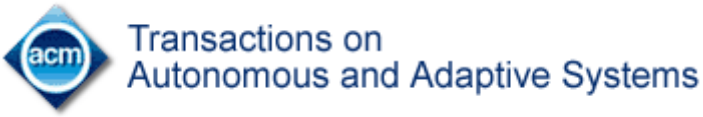


Decentralized and Adaptive Scheduling for Small Coflows in the Cloud

Journal:	*ACM Transactions on Autonomous and Adaptive Systems*
Manuscript ID	TAAS-21-0068
Manuscript Type:	Full Paper
Date Submitted by the Author:	28-Nov-2021
Complete List of Authors:	Li, Ziyang; NICEX Lab, Information Science Zhang, Yiming; NICEX Lab, Information Science Li, Huiba; NICEX Lab, Information Science Li, Dongsheng; NICEX Lab, Information Science
Keywords:	Adaptive scheduling, small I/O, coflows, corss-layer scheduling

Decentralized and Adaptive Scheduling for Small Coflows in the Cloud

ZIYANG LI, NICEX Lab
YIMING ZHANG, NICEX Lab, XMU
HUIBA LI, NICEX Lab
DONGSHENG LI, NUDT

For latency-sensitive data processing applications in the cloud, concurrent *coflows* need to be scheduled and processed promptly. The state-of-the-art schedulers collect coflow information in the cloud to optimize coflow-level performance. However, most of the coflows, classified as *small coflows* because they consist of only short flows, have been largely overlooked. This paper presents OptaX, a decentralized and adaptive flow scheduling service that efficiently schedules small coflows in commercial off-the-shelf (COTS) networks. OptaX adopts a cross-layer, COTS-switch-compatible design that leverages the `sendbuffer` information in the kernel to adaptively optimize flow scheduling in the network. Specifically, OptaX (i) monitors the system calls (syscalls) in the hosts to obtain their `sendbuffer` footprints, and (ii) recognizes small coflows and assigns high priorities to their flows. OptaX transfers these flows in a FIFO manner by adjusting TCP's two attributes: window size and round-trip time. We have implemented OptaX as a Linux kernel module. The evaluation shows that OptaX is at least 2.2× faster than fair sharing and 1.2× faster than only assigning small coflows with the highest priority. We further apply OptaX to improve the small I/O performance of Ursa, a distributed block storage system that provides virtual disks where small I/O is dominant. The result shows that URSA with OptaX achieves more than 5× improvement compared to the original URSA for small I/O latency.

CCS Concepts: •**Computing methodologies** → **Distributed algorithms;** •**Computer systems organization** → *Cloud computing;*

Additional Key Words and Phrases: Adaptive scheduling, small I/O, coflows, corss-layer scheduling

ACM Reference format:
Ziyang Li, Yiming Zhang, Huiba Li, and Dongsheng Li. 2021. Decentralized and Adaptive Scheduling for Small Coflows
in the Cloud. *ACM Trans. Autonom. Adapt. Syst.* 9, 4, Article 41 (December 2021), 23 pages.
DOI: 0000001.0000001

1 INTRODUCTION

For delay-sensitive scenarios, such as large-scale online data-intensive (OLDI) applications, the freshness of information is critical. In commercial data centers, concurrent *coflows* [12, 13] are processed promptly, where a coflow usually relies on a group of correlated short flows of one specific task. Various applications perform a large number of tasks, which in turn generate tons of streams to the network and give them rich semantics. Coflows often correspond to tasks and

Author's addresses: Z. Li, Y. Zhang, H. Li and D. Li are with the Networked Intelligent Computing at EXascale (NICEX) Lab, China.

 1556-4665/2021/12-ART41 $15.00
DOI: 0000001.0000001

describe related flows of tasks so that collaborative network scheduling can meet task-level goals, stimulating related research work [14, 16, 17, 46] on improving the performance of tasks and coflows.

Datacenter traffic follows a heavy tail distribution [16]: less than 10 percent of tasks generate more than 98 percent of traffic, while most (more than 90 percent) of tasks generate short traffic and are classified as *small coflows*. Small coflows typically deliver messages for OLDI applications such as web search, online games, and virtual disks in the cloud. Therefore, the performance of small coflows has a significant impact on the freshness of the results.

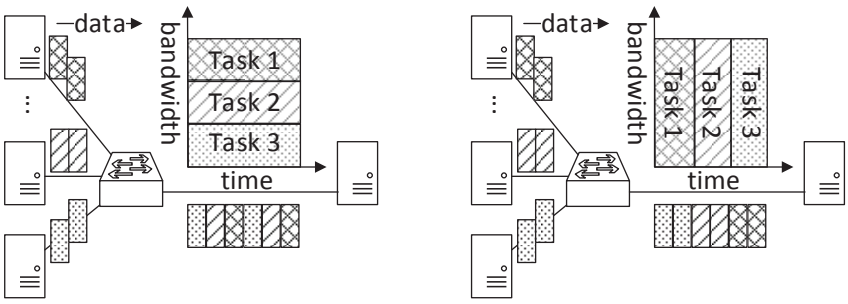
The network traffic in the data center exploded in a few seconds [9]. A single task usually consists of a set of parallel traffic. When multiple tasks run simultaneously, there will be a large number of flows overlapping in the network, and the packets of tasks may cause long queues in the switch buffer, resulting in packet loss and long queue delay. This requires the network scheduler to respond quickly.

In modern cloud applications, small coflow is dominant [16], which brings severe challenges to network sharing, requiring high performance and low overhead of flow scheduling. Traditional flow scheduling techniques [4, 21] allocate short flows with high priority and optimize flow performance indicators, such as flow completion time or FCT, but cannot achieve task-level improvement. Recently co-flu awareness program techniques use heuristics, such as FIFO-LM [17], SEBF [16], and D-CLAS [14]. They use the coflow (a set of flows of tasks) as the main scheduling unit, which improves the performance of coflow rather than independent flows. However, a large number of small coflows bring severe overhead to the centralized coflow scheduler, such as Varys [16], and existing distributed coflow schedulers such as Baraat [17] are too complex for commercial off-the-shelf (COTS) switches.

Specifically, the state-of-the-art coflow scheduling technology can not meet the network scheduling requirements of small coflows due to the following reasons.

- **Centralized scheduling overhead is not negligible.** Small coflows often do not require high bandwidth or throughput, but they are usually sensitive to network latency and require low latency traffic. For example, the centralized coflow-aware scheduler Varys [16] batches control messages at $O(100)$ milliseconds intervals, which might be important for the overall execution time of small coflows.
- **Not all flow information is available beforehand.** State-of-the-art coflow schedulers require prior flow knowledge to calculate their scheduling strategies. For example, Varys [16] and RAPIER [46] assume that complete coflow information (such as the flow sizes) is available. In practice, it is difficult to expose all the information from various applications to the network.
- **Some requirements are expensive or impractical.** Existing concurrent schedulers require modifications to user applications and even switches, which requires considerable engineering effort. For example, most commercial switches still do not support the advanced functionality required by Barratt [17].

In this paper, to address these problems, we propose OptaX, a decentralized and adaptive scheduling service for small coflows in COTS cloud networks, without modifying applications or switches. OptaX adopts a cross-layer design that monitors the system calls (syscalls) to obtain their `sendbuffer` footprints, recognizes small coflows, and assigns high priorities to their flows for scheduling. Precisely, OptaX consists of four modules, namely, (i) the task monitor (at receivers) that recognizes which flows belong to which tasks/coflows, (ii) the buffer monitor (at senders) that watches the `sendbuffer` to identify the small coflows of which all the flows are short, (iii) the policy calculator (at receivers) that calculates and suggests appropriate window size and ACK



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

Fig. 1. Motivating example

delay time, and (iv) the policy enforcer (at senders) that assigns the flows of small coflows with high priority and takes the suggested window size.

To the best of our knowledge, we are the first to propose a decentralized and adaptive monitoring and scheduling service that is commodity-switch-compatible for small coflows in the cloud. In this paper, we make the following contributions.

- We model and analyze the task-aware coflow scheduling problem.
- We design OptaX, a decentralized and adaptive coflow monitoring and scheduling system.
- We have implemented OptaX in the Linux kernel and made OptaX practical for multi-flow-type, multi-resource scenarios.

Experimental evaluation shows that shows that compared with fair sharing, OptaX improves the network performance of small coflows by more than 2.2× under under the condition of small coflow and 1.2× compared with priority queue. We further apply OptaX to improve the small I/O performance of Ursa [26], a distributed block storage system that provides virtual disks dominated by small I/O. The result shows that Ursa using OptaX is more than 5× higher than the original Ursa for small I/O latency.

The rest of this paper is organized as follows. Section 2 introduces the key observations that inspire the design of OptaX. Section 3 puts forward the optimization model and presents an overview of OptaX. Section 4 introduces the design and implementation details of OptaX. Section 5 introduces collaboration of OptaX with existing scheduling methods. Section 6 introduces how we apply OptaX to improve the small I/O performance of the Ursa block store. Section 7 describes the evaluation results. Section 8 reviews the related work and discusses various issues of OptaX. And finally, Section 9 summarizes the paper and discusses the future work.

2 MOTIVATION

Traditional fair-sharing scheduling enables all tasks[1] of network resources to compete and packages from all tasks to interact with each other. In contrast, recent traffic scheduling studies [4, 7, 12, 21] show that short traffic should be assigned a high priority to minimize the average traffic completion time (FCT). For example, suppose that each task has one flow that will finish in 1 time unit if monopolizing the whole network in Figure 1a. The average completion time for these three tasks is 3 time units to share fairly. By contrast, the average completion time is reduced to 2 time units if the network performs three tasks one by one (as shown in Figure 1b).

[1]One task corresponds to one coflow [14], and in the rest of this paper we will use the two terms interchangeably.

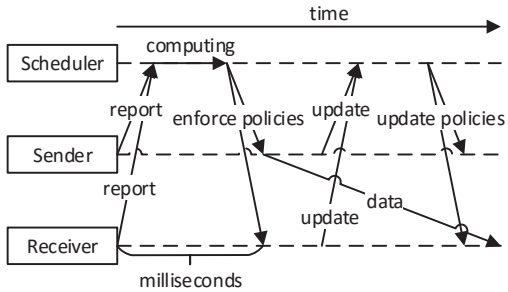


Fig. 2. Overhead introduced by centralized scheduler.

Therefore, it is well accepted that all small coflows should be given the highest priority. However, to achieve this goal, a number of challenges must be met.

(1) Small coflows are sensitive to scheduling latency.

A centralized concurrent scheduler like Varys [16] shunts the entire network based on the complete concurrent information. Central scheduling of overheads mainly affects computational and communication time, which almost affects the performance of small coflows. To achieve centralized scheduling, the sender and receiver report the current state to the scheduler, then calculate and execute the scheduling policy to the end-hosts (as shown in Figure 2). The scheduling delay is approximately a few milliseconds, and the update period is $O(100)$ milliseconds [16]. However, the duration of small can communal communication is only a few tens of milliseconds. For example, Varys [16] ignore coflows whose sizes are smaller than 25MB. In addition, because the amount of small traffic is quite large, the cumulative delay can not be ignored. The overhead of the centralized scheduler makes it not suitable for small coflow. Therefore, the small coflows should be scheduled in a decentralized and autonomous manner.

(2) It is difficult to obtain complete flow information.

Task-aware process scheduling requires prior knowledge of process information, typically including 4 tuples (`task_id, source, destination, flow_size`). In general, obtaining complete traffic information in advance is not easy or even impossible. First, it requires a lot of engineering work, including modifying the operating system and patching applications to expose flow information from the application layer to the network. For example, FlowProphet [41] requires modifications to the distributed computing framework to expose concurrent semantics. Second, some traffic information in certain applications is uncertain until it is communicated. The flow size is still unknown for streaming applications, such as Apache storms until processing is complete. Therefore, the small coflows should be scheduled in an adaptive manner.

(3) It is hard for the network to know coflow details.

Streams are generated by tasks running on the end host, and switches in the network do not have direct access to information about the flow and tasks. For example, Baraat [17] uses a logically centralized task ID server to generate a globally unique ID for each task to perform task-aware scheduling policies in the network. It uses 26 bytes in a task-aware schedule in the network that informs the switch on the path which traffic belongs to which tasks. Obviously, this requires advanced features that are not supported by commodity switches and may conflict with other optimization techniques. Therefore, the small coflows should be scheduled in a practical manner.

Takeaway. The takeaway is that *small coflow scheduling must be autonomous, adaptive, and practical for commodity switches.* The most advanced task-/coflow-aware scheduling methods [14, 16, 17] can not meet the needs for network scheduling of small coflows.

3 ANALYSIS

This section first lists the requirements of small coflow scheduling and then proposes and analyzes the optimizing scheduling problem with constraints.

3.1 Requirements

The desirable properties of a scheduling system for small coflows are listed as follows, which are the design goals of OptaX.

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

OptaX optimizes network performance for small coflows by carefully monitoring and scheduling their flows. Given multiple tasks running in a network, OptaX decides which tasks are small coflows, determines appropriate TCP parameters for each small coflow, and optimizes the average completion times of all the small coflows. OptaX can also adapt to incomplete flow information and be complementary with current scheduling systems (such as Varys [16] and RAPIER [46]).

3.2 Goal and Problem

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 [42] 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 small coflows [7], 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.

$$minimize \sum_{i=1}^{n} d_i \tag{1}$$

s.t.

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

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

$$r_i^f(t) = \frac{w_i^f(t)}{rtt_i^f(t)}, \forall i, t, f \tag{1c}$$

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

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

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. OptaX follows the constraints in the optimization problem and solves the situation practically.

4 DESIGN

4.1 OptaX Algorithms

We describe the network optimization framework of OptaX in Algorithm 1. In general, OptaX identifies and approximately transmits small coflows' data in a FIFO mode by controlling ACKs at the receiver side.

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

We maintain a list of tasks sorted by time of arrival, meaning that the latest arriving task is added to the end of the list. The header of the task list has a higher priority, which means that network resources are preferentially allocated to it.

The Policy Calculator computes the acknowledgment delay time when every packet arrives to solve the problem formulated above. The detailed algorithm is demonstrated in Algorithm 2. The procedure CalcDelay calculates the delay time for each task. If a task is at the head of the task list, no delay will be added. 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_{all}^m denotes the overall throughput. When b_{all}^m 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 set to 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

$$delay = \min(n_t w_t / (\alpha C - b_{all}^m) - base\_rtt, Delay_{max}) \tag{2}$$

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 acknowledgment, and then the flow will continue to transmit. When an acknowledgment packet is about to send, the receive window of a task is calculated based on whether it is the head of

Algorithm 1 The OptaX Framework

```
1: procedure Ack(Packet p, Task t)
2:     suggested window = CalcWindow(t)
3:     send ACK packet for p
4: end procedure
5: procedure IsCongested( )
6:     b^m_all is the overall throughput
7:     if b^m_all ≥ αC then return True
8:     else return False
9:     end if
10: end procedure
    Main loop:
11: while true do
12:     if new packet p received then
13:         find task t which p belongs to
14:         if IsCongested() then
15:             delay_time = CalcDelay(t)
16:             set ack_timer(delay_time)
17:         else
18:             call Ack(p, t)
19:         end if
20:     end if
21:     if ack_timer.expire() then
22:         call Ack(p, t)
23:     end if
24:     for task in task_list do                ▷ Sorted by arrival times
25:         if not IsCongested() then
26:             send ACK of the task
27:         end if
28:     end for
29: end while
```

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:         if IsCongested() then return Delay_max
6:         else return delay calculated as Equation (2)
7:         end if
8:     end if
9: end procedure
```

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

Algorithm 3 The Window Suggesting Algorithm

```
procedure CalcWindow(Task t)
    w_t^old is the old window size
    w_t is the window size of task t
    if t is task_list.head then
        if b^m ≥ αC then
            w_t = max(w_t^old − 1, W_min)
        else if b^m ≤ αC/2 then
            w_t = w_t^old ∗ 2
        else
            w_t = w_t^old + 1
        end if
    else
        w_t = W_min
    end if
end procedure
```

4.2 OptaX Modules

Based on the design described in Section 4.1, we implement OptaX. When implementing a network layer task-aware scheduling system, we essentially solve the following sub-problems.

1. How to collect flow information without modifying applications

For a given flow, OptaX needs to know two kinds of information. The first is which flows belong to which tasks. The second is whether the flow is short enough so that the coflow might be small. Knowing the accurate flow sizes is helpful but not mandatory in OptaX.

2. How to calculate scheduling policies

First OptaX assigns small coflows with a priority higher than large tasks to avoid head-of-line blocking. Secondly, network resources are allocated according to the start time of each small coflowa. The earliest arriving task will get a larger window size and lower round-trip time than other tasks. This strategy looks like FIFO, but it is different because subsequent tasks are not completely blocked. The constraints $(1d)$ and $(1e)$ guarantee that all tasks can transmit smoothly, otherwise the upper layer applications may be alarmed. However, FIFO is not suitable for large tasks because of the heavy traffic of large tasks, once large tasks arrive early, the performance will be seriously reduced.

3. How to deliver and enforce the policies

Our goal is to design a task-aware scheduling system in the network, so OptaX uses two controllable attributes in TCP: window size and RTT. Each flow involves a sender and a receiver, and packets are transmitted between the sender and the receiver. The window size defines how much information the sender can send without receiving an acknowledgement. The recipient suggests an affordable window size. RTT can also be controlled by delaying ACK at the receiver.

Since $base\_rtt$ is the link latency, and $delay$ is the ACK delay time, so the real RTT is

$$rtt = base\_rtt + delay.$$

Figure 3 depicts the architecture of OptaX, 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.

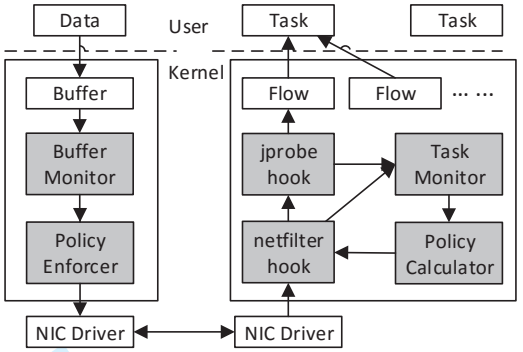


Fig. 3. OptaX architecture

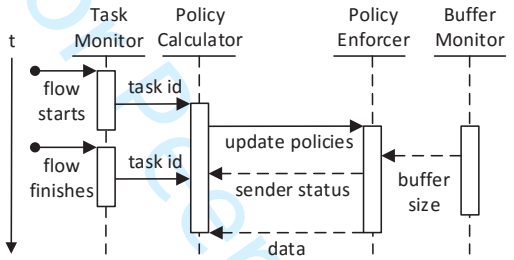


Fig. 4. Sequence diagram of OptaX

4.3 OptaX Workflow

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

- When a task starts a new flow on the receiver side, 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.
- The Buffer Monitor records send buffer status and delivers it to the Policy Enforcer on the sender side. The Policy Enforcer applies the suggested window size to the packets about to send and encodes flow size judgment in the outgoing packets.
- When a flow finishes, Task Monitor notifies the Policy Calculator. Until all flows of the task are finished, the task is completed and removed from the task list.

4.4 OptaX Architecture

We now describe the architecture of OptaX (comprising the four modules) in detail.

4.4.1 Task Monitor. To get the relationship between tasks and streams, Task Monitor tracks basic network functions, such as *tcp_recvmsg*. For example, a task monitor can record the PID for each task as it enters *tcp_recvmsg*, which is used as `task_id`. Traffic with the same `task_id` is considered concurrent. When the flow of funds begins, Task Monitor notifies the policy Calcu-lator 4-tuple (`task_id, source, destination, flow_size`*). The `source` and `destination` are the guide bars for traffic. The `flow_size` is optional because it is sometimes not known in advance.

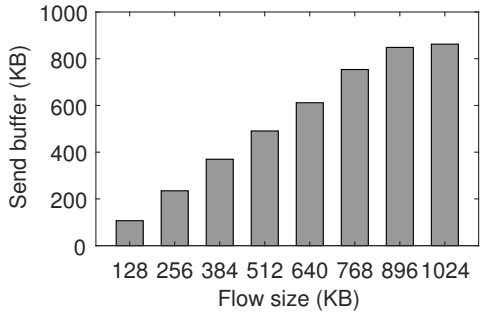


Fig. 5. sendbuffer indicates the size of small flows

At the end of the stream, it is removed from the traffic list for its related tasks. When the falling stream of the task is complete, the task is marked as full.

4.4.2 Buffer Monitor. On the sender side, sender buffering can tell us how much backlog data is waiting to be sent [1]. When the data is ready to be sent, system functions, such as *tcp_sendmsg*, are called, the data is copied from the user's space memory to the spatial send buffer, and the NIC (Network Interface Card) reads the buffer and transmits the data to the network. Therefore, the data backlogged in `sendbuffer` is the stream data ready to be transferred to the network, and we can infer the flow size from the send buffer state.

OptaX sets a threshold to filter large tasks. We add the backlog of bytes in `sendbuffer` plus the bytes that are transferred over the network to the threshold. Although traffic is transferred, if the sum of backlog bytes and transport bytes is greater than the threshold, we are confident that the stream is a long stream and involves a large task.

Figure5 depicts how the size of the send buffer varies with the size of the traffic when a stream sends its first few packets. The error send buffer size is 1MB, and when the traffic size is less than the send buffer capacity, the send buffer will be incomplete. When the current data packets are about to be sent, the sending buffer of traffic is filled with user data, so the size of the occupied buffer represents the size of traffic. Therefore, the send buffer is a good indicator of the traffic size. The traffic size threshold is not a fixed value. It depends on how we define small coflow. In our definition, small coflow means that all flows involved are less than 1MB, so we set a threshold of 1MB.

In addition, different applications and streams leave different footprints on the send buffer. Send buffer occupancy depends on traffic and specific application behavior. On the other hand, the data entry rate cannot fill these buffers or the maximum buffer size is not large enough to distinguish traffic, and the network is not a bottleneck. Although the send buffer is a good indicator of traffic size, it is not always accurate. Sometimes the send buffer cannot tell the traffic size in advance

We use *jprobe* and *netfilter* to perform tasks and buffer monitors. Jprobe is a probe tool provided by the Linux kernel, and netfilter is a series of hooks at different points in the Linux network stack. These hooks make it easy to monitor tasks and network stacks.

After being ready to collect the size of the send buffer and the size of the data sent, the Buffer Monitor can decide whether the flow is long or short. If all traffic to the task is short, the task is determined to be a "small coflow".

4.4.3 Policy Calculator. The Policy Calculator schedules incoming flows based on the algorithms described in Section 4.1.

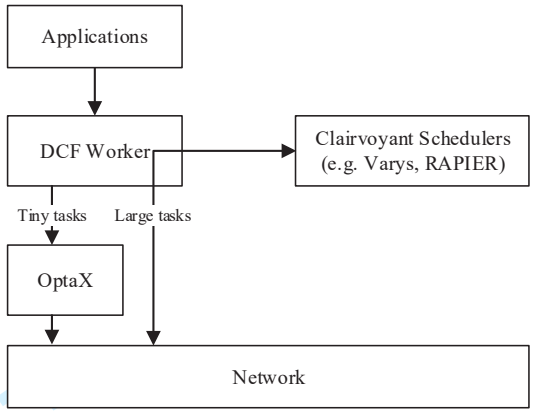


Fig. 6. Integrating with clairvoyant schedulers.

4.4.4 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 assigns the packets of short flows of small coflows with the highest priority and encodes their priorities into the Differentiated Services Code Point (DSCP) field of packet headers.
(2) The Policy Enforcer transfers packets in the window size as the Policy Calculator suggests.

5 COLLABORATIVE SCHEDULING

Production data centers typically have small coflows with short flows, and large tasks may require high bandwidth and consume other resources (e.g., CPU cycles). This section describes OptaX's collaboration with existing distributed computing frameworks (DCF).

5.1 Integrating OptaX with Existing Schedulers

Existing task-aware schedulers [12, 14, 16, 29, 44, 46] focus on significant tasks, especially large-scale data-parallel computing tasks. These centralized coflow schedulers are typically integrated with DCFs such as Hadoop and Spark. They collect coflow information from DCF workers, compute scheduling policies, and then enforce the policies to the network. The scheduling algorithm adopted is similar to the shortest job first (SJF), and its implementation method is either strict priority control or explicit rate limit.

As mentioned earlier, these schedulers are practical for large tasks, but not small ones. This leaves an opportunity for OptaX to cooperate with these centralized schedulers. OptaX is naturally capable of integrating with the centralized schedulers. OptaX and centralized schedulers are conceptually complementary.

According to the difference in utilized flow information, the centralized schedulers are in three types: clairvoyant schedulers [16, 46], non-clairvoyant schedulers [14, 44], and DAG-aware Schedulers [29].

5.2 Clairvoyant Schedulers

For clairvoyant scheduling schemes, the worker daemon collects and reports all kinds of task information to the scheduler. Therefore, the clairvoyant schedulers are aware of complete coflow

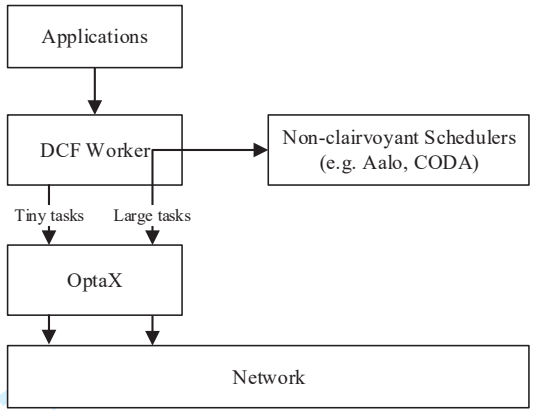


Fig. 7. Integrating with non-clairvoyant schedulers.

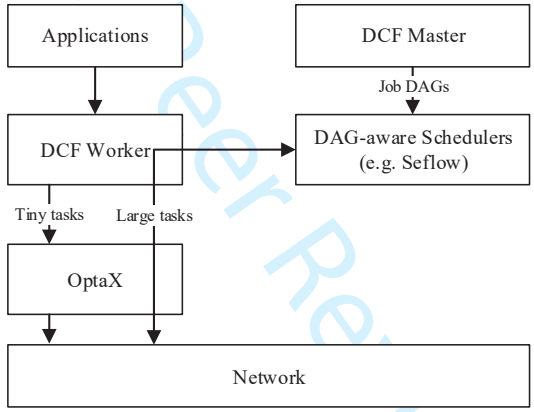


Fig. 8. Integrating with DAG-aware schedulers.

information, including coflow relationships and accurate flow sizes. Making global consistent scheduling policies introduces heavy computing load. The workers have to wait until the computation is done. This introduces two side-effects: (i) the communication and computation overhead of the centralized schedulers; (ii) the flow data is backlogged in the send buffers while scheduling synchronization. A widely adopted workaround is to bypass the tasks that comprise show flows.

Figure 6 shows how OptaX can be integrated with the clairvoyant schedulers. When integrating with clairvoyant schedulers, OptaX solely deals with the small coflows and leaves large coflows to the centralized schedulers. With the help of data backlogged in send buffers, OptaX can quickly identify the long flows and large tasks.

5.3 Non-clairvoyant Schedulers

For Non-clairvoyant scheduling schemes, the scheduler deals with incomplete application information, loosely coordinated with the workers. On the worker side, the flows are not blocked to wait for scheduling when they start. All new coflows are treated as short and initiatively scheduled as the highest priority. While coflows transmitting, the worker daemons periodically report their

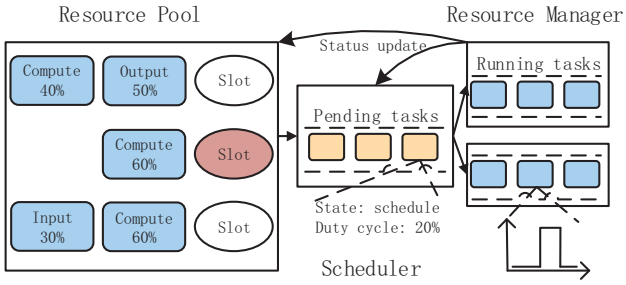


Fig. 9. Multi-resource scheduling. The network I/O (Input/Output) bandwidth resource scheduling is integrated with OptaX flow scheduling.

statistics to the scheduler. At the scheduler side, long coflows are demoted to lower priorities while their sizes increase, the scheduler calculates explicit sharing bandwidth for each coflow.

Figure 7 shows how OptaX can be integrated with non-clairvoyant schedulers. When a coflow starts, it is initialized with the highest priority and scheduled by OptaX. It goes beyond OptaX's scope, as the coflow's observed size (sent + backlogged) keeps increasing. Finally, the centralized scheduler takes over the scheduling work.

5.4 DAG-Aware Schedulers

For DAG-aware scheduling schemes, the scheduler knows not only the coflow itself, but also the relationship between coflows. DAG-aware coflow scheduling is a variant of clairvoyant scheduling. The difference is that DAG-aware coflow schedulers need the job runtime information from the job master node, as shown in Figure 8.

The DAG-aware scheduler changes the discipline that shortest coflow goes first. For DAG-aware scheduling, the priority of coflows also depends on their internal relations, not just their size. The scheduling decisions made by the DAG-aware scheduler may be conflicted with OptaX because OptaX is not aware of the inter-coflow relationships.

6 MULTI-RESOURCE SCHEDULING

In this section, we will introduce how to apply OptaX in practical scenarios by taking the Ursa block store [26] as a use case, where both small and large I/O tasks exist and multiple kinds of resources (such as compute resources, network input/output resources, etc.) need to be collaboratively scheduled.

Large-scale block storage provides virtual disks for various cloud applications and services. The state-of-the-art Ursa block store adopts a hybrid storage structure that places primary data on solid-state drives or SSDs (for high I/O performance) and stores backup data on hard-disk drives or HDDs (for low monetary cost). Although Ursa has been highly optimized in the storage architecture, it takes the underlying network layer as a *black box* and processes block I/O requests by sending/receiving packets through TCP/IP sockets. This makes Ursa inefficient in both latency and throughput for small I/O, mainly because the in-kernel network layer is agnostic to Ursa block store's I/O requests' priorities.

We observe that physical block devices (like SSDs and HDDs) do not guarantee ordering between committed writes. It is the responsibility of guest file systems and/or applications to correctly recover from crashes while enjoying block devices' parallelism. Therefore, we propose to adopt

OptaX to give the flows of small I/O tasks higher priority than the flows of large I/O tasks in Ursa, while also considering the utilization of other resources like CPUs.

To support multi-resource scheduling in practical scenarios like Ursa, we propose the *task model* which represents runtime task requirements including two parts: the state machine and the duty cycle. The state machine describes the progress of the task, including four states "schedule", "input", "compute" and "output". These states indicate the resource requirement variations of different task states: the "schedule" state is waiting at the scheduler side and consumes no resource; the "input' state causes ingress throughput; the "compute" state brings high CPU utilization; and the "output" state causes egress throughput.

Clearly, low CPU utilization implies more opportunities to share CPU resource with other tasks. Since tasks within the same state are homogeneous, the knowledge of task model can be obtained from the tasks of the job's previous rounds or the historical execution of the same application.

We further design a two-stage scheduling mechanism which enhances the collaborative OptaX network scheduler (as introduced in Section 5.1) with two extensions, namely, task placement and resource management. The multi-resource scheduling architecture is shown in Fig. 9, where (i) the Resource Pool maintains currents states of running tasks, (ii) the Scheduler finds a slot for the pending task according to its task model, and (iii) the Resource Manager monitors/controls resource usage, and updates task model only when its execution never interfered by other tasks.

Task placement. When there are adequate schedulable slots, the task placement algorithm works as a regular scheduler. Once no free slot available, it seeks a slot for the pending task that satisfies (i) the task running on it is not under "input" state and (ii) the CPU utilization is low. The first constraint allows the new task to read its input data while the former task is not competing with it for input bandwidth. The tasks assigned to the same slot are executed as a coarse-grained pipeline. The second constraint guarantees that CPU resources are not oversubscribed.

Resource management. Since multiple tasks may run on the same core, it is still possible to incur serve performance interference. We do a fine-grained resource control to avoid interference, by giving the former task with higher priority to get CPU resources.

7 EVALUATION

7.1 Methodology

7.1.1 Experiment Setup. We have implemented OptaX as a Linux kernel module and deployed it on our 37-machine testbed. The 37 physical machines are Dell PowerEdge R320 servers, each with a quad core Intel Xeons E5-1410 2.8GHz CPU, 24GB memory, and one 500GB Intel NVMe SSD, connected to a Pronto 3780 48-port 10GbE switch. The OS is Debian 6.

In our experiment, all files were stored in RAM to test network performance. File size determines whether the coflow is small or large. We use Redis, a high-performance memory base store, to set up user apps. One server is a Redis client in our test bench, while the other 36 servers are Redis servers. We run tasks on the client that read data from the remote servers. Because each task involves a set of inflows into the network, each task's completion time (that is, duration) is calculated when the last traffic is completed.

We use ICTCP [42] 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

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

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

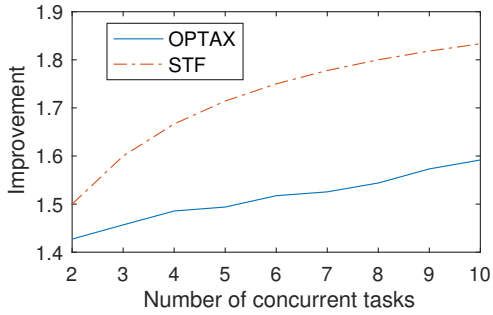


Fig. 10. Improvements in average task completion times of small coflows using OptaX over fair sharing. The omniscient STF's improvements are also included for reference.

- *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.

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

- Effectiveness: When only small coflows are present, OptaX achieves at least 1.4× 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 small coflows are mixed with large tasks, OptaX is at least 2.2× faster than fair sharing, and 1.2× than small coflows in highest priority.
- Compatibility: OptaX is compatible with other optimization solutions for large coflows, such as Varys and RAPIER. The integrated system consists of OptaX and Varys can improve as much as 1.93×.
- Overhead: OptaX brings negligible overhead to the system. The additional CPU and memory overhead OptaX introduced are positively relevant to the number of concurrent tasks, less than 1% and tens of KBs, respectively. The latency introduced by OptaX is less than 0.2%.

7.2 Effectiveness of OptaX

7.2.1 Performance Improvement. Figure 10 displays improvements under a fixed amount of traffic and volume for each task. All of these tasks are small coflows, each involving 8 streams, each 256KB in size, and the number of concurrent tasks ranging from 2 to 10. OptaX can transfer tasks at least 1.43× faster than fair sharing. Improvement factors alone increased from 1.43× to 1.59× and concurrent tasks increased from 2 to 10. The omniscient situation is that all the flow of information can be known in advance, and then we can choose the best scheduling plan, always with the smallest task first (STF).

Figure 11 displays improvements to the width of different tasks using OptaX over-empty sharing. These are 8 small streams running simultaneously, flowing the number of pertasks ranging from 2 to 14. 4 lines in Figure 11 are the total volume of tasks, which are 1MB, 2MB, 3MB and 4MB respectively. The improvement factor declined steadily in the internal [1.47, 1.77]. However, when

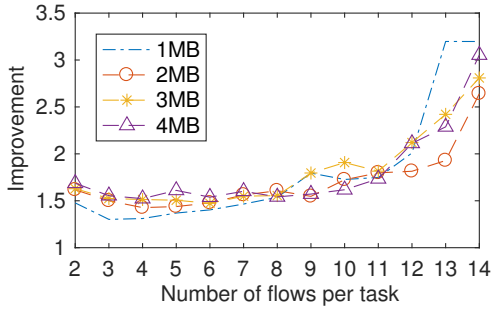


Fig. 11. Improvements in average task completion time using OptaX over fair sharing for varying task width (i.e., number of flows per task).

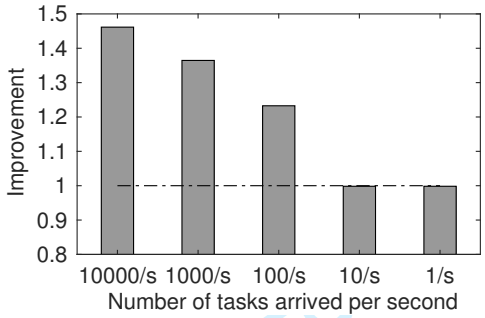


Fig. 12. OptaX's performance when the arrival rate of small coflows varies.

the width exceeds 12, severe congestion occurs in ICTCP, the goodput of ICTCP declines, so the improvement factor increases rapidly.

OptaX's performance also depends on the task's expedietic rate (illustrated in Fig. 12). The X-axis in the Figure 12 is the number of tasks per second, the Y-axis is a fair share of improvement factors. OptaX improves by up to 1.46× when a server is busy, such as 10,000 tasks per second. Performance improvements also decrease when task arrival rates drop. OptaX no longer reaps the benefits of performance when the rate drops to 10 tasks per second. This is because small coflows do not collide with each other on the network when the arrival rate is low.

These experiments show how OptaX improves network performance when only a small amount of coflow exists. In a real-world data center environment, small coflows are mixed with large tasks. We answer the following questions: (i). When mixing large tasks, how much co-flow will the small coflow perform? (ii). How large tasks be delayed when small co-flow always preempt them?

We demonstrate how OptaX performs tasks when mixed with large tasks in Figure 13. Tasks are divided into four categories based on two dimension metrics (size and width). The mixed ratio between these categories comes from the introduction of Facebook tracking in literature [16]. Detailed reports are listed in the Table 1.

When small coflows are mixed with large tasks, the plan always assigns small coflows to small coflows that take precedence over larger tasks. In the legend of Figure 13, "O", this means that small common flows are scheduled by OptaX, while large tasks are fairly shared, in the context of their higher priority. Compared to "O", "F" refers to the priority of small capital flows, which are

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 1. The mixing ratios of small and large coflows from the Facebook trace [16].

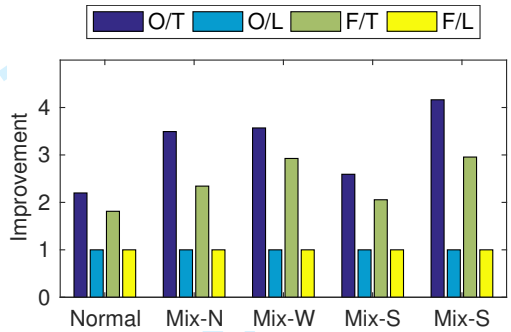


Fig. 13. OptaX's performance under different mixing ratios with comparison to only assigning small coflows with high priority.

scheduled in a fair and shared manner. "T" and "L" are improvement factors for small coflows and large tasks.

In Figure 13, O/T represents small coflow OptaX, and F/T represents fair sharing between small coflows. By looking at Figure 13, we can draw four conclusions.

(1) When the small and large coflows are mixed, OptaX can improve the performance of small coflows significantly. The improvement factors of small coflows 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 small coflows ("F/T"), OptaX outperforms by about 20%. The improvement of OptaX depends on how the small and large coflows are mixed.
(3) The degradation of large tasks is less than 0.1% when they are preempted by small coflows ("O/L" and "F/L" in Figure 13).
(4) The black dot dash line line in Figure 13 is the overall improvement counts for both small and large coflows. Since most network traffic is generated by large task, if we only optimize small coflows and abandon large one, the average overall improvement is only 1.02×.

As a result, OptaX can optimize small coflow symbols with little impact on large tasks. However, each form optimization of large tasks is left to other technologies, such as Varys, RAPIER, when the full concurrent information can be known, and Aalo when the information is unknown. When we bring OptaX and Varys together in §7.3, we will evaluate how the system works.

7.2.2 Scheduling Overhead. Because OptaX is implemented as a background kernel module, it is difficult to measure overhead directly. In our front experiments, we do not observe obvious CPU and memory resource consumption. When OptaX does not output perforation improvements, its

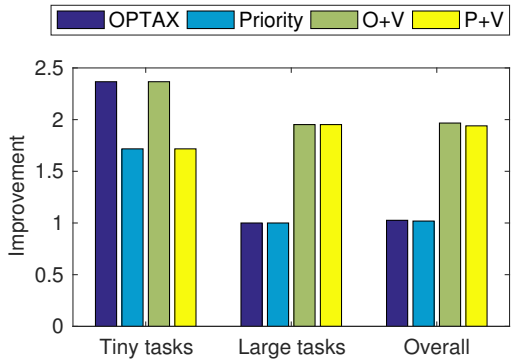


Fig. 14. The overall performance improvement when integrate OptaX with Varys.

overhead can be measured by performance degradation. Since Figure 12, the last two columns of OptaX will add a bit of overhead to the task, which is about 0.2%. In summary, OptaX's resource consumption and overhead are negligible.

7.3 Collaborative Scheduling

We integrate OptaX with Varys [16] and evaluate its perforation. Performance is compared to fair sharing. In our tests, the Varys scheduler had global process knowledge, and none of the schedulers incurred any additional overhead. According to the "Normal" recipe in Table 1, we synthesize the trace.

In the legend of Figure 14, "OptaX" means that small processes are scheduled by OptaX, while large tasks are sharing fairly; "Priority" means that small co-flow are high priority and are fairly shared within small and large coflows; OptaX has a performance of 1.38× only high priority of small coflows.

However, transmission time in the network is dominated by large tasks. If we do not optimize large tasks, the overall performance improvements are only 1.03× and 1.02× for OptaX and priority, respectively. When we integrate with Varys, OptaX handles small processes, while Varys focuses on large tasks. "O+V" means that OptaX integrates with Varys; "P+V' means that small common flows are at the highest priority and large tasks are scheduled by Varys. Improvements to both "O+V" and "P+V" exceed 1.93×, and "O+V" is slightly higher. Figure 14 shows that OptaX, together with Varys, can better optimize small concurrent and total performance.

7.4 Multi-Resource Scheduling for Ursa

As introduced in §6, OptaX can be extended for multi-resource scheduling in practical application scenarios, e.g., improving the small I/O performance of the Ursa block store without affecting its large I/O performance. To verify OptaX's effectiveness for Ursa, we use 18 machines to run Ursa storage servers (with three-way replication), 18 machines to run Ursa clients, and one machine to run Ursa master.

We evaluate the mean I/O latency and throughput of Ursa with OptaX. On each of the 18 Ursa client machines, we run two VMs each mounting a 100GB virtual disk provided by Ursa block store. One VM runs the `fio` benchmark with block size = 4KB for latency test, and meanwhile the other VM runs `fio` with block size = 4MB for throughput test. The qd (queue depth) is 16 for both tests. We compare the results to that of the original Ursa without OptaX.

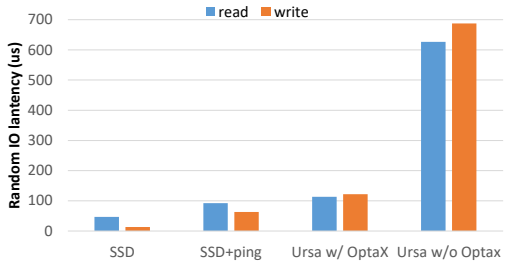


Fig. 15. Random I/O latency of Ursa with and without OptaX (BS = 4KB).

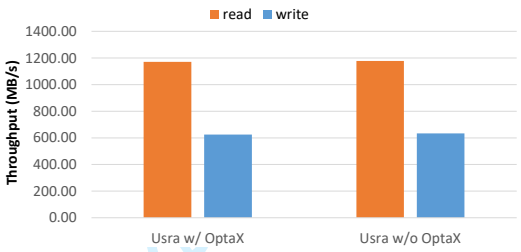


Fig. 16. Throughput of Ursa with and without OptaX (BS = 4MB).

The latency result is shown in Fig. 15, where we also include the latencies of local SSD and SSD plus network ping for comparison. the latencies of Ursa with OptaX for both reads and writes are about 100 microseconds, close to the optimal read/write latencies (SSD plus ping) and significantly lower than that of the original Ursa without OptaX. This is mainly because OptaX effectively schedules the short flows of small I/O to have higher priority than the large flows. The throughput result is shown in Fig. 16, where Ursa with OptaX achieves similar throughput as Ursa without OptaX. The throughput of writes is only half that of reads because Ursa adopts passive replication [5] for large writes. The result proves that OptaX can effectively adapt to practical application scenarios where short flows co-exist with large flows in the network.

8 RELATED WORK

8.1 Flow scheduling

Various scheduling schemes have been developed to meet different design goals, such as congestion control [6, 10, 34, 42], performance isolation [3, 36], failure recovery [45] and flow scheduling [4, 7, 21, 33].

These studies focus on scheduling independent flows to improve network utilization and reduce the average flow completion time (FCT). For example, PDQ [21] and pFabric [4] are flow scheduling schemes to minimize FCT by assigning flows with priorities. Congestion avoidance techniques, such as ICTCP [42] 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, also used by OptaX.

PASE [33], Qjump [20], NUMFabric [34] show different kinds of implementation of flow schedulers. Karuna [10] schedules mixed flows with multiple goals. 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 [2] and MicroTE [8], cannot directly be used to optimize coflows either.

8.2 Task-aware flow scheduling

In recent years, the network resources co-scheduling and computing tasks have been highly valued by scientific researchers. It is recommended that the centralized scheduler be task-aware [11, 13, 15, 16, 18, 19, 23–25, 27, 30–32, 40, 43, 46], better enable the use of the network. The representative works we are discussing are as follows. Orchestra [15] optimize transmission times by understanding communication patterns and using FIFO. Coflow [13] define a network abstraction of a parallel flow. Varys [16] and RAPIER [46] apply the concept of coflow to its network optimization. They both use centralized schedulers, which incur significant overhead costs for short traffic. RAPIER considers concurrent scheduling, which can be easily implemented for OptaX by routing together with existing sources (e.g., XPath [22]).

HadoopWatch [35] and FlowProphet [41] expose coflow information from distributed computing frameworks to coflow schedulers. They can only work on specific distributed computing frameworks. Baraat [17] is a low overhead, decentralized task-aware scheduling scheme, which needs a centralized server to sort tasks in FIFO order. Unfortunately, Baraat can not be implemented using existing commodity switches. Aalo [14] schedules coflow without prior knowledge and perform well even for small coflows by avoiding coordination. However, its rate control on short flows is not accurate. CODA [44] identifies coflow with no modification on distributed computing frameworks. Refs. [37, 39] improves the bound of completion times of total weighted completion time of coflows.

Similarly, centralized concurrent schedulers are often limited to data parallel computing frameworks and do not fit into a large number of OLDI applications. Theoretical work [25, 38] provide approximation algorithm for process scheduling problems, which have proven to be NP-Hard issues.

OptaX is superior to the previous task-aware scheduler in three ways. First, by monitoring syscalls and buffering footprints, OptaX accurately schedules a large number of broccoli in a decentralized manner. Second, OptaX works with TCP without assuming unrealistic support from the commodity switch. Third, OptaX saves a lot of engineering work by eliminating the need to modify user applications.

9 CONCLUSION

This paper focuses on the problem of large-scale network scheduling and presents a decentralized and adaptive small coflow monitoring and scheduling service called OptaX. The key design of OptaX are two-fold. (i) OptaX obtains necessary information without any modification of user applications. (ii) OptaX schedules flows of small coflows by adjusting the window size and the time to send ACKs. We have implemented OptaX as a Linux kernel module, and evaluated it on our 37-server testbed. The results show that OptaX can efficiently improve the performance of small coflows. OptaX is at least 2.2× faster than fair sharing, and 1.2× faster than only assigning small coflows with the highest priority. In our future work, we will further improve the latency performance by integrating OptaX with DPDK/SPDK techniques, and apply the key idea of OptaX to RDMA-enabled network scheduling.

ACKNOWLEDGMENT

This work is supported by the National Key Research and Development Program of China (No. 2016YFB1000100), and the National Natural Science Foundation of China (No. 61772541 and 61872376). A preliminary version [28] of this work was published in IEEE INFOCOM 2016. We

thank Kai Chen and Wei Bai (SING@HKUST) and Ling Liu (DiSL@Georgia Institute of Technology) for discussion on the design and implementation of OptaX.

REFERENCES

[1] Agache, A., and Raiciu, C. Oh flow, are thou happy? tcp sendbuffer advertising for make benefit of clouds and tenants. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)* (Santa Clara, CA, 2015), USENIX Association.

[2] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2010), NSDI'10, USENIX Association, pp. 19–19.

[3] 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 *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 253–266.

[4] Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 435–446.

[5] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., Caro, A. D., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolic, M., Cocco, S. W., and Yellick, J. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018* (2018), R. Oliveira, P. Felber, and Y. C. Hu, Eds., ACM, pp. 30:1–30:15.

[6] Bai, W., Chen, K., Wu, H., Lan, W., and Zhao, Y. Pac: Taming tcp incast congestion using proactive ack control. In *2014 IEEE 22nd International Conference on Network Protocols* (Oct 2014), pp. 385–396.

[7] Bai, W., Chen, L., Chen, K., Han, D., Tian, C., and Wang, H. Information-agnostic flow scheduling for commodity data centers. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association, pp. 455–468.

[8] Benson, T., Anand, A., Akella, A., and Zhang, M. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies* (New York, NY, USA, 2011), CoNEXT '11, ACM, pp. 8:1–8:12.

[9] Benson, T. A., Anand, A., Akella, A., and Zhang, M. Understanding data center traffic characteristics. In *ACM SIGCOMM Workshop: Research on Enterprise Networking* (January 2009), Association for Computing Machinery, Inc.

[10] Chen, L., Chen, K., Bai, W., and Alizadeh, M. Scheduling mix-flows in commodity datacenters with karuna. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 174–187.

[11] Chen, L., Cui, W., Li, B., and Li, B. Optimizing coflow completion times with utility max-min fairness. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications* (April 2016), pp. 1–9.

[12] Chowdhury, M., Khuller, S., Purohit, M., Yang, S., and You, J. Near optimal coflow scheduling in networks. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019* (2019), C. Scheideler and P. Berenbrink, Eds., ACM, pp. 123–134.

[13] Chowdhury, M., and Stoica, I. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2012), HotNets-XI, ACM, pp. 31–36.

[14] Chowdhury, M., and Stoica, I. Efficient coflow scheduling without prior knowledge. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 393–406.

[15] Chowdhury, M., Zaharia, M., Ma, J., Jordan, M. I., and Stoica, I. Managing data transfers in computer clusters with orchestra. In *Proceedings of the ACM SIGCOMM 2011 Conference* (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 98–109.

[16] Chowdhury, M., Zhong, Y., and Stoica, I. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 443–454.

[17] Dogar, F. R., Karagiannis, T., Ballani, H., and Rowstron, A. Decentralized task-aware scheduling for data center networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 431–442.

[18] Fu, Z., Song, T., Wang, S., Wang, F., and Qi, Z. Seagull – a real-time coflow scheduling system. In *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing* (Nov 2015), pp. 540–545.

[19] Gao, Y., Yu, H., Luo, S., and Yu, S. Information-agnostic coflow scheduling with optimal demotion thresholds. In *2016 IEEE International Conference on Communications (ICC)* (May 2016), pp. 1–6.

[20] Grosvenor, M. P., Schwarzkopf, M., Gog, I., Watson, R. N. M., Moore, A. W., Hand, S., and Crowcroft, J. Queues don't matter when you can jump them! In *12th USENIX Symposium on Networked Systems Design and Implementation*

(NSDI 15) (Oakland, CA, 2015), USENIX Association, pp. 1–14.

[21] Hong, C.-Y., Caesar, M., and Godfrey, P. B. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 127–138.

[22] 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 *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association, pp. 15–28.

[23] Huang, X. S., Sun, X. S., and Ng, T. E. Sunflow: Efficient optical circuit scheduling for coflows. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies* (New York, NY, USA, 2016), CoNEXT '16, ACM, pp. 297–311.

[24] Jiang, J., Ma, S., Li, B., and Li, B. Tailor: Trimming coflow completion times in datacenter networks. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)* (Aug 2016), pp. 1–9.

[25] Khuller, S., and Purohit, M. Brief announcement: Improved approximation algorithms for scheduling co-flows. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2016), SPAA '16, ACM, pp. 239–240.

[26] Li, H., Zhang, Y., Li, D., Zhang, Z., Liu, S., Huang, P., Qin, Z., Chen, K., and Xiong, Y. URSA: hybrid block storage for cloud-scale virtual disks. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019* (2019), G. Candea, R. van Renesse, and C. Fetzer, Eds., ACM, pp. 15:1–15:17.

[27] Li, Y., Jiang, S. H.-C., Tan, H., Zhang, C., Chen, G., Zhou, J., and Lau, F. C. M. Efficient online coflow routing and scheduling. In *Proceedings of the 17th ACM International Symposium on Mobile Ad Hoc Networking and Computing* (New York, NY, USA, 2016), MobiHoc '16, ACM, pp. 161–170.

[28] Li, Z., Zhang, Y., Li, D., Chen, K., and Peng, Y. OPTAS: decentralized flow monitoring and scheduling for tiny tasks. In *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016* (2016), IEEE, pp. 1–9.

[29] Li, Z., Zhang, Y., Zhao, Y., and Li, D. Efficient semantic-aware coflow scheduling for data-parallel jobs. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)* (Sept 2016), pp. 154–155.

[30] Luo, S., Yu, H., and Li, L. Decentralized deadline-aware coflow scheduling for datacenter networks. In *2016 IEEE International Conference on Communications (ICC)* (May 2016), pp. 1–6.

[31] Luo, S., Yu, H., Zhao, Y., Wang, S., Yu, S., and Li, L. Towards practical and near-optimal coflow scheduling for data center networks. *IEEE Transactions on Parallel and Distributed Systems 27*, 11 (Nov 2016), 3366–3380.

[32] Ma, S., Jiang, J., Li, B., and Li, B. Chronos: Meeting coflow deadlines in data center networks. In *2016 IEEE International Conference on Communications (ICC)* (May 2016), pp. 1–6.

[33] Munir, A., Baig, G., Irteza, S. M., Qazi, I. A., Liu, A. X., and Dogar, F. R. Friends, not foes: Synthesizing existing transport strategies for data center networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 491–502.

[34] Nagaraj, K., Bharadia, D., Mao, H., Chinchali, S., Alizadeh, M., and Katti, S. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 188–201.

[35] 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 *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications* (April 2014), pp. 19–27.

[36] Perry, J., Ousterhout, A., Balakrishnan, H., Shah, D., and Fugal, H. Fastpass: A centralized "zero-queue" datacenter network. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 307–318.

[37] Qiu, Z., Stein, C., and Zhong, Y. Minimizing the total weighted completion time of coflows in datacenter networks. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures* (2015), ACM, pp. 294–303.

[38] Qiu, Z., Stein, C., and Zhong, Y. Minimizing the total weighted completion time of coflows in datacenter networks. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2015), SPAA '15, ACM, pp. 294–303.

[39] Shafiee, M., and Ghaderi, J. An improved bound for minimizing the total weighted completion time of coflows in datacenters. *IEEE/ACM Transactions on Networking (TON) 26*, 4 (2018), 1674–1687.

[40] Susanto, H., Jin, H., and Chen, K. Stream: Decentralized opportunistic inter-coflow scheduling for datacenter networks. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)* (Nov 2016), pp. 1–10.

[41] 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 *2015 IEEE 35th International Conference on Distributed Computing Systems* (June 2015), pp. 349–358.

[42] Wu, H., Feng, Z., Guo, C., and Zhang, Y. Ictcp: Incast congestion control for tcp in data center networks. In

Proceedings of the 6th International COnference (New York, NY, USA, 2010), Co-NEXT '10, ACM, pp. 13:1–13:12.

[43] Yu, R., Xue, G., Zhang, X., and Tang, J. Non-preemptive coflow scheduling and routing. In *2016 IEEE Global Communications Conference (GLOBECOM)* (Dec 2016), pp. 1–6.

[44] Zhang, H., Chen, L., Yi, B., Chen, K., Chowdhury, M., and Geng, Y. Coda: Toward automatically identifying and scheduling coflows in the dark. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 160–173.

[45] 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 *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association, pp. 529–542.

[46] 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 *2015 IEEE Conference on Computer Communications (INFOCOM)* (April 2015), pp. 424–432.

Received November 2021

Cover Letter

Existing coflow scheduling methods either require a centralized controller or require special switch/NIC support. In contrast, this paper proposes a decentralized method (OPTAX) which allows end hosts autonomously perform coflow scheduling without a centralized controller. The key idea is to leverage the sendbuffer information in the kernel to adaptively assign high priorities to urgent flows. Therefore, the contribution of this paper is significant for the TAAS audience.

This journal submission presents substantial improvements of a previous version published in IEEE INFOCOM 2016. This journal version is different from the conference paper (available at the following link) in a number of aspects:
http://www.cse.ust.hk/~kaichen/papers/optas-infocom16.pdf
(1) In Section 5, we integrate OPTAX with existing distributed computing frameworks for collaborative scheduling.
(2) In Section 6, we realize multi-resource scheduling for the Ursa block store.
(3) In Section 7.4, we compare the performance of Ursa with and without OPTAX scheduling.
(4) In Section 8.3, we add more discussion (e.g., the coflow starvation problem).
Besides, we have revised and improved the description and explanation of the ideas and algorithms.