure send rate over the same time period. If T is too small, the measured send rate of the VM (SR) may be inaccurate. The primary source of inaccuracy is the inherent burstiness of TCP. To estimate the inaccuracy for various values of T, we carried out the following experiment.

Experiment: A single VM hosted on one of the servers sends data to an external server using TCP. In all, 32 TCP connections were used. The application generated data at 7.2Gbps. All rate allocation functionality of MBFQ was disabled, only the measurement code was active. We logged the measured send rate (SR) over 100 intervals. We repeated experiment for values of T ranging from 15ms to 1000ms. Figure 5 shows the standard deviation as a function of T.

Discussion: A value of T between 60 - 100ms reduces standard deviation of SR to 5%. Larger T will lead to further reduction, but decrease is small.

While a 100ms initial response time is not suitable for some latency-sensitive applications, many applications that we tested, including video streaming and file transfers, can tolerate this delay. Latency-sensitive applications can use hardware-assisted QoS such as 802.1p to isolate their traffic

[1] In modern data centers, most traffic is intra-DC [8, 6].

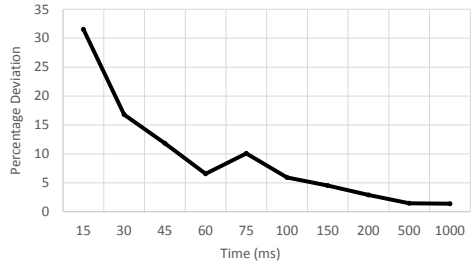


Figure 5: Impact of T on variance in SR

from the general VM traffic. Alternatively, an implementation of MBFQ could have a combination of static and dynamic bandwidth allocations where latency-sensitive VMs are assigned static queues, and other VMs dynamically share the remaining bandwidth.

5.2 Bandwidth Ramp Up and Ramp Down

Question: A VM may have to wait for up to three macroscheduler intervals to reach its minimum guaranteed bandwidth. Also, we wait for up to 500 milliseconds before we reclaim bandwidth from a VM that is not fully using the allocated bandwidth. Are these time intervals appropriate?

Motivation: While the allocated bandwidth to a VM ramps up to the minimum guaranteed bandwidth in three intervals, the VM may not be able to ramp up as quickly, due to limitations of TCP congestion control. Furthermore, waiting for 500 milliseconds before reclaiming the bandwidth may lead to under-utilization of the link.

Experiment: A test machine hosts 4 VMs with the following minimum bandwidth guarantees: VM1: 100Mbps (relative weight: 1), VM2: 100Mbps (relative weight: 1), VM3: 1Gbps (relative weight: 10), VM4: 3Gbps (relative weight: 30). Each of the VMs sends traffic to an external machine ("client machine") over the shared 10G external physical NIC. The link is configured to share 9.5Gbps among the VMs. Each VM is always backlogged and tries to send data as fast as it can.

Figure 6 shows the transmit bandwidth for each VM, and the total transmit bandwidth in several phases.

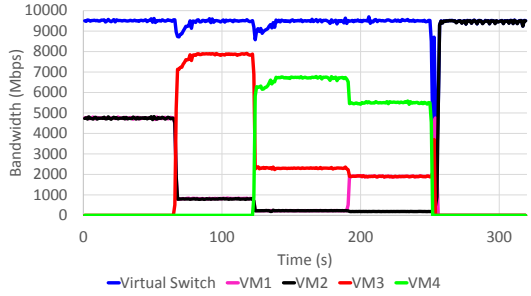


Figure 6: Ramp up and ramp down test

Phase 1: At time 0, only VM1 and VM2 are active. As expected, they share the bandwidth equally, with each getting 4.75Gbps.

Phase 2: At time 69, VM3 starts transmitting. MBFQ quickly throttles VM1 and VM2 to 792Mbps each, while VM3 is allowed to use 7.92Gbps.

Phase 3: At time 119, VM4 starts transmitting. The four VMs now get their weighted fair share: VM1 and VM2 get 226Mbps, VM3 gets 2.26Gbps and VM4 gets 6.79Gbps.

Phase 4: At time 189, we change the min bandwidth guarantee of VM1 from 100Mbps to 1Gbps. VM1 and VM3 now both transmit at the same rate, and the rates of VM2, VM3, and VM4 were reduced to accommodate the increase in VM1's rate.

Phase 5: At time 249, VM1, VM3, and VM4 stop sending. VM2 ramps up to consume the full link bandwidth.

Discussion: We see that while MBFQ generally performs well, there are dips in link utilization at times 69 and 119 as VMs ramp up. The dips in the aggregate bandwidth at the beginning of phases 3 and 4 are due to our desire to allow a VM to quickly attain minimum guaranteed bandwidth. As VM3 and VM4 are ramping up, the algorithm detects that the VMs requests for additional bandwidth in several consecutive iterations. Therefore, in order to quickly provide the VMs their subscribed bandwidth, after three consecutive iterations of additional bandwidth requests, the algorithm allocates the full fair share of bandwidth to the VM. However, even after the bandwidth is allocated, the VMs could take some time to consume all allocated bandwidth, due to TCP artifacts. Thus, the dip represents a trade-off between how fast the algorithm should grant a VM its fair share versus how cautious it should be in allocating the VM bandwidth that it might not be ready to consume (and thus risk being non-work conserving). A more extreme dip is seen at the beginning of phase 5, we shall discuss that in Section 5.4.

5.3 Can we ramp up any faster?

Question: Is the ramp up delay shown in Figure 6 for VM3 at 69 seconds caused by our algorithm or it due to the VM itself (e.g., its TCP behavior)?

Motivation: The previous experiment suggests the algorithm may be too slow in allocating bandwidth to a newly active VM.

Experiment: We measure bandwidth ramp up in two scenarios. First, we measure a "standalone" scenario where VM3 is the only VM transmitting (i.e. the link was idle before VM3 started sending). Second, we measure a "sharing" scenario in which the link was fully utilized before VM3 started sending. When VM3 starts sending, MBFQ performs bandwidth allocation to give VM3 its fair share.

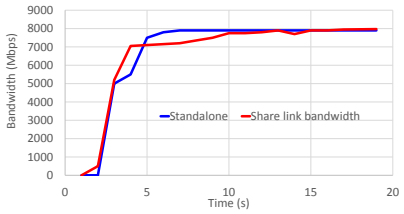


Figure 7: TCP and MBFQ ramp up

Discussion: As seen in Figure 7, it takes about the same amount of time in both scenarios for VM3 to achieve most of its bandwidth (up to 7Gbps). However, it takes an additional 15s for VM3 to reach its steady state in the "Share link bandwidth" scenario. However, in the sharing scenario, the VM is transmitting on top of a NIC that is already fully utilized.

But in the standalone case, the VM is transmitting on top of an idle NIC.

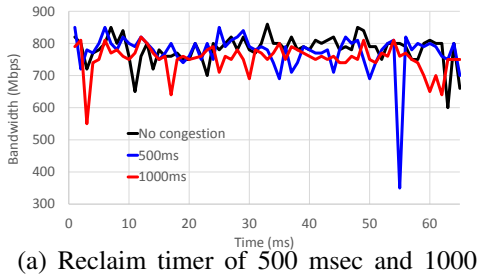
Conclusion: It is probably not worthwhile to ramp up faster because TCP may not be able to fully utilize the extra bandwidth quickly.

5.4 How fast should we reclaim bandwidth?

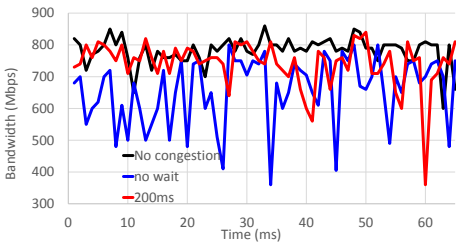
Question: The faster we reclaim bandwidth the faster we redistribute it other needy VMs who can use it. How large should the reclaim period T be?

Motivation: We see a large dip in the utilization at the beginning of Phase 5 in Figure 6 when VM1, VM3, VM4 stop sending. In addition to the TCP ramp-up time of VM2, the dip is also partly due to another parameter in the algorithm where we configure the algorithm to wait for 500 milliseconds before reclaiming residual bandwidth. Unfortunately, this is a tradeoff. The faster we reclaim, the more likely the algorithm is to spuriously reclaim bandwidth from a paid customer VM which has short term bursts.

Experiment: We measure the impact of different reclaim timer values on the throughput of a large file transfer operation in a VM. We have VM1 host a large file that is being copied to a remote machine, while VM2, VM3, VM4 send background CBR traffic to fill up the idle link. The file transfer application on VM1 uses about 800Mbps, and the rest of the link bandwidth is distributed among VM2, VM3, VM4. We change the wait time parameter from No Wait (at every macroshedular iteration, bandwidth is immediately reclaimed if VM1 is sending less than 85% of its allocated rate) to 1000ms wait (bandwidth is not reclaimed unless the VM has been sending less than 85% of its allocated rate in the last 1000ms)



(a) Reclaim timer of 500 msec and 1000 msec



(b) Reclaim timer of 0 msec and 200 msec.

Figure 8: Impact of reclaim timer

Figure 8(a) shows that the bandwidth stays roughly the same with MBFQ and without MBFQ with a reclaim timer of 500 or 1000 msec. On the other hand, Figure 8(b) shows that with instantaneous reclaiming (reclaim timer of 0) the bandwidth allocated to the VM is significantly affected and its noticeable even at a reclaim timer of 200 msec.

Conclusion: A reclaim timer of 500 milliseconds was chosen as a compromise.

5.5 CPU Utilization

We measure the CPU overhead of MBFQ for the setup in Section 5.2 by comparing the CPU utilization of that setup to the CPU utilization of a base scenario without MBFQ.

For the base scenario, we assign static rate limits to VM1, VM2, VM3, and VM4 with 226Mbps, 226Mbps, 2.26Gbps, and 6.79Gbps respectively. These are the rates that MBFQ dynamically distributes to the VMs when all VMs are active.

With all VMs active, the CPU utilization of the base scenario is 22.74%, and CPU utilization with MBFQ is 23.22%. This shows that MBFQ has minimum CPU overhead.

6. CONCLUSION

We presented MBFQ, a measurement-based fair queuing model for allocating bandwidth to virtual machines. MBFQ's two-tier scheduling model is scalable across multiple processors. MBFQ provides weighted fairness among virtual machines without any specific knowledge of the NIC. Our implementation of MBFQ in Windows is a standard NDIS filter driver that can reside in any part of the networking stack. Our evaluation on Windows Server 2016 shows that MBFQ can allocate fair bandwidth sharing among VMs of different weights. It responds quickly to changes in bandwidth demands among VMs with minimum CPU overhead. An implementation of MBFQ is available with Windows Server 2016 Technical Preview.

7. REFERENCES

[1] Data plane development kit. *http://dpdk. org*.

[2] Windows filter driver. *https://msdn.microsoft.com/en-us/library/windows/hardware/ff545890(v=vs.85).aspx*.

[3] Amazon elastic compute cloud (amazon ec2). 2010.

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

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

[6] S. Kandula et al. The nature of data center traffic: measurements & analysis. In *Proc. of IMC*, 2009.

[7] L. Rizzo. Netmap: a novel framework for fast packet i/o. In *USENIX Security*, 2012.

[8] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM*, 2015.

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