OPTAS: Decentralized Flow Monitoring and Scheduling for Tiny Tasks

Ziyang Li[1,2], Yiming Zhang[1], Dongsheng Li[1], Kai Chen[2], Yuxing Peng[1]
[1]PDL Lab, National University of Defense Technology
[2]SING Group, Hong Kong University of Science and Technology

Abstract—**Task-aware flow schedulers collect task information across the data center to optimize task-level performance. However, the majority of the tasks, which generate short flows and are called *tiny tasks*, have been largely overlooked by current schedulers. The large number of tiny tasks brings significant overhead to the centralized schedulers, while the existing decentralized schedulers are too complex to fit in commodity switches. In this paper we present OPTAS, a lightweight, commodity-switch-compatible scheduling solution that efficiently monitors and schedules flows for tiny tasks with low overhead. OPTAS monitors system calls and buffer footprints to recognize the tiny tasks, and assigns them with higher priorities than larger ones. The tiny tasks are then transferred in a FIFO manner by adjusting two attributes, namely, the window size and round trip time, of TCP. We have implemented OPTAS as a Linux kernel module, and experiments on our 37-server testbed show that OPTAS is at least $2.2\times$ faster than fair sharing, and $1.2\times$ faster than only assigning tiny tasks with the highest priority.**

I. INTRODUCTION

Various data center applications perform a large number of user tasks, which generate tons of flows into the networks and endow them with rich semantics. For example, Coflow [9] describes a group of flows of the same task, and enables collaborative network scheduling to meet task-level goals. This has motivated research efforts [10, 12, 13, 24] to improve the performance of tasks/coflows.

Since the traffic in data centers conforms to a heavy-tail distribution [12]. Less than 10% of the tasks generate more than 98% traffic, and the majority (more than 90%) of the tasks generate short flows and are called *tiny tasks*. The tiny tasks usually deliver messages for user-facing applications such as web search and online games, so the performance of tiny tasks has significant impact on user experience.

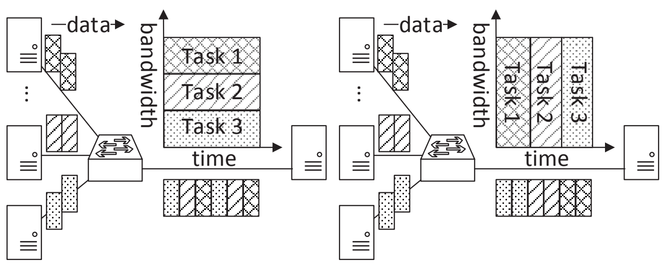
The network traffic in data centers is bursty at the granularity of seconds [8]. A single task usually comprises a group of parallel flows. When many tasks running at the same time, a large number of flows will occur simultaneously in the network, and the packets of the tasks may cause long queues in switch buffers, resulting in packet loss and long queuing delay. This requires network schedulers to response quickly.

Since the quantity of tiny tasks in data center is quite large, the optimization of network scheduling for them will effectively improve user experience. This requires network schedulers to be low overhead. Conventional flow scheduling techniques [5, 14] assign short flows with high priorities, which optimizes flow-level metrics (e.g. flow completion time or FCT) but cannot achieve task-level improvement. Recently-proposed coflow-aware scheduling techniques use heuristics such as FIFO-LM [13], SEBF [12] and D-CLAS [10]. They consider coflow as the basic scheduling unit and improve performance for tasks instead of independent flows. However, the large number of tiny tasks brings significant overhead to the centralized schedulers such as Varys [12], while the existing decentralized schedulers such as Baraat [13] are too complex to fit in commodity switches. Specifically, the state-of-the-art coflow scheduling techniques cannot meet the requirements of network scheduling for tiny tasks due to following reasons.

- **Centralized scheduling overhead is not negligible.** Tiny tasks usually do not need high bandwidth or throughput, but they usually require low-latency communication that is sensitive to the network delay. For example, the centralized coflow-aware scheduler Varys [12] batches control messages at $O(100)$ milliseconds intervals which might be significant to the overall execution time of tiny tasks.
- **Not all flow information is available beforehand.** Current coflow schedulers require prior flow knowledge to calculate scheduling policies. For example, Varys [12] and RAPIER [24] assume the complete coflow information (e.g., the flow sizes) to be available. In practice, it is difficult to expose all the information from various applications to network.
- **Some requirements are expensive or impractical.** Existing task-aware schedulers need to modify the user applications or even the switches, which requires significant engineering effort. For example, most commodity switches still do not support the advanced functions needed by Baraat [13].

To address these problems, in this paper we propose OPTAS, a lightweight, commodity-switch-compatible scheduling system for tiny tasks without modification to applications or switches. OPTAS obtains flow information by monitoring system calls and buffer footprints to identify the tiny tasks. OPTAS maintains a sorted tiny task list based on their arrive time. The tiny tasks are assigned higher priorities than the large ones to avoid blocking. OPTAS calculates the suggested window size and ACK delay time at the receiver side to approximatively transmit tiny tasks in a FIFO manner. The head task of the task list will get bandwidth resource preferentially and no ACK

This work was performed when Ziyang Li was an intern student at SING Group @ HKUST.



(a) Fair sharing between tasks (b) Serialization between tasks

Fig. 1: Motivating example

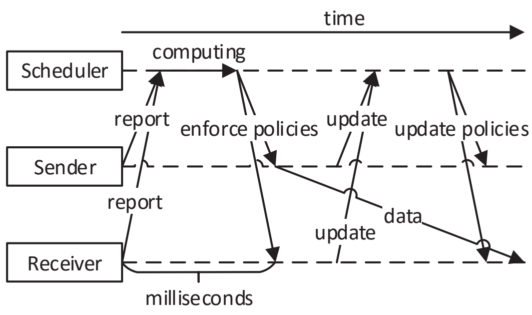


Fig. 2: Overhead introduced by centralized scheduler.

delay.

In summary, we make the following contributions.

- We model and analyze the task-aware flow scheduling problem for tiny tasks.
- We design OPTAS, an efficient decentralized tiny-task-aware flow monitoring and scheduling system.
- We have implemented OPTAS both in Linux kernel and on network simulation platform [1], and evaluate OPTAS on a real testbed. Our experiments show that OPTAS improves the network performance of tiny tasks by over $2.2\times$ compared with fair sharing and $1.2\times$ compared with priority queue.

To the best of our knowledge, we are the first to propose (and implement) a decentralized, commodity-switch-compatible flow monitoring and scheduling system for tiny tasks. The rest of this paper is organized as follows. Section II introduces the key observations that inspire the design of OPTAS. Section III proposes the optimization model and presents an overview of OPTAS. Section IV introduces the design and implementation details of OPTAS. Section V introduces the evaluation results. Section VI reviews the related work. And finally Section VII concludes the paper.

II. MOTIVATION

Conventional fair sharing scheduling makes all the tasks compete for network resources and the packets from all the tasks intersect with each other, while recent flow scheduling studies [5, 14, 21] have shown that the short flows should be assigned with high priority to minimize the average flow completion times (FCT). For example, suppose that each task has one flow that will finish in 1 time unit if monopolizing the whole network in Fig. 1a. For fair sharing, the average completion time of the three tasks is 3 time units. In contrast, if the network serves the three tasks one by one (as depicted in Fig. 1b), then the average completion time decreases to 2 time units.

Therefore, it has been well accepted that all the tiny tasks should be assigned with the highest priority. However, several challenges have to been addressed to achieve this goal.

Tiny tasks are sensitive to scheduling latency. Centralized coflow schedulers like Varys [12] make decisions according to complete coflow information of the whole network. The centralized scheduling overhead mainly includes the time of computing and communication, which severely affects the performance of tiny tasks. To enable centralized scheduling, the senders and receivers have to periodically report their current states to the scheduler, then the scheduler computes and enforces scheduling policies to the end-hosts (as shown in Fig. 2). The scheduling latency is about serval milliseconds, and the update period is $O(100)$ milliseconds [12]. However, the communication duration of tiny tasks is only about tens of milliseconds. For example, Varys [12] simply ignores coflows whose sizes are smaller than 25MB. Moreover, since the quantity of tiny tasks is quite large, the accumulated latency can not be neglected, and the overhead of centralized schedulers makes them not applicable to tiny tasks. Therefore, tiny tasks' flows should be scheduled in a decentralized manner for performance purpose.

It is challenging to obtain complete flow information. Task-aware flow scheduling needs to know the flow information ahead of time, which usually includes 4-tuple (`task_id, source, destination, flow_size`). In general, it is not easy or even impossible to obtain complete flow information beforehand. First, it requires much engineering effort, which includes modifications to operating systems and patching on applications, to expose flow information from the application layer to network. For example, FLOWPROPHET [20] requires to modify the distributed computing frameworks to expose coflow semantic. Second, some flow information in some specific applications is uncertain before communication. For stream processing applications such as Apache Storm, the flow sizes remain unknown until the processing is finished.

It is hard for the network to know coflow details. The flows are generated by tasks running on end-hosts, and the switches in the network cannot directly obtain the information about the flows and tasks. To enforce task-aware scheduling policies in the network, for instance, Baraat [13] employs a logical centralized task id server to generate globally unique ID for each task. It employs 26 bytes (which inform the switches along the path about which flows belong to which tasks) in the packet header to enable task-aware scheduling in the network. Clearly, this requires advanced functions that cannot be supported by state-of-the-art commodity switches and that may conflict with other optimizing techniques.

The conclusion is that *practical limitations such as scalability and information availability must be considered when scheduling the flows of tiny tasks*. The state-of-the-art task-/coflow-aware scheduling methods [10, 12, 13] cannot meet the needs of network scheduling for tiny tasks.

III. Analysis and Design Overview

In this section we first list the desirable properties of tiny task scheduling, and then propose and analyze the optimizing scheduling problem with constraints. At the end of this section we present an overview of OPTAS, a lightweight, commodity-switch-compatible monitoring and scheduling system for tiny tasks.

A. Desirable Properties

The desirable properties of a scheduling system for tiny tasks are listed as follows, which are the design goals of OPTAS.

- *Efficiency:* The objective of optimization is to minimize task completion times rather than flow-level metrics. By distinguishing the flows according to their tasks and sizes, OPTAS transfers flows of tiny tasks in high priority and serializes them without interleaving.
- *Scalability:* Since tiny tasks are sensitive to network delay and scheduling overhead, the network scheduling system should not introduce extra overhead comparable to tiny tasks' computing overhead. To achieve scalability, OPTAS does not adopt centralized scheduling and make decentralized scheduling decisions at end-hosts.
- *Flexibility:* The scheduling system should not require any special support from switches or modification to the applications. Furthermore, it should not conflict with other optimization techniques.

OPTAS optimizes network performance for tiny tasks by carefully monitoring and scheduling their flows. Given multiple tasks running in a network, OPTAS decides which tasks are tiny tasks, determines appropriate TCP parameters for each tiny task, and optimizes the average completion times of all the tiny tasks. OPTAS can also adapt to incomplete flow information and be complementary with current scheduling systems (such as Varys [12] and RAPIER [24]).

B. Analysis

TCP is the *de facto* standard for reliable transport layer protocol in data center communications. As a window-based protocol, TCP has two parameters that control the network flows, i.e., the window size and the round trip time. For example, PAC [6] and ICTCP [22] use the two parameters for congestion control. However, these flow-level protocols are agnostic to task-level flow semantics, and flow-level optimization does not necessarily improve task performance.

Since the flow sizes do not vary significantly in the situation of tiny tasks, we consider the size of the longest flow of the task as that of all the flows for simplicity. Assuming that n tasks are running on the node, we formulate the problem to minimize the task completion times as follows.

$$minimize \sum_{i=1}^{n} d_i \quad (1)$$

s.t.

$$\forall t, \sum_{i=1}^{n} \sum_{f=1}^{n_i} r_i^f(t) \le C \quad (1a)$$

$$\forall i,t,f, \sum_{t=t_i^s}^{t_i^s+d_i} r_i^f(t) \cdot \Delta t \ge S_i^f \quad (1b)$$

$$r_i^f(t) = \frac{w_i^f(t)}{rtt_i^f(t)} \quad (1c)$$

$$\forall i,t,f, base\_rtt \le rtt_i^f(t) < RTO_{min} \quad (1d)$$

$$\forall i,t,f, w_i^f(t) \ge W_{min} \quad (1e)$$

In the formulation, d_i is the duration of task i, and hence the sum of d_i should be minimized. Symbol n_i is number of flows of task i, $r_i^f(t)$ is the rate of f_{th} flow of task i at time t. The C is link capacity of the node. So that constraint $(1a)$ means the sum of bandwidth of all flows should not exceed the link capacity. The $w_i^f(t)$ and $rtt_i^f(t)$ are the window size and RTT (round trip time) respectively. In constraint $(1b)$, t_i^s is the start time of task i, Δt is the scheduling interval, S_i^f is the flow size. So the constraint $(1b)$ means that all flows of task i should have finished in the interval $[t_i^s, t_i^s + d_i]$. The $(1c)$ presents the fact that rate can be calculated as window size divided by RTT. With constraint $(1d)$, the valid value range of RTT of any flow, where $base\_rtt$ is the physical link latency and RTO_{min} is the timeout threshold to trigger packet retransmission. $(1e)$ requires that the windows size should not be smaller than W_{min}.

Problem (1) defines the optimization goal and constraints to fit in TCP. These constraints guarantee that flows of all the tasks can be transferred smoothly without disturbing upper-layer applications. However, the optimal solution is hard to obtain, because the programming is nonlinear and varies over time. OPTAS follows the constrains in the optimization problem, and solves the problem in a practical way.

C. OPTAS *in a Nutshell*

We describe the network optimization framework of OPTAS in Algorithm 1. OPTAS makes use of the window size and round trip time to realize TCP-compatible, task-aware transport layer control. When a packet arrives, OPTAS determines which flow it belongs to, then checks whether the network is congested. If so, OPTAS postpones the transmission of the ACK for a calculated period of time. Once the network is not congested or the delay time is out, OPTAS will send ACK for the received packets. An ACK packet will carry suggested window size from receiver to the sender side.

IV. Design and Implementation

A. Architecture

To achieve all these goals described in Section III-A, we design and implement OPTAS. When implementing a network layer task-aware scheduling system, we are essentially solving the following sub-problems.

How to collect flow information without modifying applications? For a given flow, OPTAS needs to know two kinds

Algorithm 1 The OPTAS Framework

```
 1: procedure ACK(Packet $p$, Task $t$)
 2:     suggested window = CALCWINDOW(t)
 3:     send ACK packet for $p$
 4: end procedure
    Main loop:
 5: while true do
 6:     if new packet $p$ received then
 7:         find task $t$ which $p$ belongs to
 8:         if $CongestionAvoidance()$ then
 9:             delay_time = CALCDELAY(t)
10:             set ack_timer(delay_time)
11:         else
12:             call ACK($p$, $t$)
13:         end if
14:     end if
15:     if ack_timer.expire() then
16:         call ACK($p$, $t$)
17:     end if
18:     for task in task_list do
19:         if not $CongestionAvoidance()$ then
20:             send ACK of the task
21:         end if
22:     end for
23: end while
```

of information. The first is that the flow belongs to which task. The second is whether the flow is small enough so that the task might be tiny. Knowing the accurate flow sizes is helpful but not mandatory in OPTAS.

How to deliver and enforce the policies? We aim to design an in-network task-aware scheduling system, so OPTAS employs two controllable attributes in TCP, which are window size and RTT. The window size defines how much a sender can transmit without receiving an acknowledgment. The RTT can be controlled by delay ACK. since $base\_rtt$ is the link latency, and $delay$ is the ACK delay time, so the real RTT is

$$rtt = base\_rtt + delay$$

Each flow involves a sender and a receiver between which the packets are transmitted. The scheduling policies can be encoded in the TCP/IP header of the packets.

How to calculate scheduling policies? Firstly, OPTAS assigns tiny tasks with priority higher than large tasks to avoid head-of-line blocking. Secondly, we assign network resources based on the start time of each tiny tasks. The task arrived earliest will get bigger windows size and lower round trip time than others. This strategy looks like FIFO, but is different due to the subsequent tasks are not completely blocked. The constraints $(1d)$ and $(1e)$ guarantee that all tasks can transmit smoothly, or else the upper layer applications may get alarmed. However, FIFO is not feasible for large tasks, due to the flow sizes of large tasks vary intensely, which degrade performance severely once a extreme large task arrives earlier than others.

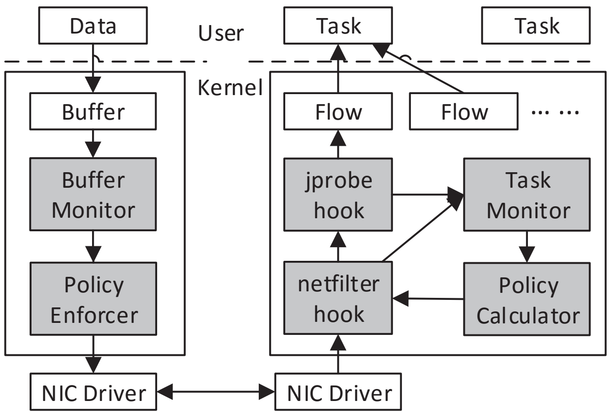


Fig. 3: OPTAS architecture

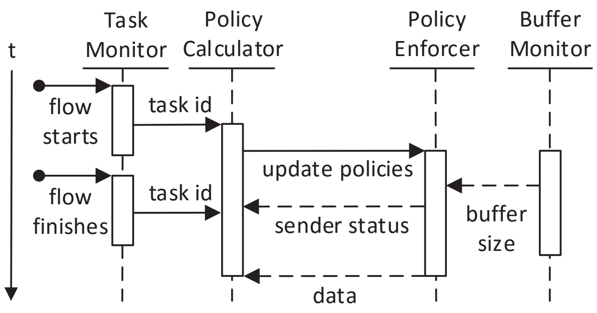


Fig. 4: Sequence diagram of OPTAS

Fig. 3 depicts the architecture of OPTAS, which mainly contains four modules: Task Monitor, Policy Calculator, Buffer Monitor and Policy Enforcer. The Task Monitor and Policy Calculator run on the receiver side, while the other two run on sender side.

B. Workflow

Fig. 4 illustrates how the modules of OPTAS cooperate to make scheduling decisions.

- On the receiver side, when a task starts a new flow, the Task Monitor identifies its `task_id`. The Task Monitor passes `task_id` to the Policy Calculator. Then, the Policy Calculator knows about the relationships between tasks and flows. The Policy Calculator sorts tasks according to their start time, then calculates the suggested window size and ACK delay time.
- On the sender side, the Buffer Monitor records send buffer status and delivers to the Policy Enforcer. The Policy Enforcer applies the suggested window size to the packets about to send and encodes flow size judgement in the outgoing packets.
- When a flow finishes, Task Monitor notifies the Policy Calculator. Until all flows of the task finished, the task is completed and removed from the task list.

C. Implementation

We now describe the implementation of the 4 modules of OPTAS in detail.

Task Monitor

To obtain the relationships between tasks and flows, Task Monitor tracks basic network functions such as *tcp_recvmsg*. For example, Task Monitor can record the PID of each task at the entry of *tcp_recvmsg*, which is used as `task_id`. Flows with the same `task_id` are treated as a coflow. When a flow is starting, Task monitor notifies the Policy Calculator with 4-tuple (`task_id, source, destination, flow_size*`). The `source` and `destination` are guideposts of the flow. The `flow_size` is optional, since it sometimes can not be known beforehand. When a flow ends, it will be removed from the flow list of its associated task. If all flows of a task are completed, the task will be marked as completed.

Buffer Monitor

On the sender side, the sender buffer can tell us how much backlogged data is waiting to send [2]. When the data is ready to send, the system function (e.g. *tcp_sendmsg*) will be called, the data will be copied from user space memory to kernel space send buffer, then NIC (network interface card) will read the buffer and transmit data to network. Thus, the data backlogged in sendbuffer is the flow data ready to be transmitted into network, we can infer flow sizes from the send buffer status.

OPTAS sets one threshold to filtrate large tasks. We compare the backlogged bytes in sendbuffer plus the transmitted bytes through network to the threshold. While a flow transmitting, if the sum of backlogged bytes and transmitted bytes is larger than the threshold, we have enough confidence to say the flow is a long flow, the task it involved is a large task.

Fig.5 depicts that the send buffer size changes along with flow size when the flow sends its first few packets. The default send buffer size is 1MB, when the flow size is smaller than the sendbuffer capacity, the send buffer will not be full. When the first few packets are about to send, send buffer of the flow is filled by user data, so the occupied buffer size reflects the size of flow. Therefore, the send buffer is a good indicator for flow sizes. The flow size threshold is not a fixed value, it depends on how we define a tiny task. In our definition, a tiny task is all flows it involves are smaller than 1MB, so we set the threshold 1MB.

Moreover, different applications and flows leave different footprints on the send buffer. The send buffer footprints depend on the flow size and specific application behaviour. On the other hand, the data input rate can not fill out the send buffer or the maximum buffer size is not large enough to distinguish flows, the network is not the bottleneck anymore. Although the send buffer is a good indicator for flow sizes, it is not always accurate. Sometimes the send buffer fails to advise flow size in advance.

We implement the task and buffer monitors by using *jprobe* and *netfilter*. Jprobe is a probe tool provided by Linux kernel, and netfilter is a series of hooks in various points in Linux network stack. These hooks provide us convenience to monitor the tasks and network stacks.

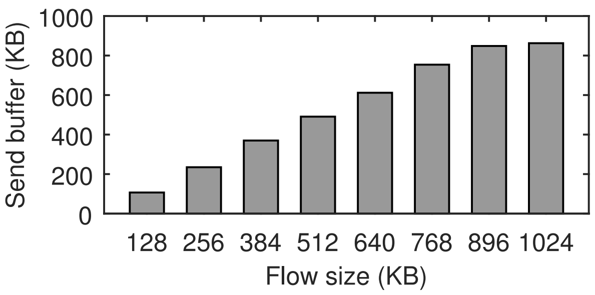


Fig. 5: Send buffer indicates the size of small flows

Algorithm 2 The Delay ACK Algorithm

```
 1: procedure CALCDELAY(Task $t$)
 2:     the bottleneck capacity is $C$
 3:     if $t$ is task_list.head then return 0
 4:     else
 5:         $b^m_{all}$ is the overall throughput
 6:         if $b^m_{all} \geq \alpha C$ then return $Delay_{max}$
 7:         else  return $delay$ calculated as Equation (2)
 8:         end if
 9:     end if
10: end procedure
```

Policy Calculator

Policy Calculator maintains a list of tasks, the tasks are sorted by arrive time, that is the latest arrived task will be added into the list tail. The head of task list has higher priority then others, which means that network resources are assigned to it preferably.

To solve the problem formulated above, the Policy Calculator computes the acknowledgement delay time when every packet arrives. The detail algorithm is demonstrated in the Algorithm 2. The procedure CALCDELAY calculates the delay time for each task. If a task is at the head of task list, no delay will added into it. Otherwise, the delay time is calculated based on the remaining bandwidth. The actual throughput is measured as the total number of bytes it received divided by the time interval, then smoothed by an exponential factor. b^m_{all} denotes the overall throughput. When b^m_{all} is larger than the threshold αC, the delay time is $Delay_{max}$, where α is a predefined threshold. In this paper, the default value of α is 0.9. $Delay_{max}$ is the maximum possible delay. If we set the delay time larger than $Delay_{max}$, it may trigger retransmission, which will hurt the goodput. If b^m is smaller than αC, the delay time will be calculated as

$$delay = \min(n_t w_t/(\alpha C - b^m_{all}) - base\_rtt, Delay_{max}) \quad (2)$$

The $\min$ operation ensures that the delay is not larger than $Delay_{max}$.

When the network is not congested or the delay time expires, the Policy Calculator sends an acknowledgement, then the flow will continue to transmit. When an acknowledgement packet is about to send, the receive window of a task is calculated based on whether it is the head of task and the

Algorithm 3 The Window Suggesting Algorithm

```
1: procedure CALCWINDOW(Task $t$)
2:     $w_t^{old}$ is the old window size
3:     $w_t$ is the window size of task $t$
4:     if $t$ is task_list.head then
5:         if $b^m \geq \alpha C$ then
6:             $w_t = \max(w_t^{old} - 1, W_{min})$
7:         else if $b^m \leq \alpha C/2$ then
8:             $w_t = w_t^{old} * 2$
9:         else
10:            $w_t = w_t^{old} + 1$
11:        end if
12:    else
13:        $w_t = W_{min}$
14:    end if
15: end procedure
```

overall throughput. OPTAS calculates the suggested window size by using procedure CALCWINDOW in Algorithm 3, where b^m is the measured throughput of the first task in task_list.

Policy Enforcer

The Policy Enforcer runs on the sender side and receives messages from the Buffer Monitor and the Policy Calculator. The Policy Enforcer performs two functions as follows.

1) The Policy Enforcer collects send buffer sizes from the Buffer Monitor and the data size already sent, then decides whether the flows are long flows, The Policy Enforcer encodes the decision into the last bit of Differentiated Services Code Point (DSCP) field of packet headers.
2) The Policy Enforcer transfers packets in the window size as the Policy Calculator suggests.

D. Discussion

OPTAS offers efficient tiny-task-aware scheduling in low overhead, and is adaptive for incomplete flow information. Here we remark on some specific cases:

Long term tasks: OPTAS also tasks care of background services. Tasks of such services are often in long term running. The accumulated data sizes of long term tasks are large, but simply assigning them with low priority is not appropriate. Some background services (e.g. ssh and heartbeat) require low latency. while some do not (e.g., HDFS rebalance and index update). To guarantee the low latency requirement, OPTAS resets the start time of a long term task after it stops communication for a while.

Traffic patterns: We ignore the other traffic patterns (i.e. one-to-many and many-to-many) due to a lack of cluster-wide insight. For one-to-many or many-to-many communication, multiple receivers are involved, it is hard to classify the sub-coflows on these receivers as one coflow. This needs additional query from the task controller (e.g. master of Spark).

Flow size skew: The flow sizes of some tasks may be skew, due to they are mixed with short and long flows. We count these tasks as large tasks, since the completion time depends on the longest flow. Otherwise, they will block other tiny tasks for a long while, if we put them in the tiny task list.

Sender side contention: The receiver side RTT the duration from the time an ACK packet issued to the time packet with sequence number ACK + WND received. Once the sender is congested, the receiver side will observe a larger RTT than usual. Then the receiver side will decrease the suggested window size by one MSS.

V. EVALUATION

A. Methodology

Experiment Setup: We have implemented OPTAS as a Linux kernel module and deployed it on our 37-server testbed. The 37 physical servers are Dell PowerEdge R320 servers with a quad core Intel Xeons E5-1410 2.8GHz CPU, 24GB DDR3 memory, 500GB hard disk and one Broadcom NetXtreme Gigabit Ethernet NIC. The OS is Debian 6.0 64-bit version with kernel 2.6.32-5.

In our experiment, all the files are stored in RAM to testify the network performance. The file sizes determine that whether the task is tiny or not. We setup user applications using Redis, a high-performance in-memory key-value store. In our testbed, one server is a Redis client, while the other 36 server are Redis servers. We run tasks on the client, which read data from remote servers. Since each task involves a group of flows into network, the completion time of each task is when its last flow finished.

We use ICTCP [22] as the performance baseline. The reason to choose ICTCP is that it is a fair sharing mechanism and can alleviate incast congestion. The performance improvement is calculated as

$$\text{Improvement Factor} = \frac{\text{Baseline duration}}{\text{Optimized duration}}$$

To evaluate OPTAS performance, we generated the following three types of traffic scenarios.

- *Fixed flows and volume per task*: In this scenario, the flow count and volume per task are fixed. The total number of flows and traffic volume are increased with the number of tasks.
- *Fixed tasks and volume per task*: In this scenario, the number of tasks and volume per task are fixed, while the number of flows per task varies.
- *Fixed tasks and flows per tasks*: In this scenario, we fix the number of tasks and flow count per task, the traffic volume per task is randomized.

Summary of results: The main highlights of our results are as follows.

- **Effectiveness:** When only tiny tasks are present, OPTAS achieves at least $1.4\times$ performance improvement over fair sharing. Along with the tasks count ranges from 2 to 10, the improvement factor rises from 1.43 to 1.59. When tiny tasks are mixed with large tasks, OPTAS is at least

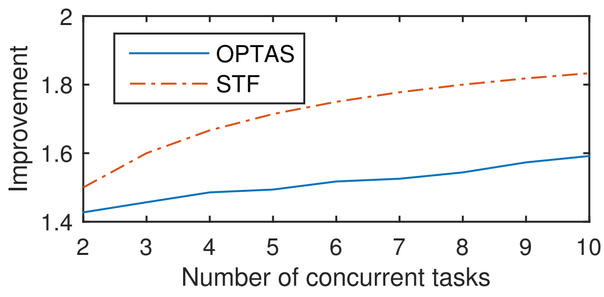


Fig. 6: Improvements in average task completion times of tiny tasks using OPTAS over fair sharing.

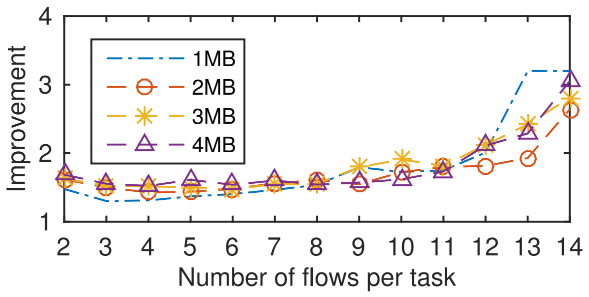


Fig. 7: Improvements in average task completion time using OPTAS over fair sharing for varying task width.

2.2× faster than fair sharing, and 1.2× than tiny tasks in highest priority.

- **Compatibility:** OPTAS is compatible with other optimization solutions for large coflows, such as Varys and RAPIER. The integrated system consists of OPTAS and Varys can reach performance improvement as much as 1.93×.
- **Overhead:** OPTAS brings negligible overhead to system. The additional CPU and memory overhead OPTAS introduced are positive relevant to the number of concurrent tasks, which are less than 1% and tens of KBs respectively. The latency introduced by OPTAS is less than 0.2%.

B. Effectiveness of OPTAS

Performance Improvement

Fig. 6 shows the improvements under the fixed flows and volume per task scenario. All these tasks are tiny tasks, and each task involves 8 flows and the size of each flow is 256KB, the number of concurrent tasks ranges from 2 to 10. OPTAS can transfer tasks at least 1.43× faster than fair sharing. The improvement factor increases from 1.43× to 1.59× alone with the number of concurrent tasks ranges from 2 to 10. The omniscient is the situation that all flow information can be known in advance, then we can choose the best scheduling plan which is always schedule smallest task first (STF).

Fig. 7 shows the improvements using OPTAS over fair sharing for varying task width. These are 8 tiny tasks running simultaneously, the number of flows per task ranges from 2 to 14. The 4 lines in Fig. 7 are the total volume of tasks, which are 1MB, 2MB, 3MB and 4MB respectively. The improvement factor falls steady in the interval [1.47, 1.77]. However, when the width exceeds 12, severe congestion happens in ICTCP, the goodput of ICTCP declines, so the improvement factor increases rapidly.

The performance of OPTAS also depends on the tasks arrive rate (as shown in Fig. 8). The X-axis in Figure 8 is the number of tasks per second, and the Y-axis is the improvement factor over fair sharing. When the server is busy, such as 10000 tasks per second, the improvement of OPTAS is as high as 1.46×. When the task arrive rate drops, the performance improvement declines too. When the rate is down to 10 tasks per second,

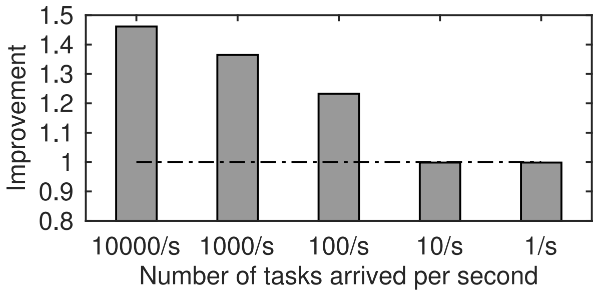


Fig. 8: OPTAS's performance when the arrive rate of tiny tasks varies.

OPTAS does not obtain performance gain any more. That is because tiny tasks do not collide with each other on network when the arrive rate is low.

These experiments prove that how OPTAS can improve network performance when only tiny tasks exist. However, the tiny tasks are mixed with large tasks in real data center environment. We answer the following questions: (i). How tiny tasks will perform with large tasks mixed? (ii). How large tasks be delayed when tiny tasks always preempt them?

We demonstrate how OPTAS performs when mixed with large tasks in Fig. 9. The tasks are classified into four classes based on two dimension metrics, size and width. The mixing ratios between these classes come from the Facebook trace which is introduced by literature [12]. The detailed recipes are listed in TABLE I.

When tiny tasks are mixed with large tasks, the scheduler always assigns the tiny tasks with higher priority than large tasks. In the legend of Fig. 9, "O" means that tiny tasks are scheduled by OPTAS while the large tasks are in fair sharing, in the context that they have higher priority. As a comparison to "O", the "F" means under tiny tasks are scheduled preferentially, the intra tiny tasks are scheduled in fair sharing way. The "T" and "L" are the improvement factors of tiny tasks and large tasks respectively.

In Fig. 9, O/T denotes OPTAS for tiny tasks, F/T denotes fair sharing between tiny tasks. By observing Fig. 9, we can make four conclusions.

1) When the tiny and large tasks are mixed, OPTAS can improve the performance of tiny tasks significantly.

Size	short	short	long	long
Width	narrow	wide	narrow	wide
Normal	52%	16%	15%	17%
Mix-N	48%	40%	2%	10%
Mix-W	22%	15%	24%	39%
Mix-S	50%	17%	10%	23%
Mix-L	39%	27%	3%	31%

TABLE I: The mixing ratios of tiny and large tasks from the Facebook trace [12].

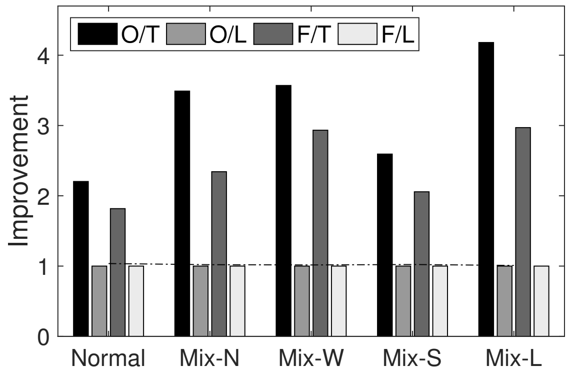


Fig. 9: OPTAS's performance under different mixing ratios with comparison to only assigning tiny tasks with high priority.

The improvement factors of tiny tasks ranges from 2 to 3.9 under different mixing ratios. The performance improvements are above 2 due to a little sacrifice of large tasks.

2) Compared to fair sharing between tiny tasks ("F/T"), OPTAS outperforms by about 20%. The improvement of OPTAS depends on how the tiny and large tasks are mixed.
3) The degradation of large tasks is less than 0.1% when they are preempted by tiny tasks ("O/L" and "F/L" in Fig. 9).
4) The black dot dash line line in Fig. 9 is the overall improvement counts for both tiny and large tasks. Since most network traffic is generated by large task, if we only optimize tiny tasks and abandon large one, the average overall improvement is only $1.02\times$.

Therefore, OPTAS can optimize tiny tasks significantly with little impact on large tasks. However, the performance optimization for large tasks is left to other techniques, such as Varys, RAPIER when the complete coflow information can be known, and Aalo when the information remains unknown. We simulate how the system performs when we integrate OPTAS and Varys together.

Scheduling Overhead

Since OPTAS is implemented as a background kernel module, the overhead is hard to measure directly. In our experiments, we do not observe obvious CPU and memory resources consumption. When OPTAS does not output performance improvement, the overhead of it can be measured by the performance degradation. As the last two bars in Fig. 8, OPTAS will add a little overhead into the tasks, which is about 0.2%. In conclusion, the resources consumption and the overhead of OPTAS is negligible.

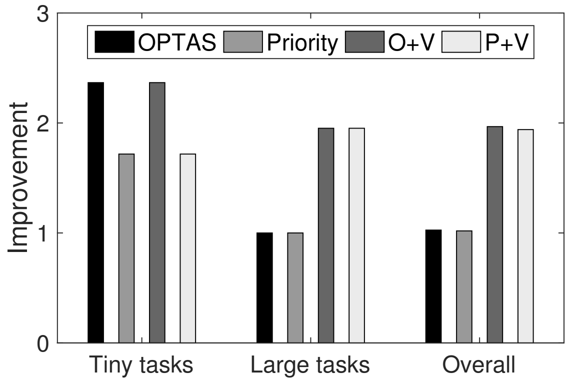


Fig. 10: The overall performance improvement when integrate OPTAS with Varys.

C. Trace-Driven Simulation

We integrate OPTAS with Varys [12] and evaluate it in the NS-2 simulator. The performance is compared against fair sharing. In our simulation, the Varys scheduler is implemented with global coflow knowledge and all schedulers do not produce any additional overhead. We synthesize the trace for simulation according to the "Normal" recipe in TABLE I.

In the legend of Fig. 10, "OPTAS" means that the tiny tasks are scheduled by OPTAS, while the large tasks are fair sharing; "Priority" means that the tiny tasks are in high priorities and fair sharing inside both tiny and large tasks; The performance improvement of tiny tasks in OPTAS is about $1.38\times$ than tiny tasks in high priorities.

However, the transmission time in network is dominated by the large tasks. If we leave the large tasks unoptimized, the overall performance improvements are just $1.03\times$ and $1.02\times$ in OPTAS and priority respectively. When we integrate OPTAS with Varys, OPTAS takes care of the tiny tasks, while Varys focuses on the large tasks. "O+V" means that OPTAS integrated with Varys; "P+V" means that the tiny tasks are in highest priority and the large tasks are scheduled by Varys. Both "O+V" and "P+V" can achieve improvement above $1.93\times$, and "O+V" is a little higher. Fig. 10 shows that OPTAS together with Varys can achieve better optimizations for both tiny tasks and overall performance.

VI. RELATED WORK

A. Flow scheduling

Various scheduling schemes have been developed to meet different design goals, such as congestion control [6, 22], performance isolation [4, 18], failure recovery [23] and flow scheduling [5, 14, 16, 21]. They focus on scheduling independent flows to achieve better network utilization and reduce the average flow completion time (FCT). For example, PDQ [14] and pFabric [5] are flow scheduling schemes to minimize FCT by assigning flows with priorities. Congestion avoidance techniques, such as ICTCP [22] and PAC [6], improve the goodput by eliminating incast congestion. ICTCP and PAC schedule flows by adjusting the window size and the proactive delay respectively, which are also used by OPTAS. All these flow scheduling studies do not take into account the flow dependency semantics and thus are coflow-agnostic. On the other hand, traffic managers, such as Hedera [3] and MicroTE [7], cannot directly be used to optimize coflows either.

B. Task-aware network scheduling

Recently, collaborative scheduling of network resource and computing tasks receives much attention from researchers. Orchestra [11] optimizes the transfer times by aware of communication patterns and uses FIFO among the patterns. Coflow [9] defines a kind of network abstraction of parallel flows. Varys [12] and RAPIER [24] apply coflow concept to their network optimizations. Both of them employ centralized schedulers, which introduce significant overhead to short flows. RAPIER jointly considers coflow scheduling and routing, OPTAS has the potential to achieve this by cooperating with some source routing works (e.g. XPath [15]). HadoopWatch [17] and FlowProphet [20] expose coflow information from distributed computing frameworks to coflow schedulers. They can only work on specific applications. Baraat [13] is a low overhead, decentralized task-aware scheduling scheme, which needs centralized server to sort tasks in FIFO order. Unfortunately, Baraat can not be implemented using existing commodity switches. Aalo [10] schedules coflow without prior knowledge, and performs well even for tiny coflows by avoiding coordination. However, rate control on short flows is not accurate. Theoretical work [19] provides an approximation algorithm for coflow scheduling problem, which has been proved as a NP-Hard problem.

OPTAS outperforms prior task-aware schedulers in three ways. First, OPTAS schedules tiny tasks in low overhead, which pads the short slab of centralized schedulers. Second, OPTAS works fine with TCP without assumptions of impractical support from commodity switches, thus is deploy-friendly. Third, OPTAS does not require to modify user applications, which saves a lot of engineering effort.

VII. CONCLUSION

In this paper, we present an efficient tiny-task-aware flow monitoring and scheduling system called OPTAS. The key design of OPTAS are two-fold. (i) OPTAS obtains necessary information without any modification of user applications. (ii) OPTAS schedules flows of tiny tasks by adjusting the window size and the time to send ACKs. We have implemented OPTAS as a Linux kernel module, and evaluated it both on our 37-server testbed and in the NS-2 simulator. The results show that OPTAS can efficiently improve the performance of tiny tasks. OPTAS is at least $2.2\times$ faster than fair sharing, and $1.2\times$ faster than only assigning tiny tasks with the highest priority.

ACKNOWLEDGMENT

This work is supported by the Hong Kong RGC 26200014, 16203715, 613113, and the National Basic Research Program of China (973) under Grant No.2014CB340303, the National Natural Science Foundation of China under Grant No.61222205, 61379055 and 61379053, the Program for New Century Excellent Talents in University, and the Fok Ying-Tong Education Foundation under Grant No.141066. We thank Wei Bai (SING@HKUST) for discussion on implementation.

REFERENCES

[1] The network simulator ns-2. http://www.isi.edu/nsnam/ns/.
[2] AGACHE, A., AND RAICIU, C. Oh flow, are thou happy? tcp sendbuffer advertising for make benefit of clouds and tenants. In *HotCloud 2015*.
[3] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic flow scheduling for data center networks. In *NSDI 2010*.
[4] ALIZADEH, M., KABBANI, A., EDSALL, T., PRABHAKAR, B., VAHDAT, A., AND YASUDA, M. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *NSDI 2012*.
[5] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pfabric: Minimal near-optimal datacenter transport. In *SIGCOMM 2013*.
[6] BAI, W., CHEN, K., AND WU, H. Pac: Taming tcp incast congestion using proactive ack control. In *ICNP 2014*.
[7] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Microte: fine grained traffic engineering for data centers. In *CoNEXT 2011*.
[8] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Understanding data center traffic characteristics. In *IMC 2010*.
[9] CHOWDHURY, M., AND STOICA, I. Coflow: A networking abstraction for cluster applications. In *HotNets 2012*.
[10] CHOWDHURY, M., AND STOICA, I. Efficient coflow scheduling without prior knowledge. In *SIGCOMM 2015*.
[11] CHOWDHURY, M., ZAHARIA, M., MA, J., JORDAN, M. I., AND STOICA, I. Managing data transfers in computer clusters with orchestra. In *SIGCOMM 2011*.
[12] CHOWDHURY, M., ZHONG, Y., AND STOICA, I. Efficient coflow scheduling with varys. In *SIGCOMM 2014*.
[13] DOGAR, F. R., KARAGIANNIS, T., BALLANI, H., AND ROWSTRON, A. Decentralized task-aware scheduling for data center networks. In *SIGCOMM 2014*.
[14] HONG, C.-Y., CAESAR, M., AND GODFREY, P. Finishing flows quickly with preemptive scheduling. In *SIGCOMM 2012*.
[15] HU, S., CHEN, K., WU, H., BAI, W., LAN, C., WANG, H., ZHAO, H., AND GUO, C. Explicit path control in commodity data centers: Design and applications. In *NSDI 2015*.
[16] MUNIR, A., BAIG, G., IRTEZA, S., QAZI, I., LIU, I., AND DOGAR, F. Friends, not foes: Synthesizing existing transport strategies for data center networks. In *SIGCOMM 2014*.
[17] PENG, Y., CHEN, K., WANG, G., BAI, W., MA, Z., AND GU, L. Hadoopwatch: A first step towards comprehensive traffic forecasting in cloud computing. In *INFOCOM 2014*.
[18] PERRY, J., OUSTERHOUT, A., BALAKRISHNAN, H., SHAH, D., AND FUGAL, H. Fastpass: A centralized zero-queue datacenter network. In *SIGCOMM 2014*.
[19] QIU, Z., STEIN, C., AND ZHONG, Y. Minimizing the total weighted completion time of coflows in datacenter networks. In *SPAA 2015*.
[20] WANG, H., CHEN, L., CHEN, K., LI, Z., ZHANG, Y., GUAN, H., QI, Z., LI, D., AND GENG, Y. Flowprophet: Generic and accurate traffic prediction for data-parallel cluster computing. In *ICDCS 2015*.
[21] WEI, B., LI, C., KAI, C., DONGSU, H., CHEN, T., AND HAO, W. Information-agnostic flow scheduling for commodity data centers. In *NSDI 2015*.
[22] WU, H., FENG, Z., GUO, C., AND ZHANG, Y. Ictcp: incast congestion control for tcp in data-center networks. In *CoNEXT 2010*.
[23] ZHANG, Y., GUO, C., LI, D., CHU, R., WU, H., AND XIONG, Y. Cubicring: Enabling one-hop failure detection and recovery for distributed in-memory storage systems. In *NSDI 2015*.
[24] ZHAO, Y., CHEN, K., BAI, W., YU, M., TIAN, C., GENG, Y., ZHANG, Y., LI, D., AND WANG, S. Rapier: Integrating routing and scheduling for coflow-aware data center networks. In *INFOCOM 2015*.