Approaches for Resilience Against Cascading Failures in Cloud Datacenters

Haoyu Wang, Haiying Shen and Zhuozhao Li
Department of Electrical and Computer Engineering
University of Virginia
Email: {hw8c, hs6ms, zl5uq}@virginia.edu

***Abstract*—In a modern cloud datacenter, a cascading failure will cause many Service Level Objective (SLO) violations. In a cascading failure, when a set of physical machines (PMs) in a failure domain are failed, their workloads are transferred to the PMs in another failure domain to continue. However, the new domain receiving additional workloads may become overloaded due to the resource oversubscription feature in the cloud, which easily leads to domain failures and subsequent workload transfer to other domains. This process repeats and a cascading failure is created finally. However, few previous methods can effectively handle the cascading failures. To handle this problem, we propose a Cascading Failure Resilience System (*CFRS*), which incorporates three methods: *Overload-Avoidance VM Reassignment (OAVR)*, *VM backup set placement (VMset)* and *Dynamic Oversubscription Ratio Adjustment (DOA)*. The experiments in trace-driven simulation show that *CFRS* outperforms other comparison methods in terms of the number of domain failures, the number of failed PMs and the number of SLO violations.**

I. Introduction

A modern cloud datacenter, with thousands of servers and switches, hosts countless virtual machines (VMs) [1, 2]. Since the VMs are generally not fully utilized, the cloud providers often oversubscribe their datacenters to increase resource utilization and to maximize their profits. Oversubscription means that the total resource capacities (e.g., CPU and memory) of the VMs placed on a physical machine (PM) are often greater than the resource capacity of the PM. When a PM is oversubscribed, it is more likely to be overloaded (the resource utilization exceeds its capacity), which makes it tend to fail. For example, long-term higher CPU utilization can increase the CPU failure rate [3], and overwhelmed network traffic may lead to failure [4, 5].

In addition to resource oversubscription, there are other common causes for datacenter failures [6] including cluster power outages, workload-triggered software bug manifestations, Denial-of-Service attacks. For example, in power outages, a non-negligible percentage (0.5%-1%) of nodes do not come back to life after power is restored [7]. Cluster failures have been documented by Yahoo! [8], LinkedIn [9] and Facebook [10]. All of these failures may cause cascading failures.

A *failure domain* is a set of machines affected by a failure such as that from power nodes or network components [11]. When a failure domain fails, its VMs will be assigned to another failure domain (say domain B). Due to resource oversubscription, domain B may end up accepting more VMs than what its physical capacity can handle and fails, so its VMs must be reassigned to another failure domain, say domain C. Then, domain C fails and the same process repeats. As a result, VMs are pouring from a domain to another domain, pushing resource utilization to the limits and causing many domain failures and SLO violations. Finally, a cascading failure is created, adversely affecting the entire cloud system. To avoid such a cascading failure, we must avoid domain overloads (hence failures) while maximizing resource utilization in the workload transfers [12–16].

Previous works for handling the overload or failures in the cloud can be classified into three categories: VM migration, VM backup and failure mitigation (i.e., taking actions that alleviate the symptoms of a failure). In the VM migration methods [17–21], when a PM is overloaded in CPU, memory, or bandwidth, it migrates a set of VMs to other underloaded PMs. Many of these methods aim to eliminate overloaded PMs at a time point rather than a time period. It can avoid overloading PMs to a certain extent, but oversubscription and time-varying VM resource demands would still lead to overloaded PMs. The proposed VM backup methods [22, 11, 23, 24] are mainly about determining the number of backups and (or) the VM backup placement according to the different failure probabilities of failure domains or PMs to increase the reliability. These methods can mitigate the adverse influence from a failure but cannot handle concurrent failures [25] or the cascading failures described above, in which the VM backups may also fail. For failure mitigation, previous studies [26–28] generally follow the three-step procedure: 1) detection, 2) diagnosis, and 3) repair. Both the diagnosis and repair actions often take time, since the sources of failures vary widely. For instance, in February 2017, a failure in Amazon's AWS service impaired the operations of many cloud services for almost 5 hours [29]. Diagnosis and repair may require the system operators' assistance, further lengthening the failure recovery time. The worst 0.09% of failures can take more than 10 days to resolve since this kind of failures cannot be solved with simple actions (e.g., restart PM or router) [30], and 5% failures cannot be fixed within one day [31]. Therefore, for failures especially cascading failures, avoiding failures in advance is very important.

To handle cascading failure while addressing the aforementioned problems, in this paper, we propose a Cascading Failure Resilience System (*CFRS*) for cloud datacenters.

CFRS incorporates three methods:

1. **Overload-Avoidance VM Reassignment (OAVR)**. To avoid PM overload and hence possible resultant failure, *OAVR* tries to maintain the load balanced state (i.e., neither overloaded nor underloaded) of each PM in a long term. When a PM is predicted to be overloaded, it migrates out VMs that contribute more workload when it is overloaded and contribute less workload when it is underloaded in different epochs during a time period. When choosing a destination PM to host a migration VM from a failed or an overloaded PM, *OAVR* chooses the PM so that the VM contributes less workload when the PM is more overloaded and contributes more workload when the PM is more underloaded in different epochs during a time period. As a result, *OAVR* helps maintain the load balanced state of the PM in a long term, which avoids the possible overload caused by VM migration after failures hence avoids causing cascading failures.

2. **VM backup set placement (VMset)**. *VMset* increases the VM backup reliability by limiting the number of PMs that store the backups of all the VMs running on the same PM. We define a vmset as a set of PMs that store all the backups of one VM. *VMset* places all VM backups on a much smaller size of vmset group compared with previous VM backup methods [32–34]. It can highly reduce the probability that a VM's backups are stored on concurrent or cascading failed PMs and then make the VM backups survive during concurrent and cascading failures.

3. **Dynamic Oversubscription Ratio Adjustment (DOA)**. Rather than using a fixed oversubscription ratio [35] that easily generates PM overload due to time-varying resource demand, *DOA* dynamically adjusts the oversubscription ratio of a PM based on the resource utilization of the PM. The adjusted oversubscription ratio can reduce the probability of PM overload and then failure.

This paper is organized as follow. Section II introduces the previous work about failure avoidance and mitigation. In Section III, we present *CFRS* containing *OAVR*, *VMset*, and *DOA* in detail. In Section IV, we evaluate the performance of our methods using trace-driven simulation. Finally, in Section V, we conclude our paper with remarks on future work.

II. Related Work

We classify all the previous works into three parts: VM migration, VM backup and failure mitigation.

VM migration Some previous methods [17–21] aim to prevent the resource utilization of each PM from exceeding its capacity. For instance, Zhang *et al.* [17] proposed a VM allocation framework called *Venice* for high VM reliability. It assigns high availability scores to the VMs with high availability requirement and lower resource requirement set by users and assigns high availability scores to the PMs with lower real-time resource utilization and lower hardware failure rate. When assigning a VM to a PM, it randomly selects one PM from the PMs that have greater availability scores than the availability score of the VM. Bodik *et al.* [18] proposed a VM allocation scheme that spreads out VMs across multiple failure-domains while minimizing the total bandwidth consumption in order to improve VM survivability. Bila *et al.* [19] proposed a partial VM migration technique, which only migrates a part of a VM to save the network load while the VM is idle. Zhang *et al.* [20] proposed a VM migration technique, in which when a PM is overloaded, only the VMs' footprints (containing the information in main memory that a VM uses or references while running) on this PM need to be transferred in order to improve the efficiency of VM migration. Sandpiper *et al.* [21] carries out dynamic monitoring the workload on PMs and identifies overloaded PMs. The VM with a larger volume-to-size ratio (VSR) has a higher priority to be migrated out, where the size is the size of the memory footprint of the VM and the volume is the product of VM's resource utilizations (i.e., $cpu \times mem$). However, these methods try to eliminate the overloaded PMs at a time point rather than a time period. Meanwhile, VM migration can avoid failures caused by overload but cannot recover VMs after they fail (e.g., using VM backups).

VM backup The VM backup method is widely used to provide reliability. Several previous works [22, 11, 23, 24] proposed methods to determine the number of backups and (or) the VM backup placement according to the different failure probabilities of PMs in different domains to increase the reliability and reduce the possibility of backup loss. Xu *et al.* [22] proposed a polynomial-time algorithm for the placement of VMs and their backups. For each possible mapping decision of all the VMs and backups, this algorithm selects the placement schedule subject to the constraints that each VM's resource requirements are satisfied before and after any single PM failure. *DieHard* [11] targets on increasing the VM backups reliability during failures and also trying to minimize the number of backups required to maintain the VM running during failures by mapping most of the backups in more reliable domains. Cirne *et al.* [23] proposed a method that attempts to place the VMs and their backups on different racks to enhance reliability with the assumption that the failure probability of each rack is independent. Yeow *et al.* [24] provided a technique for estimating the number of VM backups required to achieve the desired reliability objectives. However, these methods cannot handle the concurrent failures or cascading failures, in which a VM's backups may also be lost along with the VM itself.

Failure mitigation Previous methods [26–28] aim to mitigate the effect of failures by finding the failure reasons and then repair the failures after many diagnostic trials. Isard *et al.* [26] proposed an automated server management system called Autopilot based on the concept of recovery oriented computing. When Autopilot detects that a server is misbehaving (e.g., no response or wrong feedback), it either restarts or sends all data to other PMs. To mitigate the cloud network failures, *Netpilot* [27] first identifies several failure reasons that are likely to cause the occurred failure and iteratively takes mitigation actions for each reason until the problem is alleviated. R3 [28] is a recovery service that can quickly mitigate the influence of link failures. It pre-computes

forwarding table which stores all the information (e.g., the start and destination of the link) of each link, so that once a link fails, a link can be recreated with the same source and destination according to the table. However, since the cost of failure repair is very high, a better way to handle failures is avoiding the failure occurrence rather than repairing it after it happens.

III. Design of the Cascading Failure Resilience System (CFRS)

In the following, we introduce each component of *CFRS* which aims to avoid the probability of cascading failure occurrence in cloud datacenters. Table I shows the meanings of major notations used in this paper.

A. Overload-Avoidance VM Reassignment (OAVR)

1) Overview of OAVR*:* In a cascading failure, when a failure domain is overloaded, to reallocate the VMs hosted in the domain, the master machine can check the available resources of other failure domains to make sure that the selected destination domain won't be overloaded. For fast VM reallocation, the master machine does not have to check the available resources of other failure domains, and can just randomly select a destination domain, which however increases the probability that the selected domain becomes overloaded and fails. For the former approach, we need to make sure that the destination domain will not be overloaded not only at the current time but also in long term. For the latter approach, we let each PM periodically check its load status and migrate out VMs once it is overloaded with the objective of achieving long-term load balance. Both are for potentially avoiding cascading failures. Accordingly, we propose *OAVR* that selects VMs from an overloaded PM to achieve long-term load balance in the PM and selects destination PMs to allocate the selected VMs to achieve long-term load balance in the destination PMs.

Many previous studies [36–38] show that the VM workload often has predictable resource utilization (e.g., CPU and memory) based on historical logs. *OAVR* leverages this property to decide the VM placement. For the VMs that have been run previously or periodically, the hosting PMs are notified about the workloads of the VMs according to the historical logs. For the VMs that have not run previously, the maximum (marked) resource capacities of the VMs are notified to the PMs. A time period T consists of several epochs (denoted by $e_1, e_2, ..., e_k, ..., e_n$). We suppose there are Q types of resources. If a PM is overloaded on at least one resource type and in at least an epoch within time period T, we consider it overloaded during T and then its excess workload needs to be released to make this PM underloaded. Each PM periodically checks if it will become overloaded in the next time period T based on the workload information of the VMs placed on the PM. If so, the PM migrates out the VMs contributing more workloads in epochs when it is overloaded and less workloads in epochs when it is unloaded to release its extra workload. When determining the destination PM that each selected VM migrates to, *OAVR* selects the PM that is more underloaded when the VM introduces more workload and is more overloaded when the VM introduces less workload in different epochs. Finally, the long-term load balance state in the source PM and the destination PM can be achieved; that is, each PM is less likely to be overloaded or underloaded during time period T.

TABLE I: Notations.

P_i	the i^{th} PM	v_j	the j^{th} VM
V_{P_i}	the set of VMs running on P_i	$C_q^{P_i}$	P_i's capacity of the q^{th} resource
$U_{q,e_k}^{P_i}$	P_i's unbalanced workload of the q^{th} resource in e_k	$S_{P_i}^{v_j}$	priority score between v_j and P_i
$l_{q,e_k}^{v_j}$	v_j's workload of the q^{th} resource in e_k	$L_{q,e_k}^{P_i}$	P_i's workload of the q^{th} resource in e_k

2) VM Reallocation: In order to achieve the long-term load balance, in *OAVR*, each PM aims to balance the workload at each epoch in the next period T. Since there are many resources involved in VM running, a PM needs to evaluate the workload of itself and each of its VMs on each resource type.

We normalize the workload of VMs of a PM and the PM on the q^{th} resource by the PM's capacity on the q^{th} resource (denoted by $C_q^{P_i}$). That is, $C_q^{P_i}$ always equals to 1. The workload of VM v_j (hosted in PM P_i) on the q^{th} resource in the k^{th} epoch, denoted by $l_{q,e_k}^{v_j}$, is represented by the ratio between the v_j's demand in the k^{th} epoch and the capacity of PM P_i on the q^{th} resource (e.g., VM v_j's occupied memory size over the total memory size of PM P_i). We use V_{P_i} to denote the set of VMs running in PM P_i. Then, PM P_i's workload on the q^{th} resource in epoch e_k equals the sum of all the workloads of VMs $v_j \in V_{P_i}$:

$$L_{q,e_k}^{P_i} = \sum_{v_j \in V_{P_i}} l_{q,e_k}^{v_j} \tag{1}$$

For PM P_i, among all types of resources, if one resource type satisfies $L_{q,e_k}^{P_i} > C_q^{P_i}$ at an epoch $e_k \in T$, P_i is overloaded on the q^{th} resource type at epoch e_k and we regard P_i as an overloaded PM in T. If all types of resources satisfy $L_{q,e_k}^{P_i} < C_q^{P_i}$, we regard P_i as an underloaded PM at epoch e_k. For the master machine to schedule VM reallocation, each PM P_i reports its workload to the master machine at each epoch as $\{L_{q,e_1}^{P_i}, L_{q,e_2}^{P_i}, ..., L_{q,e_n}^{P_i}\}$ and its capacity in each resource type $C_q^{P_i}$ $(q = 1, 2, ...Q)$. Also, each overloaded PM needs to select migration VM v_j (details are in next section) to release its excess workload and report the workload of the VM at each epoch as $\{l_{q,e_1}^{v_j}, l_{q,e_2}^{v_j}, ..., l_{q,e_n}^{v_j}\}$. We present how an overloaded PM selects migration VMs below.

For each resource type, during a time period T, a PM may be overloaded on certain types of resources in some epochs while be underloaded on certain types of resources in other epochs. Therefore, when an overloaded PM in T determines which VMs to migrate out, it should give priority to the VMs that contribute more workloads on those resources when the PM is overloaded on the resources and contribute less

workloads on those resources when the PM is underloaded on the resources. This way, the extra workload on an overloaded PM can be reallocated faster and more efficiently, and also the resources on the PM can be more fully utilized. Below, we introduce how to calculate such priority of VMs in an overloaded PM in T.

For the q^{th} resource type on an overloaded PM P_i at epoch e_k, we define its unbalanced workload $U_{q,e_k}^{P_i}$ as the workload on the q^{th} resource at epoch e_k ($L_{q,e_k}^{P_i}$) minus P_i's resource capacity on the q^{th} resource ($C_q^{P_i}$):

$$U_{q,e_k}^{P_i} = \begin{cases} L_{q,e_k}^{P_i} - C_q^{P_i} & \text{if } L_{q,e_k}^{P_i} \neq C_q^{P_i} \\ -c & \text{otherwise} \end{cases} \quad (2)$$

where c is a constant. We set it to $-c$ instead of 0 in order to set a lower priority for fully utilized resources since *OAVR* selects VMs from overloaded PMs to migrate out. When unbalanced workload $U_{q,e_k}^{P_i} > 0$, it means that P_i is overloaded at epoch e_k for the q^{th} resource type. A higher $U_{q,e_k}^{P_i}$ means a higher overloaded degree. In this case, a VM that has a higher $l_{q,e_k}^{v_j}$ should have a higher priority to migrate out. On the other hand, when unbalanced workload $U_{q,e_k}^{P_i} < 0$, it means that P_i is underloaded at epoch e_k for the q^{th} resource type. A lower $U_{q,e_k}^{P_i}$ means a higher underloaded degree. In this case, a VM that has a lower $l_{q,e_k}^{v_j}$ should have a higher priority to migrate out. Combining the two cases, for the q^{th} resource type, in order not to over-utilize or under-utilize the resource capacity of a PM during T, if a VM has a higher $\sum_{e_k \in T} U_{q,e_k}^{P_i} \cdot l_{q,e_k}^{v_j}$ value, it should have a higher priority to migrate out.

In the following, we introduce how to calculate the priority considering all types of resources. Recall that there are Q types of resources that need to consider. We use a $Q-$dimensional vector $\vec{l}_{e_k}^{v_j} =< l_{1,e_k}^{v_j}, ...l_{q,e_k}^{v_j}, ...l_{Q,e_k}^{v_j} >$ to represent the workload of VM v_j at epoch e_k for all the types of resources, and use $\vec{U}_{e_k}^{P_i} =< U_{1,e_k}^{P_i}, ...U_{q,e_k}^{P_i}, ...u_{Q,e_k}^{P_i} >$ to represent the unbalanced workload of PM P_i at epoch e_k for all the types of resources. For an overloaded PM (with any $U_{q,e_k}^{P_i} > 0$), we tend to migrate the VMs which contribute more workloads on the overloaded resources and contribute less workloads on the underloaded resources. Thus, we define the VM migration priority score of VM v_j on PM P_i in the period T (denoted by $S_{P_i}^{v_j}$) as the dot product of $\vec{U}_{e_k}^{P_i}$ and $\vec{l}_{e_k}^{v_j}$:

$$S_{P_i}^{v_j} = \sum_{e_k \in T} \vec{U}_{e_k}^{P_i} \bullet \vec{l}_{e_k}^{v_j}. \quad (3)$$

According to this equation, the VM with a larger $S_{P_i}^{v_j}$ has a higher priority to be reallocated. That is, the VMs that contribute more workload for the overloaded resource types and contribute less workloads on the underloaded resource types on an overloaded PM in different epochs during T have higher priorities to be selected and then reallocated. PM P_i sorts its VMs in the descending order of their priority scores. It then selects the VMs from the top of the list one by one and removes these selected VMs from the list until P_i becomes non-overloaded in each epoch within the time period T for each resource type. Next, PM P_i reports all the selected VMs to the master machine. The master machine will send the VM reallocation schedule to PM P_i and then P_i migrates the VMs accordingly. As a result, the long-term load balanced state of different types of resources on the PMs can be maintained.

3) Destination PM Selection: Next, we introduce how the master machine determines the destination PM for each VM that needs to be reassigned from its host PM or failed PM. When the master machine checks the resource utilization of the destination PMs, in VM reassignment scheduling, a number of rules need to follow. First, in order to increase the possibility of achieving the load balanced state for each PM in the system, VMs with higher workloads should be scheduled first. Second, in order to avoid overloading the destination PMs and then avoid the cascading failure as much as possible, we should migrate VMs with the highest workload on some resource types to the most underloaded PMs on the resource types. Third, a VM should be migrated to its best-fit PM; that is, the priority score between the VM and the PM is the lowest among all PM candidates (since a higher priority score between a VM and a PM means a higher degree of dismatch between them and vice versa) or the VM contributes more workloads on some resource types when the PM is underloaded on these resource types and less workloads on some resource types when the PM is overloaded on these resource types at different epochs during T.

To follow the first rule above, after all the overloaded PMs report their selected VMs to the master machine, the master machine calculates the modulus of each VM workload vector as a measure of its overall workload combining all resource types during T. That is:

$$L^{v_j} = \sum_{e_k \in T} |\vec{l}_{e_k}^{v_j}|. \quad (4)$$

The master machine sorts these VMs in the descending order of their overall workloads L^{v_j} during T. Then, the master machine selects the VMs from the top of the sorted VM list one by one and finds a destination PM for each VM. *CFRS* employs the VM backup strategy [36]. For a selected VM v_j, to reduce the interruption to the VM running from VM migration, the master machine first finds the PMs that store the backups of VM v_j. If one of these PMs, say P_k, has enough available resources for VM v_j, the master machine notifies v_j's host PM P_i to transfer the current footprint of v_j to P_k so that VM v_j's backup can continue to run on P_k directly without the need of re-launching a new VM. The master machine then updates the workload of P_k at each epoch e_k during T by $L_{q,e_k}^{P_k} \leftarrow L_{q,e_k}^{P_k} + l_{q,e_k}^{v_j}$ and updates the workload of P_i at each epoch e_k during T by $L_{q,e_k}^{P_i} \leftarrow L_{q,e_k}^{P_i} - l_{q,e_k}^{v_j}$.

If all the PMs with the backups of v_j cannot provide enough resources for v_j, to follow the third rule above, the master machine then calculates the priority scores of v_j with regard to every other PM that has enough resource for v_j in the next period T according to Formula (3) and selects the PM with the lowest priority score. Specifically, the master machine sorts the PMs in ascending order of priority scores, and checks the PM from the top of list one by one until it finds a PM that has

enough available capacity to host the VM. Note that a lower resource utilization of a PM decreases the value of the priority score. Thus, the selected PM also has relatively low resource utilization and the second rule in the above is followed. For the picked PM P_i, the master machine checks whether P_i has enough resources for v_j in each epoch. If yes, the master machine reallocates v_j to P_i and updates the workload of P_i as described above. Once the VM v_j is migrated, the master machine removes v_j from the VM list and repeats the above process until all the VMs in the list are reallocated.

As a result, through the VM migrations from overloaded PMs to underloaded PMs introduced above, long-term load balanced state of PMs over the time period T can be achieved.

B. VM Backup Set Placement (VMset)

In a datacenter, for a VM hosted in a PM, creating several VM backups in other PMs can increase the resilience to PM failures [23, 11]. A VM's backups periodically receive footprints from the running VM [39, 40]. When a VM's host PM fails, its VM backup can continue running without the need of launching and restarting a new VM. Thus, the VM backup method reduces the interruption to the VM running caused by PM failures.

Correlated machine failures often occur in large-scale systems [41–43] due to common failure causes (e.g., cluster power outages, workload-triggered software bug manifestations, Denial-of-Service attacks). In the cascading failure, the failures of PMs are correlated (i.e., concurrent or fast sequential) since subsequent failures are caused by overload due to the transferred load from previous failures. Then, when a VM fails, all of its backups may also fail almost concurrently. It decreases the performance of the application of the VM, since a new VM needs to be submitted, allocated, and started over again. It may also exacerbate the cascading failure since many VMs need to be reallocated to the remaining alive PMs, which may make them overloaded and possibly cause subsequent PM failures. To avoid the concurrent (or fast sequential) failures of a VM and its backups, we propose *VMset* that determines the locations of VM backups to increase their reliability during concurrent failures and cascading failures. *VMset* makes it more likely to recover a failed running VM by its backups, thus enhances the performance of *OAVR* in avoiding cascading failures or mitigating the adverse influence of cascading failures. It can achieve higher VM backup reliability than previous VM backup methods such as the random backup algorithms [32, 33, 8, 44].

We use R (e.g., 3) to denote the number of backups of each VM, N to denote the total number of PMs in the datacenter and W to denote the *VM backup spread width*, which means that all the backups of the VMs running on the same PM can only be stored in a group of W PMs. We define a vmset as a set of W PMs that store all the backups of one VM. In the random backup algorithm, the primary or first VM backup is placed on a randomly selected PM from the entire datacenter. Assuming the first backup is placed on PM P_i, the remaining $R-1=2$ backups are placed in PMs randomly chosen from the PMs $\{i+1,\ i+2,\ ...\ ,\ i+W-1\}$. If $W=N-1$, the second backup's PMs are chosen randomly from all the PMs in the datacenter.

If we consider that each PM is independent to other PMs, the random backup algorithm can provide high durability. For example, considering only one VM and its three backups, if a concurrent failure leads to 1% of the PMs in the datacenter to fail, the probability that the failure causes the exact three PMs which store the VM backups to fail is only $(1\%)^3 = 0.0001\%$. In another word, the probability that these VM backups survive during the failure is $1-0.001\% = 99.9999\%$. However, the system replicates millions of VMs and needs to ensure that every single VM should survive during the failures even though each VM backup may be very safe. Considering a large number (e.g., 5000000) of VMs and their backups storing in the datacenter, when 1% of the PMs in the datacenter fail, the loss probability of all the backups for any one VM will be $1-(99.999\%)^{5000000} = 99.99\%$ rather than 0.0001% [7]. Therefore, the random backup algorithm cannot effectively handle the cascading failures or concurrent failures.

For instance, assume the datacenter has the following parameters: $R=3$, $N=12$, and $W=N-1=11$. When we use the random backup algorithm, the first VM backup is placed on a PM randomly chosen from 12 PMs, say PM P_1. The second and third backups are placed in PMs randomly chosen from all the other PMs in the datacenter. Thus, when we have a large number of VMs and their backups are stored in the entire datacenter, the total number of vmsets is:

$$\binom{12}{3} = 220. \tag{5}$$

Suppose that three PMs fail in the datacenter. Then the possibility of VM backup loss is the number of vmsets divided by the maximum number of sets:

$$\frac{\#\ \ vmsets}{\binom{N}{R}} = \frac{220}{\binom{12}{3}} = 100\%. \tag{6}$$

Using a lower spread width (W) can decrease the probability of VM backup loss from correlated failures. Suppose $W=4$ and the first VM backup is also placed on a PM randomly chosen from 12 PMs, say PM P_1. The second and third backups can be placed in PMs randomly chosen from subsequent PMs PM P_2, P_3, P_4 and P_5 due to the spread width $W=4$. Thus, when we have a large number of VMs and their backups are stored in the entire datacenter, the total number of vmsets is:

$$12 \cdot \binom{4}{2} = 72. \tag{7}$$

Suppose that three PMs fail in the datacenter. Then the possibility of VM backup loss is the number of vmsets divided by the maximum number of sets:

$$\frac{\#\ \ vmsets}{\binom{N}{R}} = \frac{72}{\binom{12}{3}} = 32.7\%. \tag{8}$$

Now, we introduce the *VMset* method that produces lower possibility of VM backup loss with the same W value. *VMset*

is developed based on the copyset algorithm [7] which is for data replica placement to increase the reliability of replicas during a concurrent failure. *VMset* provides high VM reliability by limiting the number of PMs that store the backups of all the VMs running on the same PM. Compared with the previous VM backup random placement methods, *VMset* places all VM backups on a much smaller size of vmset group to achieve lower probability of VM backup loss.

For instance, suppose we still have the same parameters for the datacenter. The first VM backup is also placed in a PM randomly chosen from 12 PMs. We only allow all the VMs to store their backups on the following 8 vmsets:

$$\{P_1, P_2, P_3\}\{P_4, P_5, P_6\}\{P_7, P_8, P_9\}\{P_{10}, P_{11}, P_{12}\},$$
$$\{P_1, P_5, P_9\}\{P_2, P_6, P_{10}\}\{P_3, P_7, P_{11}\}\{P_4, P_8, P_{12}\}, \quad (9)$$

That is, if the first backup is placed on PM P_2, the second and the third backups can only be randomly placed on PM P_1 and P_3 or P_6 and P_{10} as shown in Figure 1, which is different from the above backup placement that randomly places the two replicas on the other PMs chosen from 11 PMs ($W = 11$) or 4 sequent PMs ($W = 4$). Note that with *VMset*, the backups of VMs on the same PM are split uniformly to four other PMs (P_1, P_3, P_6, P_{10}), as the VM backup spread width $W = 4$.

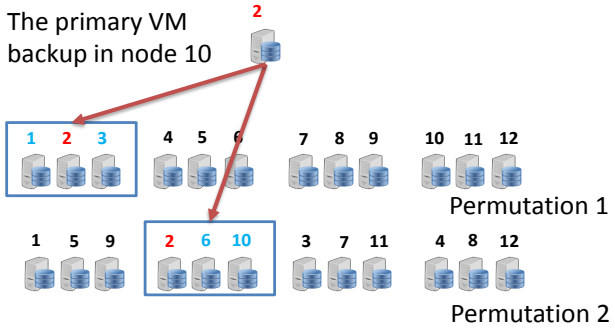

Fig. 1: Illustration of the *VMset* method.

This method creates only 8 vmsets. If three PMs fail concurrently, the probability of VM backup loss is:

$$\frac{\#\ VMsets}{\binom{N}{R}} = \frac{8}{\binom{12}{3}} = 4\%, \quad (10)$$

which significantly reduces the loss probability of VM backups of the random backup algorithm. With *VMset*, we will only lose VMs and backups if all the PMs in a vmset fail simultaneously. For example, choosing a system with $N = 5000$, $R = 3$, $W = 10$, when 1% of the PMs fail simultaneously, *VMset* yields the VM backup loss probability as:

$$\frac{\#\ VMsets}{\binom{N}{R}} = \frac{\frac{10}{3-1} \cdot \frac{5000}{3}}{\binom{5000}{3}} = \frac{8333}{2.01 \times 10^{10}} = 0.000042\%, \quad (11)$$

which is much lower than 99.99% in the random backup algorithm.

In the following, we introduce how to create the vmsets. We define a *permutation* as an ordered list of all PMs in the system. We perform $e = \frac{W}{R-1}$ permutations on the PMs in the datacenter. If the number e is not an integer, we only choose its integer part. The vmset generation needs to follow two rules. First, each vmset overlaps with each other vmset by at most one PM (e.g., the only overlapping PM of vmset $\{P_1, P_2, P_3\}$ and $\{P_1, P_5, P_9\}$ is PM P_1). This ensures that each vmset increases the spread width for its PMs by exactly $R - 1$. Second, the mechanism ensures that the vmset should cover all the PMs. In the above example, $W = 4$, $R = 3$ and $N = 12$. We have $e = 2$ permutations, say $\{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9, P_{10}, P_{11}, P_{12}\}$ and $\{P_1, P_5, P_9, P_2, P_6, P_{10}, P_3, P_7, P_{11}, P_4, P_8, P_{12}\}$ as shown in Figure 1. Each permutation form $\frac{N}{R}$ vmsets as shown in (9). Therefore, e permutations form $e \cdot \frac{N}{R} = \frac{W}{R-1} \cdot \frac{N}{R}$ vmsets. However, in the random backup algorithm, the number of vmsets is around $O(W^{(R-1)})$ [7].

In *VMset*, when the PMs in one vmset fail together, the VMs and backups will be lost. The above vmset creation method does not consider the different failure rates of different PMs in different failure domains to reduce the probability that the PMs in one vmset fail together. In order to increase the reliability of VMs and their backups further, we introduce a more advanced vmset creation method. Actually, we can generate $E = \frac{N-1}{R-1}$ (that is greater than e) permutations that follow the two rules indicated above. In the example above, we can generate 5 permutations (e.g., permutation $\{P_1, P_4, P_7, P_2, P_9, P_{11}, P_3, P_6, P_{12}, P_5, P_8, P_{10}\}$) that follow the two rules. Then, to create reliable vmsets, we select e permutations from E permutations with the smallest probability that all the PMs in one vmset fail together. Suppose PM P_i has a failure rate f_{P_i} (considering both the hardware failure probability and the failure rate of failure domain where this PM resides in), and then we calculate the concurrent failure probability of one vmset f_{vmset} as the product of the failure rates of PMs in this vmset:

$$f_{vmset} = \prod f_{P_i}. \quad (12)$$

Then, we calculate the failure probability of each permutation $F_{permutation}$ as the sum of concurrent failure probabilities of all the vmsets in the permutation:

$$F_{permutation} = \sum f_{vmset}. \quad (13)$$

VMset then sorts the permutations in descending order of their failure probabilities and and selects top e permutations to map all the VM backups according to the selected permutations. Therefore, the selected permutations can always achieve the lowest VM backup loss probability with the consideration of the different failure rates of different PMs.

Hence, the *VMset* method significantly reduces the probability of VM backup loss with much lower number of PMs involved in storing the backups of VMs running on the same PM, and then avoids VM reallocation which can reduce the probability of PM overload. The lower probability of PM overload can also reduce the possibility of cascading failure occurrence.

C. Dynamic Oversubscription Ratio Adjustment (DOA)

As mentioned in Section I, the cascading failure is caused by oversubscription on PMs and then the PMs and domain overloads due to workload migration during failures. We assume that each resource type has an oversubscription ratio. In order to avoid the cascading failures and decrease the failure possibility, we propose *Dynamic Oversubscription Ratio Adjustment* (*DOA*) to dynamically adjust the oversubscription ratio of each PM based upon the current resource utilization on the PM. *DOA* adaptively decreases the oversubscription ratio of a resource type when the resource is over-utilized and increases it when a resource is under-utilized.

In *DOA*, each PM monitors the resource utilization (e.g., CPU and memory) of each resource type of all the VMs running on itself. If the utilization of one resource on a PM is not in a pre-defined threshold range (e.g., 85%-100%), the PM changes the oversubscription ratio of this resource type. Next we introduce the details on how to dynamically adjust the oversubscription ratio based on the real-time workload on each PM. For simplicity, we use the memory resource of PMs as an example and this method can be for any resource type.

Suppose the memory capacity of a PM is C_M and the master machine can assign it VMs (whose total required memory resource is L_M) to reach the specified oversubscription ratio as $\frac{L_M}{C_M}$. At the beginning of each time period, the PM examines whether the memory resource utilization exceeds the pre-defined threshold range. If the utilization exceeds the higher bound of the range, the PM notifies the master machine and then the master machine reduces the maximum total memory of VMs that is allowed to allocate to the PM by $\frac{L_M}{a}$, where a (e.g., 2) is a small constant since we want to fast reduce the memory resource demand of VMs on this PM. Thus, it can reduce the overload occurrence probability of the PM caused by memory resource overload. If the utilization is lower than the lower bound of the threshold range, the master machine increases the maximum total memory of VMs that is allowed to assign to this PM from L_M to $L_M + \frac{C_M}{b}$ until the memory utilization exceeds the lower bound of the threshold range. Here, b (e.g., 10) is a large constant since we want to increase the resource demand of VMs that is allocated to this PM slowly, so that *DOA* can find a more suitable oversubscription ratio and reduce the overload occurrence possibility of PM efficiently. This way, *DOA* can maintain the resource utilization within the pre-defined threshold range as much as possible, which reduces the overload possibility of PMs. The oversubscription ratio adjustment here is similar to the window size adjustment in the TCP/IP sliding window protocol [45] in order to find an appropriate value.

Figure 2 shows a simple example of *DOA*. The number shown below each PM indicates the maximum size of VMs' memory that can be assigned to it. *DOA* examines the resource utilization periodically and changes the oversubscription ratio accordingly. For example, in period 1, the leftmost PM has a memory utilization of 100% and the maximum size of memory of VM that can be assigned to the PM is $8GB$.

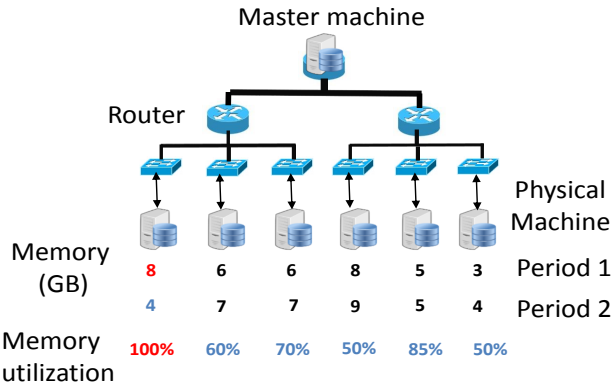

Fig. 2: Illustration of dynamic oversubscription ratio adjustment.

After *DOA* detects that the memory utilization on this PM exceeds the pre-defined threshold range (i.e., 100%>95%), in the next time period (period 2), the oversubscription ratio is reduced and then the maximum size of memory of VMs that can be assigned to this PM is reduced to 4. On the other hand, all the resource utilizations of other PMs are below 95% so that the maximum size of memory which can be assigned to the PMs are increased by $1GB$ at period 2. In this way, the oversubscription ratio of each resource type in a PM is dynamically adjusted according to the real-time resource utilization of the PM and will not exceed the higher bound or lower bound of the threshold range.

IV. Performance Evaluation

A. Simulation Setup

We conducted trace-driven simulation on a Java-based simulator to evaluate the performance of our proposed *CFRS*. We used the VM resource utilizations in CPU and memory from the Google Cluster trace [46, 47] to generate VM workloads. The Google Cluster trace records CPU and memory usage of VMs on a cluster of about 11000 machines from May 2011 for 29 days. We simulated a cloud datacenter, in which 19200 PMs are connected through 240 Top-of-Rack (ToR) switches and 80 PMs are in one rack, and each power station supplies 20 racks [11]. All the PMs in the datacenter are organized into 240 network failure domains and 12 power failure domains shown in Figure 3. The failure rate was randomly chosen from $[0.000022, 0.000032]$ per hour for a network failure domain and 0.4×10^{-6} per hour for a power failure domain [11]. For each overloaded PM, the failure rate is 0.0001 per minute to stop working and fail. When a PM fails and its VMs need to be reassigned to other PMs to run, in order to restart the VMs quickly, the capacities of the destination PMs for the VMs may not be checked as explained in Section III-A1. We set the probability of not ensuring sufficient capacity of destination PMs to 2%-15% as the X axis of the experimental result figures.

We configured the PMs in the datacenter with the capacities of Intel Xeon 6 CPU cores and 16 GB memory. We also configured each VM with the demand number of CPU cores randomly chosen from 1 to 4 and the demand memory randomly chosen from 1GB to 8GB like the Amazon EC2 instances (e.g., *t2.small, t2.medium*, and *t2.large*) [48]. The

default number of backups for each VM is 3. We repeatedly carried out each simulation for 10 times and reported the average results. Each PM conducts VM migration periodically every 60 seconds. When the simulation was started, the simulator updates the VM's resource utilization in the datacenter every 60 seconds, and records the number of the occurrences of overloaded PMs during the experiment period.

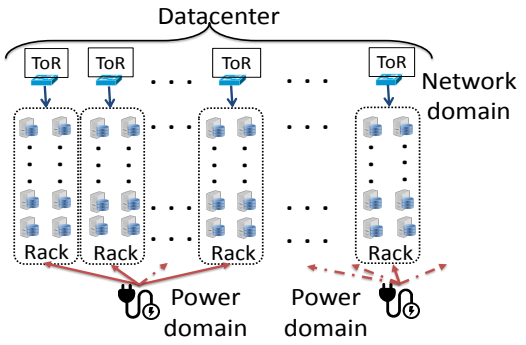

Fig. 3: A datacenter with failure domains.

We compare the performance of *CFRS* with three other methods, *Random*, *DieHard* [11] and *Venice* [17]. *Venice* does not have VM backup method and we add this method to *Venice* to make it comparable to other methods. The details of these methods are presented in Section II. In *Random* and *Venice*, the PMs to host VM backups are randomly chosen from all the PMs. In order to guarantee the fairness between the methods, since there is no VM migration in *Random* and *DieHard*, we add the periodical VM reassignment (i.e., load balancing) function (which randomly selects the migration VMs and destination PMs with enough capacity for migrated VMs) into them and represent these two enhanced methods by *Random** and *DieHard**. For all the four methods, when a PM, say P_i, is overloaded or failed, to recover each VM running in P_i, the backups of the VM will be used first; only when the backups are not available, a new VM is relaunched. The details are presented in Section III. The default oversubscription ratio is 2 for each method, and we set $a = 2$ and $b = 10$ in *CFRS*. The default number of backups in the simulation is 3 and only *DieHard* determines the number of backups for each VM based on a strategy that we will explain later. *DieHard* and *Venice* need the users to set VMs' reliability requirement to determine the minimum number of backups (for *DieHard*) or calculate the availability score of VM (for *Venice*). Then we set the reliability requirement as that one VM can get its required resource in 95% of its running time [17].

B. Evaluation Metric Description

Each simulation ran for 24 hours simulation time. We measured the following metrics after each simulation.

1. *The number of domain failures.* In a cascading failure, a domain fails and then its workload is moved to other domain, which becomes overloaded and fails, and this process repeats, leading to a cascading failure. In the experiment, when 95% PMs within one failure domain are failed by overload, we consider that this domain is failed and count it as a domain failure. Since the cascading failure we consider is mainly caused by PM overload, we exclude the first domain failufre when counting the number of domain failures.

2. *SLO violation.* This metric aims to evaluate the performance of each method on satisfying SLOs. SLO violation is determined by the percentage of a PM's running time, during which the PM has a 100% CPU utilization (SLOVO) and the performance degradation due to VM migration (SLOVM) [49]. $SLOVO = \frac{1}{N}\sum_{i=1}^{N}\frac{T_{s_i}}{T_{a_i}}$, where N is the number of running PMs, T_{s_i} is the total running time of PM i with 100% CPU utilization, and T_{a_i} is the total time during PM i serving VMs. $SLOVM = \frac{1}{M}\sum_{j=1}^{M}\frac{C_{d_j}}{C_{r_j}}$ where M is the total number of VMs. C_{d_j} is the estimation of VM performance degradation caused by VM migration (we use 10% as in [49]). If the VMs needed to be reassigned to another PM via footprint transmission, the performance degradation is set to 0. C_{r_j} is the total CPU requested capacity by VM j. Finally, SLOV is the product of SLOVO and SLOVM.

3. *The number of failed PMs.* This metric aims to evaluate the failure severity of the datacenter. A smaller number of failed PMs means better performance on PM failure avoidance. All types of failures including the first failure caused by power outage and network failure are counted in this metric.

4. *Computing time.* This metric measures the system overhead and then represents the efficiency of the system in time scale. The computing time is the sum of all the running time for each method excluding the VM transmission time.

C. Simulation Results

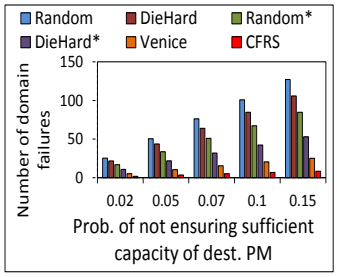

Fig. 4: Number of domain failures.

Figure 4 shows the number of domain failures versus the probability of not ensuring sufficient capacity of destination PM which is from 2% to 15%. The result follows *Random>DieHard>Random*> DieHard*>Venice>CFRS*. Since the domain failures are mainly caused by overloaded PMs, *Random* and *DieHard* without periodical load balancing always yield the largest number of domain failures. The *Random** randomly assigns all the VMs to PMs that have enough capacity for the VMs at the start of the simulation. During the simulation, when some PMs fail or become overloaded, the master machine may reallocate the VMs from these PMs without checking whether the available capacities of the destination PMs are enough to host the VMs. When the master machine checks the resource utilization of the destination

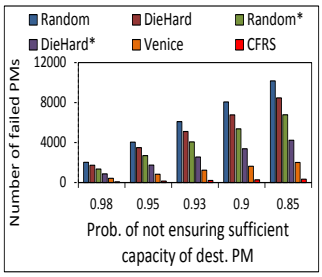

Fig. 5: The number of failed PMs.

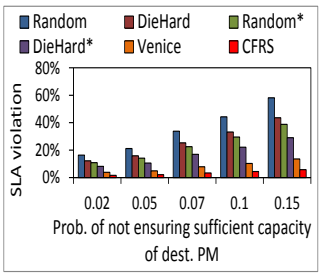

Fig. 6: SLO violation.

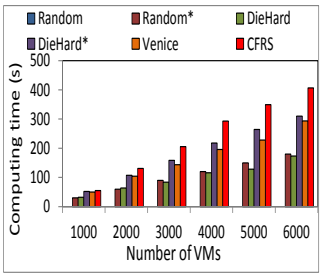

Fig. 7: Computing time.

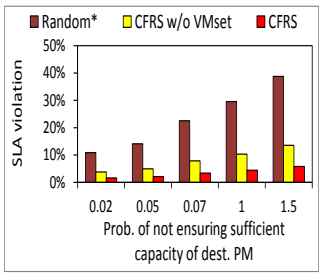

Fig. 8: SLO violation for different number of VM backups of a VM.

PMs and reallocate VMs to PMs that have enough available capacities, since it does not achieve long-term load balance, the destination PM has a high probability to be overloaded and may fail and generate a cascading failure. Also, to select VMs from an overloaded PM, *Random** randomly selects VMs from the PM, which also cannot achieve long-term load balance, and generates PM failure and a cascading failure. Note that *Random** randomly places VM backups without any VM backup placement optimization, so the backups of a VM may also fail in the cascading failures. When a VM's backups fail, the VM's host PM needs to migrate all the data of the VM to another PM, which costs much longer time in VM migration compared with only footprint transmission to the PM host of a backup. The longer time in VM migration increases the overload duration of the source PM, which makes the PM more likely to fail with its failure rate. As a result, more PMs are overloaded, which leads to cascading failures and then a larger number of domain failures compared with other methods.

Like *Random**, in VM reallocation to handle the PM overload or failure, *DieHard** cannot achieve long-term load balanced state for the source PMs and the destination PMs. Therefore, the PMs are likely to be overloaded and fail, leading to cascading failure. *DieHard** is more advantageous than *Random** in that it has the VM backup placement optimization by placing more VMs and their backups on the domains with lower failure rates, which helps reduce loss probability of VMs and their backups. Since more VM backups are available during PM failures compared to *Random**, in VM reallocation, the VM migration has a higher probability to only conduct footprint transmission rather than transmitting all data of VMs. Shorter time in VM migration means that source PMs are less likely to be overloaded in *DieHard** compared to *Random**. Thus, few PMs are overloaded, which leads to fewer domain failures in *DieHard** compared with *Random**.

In the initial VM allocation and VM reallocation during PM overload or failure, *Venice* allocates a VM to a PM based on the availability scores. *Venice* randomly allocates VM backups to PMs to increase VM reliability but does not have VM backup placement optimization compared with *DieHard**.

*DieHard** randomly selects PMs with enough capacity for migrated VMs without considering the reliability of the destination PMs. *DieHard**'s VM backup optimization can increase the probability that a VM backup is available in a domain failure, but cannot ensure VM back availability in cascading failures which may make all backups of a VM fail. Therefore, *Venice* can achieve better VM reliability after VM migration, which generates few VM reallocations and then a smaller number of PM failures and domain failures.

We see that *CFRS* produces the smallest number of domain failures. In a PM overload or failure, the master machine needs to conduct VM reallocation. When the master machine checks the available capacities of the destination PMs, *CFRS* estimates the workload on PMs and VMs in each epoch during a time period and balances the workload on both source PMs and destination PMs in long term by carefully selecting VMs to migrate out and the destination PMs for VMs that need to reallocate. The long-term load balance avoids PM overload in long term and hence avoids PM failures or cascading failures. Meanwhile, when the master machine does not check the destination PM, the PM has a high probability to be overloaded and failed. In this case, *CFRS* can help achieve long-term load balance in the next VM reallocation period, which help avoid PM failures and cascading failures. Like *DieHard** and *Venice*, *VMset* considers the failure rates of different PMs in VM backup placement to increase VM backup reliability. Further, in cascading failures and concurrent failures, *VMset* increases the reliability of VMs' backups, while *DieHard** and *Venice* tend to lose VM backups. Fewer VM backup losses lead to shorter time needed for VM reallocation (as transmitting footprint needs shorter time than transmitting all the data of a VM), which reduces the time duration that the source PM is overloaded and then the failure probability of the PM. Furthermore, *CFRS* dynamically adjusts the oversubscription ratio based on the real resource utilization of each PM. Once the resource utilization is higher than the pre-defined threshold range, the oversubscription ratio is reduced to limit the resource usage of the PM and vice versa. In summary, *CFRS* produces the lowest number of domain failure occurrences among all of the methods.

Figure 5 shows the number of failed PMs versus the probability of not ensuring sufficient capacity of destination PM which is from 2% to 15%. The result follows *Random>DieHard>Random*> DieHard*>Venice>CFRS* due to the same reasons as in Figure 4

In Figure 6, we measure the performance of SLO violation versus the probability of not ensuring sufficient capacity of destination PM which is from 2% to 15%. The result follows *Random>DieHard>Random*> DieHard*>Venice>CFRS*, which is the same as the result in

Figure 4. This is because more failed PMs mean that more VMs' requests cannot be satisfied, which results in higher SLO violation.

Figure 7 shows the computing time of each method versus the number of VMs. The result follows *Random*<DieHard<Venice<DieHard*<CFRS*. For *Random**, it does not have any optimization algorithm and its computing time is only for calculating the available capacities of PMs to find PMs with enough capacity for VMs in periodical load balancing. In *DieHard*, it needs to calculate the necessary number of VM backups and generate the VM placement to provide VM reliability guarantee. *DieHard** needs additional time for periodical load balancing. In *Venice*, it calculates the availability score of each PM according to the PM's resource utilization and failure rate, and the score of each VM according to the required resource of the VM and the reliability requirement set by users in initial VM allocation and periodical load balancing. *CFRS* has the highest computing time compared with other methods, since it needs to calculate the priority score for each overloaded PM and its VMs in periodical load balancing operation. It also needs to generate a reliable backup placement to achieve higher VM backup reliability. Meanwhile, in *DOA*, each PM needs to check if its resource utilization is in the threshold range and then adjust the oversubscription ratio if needed.

D. Effectiveness of Each Method

Now we evaluate the effectiveness of *VMset* and *DOA* in *CFRS*. We first evaluate the effectiveness of *VMset*. Figure 8 shows the SLO violation versus the probability of not ensuring sufficient capacity of destination PM which is from 2% to 15%. Here, *CFRS w/o VMset* means *CFRS* without *VMset*, which uses random VM backup placement. The result follows *Random*>CFRS w/o VMset>CFRS*. *CFRS w/o VMset* does not have a VM backup placement optimization method. *VMset* increases the reliability of VM backups in cascading failures and concurrent failures, and further considers the failure rates of different PMs in VM backup placement to increase VM backup reliability. It reduces the time duration that a source PM is overloaded and then the failure probability of the PM. In *CFRS w/o VMset*, the VM backup reliability is not as high as *CFRS* and hence yields a larger number of failed or overloaded PMs. Thus, *CFRS w/o VMset* generates a higher SLO violation value than *CFRS*.

Figure 9 shows the SLO violation versus the number of VM backups for each VM. R means the number of VM backups for each VM. The result shows that $R = 1 > R = 5 > R = 10 > R = 15$; that is, a higher R value leads to less SLO violation. For a smaller R (i.e., a limited number of backups for a VM), when some PMs fail, then some VMs may lose all if their backups that are stored on the failed PMs. As a result, the VMs on them must be reallocated to other PMs without the backups. More total VM migrations generate more performance degradation, which leads to larger SLO violation value. When R is a little larger, the VMs that are hosted in failed PMs have backups on available PMs with higher probability

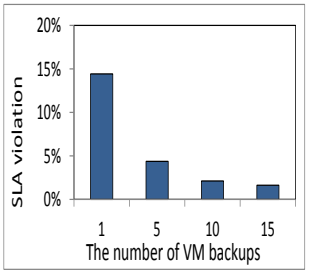

Fig. 9: SLO violation for different number of VM backups.

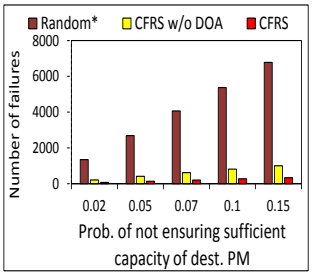

Fig. 10: The number of failed PMs.

and then these VMs can be reassigned to PMs with backups by footprint transmission with less VM migration performance degradation. For a larger R, while some PMs fail, more VM backups can survive during failures. Then, more VMs can be reassigned to other PMs with the backups without transferring the total VMs, generating a lower SLO violation value.

Figure 10 shows the number of failed PMs versus the probability of not ensuring sufficient capacity of destination PM. Here, *CFRS w/o DOA* means *CFRS* without *DOA*. The result follows *Random*>CFRS w/o DOA>CFRS*. *CFRS w/o DOA* does not adjust the oversubscription ratio based on the real resource utilization. In *DOA*, the adjusted oversubscription ratio can decrease the possibility that the total resource using by all the VMs on a PM exceeds its capacity. It can reduce the number of overloaded PMs and failed PMs. In *CFRS w/o DOA*, the PM is more likely to be overload compared with *CFRS*. Thus, *CFRS w/o DOA* can cause a PM to be overloaded with a higher probability than *CFRS*, which may lead to the failure of this PM.

V. Conclusion

After a failure (e.g., power outage, network failure) occurs, due to the load transfer from failed domains to other domains, the destination domains may become overloaded and fail, which cause a cascading failure. Previous methods to handle failures are not sufficiently resilient to such cascading failures. In this paper, we propose a Cascading Failure Resilience System (*CFRS*). First, *CFRS* chooses VMs in overloaded PMs to migrate out and selects destination PMs to host the VMs from overloaded PMs or failed PMs to achieve long-term load balance, which reduces the probability of failure occurrence. Second, *CFRS* places VM backups to PMs to increase the backup reliability in cascading failures. Third, *CFRS* dynamically adjusts oversubscription ratio of each resource in a PM according to the real workload of the PM to avoid PM overload and hence possible failure. Our trace-driven simulation shows the superior performance of *CFRS* in cascading failure avoidance compared with other methods. In the future work, we will study how to estimate the effect of failures and explore how to mitigate different types of failures.

Acknowledgment

This research was supported in part by U.S. NSF grants OAC-1724845, ACI-1719397 and CNS-1733596, and Microsoft Research Faculty Fellowship 8300751.

REFERENCES

[1] F. Tso, K. Oikonomou, E. Kavvadia, and D. Pezaros. Scalable traffic-aware virtual machine management for cloud data centers. In *Proc. of ICDCS*, 2014.

[2] H. Shen and Z. Li. New bandwidth sharing and pricing policies to achieve a win-win situation for cloud provider and tenants. *Trans. on TPDS*, 2016.

[3] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C. Chuah, and C. Diot. Characterization of failures in an ip backbone. In *Proc. of INFOCOM*, 2004.

[4] H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter. Traffic engineering with forward fault correction. In *Proc. of SIGCOMM*, 2015.

[5] S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred. Taking the blame game out of data centers operations with netpoirot. In *Proc. of SIGCOMM*, 2016.

[6] H. S. Gunawi, T. Do, J. M. Hellerstein, I. Stoica, D. Borthakur, and J. Robbins. Failure as a service (faas): A cloud service for large-scale, online failure drills. *University of California, Berkeley, Technical Report*, 2011.

[7] C. Asaf, R. Stephen, S. Ryan, K. Sachin, O. John, and R. Mendel. Copysets: Reducing the frequency of data loss in cloud storage. In *Proc. of ATC*, 2013.

[8] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proc. of MSST*, 2010.

[9] R. J. Chansler. Data availability and durability with the hadoop distributed file system. In *Proc. of USENIX*, 2012.

[10] D. Borthakur, J. Gray, J. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, and S. Rash. Apache hadoop goes realtime at facebook. In *Proc. of SIGMOD*, 2011.

[11] M. Sedaghat, E. Wadbro, J. Wilkes, S. De Luna, O. Seleznjev, and E. Elmroth. Diehard: reliable scheduling to survive correlated failures in cloud data centers. In *Proc. of CCGrid*, 2016.

[12] Z. Tan and S. Babu. Tempo: robust and self-tuning resource management in multi-tenant parallel databases. In *Proc. of VLDB*, 2016.

[13] Q. Zhang, M. Zhani, R. Boutaba, and J. Hellerstein. Dynamic heterogeneity-aware resource provisioning in the cloud. *Trans. on Cloud Computing*, 2014.

[14] H. Shen, A. Sarker, L. Yu, and F. Deng. Probabilistic network-aware task placement for mapreduce scheduling. In *Proc. of Cluster*, 2016.

[15] H. Wang and H. Shen. Proactive incast congestion control in a datacenter serving web applications. In *Proc. of INFOCOM*, 2018.

[16] H. Wang, H. Shen, and G. Liu. Swarm-based incast congestion control in datacenter serving web applications. In *Proc. of SPAA*, 2017.

[17] Z. Qi, Z. M. Faten, J. Maissa, and B. Raouf. Venice: Reliable virtual data center embedding in clouds. In *Proc. of INFOCOM*, 2014.

[18] B. Peter, M. Ishai, C. Mosharaf, M. Pradeepkumar, M. David A, and S. Ion. Surviving failures in bandwidth-constrained datacenters. In *Proc. of SIGCOMM*, 2012.

[19] N. Bila, E. de Lara, K. Joshi, H. Lagar-Cavilla, M. Hiltunen, and M. Satyanarayanan. Jettison: efficient idle desktop consolidation with partial vm migration. In *Proc. of EuroSys*, 2012.

[20] X. Zhang, Z. Shae, S. Zheng, and H. Jamjoom. Virtual machine migration in an over-committed cloud. In *Proc. of NOMS*, 2012.

[21] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proc. of NSDI*, 2007.

[22] J. Xu, J. Tang, K. Kwiat, W. Zhang, and G. Xue. Survivable virtual infrastructure mapping in virtualized data centers. In *Proc. of ClOUD*, 2012.

[23] W. Cirne and E. Frachtenberg. Web-scale job scheduling. In *Proc. of JSSPP*, 2012.

[24] Y. Wai-Leong, W. Cédric, and K. Ulas. Designing and embedding reliable virtual infrastructures. In *Proc. of SIGCOMM*, 2011.

[25] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, and P. Germano. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proc. of SIGCOMM*, 2015.

[26] M. Isard. Autopilot: automatic data center management. In *Proc. of SIGOPS*, 2007.

[27] W. Xin, T. Daniel, C. Chao-Chih, M. David A, Y. Xiaowei, Y. Lihua, and Z. Ming. Netpilot: automating datacenter network failure mitigation. In *Proc. of SIGCOMM*, 2012.

[28] Y. Wang, H. Wang, A. Mahimkar, R. Alimi, Y. Zhang, L. Qiu, and Y. R. Yang. R3: resilient routing reconfiguration. In *Proc. of SIGCOMM*, 2010.

[29] Summary of the amazon service disruption in the northern virginia region. http://aws.amazon.com/cn/message/41926/, [accessed in Sep. 2017].

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

[31] G. Phillipa, J. Navendu, and N. Nachiappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proc. of SIGCOMM*, 2011.

[32] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proc. of OSDI*, 2010.

[33] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: practical power-proportionality for data center storage. In *Proc. of Computer systems*, 2011.

[34] L. Yan, K. Chen, H. Shen, and G. Liu. Mobilecopy: Resisting correlated node failures to enhance data availability in DTNs. In *Proc. of SECON*, 2015.

[35] B. Salman A, W. Long, and T. Chunqiang. Towards an understanding of oversubscription in cloud. In *Proc. of*

USENIX Workshop, 2012.

[36] C. Tang. Fvd: A high-performance virtual machine image format for cloud. In *Proc. of USENIX ATC*, 2011.

[37] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proc. of EuroSys*, 2012.

[38] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proc. of SIGCOMM*, 2015.

[39] Ibm knowledge center backup vm guide. https://www.ibm.com/support/knowledgecenter/en/SSGSG7_7.1.6/client/r_cmd_bkupvm.html, [accessed in Sep. 2017].

[40] Vmware virtual machine backup guide. https://www.vmware.com/pdf/vsphere4/r40/vsp_vcb_15_u1_admin_guide.pdf, [accessed in Sep. 2017].

[41] S. Nath, H. Yu, P.B. Gibbons, and S. Seshan. Subtleties in tolerating correlated failures in wide-area storage systems. In *Proc. of NSDI*, 2006.

[42] M. Zhong, K. Shen, and J. Seiferas. Replication degree customization for high availability. In *Proc. of EuroSys*, 2008.

[43] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. of NSDI*, 2005.

[44] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proc. of SOSP*, 2011.

[45] C. Kozierok. *The TCP/IP guide: a comprehensive, illustrated Internet protocols reference*. Google books, 2005.

[46] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces: format + schema. Technical report, 2011. Posted at https://github.com/google/cluster-data [accessed in Sep. 2017].

[47] J. Wilkes. More Google cluster data. Google research blog, 2011. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html [accessed in Sep. 2017].

[48] Amazon ec2 instance types-amazon web services(aws). https://aws.amazon.com/cn/ec2/instance-types/, [accessed in Sep. 2017].

[49] A. Beloglazov and R. Buyya. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. In *Proc. of CCPE*, 2012.