Near Optimal Multi-Faced Job Scheduler for Datacenter Workloads *

[1,2]Hengky Susanto, [1,3]Ahmed M. Abdelmoniem, [4]Honggang Zhang, [5]Benyuan Liu, and [6]Don Towsley
[1]Dept. CSE, Hong Kong University of Science and Technology, HK. [2]Huawei Future Network Theory Lab, HK.
[3]CS Dept., FCI, Assiut University, Egypt. [4]Engineering Dept., University of Massachusetts Boston, US.
[5]CS Dept., University of Massachusetts Lowell, US. [6]CS Dept., University of Massachusetts Amherst, US.
hsusanto@cs.uml.edu, amas@cse.ust.hk, honggang.zhang@umb.edu, bliu@cs.uml.edu, towsley@cs.umass.edu

Abstract—**As data-parallel applications process more complex data, the dependencies between computation jobs in a multi-stage job also become more complicated. However, most of the existing scheduling solutions primarily rely on total bytes sent (job size) to differentiate jobs where jobs with fewer bytes sent are prioritized over the larger ones. This approach overlooks the fact that jobs may consist of multiple computation stages, and that the completion of a computation job stage depends on the completion of other jobs' stage. In this paper, we present a coflow scheduler of multi-stage jobs that minimizes the average job completion time. Our solution prioritizes jobs based on the multi-faceted characteristics of multi-stage job structure per stage, instead of total bytes sent. Our experiments show that our approach provides twice the performance of existing solutions on average and by four times in bursty traffic scenario.**

I. Introduction

Today, modern distributed parallel computing frameworks (E.g. MapReduce[18], Dryad [21]) are commonly employed in datacenters to process distributed computing jobs for data analyzation or queries. These frameworks typically have multiple successive computation stages. By multiple stages we refer to a pipeline of successive computation phases where a phase only begins processing when the previous computation stage completes and the data flows generated by the previous computation stage also complete. Each computation stage usually consists of multiple tasks that are processed in parallel.

Data transfers between two successive stages usually involves a collection of flows with the same performance objective, which is referred as coflow [4]. Coflow provides context to this collection of flows, where data transfer between two successive stages complete when *all* its flows complete. Studies in [3-9, 34, 36, 37] show that faster data transfers will lead to shorter completion time of a computation stage. In other words, a task only complete faster when all its flows complete faster, and therefore a job stage completion time is shorter when all its tasks in its computation stage complete faster. In this paper, our study focuses on minimizing job computation time (JCT) to complete flows between two successive computation stages.

A study from a large production datacenter in Microsoft [28] points out that job structures come in different shapes, such as "W" shape, tree shape, chain shape, inverted "V" shape and more complex shapes with multiple roots (outputs). Moreover, this study also reports that a job can consist of more than ten stages. The varied attributes of a job structure can result in a job generating different amounts of bytes at different stages. Currently, many of the existing scheduling studies [3-9, 11-14, 34,36, 42-47] do not consider the shape of the job structure and the effect of dependencies between tasks within the same job.

Most current schemes minimize average job (or coflow) completion time by implementing some version of a scheduling scheme based on the total amount of bytes sent. In other words, in these schemes a job is scheduled according to its size, the total bytes sent. These total-bytes-sent (TBS) based schemes compare the total accumulated bytes sent to a set of thresholds to determine which job should be scheduled first. For instance, in [11], a job with fewer bytes sent are scheduled earlier than one having sent more.

However, we observe that scheduling a job simply based on TBS without considering what stage it is in can punish jobs that transmit more bytes in the early stages by scheduling them later for the subsequent stages *even when* these jobs send almost no data in subsequent stages. Moreover, TBS based strategy also punishes jobs that transmit more bytes in some stages but fewer bytes in other stages (on-and-off Job) [19]. Further, conventional TBS based approaches also do not distinguish that a job with many stages but generate small amount of bytes per stage from a job with few stages that generate a large amount of bytes, especially if both jobs generate similar amount of TBS. Therefore, schemes that do not consider stages may result in longer completion time for jobs that include small delay sensitive ones.

Additionally, in some special cases a new computation stage can begin processing without having all tasks from the previous stage complete. This is because some tasks may complete faster than other tasks in the same stage. Thus, the task in the next stage can begin processing as soon as its dependent tasks complete (e.g. a job with multiple parallel

* Published in IEEE ICDCS 2019

chain shape structure). This is also not captured in TBS based schemes.

For these reasons, conventional TBS based schemes do not account for the multi-faceted characteristics of multi-stage job setting. In our study, we identify there are three dimensions to coflow in this setting: the *horizontal* dimension (number of flows per stage), the *vertical* dimension (in this paper the maximum size of flows in each stage, while in [3-9] it refers to the aggregated bytes), and the *depth* dimension (number of computation stages). Previously discussed most TBS based approaches are not sensitive to the nature of depth dimension of multi-stage job structure.

In this paper, we present *Gurita*, a scheduling scheme to coordinate coflows of a multi-stage job that incorporates the multi-dimensional characteristics of multi-stage job, achieving lower average JCT. To design our solution, we first identify the nature of the multi-stage job scheduling problem. Current solutions [3-9, 12-14] generally reduce the coflow scheduling problem to *concurrent open shop problems* (COSP). However, COSP does not accurately describe how jobs are processed in the network (§ III.A). For this reason, we model our job scheduling problem as a *Flexible Flow Shop Multi-Stage jobs Problem* (FFS-MJ), which is rooted in the widely studied *Flexible Flow Shop Problem* (FFS) [30, 32, 33]. The design of our job scheduler leverages wisdom and insights from earlier studies of FFS.

One of the key insights to solving FFS is to obey the classic Johnson's rules [30], that is, system performance can be improved by prioritizing coflows of jobs that are least likely to delay the completion of other jobs. With this insight, we define a set of rules (*Gurita's rules*) to guide our job scheduling design. First, we propose *Least Blocking Effect First* (LBEF) based scheme for the coordination of coflows in multi-stage jobs according to per stage blocking effect, that is, a job's likelihood to delay the completion of another job. In addition, we also incorporate the concept of critical path to further minimize job completion time. To achieve scalability, the scheme is designed without resorting to a centralized controller to avoid high overheads in managing a centralized system. Further, the scheme does not require modification of switch hardware, making the scheme easier to deploy.

To minimize the average JCT, Gurita schedules coflows according to a job's least blocking effect per stage. In other words, the scheduler prioritizes coflows in different job stage that are least likely to delay the completion of other jobs in a given stage. Initially, a job is assigned the highest priority. The priority is then progressively adjusted in each stage according to its impact on other jobs, We note that information on job (e.g. task dependency structure within a job, coflow size, when flows are generated, etc.) is unknown a *priori*, which makes determining coflow stage difficult. To address these challenges, Gurita estimates the job blocking effect by utilizing available information. This includes information received on total bytes sent per stage, number of flows that are currently transmitting data, etc. By doing this, Gurita not only helps small multi-stage jobs complete earlier, it also improves the performance of on-and-off jobs and larger jobs that transmit bytes early, resulting in smaller average JCT. Another advantage is that priority can be adjusted without introducing TCP out-of-order problem.

We implement Gurita in a simulator utilizing real data traces of coflow traffic collected from 3000 machines (150-racks) in Facebook datacenter [4] with two industrial benchmarks: TPC-DS query [5] and Facebook Tao structure [10]. Our result shows that Gurita outperforms a baseline and existing solutions that utilize accumulated total bytes sent, up to 2× and 1.8× faster JCT on average respectively (for smaller jobs 8.5× compared to a baseline (Per Flow Fair Sharing based approach) and 5× faster relative to other existing solutions). At the same time, Gurita achieves performance comparable to that obtained from a centralized solution. For further evaluation, we consider bursty traffic scenario, where jobs arrive in short interval followed by large interval with no arrival. Our experiment shows generally Gurita improves JCT by up to 2× compared to the baseline, while achieving 1.8× improvement relative to existing decentralized solutions. Moreover, Gurita's performance without global view matches the performance of an existing centralized solution with global view.

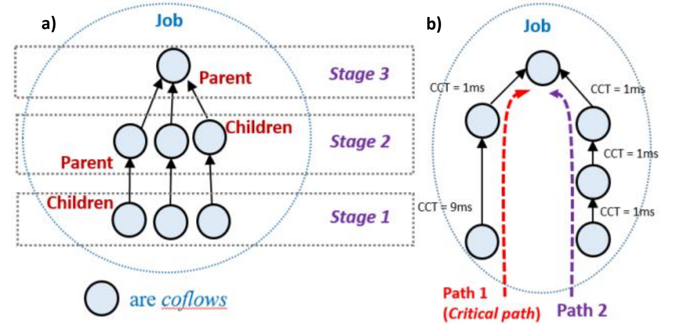

Figure 1. a) An illustration of a three stages job . b) Paths that described the dependency between coflows in a job, the order in which coflows must complete, and CCT of each coflow .

Our contributions. We make the following contributions:

1. We identify the nature of multi-stage job scheduling problem and model the problem as FFS-MJ. We prove this problem to be an NP-Hard problem. We also identify there are multi dimensions to multi-stage job: horizontal, vertical, and depth. This provides an insight into the nature of multi-stage job.
2. We then define Gurita's rules to guide our design and propose multi-stage job scheduling schemes (under both ideal conditions and in practice) without resorting to a centralized controller. Our design also takes advantage of the concept of critical path in a job, which describes a sequence of a job's completion time for every stage, summing up the JCT.
3. We demonstrate the benefits from considering the granularity of job characteristics at different stages in our scheduling through simulation.
4. We address practical challenges using Gurita (e.g. starvation problem) encountered while designing our scheduler.

II. Background, Model, and Motivation

Coflow communication pattern. A coflow is a collection of flows between two groups of machines [15]. In other words,

it is also a group of flows between two set of tasks in two successive computation stages during shuffle phase. Shuffle transfers the output from the previous stage to the next one. The machines that send the outputs are called *senders* and the machines receiving the data are called *receivers*. In a coflow, each receiver communicates with one or multiple senders to complete a single coflow [14]. In multi-stage scenario, a sender may function as a receiver when the sender invokes new (children) senders, such that the parent can only be processed after all of its children complete.

DAG structure. Dependencies between coflow in a multi-stage job can be modelled as a Directed Acyclic Graph (DAG) [5,27]. A parent coflow only completes when all coflows it depends on complete. The relationship between coflows of the same job can be described as follows. Represent each job as DAG $G = (E, V)$, where a vertex in V is a coflow and edge $(u, v) \in E$ represents a dependency between two coflows u and v in V, where u's completion depends upon v's completion (Figure 1.a).

Computation stages. A stage is a computation step in G, such that i^{th} stage is the ith computation step and i^{th} stage must be completed before $(i+1)^{th}$ stage can be processed. Further, a job may have one or more coflows in a stage [5].

Jobs in production. As observed in Microsoft datacenter [28], a job is typically made up of multiple tasks, and job dependency structure may come in different shapes, such as "W" shape, tree shape, chain shape, inverted "V" shape or more complex shapes with multiple outputs (roots). Approximately 40% of jobs exhibit a tree structure. Additionally, a job may have multiple parallel chains of dependencies and the average depth of a job is five stages and may go to more than ten stages. This observation offers an explanation in [10, 15, 19] as to why a job may generate different amount of bytes at different stage.

Settings. To analyze the job characteristic, we abstract the datacenter network as a non-blocking datacenter fabric connecting two sets of M machines [4, 5], where the ingress machines (senders) generate data flow and egress machines (receivers) are the destination. This abstraction allows a simpler conception for analysis. However, we do not impose this concept in our design and evaluation. In our design and evaluation, we consider the more realistic scenario where the network in datacenter can be bursty and jobs compete for network resources (e.g. switch) [17, 23].

Motivation example (Figure 2). Consider job A transmitting 10, 1, 1, and 1 units size of data at stage $\mathcal{S}$, $\mathcal{S}+1$, $\mathcal{S}+2$ and $\mathcal{S}+3$ respectively. We have single stage job B, C, and D each transmitting 2 units size data. The processing rate is 1 unit size per unit time. In the first scenario, these jobs are scheduled using TBS based scheme which prioritizes jobs with less TBS. We have job A complete after $\mathcal{S}+3$ with JCT = 19 units time. Job B, C, and D each has JCT = 2 units time. The average JCT is $\frac{19+2+2+2}{4} = 6.25$ units time. In the second scenario, the scheduler prioritizes jobs according to how much data is transmitted per stage instead of TBS. Job A completes with JCT = 13 units time and Job B, C, and D each has JCT = 3 units time. The average JCT in the second scenario is $\frac{13+3+3+3}{4} = 5.5$ unit time, which is lesser than the JCT in the first scenario.

III. Multi-Stage Coflow

Gurita coordinates jobs according to how a job may impact (or delay) the completion of other coflows. Like prior works [3-13], Gurita assumes that coflow IDs can be obtained from upper layer applications. However, CODA [12] shows that it is possible to infer flows of a coflow using machine learning. Gurita's scheduler resides in one of the many receivers of a coflow and coordinates all the receivers in order to manage the flows in each receiver. Finally, scheduling decisions are enforced in the network by employing a built-in function commonly available in today's commodity switches, namely strict priority queuing (SPQ), which means Gurita is deployable friendly.

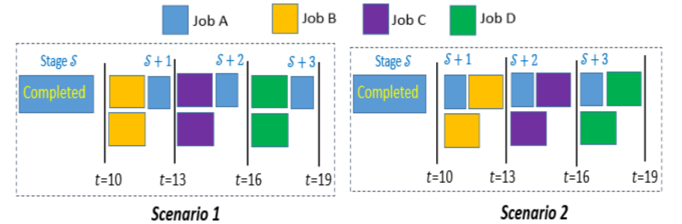

Figure 2. Illustration on the disadvantages of stage agnostic scheme on JCT. Scenario 1 demonstrates the results of solution based on Shortest Job First. Scenario 2 demonstrates the results of scheduling solution that takes into account the multi-stage attribute of jobs.

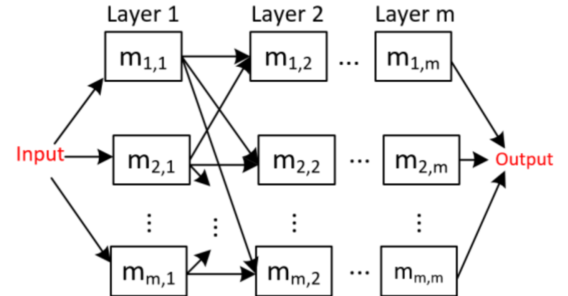

Figure 3. The illustration of modeling topology of datacenter network in Flexible Flow Shop Problem. Machine $m_{i,j}$ is the jth machine in layer i.

A. Problem Formulation

Consider the following multi-stage job scheduling problem with n jobs in a system indexed by 1, 2, …,n. Let T_J denotes the JCT of job J; the problem is formulated as follows.

$$minimize \sum_{j=1}^{n} T_J \quad (1)$$

$$s.t. \quad C(\mathcal{S}) \succ C(\mathcal{S}+1), \quad \text{for } \mathcal{S} > 0, \quad (1.a)$$

where $C(\mathcal{S})$ denotes coflow C in computation stage $\mathcal{S}$ of job J. Constraint (1.a) describes the dependency between two coflows in J such that the completion of $C(\mathcal{S}+1)$ depends on the completion of $C(\mathcal{S})$. This ensures that the parent coflow is processed only after its child flows are complete.

Unlike a single-stage job scenario, where JCT is determined by its coflow with the slowest flow completion time, JCT of a multi-stage job is influenced by the number of stages in a job and JCTs in each stage. Let $\Phi_{SET}(J)$ denote a set of paths in job J, where each path $\Phi_i \in \Phi_{SET}(J)$ describes the order of tasks in which they must be completed from leaf

nodes (tasks in stage 1) to root nodes (tasks in final stage) in a job, as illustrated in Figure 1. Let $CT(\Phi_i)$ denote the total time required to deliver data between tasks in path Φ_i. Thus, minimizing JCT is equivalent to

$$JCT = min\ max\big(CT(\Phi_i),\ \forall\Phi_i \in \Phi_{SET}(C)\big).$$

In other words, the concept of critical path is an expression that consists of a sequence of coflow completion time (CCT) that covers the JCT. It also offers a way to relate CCT at different stages from JCT perspective. Therefore, when the CCT of a coflow on critical path is increased, the job completion time is also increased. Based on this observation, coflows on critical paths of different jobs should not be scheduled together. This insight is incorporated in our design to solve problem (1).

To design an appropriate scheduling scheme, we need to understand the nature of the problem of scheduling jobs. Initial studies on coflow scheduling [4, 9] reduce the coflow scheduling problem to an NP-Hard problem - *concurrent open shop problem* (COSP) [2] - consisting of n jobs and m layers of machines, where jobs correspond to coflows and machines can be interpreted as egress ports of datacenter fabric (or switches). The objective is to minimize the total job completion time. Since solving COSP is very difficult [2, 7], the current practice [4, 5, 9,12, 34] simplifies COSP to 2 layers ($m \times 2$ datacenter fabrics). In the simplification, a datacenter fabric is assumed to be a non-blocking big switch, where each machine in COSP represents NIC cards at the end-host (sender and receiver ends). Thus, this setup allows coflow scheduler to be concerned only with scheduling a set of flows of a coflow at the senders' end such that the waiting time for resources at the receivers' end is minimized.

However, delving deeper into the nature of COSP, we realize that it is not concerned with the order in which jobs are processed first [2]. For example, a flow can be first processed at a receiver, and then at the sender, which is not the order in which a flow is processed in the network. The processing order in the network should be: the flow is processed at the sender before it is processed at the receiver. In other words, COSP allows flow operation to be processed at a random order. This is because COSP is formulated for scheduling problem in manufacturing where products are processed in multiple assembly lines and can be moved between assembly lines. For this reason, although there have been numerous of attempts to solve COSP [31, 32], the insights gained from these endeavors do not apply to job scheduling.

Therefore, in order to design an effective multi-stage job scheduling scheme, we first must understand the nature of the job scheduling problem and identify what existing problem the multi-stage job scheduling problem should be reduced to. We identify a class of *Flexible Flow Shop Problem* (FFS) [30, 32, 33] that not only captures the characteristics of scheduling jobs, but also reflects network processing order, i.e. each operation (how coflow is processed) must be performed in a sequential order according to the order of the sender and then the receiver. The objective is to minimize the average JCT, while considering the constraints of the order of when coflows can be processed at different machines. Since flow shop scheduling has been widely studied in operation research, reducing job scheduling problem to this problem allows us to naturally exploit the wealth of insights and lessons learned from existing solutions.

B. Flexible Flow Shop Problem for Multi-Stage Jobs

FFS only considers the scenario of a single stage job. Thus, we extend the FFS problem to *Flexible Flow Shop Problem with Multi-stage Jobs Problem* (FFS-MJ). In this section, we first formally define FFS-MJ. Then, we convert intra multi-stage job scheduling to FFS-MJ.

Definition 1: *Flexible Flow Shop Problem with Multi-stage Jobs Problem* (FFS-MJ). Consider a set of n independent jobs $J_1, J_2, \dots, J_n$, where each Job J_i consists of a set of coflows. These jobs have to be scheduled through multiple layers of processing, where each layer is made up of parallel machines (Figure 3).

We first define the relationship between two coflows. Let $C(\mathcal{S})$ denote coflow $C \in J_i$ in computation stage $\mathcal{S}$. If coflow $C(\mathcal{S}+1)$ is the parent of $C(\mathcal{S})$, then, there is a precedence constraint requiring the completion time of $C(\mathcal{S})$ before its parent coflow $C(\mathcal{S}+1)$. In other words, there is a dependency between coflows in J_i that forms DAG such that a parent task can only be processed after all its children complete. The relationship between parent and child can be interpreted as that of coflows each belonging to different sequential computation stages.

Next, we define how a coflow is processed. Every coflow $C(\mathcal{S}) \in J_i$ is processed in a set of parallel operations $O\big(C(\mathcal{S})\big)$. Namely, $O\big(C(\mathcal{S})\big)$ can be interpreted as a set of operations processing a set of flows of a coflow C at stage $\mathcal{S}$. Thus, $O\big(C(\mathcal{S}+1)\big)$ cannot be processed until operations in $O\big(C(\mathcal{S})\big)$ complete.

Here, we define how a machine processes an operation in $O\big(C(\mathcal{S})\big)$. Each machine, which can be interpreted as a port in the big switch, can only process at most one operation at a time, such that the $\big(C(\mathcal{S})\big)$ operation must be processed at one of the machines in layer $\mathcal{S}$.

Finally, the objective is to arrange the job sequence such that the aggregated job completion time is minimized. ■

NP-Hard problem. We show the hardness of the FFS-MJ.

Theorem 1. FFS-MJ is NP-Hard.

Proof. Let $H(\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_m)$ be FFS-MJ problem and problem H has m stages to complete, such that stage $\mathcal{S}_i$ must be completed before $\mathcal{S}_{i+1}$ can be processed, for $1 \le i < m$. We partition problem H into smaller problems $H(\mathcal{S}_1)$, $H(\mathcal{S}_2)$, …, and $H(\mathcal{S}_m)$, where these problems are solved individually. However, they must be processed one after another in the sequence according to the order of its stage, such that $H(\mathcal{S}_{i+1})$ is solved only after $H(\mathcal{S}_i)$ completes, for $i = 1, 2, \dots, m-1$. Since solving each $H(\mathcal{S}_i)$ is NP-Hard [2, 31, 32], $H(\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_m)$ is also NP-Hard. ■

Theorem 1 implies that solving multi-stage job scheduling problem is equivalent to solving FFS-MJ, which is also an NP-hard problem.

Reducing multi-stage job problem to FFS-MJ. In Definition 1, we have described how multi-stage job is mapped to FFS-MJ. To reduce datacenter network to multiple layers of parallel machines in FFS-MJ, conceptually machines

in m^{th} and m-1th layer in FFS-MJ can be viewed as receivers and senders respectively in the big switch abstraction, for $1 < m < M$. M denotes total number of machines.

Identifying the nature of multi-stage job scheduling problem provides us with a direction and intuition to solve the problem. However, intuition alone is not sufficient. In order to properly design a scheduler, we must also understand the job and coflow characteristics in multi-stage jobs setting.

C. Characteristics in Multi-Stage Job

In this section, we investigate job characteristics in a multi-stage setting by analyzing data collected from a production datacenter in Facebook [4, 5] and make the following observations.

(i) Flows at the leaf node position of any job structures are the first flows to be processed.

(ii) A receiver may have a large number of parallel senders transmitting data from different ports simultaneously. This means the aggregate traffic from this group of flows may quickly create bottleneck and increase completion times of other jobs.

(iii) The time to complete a single stage is determined by the amount of bytes sent is processed and the processing speed in each stage. Thus, the larger numbers of bytes sent per stage or the lower processing speed can lead to longer completion times. Based on these observations, a coflow in this setting has these following dimensions: *horizontal* (the width of coflow per stage), *vertical* (the largest flow size in a coflow per stage), and *depth* (the number of stages that needs to be completed). This revelation provides us with an important insight for designing our scheduling scheme.

IV. GURITA'S RULES AND SCHEDULING SCHEME

In this section, we present our *Gurita* multi-stage job scheduling scheduler. We begin with rules that provide the foundations of our solution. We then design our scheduler using these rules and observations made in the previous section.

A. Gurita's Rules

Since FFS has been widely studied and FFS-MJ is rooted in FFS, reducing multi-stage job scheduling to FFS-MJ allows us to design a scheduler that leverages insights observed from existing studies on FFS [30, 31, 32]. Most solutions for FFS obey the classic Johnson rules [30], which are: (i) minimize resource idle time, (ii) make the machine available quickly to reduce waiting time for resources is minimized, (iii) avoid blocking other jobs, and (iv) avoid tardiness. Here, tardiness of a job is comparable to the amount of time that elapses between when a job is supposed to complete (e.g., due date) and when it actually completes.

To illustrate the impact of the blocking concept proposed by Johnson (Figure 4), consider a single stage job A, B, C, and D each transmitting data 6 units size. Job A has three coflows each consist of 2 units size, while job B, C, and D have two coflows each of 3 units size. Thus, all jobs have the same total size. The processing rate is 1 unit size per unit time. In the first scenario, job A has three coflows blocking coflows from job B, C, and D. Job A has JCT = 2 units time, while job B, C, and D have JCT = 5 unit times. The average JCT is 4.25 units time. On the other hand, in the second scenario, if Job B, C, and D are prioritized over job A, then we have job B, C, and D each incurs JCT = 3 units time and job A incurs JCT = 5 unit times. The average JCT = 3.50 units time, which is lower than the JCT in the first scenario.

Next, we describe how insights observed from Johnson's rules and our interpretations for *Gurita*.

(i) Since a multi-stage job is processed through multiple layers of machines (e.g. server nodes), a longer processing time in a layer can lead to longer idle times of machines in the next layer. Based on Johnson's first rule, the key to reducing machine idle time is to quickly complete the job's stage in predecessor machines. The time to process at each layer can be sped up by prioritizing a job's stage that consist of small number of small size flows because they can be processed quickly.

(ii) Further, consistent with Johnson's second rule, prioritizing a stage consisting a small number of small size flows means machines can be made available quickly.

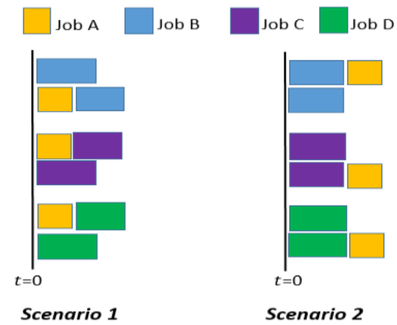

Figure 4. Example of blocking Impact.

(iii) We observe that a job's stage may block other job(s) vertically or horizontally. Vertical blocking is caused by a set of elephant flows in a job stage causing another job stage to wait longer for elephant flows to complete. Horizontal blocking is caused by a job stage consisting of a large number of concurrent flows that require more resources, resulting in the blocking of another job stage. Finally, the worst case scenario is a combination of horizontal and vertical blocking. Thus, in accordance to Johnson's third rule, a job's stage that block a other job stage (either horizontal, vertical, or both) should be assigned lower priority.

(iv) Last, as pointed in Johnson's final rule, tardiness can be avoided by prioritizing a job stage with the smallest slack (e.g., slack = time remaining before due date - remaining processing time). This means a job that has reached the final stage of its completion should be quickly completed to further minimize a job completion time and to avoid causing delays to other jobs.

Based on these applications of Johnson's rules, we propose *Gurita's rules*.

Rule 1. To avoid machine waiting for jobs and jobs waiting for a resource's availability, the scheduler should prioritize job stages that consists of smaller numbers of shorter flows.

Rule 2. To avoid horizontal blocking, the scheduler prioritizes job's stage that consist of smaller number of flows. To avoid vertical blocking, scheduler prioritizes job stage that consist of short flows.

Rule 3. Jobs in the final stage should be prioritized over those that are not in final stage.

Rule 4. Based on our observations discussed earlier, blocking coflows on a critical path increases job completion time. Therefore, coflows on critical path should be prioritized over those that not on critical path.

B. Coflow Scheduler

Here, we design our scheduler by considering *two scenarios: ideal* condition and *in practice*. In the ideal condition scenario, the scheduler is assumed to operate with all information on coflow and job structure available ahead of time (e.g. flow size, coflow arrival time, number of flows, number of stage, etc.). This assumption allows us to holistically apply the Gurita rules and insights to the initial design. Following this, we adjust the design taking into account more realistic conditions in practice, where *some information is not available ahead of time*.

We begin by presenting the formulation of the coflow blocking effect Ψ_J^C of coflow C in job J, and then we formulate job blocking effects $\Psi_J(\mathcal{S})$ at stage $\mathcal{S}$ with Ψ_J^C. First, Ψ_J^C is formulated as follows.

$$\Psi_J^C = \beta \times w^C \times \ f_{max}^C \times \mathbb{F}^C, \quad (2)$$

Here, β denotes a weight factor for a coflow that reaches final stage (Gurita's 3rd rule), where $\beta = 1 - \frac{\hat{\mathcal{S}}}{total_\mathcal{S}}$ and $\hat{\mathcal{S}}$ denote the number of completed stages and $total_\mathcal{S}$ describes the total number of stages required to complete a job. β decreases as the job's stage approaches the final stage. Moreover, the two dimensions of coflow, f_{max}^C and $\mathbb{F}^C$ denote the size of the largest flow and the total number of flows in coflow C respectively. We have $f_{max}^C \times \mathbb{F}^C$ to model the horizontal and vertical blocking effect per stage (Gurita's 2th rule). The area under $f_{max}^C \times \mathbb{F}^C$ approximates the severity of blocking effect with both dimensions combined. Additionally, since the blocking time duration is affected by flow size (Gurita's 1th rule), Ψ_J^C is adjusted with w^C, which is expressed as follows.

$$w^C = \begin{cases} 1-\gamma, & \gamma < 1 \\ 0.1, & otherwise \end{cases},$$

for $\gamma = \delta \frac{\overline{\mathbb{F}}^C}{f_{max}^C}$, where $1 > \delta > 0$ is constant and $\overline{\mathbb{F}}^C$ denotes the average flow size in a coflow. Here, f_{max}^C can be interpreted as the worst case scenario and w^C normalizes the blocking effect of f_{max}^C relative to other flows in C. Thus, if f_{max}^C is large and $\gamma \rightarrow 1$, w^C means a coflow may further delay the completion of other coflows from different jobs.

Next, per stage job blocking effect $\Psi_J(\mathcal{S})$ with Ψ_J^C can be formulated as follows. First, we consider the case when there is only a coflow in stage $\mathcal{S}$, we have $\Psi_J(\mathcal{S}) = \Psi_J^C$ for $C \in J(\mathcal{S})$. Otherwise,

$$\Psi_J(\mathcal{S}) = max(\Psi_J^C), \forall C \in J(\mathcal{S}),$$

for $|C| > 1$. Then, at the abstract level, $\Psi_J(\mathcal{S})$ is utilized to coordinate coflows such that coflows with smallest value of $\Psi_J(\mathcal{S})$ are scheduled ahead of coflow with a higher blocking effect.

To satisfy Gurita's 4th rule, we first approximate the $CCT \approx \frac{f_{max}^C}{B}$ of each coflow and then we assign it to each vertex in DAG G discussed above respectively. Here, B denotes the processing rate. Then, the critical path in a job can be determined using Breath First Search [24].

However, in practice, information on job such as flow size, coflow size, and job structure is usually unknown a priori, which makes determining Ψ_J^C and satisfying Gurita's 4th rule difficult. To address this problem, we demonstrate in the following section how we adjust the initial design of per stage blocking effect based on available information on job without resorting to a centralized controller.

From concept to practice. The unavailability of information makes scheduling coflows of different jobs challenging. Another challenge is enforcing the scheduling scheme without requiring the scheme to resort to a centralized controller and modifications to existing hardware (e.g., switches). First, we describe how to estimate $\Psi(C_\mathcal{S})$ when information on job is unknown a priori. Gurita addresses this problem by estimating Ψ_J^C with information that becomes available at the receiver end, such as number of open connections between senders, amount of bytes received by each flow, and the aggregated number of bytes received of a coflow. Here, this information can be transparent at the shim-layer between TCP/IP stack (or VMs) and the link-layer (or Hypervisor) by leveraging the NetFilter framework [22], which is an integral part of Linux OS. Netfilter hooks are attached to the data path in the Linux kernel just above the physical interface, allowing Gurita to intercept outgoing and incoming packets without modifying either the TCP/IP stack of the host or guest VM's operating system. The interception is performed before packets are pushed down to the TCP/IP stack for further processing (i.e., at the pre-routing hook).

Algorithm 1: *Least-Blocking Effect First* (LBEF)

```
1.  J̄   // set of multi-stage Jobs
2.  Procedure Gurita_scheduler ( J̄ )
3.    J̈  // empty array to keep track Jobs
4.    for each coflow C ∈ J ∈ J̄ do
5.        C(S) = stage( J )  // The stage of cofow C
6.        Ψ_J^C = compute_blocking_effect ( C(S) )
7.        Compute Ψ_J(S)
8.        J̈ ← Ψ_J(S)      // insert Ψ_J(S) to J̈
9.    end for
10.   Sort(J̈)  ⊳ Sort in descending order according to Ψ_J(S)
11.   for each C(S) ∈ J̈ do
12.     Process ∀f ∈ C(S) // Process all flow in C(S)
13.   end for
14. end procedure
15. Procedure compute_blocking_effect (C(S))
16.   Utilizing eq. (3) to compute and return Ψ_J^C
17. end procedure
```

Here, Gurita utilizes a designated receiver to collect the information on number of open connections and bytes received by each flow locally and from its peers (receivers of the same job) to estimate f_{max}^C, $\mathbb{F}^C$, and w^C. The details of designated receiver will be discussed later in this paper. Then, Gurita estimates β without prior knowledge of job structure by leveraging the number of completed stage $\hat{\mathcal{S}}$. For instance, $\beta \approx \frac{1}{log(\hat{\mathcal{S}}+1)}$. The β influence diminishes as $\hat{\mathcal{S}} \rightarrow \infty$ to

prevent false positive of nearing final stage caused by job with many stages. Information on $\hat{\mathcal{S}}$ can be obtained through the master controller (e.g. Map stage and Reduce stage), but there are cases when obtaining $\hat{\mathcal{S}}$ is not obvious [28]. One way to address this issue is by Gurita utilizing a controller to keep track of the job's stage when coflow registers through an API [4, 5], or by utilizing machine learning [12].

Based on the above discussion, Ψ_J^C is estimated by

$$\Psi_J^C \approx \ddot{\beta} \times \ddot{w}^C \times \ddot{f}_{max}^C \times \ddot{\mathbb{F}}^C \quad (3)$$

where $\ddot{\beta}$, $\ddot{w}^C$, $\ddot{f}_{max}^C$, and $\ddot{\mathbb{F}}^C$ are approximation of β, w^C, f_{max}^C, $\mathbb{F}^C$, and w^C respectively. Coflows in stage $\mathcal{S}$ is scheduled according to $\Psi_J(\mathcal{S})$, such that coflows with lower $\Psi_J(\mathcal{S})$ receive higher priority.

To satisfy Gurita's 4th rule without job structure information available ahead of time and without using a central controller, we first make the following observations. A critical path usually either has coflows with high CCT, a long chain of coflow dependencies, or a combination of the two. Moreover, prioritizing coflows on critical path with strong blocking effect may increase the JCT of other jobs. At the same time, prioritizing coflows on critical path with the least blocking effect may not be advantageous because Gurita's 2th rule guarantees its prioritization. Interestingly, we observe from our experiments that prioritizing coflows on critical path with marginally larger blocking effect than coflows with the least effect may benefit from Gurita's 4th rule (§V). It can lower job JCT without significantly delaying other jobs. At last, we also notice that the number of coflows on critical path per job's stage is bounded by the number of critical paths per job. Based on these observations, we extend eq.(3) as follows.

$$\Psi_J^C \approx \ddot{\beta} \times \ddot{w}^C \times \ddot{f}_{max}^C \times \ddot{\mathbb{F}}^C - \left(\alpha \, f_{max}^C \times z_{\mathcal{S}}\right),$$

where $z_{\mathcal{S}} \in \{0,1\}$, such that $z_{\mathcal{S}} = 1$ if a coflow is possibly on a critical path, and $0 < \alpha \leq 1$ denotes a constant variable. Additionally, Gurita leverages $\ddot{f}_{max}^C$ to estimate coflow on critical path because CCT is influenced by f_{max}^C. Since job structure information is unknown a priori, f_{max}^C behaves like a random variable. Then, Gurita utilizes *Average Value Approximation* technique (AVA) [38] to estimate whether a coflow is on a critical path. It is a technique that is often used in performance modeling to replace random variable by its means. Gurita computes the average of largest N observed f_{max}^C using AVA. In our experiment, we have $N < 5$, where 5 is the average number of stages in a job in production [28]. The AVA may not be as precise but suffices to lower the JCT.

The scheduling scheme. Gurita schedules coflows according to their per stage blocking effect $\Psi_J(\mathcal{S})$ using *Least- Blocking effect First* (LBEF) (Algorithm 1). The general idea is that, given the stage of jobs, the scheme gives preference to coflows with the smallest per stage blocking effect $\Psi_J(\mathcal{S})$. In other words, coflows of job $J(\mathcal{S})$ with lower $\Psi_J(\mathcal{S})$ are assigned higher priority. This can be interpreted as follows. A coflow that blocks other coflows vertically or horizontally is given lower priority, but the priority depends on how many coflows are blocked. However, since a job may have different blocking effect at different stages, Gurita adjusts each coflow's priority assignment whenever a coflow starts a new stage. Moreover, we also consider the scenario where different coflows may be in different stages. Thus, $\Psi_J(\mathcal{S})$ is updated when new coflows begin and complete, and the priority is also adjusted.

Gurita assigns priority at (i) job level and (ii) coflow level. At the job level, Gurita compares $\Psi_J(\mathcal{S})$ to a demotion threshold $\mathcal{T}$: if $\Psi_J(\mathcal{S})$ exceeds $\mathcal{T}$, then the coflows in job $J(\mathcal{S})$ will be deprioritized, resulting in lower priorities for all its coflows. In our design, each threshold is associated with a priority level. Details of priority decision are discussed later in this paper. At the coflow level, every newly generated flow is also initially assigned to the highest priority by its receiver. This is because job information is not known a priori. A newly arrived coflow of job $J(\mathcal{S})$ is deprioritized according to the following conditions: First, when the coflow's blocking effect Ψ_J^C exceeds the highest priority threshold of job blocking effect, then the flows of this coflow are assigned to the priority previously assigned to job $J(\mathcal{S})$. Secondly, when the job itself is deprioritized, then these flows are assigned to the job's new priority. This strategy allows Gurita to increase the priority of a job while avoiding TCP out of order problem [24]. In other words, only newly generated flows are affected when a job is assigned to priority, while flows that are generated earlier continue to transmit at the previously assigned priority. Gurita employs a flow hash table (e.g. Jenkins hash [29]) to keep track of flow information at the receiver's end using 5 tuples (i.e., src IP, dest IP, src port, dest port, and protocol) to identify different flows. Gurita then updates and stores flow information (i.e., coflow ID, flow ID, byte received counts, number of open connections, and etc.) into a flow table.

Next, we describe how Gurita enforces LBEF without requiring hardware modification. The scheduling policy is then enforced in network by utilizing strict priority queuing (SPQ) [24], a built-in feature in existing commodity switches that utilizes multiple queues [1]. This enables packets belonging to higher priority coflows to be processed ahead of lower priority coflows. By exploiting SPQ to locally govern inter coflows traffic, Gurita achieves approximate global coordination. In other words, SPQ allows Gurita to function similar to traffic road management, where traffic lights locally govern road intersections, corresponding to commodity switches in our case. As long as all parties on the street abide by the traffic lights, which is the scheduling policy in this analogy, approximate global coordination occurs, resulting in smooth traffic.

In practice, bottleneck may occur in datacenter network due to its bursty nature [17,23,40]. Leveraging SPQ to enforce the scheduling policy in the switches naturally allows Gurita to extend the description of machines in FFS-MJ (Definition 1) to also include switches. The egress ports of these switches function similarly to the machines in FFS-MJ such that jobs are processed in sequential order according to the order of switches in the path that connects senders and receivers, where data traverses from senders to receivers. This demonstrates that FFS-MJ is a proper problem to reduce to.

Moreover, given SPQ is a built-in feature in existing commodity switches, this makes Gurita deployable friendly.

The next discussion describes the details of how Gurita leverages SPQ to coordinate coflows, as well as how to address the drawback of SPQ.

Job and coflow prioritization. Consider K priority queues in commodity switches [1] and job's stage $J(\mathcal{S})$, priority P_J^k denotes k^{th} priority queue assigned to coflows in $J(\mathcal{S})$, such that $0 \le k \le K$. Then, the priority arrangement is defined as follows: $P_J^0 > P_J^1 > \cdots > P_J^k > \cdots > P_J^K$, where P_J^0 is the highest priority and P_J^K is the lowest priority. Every P_J^k is associated to threshold $\mathcal{T}_k$, where $\mathcal{T}_0 < \mathcal{T}_1 < \cdots < \mathcal{T}_k <..< \mathcal{T}_K$. Currently, existing commodity switches typically support 8 priority queues [1]. The priority decision is determined by comparing $\Psi_J(\mathcal{S})$ to a set of thresholds. Initially, coflow $C \in J(\mathcal{S})$ is assigned to P_J^0, such that all flows in C is also assigned to P_J^0.

All coflows in $J(\mathcal{S})$ are demoted to lower priority when the estimated blocking effect $\Psi_J(\mathcal{S})$ exceeds threshold $\mathcal{T}_k$, for $0 \le k \le K$. In other words, in stage $\mathcal{S}$, coflow $C \in J(\mathcal{S})$ transmits data at priority P_J^k when $\mathcal{T}_{k-1} < \Psi_J(\mathcal{S}) \le \mathcal{T}_k$ is satisfied. Otherwise, job J is demoted to priority P_J^{k+1}. When $\Psi_J(\mathcal{S}) > \mathcal{T}_K$, job $J(\mathcal{S})$ will be assigned to priority P_J^K. When stage $\mathcal{S}$ completes, Gurita adjusts the priority assignment of job $J(\mathcal{S}+1)$ according to $\Psi_J(\mathcal{S}+1)$. These thresholds are determined using exponentially-spaced as recommended by [5]. As part of our future work, we will extend the study in [35] on using machine learning to determine thresholds.

Priority decision. Job priority is determined by a head receiver (HR). HR is the first receiver of invoked in a coflow. Other receivers learn about HR (e.g. HR's IP address) from the "master" (or "manager" or "coordinator") [18,19, 20, 21] if these receivers are invoked by the master. If new receiver are invoked by existing receiver, then parent receivers inform their children about HR. The HR then determines job priority using eq. (3) and information collected from other receivers of the same job. Once HR communicates the priority decision to the receiver, the receiver informs its senders using reserved field in the TCP header of ACK packet about the decision. Then, the senders sets DSCP bits in the IP header of their outgoing packets accordingly. Here, Gurita uses DSCP to communicate priority decision and to schedule coflows.

Next, receivers provide updates on locally-observed information (e.g. byte received at the receiver's end, etc.) to HR at regular interval of δ unit time. Information such as number of flows is determined by counting the number of open connections. Therefore, the HR utilizes information collected from its peers to determine job priority, and HR communicates the decision to other receivers through update messages. Upon receiving updates from HR, receivers compare the new priorities with the old ones. If new priorities are lower than the old ones, then receivers update their respective flows to transmit data according to the new priorities. Otherwise, flows continue the transmitting using the old priorities.

Some coflows of a job are too small to wait for decisions from HR. Therefore, newly-arriving flows of coflow are automatically assigned the highest priority and are allowed to transmit data at that priority until a threshold is exceeded or an update is received from HR. Last, when a receiver completes its task (all senders close their connections to their receiver), the receiver informs their HR, and the HR excludes information of completed flows from being considered to determine job blocking effect, and HR updates $\Psi_J(\mathcal{S})$.

Starvation Mitigation. As pointed in [14, 25], SPQ based schedulers often introduce starvation, where low priority traffic is denied resources. To alleviate the starvation problem, we emulate SPQ by mimicking the behavior of SPQ using Weight Round Robin (WRR) [24]. This allows lower priority traffic to transmit data at much lower rate than higher priority traffic. Emulation is achieved by using per queue waiting time in SPQ scenario to determine the weight of each queue in WRR scenario, which is described as follows. Given a link of capacity B (i.e. bandwidth), the traffic load in priority queue 0 at each link is $\rho_0 = \frac{\lambda_0}{B}$, where λ_0 denotes arrival rate at priority queue 0. Information on arrival rate is generally available and can be retrieved from switches [1]. Then, the average waiting time W_0 in priority queue 0 under SPQ is $W_0 = \frac{B}{1-\rho_0}$ [36] Given k priority queues, the average wait time at the kth priority queue is

$$W_k = \frac{B}{(1-\rho_0-\cdots-\rho_{k-1})(1-\rho_0-\cdots-\rho_k)}.$$

Next, we utilize average waiting time W_k to determine the weight of ith queue in WRR scheme ω_i as follows.

$$\omega_i = \frac{W_i}{\sum_{j=0}^{k} W_j},$$

for $i = 0, 1, \ldots, k$ and $\sum_{i=0}^{k} \omega_i = 1$. After that, the rate allocated to i^{th} queue in WRR scenario is determined by $\omega_i * B$. This completes SPQ emulation using WRR, which allows Gurita to resolve starvation in SPQ.

V. EVALUATION

In this section, we evaluate the performance of Gurita through large scale simulation using data trace collected from Facebook datacenter. Our primary metrics for comparison is the average CCTs and performance improvement factor, which is described as follows.

$$Improvement = \frac{Compared\ JCTs}{Gurita's\ JCTs}$$

If the improvement is greater (smaller) than one, Gurita is faster (slower). Additionally, performance is evaluated in seven different categories of job size as described in table 1. We compare Gurita to existing solutions that do not rely on a central controller, as well a solution that requires one.

The main results are summarized below:

1. Our experiments show that Gurita outperforms the baseline and decentralize solutions (Baraat [3] and Stream [4]) by up to 2× and 1.8× faster on average respectively. Relative to the centralized solution with global view, Aalo [5], Gurita is able to match the performance without complete information.
2. In Bursty traffic scenario in large scale network, Gurita achieves a faster average JTT with small jobs (job in category 1 of table 1) by up to 2× and 1.7× compared to

the baseline and decentralize solutions. Relative to Aalo, Gurita also achieves similar performance.

3. In comparison to GuritaPlus, a scheduler with coflow information available ahead of time, Gurita achieves comparable performance.

Trace Driven Simulations

In this section, we analyze the performance of Gurita through simulation experiments.

Simulation setting: We develop a flow-level simulator and it accounts for the flow arrival and departure events, rather than packet sending and receiving events. It updates the rate and the remaining volume of each flow when event occurs. Our simulations employ 8 pods FatTree network topology [26] (Figure 4.c) with 128 servers and 80 switches. Here, we utilize 10 Gigabit (10G) switches in our evaluation.

In our simulations, we compare Gurita's performance to Per Flow Fair Sharing, Baraat [3], Stream [14], and Aalo [5]. *Per-Flow-Fair-Sharing* (PFS) mechanism is a scheduling scheme that divides the resource capacity equally among flows traversing the same link, which is also the baseline in our analysis. *Baraat*, a FIFO with limited multiplexing (FIFO-LM) scheduler, is the current state of the art decentralized scheduler. *Stream*, another decentralized scheduling scheme, leverages coflow communication pattern to schedule coflows.

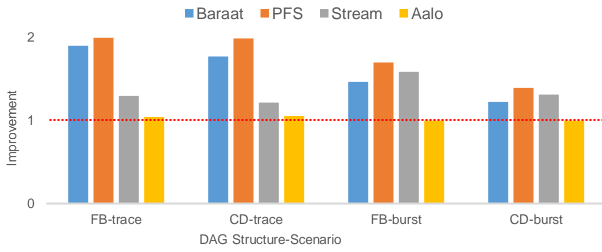

Figure 5. Average performance in a scenario utilizing production trace with Facebook (FB-t) and Cloudera (CD-t) structure and in bursty scenario Facebook (FB-b) and Cloudera (CD-b).

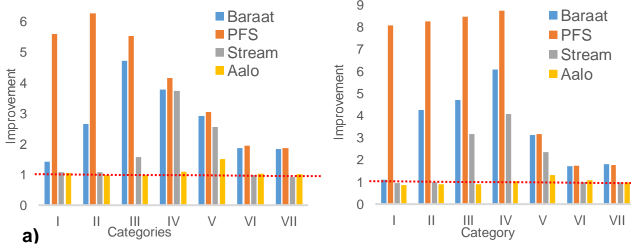

Figure 6. Average JCT in seven categories from replying production data trace with FB-Tao (Fig. 6.a) and TPC-DS (Fig. 6.b) structure.

I	II	III	IV	V	VI	VII
6MB-80MB	81MB-800MB	801MB-8GB	8GB-10GB	10GB-100GB	100GB-1TB	> 1TB

Table 1. Seven categories of multi-stage job size.

To analyze how Gurita performs against centralized solution, we compare our solution to *Aalo*. For simplicity, Aalo's additional delay from managing centralized system is not considered in the simulator and information on job is made available instantaneously to the centralized controller. Additionally, we employ four priority queues in Aalo, Stream, and Gurita, sufficient to provide the satisfactory outcomes according to findings in [5, 14]. In principle, all schemes assume that job characteristics are unknown ahead of time. Moreover, in our simulation, the dependency between coflows in a job is detected when one of the sender sends requests to a set of senders (or a sender) for data.

Traffic pattern and load. To evaluate Gurita's performance, we use production trace collected from 150-racks (3000 machines) in Facebook datacenter. Then, we further evaluate Gurita in bursty scenario, which is when jobs arrive within small time intervals, a common occurrence in datacenter [17]. Facebook trace does not provide details at flows, coflows, or jobs level, including information on job structure. For example, the data trace does not specify the relationship between coflows. Therefore, we utilize industrial benchmark Cloudera Industrial benchmark, TPC-DS query-42 (TPC-DS) [4], and Facebook Tao structure (FB-Tao) [10] to generate DAG structure (Figure 4a and 4b). Each DAG structure is made up of coflows that are exact replications of jobs taken from the original trace.

Our simulation also employs Equal-Cost multi-path routing (ECMP) [24], that is commonly used in datacenter to route packets and load balance network in datacenter, is also incorporated into our flow simulator. Additionally, since TCP is the common transport protocol in datacenter, we implement rate limiter that behaves like TCP for all schemes, except for Baraat where the rate limiter is implemented according to its design in [3].

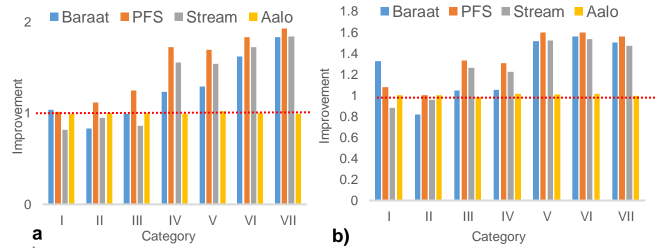

Figure 7. Average JCT in seven categories with FB-Tao (Fig. 7.a) and TPC-DS (Fig. 7.b) structure in bursty traffic scenario.

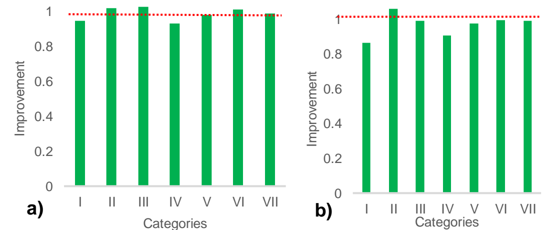

Figure 8. Average JCT in seven categories against the ideal Gurita (Gurita+) with FB-Tao (Fig. 8.a) and TPC-DS (Fig. 8.b) structure.

Simulation results. Here, we discuss Gurita's average performance in trace driven and bursty scenarios (Figure 5). Our experiment demonstrates that Gurita outperforms *PFS* by up to 2× in both scenarios (TPC-DS and FB-Tao). This is because Gurita dedicates its resources to higher priority jobs allowing more coflows in the network, while PFS allocates its resources equally among flows of coflow from different jobs that traversing the same link.

Gurita also outperforms *Baraat* by up to 1.8× faster in both scenarios. Baraat's performance suffers from lower priority

mice coflows queuing behind larger higher priority coflows in every job stage. Gurita is able to avoid this problem by allowing smaller coflows in a job stage to jump ahead of the queue, enabling job stages with fewer bytes to complete faster. In other words, Baraat punishes jobs for sending more bytes in some stages but fewer bytes in other stages by forcing them to share the resources with lower priority jobs. In contrast, Gurita is sensitive to the characteristic of multi-stage allowing larger jobs to accelerate their completion time when they have less bytes to send in some stages and at the same time to avoid being blocked by other jobs. Thus, Gurita allows larger jobs to complete faster.

Compared to *Stream*, Gurita achieves better average JCT by up to 1.5× faster in both scenarios. This is because Gurita allows larger jobs to shorten their JCT by allowing them to send data at a higher priority when they transmit less data in a stage, at the same time making more resources available to process more jobs. In contrast, Stream requires larger jobs to transmit at lower priority regardless of the amount of byte sent per stage, at the same time blocking jobs from resources.

Another advantage of Gurita over these decentralized schemes (Baraat, PFS, and Stream) is that, by prioritizing of Jobs with lower per stage blocking effect, it frees up more resources for other jobs. Thus, it achieves faster per stage completion time resulting in lower job completion time. Compared to centralized scheme *Aalo* with global view, Gurita achieves similar performance with less information by up to 1.05× faster in trace driven scenario but slightly slower in bursty scenario at just 0.01× slower. The key insight is jobs in Aalo may experience blocking, while Gurita leverage insight from the characteristic of multi-stage job to help jobs to avoid being blocked by other jobs with higher blocking effects.

Trace driven scenario. Next, we delve deeper by analyzing Gurita in seven different job categories described in table 1 (Figure 6) using data trace from Facebook datacenter. Gurita outperforms PFS across all categories, particularly for jobs in categories I and II (smaller size jobs) by up to 8.5× better performance. Similarly, Gurita also outperforms Baraat across all categories by up 5× faster. In comparison to Stream, Gurita achieves better performance in category acrosscategories in both scenarios, because measuring blocking effect per stage enables Gurita to quickly recognize jobs with fewer bytes per stage. Moreover, Gurita allocates more resources to allow larger jobs to transmit at a higher priority when they have less bytes in a stage, allowing them to complete faster while Stream punishes jobs for sending more bytes early. Therefore, by considering job blocking effect per stage, Gurita improves the average JCT and outperforms Stream in most categories by up to 4× faster.

Compared to Aalo, Gurita matches outcomes across categories with TPC-DS. However, Gurita's performance is comparable in category I with the FB-Tao structure at just 0.1 × slower. In this instance, Aalo is slightly more advantageous over Gurita (by 0.1× faster) because it is a centralized system with a global view, enabling it to be more precise in distinguishing small jobs. By recognizing the characteristic of multi-stage, Gurita matches the performance of the centralized solution without complete information.

Bursty traffic scenario in large scale network. In this scenario (Figure 7), jobs arrive within 2 microseconds intervals [17, 32] in much larger (48 pods FatTree) network topology of which consisting of 27648 servers and 2880 switches. Since the production trace used in the previous experiment is too small to generate network congestion, we generate 10000 jobs according workload provided in [4].

Here, we demonstrate that Gurita is scalable and is also able to precisely differentiate job's characteristics at different stages when jobs arrive within a small time interval. In Figure 7, Gurita outperforms PFS across all categories by up to 2× faster across all categories in both scenarios (FB-Tao and TPC-DS). Compared to Baraat, Gurita achieves 1.8 × improvement with lower JCT across all categories. Gurita also largely outperforms Stream by up to 1.9× faster JCT across all categories, except in category 1 in both scenario. This is because Stream utilizes strict priority queue to schedule coflow of different jobs, which allows Stream to quickly allocate the entire resources to small jobs (job in category 1 of table 1).

On the other hand, Gurita allocates some portions of the resources to mitigate starvation that can occur to jobs with lower priority [14, 25]. Generally, Gurita matches the performance compared to Aalo across categories. In these experiments, we have demonstrated that considering job characteristics granularity at different stage allowing Gurita to outperform PFS, Baraat, and Stream. At the same time, Gurita matches the performance of the centralized scheme with global view, Aalo.

Comparison to GuritaPlus. Here, we compare Gurita to an enhanced version, we call it GuritaPlus, where information on the total amount of bytes sent per stage is available and job priority can be adjusted spontaneously without concerning TCP out of order problem. GuritaPlus determines the blocking effect per stage by utilizing total in-flight bytes sent per stage. In-flight bytes are bytes that have been transmitted into the network but have not reached the destination. These assumptions allow GuritaPlus to be more precise in the scheduling decision.

Additionally, the simulation is also conducted using trace from Facebook datacenter in 8 pods FatTree network. Figure 8 demonstrates that Gurita achieves similar outcome compared to GuritaPlus across categories. In the worst case, Gurita is only slightly behind GuritaPlus by at most within 0.15% of GuritaPlus' performance. These outcomes demonstrate that the utilization of observed information at the receiver's end also provides sufficient approximation of a more ideal scenario that used the total amount of in-flight bytes per stage.

RELATED WORK

One of the early works on this theme is Orchestra [6], where the semantic among flows is accounted in the design of the flow transfers optimization. Varys [4] and Aalo [5] improve the performance in [6] by adopting Shortest Job First (SJF) in their scheduling mechanisms. RAPIER [7] and OMCoflow [13] incorporate routing algorithm into the scheduling scheme. The authors of [9,11] formulate the scheduling problem into weighted CCTs minimization problem. CODA

[12] leverages machine learning techniques to infer and schedule coflows. Baraat [3], a heuristic that adopts FIFO with some level of multiplexing allows mice flows to be processed in the background in the presence of large coflows. Stream [13] and Creek [41] are coflow scheduling schemes that takes advantage of coflow communication patterns. Coflex [34] is a coflow scheduling that takes max-min fairness into consideration. MCS [38] schedules coflows according to number of flows and flow length of a coflow. Another study in MRTF [39] propose a coflow scheduling scheme that takes in-network congestion in the design consideration. All of these approaches use total accumulated bytes sent to schedule coflows, but overlooks that multi-stage coflows may different characteristics and transmit different amount of bytes at different stages. However, it assumes that job size and structure are known ahead of time, limiting use in practice.

VI. Conclusion

Gurita is a scheduling scheme for coflows of multi-stage job that leverages job and coflow characteristics at different stages to minimize average JCT without resorting to central controller. The outcomes from our experiments demonstrate that Gurita is an effective solution in improving network performance in datacenter. Gurita outperforms decentralized schemes such as PFS, Baraat and Stream, and matches the performance of the centralized scheme Aalo that has access to global view, despite Gurita not having complete information.

Reference

1. http://www.pica8.com/documents/pica8-datasheet-picos.pdf
2. T. Gonzales and S. Sahni, “Open Shop Scheduling to Minimize Finish Time” J. for ACM Vol 23, No. 4, 1976, pp 665-679.
3. F. Dogar, et al, “Decentralized Task-Aware Schduling for Data Center Networks”, ACM SIGCOMM, 2014.
4. M. Chowdhury, Y. Zhong, and I. Stoica, ”Efficient Coflow Scheduling with Varys”, ACM SIGCOMM, 2014.
5. M. Chowdhury and I. Stoica, ”Efficient Coflow Schduling Without Prior Knowldege”, ACM SIGCOMM, 2015.
6. M. Chowdhury, et al,”Managing Data Transfer in Computer Clusters with Orchestra”, ACM SIGCOMM, 2011.
7. Y. Zho, et al, “RAPIER: Integrating Routing and Scheduling for Coflow-aware Data Center Networks”, IEEE INFOCOM 2015.
8. Z. Huang, et al “Need for Speed: CORA Scheduler for Optimizing Completion Time in the Cloud”, INFOCOM 2015.
9. Z. Qiu, et al, “ Minimizing the Total Weighted Completion Time of Coflows in Datacenter Networks”, ACM SPAA, 2015.
10. N Bronson, et al, “TAO: Facebook’s Distributed Data Store for the Social Graph”, USENIX ATC, 2013..
11. B. Tian, et al., “Scheduling Coflows of Multi-stage Jobs to Minimize the Total Weighted Job Completion Time”, IEEE INFOCOM, 2018.
12. H. Zhang, et al.,”CODA: Toward Automatically Identifying and Scheduling Coflows in the Dark”, ACM SIGCOMM, 2016.
13. Y. Li, et al.,”Efficient Online Coflow Routing and Scheduling”, ACM MOBICHOC, 2016.
14. H. Susanto, et al., “Stream: Decentralized Opportunistic Inter Coflows Scheduling for Datacenter Networks”, IEEE ICNP, 2016.
15. M. Chowdhury and I. Stoica, “Coflow: A Networking Abstraction for Cluster Applications”, USENIX HotNets, 2012.
16. A. Roy, et al, “Inside the Social Network’s (Datacenter) Network,” in ACM SIGCOMM, 2015.
17. T. Benson, A. Akella, and D. A. Maltz, ”Network Traffic Characteristics of Data Centers in the Wild”, ACM IMC, 2010.
18. J. Dean and S. Ghemawat, “MapReduce: Simplifed Data Processing on Large Clusters”, USENIX OSDI, 2004.
19. G. Malewicz, et al.,”Pregel: A System for Large-Scale Graph Processing”, ACM SIGMOD, 2008.
20. M. Zhaharia, et al., “Resilent Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Custer Computing”, USENIX NSDI, 2008.
21. Y. Yu, et al., “DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High Lelvel Language”, OSDI, 2012.
22. NetFilter Packet Filtering Framework for Linux, www.netfilter.org/
23. M. Alizadeh, et al, “Data Center TCP (DCTCP)”, SIGCOMM, 2010.
24. J. Kurose and K. Ross, “Computer Networking, a Top Down Approach 6th addition”, Pearson, 2013.
25. W. Bai, et al, ”Information-Agnostic Flow Scheduling for Comodity Data Centers”, USENIX NSDI, 2015.
26. M. Al-Fares, A. Laukissas, and A. Vahdat, “A Scalable, Commodity Data Center Network Architecture”, ACM SIGCOMM, 2008.
27. E. Tardos and J. Kleinberg, “Algorithm Design”, Pearson, 2005.
28. R. Grandl, et al, “Graphene: Packing and Dependency-aware Scheduling for Data-Parallel Clusters”, USENIX OSDI, 2016.
29. A Hash Function for Hash Table Lookup, http: //burtleburtle.Net/bob/ hash/doobs.html
30. S. M. Johnson, “Optimal Two and Tree Stage Production Schedules with Set-up Time Included”, Naval Research Log. Quart. Vol. 1, 1954.
31. B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlos, “On Scheduling in Map-Reduce and Flow-Shops”, ACM SPAA, 2011.
32. L. A. Hall, “Approximity of Flow Shop Scheduling”, FOCS, 1995.
33. M. R., Johnson, et al. ,“The complexity of flowshop and jobshop scheduling”. Mathematics of operations research, 1(2), 117-129 (1976).
34. W. Wang S. Ma, B. Li, and B. Li, “Coflex: Navigating the Fairness-Efficiency Tradeoff for Coflow Scheduling”, IEEE INFOCOM, 2017.
35. P. Poupart, et al., “Online Flow Size Prediction for Improved network Routing”, IEEE ICNP, 2016.
36. L. Kleinrock, “Queuing Systems Vol. 2 Computer Application”, New York, Wiley, 1976.
37. S. Wang, et al., “Multi-Attributes-Based Coflow Scheduling Without Prior Knowledge”, IEEE/ACM ToN, Vol 26, NO. 4, August 2018.
38. A. A. Nair, et al., “A First-Order Mechanistic Model for Atchitectural Vunerability Factor”, ISCA, 2012.
39. T. Zhang, et al., “Distributed Bottle-Aware Coflow scheduling in Data Centers “, ACM TPDS, 2018.
40. G. Judd, “Attaining Promise and Avoiding Pitfalls of TCP in the Datacenter”, USENIX NSDI, 2015.
41. H. Susanto, “Creek: Inter Many-to-Many Coflows Scheduling for Datacenter Networks, *IEEE ICC*, 2019.
42. A. M. Abdelmoniem, B. Bensaou. “Efficient Switch-Assisted Congestion Control for Data Centers: an Implementation and Evaluation”, IEEE IPCCC, 2015.
43. A. M. Abdelmoniem, B. Bensaou., “HyGenICC: hypervisor-based generic IP congestion control for virtualized data centers”, IEEE ICC, 2016.
44. A. M. Abdelmoniem, B. Bensaou., “IncastGuard: An efficient TCP-incast mitigation mechanism for cloud networks”, IEEE Globecom, 2018.
45. A. M. Abdelmoniem, B. Bensaou., “SDN-based Incast Congestion Control Framework for Data Centers: Implementation and Evaluation”,
46. A. J. Abu, B. Bensaou and A. M. Abdelmoniem, "A Markov model of CCN pending interest table occupancy with interest timeout and retries,", IEEE ICC, 2016
47. A. S. Sabyasachi, H. M. D. Kabir, A. M. Abdelmoniem and S. K. Mondal, "A Resilient Auction Framework for Deadline-Aware Jobs in Cloud Spot Market,", IEEE, SRDS, 2017.