Providing Bandwidth Guarantees, Work Conservation and Low Latency Simultaneously in the Cloud

Shuihai Hu[1], Wei Bai[1], Kai Chen[1], Chen Tian[2], Ying Zhang[3], Haitao Wu[4]
[1]SING Group @ HKUST [2]Nanjing University [3]HP Labs [4]Microsoft
{shuaa,wbaiab,kaichen}@cse.ust.hk tianchen@nju.edu.cn ying.zhang13@hp.com hwu@microsoft.com

***Abstract*—Today's cloud is shared among multiple tenants running different applications, and a desirable multi-tenant datacenter network infrastructure should provide bandwidth guarantees for throughput-intensive applications, low latency for latency-sensitive short messages, as well as work conservation to fully utilize the network bandwidth. Despite significant efforts in recent years, none of them can achieve these three properties simultaneously. In this paper, we identify the key deficiency of prior solutions and use this insight to motivate our design of Trinity—a simple, practical yet effective solution that achieves bandwidth guarantees, work conservation and low latency simultaneously in the cloud. We implement Trinity using existing commodity hardwares and demonstrate its superior performance over prior solutions using testbed experiments.**

I. Introduction

In today's clouds, the network resource, unlike the compute and storage resources, is shared in an uncoordinated best-effort manner among multiple tenants. For this reason, the tenants may experience varied network performance which can adversely affect their application performance and increase their cost. For example, recent studies on several major cloud infrastructures have revealed that bandwidth and packet latency can vary significantly by an order of magnitude [1]–[5]. The lack of predicted performance has prevented users and enterprises from migrating their applications into the cloud, especially for delay sensitive applications such as web search, retail, advertising, recommendation systems, etc.

A natural way to provide predicted network performance is to let the users specify the amount of bandwidth they need and allocate dedicated bandwidth to them, i.e., providing *bandwidth guarantees* to the tenants. However, such strict bandwidth allocation may result in bandwidth waste if the tenant cannot fully utilize his share. Thus, the cloud network should also provide *work conservation* to enable the multiplexing economic benefits for the cloud provider. At the same time, it should provide *low latency* to short flows for small response time. As a result, a good cloud network design should be able to meet these three objectives simultaneously.

While significant efforts [1, 6]–[15] have been made toward sharing the cloud and obtaining predictable network performance, none of them achieves all the three goals simultaneously. For example, SecondNet [9] and Oktopus [1] provide bandwidth guarantees, but they are not work-conserving. ElasticSwitch [6] aims at work-conserving bandwidth guarantees, however it cannot ensure low latency for short flows, and more importantly, its work conservation is sacrificed due to a fundamental tradeoff between accurately providing bandwidth guarantees and being work-conserving (see details in §II).

We identify that a key deficiency of prior solutions such as ElasticSwitch [6] is that they heavily rely on end-to-end rate control while neglecting important support from network. The reason why ElasticSwitch has to sacrifice work conservation for bandwidth guarantees is that: it injects without distinction the traffic of both bandwidth guarantees and work conservation into the network; the network, by itself, cannot automatically avoid the interference between these two types of traffic. Consequently, work-conserving traffic of one tenant, if too aggressive, can adversely affect bandwidth guarantee traffic of other tenants and hurt the latency of their short flows.

This directly motivates our design of Trinity in this work. By Trinity, we show that simple network support can be explored to solve the problem. Observing that today's commodity switches already support 4–8 priority queues [16]–[18], our key idea in Trinity is that by simply differentiating the two types of traffic at the end and prioritizing them in the network, we can readily achieve all the triple goals simultaneously.

Basically, Trinity decouples providing bandwidth guarantees from being work-conserving by segregating these two types of traffic at the end, and leveraging commodity switches to enforce priority queueing in the network. The traffic of bandwidth guarantees is prioritized over that of work conservation. With such prioritization, work conservation can be designed aggressively without affecting bandwidth guarantees. Furthermore, such prioritization also makes it easier for Trinity to achieve low latency for short flows: it only needs to classify packets of short flows as bandwidth guarantee traffic and let them receive priority in the network (see details in §III).

Despite being conceptually simple, there are still a few concrete issues we need to address before making Trinity truly effective. First, how to design an aggressive rate control algorithm so that the work-conserving traffic can fully utilize spare bandwidth in the network, while not causing a large number of packet drops at switches. Second, how to handle packet trapping (or starvation) of the work-conserving traffic in the lower priority queue. Third, how to deal with possible packet re-ordering which might occur when a long flow promotes from the lower priority queue to the higher one. In §III-C, we introduce how Trinity addresses each of them.

We have implemented a Trinity prototype with commodity servers and switches (§IV). On the end host, our Trinity kernel module is located as a shim layer over the physical NIC

978-1-4673-9953-1/16/$31.00 ©2016 IEEE

	Design objectives			System requirements		
Related Work	BW guarantee	Work conservation	Low latency	Switch hardware	Topology	Control model
Oktopus [1],TIVC [10] SecondNet [9]	Yes	No	No	None MPLS	None	Centralized
GateKeeper [14] EyeQ [8]	Yes	Yes	No	None ECN	Congestion-free core	Distributed
Seawall [11], NetShare [12] FairCloud PS-L/N [13]	No	Yes	No	None	None	Distributed
FairCloud PS-P [13]	Yes	Yes	No	Per-VM queues	Tree	Distributed
Silo [7]	Yes	No	Yes	None	None	Distributed
ElasticSwitch [6]	Yes, tradeoff with work conservation	Yes, tradeoff with BW guarantee	No	None	None	Distributed
Trinity	Yes, without tradeoff	Yes, without tradeoff	Yes	Priority queues, ECN	None	Distributed

TABLE I: Summary of previous approaches and comparison to Trinity

(Network Interface Card) driver in hypervisor. It does not introduce any modification to network stacks or applications of tenants. In the switch, Trinity only requires strict priority queueing and Explicit Congestion Notification (ECN) which are both built-in functions for existing commodity switches.

To evaluate Trinity, we build a testbed with 2 Pronto-3295 Gigabit Ethernet switches and 16 Dell servers. Our experimental results show that:

- Trinity provides accurate bandwidth guarantees while achieving good work conservation. For example, Trinity outperforms ElasticSwitch by 20.88%–53.06% in terms of the average throughput under different settings;
- Trinity delivers low latency for short flows and improves their flow completion time (FCT) significantly. For example, as for 1 KB short flows, compared to ElasticSwitch, Trinity reduces their FCT by $22\% - 33\%$ on average and by $68\% - 71\%$ at the 99th percentile;

The rest of the paper is organized as follows. §II discusses the problem and related work. §III introduces the Trinity design in detail. §IV and §V describe the Trinity implementation and testbed experiments. §VI concludes this paper.

II. Problem and Related Work

We consider the cloud sharing problem in this paper. When tenants arrive with specific bandwidth demands, the cloud provider needs to have a mechanism to handle their traffic. The mechanism should satisfy three properties in the following.

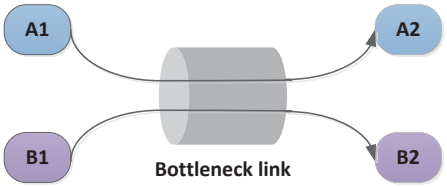
- Providing *bandwidth guarantees* means that each VM can share a minimum guaranteed bandwidth to send and receive traffic whenever needed. This is crucial for the predictable application performance, especially for data-intensive applications [19, 20] whose completion time mainly rely on the available network bandwidth.
- Being *work-conserving* requires that the bottleneck link should be always fully utilized as long as there are sufficient demands. This means that a tenant should be able to dynamically grab free bandwidth, which are either unallocated, or allocated but are not currently used by other tenants. Work conservation benefits both tenants and the provider because tenants can finish their jobs faster and the provider can achieve high resource utilization.
- Delivering *low latency* for short flows is crucial for many online data-intensive (OLDI) applications such as web services. For better user experience, many OLDI applications operate under soft real-time constraints that requires short flows to be completed before deadlines [21].

To the best of our knowledge, prior solutions do not achieve the three goals simultaneously. Table I summarizes some related work according to the objectives they meet and the assumptions they have. Specifically, SecondNet [9], Oktopus [1] and TIVC [10] provide bandwidth guarantees but not work-conserving, while Seawall [11] does the opposite. EyeQ [8] and GateKeeper [14] are work-conserving, but they require the network core to be congestion-free which is not the case for production datacenters [6]. Similarly, FairCloud PS-P [13] is also work-conserving, but at the cost of expensive switch hardware support especially per-VM queues. Furthermore, all these solutions do not consider low latency. On the other hand, Silo [7] considers guaranteed bandwidth and packet latency, but it does not achieve work conservation.

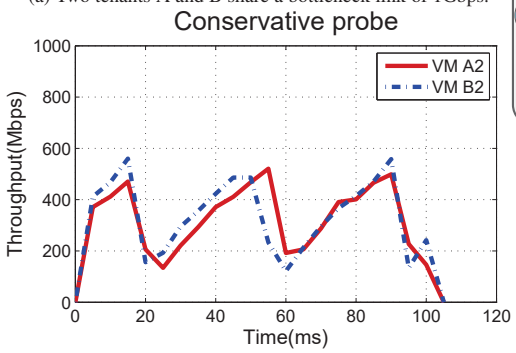
We also note that there exist some traditional solutions that tackles similar problems in the broader context of the Internet. For example, weight fair queuing (WFQ) [22] can be borrowed to achieve bandwidth guarantees and work conservation by using per-tenant dedicated queues. However, today's commodity switches have a limited number of queues (e.g, 4-8), which is far from enough for clouds with many tenants. For some other advanced schemes like [23, 24], their algorithms are too complicated to be implemented in commodity switches.

Deep dive: The work closest to Trinity is ElasticSwitch [6]. However, there is a fundamental tradeoff between accurately providing bandwidth guarantees and being work-conserving in ElasticSwitch. In order for a tenant to detect the spare bandwidth not being used by other tenants, ElasticSwitch needs to probe the available bandwidth by increasing the flow rates. However, probing too conservatively (i.e., increase gradually but drop dramatically) may under-utilize the available bandwidth and is not sufficiently work-conserving; while probing too aggressively (i.e., increase dramatically but drop gradually) may affect bandwidth guarantees of other tenants when their traffic come back to network.

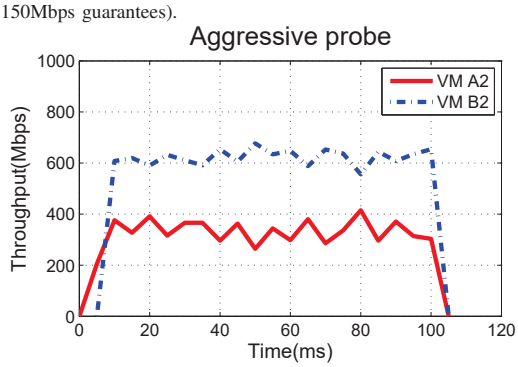
We show this dilemma using testbed experiments in Fig. 1. As shown in Fig. 1a, there are four VMs of two tenants A and B sharing a same bottleneck link, and VM A1 and B1 send traffic to A2 and B2, respectively. We measure the throughput at A2 and B2 every 5ms. In the first experiment, we assume both tenants have 150Mbps guarantees and use conservative



(a) Two tenants A and B share a bottleneck link of 1Gbps.



(b) Conservative probe is not work-conserving (both tenants have 150Mbps guarantees).



(c) Aggressive probe affects bandwidth guarantees (tenant A and B have 150Mbps and 750Mbps guarantees respectively).

Fig. 1: Deep dive experiments to show the dilemma.

probe. In this case, the ideal work conservation result should be that both tenants stay around 500Mpbs. However, in Fig. 1b, we can see that the scheme is not fully work-conserving, because it probes available spare bandwidth too conservatively by increasing rates slowly at the beginning, but dropping too dramatically once it senses congestion. Spare bandwidth in the valleys is wasted.

In the second experiment, we assume tenant A and B have 150Mbps and 750Mbps guarantees respectively and use aggressive probe. We let A take more spare bandwidth first, and later on more traffic from B arrives. However, in Fig. 1c, we can see that, under such aggressive probe, B even cannot get back its guaranteed bandwidth for a long while. The reason is that the work-conserving traffic of A adversely throttles the bandwidth guarantee traffic of B. Because A drops gradually upon congestion (but increases dramatically when seeing spare bandwidth), it makes B unable to grab its minimum guarantee of 750Mbps in a short time (although eventually it will).

We note that, in ElasticSwitch [6], they proposed solutions such as 10% headroom, hold-increase and rate-caution which essentially trade work-conservation for bandwidth guarantees, but do not completely solve the problem.

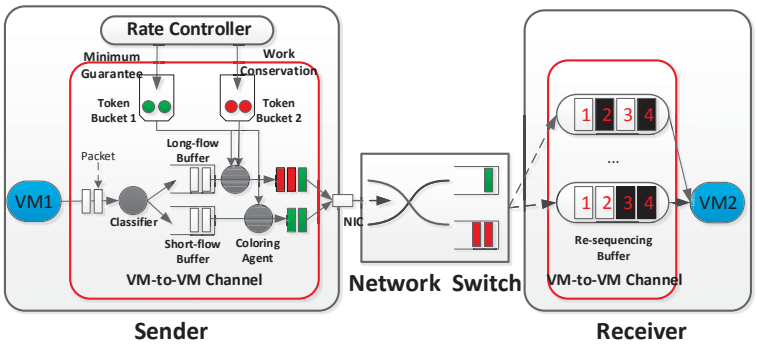


Fig. 2: Trinity system framework.

III. The Trinity Design

A. Design Overview

To solve the above dilemma, we seek network support instead of sticking to pure end-to-end solution. By simply differentiating the two types of traffic at the end and prioritizing them in the network, we break the impasse.

Specifically, Trinity decouples providing bandwidth guarantees from being work-conserving by differentiating traffic of bandwidth guarantees from that of work conservation with two colors at the end (i.e., green indicates bandwidth guarantee traffic, and red indicates work-conserving traffic), and leveraging commodity switch capability to enforce strict priority queueing in the network. That is, the traffic of bandwidth guarantees is always prioritized over that of work conservation in the network. With such prioritization enforced, work conservation can now be designed more aggressively without causing any interference to bandwidth guarantees. This effectively enables Trinity to achieve *absolute* bandwidth guarantees and work conservation without any tradeoff.

Meanwhile, such prioritization of bandwidth guarantee traffic over work conservation traffic also enables Trinity to optimize and ensure low latency for short flows: it only needs to make sure that the packets of short flows are colored as bandwidth guarantee packets. The reason is as follows. Since tenant's bandwidth guarantee requirement has already been met by the provider based on the network capacity in the tenant admission control phase, pure bandwidth guarantee traffic can be accommodated by the network without congestion and the packets will experience little, if any, queueing delay. In the case of mixed bandwidth guarantee and work-conserving traffic, as long as the prioritization is in place, bandwidth guarantee packets will not be blocked by work-conserving traffic, and thus still be able to see low latency.

B. System Framework

The system framework of Trinity is shown in Fig. 2. We have one Trinity software component for each VM-to-VM channel running in the hypervisors (shown in the red rectangle).

At the sender side, the Rate Controller (RC) module is responsible for determining the traffic rates for bandwidth guarantees and work conservation between the VM pair. Similar to ElasiticSwitch [6], Trinity provides hose model guarantees by transforming a hose model into a set of minimum bandwidth guarantees for each source-destination VM pair. For minimum bandwidth guarantee rate between a VM pair, Trinity directly employs guarantee partitioning (GP) technique developed in [6]. For work-conserving rate, Trinity fully utilizes available spare bandwidth between the VM pair through an aggressive rate control algorithm as we will introduce in detail later.

As a result, RC updates the minimum bandwidth guarantee rate and work-conserving rate for each active VM pair periodically. These two rates are fed to two token buckets to control the coloring process of outgoing packets (see Fig. 2): Token Bucket 1 generates tokens (green) at the rate of minimum guarantee, and Token Bucket 2 generates tokens (red) at the rate of work conservation. Unlike ElasticSwitch where work conservation must be compromised in order not to affect bandwidth guarantees, the work-conserving rate control function in Trinity has no such restriction. It can grow aggressively or drop conservatively to fully utilize available spare bandwidth.

To achieve low latency for short flows, as introduced, Trinity only needs to color all packets of short flows as bandwidth guarantee packets and lets them receive higher priority in the network. In the framework of Trinity, we employ a Classifier module to assign each flow to either a short-flow class or a long-flow class. To be practical, Trinity does not assume any prior knowledge of flow sizes; Instead, it prioritizes the first few packets of every new flow. The threshold can be initially set as a few or tens of KBs, a typical size of short flows for latency sensitive applications [25], and subject to improve by using advanced thresholding schemes such as [17]. In our implementation, the classifier keeps track of the bytes sent of every flow; if the bytes sent of a flow is less than a given threshold, then the flow remains in the short-flow class; otherwise, it is moved to the long-flow class until finish.

At the receiver side, Trinity employs a re-sequencing buffer, a common technique used by many prior works [26]–[28], to absorb potential out-of-order packets.

In the network switch, Trinity simply leverages 2-level priority queueing to enforce a strict prioritization of bandwidth guarantee traffic over work-conserving traffic. Furthermore, Trinity also leverages the ECN support of commodity switches for its rate control as shown later.

In general, the workflow of Trinity is simple. For packets in short-flow buffer, they only consume tokens in Token Bucket 1 (colored as green) and enjoy low latency in the network. In case Token Bucket 1 runs out of tokens (which could happen very occasionally, e.g., a persistent long flow consumes the last green token right before a new flow starts), the packets just wait temporarily for the new green tokens to be generated.

For packets in long-flow buffer, they can be colored as either green or red. When there are available tokens in Token Bucket 1 and short-flow buffer is empty, they are colored as green and identified as bandwidth guarantee traffic in the network; Otherwise, they are colored as red and identified as work-conserving traffic. This includes two possibilities: 1) no token in Token Bucket 1, this means the minimum bandwidth guarantee is reached; and 2) tokens available in Token Bucket 1 but short-flow buffer is not empty, in such case packets in long-flow buffer do not consume green tokens in order not to cause any delay to packets in short-flow buffer. It is possible that even Token Bucket 2 can run out of tokens, in this case, Trinity tries to buffer the packets in the long-flow buffer before dropping them when buffer occupancy grows too large.

C. Detailed Mechanisms

Despite being conceptually simple, there are still a few concrete design issues we need to address. We now discuss these problems and our solutions to them.

Problem #1: Rate control. As introduced, a key benefit of Trinity is that, by prioritizing bandwidth guarantee traffic over work-conserving traffic, we can employ aggressive rate control algorithm for work conservation without affecting bandwidth guarantees. Then the question is: what kind of rate control we should employ?

Solution: For each VM-to-VM channel, RC decouples minimum bandwidth guarantee rate (denoted as R_G) from work-conserving rate (denoted as R_W).

For R_G, we follow the approach of ElasticSwitch [6]: for a channel $X \rightarrow Y$, RC sets its bandwidth guarantee rate as:

$$R_G^{X \rightarrow Y} = min(B_X^{X \rightarrow Y}, B_Y^{X \rightarrow Y}) \quad (1)$$

where $B_X^{X \rightarrow Y}$ is the guaranteed bandwidth assigned by X's hypervisor for the traffic to Y, and $B_Y^{X \rightarrow Y}$ is the guaranteed bandwidth assigned by Y's hypervisor for the traffic receiving from X. Let B_X be the bandwidth guarantee of VM X. If X is sending traffic to N destination VMs with unbounded bandwidth demand, we have $B_X^{X \rightarrow Y} = B_X/N$. The computation for $B_Y^{X \rightarrow Y}$ is similar.

For R_W, RC uses an aggressive algorithm to update its value periodically. The idea is that when there is no congestion feedback from network, we let a VM-to-VM channel to send as much work-conserving traffic as the NIC allows; When there is congestion feedback from network, we reduce R_W in proportion to an estimation of network congestion.

Formally, let S be the set of VMs hosted on a server, and $\forall X \in S$, we use B_X to denote the bandwidth guarantee of X. Assume the capacity of the NIC is C, then the spare capacity:

$$C_W = C - \sum_{X \in S} B_X \quad (2)$$

To apply the idea mentioned above, the hypervisor divides all the active VM-to-VM channels into two sets P and Q, and computes work conserving rates for channels in different sets using different schemes. Here P is the set of congestion-free channels, while Q is the set of congestion-caution channels. Initially, we put all active channels in P.

Let C_P be the total spare capacity belonging to the congestion-free channels. It is easy to know that:

$$C_P = C_W - \sum_{u \to v \in Q} R_W^{u \to v} \quad (3)$$

Here $R_W^{u \to v}$ is the work-conserving rate of channel $u \to v$. For a channel $X \to Y$ in P, its work conserving rate is:

$$R_W^{X \to Y} = C_P * \frac{R_G^{X \to Y}}{\sum_{u \to v \in P} R_G^{u \to v}} \quad (4)$$

It means that all channels in P share spare capacity C_P in a weighted fair sharing fashion, where the weight is set as the bandwidth guarantee.

Although our Trinity ensures that work-conserving traffic will not affect bandwidth guarantee traffic, sending too much work-conserving traffic may cause a large number of packet losses in the low priority queues on switches, which will result in TCP timeout and thus hurt the throughput of TCP flows.

To address this, we enable ECN in the low priority switch queues. On the end hosts, we let hypervisors monitor the congestion feedback of ECN marking. Specifically, hypervisors will maintain an estimation of the fraction of red packets that are marked with ECN (denoted as β) in each period for all channels. If a congestion-free channel is detected to be congested, it will be moved from P to Q. For any congestion-caution channel $X \to Y$ in Q, in each period, if β is non-zero, we reduce its work conserving rate in proportion to β in a manner similar to DCTCP [25], i.e.,

$$R_W^{X \to Y} = (1 - \beta/2) * R_W^{X \to Y} \quad (5)$$

If β is zero, it means that there is no congestion in the network. We then increase its work conserving rate as follows:

$$R_W^{X \to Y} = min(R_P, (1 + \alpha) * R_W^{X \to Y}) \quad (6)$$

Here R_P is the work-conserving rate this channel will be allocated if it is a congestion-free channel. A congestion-caution channel should get no more allocation than its share as a congestion-free channel. α is a factor used to control the aggressiveness of rate increase. If $R_W^{X \to Y} = R_P$ after updating rate, the hypervisor will move this channel back to P.

In a public cloud, we cannot assume all tenants support ECN in their transport layer protocols. To make our ECN-based solution practical: at the sender side, for all out-going packets, the hypervisor sets the ECN-capable bits in IP header to be true; at the receiver side, the hypervisor estimates the fraction of ECN marked incoming red packets for every VM-to-VM channel, and sends this estimation back to the corresponding hypervisor at the sender side periodically.

In addition, the hypervisor should also record whether a connection supports ECN. For a TCP connection, the hypervisor can know whether it supports ECN in 3-way handshakes. For those flows that disable ECN, to avoid disturbing the function of their transport layer protocols, the hypervisor will clear the ECN bits when delivering packets to upper layer.

Problem #2: Packet trapping. There exist scenarios that red packets can get trapped (starved) in the lower priority queue of a bottleneck switch. For example, initially the switch has spare bandwidth (by other tenants' bandwidth guarantees but currently not being used) for work-conserving traffic and thus some red packets get in the lower priority queue. Suddenly, the bandwidth guarantee packets of other tenants come back and occupy the bandwidth for a long duration. Then, the work-conserving red packets get trapped due to lower priority. As a consequence, the TCP sender of those red packets responds by retransmitting the packets repeatedly, and these retransmitted packets may get dropped persistently since the bottleneck queue is already full.

Solution: Reserving sufficient bandwidth headroom for work-conserving traffic can potentially address this problem, however it is a waste of bandwidth and thus not desirable.

We introduce a simple solution to this problem without bandwidth headroom. As mentioned above in rate control, the hypervisor at the receiver side will estimate the fraction of ECN marked incoming red packets for every VM-to-VM channel periodically. Then if a hypervisor does not receive any red packets for a VM-to-VM channel in the last period, it can send a message to inform the corresponding hypervisor at the sender side of the possible packet trapping. The hypervisor at the sender then checks how many red packets the source VM has sent out for this VM-to-VM channel in the last period. If the source VM does send out some red packets, it indicates packet trapping in the network. The hypervisor then sets the work-conserving rate R_W to a small value (e.g., 10Kbps), and marks this channel as congestion-caution channel.

Problem #3: Packet re-ordering. For a short flow, all of its packets are colored as green, there is no out-of-order problem. While for a long flow, due to instantaneous token availability in Token Bucket 1, the packets can alternate between green and red. It is possible that in the same long flow, some packets with smaller sequence numbers are colored as red as tokens run out in Token Bucket 1, while subsequent packets with larger sequence numbers are colored as green because new tokens are being generated. In such case, packet re-ordering could arise because red packets may experience longer queueing delay in the network and reach the destination later than green ones. This is detrimental to TCP throughput, by triggering window collapse and unnecessary retransmissions.

Solution: The solution to this problem is twofold. At the sender, we minimize the case that packets of a flow alternate from red back to green. Specifically, we introduce a color transition delay parameter τ: when there is a need to change the colors of packets from red to green, we defer the change by τ seconds. There are two benefits for this delay. First, it increases the chance that some other flows may come up and consume the tokens in Token Bucket 1 without packet re-ordering. Second, it decreases the chance that packet re-ordering happens with the flow itself because this has already reserved some additional time for the red packets to transmit. At the receiver, we adopt a re-sequencing buffer [26]–[28] to absorb possible out-of-order packets as shown in Fig. 2. More specifically, if a green packet p is received and some packets

p_i prior to it have not been received yet, Trinity puts p into the re-sequencing buffer and a timer is initiated. If all p_is are received at a time t before timeout, they are submitted to TCP receiver together with p immediately; Otherwise, the whole buffer is submitted when timeout.

IV. Implementation and Testbed Setup

A. Trinity Implementation

Trinity consists of two components on end-hosts: receiver RX processing and sender TX processing. As a prototype, we have implemented TX and RX processing as a Linux kernel module. We also implemented a ElasticSwitch-like kernel module following the description of [6]. The kernel module is located as a shim layer above the physical NIC driver in hypervisor, without touching network stacks and applications of tenant's VMs. We also developed an application to configure Trinity kernel module in user space. The application communicates with kernel module using IOCTL [29]. We now describe each component in detail.

Sender Trinity Module: The sender module consists of a hash based flow table and multiple TX contexts. The flow table is used for tracking per-flow state and packet classification. Its operations are as follows: 1) All of the outgoing packets are intercepted by NETFILTER hook at LOCAL_OUT and directed to the flow table [30]. 2) Each flow in the flow table is identified by the 5-tuple: source/destination IPs, source/destination ports and protocol. When a packet comes in, we identify its corresponding flow entry (or create a new entry) and update the amount of bytes sent[1]. 3) Based on the bytes sent information, we classify packets and direct them to FIFO queues of corresponding TX contexts.

We allocate a TX context for each VM-to-VM pair. The TX context maintains basic TX information and a rater limiter. Unlike traditional token bucket rate limiter, our rate limiter has two associated FIFO queues, a timer, two rates (R_W and R_G) and corresponding two kinds of tokens. The packets of short-flow class and long-flow class are segregated by two FIFO queues. To enforce accurate rates over short timescales and avoid long delay to short-flow packets, we use Linux high-resolution kernel timer, HRTIMER [31], for our rate limiters. Once the timer fires, we update two kinds of tokens and begin packet scheduling. The packets from short-class FIFO queue has the high priority to be dequeued but they can only consume tokens for bandwidth guarantee traffic. After scheduling short-flow class packets, the packets from long-flow class can be dequeued and they can consume both of two kinds of tokens. The dequeued packets consuming different kinds of tokens will be marked with different Different Service Code Point (DSCP) values and enqueued to different priority queues in network switches. To make ECN fully effective for every packet regardless of their protocols, we set ECN-capable (ECT) codepoint to every dequeued packet.

[1]Tenants may establish persistent TCP connections to reduce connection establishment overhead and keep delivering short messages over these connections. These persistent connections will be eventually be assigned to the low priority by Trinity after long time. We can periodically update flow states based on more comprehensive network behaviors. For example, when a flow idles for some time, we can reset the bytes sent of this flow back to 0.

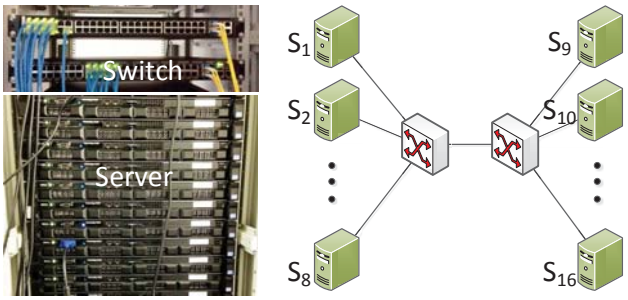
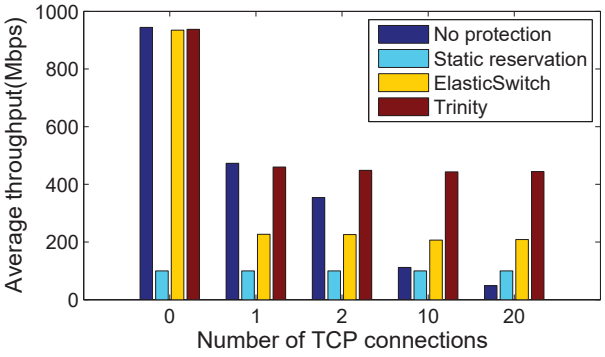


Fig. 3: Trinity testbed.

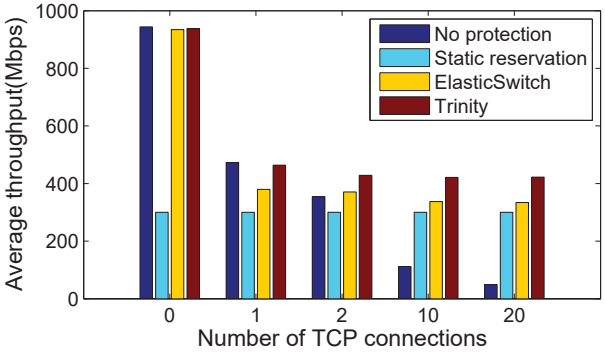
Receiver Trinity Module: The receiver modules consists of multiple RX contexts and a control packet generator. We pre-allocate a RX context for each VM-to-VM communication pair. The RX context tracks the VM-to-VM pair's receive traffic and measures incoming throughput. In each control interval, the RX context calculates the fraction of ECN marking packets and delivers this to source VMs using special feedback packets. Similar to EyeQ [8], our feedback packet is a special minimum sized IP packet (64 bytes) with a special unused IP protocol number (143 in our implementation). We encode the ECN fraction in the IP identification field. Since we only generate a packet for each VM-to-VM pair every control interval, the feedback traffic consumes limited network bandwidth. Considering a VM concurrently receiving traffic from 100 VMs, the feedback traffic only consumes ∼50Mbps throughput over the control interval of 1ms. Furthermore, we can also piggyback the feedback information on packets back to the source VM. To achieve low latency for control messages, the feedback packets will be marked with DSCP of bandwidth guarantee traffic and sent out without going through rate limiters. To not disturb tenant's network stacks, the RX context also clears any possible ECT and ECN marks in incoming packets when a tenant disables ECN function.

B. Testbed Setup

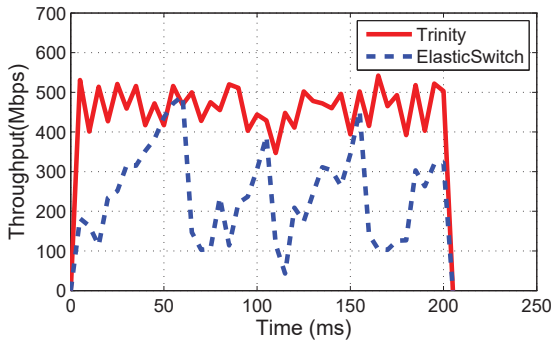
To evaluate Trinity, we build a dumbbell testbed with 16 servers connected to 2 Pronto-3295 48-port Gigabit switches as shown in Fig. 3. We configure strict priority queueing and per-queue ECN marking on switches. The shared buffer is enabled on our switches by default. With per-queue ECN marking, each queue has its own marking threshold and performs ECN marking independently to other queues. Packets are classified into different priority queues based on their DSCP values. Each server is a Dell PowerEdge R320 with a 4-core Intel E5-1410 2.8GHz CPU, 8G memory, a 500GB hard disk, and a Broadcom BCM5719 NetXtreme Gigabit Ethernet NIC. Each server runs Debian 6.0-64bit with Linux 2.6.38.3 kernel. Due to the limited number of CPU cores in our physical servers, we emulate multiple VMs by creating multiple virtual network interfaces with different IP addresses



(a) Both tenants have 100Mbps guarantees.



(b) Both tenants have 300Mbps guarantees.



(c) Throughput of VM A2 (100Mbps guarantees).

Fig. 4: Average throughput of VM A2 when the number of TCP connections used by tenant B varies.

to avoid virtualization overheads. In our experiments, each tenant has its own virtual subnets.

V. Evaluation

We evaluate Trinity using testbed experiments. Our evaluation centers around two key questions:

- **Does Trinity have any tradeoff between bandwidth guarantee and work conservation?** By comparing to three other schemes: no protection, static reservation and ElasticSwitch, we show that Trinity can accurately provide minimum bandwidth guarantees while at the same time enabling VMs with large bandwidth demand to fully utilize spare link capacity. Specifically, Trinity outperforms ElasticSwitch by 20.88%–53.06% in terms of average throughput under different settings.
- **Can Trinity deliver low latency for short flows and benefit their flow completion time (FCT)?** We evaluate the scenarios where short flows coexist with long flows. Our results show that, compared to ElasticSwitch, Trinity improves the FCT by 22%-33% on average and 68%-71% at the 99th percentile for 1KB short flows; furthermore, it reduces the FCT by 21%-38% on average and 62%-70% at the 99th percentile for 20KB short flows.

Schemes compared: We mainly compare Trinity against ElasticSwitch [6], static reservation (Oktopus-like [1]) and no reservation in our testbed. Among them ElasticSwitch is our closest work to compare. Qualitative analysis of other schemes like Gatekeeper [14] and EyeQ [8] shows that those approaches cannot provide guarantees when the network core is congested, so we exclude them in our testbed experiments.

Parameters: The rate control interval is set to 5ms. We set ECN marking threshold to be 30KB as DCTCP [25] recommends. For the rate control algorithm of ElasticSwitch [6], we also use its recommended algorithm.

A. Bandwidth Guarantees and Work Conservation

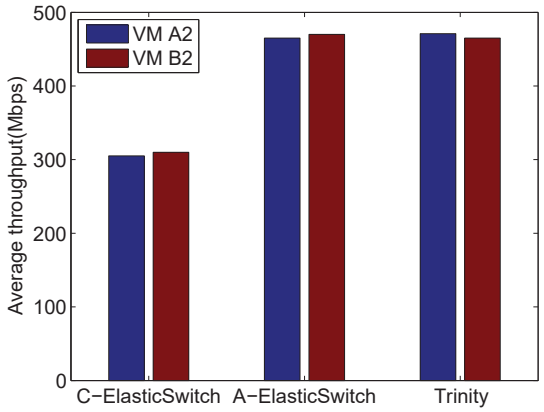
We show that Trinity can provide bandwidth guarantee while achieving good work conservation when multiple tenants are competing for the same bottleneck link.

Many connections vs one connection: In this experiment, there are four VMs (A1, A2, B1 and B2) of two tenants A and B sharing a same bottleneck link. VM A1 on server S_1 sends traffic to VM A2 on server S_9 using one TCP connection, while VM B1 on server S_2 sends traffic to VM B2 on server S_{10} using different numbers of TCP connections.

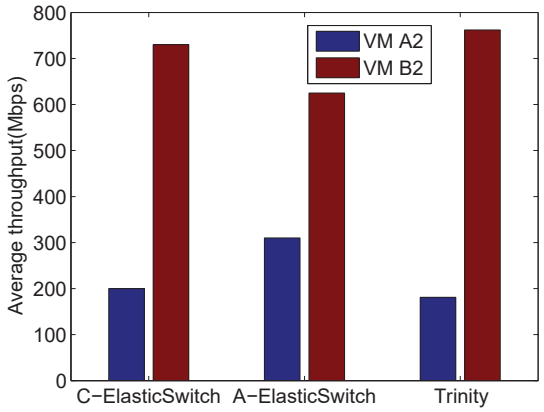
We measure the throughput at VM A2 under four schemes: no protection, static reservation [1, 9], ElasticSwitch, and Trinity. In Fig. 4a, both tenants are provisioned with 100Mbps guarantees. In Fig. 4b, both tenants are provisioned with 300Mbps guarantees.

From the results, we make the following two observations: 1) No protection does not provide any bandwidth guarantee as link capacity is shared among all TCP connections. Static reservation provides minimum bandwidth guarantee, but does not utilize any spare bandwidth. ElasticSwitch provides bandwidth guarantees and utilizes part of the spare bandwidth. In contrast, Trinity not only provides bandwidth guarantee but also fully utilizes all the spare bandwidth. In terms of the average throughput, Trinity outperforms ElasticSwitch by 20.88% to 53.06% in different bandwidth guarantee settings. 2) ElasticSwitch wastes around 50% of the spare bandwidth. For instance, in Fig. 4a, when reserving 20% of the link capacity on the bottleneck link as bandwidth guarantees, ideally, VM A2 should achieve around 500Mbps throughput on average. However, under ElasticSwitch, the average throughput of VM A2 is only about 230Mbps.

We further look into the reason behind it by measuring the throughput of VM A2 every 5ms (i.e., rate control interval).



(a) Both tenants have 200Mbps guarantees.



(b) A and B have 150Mbps and 750Mbps guarantees, respectively.

Fig. 5: Average throughput under 3 schemes.

In Fig. 4c, we show the result of the case where there are 10 TCP connections between VM B1 and B2. As illustrated, under ElasticSwitch, the throughput of VM A2 drops back to minimum guarantee as long as it senses congestion. Due to this conservative rate control, ElasticSwitch can only utilize about half of the spare bandwidth on average. On the other hand, our Trinity achieves nearly ideal throughput at the granularity of millisecond, this is because Trinity adjusts rate for each active VM pair based on a fine-grained estimation of the network congestion as introduced in §III-C.

A follow-up question may arise: can ElasticSwitch provide bandwidth guarantee and achieve good work-conservation by using the rate control algorithm of Trinity? We answer this question in the following experiment.

Tradeoff between bandwidth guarantees and work conservation: We denote ElasticSwitch with original rate control as conservative ElasticSwitch (C-ElasticSwitch), and with Trinity's rate control as aggressive ElasticSwitch (A-ElasticSwitch). In this experiment, we use the same scenario as above, and measure the average throughput of VM A2 and B2 under C-ElasticSwitch, A-ElasticSwitch and Trinity. The number of TCP connections between VM B1 and B2 is set to 10.

The results in Fig. 5 show: 1) C-ElasticSwitch provides bandwidth guarantees but cannot fully utilize spare bandwidth as shown in Fig.5a; 2) A-ElasticSwitch achieves good work conservation, but fails to provide bandwidth guarantees as shown in Fig.5b; 3) Trinity provides accurate bandwidth guarantees while achieving good work conservation in both cases.

The takeaway of this experiment is that: 1) There is a tradeoff between bandwidth guarantee and work-conservation. Pure end-to-end solutions are difficult to achieve both goals simultaneously. 2) In-network prioritization with priority queueing is key to eliminating this tradeoff.

	Trinity		ElasticSwitch	
Flow size	1KB	20KB	1KB	20KB
Average FCT(us)	212	857	272	1083
99th percentile FCT(us)	274	1104	857	2878

TABLE II: FCT of short flows (60% of link capacity is reserved as guarantees).

	Trinity		ElasticSwitch	
Flow size	1KB	20KB	1KB	20KB
Average FCT(us)	219	878	328	1413
99th percentile FCT(us)	291	1218	1002	3997

TABLE III: FCT of short flows (100% of link capacity is reserved as guarantees).

B. Low latency for short flows

We show that Trinity can deliver low latency for short flows when short flows coexist with long flows.

Tradeoff between low latency and work conservation: It has been shown that, when most of the link capacity are reserved as guarantees, ElasticSwitch is work-conserving. However, we will show that there is actually a tradeoff between low latency and work-conservation. In this experiment, we have 6 VMs A1, A2, B1, B2, C1 and C2 of three tenants A, B and C. They are hosted on servers S_1, S_9, S_2, S_{10}, S_3 and S_{11}, respectively.

In this experiment, VM A1 sends 1KB or 20KB short flows to A2 periodically, and in the meantime, VM B1 and C1 send long flows to VM B2 and C2, respectively. To explore the tradeoff between low latency and work-conservation, we study two cases: 1. Three tenants are all provisioned with 200Mbps guarantees on the bottleneck link, and thus we have 400Mbps spare bandwidth; 2. Tenant A is provisioned with 200Mbps guarantee on the bottleneck link. Tenants B and C are both provisioned with 400Mbps guarantees on the bottleneck link. Hence no spare bandwidth is left in this case.

For case 1, the results are shown in Table II. For case 2, the results are shown in Table III. From the results, we observe that: 1) Compared to ElasticSwitch, Trinity reduces the FCT by $22\% - 33\%$ on average and by $68\% - 71\%$ at the 99th percentile for 1KB short flows; furthermore, it reduces the FCT by $21\% - 38\%$ on average and by $62\% - 70\%$ at the 99th percentile for 20KB short flows. 2) Although ElasticSwitch is work-conserving when 100% of link capacity is reserved as guarantees, it is at the cost of sacrificing latency of short flows. By comparing the results in Table II with that in Table III, we

	Trinity		ElasticSwitch	
Flow size	1KB	20KB	1KB	20KB
Average FCT(us)	252	1105	1378	4989
99th percentile FCT(us)	302	1574	2160	7431

TABLE IV: FCT of short flows when short flows are mixed with long flows on end-host (60% of link capacity is reserved as guarantees).

can see that, under ElasticSwitch, the FCT increases by $17\% - 21\%$ on average, and by $30\% - 39\%$ at the 99th percentile. In contrast, under Trinity, we do not observe any significant increase on the FCT.

The takeaway of this experiment is two-fold: 1) There is a tradeoff between low latency and work conservation. Pure end-host based solutions are difficult to achieve both goals simultaneously. 2) By letting packets of short flows receive high priority in the network, we can well address this tradeoff and improve the FCT of short flows significantly.

Short flows mixed with long flows on end-host: If a VM is sending both long flows and short flows to a remote VM, then the congestion on end-host cannot be simply ignored anymore. Recall that in the design of Trinity, packets of short flows have higher priority to consume tokens in Token Bucket 1 over packets of long flows. We show that this mechanism can reduce end-host delay of short flows when short flows are mixed with long flows on end-host.

In this experiment, we change the scenario above by letting VM A1 send both short flows and long flows with unbounded demand to VM A2. In Table IV, we show the results of the case when only 60% of link capacity is reserved as guarantees.

From the results, we can find that: compared to ElasticSwitch, Trinity reduces the FCT by 82% on average and by 86% at the 99th percentile for 1KB short flows; Furthermore, it reduces the FCT by 78% on average and by 79% at the 99th percentile for 20KB short flows. This implies that Trinity can reduce both in-network delay and end-host delay.

VI. Conclusion

This paper presented Trinity, a simple yet effective solution that provides triple properties: bandwidth guarantees, work conservation and low latency simultaneously in the cloud. By differentiating traffic at the end and enforcing prioritization in the network, Trinity eliminates the tradeoff between providing bandwidth guarantees and being work-conserving, while achieving low latency for short flows. We have implemented Trinity using commodity switches and servers, and demonstrated its performance with testbed experiments.

VII. Acknowledgements

This work is supported by the Hong Kong RGC 26200014, 16203715, 613113, CRF C7036-15G and the National Basic Research Program of China (973) under Grant No.2014CB340303.

References

[1] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *SIGCOMM*, 2011.

[2] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding long tails in the cloud," in *NSDI*, 2013.

[3] A. Li, X. Yang, S. Kandula, and M. Zhang, "Cloudcmp: Comparing public cloud providers," in *IMC*, 2010.

[4] J. Schad, J. Dittrich, and J. Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," in *VLDB*, 2010.

[5] Y. Zhang, C. Guo, D. Li, R. Chu, H. Wu, and Y. Xiong, "Cubicring: Enabling one-hop failure detection and recovery for distributed in-memory storage systems," in *NSDI'15*.

[6] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos, "Elasticswitch: practical work-conserving bandwidth guarantees for cloud computing," in *SIGCOMM 2013*.

[7] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message completion time in the clouds," in *SIGCOMM*, 2015.

[8] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg, "Eyeq: practical network performance isolation at the edge," in *NSDI 2013*.

[9] C. Guo, G. Lu, H. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "Secondnet: A data center network virtualization architecture with bandwidth guarantees," in *CoNEXT*, 2010.

[10] D. Xie, N. Ding, Y. C. Hu, and R. Kompella, "The only constant is change: Incorporating time-varying network reservations in data centers," in *SIGCOMM*, 2012.

[11] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha, "Sharing the data center network," in *NSDI'11*.

[12] T. Lam, S. Radhakrishnan, A. Vahdat, and G. Varghese, *NetShare: Virtualizing data center networks across services*, 2010.

[13] L. Popa and et.al., "Faircloud: Sharing the network in cloud computing," in *SIGCOMM*, 2012.

[14] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes, "Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks." in *WIOV*, 2011.

[15] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma, "Application-driven bandwidth guarantees in datacenters," in *SIGCOMM 2014*.

[16] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," in *SIGCOMM 2013*.

[17] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," in *NSDI 2015*.

[18] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. Liu, and F. Dogar, "Friends, not foes - synthesizing existing transport strategies for data center networks," in *SIGCOMM 2014*.

[19] S. Ghemawat, H. Gobioff, and S. Leung, "The google file system," in *SOSP*, 2003.

[20] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, pp. 107–113, 2008.

[21] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *SIGCOMM'11*.

[22] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *SIGCOMM '89*.

[23] I. Stoica and H. Zhang, "Providing guaranteed services without per flow management," in *SIGCOMM '99*.

[24] I. Stoica, S. Shenker, and H. Zhang, "Core-stateless fair queueing: A scalable architecture to approximate fair bandwidth allocations in high-speed networks," in *SIGCOMM '98*.

[25] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *SIGCOMM 2010*.

[26] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," in *SIGCOMM 2015*.

[27] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley, "How hard can it be? designing and implementing a deployable multipath tcp," in *NSDI'12*.

[28] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. Maltz, "Per-packet load-balanced, low-latency routing for clos-based data center networks," in *CoNEXT 2013*.

[29] "Linux ioctl," http://man7.org/linux/man-pages/man2/ioctl.2.html.

[30] "Linux netfilter," http://www.netfilter.org.

[31] "Hrtimer," https://www.kernel.org/doc/Documentation/timers/hrtimers.txt.