Distributed Bottleneck-Aware Coflow Scheduling in Data Centers

Tong Zhang, Ran Shu, Zhiguang Shan, Fengyuan Ren, *Member, IEEE*

Abstract—With the booming development of data parallel frameworks, the coflow abstraction has been greatly favored by data center transport designs, for its prominent ability in capturing application-level semantics. To accelerate job completion, coflow completion time (CCT) is a most important metric, and coflow scheduling is the most effective and widely-adopted means of optimizing CCT. However, most existing coflow scheduling mechanisms neglect the ubiquitous in-network bottlenecks and schedule coflows based on non-blocking giant switch hyperthesis. Such a practice is likely to result in undesired link contention inside the fabric, finally impairing CCT performance. To address this problem, we propose the Distributed Bottleneck-Aware coflow scheduling algorithm called DBA, which approximates the minimum remaining time first (MRTF) heuristic on all fabric-wide links. In this way, core link bandwidths are allocated to coflows as expected and the CCT performance will not be violated. As an evolutionary algorithm, DBA enhances the traditional dual decomposition method thus converges to the optimal bandwidth allocation very fast. Extensive simulations verify DBA's outstanding CCT performance as well as high link utilization. Furthermore, DBA introduces very little overhead and is robust to routing strategies, parameter variations and computation delays.

Index Terms—Coflow Scheduling, Coflow Completion Time, In-network Bottleneck, Data Center

1 Introduction

Nowadays, data parallel frameworks such as MapReduce [1], Dryad [2] and Spark [3] are very popular among data center applications. In such frameworks, jobs go through a series of alternative computation and communication stages to produce the final results. And generally there is a barrier at the end of each communication stage: the succeeding computation stage cannot begin until all its input data are received [4]. Since communication could account for over 50% of job completion time [5], it significantly affects application performance. However, traditional flow-level optimizations (e.g., optimizing flow completion time (FCT) and per-flow fairness) fail to capture application-level requirements, thus are not effective enough in application speedup. Emerging at the right moment, the coflow abstraction [4] bridges the gap between applications and frameworks.

A coflow refers to a set of parallel flows with associated semantics and collective performance goals, e.g., minimizing overall completion time or meeting a common deadline [4]. A coflow completes only when all its inner flows finish transferring. The coflow definition captures all concurrent flows within a communication stage. Therefore, minimizing the coflow completion time (CCT) could actually accelerate the corresponding job. On this issue, coflow scheduling is an effective way and has been widely employed in data center transport designs [5–16].

However, most existing coflow scheduling mechanisms do not take in-network bottlenecks into account. They assume the whole data center fabric is a non-blocking switch and all flows can only be throttled by edge links [7–9, 11, 12, 16, 17]. In fact, today's data center fabrics are far from non-blocking [13, 18, 19], because of oversubscribed topologies, in-network link failures [20], imbalanced routing [19], as well as the NIC speed catching up with the switch processing speed [21]. In this context, a flow's bottleneck often occurs inside the network rather than on edge links. Although there are many efforts in full bisection bandwidth [10, 18, 20, 22–24], they either fail to fully eliminate in-network flow collisions or cannot work at scale. Ignoring in-network bottlenecks can lead to incorrect coflow priority judgment as well as undesired bandwidth contention on core links, greatly hurting the CCT performance.

In this paper, we address the problem of minimizing CCT with considering in-network bottlenecks. In order to explicitly understand coflow scheduling in data centers, we first abstract out the data center fabric into a general undirected graph. Then by this abstraction, we formalize the coflow scheduling problem into an optimization problem.

On this basis, we propose DBA, a distributed bottleneck-aware coflow scheduling algorithm in data centers. DBA leverages the existing transport protocol to learn coflows' remaining completion time, and approximates the well-known minimum remaining time first (MRTF) heuristic. Detailedly, the formalized optimization problem essentially assigns higher priorities to coflows with less remaining time. DBA distributedly solves this problem and allocates the calculated rates to coflows by evolution. On the other hand, the bandwidth allocation is under all link capacity constraints, not only including edge links, but also core

- *T. Zhang, F.Ren are with the Department of Computer Science and Techonolgy, Tsinghua University, Beijing, 100084, China. E-mail: zhang-t14@mails.tsinghua.edu.cn, renfy@tsinghua.edu.cn*
- *R. Shu is with Microsoft Research, Beijing, 100084, China. E-mail: ran.shu@microsoft.com*
- *Z. Shan is with Informatization and Industry Development Department, State Information Center, Beijing, 100045, China.*

links. In this way, even if there exist in-network bottlenecks, the bottleneck link bandwidth can still be properly allocated to contending flows, which reflects "bottleneck-aware". Furthermore, DBA enhances the traditional dual decomposition method so that it converges to the optimal bandwidth allocation very fast. We also conduct theoretical analysis to prove DBA's quadratic convergence rate.

We evaluate DBA using extensive trace-driven simulations under realistic settings. DBA outperforms the giant-switch-based mechanism Varys [7] by 37% in average CCT and 35% in 99th percentile CCT. And compared with Baraat [6] and Fastpass [25] that are not based on giant switch assumption, DBA still has a 35% and 39% decrease in average CCT. Besides CCT performance, DBA also achieves a high throughput. Moreover, DBA converges very fast and introduces very little overhead to bandwidth, CPU and memory. Finally, DBA maintains good performance under a wide range of routing strategies, algorithm parameters, and computation delays.

In brief, our contributions are three-fold:

- Addressing the significant influence that ever-present in-network bottlenecks have on coflow scheduling and CCT performance.
- Designing a distributed coflow scheduling algorithm that takes in-network bottlenecks into account.
- Comprehensively evaluating the proposed algorithm through simulations under practical configurations.

The remainder of the paper is organized as follows. In Section 2, we briefly introduce the coflow abstraction and in-network bottlenecks in data centers. Through a simple example, we illustrate how and how severely a bottleneck-agnostic coflow scheduler affects CCT performance. To address such performance degradation, in Section 3, we model the data center fabric and formalize the bottleneck-aware coflow scheduling problem. Section 4 proposes DBA, and also theoretical analysis on DBA's convergence rate and overheads. In Section 5, extensive simulations verify DBA effectiveness. Finally, the paper is concluded in Section 6.

2 Background and Motivation

In this section, we mainly introduce the coflow abstraction, in-network bottlenecks, as well as the influence of these bottlenecks on coflow scheduling. After that, we illustrate the CCT performance loss without considering in-network bottlenecks using a simple example.

2.1 Background

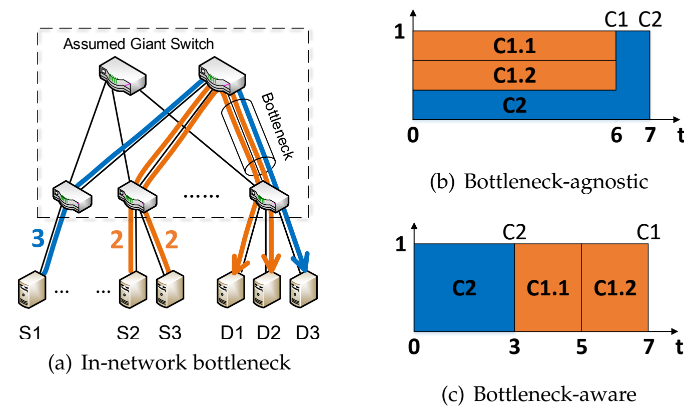
In recent years, data parallel computation frameworks (e.g. MapReduce [1], Dryad [2], Spark [3]) are widely employed by data center applications. Following these frameworks, applications work in the pattern that alternates between computation and communication stages. According to the measurement in [5], communication could account for over 50% of the job completion time. A communication stage usually involves a large number of parallel data transfers, moreover, there is always a barrier (synchronization constraint) at the end of each communication stage. That is, only when all transfers in the preceding communication stage complete can the next computation stage start. As a consequence, it is the final completion time of the entire communication stage that influences application performance. In this context, traditional data center transport designs addressing flow-level metrics (e.g., FCT, deadline meeting rate, per-flow fairness), such as PDQ [26], pFabric [17], PIAS [27], and pHost [28], do not conform to the application-level requirements on the lower framework. To solve this problem, [4] introduces the concept of coflow, devoting to faithfully capture application-level semantics and expose them to the framework.

A coflow is a collection of concurrent flows with associated semantics and a collective performance goal [4], like minimizing the completion time of the latest flow or ensuring all inner flows meet a common deadline. A coflow exactly captures a communication stage in data parallel frameworks. Take MapReduce framework for example, all flows involved in a shuffle phase make up a typical coflow.

By coflow definition, optimizing CCT can truly speed up data center applications. Naturally, minimizing CCT becomes a significant goal of data center transport protocols. Coflow scheduling is a widely-adopted and effective way to optimize CCT. By ordering coflows and regulating how much bandwidth they occupy, data center fabrics can realize various scheduling principles and achieve diverse performance objectives, including minimizing CCT of course. Actually, there have been multiple coflow scheduling mechanisms focusing on CCT optimization [5–16].

However, current data centers make it nontrivial to appropriately schedule coflows, because of in-network bottlenecks. For an individual flow, its bottleneck refers to the link providing the least bandwidth to it among all its passing links, which finally determines its available bandwidth. Obviously, bottlenecks hinge on both link capacities and competing traffic, and a flow could be bottlenecked due to either a narrow link or a congested link, or a combination of both. In today's data centers, the internal networks are far from non-blocking [13, 18, 19], as a result of in-network link failures [20], imbalanced routing [19], as well as NIC speed catching up with switch processing speed in data centers [21]. Congestion is likely to happen on core links and a flow's bottleneck will move to the interior of the network. Worse still, due to the close correlation with real-time traffic and routing, in-network bottlenecks are difficult to predict.

Fig.1(a) briefly depicts a typical data center fabric with in-network bottlenecks. Suppose all links have the same capacity, the two orange (light) flows and the blue (dark) flow separately follow their own routing decisions and collide at the top right downlink that is specially marked. Then the marked link becomes the bottleneck of all three flows and throttles them to use one third of the link capacity each. In this case, the bottleneck lies right inside the network and there will be bandwidth left on the three flows' other passing links. This situation can easily occur. According to [18], the bisection bandwidth loss can even reach 60%! Both measurement in [19] and simulations in [13] confirm that bottlenecks shift from edge links and become more distributed throughout the fabric. In spite of advances in full bisection topologies like Fat-Tree [22], VL2 [23] and PortLand [24], they only mitigate core-link bottlenecks by oversubscription, but cannot handle those by uneven workloads or imbalanced routing. On the other hand, present global routing [10, 18] and load balancing



(a) In-network bottleneck

(b) Bottleneck-agnostic

(c) Bottleneck-aware

Fig. 1: A motivating example

mechanisms [20, 29] either fail to thoroughly balance traffic, or have problem with scalability and have not worked at large scale yet. In other words, in-network bottlenecks will continue to exist at present and are hard to be rooted out in the short term.

In-network bottlenecks decide how much bandwidth flows can actually use, directly affecting CCT. Therefore, in order to achieve ideal CCT performance, in-network bottlenecks (location, available bandwidth, etc.) should certainly be taken into account. However, most state-of-the-art coflow scheduling mechanisms [5, 7–9, 11, 12, 16] adopt the giant switch hypothesis, in which the core links in the dashed box in Fig.1(a) are regarded as non-blocking. In this way, they neglect in-network bottlenecks and only consider edge links. Such practice is bound to cause unexpected scheduling results and hurting CCT performance. The performance loss will be demonstrated in the next subsection.

2.2 Motivation

Concretely, we illustrate how and to what extent ignoring in-network bottlenecks harms CCT, using a simple example. Reconsider the straightforward scenario in Fig.1(a) from the view of coflow. There are two coflows denoted with different colors (intensities): coflow1 ($C1$) with orange (light), and coflow2 ($C2$) with blue (dark). $C1$ consists of 2 flows ($C1.1$ and $C1.2$), while $C2$ only has one. The arrows indicate flow directions and the numbers express flow sizes. In today's data center fabric, the in-use ECMP routing policy [30] can make such traffic very common. All links have the same capacity 1 (one data unit per time unit). We can see the three flows have different sources and destinations. Suppose all coflows arrive at time 0, an intuitive idea is to prioritize them in ascending order of their remaining completion time when they each exclusively occupy all passing links, which is the well-known MRTF heuristic and has been demonstrated to be CCT-effective [7, 10, 11]. Adopting MRTF, Fig.1(b) and Fig.1(c) respectively show the scheduling results with and without considering in-network bottlenecks.

Fig.1(b) depicts the bottleneck bandwidth allocation when the scheduler does not consider in-network bottlenecks. In this case, the scheduler only sees bandwidth contention on edge links. Because three flows have no contention at any edge link, the scheduler believes all three flows can take up full link bandwidth. Then in the scheduler's view, $C1$ can complete within 2 time units, while $C2$ needs 3 time units to finish. Therefore $C1$ is assigned a higher priority than $C2$. After scheduling $C1.1$ and $C1.2$ both with full link bandwidth, the scheduler will then schedule $C2$ since it supposes $C2$ can also transfer with full link bandwidth. In practice, due to the in-network bottleneck and congestion control, the bottleneck bandwidth will be equally divided to three flows. Although the scheduler devotes to realize a priority algorithm, the obtained result degenerates to fair sharing. $C1$ and $C2$ respectively finish at time 6 and 7, and the average CCT is 6.5. However, the situation is quite different if the scheduler is omniscient and know the in-network bottleneck before scheduling, which is shown in Fig.1(c). The scheduler knows $C1$ needs 4 time units and $C2$ only needs 3. Then $C2$ is first scheduled with full link bandwidth, then the scheduler notices that the bottleneck link has been monopolized by $C2$ and can provide no bandwidth to $C1$ any more. Therefore, $C1$ is not scheduled for the moment. After $C2$ completes, $C1.1$ and $C1.2$ will subsequently inherit the bottleneck link bandwidth to finish transfer. As a result, two coflows respectively finish at time 3 and 7, and the average CCT is 5, decreasing by 23%.

The above example reveals that ignoring in-network bottlenecks not only causes coflow priority misjudgment, but also leads to improper flow injection into the fabric network, both hurting the CCT performance. In reality, the number of concurrent coflows and each coflow's inner flows are very large, thus this impairment can be more serious.

However, it is impractical to know in-network bottleneck information before flows really enter the network, because a flow's path is only determined upon its packet forwarding. But from another perspective, once a flow enters the network, each of its passing nodes will learn its next hop. If all these information is aggregated, the entire flow path can be restored and bottleneck information can be obtained. On this issue, most coflow scheduling mechanisms [5, 7–10, 12, 14, 16, 31, 32] are centralized, where a central controller collects all coflow information and makes scheduling decisions for all flows. Nevertheless, it is an extra huge delay overhead for a centralized scheduler to gather all path information and to assign scheduling decisions. In addition, a centralized algorithm has problem with scalability thus cannot support large-scale scenarios in real time. Comparatively, a distributed mechanism operating on separate nodes not only can exchange coflow information (including flow bottlenecks) by embedding it in data flows, but also has great superiority in scalability. However, the only distributed coflow scheduling mechanisms [6, 13] respectively employ first-in-first-out (FIFO) and smallest-coflow-first (TCF) principles, which do not perform well enough in minimizing CCT [7]. In this work, our goal is a distributed CCT-oriented coflow scheduling algorithm with awareness of in-network bottlenecks.

3 Model and Formalization

In this section, we develop a graph model to abstract out the data center fabric. On this basis, we describe our target coflow scheduling problem into an optimization problem over the graph, which provides the foundation of the algorithm design in the next section.

3.1 Fabric Model

Consider a data center consisting of N hosts, M switches and L links, with an arbitrary topology. Then this fabric can be described by an undirected graph $G(V, E)$, where $|V| = N + M$ and $|E| = L$. Each vertex corresponds to a fabric node (host or switch), and each edge represents a full duplex link. The N vertices denoting hosts and the other M indicating switches are separately indexed with $\{1, 2, \cdots, N\}$ and $\{1, 2, \cdots, M\}$. After indexing the edges with $\{1, 2, \cdots, L\}$, each edge has a capacity B_L equal to the corresponding link's bandwidth capacity (the uplink and downlink have equal capacities in general).

A coflow is a collection of concurrent flows. For example, $C_k = \{f_{k1}, f_{k2}, \cdots, f_{k_q}\}$ means that the k^{th} coflow C_k totally consists of q flows $f_{k1}, f_{k2}, \cdots, f_{kq}$. Specifically, an independent flow not sharing semantics or performance goals with any other flows is also regarded as a single-flow coflow. Let $\mathbb{F}$ be the set of all current flows and $\mathbb{C}$ the set of all current coflows. Apparently, each flow must belong to and can only belong to one coflow, that is, $\mathbb{C}$ forms a partition of $\mathbb{F}$. We assume that a coflow's all inner flows arrive simultaneously, and the fabric does not know a coflow's arrival time until it really comes [8]. It should be stressed that this assumption does not really make demand on coflow arrival, but only specifies the way to view it. Even if inner flows arrive asynchronously, we could think of them as arriving together and deem the arrival time of the last flow as the coflow's arrival time. Additionally, we assume that an inner flow's information (source, destination, size, belonging coflow) is known by its source host upon arrival. This assumption has been adopted extensively [5–7, 9–11, 16] and is reasonable in that flow information can be acquired from upper-layer applications [4].

3.2 Problem Formalization

Remember that our design target is a distributed bottleneck-aware CCT-optimizing coflow scheduling algorithm. Focusing on the two requirements – bottleneck-aware and CCT-optimizing, we formalize the coflow scheduling problem into the following optimization problem at every instant t:

$$\text{Maximize:} \quad \sum_{i=1}^{n(t)} \frac{b_i(t)}{T_{k(i)}(t)} \tag{1}$$

$$\text{Subject to:} \quad \sum_{i=1}^{n(t)} b_i(t) x_{il} \leq B_l \quad \forall l \tag{1a}$$

$$b_i(t) \geq 0 \quad \forall i \tag{1b}$$

At any time t, there are totally $n(t)$ active flows and they separately belong to different coflows. $b_i(t)$ $(i = 1, 2, \cdots, n)$ is the bandwidth assigned to flow i at time t, which is also the optimization variable. $k(i)$ means the index of the coflow that contains flow i, and $T_{k(i)}(t)$ denotes the remaining completion time of coflow $k(i)$ at time t. In constraint (1a), x_{il} is an indicator of whether flow i passes link l and B_l is link l's capacity as defined above. Constraint (1a) expresses that at any instant t the aggregated flow bandwidth cannot exceed the link capacity. And on account of $b_i(t)$'s physical meaning, constraint (1b) requires that all $b_i(t)$ are nonnegative. Theoretically, the scheduler solves this optimization problem at every moment (far from necessary in practice), that is to say, we schedule coflows by allocating bandwidth to their individual flows at every instant.

On the one hand, to minimize CCT, the objective function approximates the well-known MRTF heuristic. $\frac{b_i(t)}{T_{k(i)}(t)}$ assigns a weight $\frac{1}{T_{k(i)}(t)}$ to each flow rate $b_i(t)$, and then the weighted sum is to be maximized. Note that the weight is inversely proportional to the remaining completion time of the flow's parent coflow. In this way, the faster a coflow can complete (less remaining completion time), the larger weights its inner flows will get. And to maximize the weighted sum, inner flows of a faster coflow will get more bandwidth, coinciding with MRTF. Letting faster coflows complete first could reduce the waiting time of others, then decreases average CCT. Besides, because of the maximization, flows will try their best to occupy the link bandwidth, in which case the work-conservation property is achieved.

On the other hand, considering in-network bottlenecks is reflected in the constraint (1a). The weighted sum maximization is under all links' capacity constraints, not only edge links, but also in-network core links. With awareness of in-network capacity constraints, if a flow's bottleneck really lies inside the network, the scheduler can identify it. In this way, the scheduler could precisely predict each coflow's remaining completion time and make in-network bandwidths truly occupied by those high-priority flows. From this point of view, those scheduling mechanisms based on the giant switch hypothesis can also be regarded as optimizing some goals only under edge link capacity constraints. Therefore, there could be unexpected in-network bandwidth oversubscription that impacts scheduling accuracy.

If treating $\frac{b_i(t)}{T_{k(i)}(t)}$ as the utility achieved by flow i, then problem (1) is a standard Network Utility Maximization (NUM) problem, and is also a linear programming problem. Nevertheless, problem (1) is just a formal representation, and cannot be directly solved. That is because before flows are really routed, all relevant x_{il} and their parent coflows' $T_{k(i)}(t)$ remain unknown. And as the coflow proceeds, its remaining completion time T_k will decrease, which we should also keep track of. To develop a practical coflow scheduling algorithm, we leverage the data center network protocol to probe x_{il} and $T_{k(i)}(t)$, and to deliver such information among hosts in real time. Then hosts distributedly make decisions about flow rates according to obtained information. In the next section, we will detailedly propose our coflow scheduling algorithm that skillfully solves problem (1).

4 Distributed Bottleneck-aware Coflow Scheduling Algorithm

Based on the fabric model and problem formalization in Section 3, we propose DBA, a **D**istributed **B**ottleneck-**A**ware coflow scheduling algorithm. Next we respectively present the algorithm principle, design, and analysis.

4.1 Algorithm Principle

Essentially, DBA practically solves problem (1). On the whole, "practically" reflects in two aspects. First, DBA scheduling decisions (flow rates) are made by separate nodes locally. Second, DBA directly leverages data flows to carry and deliver information (coflow information, scheduling decisions, algorithm parameters, etc.). Through a delicate design, DBA can indeed find the near-optimal solution to problem (1) with little overheads.

To start with, we conduct a slight modification to problem (1), that is adding an index $(1-\epsilon)$ to $b_i(t)$ in the objective function, as shown in Equation 2. Note that ϵ should take a small positive value so that the changed objective function is strictly concave. Correspondingly, all link capacities are normalized to the range $[0, 1]$. The concrete approach is normalizing the network-wide largest link capacity to 1, and then handle all other link capacities with the same normalization factor. Consequently, optimization variables $b_i(t)$ $(i \in \{1, \cdots, n\}, \forall t)$ are also limited to $[0, 1]$, where the difference between $b_i(t)$ and $b_i(t)^{1-\epsilon}$ is particularly small. We define B'_l to be the normalized B_l and β to be the normalization factor, i.e., $\frac{B_l}{B'_l} = \beta$ $(l = 1, \cdots, L)$. In this way, the modification greatly helps the design, but has very little impact on the scheduling result. The modified problem is listed in the following problem (2) .

$$\text{Maximize:} \quad \sum_{i=1}^{n(t)} \frac{b_i(t)^{1-\epsilon}}{T_{k(i)}(t)} \tag{2}$$

$$\text{Subject to:} \quad \sum_{i=1}^{n(t)} b_i(t) x_{il} \le B'_l \quad \forall l \tag{2a}$$

$$b_i(t) \ge 0 \quad \forall i \tag{2b}$$

After modification, problem (2) is a strictly convex optimization problem, thus those gradient-based methods can be used to solve it. DBA's basic idea exactly comes from an enhanced dual decomposition method by us.

It can be observed that the partial "utility" $\frac{b_i(t)^{1-\epsilon}}{T_{k(i)}(t)}$ is only associated with flow i, thus problem (2) satisfies the decomposition structure. Then the dual decomposition method [33], which corresponds to our distributed design intention, applies to our problem. A traditional dual decomposition method runs as follows. Firstly, obtain the Lagrangian:

$$\text{Maximize:} \quad \sum_{i=1}^{n(t)} \frac{b_i(t)^{1-\epsilon}}{T_{k(i)}(t)} - \sum_{l=1}^{L} \lambda_l(t) \left(\sum_{i=1}^{n} b_i(t) x_{il} - B'_l \right) \tag{3}$$

$$\text{Subject to:} \quad b_i(t) \ge 0 \quad \forall i \tag{3a}$$

$$\lambda_l(t) \ge 0 \quad \forall l \tag{3b}$$

where $\lambda_l(t)$ $(l = 1, 2, \cdots, L)$ is the Lagrange multiplier, and also serves as the price of link l in the physical sense, indicating the link congestion degree.

Then divide the Lagrangian into $n(t)$ subproblems and a master problem. Each subproblem corresponds to a $b_i(t)$, while the master problem manages all $\lambda_l(t)$. The i^{th} subproblem takes the following form:

$$\text{Maximize:} \quad \frac{b_i(t)^{1-\epsilon}}{T_{k(i)}(t)} - \sum_{l \in L(i)} \lambda_l(t) b_i(t) \tag{4}$$

$$\text{Subject to:} \quad b_i(t) \ge 0 \quad \forall i \tag{4a}$$

where $L(i)$ represents the set of links passed by flow i. Here $b_i(t)$ is the only variable, and all $\lambda_l(t)$ are parameters. Obviously, each subproblem can be locally solved by flow i itself. As the subproblem's objective function is strictly concave, the optimal solution is unique. Given non-negativity of all $\lambda_l(t)$, the optimal solution can be directly obtained by derivation. By definition, the optimal solution is a function of all $\lambda_l(t)$ and thus denoted as $b_i^*(\boldsymbol{\lambda}(t))$ with $\boldsymbol{\lambda}(t)$ being the vector of $\lambda_l(t)$. Then the master problem can be written as:

$$\text{Minimize:} \quad \sum_{l=1}^{L} \lambda_l(t) \left(B'_l - \sum_{i \in F(l)} b_i^*(\boldsymbol{\lambda}(t)) \right) + \sum_{i=1}^{n(t)} \frac{b_i^*(\boldsymbol{\lambda}(t))^{1-\epsilon}}{T_{k(i)}(t)} \tag{5}$$

$$\text{Subject to:} \quad \lambda_l(t) \ge 0 \quad \forall l \tag{5a}$$

where $F(l)$ is the set of flows passing link l. Note that all $b_i(t)$ in the master problem are replaced by $b_i^*(\boldsymbol{\lambda}(t))$, the optimal solutions of subproblems. Therefore, only $\lambda_l(t)$ $(l = 1, 2, \cdots, L)$ are optimization variables. When all $b_i^*(\boldsymbol{\lambda})$ are known, the master problem can also be settled locally by each link due to the objective's decomposition structure.

Finally, the master problem is solved with the gradient descent algorithm. Simply, link l iteratively updates its price λ_l following the below formula.

$$\lambda_l(t+1) = \left[\lambda_l(t) - \alpha \left(B'_l - \sum_{i \in F(l)} b_i^*(\boldsymbol{\lambda}(t)) \right) \right]_+ \tag{6}$$

where t serves as the iteration index and α is a positive step size. $b_i^*(\boldsymbol{\lambda}(t))$ is the optimal solution of the i^{th} subproblem at iteration t, and $[\cdot]_+$ is to guarantee the non-negativity of λ_l. After iteration $t+1$, all b_i^* are recomputed with the newly obtained $\boldsymbol{\lambda}(t+1)$. The newly acquired $b_i^*(\boldsymbol{\lambda}(t+1))$ is then applied to the calculation of $\boldsymbol{\lambda}(t+2)$. Through this iterative process, $\boldsymbol{\lambda}$ will converge to the dual optimal $\boldsymbol{\lambda}^*$, and $b_i^*(\boldsymbol{\lambda}^*)$ is right the optimal solution to problem (2).

However, the gradient descent algorithm owns a serious drawback, that is slow convergence. In data centers, a massive number of flows are continuing to arrive (complete). If the workload changes before the algorithm convergence, the algorithm is always trying to catch up with the new optimal bandwidth allocation. In other words, the algorithm will never converge and flows will never get their stable rates. In hence, DBA should hold the property of fast convergence.

In fact, for problem (2), KKT condition in Equation (7) is already the sufficient condition of an optimal solution, and the gradient descent algorithm just specifies a locus to it.

$$\begin{cases} (1-\epsilon) \dfrac{b_i(t)^{-\epsilon}}{T_{k(i)}(t)} - \displaystyle\sum_{l \in L(i)} \lambda_l(t) = 0 \quad \forall i & (a) \\ \lambda_l(t) \left(\displaystyle\sum_{i \in F(l)} b_i(t) - B'_l \right) = 0 \quad \forall l & (b) \end{cases} \tag{7}$$

However, to reach the optimal, it is not necessary to completely follow this locus. Therefore, DBA improves the convergence rate by adopting another locus. Concretely, DBA introduces weighted fair queuing (WFQ) [34] and

Nesterov accelerated gradient descent method (AGD) [35] to optimal solution searching. We will detailedly describe the working process of DBA in the next subsection.

4.2 Algorithm Design

As a whole, DBA contains two parts: one for acquiring remaining coflow completion time, the other for driving flow rates to the optimal by means of evolution. DBA design is completely distributed and leverages the underlying transport protocols in data centers.

4.2.1 Coflow Information Coordination

In obtaining coflows' initial remaining completion time, DBA leverages the three-way handshake mechanism at connection establishment to communicate among senders, receivers, and intermediate nodes. And in the data transferring phase, DBA relies on senders to maintain and update coflows' real-time remaining completion time.

Structurally, each intermediate node maintains a table recording each passing coflow's local transmission time T_{kl} and capacity-normalized local flow number N_{kl}, as depicted in Table 1. Here each node only maintains the values for its outgoing links. The "local transmission time" means the transmission time of a coflow's part passing a specific link, calculated as the sum of sizes of one coflow's inner flows through a certain link divided by this link's capacity ($T_{kl} = \frac{\sum_{i \in C_k \bigcap F(l)} S_i}{B_l}$). And the "local flow number" records the number of a coflow's inner flows passing a specific link normalized by this link's capacity ($N_{kl} = \frac{|C_k \bigcap F(l)|}{B_l}$), which is for remaining completion time update during data transferring phase. For each coflow, from all its T_{kl} in all passing nodes, the maximum one is its real remaining completion time ($T_k = \max_l T_{kl}$). We will introduce how DBA calculates and communicates T_k for all coflows next.

TABLE 1: Local remaining completion time

Outlink / Coflow	outlink 1		outlink 2		...
coflow id1	T_{id11}	N_{id11}	T_{id12}	N_{id12}	
coflow id2	T_{id21}	N_{id21}	T_{id22}	N_{id22}	
...					

At every flow arrival, DBA allows the source host to establish the connection, which is through the three-way handshake mechanism. The source host encapsules four messages in the SYN packet header: flow size S_i, parent coflow id k, parent coflow completion time $T_k^{(i)}$, and capacity-normalized bottleneck flow number $N_k^{(i)}$. The flow size S_i as well as parent coflow id k are assigned with actual flow information. The completion time $T_k^{(i)}$ is directly initialized to the source host's T_{kl} – coflow k's total transmission time on flow i's sender-side access link l. Similarly, $N_k^{(i)}$ is initialized to the sender's N_{kl}. When the SYN packet passes through each intermediate node on its path, this node will add this flow's transmission time to the corresponding T_{kl} item ($T_{kl} \leftarrow T_{kl} + \frac{S_i}{B_l}$), and add the paired N_{kl} item by $\frac{1}{B_l}$ ($N_{kl} \leftarrow N_{kl} + \frac{1}{B_l}$). If the table item for coflow k does not exist yet, the node first create this item then let $T_{kl} \leftarrow \frac{S_i}{B_l}$, and $N_{kl} \leftarrow \frac{1}{B_l}$. If there are multiple SYN packets arriving in a burst, the node will conduct the above operations for all of them in batch. Finally, the updated T_{kl} is compared with $T_k^{(i)}$ in the SYN packet header. If $T_k^{(i)} < T_{kl}$, then $T_k^{(i)} \leftarrow T_{kl}$ and $N_k^{(i)} \leftarrow N_{kl}$. Therefore, the SYN packet will finally carry the maximum local transmission time along its path and the paired capacity-normalized flow number to the receiver.

Every end host holds another table (Table 2) to maintain each correlated coflow's remaining completion time, which are collected from packet headers. Here T_k^{in} and N_k^{in} are for sender, while T_k^{out} and N_k^{out} are for receiver. Because a host's outgoing coflow set is very likely to be inconsistent with its ingoing one, there could be null items in the table.

Firstly, receivers are responsible for extracting a coflow's real T_k from all $T_k^{(i)}$ values. That is because in data centers, the many-to-one traffic pattern is very common [13], and within each coflow the number of senders is always larger than (even dozens of times of) that of receivers [7]. Because of smaller quantity, each receiver could get more local information than senders, and have a greater opportunity to acquire the largest T_k. T_k^{out} is right the maximum $T_k^{(i)}$ value for each coflow k received by the receiver itself, and N_k^{out} is the corresponding capacity-normalized flow number.

TABLE 2: Remaining completion time at end hosts

Coflow	completion time		#flow/capacity	
coflow id1	T_{id1}^{in}	T_{id1}^{out}	N_{id1}^{in}	N_{id1}^{out}
coflow id2	T_{id2}^{in}	T_{id2}^{out}	N_{id2}^{in}	N_{id2}^{out}
...				

On receiving every SYN packet, the receiver will take out $T_k^{(i)}$ and $N_k^{(i)}$ from the header and compare $T_k^{(i)}$ with T_k^{out} in the table. If $T_k^{out} < T_k^{(i)}$, let $T_k^{out} \leftarrow T_k^{(i)}$, and $N_k^{out} \leftarrow N_k^{(i)}$. And if there is no table entry for this coflow k, a new entry will be created first, then $T_k^{out} \leftarrow T_k^{(i)}$ and $N_k^{out} \leftarrow N_k^{(i)}$. If multiple SYN packets come in a burst, the receiver performs above operations for all of them in batch. In this way, after a coflow's all SYN packets have been received, its receivers will all get their maximum possible T_k^{out} values, from which at least one is the coflow's real T_k. Finally, in response to each SYN packet, the receiver piggybacks its corresponding T_k^{out} and N_k^{out} in the responding SYN-ACK packet header and sends it back to the sender host.

The action of sender hosts for SYN-ACK packet handling is very similar to that of receivers. In Table 2, T_k^{in} records the maximum T_k^{out} value collected from SYN-ACK packet headers, and N_k^{out} the paired capacity-normalized flow number. On receiving an SYN-ACK, the sender host first takes out T_k^{out} and N_k^{out} from the header, then compares T_k^{out} with the corresponding T_k^{in} in the table. If $T_k^{in} < T_k^{out}$, then $T_k^{in} \leftarrow T_k^{out}$ and $N_k^{in} \leftarrow N_k^{out}$. This behavior further refines T_k^{in} from T_k^{out}, enlarging the opportunity for senders to obtain T_k. Finally, the sender sends the last ACK in three-way handshake to the receiver.

We can see that, although not all senders could get accurate T_k, most of them could. Firstly, all bottleneck flows will definitely carry T_k. Then they spread this value to their respective receivers. These receivers collect all received $T_k^{(i)}$ and extract the maximum one from them, thus will certainly get T_k. Finally, these receivers further spread T_k

to all their flow-relevant senders. Since each sender will again extract the maximum from all received T_k^{out}, these senders will also get T_k. That is, as long as a flow shares the same sender or receiver with any bottleneck flow, its sender will learn T_k. In the extensive many-to-one or one-to-many patterns, the coflow bottleneck must lie in an access link, and all senders and receivers will explicitly get precise T_k. While in the many-to-many pattern (e.g., MapReduce shuffle), the inner flows often have relatively uniform spatial distributions among senders and receivers. Therefore if the coflow bottleneck lies in an access link, other relevant access links will also have close values, thus for most senders $T_k^{in} \approx T_k$. And if the bottleneck rests on a core link, the bottleneck flows must have widespread senders and receivers (or else the bottleneck should be on the access link). Thus a large proportion of senders and receivers could still get T_k. Given the above, senders directly use their T_k^{in} and N_k^{in} to approximate T_k and N_k, respectively.

However, above calculated is just initial remaining completion time. As inner flows proceed, a coflow's remaining completion time will decrease in real time. Therefore, besides connection establishment, during flow data transferring senders also keep updating their T_k^{in}. In the feedback mechanism, an ACK means the corresponding data packet(s) have been received, thus senders update T_k^{in} in terms of ACKs. Each time receiving a window of ACKs, the sender host updates the corresponding T_k^{in} as follows:

$$T_k^{in} \leftarrow \max\left\{T_k^{in} - \frac{\widetilde{S}_i N_k^{in}}{|C_k \bigcap F(l)|}, T_{kl}\right\} \qquad (8)$$

where $\widetilde{S}_i$ represents the total ACKed data size by the received ACK packets, and l denotes the sender's outgoing access link. Then we can see the above maintained N_k^{in} exactly plays a part here. N_k^{in} aims to approximate N_k, the capacity-normalized flow number on coflow k' bottleneck. $N_k = \frac{|C_k \bigcap F(l')|}{B_{l'}}$, where l' denotes coflow k's bottleneck link. The explanation is that each sender estimates the transferred data volume of bottleneck flows based on its own transferred volume. More specifically, each sender of coflow k believes all other senders hold the same number of inner flows as well as transfer the same data volume as itself. By replacing N_k^{in} with $\frac{|C_k \bigcap F(l')|}{B_{l'}}$ in Equation (8), $\frac{|C_k \bigcap F(l')|}{|C_k \bigcap F(l)|}$ estimates the number of senders of all bottleneck flows, then $\frac{\widetilde{S}_i |C_k \bigcap F(l')|}{|C_k \bigcap F(l)|}$ calculates the total data size transferred by these senders in current round of ACKs, and finally $\frac{\widetilde{S}_i |C_k \bigcap F(l')|}{B_l' |C_k \bigcap F(l)|}$ the corresponding reduced transmission time. By subtracting $\frac{\widetilde{S}_i N_k^{in}}{|C_k \bigcap F(l)|}$ every window, T_k^{in} is updated at sender. The $max\{\cdot\}$ operator is because a coflow's total remaining time must be no less than the remaining transmission time on any access link. Although the above deduction is not completely precise, it keeps a correct priority order among coflows at each sender locally, which is enough for DBA bandwidth allocation. More importantly, it substantially decreases the communication overheads in T_k access.

We can see all operations are driven by event (flow arrival, SYN arrival, and SYN-ACK arrival). Detailed operations of sender, receiver, and intermediate node are described in Algorithm 1, 2, and 3 respectively.

Algorithm 1 Sender Operation in T_k Access

```
 1: if event == flowArrival then
 2:     $SYN.k \leftarrow flow.parentCoflowId$
 3:     $SYN.S_i \leftarrow flow.size$
 4:     $T_k^{(i)} \leftarrow \frac{\sum_{j \in C_k \bigcap F(l)} S_j}{B_l}$, $N_k^{(i)} \leftarrow \frac{|C_k \bigcap F(l)|}{B_l}$
 5:     Send SYN packet
 6: end if
 7: if event == SYN-ACKArrival then
 8:     if $SYNACK.T_k^{out} > T_k^{in}$ then
 9:         $T_k^{in} \leftarrow SYNACK.T_k^{out}$
10:         $N_k^{in} \leftarrow SYNACK.T_k^{out}$
11:     end if
12:     Send ACK packet
13: end if
14: if event == ACKArrival then
15:     $T_k^{in} \leftarrow \max\{T_k^{in} - \frac{ACK.ACKedSize \times N_k^{in}}{|C_k \bigcap F(l)|}, T_{kl}\}$
16:     Send next data packet
17: end if
```

Algorithm 2 Receiver Operation in T_k Access

```
1: if event == SYNArrival then
2:     if $T_k^{out} < T_k^{(i)}$ then
3:         $T_k^{out} \leftarrow T_k^{(i)}$, $N_k^{out} \leftarrow N_k^{(i)}$
4:     end if
5:     $SYNACK.T_k^{out} \leftarrow T_k^{out}$, $SYNACK.N_k^{out} \leftarrow N_k^{out}$
6:     Send SYN-ACK packet
7: end if
```

4.2.2 Bandwidth Allocation

In setting flow rates, DBA adapts to existing window-based rate control protocols and converts the allocated flow rate into window size. Each flow i has a weight ω_i and a rate βb_i, where ω_i is a network-wide relative value and b_i is the real flow rate. Each link maintains a price λ_l and an auxiliary variable ν_l. And there is a global parameter δ. In the initial state ($t = 0$), all link prices $\lambda_l(0)$ as well as $\nu_l(0)$ are set to 0, and the global parameter $\delta(0) = 0$. Here DBA adds another message: path length (hop count) L_i to the SYN packet header, and three messages: price sum along the path P_i, flow weight ω_i, and per-link deviation between flow weight and rate e_i to the normal data packet header, from which P_i and e_i are initialized to 0, while ω_i is initialized to a large value (e.g., 10^{10}). We will explain the intension to do so next.

When the SYN packet passes through each intermediate node or receiver, L_i in the header will increase by one. In this way, when this SYN packet arrives at the receiver, L_i will record the total hop count. Then the receiver host encapsules this L_i value in the corresponding SYN-ACK packet header, so that the sender will then get the path length.

After connection establishment, DBA firstly sets a fixed initial window size W_0 (e.g., 3 packets) for all active flows, i.e., every flow could send W_0 bytes at the beginning. Besides, the initial flow weight ω_i is orders of magnitude larger than normal flow weights later, which helps flows get large rate at the beginning. The initial window has two purposes: bringing back the link price sum to sender, and allowing short-lived flows to finish all at once to reduce completion time for tiny coflows. When the first packet in the initial

Algorithm 3 Intermediate Node Operation in T_k Access

```
1: if event == SYNArrival then
2:     $k \leftarrow SYN.k$, $l \leftarrow SYN.outLink$
3:     $T_{kl} \leftarrow T_{kl} + \frac{SYN.S_i}{B_l}$, $N_{kl} \leftarrow N_{kl} + \frac{1}{B_l}$
4:     if $T_k^{(i)} < T_{kl}$ then
5:         $T_k^{(i)} \leftarrow T_{kl}$, $N_k^{(i)} \leftarrow N_{kl}$
6:     end if
7:     Forward SYN packet
8: end if
```

window goes through the fabric, it will add up the link price λ_l along the path and record the sum in the P_i field. And each intermediate node adopts WFQ to distribute outgoing link bandwidth to flows according to their initial weights. That is, each link allocates its bandwidth among passing flows in proportion to their initial weights. During packet arrivals to receiver, the receiver host could perceive the instant flow rate βb_i (since b_i represents the normalized rate) by inter-packet interval I_i. The responding ACK packet(s) will bring back both P_i and I_i values to the sender host.

Then DBA works in the iteration form, and each iteration corresponds to an RTT (a window of packets). In iteration t, the sender firstly calculates the window size $W(t)$:

$$W(t) = \beta \bar{b}_i(t-1) \times RTT_0 \tag{9}$$

where $\bar{b}_i(t-1)$ is the average normalized flow rate from all I_i values brought by ACK packets in the last window, and RTT_0 the baseline RTT without queuing delay. The calculated $W(t)$ is used as the next congestion window size. Note that in the next round of initial window, we still let $W(t) = W_0$ because flow rate now is by the large initial weight and not its deserved rate. Then the sender updates the flow weight ω_i with the current parent coflow remaining time $T_{k(i)}(t)$ and the latest link price sum $\sum_{l \in L(i)} \lambda_l(t-1)$:

$$\omega_i(t) = \left(\frac{T_{k(i)}(t) \sum_{l \in L(i)} \lambda_l(t-1)}{1-\epsilon} \right)^{-\frac{1}{\epsilon}} \tag{10}$$

and computes the per-link deviation in Equation 7(a):

$$e_i(t) = \frac{1}{|L(i)|} \left((1-\epsilon) \frac{\bar{b}_i(t-1)^{-\epsilon}}{T_{k(i)}(t)} - \sum_{l \in L(i)} \lambda_l(t-1) \right) \tag{11}$$

When sending the window of packets in iteration t, the sender stamps $\omega_i(t)$ in the header of every packet, and $e_i(t)$ in the header of the last packet in the window. And then, each intermediate node still adopts WFQ to distribute the outgoing link bandwidth to flows according to their $\omega_i(t)$. Of course a flow i's final rate is equal to the minimum bandwidth allocated to it among all its passing links. The receiver continues to perceive flow rates and writes $I_i(t)$ in every ACK. Also, the first packet in the window adds up link price $\lambda_l(t)$ along its path in P_i, and the first ACK feedbacks P_i value to the sender.

After each link l receives all associated $e_i(t)$ $(i \in F(l))$ in iteration t, the link updates its price λ_l according to:

$$\nu_l(t+1) = \lambda_l(t) + \min_{i \in F(l)} e_i(t) - \gamma(B'_l - \sum_{i \in F(l)} b_i(t)) \tag{12}$$

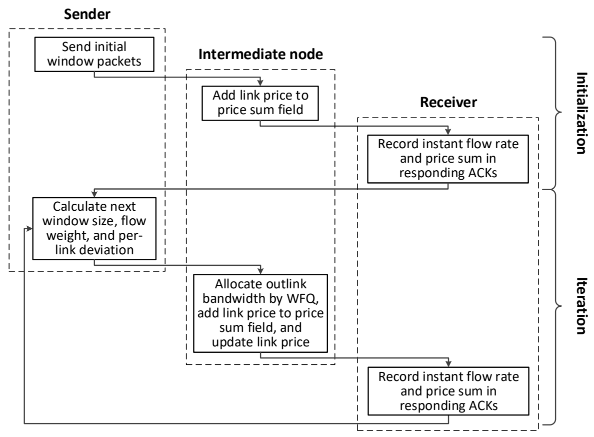

Sender
Send initial window packets
Intermediate node
Add link price to price sum field
Receiver
Record instant flow rate and price sum in responding ACKs
Calculate next window size, flow weight, and per-link deviation
Allocate outlink bandwidth by WFQ, add link price to price sum field, and update link price
Record instant flow rate and price sum in responding ACKs
Initialization
Iteration

Fig. 2: Bandwidth allocation flowchart

$$\delta(t) = \frac{1 + \sqrt{1 + 4\delta^2(t-1)}}{2} \tag{13}$$

$$\lambda_l(t+1) = \left[\nu_l(t+1) + \frac{1-\delta(t-1)}{\delta(t)} (\nu_l(t+1) - \nu_l(t)) \right]_+ \tag{14}$$

If a link encounters new flow arrival in iteration t, it will give up above price update in the next iteration $t+1$ and reserve the price value in iteration t. This is because the new flow's initial weight does not come from current link prices, then its resulting flow rates should not act on link prices either. This also avoids one-iteration tiny flows affecting other flows' convergence. So far, iteration t has ended. Detailed operations of sender, intermediate node, and receiver are depicted in Fig.2. They work in coordination and trigger each other. The top half explains the initialization process, while the remaining half describes iterations.

The iteration stops when both $\boldsymbol{b}$ and $\boldsymbol{\lambda}$ no longer change. Each time a flow finishes, no additional actions should be taken and other flows will automatically occupy the vacated bandwidth. DBA sender each time only sends the data volume that could be exactly transmitted, thus packet loss will be rare. However, DBA still reserves TCP fast retransmit and recovery mechanism to respond to packet drops, in case of extreme conditions like numerous concurrent tiny flows.

It is obvious that both Equation 10 and 11 are for condition 7(a). Equation 10 inversely solves b_i in 7(a), but the result is used as the flow weight rather than rate, and the actual flow rate hinges on WFQ. Therefore, condition 7(a) may not always set up and there may be an offset. Equation 11 calculates this offset, which is rectified in link price updates ($\min_{i \in F(l)} e_i(t)$ in Equation 12). In this way, 7(a) is pushed to be true. For another, Equation 12, 13, and 14 compose Nesterov's AGD method and work for condition 7(b). Because of WFQ, link bandwidth can never be oversubscribed. Therefore, for any l and t, $\sum_{i \in F(L)} b_i(t) - B'_l \leq 0$. For bottleneck links, $\sum_{i \in F(L)} b_i(t) - B'_l = 0$, thus 7(b) is already satisfied and λ_l can take any positive value. However, for underutilized links, λ_l must be 0. Correspondingly, except the correction term $\min_{i \in F(l)} e_i(t)$, Equation 12, 13, and 14 make no difference on bottleneck link prices, but always drive the price to 0 for underutilized links. In our enhanced dual decomposition algorithm, WFQ guarantees feasibility of all $b_i(t)$ in every moment, and AGD accelerates the flow rate convergence. In subsection 4.4 we will prove that the bandwidth allocation will converge to the optimal solution as well as the accelerated convergence rate.

4.3 Algorithm Practical Design

We functionally outline the operations at the sender, receiver, and switch. In the transport layer, DBA adds five fields to SYN packet header: `flowSize`, `parentCoflowId`, `parentCCT`, `virtualFlowNum`, and `pathLen`, and three fields to SYN-ACK packet header: `parentCCT`, `virtualFlowNum`, and `pathLen`; adds three fields to normal data packet header: `pathPrice`, `flowWeight`, and `perLinkDev`, and two fields to ACK packet header: `pathPrice` and `pktInterval`.

4.3.1 Receiver

The receiver gets the maximum local transmission time, capacity-normalized local number and hop count from `parentCCT`, `virtualFlowNum` and `pathLen` in SYN header, and the link price sum from `pathPrice` in data packet header.

On getting each SYN packet, the receiver compares `parentCCT` value with the native T_k^{out}, if `parentCCT` value is larger, use it to update T_k^{out} and use `virtualFlowNum` value to update N_k^{out}. Then simply reflects T_k^{out}, N_k^{out} and `pathLen` value back to the sender in `parentCCT`, `virtualFlowNum` and `pathLen` fields in SYN-ACK.

On getting each data packet, the receiver reflects the inter-packet time in the `pktInterval` field in ACK header. If this packet also has a valid `pathPrice` value, the receiver also reflects it back to the sender in `pathPrice` field.

4.3.2 Sender

The sender maintains the inverse of the marginal utility function $U_i'^{-1}(x) = (\frac{xT_{k(i)}(t)}{1-\epsilon})^{-\frac{1}{\epsilon}}$.

On getting each SYN-ACK packet, the sender gets the maximum local transmission time, capacity-normalized local number and hop count from `parentCCT`, `virtualFlowNum` and `pathLen` fields respectively. Then it compares `parentCCT` value with the native T_k^{in}, if `parentCCT` value is larger, use it to update T_k^{in} and use `virtualFlowNum` value to update N_k^{out}.

On getting each ACK packet, the sender gets the instant packet interval I_i from `pktInterval`. It then uses I_i to update the average flow rate by $\beta\bar{b}_i(t-1) \leftarrow \frac{\widetilde{S}_i}{\sum_{n_i} I_i}$, where n_i denotes the number of received ACKs of flow i's last window and $\widetilde{S}_i$ the total ACKed data size. Then the window size $W(t)$ is recalculated with the updated $\beta\bar{b}_i(t-1)$. If this ACK also has a valid `pathPrice` value, the sender will also use it combined with $U_i'^{-1}(\cdot)$ and $\bar{b}_i(t-1)$ to compute the flow weight $\omega_i(t)$ and per-link deviation $e_i(t)$.

4.3.3 Switch

The DBA switch needs to maintain the local remaining completion time, and implement both WFQ scheduling and link price computation.

For local remaining time maintenance, each time a SYN packet arrives, the switch only needs to conduct simple functions of `divide` ($\frac{S_i}{B_l}$ and $\frac{1}{B_l}$), `add` ($T_{kl} + \frac{S_i}{B_l}$ and $N_{kl} + \frac{1}{B_l}$), `compare` ($T_k^{(i)} < T_{kl}$?), `assign` ($T_k^{(i)} \leftarrow T_{kl}$ and $N_k^{(i)} \leftarrow N_{kl}$), `itemCreate` (when new coflow comes), and `itemDelete` (when coflow completes).

WFQ packet scheduling could be through the Start Time Fair Queuing (STFQ) algorithm [36], and here we briefly sketch out its conceptual design. STFQ assigns a virtual start time and a virtual finish time to each packet. For the n^{th} packet of flow i, STFQ computes its virtual start time $s_n^{(i)}$ and finish time $f_n^{(i)}$ upon its arrival as follows:

$$s_n^{(i)} = \max(V, f_{n-1}^{(i)}) \tag{15}$$

$$f_n^{(i)} = s_n^{(i)} + \frac{L_n^{(i)}}{\omega_i} \tag{16}$$

where V is the virtual time at packet arrival, $L_n^{(i)}$ is the packet size, and ω_i the current weight of flow i. Note that the virtual time is not the actual time, because ω_i is not the real flow rate. Even so, the relative order with which packets would depart in WFQ is guaranteed, which is already enough. STFQ then schedules packets in ascending order of their virtual start time, each with full link bandwidth. By this means, STFQ approximates WFQ.

The above design requires a priority queue to schedule packets according to virtual start time, which is not supported in commodity switches. However, recent work [37] has shown that it is feasible to implement priority queues in emerging programmable switching chips. The basic principle is that packets are inserted into the queue based on a programmable rank value that is precomputed. Here we directly refer the readers to [37].

For link price computation, the switch updates the price for each outgoing link periodically. As long as a same update period is set for all switches, the price updates of all links could be synchronized. This can be achieved with Precision Time Protocol (PTP) [38], a general function in commodity switches. The update period T could be set to the minimum RTT (without queueing delay).

A switch maintains not only a price λ_l but also four auxiliary variables ν_l, ν_l', $minDev_l$, and $bytesServ_l$ for each outgoing link. Besides, there is a global variable δ. Each time the switch receives a data packet, it first adds the packet size to $bytesServ_l$. Then it judges whether `pathPrice` and `perLinkDev` fields are valid. If it is the former, it adds the corresponding λ_l to `pathPrice` value. And it judges whether the `perLinkDev` field is valid. If it is the latter, it compares `perLinkDev` value with the corresponding $minDev_l$. If `perLinkDev` value is smaller, then use it to update $minDev_l$. Each time the update timeout occurs, the switch updates all its outgoing link prices as follows:

$$\nu_l \leftarrow \lambda_l + minDev_l - \frac{\gamma}{\beta}(B_l - \frac{bytesServ_l}{T})$$

$$\lambda_l \leftarrow \max\{\nu_l + \frac{1-\sqrt{1+4\delta^2}}{1+\sqrt{1+4\delta^2}}(\nu_l - \nu_l'), 0\}$$

$$\delta \leftarrow \frac{1+\sqrt{1+4\delta^2}}{2},\ \nu_l' \leftarrow \nu_l,\ minDev_l \leftarrow \infty, bytesServ_l \leftarrow 0$$

If the switch fails to do timely computations, there will be a lag in link price update, which may slower the flow rate convergence. However, it does not affect the fact that the bandwidth allocation will finally converge to the optimal, and the resulting influence on DBA performance is not serious. We will evaluate this influence in Section 5.4.2.

4.4 Algorithm Analysis

In this subsection, we conduct analysis on our DBA algorithm, including its convergence rate and introduced overheads.

4.4.1 Convergence Rate

We have the following theorem for DBA convergence:

Theorem 1. *DBA has a convergence rate of* $\Theta(\frac{1}{t^2})$*, where* t *is the iteration number.*

Proof. Let $\tilde{e}_l(t) = \min_{i\in F(l)} e_i(t)$, f_l denote the partial objective function $\lambda_l(B'_l - \sum_{i\in F(l)} b^*_i)$ of λ_l in the master problem (5), $\forall t \geq 0, \forall l$,

$$\begin{aligned} & f_l(\nu_l(t+1)) - f_l(\nu_l(t)) = f_l(\lambda_l(t) + \tilde{e}_l(t+1) - \gamma f'_l(\lambda_l(t))) \\ & - f_l(\nu_l(t)) \\ \leq\ & f_l(\lambda_l(t) + \tilde{e}_l(t+1) - \gamma f'_l(\lambda_l(t))) + f'_l(\lambda_l(t))(\lambda_l(t) - \nu_l(t)) \\ & - f_l(\lambda_l(t)) \\ \leq\ & f'_l(\lambda_l(t))(\tilde{e}_l(t+1) - \gamma f'_l(\lambda_l(t))) + \frac{1}{2}(\tilde{e}_l(t+1) - \gamma f'_l(\lambda_l(t)))^2 \\ & + f'_l(\lambda_l(t))(\lambda_l(t) - \nu_l(t)) \\ =\ & -\frac{1}{2\gamma}(\nu_l(t+1) - \lambda_l(t) - \tilde{e}_l(t+1))^2 + \frac{1}{2\gamma}\tilde{e}_l^2(t+1) \\ & - \frac{1}{\gamma}(\nu_l(t+1) - \lambda_l(t) - \tilde{e}_l(t+1))(\lambda_l(t) - \nu_l(t)) \end{aligned} \quad (17)$$

Replace $\nu_l(t)$ with λ^*_l, the above inequality still holds, i.e.,

$$\begin{aligned} & f_l(\nu_l(t+1)) - f_l(\lambda^*_l) \leq -\frac{1}{2\gamma}(\nu_l(t+1) - \lambda_l(t) - \tilde{e}_l(t+1))^2 \\ & + \frac{1}{2\gamma}\tilde{e}_l^2(t+1) - \frac{1}{\gamma}(\nu_l(t+1) - \lambda_l(t) - \tilde{e}_l(t+1))(\lambda_l(t) - \lambda^*_l) \end{aligned} \quad (18)$$

Multiplying (17) by $(\alpha(t)-1)$ and adding to (18), we can obtain the following inequality with $\delta_l(t) = f_l(\nu_l(t)) - f_l(\lambda^*_l)$

$$\begin{aligned} & \alpha(t)\delta_l(t+1) - (\alpha(t)-1)\delta_l(t) \leq -\frac{1}{\gamma}(\nu_l(t+1) - \lambda_l(t) - \tilde{e}_l(t+1)) \\ & * (\alpha(t)\lambda_l(t) - (\alpha(t)-1)\nu_l(t) - \lambda^*_l) + \frac{\alpha(t)}{2\gamma}\tilde{e}_l^2(t+1) \\ & - \frac{\alpha(t)}{2\gamma}(\nu_l(t+1) - \lambda_l(t) - \tilde{e}_l(t+1))^2 \end{aligned}$$

Multiplying this inequality by $\alpha(t)$, using the relation $\alpha(t-1)^2 = \alpha(t)^2 - \alpha(t)$ and picking out the items containing $\tilde{e}_l(t+1)$, we obtain

$$\begin{aligned} & \alpha^2(t)\delta_l(t+1) - \alpha^2(t-1)\delta_l(t) \leq -\frac{1}{2\gamma}\ (\ 2\alpha(t)(\nu_l(t+1) - \lambda_l(t)) \\ & * (\alpha(t)\lambda_l(t) - (\alpha(t)-1)\nu_l(t) - \lambda^*_l) + \alpha^2(t)(\nu_l(t+1) - \lambda_l(t))^2\ ) \\ & - \frac{\alpha(t)}{\gamma}\ (\alpha(t)(\nu_l(t) - \nu_l(t+1)) - \nu_l(t) + \lambda^*_l)\ \tilde{e}_l(t+1) \end{aligned} \quad (19)$$

One can verify that,

$$\begin{aligned} & 2\alpha(t)\ (\nu_l(t+1) - \lambda_l(t)) * (\alpha(t)\lambda_l(t) - (\alpha(t)-1)\nu_l(t) - \lambda^*_l) \\ & + \alpha^2(t)(\nu_l(t+1) - \lambda_l(t))^2 = -(\alpha(t)\lambda_l(t) - (\alpha(t)-1)\nu_l(t) - \lambda^*_l)^2 \\ & + (\alpha(t)\nu_l(t+1) - (\alpha(t)-1)\nu_l(t) - \lambda^*_l)^2 \end{aligned} \quad (20)$$

By definition,

$$\alpha(t+1)\lambda_l(t+1) - (\alpha(t+1)-1)\nu_l(t+1) = \alpha(t)\nu_l(t+1) - (\alpha(t)-1)\nu_l(t) \quad (21)$$

Synthesizing (19), (20), and (21) and defining $u_l(t) = \alpha(t)\lambda_l(t) - (\alpha(t)-1)\nu_l(t) - \lambda^*_l$, we have

$$\begin{aligned} & \alpha^2(t)\delta_l(t+1) - \alpha^2(t-1)\delta_l(t) \leq \frac{1}{2\gamma}(u_l^2(t) - u_l^2(t+1)) \\ & - \frac{\alpha(t)}{\gamma}(\alpha(t)(\nu_l(t) - \nu_l(t+1)) - \nu_l(t) + \lambda^*_l)\tilde{e}_l(t+1) \end{aligned} \quad (22)$$

One can verify that, for any link l the last term containing $\tilde{e}_l(t)$ constitutes a sequence with respect to t, where the sum of any first n terms is non-positive. Using this property and summing up inequality (22) from 0 to $t-1$, we have

$$\delta_l(t) \leq \frac{1}{2\gamma\alpha^2(t-1)}u_l^2(0)$$

It is easy to see that $\alpha(t-1) \geq \frac{t}{2}$, and Theorem 1 is proved. □

Theorem 1 indicates DBA will converge to the optimal solution of problem (2), and the convergence rate is quadratic. For example, if the difference between a link's initial price and the optimal value is 9, only after 3 iterations, the difference reduces to 1.

4.4.2 Overhead

Here we briefly analyze the overheads DBA imposes on bandwidth. In the next section, we will further evaluate these overheads by simulations.

For SYN and SYN-ACK, `flowSize` needs 4 bytes, because flow sizes in data centers are generally below 100MB [7, 39, 40]. `parentCCT` and `virtualFlowNum` each need 4 bytes, since they each could be covered by a floating point number. `parentCoflowId` requires 2 bytes, because coflow id could be reused in cycles and 2 bytes are enough to guarantee no id overlap at the same time. The above fields are all added in the TCP option field. Finally, `pathLen` could directly use the 6-bit reserved field in original header and generate no additional overhead, since hop count in data centers will not exceed 64 [41]. As a whole, DBA adds extra 14 bytes to SYN and 8 bytes to SYN-ACK.

For data packet and ACK, `pathPrice`, `flowWeight`, `perLinkDev`, and `pktInterval` all can be covered by a floating point number, thus they each need 4 bytes. Namely, DBA introduces 12 bytes in data packet and 8 bytes in ACK.

According to measurements [39, 40], the average flow size in the data center is no less than 1MB (80% below 5KB [40] and 20% between 5KB and 100MB [39]). In original TCP, 1MB of data need 685 packets (MSS=1460B), each with a 20-byte header. Correspondingly, there also need 686 20-byte ACKs (the extra one is during three-way handshake). Besides, SYN and SYN-ACK are both 20 bytes long. While in DBA, there needs 691 data packets (MSS=1448B), each with a 32-byte header, and 692 28-byte ACKs. In addition, DBA's SYN and SYN-ACK are separately 34 and 28 bytes long. Finally, the average DBA overhead is $\frac{1MB+32B*691+28B*692+34B+28B}{1MB+20B*(685+686+2)} - 1 = 1.37\%$, which is small. Besides, except the normal data transmission and ACK feedback, there is no extra communications in DBA.

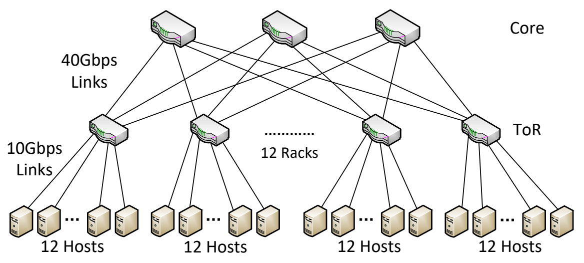


Fig. 3: Simulation topology

5 Evaluation

In this section, we evaluate DBA performance through a series of trace-driven simulations. We first evaluate DBA's CCT performance, throughput, convergence time and overheads. Then we explore the influence of routing strategies, parameters, and computation delays on DBA performance.

5.1 Simulation Setup

Platform and topology: The simulation platform is our self-developed flow-level simulator in java. And the simulation topology is the commonly used leaf-spine topology shown in Fig.3. The fabric has 3 layers and interconnects 144 servers through 12 top of rack (ToR) and 3 aggregation switches. Each rack contains 12 servers with each connecting to the ToR switch by a 10Gbps link. All 12 ToR switches and 3 core switches are fully connected with 40Gbps links. Such settings result in a physically non-oversubscribed topology.

Workload: The coflow workloads in our simulations come from [7], where large-scale Facebook traces as well as distributions of coflow characteristics (length, width, size, skew, etc.) are provided. We replay these traces and scale them down to adapt to our topology and 1440Gbps of bisection bandwidth. The Facebook traces only provide coflow arrival time, endpoints, and aggregated shuffle size, but we still need individual flow sizes and routing paths. For flow size, we additionally leverage the CDF of variation coefficient of flow sizes within each coflow [7]. From Facebook traces, we can get a coflow's total size and inner flow number, hence can calculate average flow size. Then we randomly take a variation coefficient value in terms of the CDF. With both the average and variation coefficient, the standard deviation is the product of them. Finally, we stochastically generate normally distributed inner flow sizes according to the above average and standard deviation. For spatial distribution, each individual flow's source and destination are randomly chosen from the whole fabric with homogeneous probability. However, the route path is not determined now, but at runtime by the routing algorithm in the simulator. In order to study DBA performance under different load conditions, we scale the workload from 10% to 100%. For our 10Gbps edge links, we respectively generate 1Gbps, 2Gbps, $\cdots$, 10Gbps of traffic loads on each server.

Performance metrics: We mainly concern 3 metrics:

- **CCT**. CCT is the most primary and important metric for coflow scheduling mechanisms. We separately calculate the average and 99th percentile CCT because they both matter in application performance.
- **Throughput**. Throughput is another basic metric that data center applications care for. In data centers, besides those small coflows, there are also very large background coflows ($>$ 1GB) working for data updates. Such large coflows account for 99.6% of total bytes [7] and are typically throughput-sensitive.
- **Convergence time**. Because fast convergence is a key merit of DBA, we observe the time for flows to reach the optimal rate to evaluate DBA convergence.

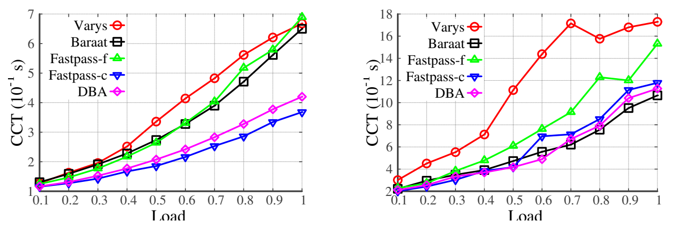
Reference Algorithms: We take Varys [7], Baraat [6], and Fastpass [25] as reference algorithms for DBA. First of all, Varys is a typical centralized mechanism based on the giant-switch hypothesis and has superior CCT performance among online coflow scheduling mechanisms. Second, Baraat is a distributed mechanism not assuming non-blocking in-network links but adopting the FIFO principle. Finally, Fastpass is a centralized packet-level scheduling mechanism, including both timeslot allocation and path selection. It can exactly manipulate every packet's transmission time and routing, guaranteeing no packet collision on any link. Thus in theory, it could achieve MRTF ideally. Unfortunately, Fastpass is an individual flow scheduling mechanism and not coflow-aware. However, here we implement both flow-level and coflow-level MRTF with Fastpass and use the coflow-aware Fastpass to provide the optimal measure.

5.2 Algorithm Performance

When exploring basic performance metrics, we simply set the two DBA parameters $\epsilon = 0.5$ and $\gamma = 5 \times 10^{-18}$ for the moment, and the routing strategy is ECMP. We will discuss the influence of these factors later.

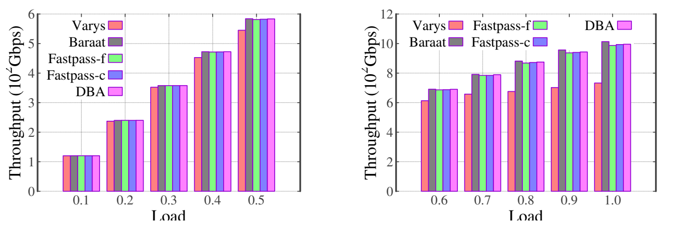
Fig.4 depicts average as well as 99th percentile CCTs of Varys, Baraat, Fastpass and DBA under various traffic loads. Here Fastpass-f implements flow-level MRTF while Fastpass-c the coflow-level one. We can see that, on average CCT, DBA outperforms Varys, Baraat and Fastpass-f by 37.1%, 35.4% and 39.1% at 100% workload respectively. This is mainly because DBA considers in-network bottlenecks thus obtains more precise coflow priorities and bandwidth allocation than Varys, and adopts more CCT-effective coflow-level MRTF than Baraat (FIFO) and Fastpass (flow-level MRTF). Another potential reason is that Varys suffers from link utilization loss [16], while DBA inherently achieves high link utilization (KKT condition). Relative to Fastpass-c (theoretical optimum), DBA only has a 14% increase, which comes from distributed implementation, rate convergence time and routing collisions. On 99th percentile CCT, DBA is lower by 34.8%, 26.4%, and 4.4% than Varys, Fastpass-f and Fastpass-c respectively, and a little higher by 5.8% than Baraat. This is because DBA and Baraat achieve similar bandwidth utilization and Baraat's FIFO is more friendly to large flows than DBA's MRTF.

Fig.5 depicts CDFs of CCT for the five tested algorithms. Fig.5(a) is under 50% workload and Fig.5(b) is under 100%. It can be observed that under both loads, DBA outperforms Varys, Baraat and Fastpass-f in almost all percentiles, and is close to Fastpass-c. Only below 15th percentile, DBA's CCT is a little larger than Varys, which is mainly due to the convergence time. And at tail CCTs, DBA performs similarly to Baraat, which is consistent with the results in Fig.4(b).



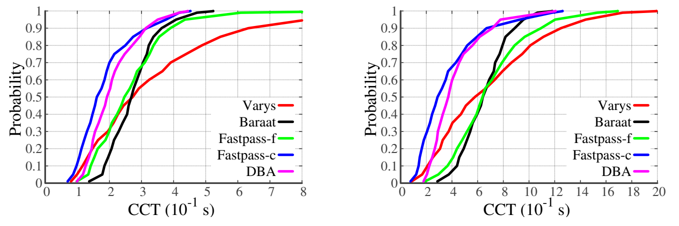
(a) Average CCT (b) 99th percentile CCT

Fig. 4: CCT performance under varying loads



(a) Throughput under low loads (b) Throughput under high loads

Fig. 6: Throughput under different workloads



(a) CDF under 50% workload (b) CDF under 100% workload

Fig. 5: CCT distributions under low and high workloads

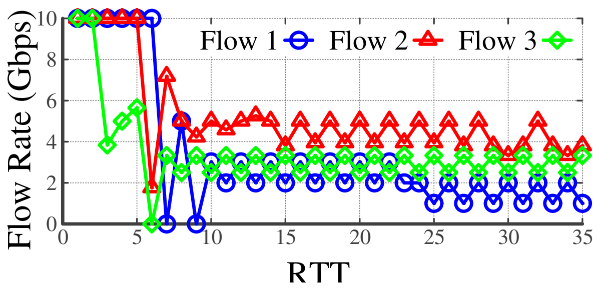


Fig. 7: Flow rate convergence

Fig.6 compares the throughput of five mechanisms under different workloads, which also partly explains the better CCT performance of DBA in Fig.4. The throughput here is all data volume transferred through the whole fabric divided by the makespan of all coflows. Under lower loads ($\leq$ 50%) five algorithms have similar throughputs, while under higher loads ($>$ 50%) DBA achieves the throughput apparently higher than that of Varys (35.8% at 100% load), a little less than that of Baraat (1.7% at 100 load), and similar to that of Fastpass-f and Fastpass-c. On one hand, Varys has problem with bandwidth utilization. On the other hand, relative to Baraat, DBA has a convergence process, thus has slight throughput loss. Thus we can say, DBA maintains good link utilization at the same time of optimizing CCT.

Next, DBA convergence is investigated. We randomly pick out three flows and depict their rate convergence in Fig.7. The horizontal axis denotes the number of RTTs, and the vertical represents flow rate. All three flows converge to their optimal rates within 6 RTTs. By our statistical analysis, the convergence time is 5-6 RTT. For tiny flows with size less than W_0, they directly complete at the large initial rate. For the rest, except when giving way to initial flow packets, flows lasting for over 6 RTTs will mostly converge to their optimal rates, and the average rate before convergence is no smaller than optimal because flow rate converges from above. Thus, flows failing to converge before completion see no rate loss by convergence process either.

5.3 Algorithm Overheads

Fig.8 respectively depicts the overheads DBA introduces to bandwidth (Fig.8(a)), CPU (Fig.8(b)) and memory (Fig.8(c)).

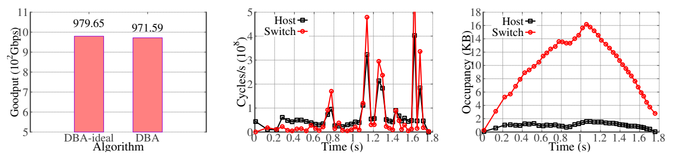
Fig.8(a) shows the goodputs of DBA and DBA-ideal at 100% workload. Here DBA-ideal reproduces all DBA functions but makes no changes to original packets, thus consumes no extra bandwidth. Note that DBA-ideal is unrealizable in practice, and we only use it in simulations to measure DBA overheads. We can see the goodput of DBA is 8.06Gbps less than the ideal value, accounting for 0.82%, which is really small. In Fig.8(b), DBA's CPU consumption is illustrated. According to the number of CPU cycles each operation needs [42], the black and red lines separately describe the consumed CPU cycles per second on a host and an aggregation switch at 100% workload. On average, DBA needs 8.0×10^7 and 8.9×10^7 cycles per second on host and switch, which only account for 3.3% and 3.7% in common 2.4GHz CPUs. Besides, DBA adds one `compare`, one `divide`, and two `add` operations to each packet processing on switch, which occupy minimal chip resources. Similarly, Fig.8(c) shows the memory footprint of DBA on a host and an aggregation switch at 100% workload. It can be observed that the maximum DBA occupancy are only 2KB and 16KB.

5.4 Discussion on Factors

After basic evaluations, we discuss the influence of different factors on DBA, including routing policies and parameters.

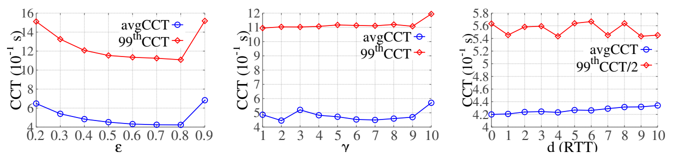
5.4.1 *Routing Policy*

In-network bottlenecks also depend on flow routing. The more evenly the routing policy distributes traffic, the less bottleneck will occur. On performance metrics, routing first affects throughput, and throughput is inversely proportional to makespan at full load. If coflows fairly share link bandwidth, when routing policy A's throughput is x times that of policy B, for any coflow its completion time ratio under two policies $\frac{C_B}{C_A} = \frac{xT_C - T_A}{T_C - T_A}$, which is larger than x with T_A denoting the coflow's arrival moment and T_C the completion moment under policy A. However, under whichever routing policy, DBA always approximates MRTF coflow order on every link, which makes a difference. If there appear more collisions, small coflows will still go first, whose CCT will be less harmed. Only a part of large coflows are slowed down by the same proportion as in fair-sharing.



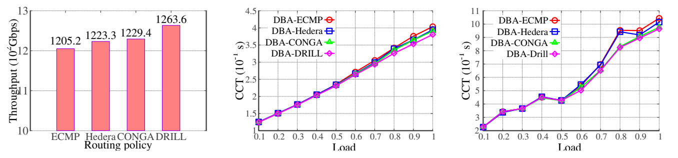
(a) Bandwidth (b) CPU cycle (c) Memory

Fig. 8: DBA overheads



(a) Power influence (b) Stepsize influence (c) Delay influence

Fig. 10: Influence of DBA Factors



(a) Throughput (b) Average CCT (c) 99th percentile CCT

Fig. 9: DBA under different routing Policies

Fig.9 shows throughput, average and 99th percentile CCT of DBA under different routing policies, including flow-level ECMP [30] and Hedera [18], flowlet-level CONGA [20], and packet-level DRILL [29]. Among them DRILL is a near-optimal load balance mechanism. Here we increases the number of hosts per rack to construct a 4:3 oversubscription to highlight routing effect. At 100% load, DRILL achieves the throughput 1.048, 1.033 and 1.028 times that of ECMP, Hedera and CONGA. In turn, 50th and 99th percentile CCTs of these three are respectively 1.060, 1.036, 1.027, and 1.124, 1.071, 1.050 that of DRILL, which is consistent with above analysis. And we can see under these policies DBA performance exhibits very little difference (the maximum gaps are 6% in average CCT and 12% in 99th percentile CCT). Thus we can say the influence of routing policies on DBA performance is not evident.

5.4.2 Parameters

In DBA, there are two adjustable parameters: power ϵ in the objective function and step size γ in the evolution formula. In addition, there may exist a lag d in link price update. Fig.10 depicts DBA performance with these three quantities varying in a large range. Fig.10(a) manifests average and 99th percentile CCTs under different ϵ values. ϵ serves in the objective function $\sum_{i=1}^{n(t)} \frac{b_i(t)^{1-\epsilon}}{T_{k(i)}(t)}$ to make it strongly convex. As ϵ increases, the objective function becomes more convex and CCT sees reduction. However, CCT always keeps at a low level when $\epsilon \geq 0.3$, i.e., DBA performs well in a wide range of ϵ values. When it comes to γ, Fig.10(b) describes the variation of CCTs in a wide γ range. The very small magnitude of γ is to counteract the very large link bandwidth (10^{10}bps). As γ grows, there is no obvious changes in both average and 99th percentile CCTs. That is, DBA is not sensitive to the parameter γ. Therefore we can say, DBA has great robustness to parameter changes. Finally, Fig.10(c) illustrates the CCTs with different computation delay d. As d increases from 0 to 10 RTTs, the average CCT gently grows from to 0.420 to 0.434, the growth rate being only 3.3%. And the 99th percentile CCT always fluctuates in a small range and exhibits no correlation with d. That is, DBA is also robust to computation delays.

6 Conclusion

In this paper, we propose DBA, a distributed bottleneck-aware coflow scheduling algorithm in data centers. DBA addresses the performance loss by neglecting in-network bottlenecks, leveraging the underlying transport protocols to achieve the network-wide MRTF scheduling. Besides, since DBA enhances the dual decomposition method in solving flow rates, it converges very fast. We conduct extensive simulations under practical traffic loads and configurations to evaluate DBA. Results show that DBA not only achieves superior CCT performance (lower by 37%, 35% and 39% in average CCT than Varys, Baraat and Fastpass), but also holds high throughput (larger by 36% than Varys, similar to Baraat and Fastpass). Furthermore, DBA indeed converges very fast and introduces very little overhead to bandwidth, CPU and memory. Finally, DBA is robust to routing strategies, parameter variations and computation delays. Benefiting from the decentralized design, DBA has no problem with scalability.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, 2008.
[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS*, 2007.
[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.
[4] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Hotnets*, 2012.
[5] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *SIGCOMM*, 2011.
[6] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *SIGCOMM*, 2014.
[7] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *SIGCOMM*, 2014.
[8] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *SIGCOMM*, 2015.
[9] Z. Qiu, C. Stein, and Y. Zhong, "Minimizing the total weighted completion time of coflows in datacenter networks," in *SPAA*, 2015.
[10] Y. Zhao, K. Chen, W. Bai, M. Yu, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang, "Rapier: Integrating routing and scheduling for coflow-aware data center networks," in *INFOCOM*, 2015.
[11] S. Luo, H. Yu, Y. Zhao, S. Wang, S. Yu, and L. Li, "Towards practical and near-optimal coflow scheduling for data center networks," *TPDS*, 2016.
[12] X. S. Huang, X. S. Sun, and T. Ng, "Sunflow: Efficient optical circuit scheduling for coflows," in *CoNEXT*, 2016.
[13] H. Susanto, H. Jin, and K. Chen, "Stream: Decentralized opportunistic inter-coflow scheduling for datacenter networks," in *ICNP*, 2016.

[14] Y. Li, S. H.-C. Jiang, H. Tan, C. Zhang, G. Chen, J. Zhou, and F. Lau, "Efficient online coflow routing and scheduling," in *MobiHoc*, 2016.

[15] Z. Li, Y. Zhang, D. Li, K. Chen, and Y. Peng, "Optas: Decentralized flow monitoring and scheduling for tiny tasks," in *INFOCOM*, 2016.

[16] J. Jiang, S. Ma, B. Li, and B. Li, "Adia: Achieving high link utilization with coflow-aware scheduling in data center networks," *TCC*, 2017.

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

[18] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." in *NSDI*, 2010.

[19] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *SIGCOMM*, 2013.

[20] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," in *SIGCOMM*, 2014.

[21] A. Vahdat, M. Al-Fares, N. Farrington, R. N. Mysore, G. Porter, and S. Radhakrishnan, "Scale-out networking in the data center," *IEEE Micro*, 2010.

[22] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*, 2008.

[23] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *SIGCOMM*, 2009.

[24] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: a scalable fault-tolerant layer 2 data center network fabric," in *SIGCOMM*, 2009.

[25] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized zero-queue datacenter network," *SIGCOMM*, 2015.

[26] C.-Y. Hong, M. Caesar, and P. Godfrey, "Finishing flows quickly with preemptive scheduling," *SIGCOMM*, 2012.

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

[28] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker, "phost: Distributed near-optimal datacenter transport over commodity network fabric," in *CoNEXT*, 2015.

[29] S. Ghorbani, Z. Yang, P. Godfrey, Y. Ganjali, and A. Firoozshahian, "Drill: Micro load balancing for low-latency data center networks," in *SIGCOMM*, 2017.

[30] C. Hopps, "Analysis of an equal-cost multi-path algorithm," Tech. Rep., 2000.

[31] Z. Huang, B. Balasubramanian, M. Wang, T. Lan, M. Chiang, and D. H. Tsang, "Need for speed: Cora scheduler for optimizing completion-times in the cloud," in *INFOCOM*, 2015.

[32] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "Coda: Toward automatically identifying and scheduling coflows in the dark," in *SIGCOMM*, 2016.

[33] D. P. Palomar and M. Chiang, "A tutorial on decomposition methods for network utility maximization," *IEEE Journal on Selected Areas in Communications*, 2006.

[34] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *SIGCOMM*, 1989.

[35] Y. Nesterov, *Introductory lectures on convex optimization: A basic course*, 2013.

[36] P. Goyal, H. M. Vin, and H. Chen, "Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks," in *SIGCOMM*, 1996.

[37] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet scheduling at line rate," in *SIGCOMM*, 2016.

[38] J. Eidson and K. Lee, "Ieee 1588 standard for a precision clock synchronization protocol for networked measurement and control systems," in *ISA/IEEE*, 2002.

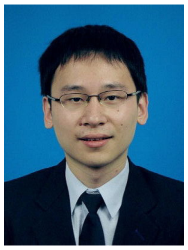
[39] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," *SIGCOMM*, 2011.

[40] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *SIGCOMM*, 2010.

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

[42] A. Fog, "Instruction tables (2014)," *URL: http://www. agner. org/optimize/instruction_tables. pdf. Citations in this document*, 2018.

Tong Zhang Tong Zhang is currently a Ph.D candidate majoring in Computer Science and Technology at Tsinghua University, under the advising of Prof. Fengyuan Ren and Prof. Chuang Lin. In 2014, she received her B.E. degree in Computer Science and Technology from Xi'an Jiaotong University. Tong Zhang's research interests are primarily centered on flow and coflow scheduling in clusters and data centers.



Ran Shu Ran Shu is currently a of Microsoft Research, Beijing, China. In 2011, he received his B.E. degree in Computer Science and Technology from Tsinghua University. In 2018, he received his Ph.D degree in Computer Science and Technology from Tsinghua University. His recent research mainly focuses on the field of traffic management in data center networks. He has published over 5 papers on top conferences or journals of networking.

Zhiguang Shan SHAN Zhiguang, born in 1974, PhD, professor and the director of Informatization and Industry Development Department, State Information Center of China. He received the B.A. degree in automation engineering, and the Ph.D. degree in computer science, both from the University of Science and Technology Beijing, Beijing, China, in 1997 and 2002, respectively. His main research interests include computer networks, performance evaluation, strategic planning and top design of smart city, macro planning and developing policies of informatization. He has co-authored more than 60 papers in research journals and conference proceedings and 9 books in these areas.

Fengyuan Ren Fengyuan Ren (M04) is a professor of the Department of Computer Science and Technology at Tsinghua University, Beijing, China. He received his B.A and M.Sc. degrees in Automatic Control from Northwestern Polytechnic University, China, in 1993 and 1996 respectively. In Dec. 1999, he obtained Ph.D degree in Computer Science from Northwestern Polytechnic University. From 2000 to 2001, he worked at Electronic Engineering Department of Tsinghua University as a post doctoral researcher. In Jan. 2002, he moved to the Computer Science and Technology Department of Tsinghua University. His research interests include network traffic management and control, control in/over computer networks, wireless networks and wireless sensor networks. He (co)-authored and co-authored more than 80 international journal and conference papers. He is a member of the IEEE, and has served as a technical program committee member and local arrangement chair for various IEEE and ACM international conferences.