Towards Shorter Task Completion Time in Datacenter Networks

Yuchao ZHANG*, Ke XU*, Haiyang WANG†, Meng SHEN‡
*Department of Computer Science & Technology, Tsinghua University
†Department of Computer Science, University of Minnesota at Duluth
‡Department of Computer Science, Beijing Institute of Technology
Email:zhangyc14@mails.tsinghua.edu.cn, xuke@mail.tsinghua.edu.cn,
haiyang@d.umn.edu, shenmeng@bit.edu.cn

Abstract—**Datacenters are now used as the underlying infrastructure of many modern commercial operations, powering both large Internet services and a growing number of data-intensive scientific applications. The tasks in these applications always consist of rich and complex flows which require different resources at different time slots. The existing data center scheduling frameworks are however base on either task or flow level metrics. This simplifies the design and deployment, but hardly unleashes the potentials of obtaining low task completion time for delay sensitive applications.**

In this paper, we show that the performance (e.g., tail and average task completion time) of existing flow-aware and task-aware network scheduling is far from being optimal. To address such a problem, we carefully examine the possibility to consider both task and flow level metrics together and present the design of TAFA (Task-Aware and Flow-Aware) in data center networks. This approach seamlessly combines the existing flow and task metrics together while successfully avoids their problems as flow-isolation and flow indiscrimination. The evaluation result shows that TAFA can obtain a near-optimal performance and reduce over 35% task completion time for the existing data center systems.

I. Introduction

Nowadays, data centers have become the cornerstones of modern computing infrastructure and one dominating paradigm in the externalization of IT resources. The data center tasks always consist of rich and complex flows which traverse different parts of the network at potentially different times. To minimize the network contention among different tasks, task serialization was widely suggested. This approach applies task level metric and aims to serve one task at a time with synchronized network access. While serialization is a smart design to avoid task level interference, our study shows that the flow level network contention within a task can however largely affect the task completion time. This prolongs the tail as well as the average task completion time and unavoidably reduces the system applicability to serve delay-sensitive applications.

In this paper, we for the first time investigate the potential to consider both flow level and task level interference together for data center task scheduling. We provide the design of TAFA (task-aware and flow-aware) to obtain better serialization and minimize possible flow and task contentions. TAFA adopts dynamic priority adjustment for the task scheduling. Different from FIFO-LM [1], this design can successfully emulate Shortest-Task-First scheduling while requires no prior knowledge about the task. Further, TAFA gives a more reasonable and efficient approach to reduce task completion time by considering the relationship among different flows in one task, rather than treating them all the same. With this intelligent adjustment in flow level, TAFA provides the shorter flow waiting time, and leading to earlier finish time.

In short, this paper mainly makes two contributions, which are described as follows.

Firstly, we point out that the flow contentions in one task will also make system performance degrade, and a task-aware scheme which ignores flow relationship will achieve longer completion time. We give a simple example in Section. III and analyze the disadvantages of leaving out information on flow contention.

Secondly, we design an scheduling algorithm called TAFA, which can achieve both task-awareness and flow-awareness. Task-awareness ensures short tasks are prioritized over long ones, and enables TAFA to emulate STF scheduling without knowing task size beforehead. Flow-awareness optimizes scheduling order, and achieves shorter task completion time.

The rest of this paper is organized as follows. Based on the background and related work given in Section II, we motivate this paper in Section III and introduce the main system framework TAFA in Section IV. Section V shows the simulation results. Finally, we concludes this paper and points out the future work in Section VI.

II. Background and Related Work

Although the ever-increasing used data center networks have configured high bandwidth and calculative ability, the task completion time still can be reduced to a large extent [1]. Tasks' completion times in DCNs (i.e., DCN is often multiplexed by many tasks) are depend on many factors, such as the bandwidth allocation [2], traffic variability [3] and VM performance management [4], [5]. In this section, we describe the nature of today's datacenter transport protocols, either

This work has been supported in part by NSFC Project (61170292, 61472212), 973 Project of China (2012CB315803), 863 Project of China (2013AA013302, 2015AA010203), EU MARIE CURIE ACTIONS EVANS (PIRSES-GA-2013-610524) and multidisciplinary fund of Tsinghua National Laboratory for Information Science and Technology. H. Wang's work was supported by Chancellors Small Grant and Grant-in-aid programs from the University of Minnesota.

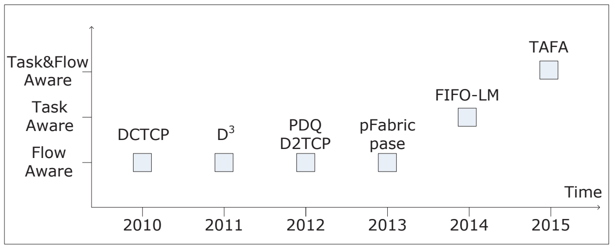


Fig. 1: Brief development history of transport protocols.

flow-aware or task-aware, and how does the awareness of different levels isolate with each other. As a result, although flow completion time or task completion time seems to reduce obviously, flow-aware protocols are blind to task level, and vice versa. In particular, good flow-level awareness can help make task completion time shorter, while good task-level awareness can help flows cooperate harmonically.

Fig. 1 shows the development history of scheduling protocols, from DCTCP (2010) to FIFO-LM (2014), which can be categorized into two broad classes, flow-aware and task-aware, both of which do advance the state-of-art technique. We'll give a brief description and explanation to this progress.

As the founder of many flow-aware TCP-like protocols, DCTCP [6] leverages Explicit Congestion Notification (ECN) in the network to provide feedback to end hosts. Experiments show that DCTCP delivers better throughput than TCP while using 90% less buffer because it elegantly reduce the queue length. However, it is a deadline-agnostic protocol that equally throttles all flows, irrespective of whether their deadline are near or far, so it may be less effective for the Online Data Intensive applications (OLDI) [7]. Motivated by this observations, D^3 [8] use explicit rate control for the datacenter environment according to flow deadlines. D^3 can determine the rate needed to satisfy the flow deadline when having the knowledge of flow's sizes and deadlines. Although it outperforms TCP in terms of short flow latency and burst tolerance, D^3 has the practical drawbacks of requiring changes to the switch hardware, which makes it not able to coexist with legacy TCP [7]. Deadline-Aware Datacenter TCP (D^2TCP) [7] is a deployable transport protocol compared to D^3. Via a gamma-corretion function, D^2TCP uses ECN feedback and deadlines to modulate the congestion window. Besides, D^2TCP can coexist with TCP without hurting bandwidth or deadline. Preemptive Distributed Quick (PDQ) flow scheduling [9] is designed to complete flows quickly and meet flow deadlines, and it builds on traditional real-time scheduling techniques: Earliest Deadline First and Shortest Job First, which help PDQ outperform TCP, RCP [10] and D^3 significantly. pfabric [11] decouples flow scheduling from rate control. Unlike the protocols above, in pfabric, each flow carries a single priority number set independently, according to which switches execute a scheduling/dropping mechanism. Although pfabric achieves near-optimal flow completion time, it does not support work-conservation in a multi-hop setting because end-hosts always send at maximum rate. To make these flows back-off and let a lower priority flow at a subsequent hop, we need a explicit feedback from switches, i.e., a higher layer control.

From a network perspective, tasks in DCNs typically comprise multiple flows, which traverse different servers at different times. Treating flows of one task in isolation will make flow-level optimizing while hurting task completion time. To solve this boundedness in flow-aware schemes, task-aware protocols have been proposed to explicitly take the higher layer information into consideration.

A task-aware scheduling was proposed by Fahad R. Dogar in [1]. Using First-In-First-Out to reduce both the average and the tail task completion time, Dogar implement First-In-First-Out with Limited Multiplexing (FIFO-LM) to change the level of multiplexing when heavy tasks are encountered, which can help heavy tasks not being blocked, or even starved. But as we all know, FIFO is not the most effective method to reduce average completion time nomatter in flow-level or task-level, and the simple distinguish just between elephant tasks and mouse tasks is in coarse granularity as [12] said that the DCNs should be more load, more differentiation. Further, [13] and [14] give methods that can ensure user-level performance guarantee.

Without cross layer cooperation, these protocols have great blindness to each other, making scheduling inefficient. For our object to achieve advantages in both task-aware and flow-aware, we praise TAFA with the idea of co-existence. What's more, this work should perform well even in the real multiple resource sharing environment.

III. Motivation

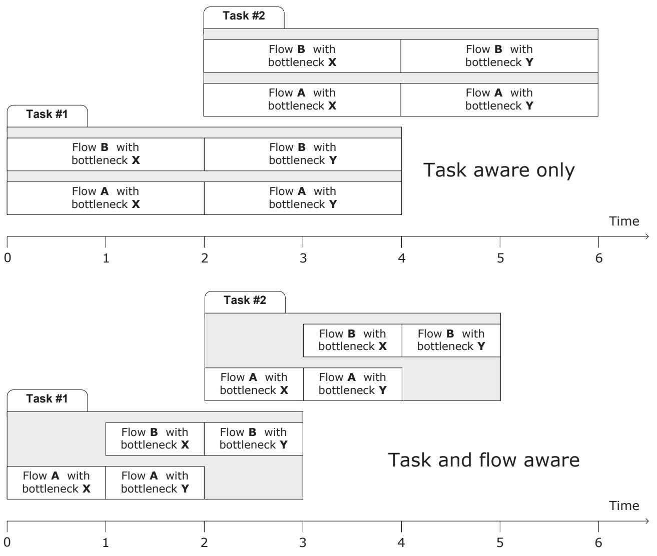
Scheduling policies determine the order in which tasks and flows are scheduled across the network. In this section, we'll show how do flow-aware and task-aware waste resources when being applied separately, and we motivate TAFA by combining the two layer awareness together, making task completion time 2 times shorter than flow-aware scheme and 20% shorter than task-aware FIFO-LM.

Before giving a specific example, we introduce the definition of flow and task:

Definition 1. Flow
Flow is a fundamental unit of a basic action, and a series of flows can make up one task to accomplish a specific request.

Definition 2. Task
Tasks consist of multiple flows, and can response to a user request completely. Applications in DC perform rich and complex tasks (such as executing a search query or loading a user's required page).

In addition, flows traverse different parts of the network at potentially different times, and there are tight relationship among flows, such as sequencing and parallelization. The TCT of a particular task is depended on the finish time of the last flow belonged to this task. With the concept of task and flow, we consider a small cluster with CPU and link resources, and there 2 tasks each has two steps that are separated by



Task #2
Flow **B** with bottleneck **X**
Flow **B** with bottleneck **Y**
Flow **A** with bottleneck **X**
Flow **A** with bottleneck **Y**
Task #1
Flow **B** with bottleneck **X**
Flow **B** with bottleneck **Y**
Flow **A** with bottleneck **X**
Flow **A** with bottleneck **Y**
Task aware only
Time
0
1
2
3
4
5
6
Task #2
Flow **B** with bottleneck **X**
Flow **B** with bottleneck **Y**
Flow **A** with bottleneck **X**
Flow **A** with bottleneck **Y**
Task #1
Flow **B** with bottleneck **X**
Flow **B** with bottleneck **Y**
Flow **A** with bottleneck **X**
Flow **A** with bottleneck **Y**
Task and flow aware
Time
0
1
2
3
4
5
6

Fig. 2: Distilling the benefits of both task and flow aware scheme (TAFA) over task-aware.

a barrier. This situation resembles map-reduce; map tasks are CPU intensive while reduce tasks are network-intensive, like the example in [15]. There are two flows in each task, and each flow has two stages. The CPU processing stage need 2 unit of CPU time and the network processing stage consumes 2 unit of link time. Further, network processing stage can not begin until CPU processing stage finishes.

Flow-aware. Consider the flow-aware fair sharing (FS) scheduling scheme. When assuming all flows are infinitely divisible, scheduling all 4 map flows would fully use up the cluster's CPU for the first 4t, and then 4 reduce flows become runnable and the cluster will fairly allocate link to them. Each flow gets only 1/4 of resources due to contention, keeping link busy for another 4t. Thus, both of the two tasks will finish scheduling at time 8t.

Task-aware. Obviously, flow-aware fair sharing is not a good choice here. Now we consider the task-aware scheduling FIFO-LM in [1]. According to FIFO, the two flows of A should be scheduled first, and with the same *task-id*, these CPU stages of the two flows share the CPU fairly, so the this phase will occupy 2t of CPU, then at 2t, network stage of $task\#1$ and CPU stages of $task\#2$ can start. At 4t, network stages of $task\#2$ can start. The schedule is shown in the upper part of Fig. 2 (where bottleneck x denotes CPU and bottleneck y denotes network). The 2 tasks now finish at time 4t, 6t. Average completion time reduce from 8t to 5t compared with FS.

What we should pay attention to is that this result is *far from optimal* to a large extent, and we will show how to reduce TCT over task-aware scheme.

The core idea is to make task-aware scheduling scheme flow-aware. As described in Definition.1, flow completion time is closely relative to task completion time. To reduce task completion time, we should distinguish different flows of one task, because reducing average flow completion time will also shorten task completion time. So here we discard the fair sharing method among flows [16], [17] within one step, and make the cluster serve flows one by one (later in section IV, we will introduce the FQH to decide the flow serving order). As shown in the lower part of Fig. 2, the CPU stages of the two flows are not served simultaneously, but making one of them finish processing early, so the corresponding reduce phase can start at 1t (while in FIFO, this reduce phase start at 2t). Along this line, the flows of $task\#2$ can also be scheduled in advance. Thus, the finishing times of the two tasks are: 3t, 5t. The average TCT is a half of FS (4t $\leftarrow$ 8t) and 20% less than task-aware (4t $\leftarrow$ 5t). From this simple scenario, we can see that just in simple one-by-one order, we can reduce average completion time by 20% over FIFO-LM.

The above examples highlight that isolate flow-awareness and task-awareness are inefficient and do not optimize task completion time, which indicates that disregarding cross layer relevant awareness leads to the waste of resources. Then we'll show how TAFA outperform the state-of-art protocols in Section.IV.

IV. TAFA – Task Aware & Flow Aware

In this section, we describe TAFA's scheduling heuristic, combining both task-aware and flow-aware together to make more preferable and reasonable scheduling. As task completion time depends on the last flow's finish time, to determine the order to minimize TCT, two questions should be clarified. One is the task schedule order, which is a well known NP-hard problem [1], the other is flow completion time, which can reduce task completion time. We develop heuristics that enable STF with no prior knowledge using commodity switches (IV-A). To give the detailed flows scheduling method for reducing completion time, we design FQH algorithm in IV-B.

A. Task-awareness

The task scheduling policy determines the order in which tasks are scheduled across the network, while one task consists of multiple flows, the original priorities of these flows are depended on task order.

In this subsection, we focus on task priority. At a high level, TAFA main mechanisms include priority queuing and ECN marking, which can adjust the priority of tasks dynamically according to the bytes they have sent.

1) End-host Operations: In TAFA, end-hosts are responsible for two things: one is to generate *task-id*, the other is to response to the rate control according to the marking punched by switches.

For the former, end-host assigns a globally unique identifier (*task-id*) for each task. When an end-host produces a new task, each flow of this task will be tagged with the *task-id*. To generate this id, each host maintains a monotonically increasing counter. Unlike PIAS in [18], in which tags are carried by packets, TAFA allow flows carry these tags, making tasks quite clear to the loading in switch; Unlike Task-aware in [1], which just separates heavy tasks from short ones, we catalog tasks to multiple priorities, which will be explain in IV-A2.

For the rate control, we should first explain the relationship among multiple flows. As tasks are consist of number of complex flows, which will traverse different servers at different times to respond to a user request, not all of them are active at the same time. Though there are huge diversity among different applications, according to their communication patterns, the relationship of flows can be grouped into three categories:

- For parallel flows, they may be a request to a cluster of storage servers, the flows of these tasks are parallel;
- For a sequential access task, the flows order is sequential, making the flows in one task should be scheduled one by one;
- For a partition-aggregate task, which may involve tens and hundreds of flows, the flow order are in particular importance.

However, PIAS has a serious weakness that it ignores the relationship of flows in the same task, and each flow has a adjustable priority in isolation. For a sequential access task, if the previous flow is heavy, then its priority will be gradually demoted, while the subsequent short flows with higher priorities will finish earlier. But the subsequent results are useless because of the lack of the previous result.

To avoid this situation, TAFA adjusts the sending rate and order according to the marking punched by switches whenever necessary. The detailed scheme will be describe in IV-B.

2) Switch Operations: Two functions are built-in in TAFA switches , priority queuing and ECN marking.

For priority queuing, as end-hosts tag *task-id* for each flow of each task, which is used to depend the order of this flow, so the only thing switches should do is to maintain the queue. Flows are waiting in different priority queues (more than 2) in switches. Whenever a link is idle or has enough resource to schedule a new flow, the first flow in the highest non-empty priority queue is served. With the bytes one task rising, the priority of this task was gradually demoted, so the flows of this task are also affected by the demotion, making these flows tagged a lower priority, and should wait longer in the queue.

For ECN marking (a feature already available in modern commodity switches [6]), flows will be marked with Congestion Experienced (CE) in the network to provide multi-bit feedback to the end-hosts. So TAFA employs a very simple marking scheme at switches, and there is only one parameter, the marking threshold, Υ. If the bytes of one task is greater than Υ, the flows of this task is marked with CE, and not marked otherwise. This ECN marking can notify end-host to demote its priority. The end-hosts whose flows are marked with ECN should tag their flows a lower priority than the current one. This feedback scheme can ensure that the tasks with less and shorter flows can be scheduled earlier than that with much more and longer flows. Using this Short Task First scheme, we can not only reduce the TCT, but also ensure heavy tasks not being starved.

3) Multiple priority queues: As the key problem in flow-level scheduling [18] is the determining of the threshold, Υ, here we give a vector of threshold Υ (consist of $\Upsilon_1, \Upsilon_2, ..., \Upsilon_{\tau-1}$). Where τ is the number of priority queues, and Υ_i is the threshold of priority queue i, i.e., in the ith priority queue. When the accumulated bytes of one task is greater than Υ_i, the flows of this task will be marked with ECN, and this task should be demoted to the lower priority queue.

The advantage of threshold vector than just one threshold is in the demotion process, and when there is only one threshold, it can not manage any bursts. Before explaining the reason, we now consider two flow size distributions [11], the first distribution is from a datacenter supporting web search [6], and the other distribution is from a cluster running large data mining jobs [19]. According to the analyse by Alizadeh, these two workload are a diverse mix of small and large flows. Over 95% of all bytes come from 30% of the flows in web search, while more than 95% of the bytes are from 4% flows and 80% of the flows are less than 10KB in data mining workload.

These above analyse introduces an important extreme case. Assume there are plenty of small tasks each produces plenty of flows, at the beginning, the bytes have been sent of all task are set to 0. In traditional schemes, there are only two queues,so all flows enter the higher priority queue, as they are all short tasks, so they would stay in the higher priority queue, leading all flows being concurrent due to the disability to distinguish these flows. In that case, the stagewise threshold vector can work here, as there are more than two queues each with a threshold ($\Upsilon_1 < \Upsilon_2 < ... < \Upsilon_{\tau-1}$), the speed tasks are demoted into lower priority queues is different, and this scheme can finally tell these flows apart even all of them are short ones. So the threshold vector can avoid the concurrency of priority demoting. As the value of threshold set is related to actual flows, we will further show the robustness to traffic variations in the simulation section.

B. Flow-awareness

To detailed introduce TAFA, in this subsection, we will display how to make task-aware scheduling flow-aware. Like DCTCP and D^{2}TCP, we require that the switches support ECN, which is true in nowaday's datacenter switches. As Short Flow First (SFF) is known to be the most effective way to shorten flow completion time, we modulate the congestion window in a size-aware manner. When congestion occurs, long flows back off aggressively, and short flows back off only a little. With this size-aware congestion avoidance scheme, more flows can be scheduled at early time.

To explain the flow-awareness in TAFA, we start with D^{2}TCP and build size-awareness on top of it. Like DCTCP and D^{2}TCP, the sender maintains α, the estimated fraction of packets that are marked when the buffer occupancy exceeds the threshold K. α is updated every one RTT as follows:

$$\alpha = (1 - g) \times \alpha + g \times f \tag{1}$$

where f is the fraction of packets that were marked with CE bits in the last window of data, and $0 < g < 1$ is the weight given to new samples. As D^{2}TCP maintains d as the deadline

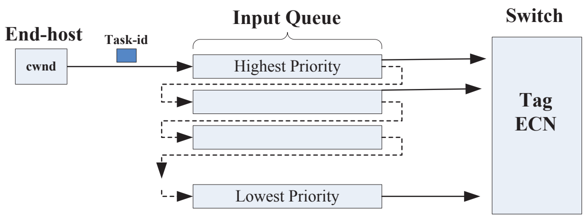


Fig. 3: TAFA Implementation.

imminence factor and larger d implies a closer deadline, so they design the penalty function as:

$$p = \alpha^d \tag{2}$$

The size of congestion window W is calculated by p. However, as [12] proposed, the flow rate control scheme should respect the differentiation principle, i.e., when traffic load becomes heavier, the differences between rates of different level flows should be increased. D^{2}TCP violates the principle and works badly in some scenarios [12]. So we design the penalty function in TAFA as:

$$p = \alpha / s \tag{3}$$

Where s is determined based on flow size information, and shown as follows:

$$s = \frac{S_{max}}{S_c} \tag{4}$$

Where S_{max} is an upper bound of flow size, and S_c is the remanent size for a flow to complete transmitting. Note that being a fraction, $\alpha \leq 1$ and $s \geq 1$, therefore, $p = \alpha/s = \frac{\alpha \times S_c}{S_{max}} \leq 1$. So we resize the congestion window W as follows:

$$W = \begin{cases} W \times (1-p) & \text{with congestion} \\ W + 1 & \text{without congestion} \end{cases} \tag{5}$$

With this simple algorithm, we compute $p = \alpha \times \frac{S_c}{S_{max}}$ and use it to adjust the congestion window size. If there is no congestion in the last window, W is increased by 1 like TCP, while any congestion occurs, W is decreased by a fraction of p. When all the flow sizes are equal, $\frac{S_c}{S_{max}} = 1$, severe congestion cause a full backoff similar to TCP and DCTCP.

For different flow size,

$$\begin{aligned} \frac{\partial(1-p)}{\partial(s)} &= \frac{\partial(1-\alpha/s)}{\partial(s)} = \frac{\alpha}{s^2} > 0 \\ \frac{\partial^2(1-p)}{\partial(s)\partial(\alpha)} &= \frac{1}{s^2} > 0 \end{aligned} \tag{6}$$

Which indicate that TAFA meets the PD principle in [12], i.e., the difference between two flows with different size would be increased when traffic loads become heavier.

C. Algorithm implementation

We now introduce the framework of "TAFA" algorithm, and Fig. 3 gives the abstract overview of TAFA.

There are several queues with different priorities for input tasks. When flows from a new task arrives, they are marked with highest priority, and enter the first queue. The length of flows from one task is added up to compare with Υ_k, which is the threshold of Queue k. When the total length exceeds Υ_k, the corresponding task is degrade from priority k to $k+1$, and the following flows from this task will enter Queue $k+1$ directly. As the scheduling begins from high priority queue to low priority queues, with this accumulated task length, TAFA can successfully emulate Shortest-Task-First scheduling while requires no prior knowledge about tasks.

Algorithm 1 TAFA

```
1:  Initialization
2:  while $F_i^j$ arrives do
3:      TimeStamp($F_i^j$)
4:      if $T_i$ == newTask then
5:          $P(T_i) \leftarrow 1$
6:      end if
7:      $P(F_i^j) \leftarrow$ CheckPriority($T_i$)
8:      EnQueue($P(F_i^j)$, $F_i^j$)
9:      $l(F_i^j) \leftarrow$ length($F_i^j$)
10:     AddLength($T_i$, $l(F_i^j)$)
11:     if length($T_i$) $> \Upsilon_k$ then
12:         Degrade($T_i$)
13:     end if
14: end while
15: FlowLevelScheduling
```

Algorithm 2 FlowLevelScheduling

```
1:  $Switch$:
2:  if Congestion occurs then
3:      Tag(ECN)
4:  end if
5:  SendBackToEndhost
6:  $Endhost$:
7:  if ECN == true then
8:      $s \leftarrow \frac{S_{max}}{S_c}$
9:      $\alpha \leftarrow (1-g) \times \alpha + g \times f$
10:     $p \leftarrow \alpha / s$
11:     $cwnd \leftarrow cwnd \times (1-p)$
12: else
13:     $cwnd \leftarrow cwnd + 1$
14: end if
15: SendBytes(cwnd)
```

We design TAFA to realize this process as shown in Algorithm.1, where F_i^j denotes the jth flow from $Task_i$, and $P(T)/P(F)$ denoted the priority of a task/flow. $EnQueue(P, F)$ is a function making F enter Queue P.

This algorithm can adjust the priority of tasks dynamically according to the task length, making short task be scheduled earlier than long ones without prior knowledge of task length. While many other researches do schedule under the assumption that task (or flow) size are already known, TAFA make this process more practical.

For flow-level adjustment, we design Algorithm.2 to take

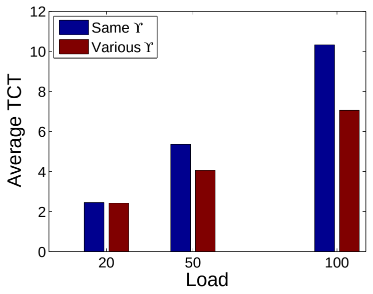


Fig. 4: Comparison between different demotion threshold.

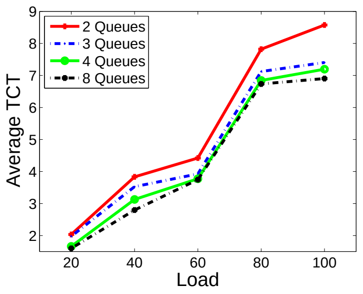


Fig. 5: TCT comparison of different number of queues.

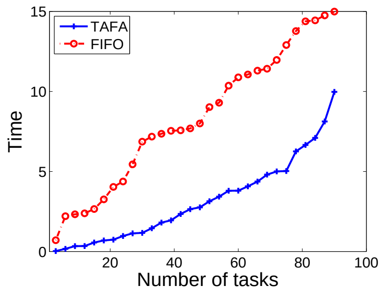


Fig. 6: Finish time of DropTail performance in TAFA and FIFO.

congestion extent and flow size into consideration. Switches will tag ECN when congestion occurs, and send back this signal within flows to end-hosts. When end-host receives flows with ECNs, it should adjust its congestion window $cwnd$. In TAFA, which aims at getting shorter completion time, the size of $cwnd$ is depended on the congestion extent α and flow size S_c, i.e., when traffic congestion α becomes serious, all flows should backoff in direct proportion to α, but for smaller flows, the penalty function p is smaller than that of longer ones, making the backoff slighter and could sending more flows. When end-host receives flows without ECNs, it acts as general TCP, and just increase $cwnd$ by 1. Then end-hosts will send packets according to this updated size of congestion window.

V. Evaluation

In this section, we evaluate the performance of TAFA using extensive simulations. To understand the performance in large scale, we do trace-driven simulations using trace from production clusters. Our evaluation consists of three parts. First, we evaluate TAFA's basic performance such as its TCT, degradation threshold, and number of priority queues. Building on these, we then show how TAFA achieves benefits compared to task-aware schemes in realistic datacenter networks running load, and finally we compare TAFA with flow-aware scheduling schemes on different scales.

A. Setup

Flows: We use the realistic workloads that have been observed in production datacenters, the web search distribution in [6] and the large data mining jobs in [19]. As [11] has given, the arrival pattern of flows is in poisson process. According to the empirical traffic distributions used for benchmarks, both of the two clusters have a diverse mix of heave and slight flows with heavy-tailed characteristics, we also analyze TAFA's performance across these two different workflows.

Trace: Using the Google cluster traces in [20] and [21], we illustrate the heterogeneity of server configurations in one of the cluster [22], where the CPUs and memory of each server are normalized. We use the information of over 900 users on a cluster of 12K servers as the input of TAFA, and evaluate its performance against other policies based on these traces.

TAFA: To generalize our work, we consider three sets of experiments. Firstly, we test TAFA's parameter performance. For tasks containing plenty of flows, we use task completion time (TCT) defined as the finish time of the last flow in this task, and consider the average TCT across all tasks from end-hosts. Besides, we find the resource utilization rate of TAFA is high, the demotion threshold and the number of priority queues effect the finishing time of flows. Secondly, we compare TAFA with task-aware policies. As we introduced before, flow completion time seriously affects TCT. With flow-level knowledge, we can schedule flows in a more proper way to advance the finish time of the last flow in one task, i.e., reducing TCT. Lastly, we compare TAFA with flow-aware policies, and demonstrate that TAFA can shorten flow response time (FRT) significantly.

B. Overall performance of TAFA

To evaluate how TAFA adapts to realistic activity, we set up the environments mimics a typical DCN scenario. The front-end comprises of three clients; each client sent out tasks persistently, and tags flows of these tasks by a maintained separate marker. Each task is initialized to the highest priority, and is degraded gradually when the size achieving the threshold. The small cluster is configured in proportion to [20]. CPU and memory units are normalized to the maximum server. The 6 kinds configuration rates are: $(0.50, 0.50)$, $(0.50, 0.25)$, $(0.50, 0.75)$, $(1.00, 1.00)$, $(0.25, 0.25)$, $(0.50, 0.12)$.

Threshold. We first evaluate the impact of varying the value of threshold in switches. As there are more than one priority queue in switches, a task may demote form one higher priority queue to a lower one depending on the bytes it has sent. The demotion is depended on the queue threshold Υ. As we described in Section 4, not using a global threshold in all queues, we give a vector of Υ (consist of $\Upsilon_1, \Upsilon_2, ..., \Upsilon_{\tau-1}$). Where τ is the number of priority queues, and Υ_i is the threshold of priority queue i. To test the performance of different values, we consider a scenario where there are three different queues in switches, and Υ_1 is set to be the mean value of the largest task size while Υ_2 is set to be three quarters of the largest task size. As a contrast experiment, we set the two threshold as one-third and two-thirds of task size. Fig. 4 shows the results in three different experiments with 20, 50, and 100 requests in different simulations. From this figure, we can find that our incremental threshold outperform the fixed threshold

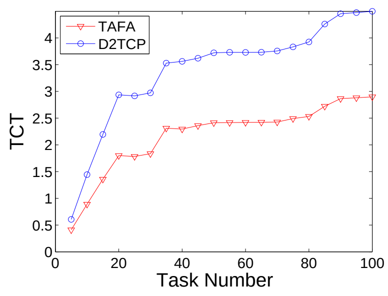


Fig. 7: 100 requests with task size from 1KB to 100KB.

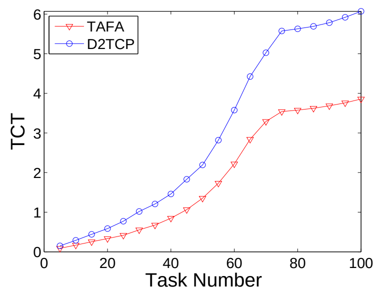


Fig. 8: 100 requests with task size from 10M to ∞.

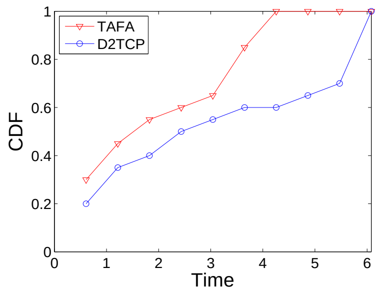


Fig. 9: Average CDF of task completion time.

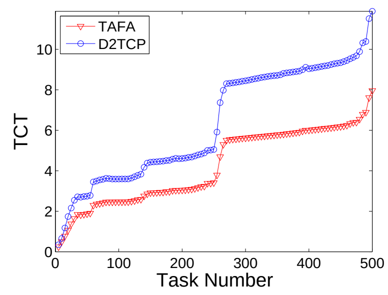


Fig. 10: 500 tasks with size from 1KB to 100KB.

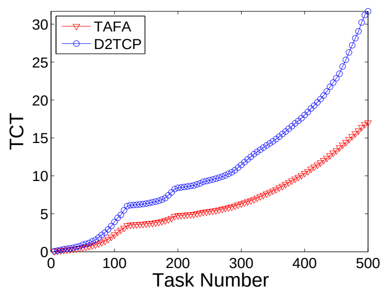


Fig. 11: 500 tasks with size from 10M to ∞.

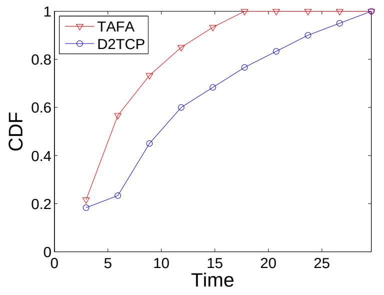


Fig. 12: Average CDF of task completion time.

obviously, and more tasks, more advantages.

Queues. Nomatter for the sequential or aggregational access schemes, the order of flows does influence the final completion result, because only when all the flows return, can the final result forms. So when the previous flow is heavy, and blocks the process of following ones, multiple queues can handle this scenario. The number of priority queues will affect the optimizing degree. We set the request from clients to 20, 40, 60, 80, and 100, respectively, using different number of priority queues (2, 3, 4, 8). Fig. 5 gives the results, from which we can conclude that multiple queue can optimize TCT to some extent, but with the queue number increasing, the superiority is not that obvious. So for a specific DCN, the number of queues is not the more the better, but should be set to a appropriate amount by considering the overhead of adding an additional queue.

C. *TAFA vs. Task-aware*

We compare TAFA's performance against FIFO, which is used in the task-aware scheduling in [1]. Fig. 6 shows results of an experiment with 3 clients, 100 tasks, task size of 1KB to 100KB. In this case, TAFA reduces task droptail completion time of by about 34% compared to FIFO.

The reason why TAFA outperforms task-aware schemes lies in the acknowledgment of flow information. As task completion time is depended on the last flow of it, making flows scheduled earlier will certainly reduce TCT. The flow-level scheduling algorithm makes TAFA flow-aware, so that TAFA can significantly improve task completion time compared to all task-aware only policies.

D. *TAFA vs. Flow-aware*

In this subsection, we evaluate TAFA with flow-aware schemes. For the experiment, we consider google's trace files in [20] and [21]. Along with the experiments in [1], we compare TAFA's performance against D^2TCP. Fig. 7 shows the results of an experiment with 100 requests, task size from 1KB to 100KB. Obviously, TAFA takes shorter time to finish these tasks, reducing the tail completion time by 36% to D^2TCP.

Also, we can observe more gains in long tasks simulation. We plot the average TCT in Fig. 8 and each end-host with the shortest task 10MB. The results show that TAFA reduces 45% average TCT compared with D^2TCP. In short, TAFA works well for both workloads.

TAFA also achieves very good performance for the CDF of task completion, shown in Fig. 9. TAFA can finish scheduling all the tasks at about 4 while it takes D^2TCP more than 6.

To test the scalability of TAFA, we increase the number of requests, and simulate the situation with 500 tasks and Fig.10, 11, 12 shows the task completion time for short tasks, long tasks, and average CDF, respectively. From these figures, we can conclude that TAFA can be adaptive to large-scale environments.

The reason why TAFA can achieve better results than D^2TCP is that TAFA takes flow size into consideration when adjusting congestion window. When congestion occurs, the back off extent of short flows is slighter than long ones, making short flows be scheduled earlier, thus, TAFA reduces average task completion time.

VI. Conclusion

In this paper, we studied the scheduling problem in datacenter networks (DCNs), where the existing protocols are either task-aware or flow-aware.

To optimize task completion time (TCT), we present TAFA, which is both task-aware and flow-aware. In the task level of TAFA, we adopt a heuristic demotion algorithm, which can demote the priority of heavy tasks without prior knowledge of task size, so that TAFA obtains the advantages of short job first, which is known to be the most effective method to reduce average completion time in one link. In the flow level of TAFA, we modulate the congestion window in a size-aware manner, thus making long flows back off more aggressively than short flows. As to the rate control problem, we take flow size into consideration, and adjust congestion window according to the estimation calculated by flow size and the fraction of packets that were marked in the last RTT. This scheme can help shorter flows back off more slighter than long ones, and make short flows to be scheduled earlier, resulting in shorter task completion time. Large-scale simulations driven by real production datacenter trace show that, compared to traditional task-aware or flow-aware only policies, TAFA can significantly reduce the average task completion time.

References

[1] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," 2013.

[2] J. Guo, F. Liu, J. Lui, and H.-J. Jin, "Fair network bandwidth allocation in iaas datacenters via a cooperative game approach."

[3] J. Guo, F. Liu, X. Huang, J. Lui, M. Hu, Q. Gao, and H. Jin, "On efficient bandwidth allocation for traffic variability in datacenters," in *INFOCOM, 2014 Proceedings IEEE*. IEEE, 2014, pp. 1572–1580.

[4] F. Xu, F. Liu, H. Jin, and A. V. Vasilakos, "Managing performance overhead of virtual machines in cloud computing: a survey, state of the art, and future directions," *Proceedings of the IEEE*, vol. 102, no. 1, pp. 11–31, 2014.

[5] F. Xu, F. Liu, L. Liu, H. Jin, B. Li, and B. Li, "iaware: Making live migration of virtual machines interference-aware in the cloud," *Computers, IEEE Transactions on*, vol. 63, no. 12, pp. 3012–3025, 2014.

[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," *ACM SIGCOMM computer communication review*, vol. 41, no. 4, pp. 63–74, 2011.

[7] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 115–126, 2012.

[8] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 50–61.

[9] C.-Y. Hong, M. Caesar, and P. Godfrey, "Finishing flows quickly with preemptive scheduling," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 127–138, 2012.

[10] N. Dukkipati and N. McKeown, "Why flow-completion time is the right metric for congestion control," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 1, pp. 59–62, 2006.

[11] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. ACM, 2013, pp. 435–446.

[12] H. ZHANG, "More load, more differentiation - a design principle for deadline-aware flow control in dcns," in *INFOCOM, 2014 Proceedings IEEE*. IEEE, 2014.

[13] M. Shen, L. Gao, K. Xu, and L. Zhu, "Achieving bandwidth guarantees in multi-tenant cloud networks using a dual-hose model."

[14] K. Xu, Y. Zhang, X. Shi, H. Wang, Y. Wang, and M. Shen, "Online combinatorial double auction for mobile cloud computing markets."

[15] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 455–466.

[16] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 4. ACM, 1989, pp. 1–12.

[17] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *Networking, IEEE/ACM Transactions on*, vol. 4, no. 3, pp. 375–385, 1996.

[18] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and W. Sun, "Pias: Practical information-agnostic flow scheduling for data center networks," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. ACM, 2014, p. 25.

[19] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4. ACM, 2009, pp. 51–62.

[20] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 7.

[21] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces," http://code.google.com/p/googleclusterdata/.

[22] W. Wang, B. Li, and B. Liang, "Dominant resource fairness in cloud computing systems with heterogeneous servers," *arXiv preprint arXiv:1308.0083*, 2013.