Surviving Failures with Performance-Centric Bandwidth Allocation in Private Datacenters

Li Chen[1] Baochun Li[1] Bo Li[2]
[1]University of Toronto
[2]The Hong Kong University of Science and Technology

Abstract—**In the context of private datacenters that are operated by Web service providers such as Google, multiple applications using data parallel frameworks, such as MapReduce, coexist and share a limited supply of link bandwidth capacities. It has been shown that failures are the norm, rather than the exception, in datacenters, and will negatively affect the performance of data parallel applications as failed tasks need to be relaunched and placed on newly selected servers. In this paper, we argue that even with the presence of failures, link bandwidth should be allocated to competing applications with *performance-centric fairness*, in that the *performance* that applications enjoy should be proportional to their weights. We formulate and solve the open challenge of jointly optimizing placement decisions for relaunched tasks and bandwidth allocation, so that the adverse effects of failures on application performance are minimized. With our proposed algorithm implemented in the Mininet emulation testbed, our experiments show the effectiveness of our solutions towards minimizing the negative effects of failures, while still achieving performance-centric fairness.**

I. Introduction

Datacenters have become the *de facto* standard computing platform for Web service providers and online social networks — such as Google and Facebook — to host a wide variety of computationally intensive applications, ranging from PageRank [1] to machine learning [2]. To process large volumes of data, these applications typically embrace *data parallel* frameworks, such as MapReduce [3] and Dryad [4], which proceed in several *computation* stages that require *communication* between them. With MapReduce, for example, input data is first partitioned into a set of *splits* [3], so that they can be processed in parallel with *map* computation tasks. The map tasks produce intermediate results, which are then shuffled over the datacenter network to be processed by *reduce* computation tasks.

As multiple data parallel applications share the same private datacenter operated by a Web service provider, we wish to allocate link bandwidth to these applications *fairly*. But how should the notion of fairness be defined when allocating bandwidth in a private datacenter? In our previous work [5], we have introduced and defined the notion of *performance-centric fairness*, in that fairness should be maintained with respect to the performance across multiple data parallel applications. In particular, as available bandwidth is allocated for data parallel applications to transfer data in their communication stages, their performance is best represented by the amount of time needed to complete the data transfer, called the *transfer time*. The guiding principle of weighted performance-centric fairness is that the reciprocal of the transfer times should be proportional to their weights across competing applications. To put it simply, applications with equal weights sharing the same private datacenter should enjoy the same performance.

The major open challenge, however, is to allocate bandwidth according to the notion of performance-centric fairness, yet *with the presence of failures*. Recent measurement studies have clearly pointed out that physical servers and their access links in datacenters exhibit a large number of short-term failures [6]. Such failures will negatively affect the performance of data parallel applications with tasks running on failed servers, since failed tasks will need to be relaunched on alternative servers, and intermediate data will need to be retransmitted from these relaunched tasks. This is typically referred to as the *straggler* problem in MapReduce applications.

The presence of failures changes the design space of performance-centric bandwidth allocation in a fundamental way. The most pressing concern, of course, is to determine which alternative servers should be selected to relaunch these failed tasks, which is a decision that needs to be made as soon as failures occur. Yet, by placing relaunched tasks in newly selected servers, the communication pattern among communicating tasks has been changed, which necessitates the re-allocation of link bandwidth such that failed tasks may finish more quickly with a larger share of bandwidth. Preferably, flows from failed tasks may even finish at the same time as other flows in the same application.

In this paper, our original contributions are based on the belief that the *placement* of relaunched tasks after failures occur and the *allocation of link bandwidth* following the principle of performance-centric fairness should be considered and optimized *jointly*. With an example, we show that such joint consideration is necessary and instrumental, due to the fact that the communication pattern is materially changed as different servers are selected to host the relaunched tasks.

By re-allocating bandwidth at the same time of placing relaunched tasks onto newly selected servers, our objective is to diversify the risk of failures on a subset of unfortunate

The co-authors would like to acknowledge the gracious research support from the NSERC Discovery Research Program and the SAVI NSERC Strategic Networks Grant at the University of Toronto, the grants from RGC under the contracts 615613 and 16211715, and the grant from NSFC and Guangdong joint project under the contract U1301253.

applications, by spreading their adverse effects on application performance among all the applications with performance-centric fairness. In order to achieve this objective, we first formulate the joint optimization problem in the general case as a mixed-integer nonlinear programming problem. In order to narrow the solution space when the Branch-and-Cut method is used to solve this problem, we propose a preliminary filtering algorithm to choose a set of server candidates to be considered. With our proposed algorithm implemented in the Mininet emulation testbed, our experiments show the effectiveness of our solutions towards minimizing the negative effects of failures, while still achieving performance-centric fairness.

The remainder of this paper is organized as follows. In Sec. II, we introduce the principles of performance-centric bandwidth allocation, and present an example to serve as a clear motivation for the joint optimization of task placement and bandwidth allocation. We formulate the joint optimization problem in Sec. III, and then present our algorithm to solve the problem in Sec. IV. Our experimental results are presented in Sec. V, demonstrating the effectiveness of our solutions towards minimizing the adverse effects of failures, while still achieving performance-centric fairness. Finally, we conclude the paper in Sec. VII.

II. Background and Motivation

A. Allocating Bandwidth with Performance-centric Fairness

We first introduce the intuition of performance-centric fairness [5], which will be used as the guiding principle for our bandwidth allocation among data parallel applications.

In a private datacenter shared by concurrent data parallel applications, the network performance, measured by the transfer time of the communication stages, achieved by these applications is the most important concern. Allocating bandwidth with performance-centric fairness across competing applications — or, simply put, *performance-centric bandwidth allocation* — has two implications, with respect to allocating bandwidth to flows *within an application* and *across applications*.

Within an application, bandwidth should be allocated in proportion to the volumes of data to be sent, so that all of the flows in its communication stage finish *at the same time*. This is based on the observation that the application performance is determined by the completion time of the slowest flow. Since faster flows do not improve the application performance, we can reduce their bandwidth allocation until they all finish at the same time with the slowest flow. In this way, the performance of the application would not be degraded, yet the saved bandwidth can be utilized by other applications.

This principle is particularly useful with the presence of failures: the delays of relaunching tasks and re-transmitting intermediate data from these tasks can be compensated by allocating more bandwidth to speed up the corresponding flows, to the extent that they can finish at the same time with the other flows.

Across applications, bandwidth should be allocated such that the performance[1] achieved by these applications is proportional to their weights. However, such a simple objective is non-trivial to achieve especially when task placement is jointly considered at the presence of failures, since the volumes of network traffic in their communication stages and the sharing relationship among their flows are influenced by the placement of their tasks.

B. Joint Task Placement and Bandwidth Allocation with Failures: An Example

We now explore the coupling relationship between task placement decisions and performance-centric bandwidth allocation, after the occurrence of failures. To begin with, we illustrate the coupling relationship between task placement and *the first principle* of performance-centric bandwidth allocation *within a single application.*

In the illustrative example shown in Fig. 1, a data parallel application *A* has two tasks *A1, A2* that produce intermediate data to be transmitted to tasks *A3, A4* in the communication stage. All servers have the same bandwidth capacity of 200 MB/s. Suppose a failure occurs to the server *S2*. As a result, task *A2* has to be relaunched on an alternative server.

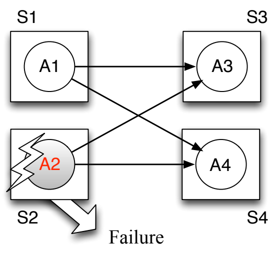

Fig. 1: Failure occurs on server S2, and task A2 has to be relaunched.

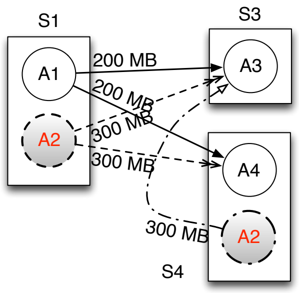

Fig. 2: Two possible placements and bandwidth allocation within a single application.

Two possible placements of *A2* are shown in Fig. 2, with *A2* either co-located with *A1* on *S1*, or relaunched on *S4*. Because of the interruption of failures, the relaunched *A2* has a larger amount of data to be transmitted than *A1*. To be particular, when *A2* starts to transfer 300 MB intermediate data by each flow, the volume of data left to be sent by each flow of *A1* is 200 MB.

Let us first consider how bandwidth is to be allocated if we relaunch *A2* on *S1*. To ensure that all the flows complete at the same time, the bandwidth of *S1* needs to be allocated to flows of *A1* and *A2* with a $2:3$ ratio. In consequence, the *A1*-*A3* flow will be allocated a bandwidth of 40 MB/s while the *A2*-*A3* flow obtains 60 MB/s, so that they can both finish in 5 seconds.

An alternative placement of *A2*, *i.e.*, co-locating *A2* with *A4* on *S4* as shown in Fig. 2, reduces the bandwidth demand, since *A2* no longer needs to transmit data to *A4* over the network. With this placement, the bandwidth bottleneck is

[1]The performance considered in this paper is the network performance. With mature techniques, computation performance of tasks is predictable, which is beyond our scope.

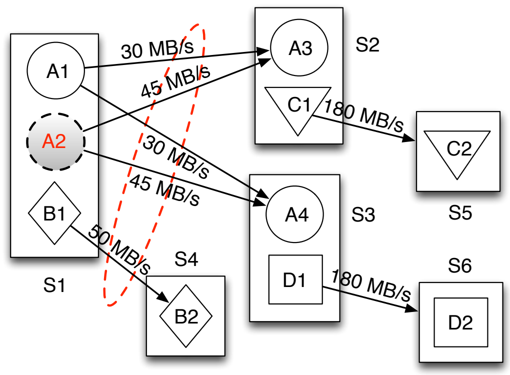

Fig. 3: One possible solution of placing relaunched tasks as multiple applications compete for bandwidth.

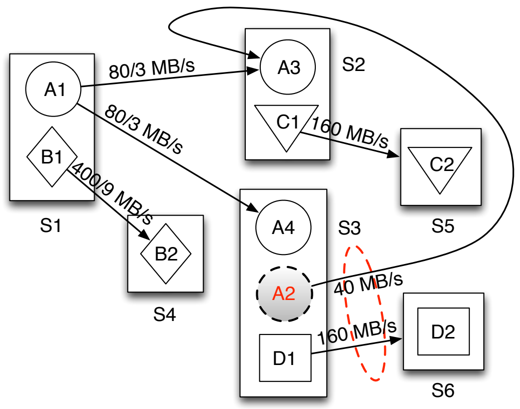

Fig. 4: An alternative solution of placing relaunched tasks as multiple applications compete for bandwidth.

at *S3*. In this case, we may allocate 80 MB/s to the *A1-A3* flow, and 120 MB/s to *A2-A3*, so that they both finish in 2.5 seconds. Therefore, if we make placement decisions with performance-centric bandwidth allocation jointly considered, better performance may be achieved, since communicating tasks may be co-located to reduce the data to be transmitted over the datacenter network.

The problem becomes more intricate when we further consider *the second principle* of performance-centric bandwidth allocation *across multiple applications*, in the case that *A* shares the datacenter network with other applications, as shown in Fig. 3 and Fig. 4.

In Fig. 3, when the relaunched *A2* on *S1* starts to transmit 300 MB data by each flow (*A2-A3, A2-A4*), the amounts of data to be sent by *B1-B2, C1-C2* and *D1-D2* are $1000/3, 300, 300$ MB, respectively. The weights of *A, B, C* and *D* are $1, 1, 4$ and 4, respectively. The best possible performance-centric bandwidth allocation coupled with this placement is illustrated in the figure, with the bandwidth of *S1* fully allocated among flows of *A* and *B*. The transfer time of *A* is $t_A = 200\text{MB}/30\text{MB/s} = 300\text{MB}/45\text{MB/s} = \frac{20}{3}s$. Similarly, we can calculate the transfer times of *B, C* and *D* as $t_B = \frac{1000}{3}\text{MB}/50\text{MB/s} = \frac{20}{3}s$, $t_C = 300\text{MB}/180\text{MB/s} = \frac{5}{3}$ s and $t_D = 300\text{MB}/180\text{MB/s} = \frac{5}{3}$ s. As we can see, weighted performance-centric fairness is successfully achieved with the weight ratio of $\frac{1}{t_A} : \frac{1}{t_B} : \frac{1}{t_C} : \frac{1}{t_D} = 1 : 1 : 4 : 4$ satisfied.

If task *A2* is placed on *S3*, however, the best possible performance-centric bandwidth allocation is illustrated in Fig. 4, where the egress link of *S3* becomes the bottleneck that is fully utilized. The transfer times achieved with this allocation are: $t_A = 200\text{MB}/\frac{80}{3}\text{MB/s} = 300\text{MB}/40\text{MB/s} = 7.5$ s, $t_B = \frac{1000}{3}\text{MB}/\frac{400}{9}\text{MB/s} = 7.5$ s, $t_C = 300\text{MB}/160\text{MB/s} = \frac{7.5}{4}$ s and $t_D = 300\text{MB}/160\text{MB/s} = \frac{7.5}{4}$ s. Therefore, we have $\frac{1}{t_A} : \frac{1}{t_B} : \frac{1}{t_C} : \frac{1}{t_D} = 1 : 1 : 4 : 4$, satisfying the requirement of weighted performance-centric fairness. Compared with the previous example shown in Fig. 3, the performance of applications achieved by the joint placement and bandwidth allocation in this example is worse. Although the co-location of *A2* and *A4* reduces the network demand, sharing bandwidth with *D1* results in less bandwidth allocated to *A* than sharing with *B1*, because *D1* is more bandwidth demanding than *B1*.

With these examples, we have clearly shown that placement decisions for failed tasks will fundamentally reshape the design space of performance-centric bandwidth allocation. Their effects on application performance are coupled with each other. To maximize the application performance, *i.e.*, to minimize the adverse effects of failures, we should jointly optimize task placement and performance-centric bandwidth allocation. We will generalize our insights from these examples to the general case, in the next section.

III. Problem Formulation

With the intuition obtained from our examples, we now formulate the problem of jointly optimizing the task placement and bandwidth allocation decisions as failures occur in a private datacenter.

We consider a datacenter where there are K data parallel applications —such as MapReduce [3] and machine learning applications [2] — running concurrently, with their tasks distributed across N physical machines (or servers interchangeably). Each application $k \in \mathcal{K} = \{1, 2, ..., K\}$ requires m_k tasks, represented by $\mathcal{T}_k = \{1, 2, ..., m_k\}$. The i-th task of application k is represented by $\mathcal{T}_k^i \in \mathcal{T}_k$. Similar to existing works [7], [8], we assume that the network load matrix $\mathcal{D}_k$ can be measured, where the (i, j)-th component $\mathcal{D}_k^{i,j}$ represents the amount of data to be sent by the flow between task $\mathcal{T}_k^i$ and $\mathcal{T}_k^j$.

Let $r_k^{i,j}$ denote the bandwidth allocated to the communicating task pair (i, j) of application k, then the completion time of the flow between the task pair (i, j) is $\frac{\mathcal{D}_k^{i,j}}{r_k^{i,j}}$. The transfer time of application k, defined as the completion time of the slowest flow in the communication stage, can thus be represented as:

$$t_k = \max_{i,j,\mathcal{D}_k^{i,j} \neq 0} \mathcal{D}_k^{i,j} / r_k^{i,j}$$

According to the first principle of performance-centric bandwidth allocation, bandwidth allocated to flows within an application should be in proportion to the volumes of data to be sent by them, *i.e.*, $r_k^{i,j} \propto \mathcal{D}_k^{i,j}$, so that all flows of the application will finish at the same time. As a result, we have:

$$r_k^{i,j} / \mathcal{D}_k^{i,j} = 1/t_k = \alpha_k, \quad \forall i, j \in \mathcal{T}_k, \mathcal{D}_k^{i,j} \neq 0, \tag{1}$$

where $\alpha_k = \frac{1}{t_k}$ represents the performance of application k, indicating that the shorter the transfer time, the better the performance.

Based on the second principle of performance-centric bandwidth allocation, performance-centric fairness should be achieved across competing applications:

$$\alpha_k = 1/t_k = w_k S / \sum_k w_k, \quad \forall k \in \mathcal{K}, \tag{2}$$

where w_k is the weight of application k, and S is a positive variable called the *total performance-centric share*, which is upper bounded given the fixed amount of bandwidth capacity in the shared datacenter.

On each server $n \in \mathcal{N} = \{1, 2, ..., N\}$, tasks from different applications share its CPU resource, with the capacity of C_n, and its link bandwidth, including both the egress link with capacity B_n^E and the ingress link with capacity B_n^I. Since the bisection bandwidth in datacenter networks has been significantly improved by multi-path routing (*i.e.*, [9]) and multi-tree topologies (*i.e.*, [10]), we assume a full bisection bandwidth network, where bandwidth is only bottlenecked at the access links of physical machines. Consequently, the completion time of each flow is determined by the bandwidth allocated at the access links.

Let the binary variable $X_{k,n}^i$ denote whether task i of application k is placed on server n, *i.e.*,

$$X_{k,n}^i = \begin{cases} 1, \text{ when } \mathcal{T}_k^i \text{ is placed on server } n \\ 0, \text{ otherwise.} \end{cases} \tag{3}$$

If we use c_k^i to denote the CPU requirement by task $\mathcal{T}_k^i$, the CPU capacity constraint on server n is represented as:

$$\sum_k \sum_i c_k^i \cdot X_{k,n}^i \leq C_n. \tag{4}$$

The total egress rate of each task $\mathcal{T}_k^i$ placed on server n is $\sum_{j, X_{k,n}^j = 0} r_k^{i,j} \cdot X_{k,n}^i$. Constraint $X_{k,n}^j = 0$ in the summation reflects the fact that there will be no data sent through the network if the task $\mathcal{T}_k^j$ receiving the intermediate data from $\mathcal{T}_k^i$ is also placed on server n. Summing over all the tasks of an application, and then all the applications, we obtain the total egress rate of server n, which should not exceed the egress link capacity:

$$\sum_k \sum_i \sum_{j, X_{k,n}^j = 0} r_k^{i,j} \cdot X_{k,n}^i \leq B_n^E.$$

From Eq. (1), $r_k^{i,j}$ can be represented as follows:

$$r_k^{i,j} = \mathcal{D}_k^{i,j} / t_k = \alpha_k \cdot \mathcal{D}_k^{i,j}, \quad \forall k \in \mathcal{K}, i, j \in \mathcal{T}_k, \tag{5}$$

which indicates that the egress link capacity constraint can be equivalently represented as:

$$\sum_k \alpha_k \sum_i \sum_{j, X_{k,n}^j = 0} \mathcal{D}_k^{i,j} X_{k,n}^i \leq B_n^E. \tag{6}$$

Similarly, the total ingress rate of server n is $\sum_k \sum_i \sum_{j, X_{k,n}^j = 0} r_k^{j,i} \cdot X_{k,n}^i$, which can be further transformed to $\sum_k \alpha_k \sum_i \sum_{j, X_{k,n}^j = 0} \mathcal{D}_k^{j,i} X_{k,n}^i$. We hence have the capacity constraint for the ingress link as follows:

$$\sum_k \alpha_k \sum_i \sum_{j, X_{k,n}^j = 0} \mathcal{D}_k^{j,i} X_{k,n}^i \leq B_n^I. \tag{7}$$

When a server fails, the transfers of all the tasks running on it would be terminated [2]. As a result, all the intermediate data needs to be retransmitted once these tasks are relaunched on other available servers. For a task of the second computation stage, it starts to fetch the intermediate data as soon as it is relaunched. For a task that belongs to the first computation stage, we know that its computation has already finished and the whole set of intermediate data has been generated. It is not necessary to relaunch it from scratch to implement the computation again. Instead, we can select a server with a backup task, or select a server that has the replication of the intermediate data, based on existing technique [11]. Therefore, in both cases, the transfers would start immediately once the placement is given[3].

For convenience, we call these tasks the failed tasks, denoted by the set $\mathcal{T}_f$. The set of servers that are undergoing failures is represented by $\mathcal{N}_f$, while $\mathcal{N}_{-f}$ denotes the set of servers that are running normally. The problem of task placement is to choose servers from $\mathcal{N}_{-f}$ to relaunch the failed tasks in $\mathcal{T}_f$. For differentiation, we use $\mathcal{X} = \{\mathcal{X}_{k,n}^i | \mathcal{T}_k^i \in \mathcal{T}_f, n \in \mathcal{N}_{-f}\}$ to represent the placement of failed tasks as variables, while $X = \{X_{k,n}^i | \mathcal{T}_k^i \in \mathcal{T}_{-f}, n \in \mathcal{N}_{-f}\}$ denotes the placement of other tasks, which is a known constant. For binary variables $\mathcal{X}_{k,n}^i \in \mathcal{X}$, the following constraint should be satisfied:

$$\mathcal{X}_{k,n}^i \in \{0, 1\}, \sum_{n \in \mathcal{N}_{-f}} \mathcal{X}_{k,n}^i = 1, \ \forall \mathcal{X}_{k,n}^i \in \mathcal{X}, \tag{8}$$

which indicates that each failed task should be placed on one of the normal servers. As aforementioned, a failed task that belongs to the first computation stage needs to be placed on a server with backup task or replication of intermediate data so that it can transmit data immediately. Hence, we can rule out the placement on the servers without backup tasks or without intermediate data, by setting the corresponding binary variables as 0, *i.e.*,

$$\mathcal{X}_{k,n}^i = 0, \quad \text{if} \quad l(k, i, n) = 1, \tag{9}$$

where $l(k, i, n)$ equals 1 if $\mathcal{T}_k^i$ is a task in the fist computation stage and server n does not have its backup task or intermediate data.

With such notations, Eq. (4), (6) and (7) can be represented as follows:

$$\sum_k \Big(\sum_{i, \mathcal{T}_k^i \in \mathcal{T}_f} c_k^i \cdot \mathcal{X}_{k,n}^i + \sum_{i, \mathcal{T}_k^i \notin \mathcal{T}_f} c_k^i \cdot X_{k,n}^i \Big) \leq C_n \tag{10}$$

$$\sum_k \alpha_k \Big(\sum_{i, \mathcal{T}_k^i \in \mathcal{T}_f} d_{k,n}^{i,E} \mathcal{X}_{k,n}^i + \sum_{i, \mathcal{T}_k^i \notin \mathcal{T}_f} d_{k,n}^{i,E} X_{k,n}^i \Big) \leq B_n^E \tag{11}$$

$$\sum_k \alpha_k \Big(\sum_{i, \mathcal{T}_k^i \in \mathcal{T}_f} d_{k,n}^{i,I} \mathcal{X}_{k,n}^i + \sum_{i, \mathcal{T}_k^i \notin \mathcal{T}_f} d_{k,n}^{i,I} X_{k,n}^i \Big) \leq B_n^I, \tag{12}$$

[2]Note that all the tasks considered in our model are in the communication stages when failures happen. The tasks that are not sending or receiving data do not require bandwidth allocation, thus relaunching these tasks is out of the scope of our joint problem.

[3]Even if this does not hold, our model can be extended to consider the delay of relaunching a task, and predict network load of other tasks when the relaunched task starts its communication stage. We leave it as future work.

where $d_{k,n}^{i,E} = \sum_{j,\mathcal{T}_k^j \notin \mathcal{T}_f, X_{k,n}^j=0} \mathcal{D}_k^{i,j} + \sum_{j,\mathcal{T}_k^j \in \mathcal{T}_f, \mathcal{X}_{k,n}^j=0} \mathcal{D}_k^{i,j}$, representing the total amount of data to be sent by task $\mathcal{T}_k^i$ when it is placed on server n, and $d_{k,n}^{i,I} = \sum_{j,\mathcal{T}_k^j \notin \mathcal{T}_f, X_{k,n}^j=0} \mathcal{D}_k^{j,i} + \sum_{j,\mathcal{T}_k^j \in \mathcal{T}_f, \mathcal{X}_{k,n}^j=0} \mathcal{D}_k^{j,i}$, representing the total amount of data to be received by task $\mathcal{T}_k^i$ at server n.

We are now ready to formulate the problem of jointly optimizing task placement and performance-centric bandwidth allocation as follows:

$$\max_{\alpha, \mathcal{X}_{k,n}^i \in \mathcal{X}} \quad S \quad \text{s.t. Eq. } (2), (8), (9), (10), (11), \text{and } (12).$$

The objective is to achieve the maximum S, so that the adverse effects to application performance caused by failures are minimized, and the performance achieved by each application is maximized. Eq. (2) represents the performance-centric fairness among applications. Eq. (10), (11), (12) correspond to capacity constraints for CPU, egress and ingress link capacities at each server. Eq. (8) indicates the property of the placement variables and Eq. (9) limits the placement of failed tasks of the first computation stage.

Substituting Eq. (2) into (11) and (12) yields:

$$S \cdot \sum_k w_k^{'} \Big(\sum_{i,\mathcal{T}_k^i \in \mathcal{T}_f} d_{k,n}^{i,E} \mathcal{X}_{k,n}^i + \sum_{i,\mathcal{T}_k^i \notin \mathcal{T}_f} d_{k,n}^{i,E} X_{k,n}^i \Big) \leq B_n^E \quad (13)$$

$$S \cdot \sum_k w_k^{'} \Big(\sum_{i,\mathcal{T}_k^i \in \mathcal{T}_f} d_{k,n}^{i,I} \mathcal{X}_{k,n}^i + \sum_{i,\mathcal{T}_k^i \notin \mathcal{T}_f} d_{k,n}^{i,I} X_{k,n}^i \Big) \leq B_n^I, \quad (14)$$

where $w_k^{'} = w_k / \sum_k w_k$, representing the relative weight of application k.

For the capacity constraint of CPU, we use $c_n = \sum_k \sum_{i,\mathcal{T}_k^i \notin \mathcal{T}_f} c_k^i \cdot X_{k,n}^i$ to represent the total CPU cycles required by the tasks running on server n, which is a known constant. Then Eq. (10) can be represented as follows:

$$\textstyle\sum_k \sum_{i,\mathcal{T}_k^i \in \mathcal{T}_f} c_k^i \cdot \mathcal{X}_{k,n}^i + c_n \leq C_n, \forall n \in \mathcal{N}_{-f}. \quad (15)$$

Therefore, the optimization problem can be transformed to the following form:

$$\max_{S, \mathcal{X}_{k,n}^i \in \mathcal{X}} \quad S \qquad \text{s.t. Eq. } (8), (9), (13), (14), \text{ and } (15)$$

This problem is a Mixed-Integer Nonlinear Programming (MINIP), as $\mathcal{X}_{k,n}^i$ are binary variables, S is continuous, and the capacity constraints for egress/ingress links are not linear. The MINIP problem is NP-hard [12], since it combines all the difficulties of its subclasses: the combinatorial nature of Mixed-Integer Programming and the difficulty of Nonlinear Programing. The dimension of difficulty with Nonlinear Programming can be removed by transforming the MINLP to the following equivalent form, with S being replaced by $p = 1/S$:

$$\begin{aligned}
\min_{p,\mathcal{X}} \quad & p & (16)\\
\text{s.t.} \quad & \textstyle\sum_k w_k^{'} \sum_{i,\mathcal{T}_k^i \in \mathcal{T}_f} d_{k,n}^{i,E} \mathcal{X}_{k,n}^i + b_n^E \leq B_n^E p & (17)\\
& \textstyle\sum_k w_k^{'} \sum_{i,\mathcal{T}_k^i \in \mathcal{T}_f} d_{k,n}^{i,I} \mathcal{X}_{k,n}^i + b_n^I \leq B_n^I p & (18)\\
& \text{Eq. } (8), (9), \text{ and } (15)
\end{aligned}$$

	explanation
SBD	flow demand of an application to achieve its fair performance share, *i.e.*, $D_k^{i,j} w_k^{'}$ for the flow between task $\mathcal{T}_k^i$ and $\mathcal{T}_k^j$
SBD-t	sum of *SBD* over all flows of task t
SBD-n	sum of *SBD-t* over all tasks on server n
rSBD-n	*SBD-n* divided by link bandwidth capacity B_n of server n

TABLE I: Notation explanation and expression.

where $b_n^E = \sum_k w_k^{'} \sum_{i,\mathcal{T}_k^i \notin \mathcal{T}_f} d_{k,n}^{i,E} X_{k,n}^i$ and $b_n^I = \sum_k w_k^{'} \sum_{i,\mathcal{T}_k^i \notin \mathcal{T}_f} d_{k,n}^{i,I} X_{k,n}^i$.

As the link capacity constraints are now linear, problem (16) is a Mixed-Integer Linear Programming (MILP), which is simpler than MINLP yet still NP-hard [13]. An intuitive way to solve the MILP is to simply change the constraint of $\mathcal{X}_{k,n}^i \in \{0, 1\}$ to $0 \leq \mathcal{X}_{k,n}^i \leq 1$, which is called Linear Programming (LP) relaxation, solve the relaxed LP problem and then round to the nearest binary variables that are feasible to the MILP. However, rounding to a feasible solution may be difficult, and more importantly, the solution may be far from the optimum. To compute the optimal solutions, we choose to use the exact algorithm with the method of Branch-and-Cut [14] in this paper, and using the solution of LP relaxation as a lower bound for MILP. Instead of applying Branch-and-Cut on the entire set of variables in the problem, we propose to first choose some promising candidate servers to reduce the search space, based on more in-depth insights on the problem itself.

IV. Algorithm

A. Interpretation and Insights

We first re-investigate Problem (16) and interpret its objective and constraints, with the hope of deriving better insights towards an optimal solution.

As shown in Table I, let us define the *standard performance-centric bandwidth demand*, or *SBD* in short, as the amount of bandwidth required by each flow of an application k to achieve its fair share of performance, *i.e.*, to finish within $1/w_k^{'}$ seconds. Obviously, the *SBD* for the flow between task $\mathcal{T}_k^i$ and $\mathcal{T}_k^j$ is $\mathcal{D}_k^{i,j} w_k^{'}$. In a similar vein, let *SBD-t* (for each task t) represent the total amount of *SBD* for all flows of a task, and let *SBD-n* (for each server n) denote the total amount of *SBD-t* for all the tasks placed on a server.

Now let us check the left side of Constraint (17) for server n. The first term $\sum_k w_k^{'} \sum_{i,\mathcal{T}_k^i \in \mathcal{T}_f} d_{k,n}^{i,I} \mathcal{X}_{k,n}^i$ can be considered as the *SBD-t* for the failed task $\mathcal{T}_k^i$ if it is to be placed on server n. Similarly, the second term $b_n^E = \sum_k w_k^{'} \sum_{i,\mathcal{T}_k^i \notin \mathcal{T}_f} d_{k,n}^{i,E} X_{k,n}^i$ represents the sum of *SBD-t* for all existing tasks on server n. Thus, the left side of Constraint (17), as the sum of *SBD-t* for all tasks on server n, is interpreted as the *SBD-n* for this server. In this way, the link bandwidth constraint for server n can be represented as *SBD-n* $\leq B_n p$, or *SBD-n*$/B_n \leq p$ equivalently. If we further denote *SBD-n*$/B_n$ for server n as the *relative SBD-n*, or *rSBD-n* in short, then our joint task placement and bandwidth allocation problem represented by (16) can be interpreted as follows:

—

The placement of a failed task t on a server n will increase the *SBD-n* for this server by the amount of *SBD-t* for this task. Moreover, the *SBD-t* for this task is also impacted by the server where it is to be placed. The objective of our problem is to find a placement of all failed tasks that enables the maximum of the *rSBD-n* across all the servers to achieve the minimum, represented as p^*, so that the performance of applications $\alpha_k^* = \frac{w_k}{\sum_k w_k} S^* = \frac{w_k}{\sum_k w_k}/p^*$ achieves the maximum.

With the interpretation above, we can derive two insights as follows: *First*, for each failed task t, the *SBD-t* for this task that will be added to the total *SBD-n* for the server, where t is to be placed, is not the same for all possible placement alternatives. When it is placed on a server where at least one of its communicating tasks within the transfer is running, the *SBD-t* for this task that will be added to the total *SBD-n* for this server is smaller than the *SBD-t* to be added when placed on a server without any communicating tasks. For example, when a failed Reduce task is placed on a server where a Map task of the same application is located, the network load will be reduced, thus resulting in a smaller *SBD-t* for this Reduce task. *Second*, for each failed task t, the servers with a smaller *rSBD-n* after the placement of the task are more preferred than those servers with a larger *rSBD-n*.

B. Preliminary Filtering

Inspired by the combination of the two insights, we propose the method of *preliminary filtering* as in Algorithm 1 to choose a set of servers as promising candidates for the placement of failed tasks.

Algorithm 1: Preliminary Filtering.

```
1:  Initialize the candidate set $\mathcal{C} = \varnothing$;
2:  Sort failed tasks $\mathcal{T}_k^i \in \mathcal{T}_f$ in a decreasing order of their
    required CPU cycles, resulting in the sorted set as $\mathcal{T}_f^s$;
3:  for all task $\mathcal{T}_k^i \in \mathcal{T}_f^s$ do
4:     for all server $n \in \mathcal{N}_{-f}$ do
5:        if Eq. (9) and Eq. (10) are satisfied then
6:           Compute rSBD-n for $n$ if $\mathcal{T}_k^i$ is placed on $n$;
7:           Insert $n$ to $S_k^n$ in increasing order of rSBD-n;
8:        end if
9:     end for
10:    Add the first $\eta$ ($\geq 1$) servers from $S_k^n$ to $\mathcal{C}$,
       and remove them from $\mathcal{N}_{-f}$;
11: end for
Output:
    The candidate set $\mathcal{C}$;
```

The main idea of preliminary filtering is that each failed task takes turns to greedily select a number of the most promising candidate servers, according to the *rSBD-n* for server n if the task is to be placed on it. More specifically, the server n with the smallest *rSBD-n* is the first to be selected to the candidate set, based on the insights we have previously derived.

The algorithm begins with sorting failed tasks according to their CPU requirements (*Line 2*). Particularly, a task with a larger CPU requirement has a smaller set of feasible placement. Hence, it would be allowed to choose the candidate servers ahead of a task with a smaller demand of CPU cycles. Among tasks with the same CPU requirement, a task of the first computation stage will take its turn first, since it has more constraint on its placement than tasks of the second stage, as indicated by Eq. (9).

Such a sorting rule is aimed at selecting more promising candidates (given a fixed setting of parameters). The simple intuition is that for a task with less feasible placement, once the feasible servers have been selected by other tasks, it would have no feasible server to choose; while for a task that has a lot of feasible placement, it could still add some feasible servers that are most promising into the candidate set.

With preliminary filtering, we will obtain a candidate set of servers, yet without fixing the mapping relationship between the failed tasks and the preferable servers selected by them. Instead, the final decision of the placement is to be made by the Branch-and-Cut algorithm, which will find the optimal placement among the reduced solution space determined by the candidate set.

Note that η (*Line 10* in Algorithm 1) is an integer parameter that can be tuned. When set as the smallest possible value, *i.e.*, $\eta = 1$, we can at least guarantee a solution space that contains a good feasible solution, since the candidates are selected in a greedy manner. In this case, the running time of our joint optimization achieves the best possible speedup, with a much smaller solution space. We may achieve the global optimum in this space when we are fortunate in some particular scenarios. However, in general, to increase the chance of achieving the global optimum, we may need to increase the solution space so that the global optimum is more likely to be reached, yet with longer running times.

The novelty of our algorithm lies in the important insights developed from the in-depth analysis of our formulation for joint performance-centric bandwidth allocation and task placement. The *standard performance-centric bandwidth demand*, as well as its varieties (*SBD*, *SBD-t*, *etc.*) used in our algorithm, are based on application-level performance-centric fairness. Also, the interplay between the communication pattern and the task placement is characterized when computing these demand metrics. Both of these insights significantly differentiate us from existing heuristics. The implementation details in practice will be discussed in the next subsection.

C. Joint Optimization

With a reduced solution space obtained from preliminary filtering, the Branch-and-Cut algorithm will be a much more feasible method to be used to find the optimum. After the optimal $S^* = 1/p^*$ and $\mathcal{X}^*$ are obtained by the Branch-and-Cut algorithm, we relaunch the failed tasks according to the optimal placement, and implement performance-centric bandwidth allocation to all the flows as follows:

$$r_k^{i,j^*} = \frac{w_k}{\sum_k w_k} S^* \cdot \mathcal{D}_k^{i,j}. \tag{19}$$

To summarize, our aforementioned algorithm for joint task placement and bandwidth allocation with performance-centric

Algorithm 2: Joint Task Placement and Performance-Centric Bandwidth Allocation.

```
Input:
    Servers with failures: $\mathcal{N}_f$; Normal servers: $\mathcal{N}_{-f}$;
    Bandwidth capacity: $B_n^E, B_n^I, \forall n \in \mathcal{N}_{-f}$;
    Network load matrix $\mathcal{D}_{i,j}^k$ and weight $w_k, \forall k \in \mathcal{K}$; Placement
    of tasks that are running on servers without failures: $X_{k,n}^i \in X$;
Output:
    Placement of tasks that are running on servers with failures:
    $\mathcal{X}_{k,n}^i \in \mathcal{X}$;
    Bandwidth allocation for all applications: $r_{i,j}^k$;
1: Obtain the candidate set $\mathcal{C}$ from Algorithm 1;
2: Branch-and-Cut [14] for problem (16) over the reduced solution
   space: $\overline{\mathcal{X}} = \{\mathcal{X}_{k,n}^i | \mathcal{T}_k^i \in \mathcal{T}_f, n \in \mathcal{C}\}$;
3: Relaunch failed task $\mathcal{T}_k^i \in \mathcal{T}_f$ on server $n$ if $\mathcal{X}_{k,n}^i = 1$.
4: Allocate bandwidth to flows of $k \in \mathcal{K}$ based on Eq. (19);
```

fairness is presented in Algorithm 2, which can be implemented in a global scheduler for resource sharing among data parallel applications in a private datacenter, such as Mesos [15]. Each application has a controller that detects failures and tracks the progress of its transfer, *i.e.*, the amounts of data left to be transmitted [7], [8].

When failures are detected by application controllers and reported to the global scheduler, Algorithm 2 will be executed in the global scheduler[4], which can gather up-to-date state information such as network load matrices and existing task placement of all the applications. According to Algorithm 1, each application controller that has reported failures will compute the relative demand (*rSBD-n*) for each server that has sufficient CPU cycles for the failed task. It then chooses η of these servers that have the smallest *rSBD-n*, and informs the global scheduler.

With the candidate servers reported by these application controllers, the global scheduler will merge them as the candidate set $\mathcal{C}$, and apply Branch-and-Cut algorithm for the joint optimization over such decreased solution space. Finally, the optimal solution is conveyed to the respective application controllers for them to relaunch failed tasks and allocate bandwidth accordingly.

With this algorithm, the best possible performance for all the applications has been achieved, and the adverse effects to application performance caused by failures are minimized. Across all the concurrent applications, fairness is achieved with respect to the application-level performance. When resource utilization is further considered, we can arbitrate the tradeoff between performance-centric fairness and bandwidth efficiency, as proposed in our previous work [5]. More intuitions and properties of our jointly optimal strategy will be investigated in our performance evaluation.

V. Performance Evaluation

In this section, we investigate how the proposed algorithm performs in minimizing the adverse effects of failures while maintaining performance-centric fairness, with a detailed analysis in a typical scenario implemented in the Mininet 2.0 emulation testbed [16]. We also conduct extensive simulations at large scale, to evaluate the performance of our algorithm in a general setting.

[4]If multiple failures are detected within a scheduling interval, which can be set by the datacenter operator, the failed tasks will be scheduled once at the end of this scheduling interval.

A. Mininet Testbed Evaluation

As we consider the network with full bisection bandwidth, we emulate a small cluster with a single switch topology in Mininet, interconnecting 8 homogeneous servers (*S0, S1, ... , S7*) with links of 10 MB/s. The cluster is shared by 5 applications (*A, B, C, D, E*), each with multiple tasks placed across several servers, as shown in Fig. 5. Such an emulation is for the sake of verification and detailed analysis. We leave real-world implementation as our future work.

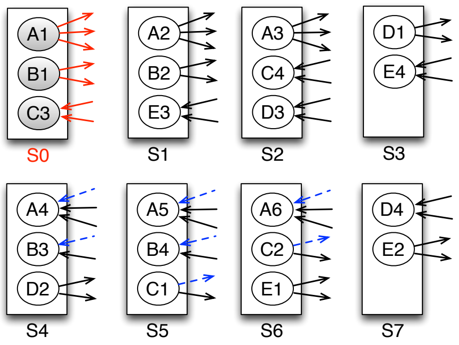

Fig. 5: A private datacenter with applications *A, B, C, D* and *E* running concurrently on 8 servers and a failure happens to *S0*.

Since our focus is on the network transfer in the communication stages of an application, we use the *Netcat* (`nc`) tool to implement file transfer between each communicating task-pair. In this way, the placement of a failed task can be represented by the establishment of several file transfers. To be specific, relaunching a failed task on a selected server is represented by initiating file transfers between the selected server and other servers hosting the tasks that are communicating with the failed task. We use the *Pipe Viewer* (`pv`) tool to measure the flow completion time of each file transfer. If the completion time of all flows of an application have been measured, the performance of this application can be obtained as the reciprocal of the slowest completion time.

In our scenario, suppose a failure occurs to *S0*, resulting in the failed tasks *A1, B1* and *C3* to be relaunched. Assume the amount of data to be sent by each flow of *A1, B1* and *C3* are $60, 80$ and 100 MB, respectively, and other normal flows of application *A, B, C* and all flows of *D, E* have $30, 40, 50, 60, 70$ MB data to be sent respectively. We evaluate the performance of our algorithm for jointly placing these failed tasks and allocating bandwidth with the constraint of performance-centric fairness in two cases: *Case I)* $w_A : w_B : w_C : w_D : w_E = 1 : 1 : 1 : 1 : 1$. *Case II)* $w_A : w_B : w_C : w_D : w_E = 1 : 2 : 1 : 1 : 1$.

Intuition of the Jointly Optimal Strategy. The optimal solution of our joint task placement and performance-centric bandwidth allocation is presented in Fig. 6 and 10 for *Case*

I, and Fig. 7 and 11 for *Case II*, respectively. The characteristic inherited with an optimal placement is that with it, the maximum of the final *relative standard bandwidth demand* (*rSBD-n*) for all servers achieves the minimum. For *Case I* shown by Fig. 6, the optimal S^* is calculated as $1/4.8$, so that the transfer time achieved by each application is $1/(w^{'}S^*) = 24$s. For *Case II* shown by Fig. 7, since the weight of *B* becomes larger, the *rSBD-t* for *B1* becomes larger. Swapping the placement of *A1* and *B1* in *Case I* will result in smaller *rSBD-n* for all servers. The optimal S^* in this case is $1/5$, so that the optimal transfer times achieved by *A, C, D* is $1/(1/6 \cdot S^*) = 30$s, while the optimal transfer time for *B* is $1/(2/6 \cdot S^*) = 15$s. The analysis of the optimal solution with respect to bandwidth allocation will be presented next, compared with per-flow fair allocation.

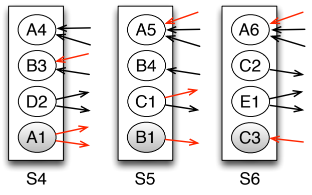

Fig. 6: Optimal placement when applications have the same weight.

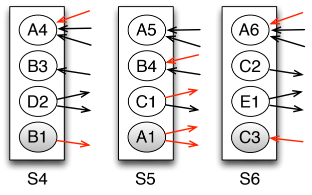

Fig. 7: Optimal placement when application have different weights.

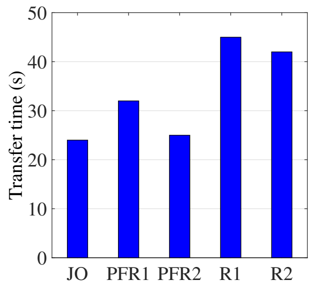

Fig. 8: Comparison among different placement strategies when applications have the same weight.

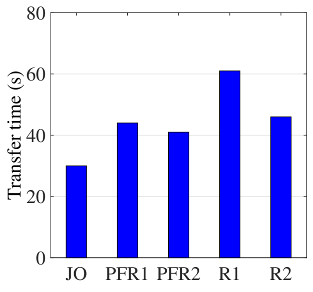

Fig. 9: Comparison among different placement strategies when applications have different weights.

Comparison among Different Strategies. As our strategy is the first to jointly optimize the two dimensional decisions of both bandwidth allocation and task placement, we evaluate it in comparison with the following two groups of one dimensional strategies, respectively:

Different placement, with the same bandwidth allocation. We compare the transfer time achieved by application *A* among the following strategies, which employ the same performance-centric bandwidth allocation, yet use different task placements: 1) Our Joint Optimization (Algorithm 2), denoted as *JO*, which is to first use Preliminary Filtering to select a set of candidate, and then use Branch-and-Cut to find the optimum. 2) Preliminary Filtering (Algorithm 1) and Random Placement, denoted as *PFR*, which randomly selects a placement in a reduced solution space given by Preliminary Filtering. 3) Random Placement, denoted as *R*, which randomly chooses among all feasible servers to relaunch the failed tasks.

As shown in Fig. 8, for *Case I*, the strategy of *JO* achieves the optimal transfer time, which is 24s aforementioned. In comparison, the placement decisions made by strategy *R*, with *R1* and *R2* as two runs, result in bad transfer times as large as about 45s. The underlying reason is that the failed task may be randomly placed on a server which is already heavy-loaded, with a large *rSBD-n*. The application performance achieved by *PRF* strategy (*PFR1* and *PFR2*) are in between *JO* and *R*. This is because Preliminary Filtering can filter out a set of promising candidate with good possible placement. With good placement covered in a reduced solution space, *PFR* is more likely to choose a fairly good placement compared with strategy *R* that randomly chooses among the entire solution space. The same analysis applies to Fig. 9 when the strategies of placement are compared for *Case II*.

Different bandwidth allocation, with the same placement. We further compare our performance-centric bandwidth allocation with the standard per-flow fair allocation, given the same placement of failed tasks, in order to illustrate how the performance-centric fairness in bandwidth allocation helps to alleviate the adverse effects of failures. With the failed tasks relaunched as what is shown in Fig. 6 for *Case I* and in Fig. 7 for *Case II*, we compare the completion time of application flows with per-flow fairness and performance-centric fairness, respectively.

As shown in Fig. 10, for *Case I* where applications have the same weight, the bandwidth allocation following per-flow fairness will result in different completion times of flows within the same application. In comparison, all flows finish at the same time if the bandwidth is allocated with performance-centric fairness. For application *A*, although flows of task *A3* finish faster than flows of task *A1* if the per-flow fairness is to be maintained, the performance of *A* is determined by slowest flows of *A1*, which still achieves the same performance as with performance-centric fairness. For application *C*, per-flow fairness will result in flows of the failed task *C3* as stragglers, which require 42s to finish, almost twice the completion time of other flows. In contrast, with performance-centric fairness, the adverse effects will be spread among all applications, so that stragglers no longer exist, and all applications enjoy the performance proportional to their weights.

Fig. 11 shows the comparison between different fairness notions in *Case II*. With performance-centric fairness, the failed task *C3* will no longer lag behind other flows, and the performance achieved by application *B* is better than other applications, since it has a larger weight. In the figure, we can see that the completion time achieved by *B* with the weight of 2 is half the completion time achieved by other applications with the weight of 1, satisfying the weighted performance-centric fairness.

B. Large Scale Simulations

We now investigate the behavior of our joint optimization strategy through large scale simulations in a general setting. To be particular, we focus on the following aspects: the ability of minimizing adverse effects caused by failures, and the running time overhead incurred by the algorithm.

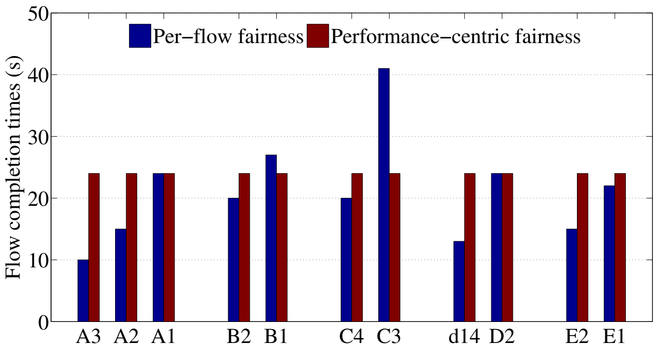

Fig. 10: Comparison between different notions of fairness when applications have the same weight.

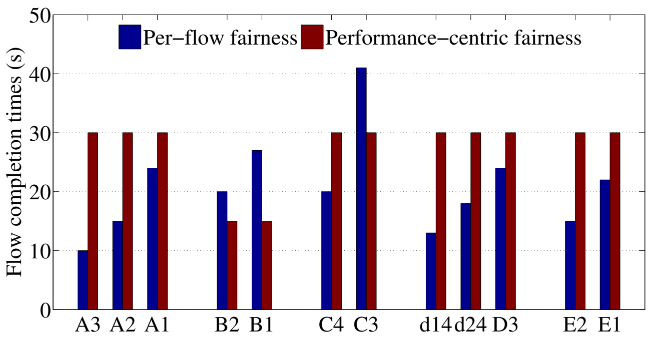

Fig. 11: Comparison between different notions of fairness when application have different weights.

Minimizing Adverse Effects Caused by Failures. We simulate a large scale datacenter where 100 data parallel applications, each with 40 tasks, are hosted by 1000 servers of the same capacity. We randomly set the placement of existing tasks and the amounts of data to be sent by each flow. As a result of host failures, 20 tasks need to be relaunched, to retransmit data. All the applications are assigned the same weight[5], without loss of generality. Similar to our previous Mininet evaluation, we compare our jointly optimal strategy to two groups of strategies as follows.

The comparison of our joint optimization strategy *JO* with other strategies (same bandwidth allocation yet different task placement) — *R*, *PFRb* and *PFRs* — over 10 runs is shown in Fig. 12. As previously defined, strategy *R* represents the Random Placement over the whole solution space, and *PFR* denotes the Random Placement over the decreased space resulted from Preliminary Filtering. *PFRb* and *PFRs* are *PFR* strategies with different parameter settings, the former of which selects a larger candidate set ($\eta = 25$), while the latter indicates a smaller space ($\eta = 15$) for the random placement. As shown in Fig. 12, *JO* achieves the best application performance, in that the transfer time is the lowest. *PFRb* and *PFRs* are both better than *R* because of using Preliminary Filtering. *PFRb* is worse than *PFRs*, since a larger space would contain more bad placement, which will make it less likely to choose a good one at random.

Given the same task placement, Fig. 13 shows the performance achieved by tasks on a server, when performance-centric fairness and per-flow fairness are to be maintained respectively in bandwidth allocation. *FT* denotes the failed task that is relaunched, while *ET1*, ... , *ET4* represent the normal running tasks. With performance-centric fairness, the adverse effect to the unlucky failed task is spread among all the applications. Compared with per-flow fairness which results in *FT* lagging behind, as slow as 15s, performance-centric fairness guides the reallocation of bandwidth so that all applications share and thus minimize the adverse effects caused by failures.

Algorithm Running Time. Finally, we evaluate the running time of our algorithm for a simulated datacenter, with 500 servers hosting 10 applications, each with 20 tasks. There are 8 tasks to be relaunched as a result of host failures. Our algorithm is implemented on a 1.86GHz dual-core Intel Core 2 server. Without using Preliminary Filtering, it takes about 5 seconds for the Branch-and-Cut algorithm to find the optimum over the whole solution space, as shown by Fig. 14, the empirical CDF of the running times for 100 runs. Fig. 15 plots the empirical CDFs of the running times when Preliminary Filtering is used, with the size of candidate servers tuned to be 250, 100 and 50 respectively. As we can see, the decrease of the candidate size due to Preliminary Filtering can achieve speedup of the algorithm.

[5]Analysis for the case of different weights is similar, as in our previous Mininet testbed evaluation.

VI. Related Work

Bandwidth allocation among multiple tenants in public cloud datacenters has received a substantial amount of recent research attention [17]–[23]. The general focus of these works has been on ensuring fair allocation among different tenants according to their payments. For example, NetShare [17] achieves tenant level fairness while Seawall [18] achieves fairness between VM sources. FairCloud [19] allocates bandwidth on congested links based on the weights of the communicating VM-pairs, thus achieving VM-pair level fairness. However, in our setting of a private datacenter running data parallel frameworks, the previous notion of fairness is not applicable.

In the context of a private datacenter, Kumar *et al.* proposed that bandwidth should be allocated with the awareness of the communication patterns of data parallel applications [24]. Their focus is mainly on effective parallelization for each application, *i.e.*, the completion time should be N times faster if the application parallelizes by N. However, when tasks of one application share bandwidth with tasks of different applications at different bottlenecks, it is not known what performance each application should expect, without a clear definition of fairness with respect to application performance. In contrast, our previous work [5] proposed the performance-centric fairness and offered a definitive guide to the problem of bandwidth allocation among multiple data parallel applications in a private datacenter. In this paper, we investigate how such kind of fairness is coupled with task placement when failures occur in the datacenter. To our best knowledge, we are the first to jointly consider the placement of failed tasks and the performance-centric bandwidth allocation in order to minimize the adverse effects of failures.

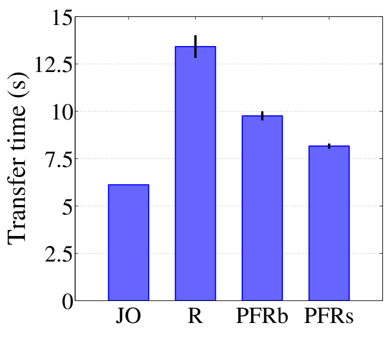

Fig. 12: Comparison among different placement strategies.

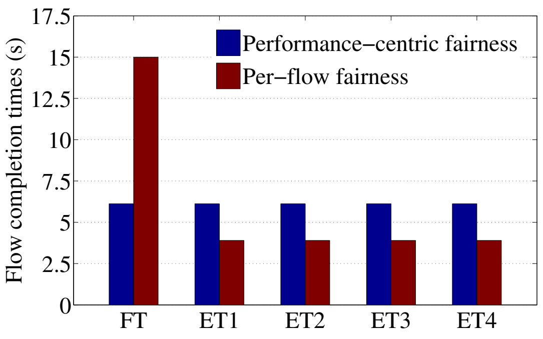

Fig. 13: Comparison between different notions of fairness (applications have the same weight).

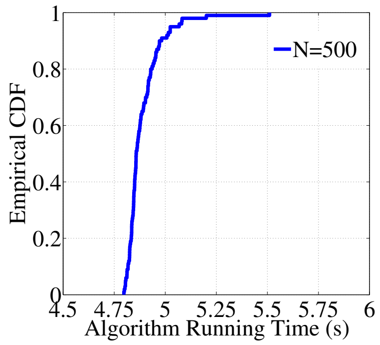

Fig. 14: Empirical CDF of algorithm running times over the whole solution space (with 500 candidate servers).

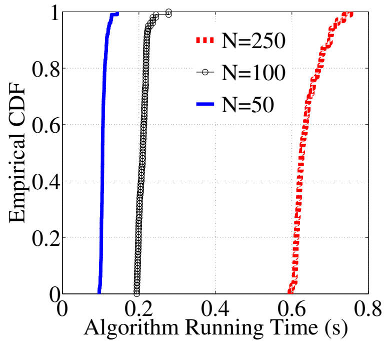

Fig. 15: Empirical CDFs of algorithm running times when tuning the Preliminary Filtering to obtain 250, 100, 50 candidate servers respectively.

VII. Concluding Remarks

Our focus in this paper is to study the placement of failed tasks and the sharing of link bandwidth among applications running data parallel frameworks in a private datacenter, with the presence of failures. We argue that even with the presence of failures, the performance achieved by applications should be proportional to their weights, which is defined as performance-centric fairness for competing applications. With a clear motivation for joint optimization of task placement and bandwidth allocation, we formulate the problem and present our algorithm to solve it efficiently. Our experimental results and extensive simulations have shown that our joint task placement and performance-centric bandwidth allocation strategy can efficiently minimize the adverse effect of failures and eliminate the straggler problem caused by failures.

References

[1] A. Langville and C. Meyer, *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, 2006.

[2] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-Scale Parallel Collaborative Filtering for The Netflix Prize," in *Algorithmic Aspects in Information and Management*. Springer, 2008, pp. 337–348.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 59–72, 2007.

[5] L. Chen, Y. Feng, B. Li, and B. Li, "Towards Performance-Centric Fairness in Datacenter Networks," in *Proc. IEEE INFOCOM*, 2014.

[6] P. Gill, N. Jain, and N. Nagappan, "Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 350–361, 2011.

[7] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the Outliers in Map-Reduce Clusters using Mantri," in *Proc. USENIX Operating Systems Design and Implementation (OSDI)*, 2010.

[8] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing Data Transfers in Computer Clusters with Orchestra," in *Proc. ACM SIGCOMM*, 2011.

[9] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving Datacenter Performance and Robustness with Multipath TCP," in *Proc. ACM SIGCOMM*, 2011.

[10] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," in *Proc. ACM SIGCOMM*, 2009.

[11] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta, "Making Cloud Intermediate Data FaultTolerant," in *Proc. ACM Symposium on Cloud Computing (SOCC)*, 2010.

[12] M. R. Bussieck and A. Pruessner, "Mixed-Integer Nonlinear Programming," *SIAG/OPT Newsletter: Views & News*, vol. 14, no. 1, pp. 19–22, 2003.

[13] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman and Company, New York, 1979, vol. 174.

[14] J. E. Mitchell, "Branch-and-Cut Algorithms for Combinatorial Optimization Problems," *Handbook of Applied Optimization*, pp. 65–77, 2002.

[15] B. Hindman, A. Konwinski, M. Zaharia, and et al., "Mesos: A Platform for Fine-Grained Resource Sharing in The Data Center," in *Proc. USENIX Networked Systems Design and Implementation (NSDI)*, 2011.

[16] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible Network Experiments Using Container-Based Emulation," in *Proc. 8th ACM CoNEXT*, 2012.

[17] T. Lam, S. Radhakrishnan, A. Vahdat, and G. Varghese, "NetShare: Virtualizing Data Center Networks across Services," UCSD-CSE, Tech. Rep. CS2010-0957, May 2010.

[18] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha, "Sharing the Data Center Network," in *Proc. USENIX NSDI*, 2011.

[19] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "FairCloud: Sharing the Network in Cloud Computing," in *Proc. ACM SIGCOMM*, 2012.

[20] J. Guo, F. Liu, D. Zeng, J. Lui, and H. Jin, "A Cooperative Game Based Allocation for Sharing Data Center Networks," in *Proc. IEEE INFOCOM*, 2013.

[21] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg, "EyeQ: Practical Network Performance Isolation at the Edge," in *Proc. USENIX NSDI*, 2013.

[22] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. OShea, "Chatty Tenants and the Cloud Network Sharing Problem," in *Proc. USENIX NSDI*, 2013.

[23] L. Popa, P. Yalagandula, S. Banerjee, and J. Mogul, "ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing," in *Proc. ACM SIGCOMM*, 2013.

[24] G. Kumar, M. Chowdhury, S. Ratnasamy, and I. Stoica, "A Case for Performance-Centric Network Allocation," in *Proc. ACM SIGCOMM HotCloud Workshop*, 2012.