Chronos: Meeting Coflow Deadlines in Data Center Networks

Shiyao Ma*, Jingjie Jiang*, Bo Li*, and Baochun Li†
*Department of Computer Science and Engineering, Hong Kong University of Science and Technology
†Department of Electrical and Computer Engineering, University of Toronto

***Abstract*—Guaranteed performance for data-parallel applications is important for both service providers and cloud data centers that host such services. A job of data-parallel applications involves communication among multiple machines to transmit intermediate results. Such communication comprises a collection of parallel flows, which is abstracted as a *coflow* in recent proposals. In this paper, we study the problem of meeting deadlines for coflows in data center networks. Existing flow-level scheduling schemes are insufficient to guarantee the coflow-level performance, since a coflow can meet its deadline only when all its constituent flows finish on time. Due to the scarce bandwidth on the network bottleneck, it is vital to coordinate concurrent coflows to meet as many deadlines as possible. We present *Chronos*, a scheduling framework that captures the correlation of flows belonging to a coflow, and handles the resource allocation among multiple concurrent coflows. *Chronos* is *work-conserving* and *starvation-free* without integrating complicated admission control mechanisms. We show via extensive simulations on ns3 that *Chronos* can make $1.6\times$ more coflows meet their deadlines compared to flow-level schemes.**

I. Introduction

With the prevalence of data-parallel computing frameworks, online service providers, such as Google and Facebook, have moved their applications to data centers. The real-time nature of these applications results in the need to serve users in a timely fashion since longer response times can lead to significant financial loss [1]. The primary goal of service providers is thus to meet service level agreements (SLAs) and satisfy user requests within the corresponding *hard* deadlines. Since data-parallel applications [2] transfer large amounts of data among different machines across the data center network, their performance highly depends on the communication quality. A Hive/MapReduce trace at Facebook show that the communication time accounts for more than 25% of job durations for more than 40% of jobs [3]. It is thus important to meet the communication requirements for the bundle of flows that transfer intermediate results of a given job.

The recently proposed *coflow* abstraction [3] depicts the multiple concurrent flows that transmit intermediate results among different computation stages of a given job. A coflow is not considered completed until all its constituent flows finish. In other words, all the concurrent flows of a coflow share a common performance goal. Nevertheless, existing job schedulers [4], [5] have only focused on allocating computation resources, whereas most network schedulers have merely focused on flow-level performance [6], [7]. Without coflow semantics, if only a fraction of a coflow's constituent flows meet deadlines, the coflow as a whole still misses its deadline. Therefore, the user-perceived performance is not improved even when the flow performance is improved. To make things worse, the limited bandwidth can be wasted on flows of the already late coflows, which potentially slows down other coflows that could have met their deadlines otherwise.

In this paper, we study the deadline-driven coflow scheduling problem with the objective of minimizing the number of late coflows. Distinguished from flow scheduling, coflow scheduling has to face unique challenges. *Firstly*, the group of parallel flows constituting a coflow are correlated with each other. It is thus necessary to treat them as a whole and collect coflow-level information to make proper scheduling decisions. *Secondly*, edge networks in current data centers still experience severe congestions in peak hours [8]. The limited bandwidth on congested links needs to be carefully allocated to improve the overall application throughput. *Finally*, when workloads are imbalanced among servers, not all the coflows traverse through the hot servers can finish on time. One has to select a subset of coflows to guarantee their performance, while avoiding to starve the unselected coflows at the same time.

To meet the challenges discussed above, we design a coflow scheduling framework, *Chronos*, to conduct deadline-driven coflow scheduling. *Chronos* tries to allocate bandwidth to flows in proportion to their sizes such that all the flows in a coflow can finish at the same time. To accommodate as many coflows as possible, *Chronos* allocates the *bare* amount of bandwidth to a coflow, which is sufficient for it to finish just at its deadline. When the network is lightly loaded, *Chronos* further distributes the idle bandwidth to flows proportionally to ensure the scheduling is *work-conserving*. When the network is heavily loaded, *Chronos* selects a maximal subset of coflows to meet their deadlines. An admission control mechanism [3] simply rejects or suspends the unselected coflows, which leads to frequent transmission failures. Instead, we embrace limited multiplexing to flexibly reserve a small amount of bandwidth for unselected coflows to guarantee *starvation-free* scheduling. Furthermore, *Chronos* gracefully degrades to a weighted fair sharing scheduler if no deadline information is available.

The research was supported in part by grants from RGC under the contracts 615613 and 16211715, a grant from NSFC and Guangdong joint project under the contract U1301253, and the NSERC Strategic Networks grant titled "Smart Applications on Virtual Infrastructure."

978-1-4799-6664-6/16/$31.00 ©2016 IEEE

The main contribution of this paper lies in the scheduling object shifted from flows to coflows. By inspecting the correlations among flows in a coflow, we more efficiently allocate bandwidth to flows on the same link. Furthermore, we focus on the timeliness of coflows rather than the average completion time of coflows like in existing schemes [9]. *Chronos* is work-conserving and starvation-free without involving a complicated admission control mechanism like in [3] such that a coflow does not have to pause and resume from time to time. Extensive experimental results demonstrate that *Chronos* is effective both in meeting coflow deadlines and reducing coflow completion times under various network settings.

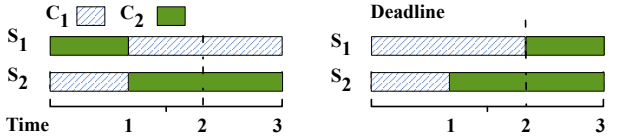
II. Related Work

Flow-level scheduling: Existing scheduling schemes in data center networks have extensively focused on flow-level scheduling. Without deadline information, scheduling schemes try to minimize flow completion times. To achieve such a goal, DCTCP [10] adjusts sending windows based on the level of congestions so that queues at switches remain short. pFabric [6] adopts the *shortest remaining size first* strategy to conduct priority-based scheduling and rate control. With flow-level deadline information, PDQ [7] approximates *earliest deadline first* strategy by permitting preemptive scheduling. Nevertheless, improving flow level performance metric does not imply optimizing coflow level performance. With respect to maximizing the number of punctual coflows, it is likely that although a large portion of flows belonging to a coflow meet the deadline, the slowest flow still misses the deadline, preventing the whole coflow from finishing on time. Lack of coflow information, all the schemes discussed above cannot guarantee coflow-level performance.

Coflow scheduling: Realizing the deficiencies of flow scheduling, recent proposals such as Varys [3], Rapier [11] and Aalo [9] try to integrate coflow semantics. Rapier [11] aims to minimize the average coflow completion time through joint scheduling and routing. Aalo [9] further schedules coflows without prior knowledge. Varys [3] has separately designed two set of strategies to minimize the average completion time and the number of late coflows respectively. Although sharing the same objective, the main difference between *Chronos* and Varys lies in that instead of embracing a complicated admission control mechanism, *Chronos* leverages multiplexing to avoid starvation such that any coflow is no longer *paused* or *canceled*. By combining priority-based scheduling and limited multiplexing, *Chronos* is able to ensure that high priority coflows can meet their deadlines and low priority coflows can proceed continuously at the same time.

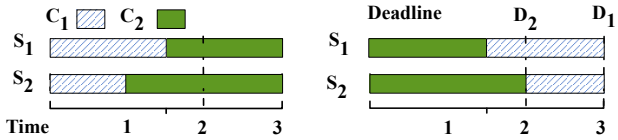
III. Background and Motivation

We abstract data center fabric as a giant switch with non-blocking internal data transmission. Each uplink is associated with an ingress port of a sender's network interface card, and each downlink is associated with an egress port of an receiver's Top-of-Rack switch. This abstraction represents the majority of current data centers, where congestion is free in the core but common on the edge, that is, on the uplinks and downlinks [8]. Under such a network model, specific network topologies [12], shed no influence on the design of scheduler. Besides, dynamic routing [11] would bring no benefit since congestion only occurs at edge links which cannot be bypassed.



(a) Flow-level (b) Coflow-level

Fig. 1. The difference between flow-level and coflow-level scheduling: the flows belonging to the first coflow are in blue (diagonal fill), while the flows of the second coflows are in green (solid fill). Under the smallest remaining time first flow-level scheduling, no coflow can meet its deadline at time 2, whereas one coflow can finish in time under coflow-level scheduling.



(a) Completion time (b) Deadline

Fig. 2. The example to show that minimizing coflow completion times is insufficient for deadline-driven scheduling: the average completion time of the two coflows is minimized in Fig. 2(a), but one coflow misses its deadline when it is possible for two coflows to finish in time as in Fig. 2(b).

Although some delay-insensitive applications have no specific deadlines, meeting deadlines are important for real-time applications, such as web search and interactive machine learning. For instance, the flows that transmit the relevant data of the query response for a web search job constitute a coflow. In a typical partition-aggregate model, these flows are first generated by workers, then reduced to an aggregator. Only after all the flows finish will the final search result be constructed and sent back to users. This implies no matter how quickly the first flow finishes, the response time of a query is determined by the slowest flow in that bundle.

We next illustrate the advantage of coflow scheduling through the following example. Consider the two coflows shown in Fig. 1, each of which consists of a flow of size 1 and a flow of size 2. Both coflows must finish after 2 time units, despite flows are bottlenecked on the uplinks with unit capacity. It is clear to see that at most one coflow can meet the deadline. Without coflow semantics, a flow-level scheme, such as the *smallest remaining size first* strategy in pFabric, would allocate bandwidth as in Fig. 1(a). As a result, neither of the two coflows can finish in time. With coflow semantics, a possible schedule is shown in Fig. 1(b), where flows belonging to one coflow are prioritized. In this case, one coflow can meet its deadline, while the other coflow is not slowed down.

The scheduling schemes to minimize coflow completion times, however, are still insufficient. As shown in Fig. 2, the deadline for the first coflow is 3, while the deadline for the second coflow is 2. A schedule to minimize coflow

completion times would prioritize the second coflow since its remaining time is smaller [3]. Indeed, the average coflow completion time is only 2.25, but the second coflow misses its deadline. In contrast, a deadline-driven strategy would allocate enough bandwidth to coflows to finish just in time as shown in Fig. 2(b): although the average completion time increases to 2.5, both coflows successfully meet their deadlines.

IV. Design

A. Scheduling Deadline-Driven Coflows

A coflow c_k is a set of flows f_{ij}^k, which sends data with size s_{ij}^k from port i to port j of the virtual switch. Denote the rate of f_{ij}^k at time t as $b_{ij}^k(t)$. c_k's completion time, denoted as t_k, then needs to satisfy:

$$\int_0^{t_k} b_{ij}^k(t)\ dt \geq s_{ij}^k \qquad \forall\ i,j \tag{1}$$

Namely, t_k is determined by when the last flow of the coflow finishes. c_k must finish before d_k to satisfy SLAs. Our objective is to maximize the total number of coflows meeting their deadlines, which is equivalent to minimizing the number of late coflows. The coflow scheduling problem is NP-hard since it can be reduced to the concurrent open shop scheduling problem [3] with the objective of minimizing the number of late jobs. Although 2-approximate algorithms exist for offline scheduling, the information of coflows is unavailable until it arrives in the system in practice. In other words, our scheduler needs to operate in an online fashion.

To accommodate as many active coflows as possible, *Chronos* allocates each flow f_{ij}^k with the *bare* bandwidth, whose value is $b_{ij}^k = s_{ij}^k/d_k$. This value merely ensures that every single flow of a coflow finishes exactly at the common deadline so that there would be more bandwidth available for incoming coflows to transmit data. This is the key to accommodating as many punctual coflows as possible. Nevertheless, since network bandwidth is rather limited, even allocating bare bandwidth might be infeasible due to harsh bandwidth contention. In this case, we need to select a subset of coflows to meet communication requirements. We next present the architecture of our scheduling framework.

B. Architecture

Chronos adopts a central scheduler (shown in Fig. 3), which acquires the information of a coflow once it arrives in the system, namely, when all its data are ready for transmission. Sender hosts are responsible for registering the essential information to the scheduler, including which coflow a flow belongs to, the size of a flow and the associated communication deadline requirements. Note that *Chronos* gracefully degrades to a flow-level scheduler when lacking some of the essential information as discussed in Sec. IV-D.

When the network load is heavy or imbalanced, it is probable that there exists no schedule to accommodate all the coflows without missing their deadlines. Existing schemes usually rely on admission control to select a subset of coflows

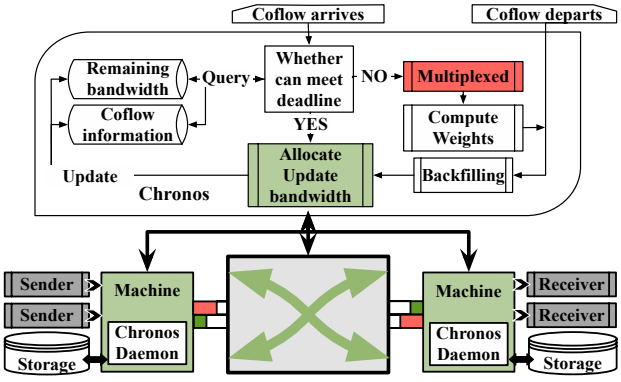


Fig. 3. The architecture of *Chronos* in a data center.

that can fit into the system without missing their deadlines and reject all the remaining coflows [3]. The advantage of such a scheme is that link bandwidth are all utilized by coflows that can meet their deadlines. Nevertheless, since unselected coflows have to resubmit requests continuously, their completion times could be significantly prolonged, leading to unfairness and dissatisfaction of users. To make things worse, some coflows may have to wait perpetually for other coflows to finish, leading to severe starvation.

Instead, *Chronos* schedules coflows by adopting relaxed priority scheduling with limited multiplexing. Since pure multiplexing raises coflow completion times [3], we prioritize a subset of selected coflows to guarantee they can meet deadlines, while making the unselected (multiplexed) coflows share the residual bandwidth. We conceptually divide the bandwidth B_i on each port P_i into two segments: N_i is used for *normal* data transmission of high priority coflows, while R_i is the *reserved* bandwidth used for *multiplexing*. The ratio between the two segments is predefined by the scheduler, and can be adjusted dynamically based on network status. We next illustrate the scheduling algorithms of *Chronos* in detail.

C. Algorithms

For a coflow c_k with the highest priority among all the unscheduled coflows, *Chronos* tries to allocate the bare bandwidth b_{ij}^k for each flow f_{ij}^k of this coflow. For a flow f_{ij}^k, if the idle bandwidth on both N_i and N_j is larger than b_{ij}^k, the scheduler distributes the bandwidth to f_{ij}^k and updates the remaining bandwidth on N_i and N_j (line 10 of Alg. 1). If either N_i or N_j does not has enough idle bandwidth, the corresponding coflow cannot meet its deadline. We mark it as a *multiplexed* coflow (line 6 of Alg. 1), and make it share the reserved bandwidth with other multiplexed coflows in Alg. 2.

Chronos allocates the multiplexed bandwidth in proportion to the bare bandwidth of each flow such that more urgent flows can get more bandwidth.

$$w_{\text{in}} = \frac{b_{ij}^k}{\sum_{j,c_k\in\mathcal{C}_M} b_{ij}^k},\ w_{\text{out}} = \frac{b_{ij}^k}{\sum_{i,c_k\in\mathcal{C}_M} b_{ij}^k}, \tag{2}$$

This process iterates for each flow (lines 2-6 of Alg. 2). Finally, *Chronos* scans through each port and redistributes the

Algorithm 1 *Chronos*'s main algorithm

procedure SCHEDULER(c_k)
  multiplex $\leftarrow$ TRUE
  for all $f_{ij}^k \in c_k$ **do**
    $\text{cond}_{ij} \leftarrow \sum_j b_{ij}^k >$ IDLE-IN(N_i)
    $\text{cond}_{ji} \leftarrow \sum_i b_{ij}^k >$ IDLE-OUT(N_j)
    if cond_{ij} **or** cond_{ji} **then** $\mathcal{C}_M = \mathcal{C}_M \cup \{c_k\}$
      return
  multiplex $\leftarrow$ FALSE ▷ Meet bandwidth requirements
  for all $f_{ij}^k \in c_k$ **do**
    $b_{ij}^k(t) \leftarrow s_{ij}^k/d_k$ ▷ Allocate bare bandwidth
  return
procedure MAIN
  Denote the set of multiplexed coflows as $\mathcal{C}_M$
  if a coflow arrives or finishes **then**
    Sort all *active* coflows based on their priorities
    for $k = 1 : K$ **do**
      SCHEDULER(c_k) ▷ Allocate bandwidth
    for all multiplexed coflows **do**
      DO-MULTIPLEX(c_k)
    for $i = 1 : n$ **do**
      BACKFILLING(i) ▷ Ensure work-conservation

remaining bandwidth to multiplexed coflows to make them finish faster through a backfilling step (Alg. 3).

We do not restrict *Chronos* to any specific priority schemes, such as the earliest deadline first [7] and the smallest effective bottleneck first [3] strategies. The key principle is that coflows are only temporarily multiplexed and would be reallocated bare bandwidth whenever possible.

Algorithm 2 *Chronos*'s starvation avoidance algorithm

procedure DO-MULTIPLEX(c_k)
  for all $f_{ij}^k \in c_k$ **do**
    $b_{\text{in}}(t) \leftarrow w_{\text{in}} *$ IDLE-IN(R_i)
    $b_{\text{out}}(t) \leftarrow w_{\text{out}} *$ IDLE-OUT(R_j)
    $b_{ij}^k(t) = \min(b_{\text{in}}(t), b_{\text{out}}(t))$
    $\mathcal{C}_M = \mathcal{C}_M - \{c_k\}$

Algorithm 3 *Chronos*'s backfilling algorithm

procedure BACKFILLING(i)
  while IDLE-IN(B_i) > 0 **do**
    for all k, j **do**
      $b = \min($IDLE-IN$(B_i),$ IDLE-OUT$(B_j))$
      $b_{ij}^k(t) = b_{ij}^k(t) + b$
      Update flow rates and idle bandwidth

Since *Chronos* only reallocates bandwidth when coflows arrive or depart as shown in Fig. 3, the scheduling procedure is not frequently invoked. Henceforth, there will not be a significant number of rate control messages exchanging between the central scheduler and end hosts. Furthermore, as observed in production data centers, the number of concurrent coflows running in a network range from tens to hundreds [3]. Therefore, the coflow database does not need to maintain too much information, and the computation time of the schedule algorithms will not become the performance bottleneck.

D. Property Analysis

1) Starvation free: *Chronos* strives to balance between starvation freedom and deadline guarantee by adopting a mixed strategy to combine priority-based scheduling and multiplexing. All the coflows are served as soon as the data to be transmitted are ready. Once scheduled, a coflow is guaranteed to get allocated a certain amount of bandwidth to transfer data continuously instead of being stuck. Unlike the complicated admission control mechanism, *Chronos* does not block or discard any transmission. All coflows can keep on sending data till they finish. In this way, *Chronos* guarantees that the selected coflows can meet their hard deadlines, while the unselected coflows will not be starved perpetually.

2) Work conserving: Even if all the active coflows can finish on time, links should not be left under-utilized. In other words, the bandwidth allocated to a flow should not be bounded to its *bare* bandwidth. Instead, flows should only be bottlenecked at their uplinks or downlinks. By allocating more bandwidth to flows, they can not only meet their deadlines, but also be able to finish as fast as possible. Consequently, the coflow completion times also decrease. *Chronos* achieves work conservation by further embracing a backfilling step to fully utilize available bandwidth. Whenever there is idle bandwidth on a port, *Chronos* will distribute the remaining bandwidth to the flows traversing through this particular port. The bandwidth reallocation procedure performs dynamically and adaptively according to the real-time network status.

3) Graceful degradation: When none deadline information is available, flows within a coflow follow the weighted fair sharing strategy in *Chronos*: all the flows share the available bandwidth proportionally since the deadlines can be viewed as the same. We can further adopt a deadline estimation algorithm to automatically assign deadlines to coflows based on the type of the job a coflow belongs to. For coflows belonging to delay-tolerate applications, we set a default deadline whose value is considerably larger than other concurrent coflows.

To avoid starvation, we embrace a hybrid multiplexing method in Alg. 2, with a knob to adjust the portion of link bandwidth reserved for multiplexed coflows. If such a knob equals to one, *Chronos* would gracefully degrade to a pure multiplexing bandwidth sharing strategy. If the multiplex factor equals to zero, *Chronos* would degrade to a pure priority-based coflow scheduling. It is thus possible that some coflows would get starved. In practice, we can either ensure the multiplex factor is always positive, or further adopt an aging mechanism to dynamically adjust the priority of coflows.

V. Simulation Results

A. Methodology

1) Benchmark workloads: We scale down the practical workloads to reflect coflow patterns observed in production

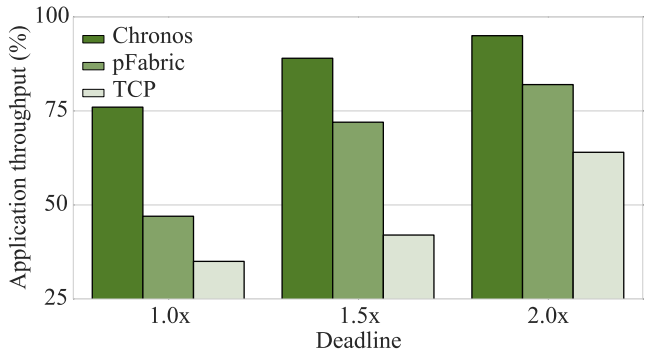


Fig. 4. Percentage of coflows that meet deadlines using *Chronos* in comparison to per-flow fairness and per-flow priority: the improvement factor of *Chronos* over the other two schemes is up to 2.17× and 1.61× respectively.

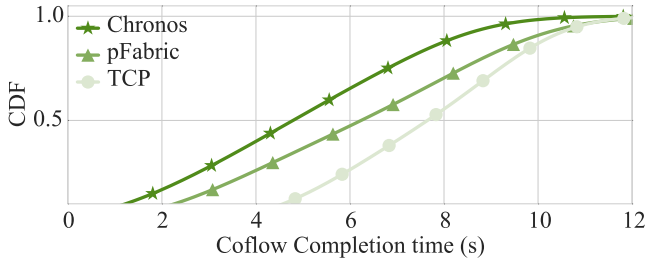


Fig. 5. Coflow completion times using *Chronos* in comparison to per-flow fairness and per-flow priority: *Chronos* reduces the average coflow completion time by 28.9% and 57.7% when compared to pFabric and TCP, respectively.

data centers [3]. The number of coflows ranges from 10 to 100, and the number of flows within each coflow (its width) is between 1 and 50, with 60% of coflow's width smaller than 16. We use the empirical deadline settings as in [3]: a coflow's deadline is set to its minimum completion time in an empty network multiplied by $1+\gamma$ where $\gamma \in (0,1)$ is a uniformly random number. There are 32 end hosts connected to a non-blocking virtual switch through a 100 Mbps access link. Coflows arrive in the network following a Poisson process with rate varying from 0.1 to 0.8 in order to thoroughly evaluate *Chronos*'s performance under different network loads. Flows in a coflow are generated randomly between end hosts with sizes following the long-tail distribution.

2) Schemes compared: The state-of-the-art flow scheduling schemes can be classified into two major categories. One kind of schemes employ the per-flow fairness strategy, such as TCP and its many other variants [10]; another kind of mechanisms adopts per-flow priority [6]. We select one most representative scheme in each of the two categories for thorough evaluations.

TCP: The flows on a congested link equally share the available bandwidth, and achieve the max-min fairness. Switches employ drop-tail queues with a fixed buffer size. Flows gradually increase their sending rate until a back-off is triggered by packet loss or time out. The initial TCP window size in our experiments is set to 12 KB as in [6].

pFabric: By approximating the *shortest remaining time first* strategy, pFabric achieves near-optimal flow-level performance. We implement the ideal version of pFabric, which is free from the inefficiencies caused by packet drops at switches, imperfect load balancing or retransmissions. The performance achieved thus guards above the performance of pFabric.

B. Overall Performance

We first examine *Chronos*'s performance in maximizing the number of coflows that meet deadlines, which is defined as the application throughput [7].

The three groups of results (Fig. 4) indicate the number of coflows meeting the actual deadlines, 1.5× and 2× deadlines. From these results we can see that *Chronos* improves the application throughput by 1.61× and 2.17× compared to pFabric and TCP respectively. With looser performance requirements, pFabric and TCP are able to catch up with *Chronos* gradually. In addition, despite pFabric's ignorance to deadline information, its effectiveness in reducing flow completion time helps itself to outperform TCP significantly. Although our primary objective is not to minimize coflow completion time, we show in Fig. 5 that *Chronos* can effectively *shorten* coflows on average since the network is able to accommodate more coflows simultaneously by allocating the bare bandwidth to coflows. From Fig. 5, we can see that, all the coflows are admitted into the network with no coflow being halted perpetually. In contrast, Varys only admits less than 80% of coflows on average as reported in [3]. This is unacceptable for most deadline-constrained applications, since not only the timeliness, but also the availability of the corresponding service is sacrificed.

C. Impact of Network Load

Since the distributions of coflows sizes and widths are generated based on the workloads collected from production date centers, we keep them unchanged when we measure *Chronos*'s performance under different network loads. The knot to control is the rate of the coflows' arrival process. The corresponding application throughput and coflow completion times of different schemes are shown in Fig. 6(a) and 7(a). As predicted, with more coflows coming into the network, the bandwidth becomes more and more scarce, leading to severer competition among flows in different coflows. Since the size and width distributions of coflows keep constant, the amount of bandwidth needed by each coflow to catch up its deadline is basically constant. When the network bandwidth is used up by active coflows, the arrival of more coflows only leads to the decrease of application throughput. The average coflow completion time would increase accordingly.

D. Impact of Multiplexing

To avoid starvation, we reserve a fixed fraction (*i.e.*, multiplex factor) of bandwidth on each link for multiplexed coflows to share. Through Fig. 6(b), we can see that two flow-level schemes is insensitive to the multiplex factor. However, when more bandwidth is reserved, there is less bandwidth available to guarantee that high-priority coflows can meet

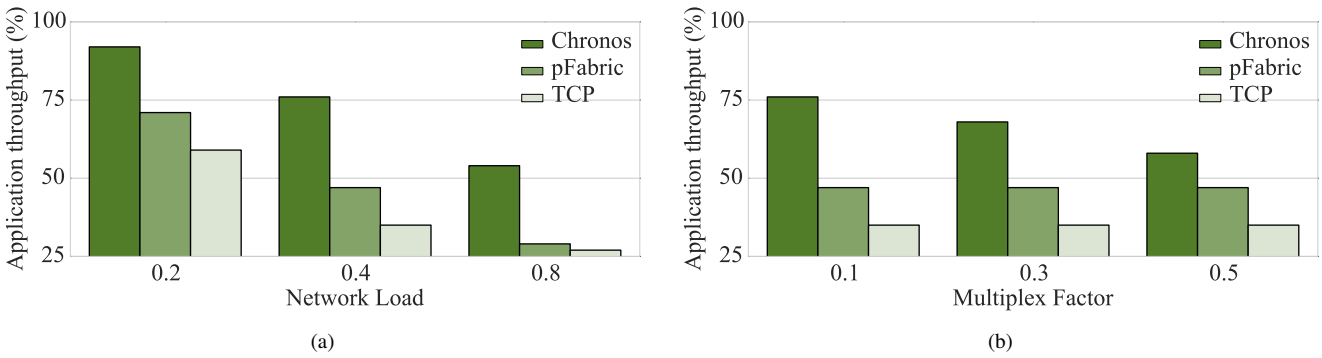


Fig. 6. The application throughput for varying network loads and extents of multiplexing. α equals to 0.1 when examining the impact of network loads; λ equals to 0.4 when analyzing the influence of multiplexing. *Chronos* achieves higher application thoughput under all the settings. The advantages of *Chronos* become more significant with heavier network loads, but the performance gains reduce when too much bandwidth is reserved for multiplexing.

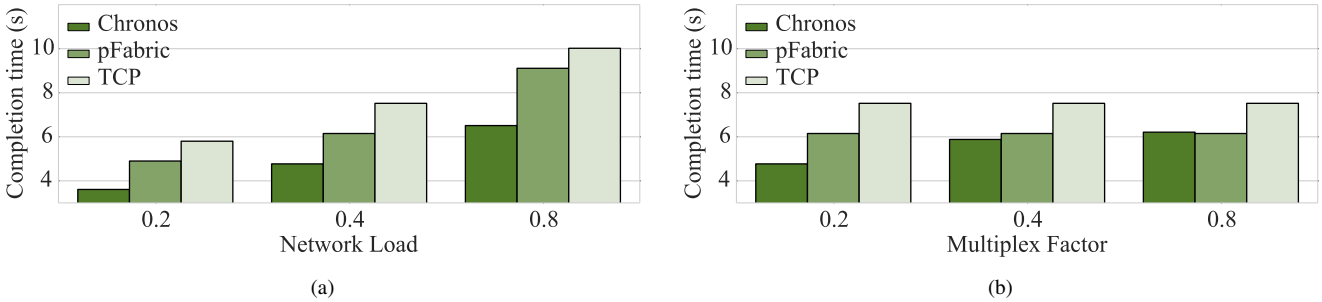


Fig. 7. The average coflow completion times of different schemes under varying network loads and extents of multiplexing: flow-level schemes are not affected by the extent of multiplexing, while *Chronos*'s performance degrades with the increase of multiplex factor.

their deadlines. As a result, *Chronos*'s application throughput decreases with the increase of multiplex factor.

In Fig. 7(b), the average coflow completion time increases with larger factors. However, such growth decreases with larger multiplex factors. With a larger factor, less coflows can finish in time, leading to larger average completion times. Nevertheless, the unselected (multiplexed) coflows can get more amount of traffic, and thus are significantly sped up. An inflection point exists when the decrease of multiplexed coflows' completion times just offsets the increase of non-multiplexed coflows' completion times.

VI. Conclusion

In this paper, we have studied the deadline-driven coflow scheduling problem with the objective of minimizing the number of tardy coflows. Different from meeting each flow's deadline in isolation, whether a coflow can finish on time depends on coordinations among all its constituent flows. To efficiently schedule coflows in an online manner, we have designed a hierarchical scheduling mechanism, *Chronos*, to allocate sufficient bandwidth for coflows to just finish on time. To avoid starvation, we embrace limited multiplexing to make every flow proceed continuously, while not hurting the guaranteed performance of high-priority coflows. The potential bottleneck at *Chronos* is resolved by only rescheduling when coflows arrive or depart. Experiment results acquired from realistic simulations demonstrate that *Chronos* effectively meets coflow deadlines under various network status.

References

[1] R. Kohavi and R. Longbotham, "Online experiments: Lessons learned," *Trans, IEEE Computer*, vol. 40, no. 9, pp. 103–105, 2007.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[3] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with Varys," in *Proc. ACM SIGCOMM*, 2014, pp. 443–454.

[4] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 69–84.

[5] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The power of choice in data-aware cluster scheduling," in *Proc. USENIX OSDI*, 2014, pp. 301–316.

[6] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal near-optimal datacenter transport," in *Proc. ACM SIGCOMM*, 2013, pp. 435–446.

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

[8] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg, "EyeQ: Practical network performance isolation at the edge," in *Proc. USENIX NSDI*, 2013, pp. 297–312.

[9] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proc. ACM SIGCOMM*, 2015, pp. 393–406.

[10] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in *Proc. ACM SIGCOMM*, 2010, pp. 63–74.

[11] Y. Zhao, K. Chen, W. Bai, M. Y. USC, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang, "Rapier: Integrating routing and scheduling for coflow-aware data center networks," in *Proc. IEEE INFOCOM*, 2015, pp. 424–432.

[12] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 63–74, 2008.