Vrije Universiteit Amsterdam

Research Project in Computer Security, Report

Design, Implementation and Evaluation of Container Migrations in Datacenter Scheduling

Author: Aratz Manterola Lasa (2690722)

1st supervisor: Prof. dr. Alexandru Iosup
daily supervisor, if different: Sacheendra Talluri

A report submitted in fulfillment of the requirements for the Research Project in Computer Security course

June 19, 2023

1 Introduction

Society's increasing dependence on digital technologies and infrastructure has led to the widespread use of datacenters for deploying digital services [4, 6]. Schedulers play a vital role in orchestrating datacenter resources to meet the demands of these services [5, 15].

However, despite the availability of abundant resources, users in datacenters often underutilize them [20], prompting providers to oversubscribe resources in order to achieve higher utilization [7]. However, this oversubscription can lead to interference among users [11, 9], even in cases where resources are not fully utilized. To mitigate interference, datacenter providers have the option to migrate virtual machines (VMs) [14, 18, 16] or allow the interference to persist. Another alternative is to inform tenants about the oversubscription and interferences so that they can take appropriate action to reduce the workload running on affected VMs. For instance, if users are running a Kubernetes (K8s) cluster on their provisioned VMs, they can migrate pods from oversubscribed VMs [8, 21], leveraging the smaller size of pods for more efficient migrations and improved resource packing. Additionally, users have the necessary business logic context to make informed decisions on workload reduction, such as determining which pods should be migrated or stopped.

We conduct an analysis of five mainstream industrial schedulers to evaluate whether they provide container migrations and, if not, how they could potentially support this functionality. The selected five industrial schedulers are: Kubernetes [2], SLURM [10], Spark [22], Condor [19], and Airflow [1]. Unfortunately, mainstream schedulers do not provide users with information about oversubscription and interferences. The information they offer primarily pertains to the resource consumption of VMs, without insight into the underlying physical resources. As a result, users are unaware of whether their VMs are oversubscribed or performing poorly. To address this limitation, we implement a scheduling extension where users are notified when their VMs are oversubscribed, enabling them to perform container migrations. This approach demonstrates that in certain scenarios, better scheduling performance can be achieved by offering users the programmability to receive callbacks and implement container migrations.

The extension centers around the evaluation of a use case where the scheduler targets the reduction of oversubscription and interferences among tenants through container migrations. By showcasing the advantages that stem from providing users with the required programmability, we shed light on the potential benefits of this approach. To conduct the evaluation, we employ trace-driven simulation, allowing us to examine the impact of container migrations on total execution time per task. The results demonstrate a remarkable 81% improvement in total execution time, emphasizing the performance gains that schedulers forego by not implementing the essential programming abstractions. All artifacts, including the traces used in the experiment, are available in `https://github.com/aratz-lasa/opendc`.

It is important to note that this work is part of a broader research endeavor focused on studying programming abstractions of datacenter scheduling. Within this context, the specific focus of this study was to assess the capabilities of existing industrial schedulers in implementing advanced programming abstractions, particularly in relation to container migrations. By examining the existing schedulers in detail and conducting extensive experiments, we provide valuable insights into the limitations of current programming abstractions and shed light on the potential performance improvements that can be attained

through the adoption of a dedicated migration API. Our research endeavors to contribute to the advancement of datacenter scheduling by identifying areas where enhancements and improvements can be made to maximize efficiency and optimize resource allocation.

2 Requirements

Before designing the experiment and the API extension, we identified specific functional and non-functional requirements for the experiment, and we present them below:

The functional requirements are the following:

FR Enable expression of oversubscription on datacenter scheduler APIs

The main requirement of this experiment is to have datacenter schedulers express oversubscription to users. Our hypothesis is based on the fact that current datacenters abstract users from oversubscriptions. This offers greater simplicity but sacrifices performance improvements that can only be obtained with the user's collaboration since users have the necessary knowledge about workload and business, which allows for taking optimal actions.

FR Enable container migrations.

Specifically, with this experiment, we enable container migrations. Container migrations are performed for tasks running inside provisioned virtual machines rather than the virtual machines themselves. In this way, we hope that the migration capacity and the packing of tasks will be greater when there are oversubscriptions in the datacenter.

FR Provide security and privacy.

By enabling oversubscription expression in the datacenter, the provider offers information about the underlying resources and possible aggregated data about other tenants. This has the potential to generate security breaches over. Therefore, the API must be able to enable the oversubscription expression while not offering compromised information about the underlying resources.

The non-functional requirements are the following:

NFR Make reproducible experiments

For an experiment to be reliable, it must be reproducible since today, there are notorious problems with the inability to reproduce scientific experiments. For this reason, it is necessary to publicly offer both the raw results of the experiments and the software artifacts used to carry them out.

NFR Make the experiment software artifacts reusable

Considering current programming standards, it is important to implement extensible and modular software artifacts so that it is easy to reuse or extend them for other projects.

Workload	VMs	Duration [days]	VM duration [days]		CPU cores		CPU capacity [GHz]		Memory [GBs]	
			Mean	σ	Mean	σ	Mean	σ	Mean	σ
Bitbrains	1250	30	28	5	3.27	4.04	2.7	0.16	11.75	32.6
Azure	1829	30	2	6	2.48	2.28	2.5	0.0	5.8	10.16
Google	1000000	2.5	0.0375	0.083	1.0	0.0	1.68	2.08	0.17	0.2

Table 1: Characteristics of the traces of the experiments

NFR Provide experiments with different workloads.

To offer greater insight and reliability of the experiment, it is necessary to use different workloads. They must be different in terms of characteristics, such as the duration of each task, the resource requirements, the nature of the workload, etc.

3 Traces

For the experiments, we use real-world trace workloads. We chose traces from private and public cloud environments, which offer anonymized requests from VMs, plus aggregated metrics on resource utilization every 5 minutes. The chosen traces are Bitbrains [17] Azure [3] and Google [13]. While Bitbrains and Azure traces are VM requests, Google traces are task requests. Generally, task requests are shorter in duration and consume fewer resources than VM requests. We include task requests to enrich our experiment and see how the results of experiments change with short-lived and small consumption requests. Even though our experiments are designed for VMs, it's straightforward to convert them for task requests and CPU core reservations instead of VM reservations. Since the logic of experiments is the same for VMs as it is for tasks. Moreover, in the simulation of the experiment, we do not have to make any major changes.

Bitbrains is a private cloud provider operating mainly in the Dutch ICT market; the aggregate duration is one month with 1250 VMs. Finally, we also chose the Azure and Google traces, which are traces from Microsoft's and Google's public cloud providers, respectively. On the one hand, the Azure trace is very recent, from 2020. The original trace contains 2 million VMs, and the aggregate duration is approximately two and a half months. However, it is a very large trace compared to the other traces, requiring much time and execution power. Therefore, 1829 VMs from the original trace are sampled using the OpenDC sampling tool [12]. On the other hand, the Google trace is from 2014, its original trace contains 17.8 million tasks, and the duration is approximately one month. This trace is very large, so we sample 1 million requests from the original trace in 2.5 days. We sample more requests than in Azure since the Google runtime is much smaller. This is because Google traces are not VM requests but task requests.

Table 1 summarizes the characterizations of these workloads.

4 Execution

The reproducibility of the experiments is crucial for their validity and for other researchers to extend them. An experiment is reproducible if the methods are sufficiently well described and the artifacts available so that others running the same experiment will get identical results. We run the experiments on a personal laptop with an Apple M1 Max

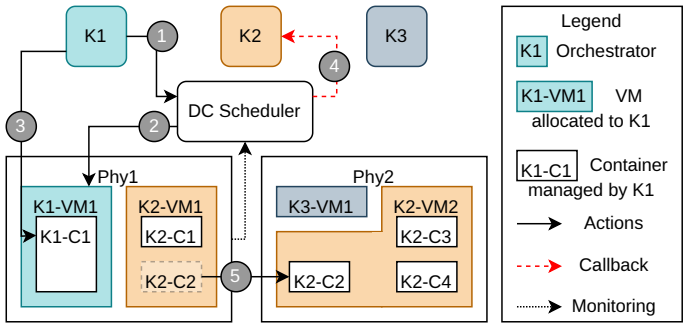


Figure 1: Container migrations experiment system model.

chip, 1TB SSD storage, and 32 GB memory. We run the experiments using OpenDC, an open-source datacenter discrete event simulator developed by AtLarge, with multiple years of development and operation. Due to its discrete event model, OpenDC generates the same results regardless of the hardware used. The execution of each configuration lasts about 1-3 minutes, and the execution of each configuration has been repeated 20 times. All artifacts, including the traces used in the experiment, are available in `https://github.com/aratz-lasa/opendc`.

5 System model

To conduct this experiment, we will simulate the oversubscription scenario within a datacenter environment. In the datacenter, multiple tenants engage in leasing and releasing virtual machines, as illustrated by ❶ and ❷ in Figure 1. Each tenant establishes a Kubernetes cluster (K1) on the leased virtual machines, deploying multiple batch tasks within the Kubernetes environment. The Kubernetes cluster consists of several nodes (K1-VM1) representing virtual machines, where application containers are launched (❸). The number of nodes varies across clusters, reflecting the unique workload requirements of each tenant. This Kubernetes layer operates as a secondary scheduling layer atop the datacenter scheduling, with tenants provisioning virtual machines and deploying Kubernetes nodes to establish their clusters. Once the task load subsides, the virtual machines are released, and the Kubernetes cluster is dismantled.

However, the utilization of resources within Kubernetes clusters can sometimes be suboptimal, leading to underutilization. To address this, the datacenter provider resorts to resource oversubscription in order to maximize performance. Consequently, the aggregate virtual resource allocation on a physical machine may exceed the underlying physical resources. When a Kubernetes cluster experiences high load under such conditions, interference and reduced performance may occur among tenants sharing the same physical machine. In such cases, the datacenter scheduler may attempt to migrate virtual machines to alternative physical machines, aiming to mitigate interference and improve overall performance.

The datacenter provides users with a simplified API, consisting of the following operations:

- `lease(requirements)`: `vm`: Users specify their resource requirements, and the

provider provisions a virtual machine that meets those specifications. The API call returns the allocated virtual machine to the user. Resource requirements typically include CPU cores, CPU capacity, and memory.

- `release(vm)`: Users release a previously leased virtual machine by passing it as a parameter to the API call, and the provider shuts down the specified machine.

6 Model extension

In this experiment, we expand on the model described in the previous section by introducing the capability for the datacenter scheduler ❹ to initiate callbacks to the user when the underlying resources are oversubscribed. The user includes a callback function named `requestUserMigration`, which the scheduler invokes. The callback function receives a target virtual machine and the amount of CPU capacity that is oversubscribed as arguments. In response, the orchestrator migrates ❺ selected containers (specifically pods in this case) and provides the amount of CPU capacity to be reduced through container migrations. This approach aims to improve task performance by reducing the migration size, as pods are smaller than nodes. Consequently, virtual machines achieve better resource packing, leading to reduced interference and improved performance.

The extended programming model offered by the scheduler includes the following functions:

- `communicate(callback, event)`: This function allows the user to specify a `callback` and an associated `event` that triggers the callback, which in this case is a migration event. The scheduler receives and stores the callback and associated event, ready to be executed when oversubscription occurs and user migrations need to be performed.

- `requestUserMigration(vm, cpuCapacity) migratedCpuCapacity`: In the user-submitted callback function, the datacenter scheduler provides the oversubscribed virtual machine (`vm`) and the amount of CPU capacity (`cpuCapacity`) that needs to be migrated. The user performs necessary calculations and returns the amount of CPU capacity from the requested migration that will be migrated.

7 Alternatives

Prior to finalizing the implementation of the migrations API, we explored various alternatives to fulfill the system extension requirements. Next, we will provide a brief overview of the alternative approaches considered and present the rationale behind our chosen design.

Transparent utilization. The simplest alternative is for the datacenter provider to offer live metrics of the underlying physical machines. Users can view the utilization of the physical machine when provisioning a virtual machine, allowing them to determine if the machine is oversubscribed and if there are interferences with other tenants. This approach enables tenants to perform container migrations when they detect oversubscription or implement optimizations as resource usage grows. However, this model raises privacy and security concerns since users gain direct access to information about the underlying

resources and can take actions that may conflict with the interests of the datacenter provider.

Oversubscription notification. Another alternative to prevent users from accessing live metrics of the physical machine directly is to replace them with notifications. The datacenter provider sends notifications to users when the underlying physical machine is oversubscribed or experiencing interference. Users receive information only when there is an oversubscription, prompting them to take necessary actions. However, this alternative can make coordination between tenants challenging, similar to the transparent utilization approach. Each tenant independently decides whether to perform container migrations, which may lead to scenarios where all tenants migrate simultaneously or none of them migrate while expecting others to migrate.

Oversubscription callback. The third alternative offers a balance between security and coordination among users. Instead of receiving notifications, users provide a callback function to the provider. This callback function receives the amount of CPU capacity that the provider requests the user to migrate, and in response, the user specifies the amount of CPU capacity that will be migrated. This bidirectional communication between the user and the datacenter allows the provider to act as a coordinator among the users. Implementing this alternative requires more complexity from the user's perspective, but we believe that the potential performance gains outweigh the added complexity. Therefore, we have chosen to implement this third alternative as the model extension for the experiment.

8 Industrial schedulers

In this section, we analyze the ability of real industrial schedulers to implement the migrations API discussed earlier, and if they are unable to do so, we explore the necessary extensions required to support this functionality. The schedulers under examination include Kubernetes (v1.27), SLURM (v23.02), Spark (v3.4.0), Condor (v10.4.3), and Airflow (v3.3.0). By examining these popular industrial schedulers, we aim to determine their capabilities and limitations regarding container migrations, providing insights into the potential enhancements needed to enable efficient migration strategies within their existing APIs. This is crucial to bridge the gap between theoretical research and practical implementation.

Kubernetes users submit callbacks for each of the container lifecycle events they are interested in, allowing the execution of a callback when the pod changes its state. Next, we show how the user would specify the migration callback of her application.

```
apiVersion: v1
kind: Pod
metadata:
  name: lifecycle-demo
spec:
  containers:
```

```
7   - name: example
8     image: example:1.28
9     lifecycle:
10      migration:
11        exec:
12          command: ["/tmp/migration.prog"]
```

In this example, a callback is assigned via the *exec.command* parameter. In this specific case, it is specified that the callback is executed when the scheduler requires a migration. After the container has started, use *postStart*. Although Kubernetes provides an abstraction for adding callbacks, there is currently no built-in lifecycle state specifically for migrations. Therefore, Kubernetes would need to extend its API to include a migration state and support container migrations effectively.

SLURM offers the ability to add callbacks to specific job events using the `strigger` command. Below we present how the user would specify the migration callback of her application.

```
strigger --set --jobid=1234
    --migration --program=/tmp/migration.prog
```

In this example, the program */tmp/migration.prog* is executed when the scheduler requires migrations by specifying the *–migration* flag. So, SLURM already provides an abstraction for adding callbacks but does not include a migration state to activate the callbacks. Therefore, SLURM would require an API extension to support container migrations.

Spark also provides the ability to add callbacks, which can be defined in code and set through a CLI flag. Here is a specific example:

```
./bin/spark-submit --class org.apache.spark.examples.SparkPi
    --master spark://207.184.161.138:7077
    --conf spark.extraListeners=listener.MigrationListener
    /path/to/examples.jar
```

In this example, a configuration flag *spark.extraListeners* is set to the value *listener.MigrationListener* specifies that a custom-made callback is passed to receive up-calls when migrations are required from the scheduler. The callback is defined in a code file named *listener*, and inside it, the listener is implemented as an object named *MigrationListener*. In Spark, there is already the abstraction to add callbacks as listeners. However, the current Spark API lacks a method specifically for receiving migration requests. To enable container migrations, Spark would need to extend its API by adding an interface method for receiving migration requests and implementing the necessary migration logic.

Condor is the only industrial scheduler among the ones analyzed that does not support callbacks. Consequently, it cannot provide container migrations based on the proposed alternative extension. Next, we show how the scheduler could implement callbacks in its API.

```
executable          = example
arguments           = SomeArgument
callback.migration = /tmp/callback.prog

queue
```

In this example, the callback is implemented in Condor by adding a new parameter to the submission file *callback.migration*, which is set to */tmp/callback.prog* that specifies where the program that performs the container migrations is located. The implementation of callbacks in Condor can take different forms, including using a binary or defining the callback code directly in the submission file.

Airflow allows users to assign callbacks to tasks, triggering their execution when specific events occur, such as failures. Next, we present how Spark could use callbacks to implement container migrations.

```
from datetime import datetime
from airflow import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.python_operator import PythonOperator

def task_migration(context):
    vm = context['vm']
    cpu_capacity = context['cpu_capacity']def example():
    ret 'Hello world from Airflow DAG!'

dag = DAG('hello_world', description='Hello World DAG',
    on_migration_callback=task_migration)

example_operator = PythonOperator(task_id='example_task',
    python_callable=example, dag=dag)

example_operator
```

In this example, a function named *task_migration* is executed whenever the scheduler requires container migrations. A python function is defined and passed to the dag object through the *on_migration_callback* argument for setting the callback. While Airflow already offers an abstraction for adding callbacks, it currently lacks a way to specify callbacks for migrations. To enable container migrations, Airflow would need to extend its API by introducing a new parameter, such as `on_migration_callback`, which would receive migration requests and facilitate the implementation of container migrations.

9 Configuration and design of the experiment

Our objective is to perform a series of experiments that highlight the limitations of a scheduler lacking container migrations programming abstractions. The configurations for this extension are summarized in Table 2. These experiment configurations encompass three key dimensions: trace, container migrations, and oversubscription ratio. In the following sections, we will delve into each dimension and discuss the various choices available, along with the metrics collected during the experiments.

Traces	**container migrations**	**Oversubscription ratios**
Bitbrains	true false	3.0 4.0 5.0
Azure	true false	3.0 4.0 5.0
Google	true false	3.0 4.0 5.0

Table 2: An overview of all the configurations of the migrations extension experiment.

In addition to the traces, container migrations, and oversubscription ratios, there is another dimension to consider: the utilization level and the number of Kubernetes clusters. For our experiments, we set these two dimensions to a fixed value of 85% utilization and 5 Kubernetes clusters. This means that we calculate the underlying physical topology of the physical machines based on the average CPU and memory usage observed in each trace. We then set the CPU capacity and CPU core amounts to 85% of their maximum values, reflecting the desired utilization level. This fixed configuration allows us to maintain consistency in the experiments and evaluate the impact of other dimensions, such as traces, container migrations, and oversubscription ratios, on the performance metrics.

Container Migrations and Utilization Ratio in addition to the trace analysis, we evaluate various combinations of container migrations and oversubscription ratios to gain a comprehensive understanding of their impact on system performance.

To ensure the experiment's completeness and validity, we explore different oversubscription ratios. The oversubscription ratio represents the ratio between the number of virtual CPUs provisioned and the available physical CPUs. For instance, if the oversubscription ratio is set to 4.0 and there are 2 physical CPUs, the corresponding Kubernetes clusters will have 2 x 4.0 = 8 CPUs. We conduct experiments using three oversubscription ratios: 3.0, 4.0, and 5.0.

Furthermore, we examine the effects of container and datacenter migrations to simulate beneficiary and neutral scenarios. When container migrations are enabled, pods are migrated upon detecting oversubscription, while nodes/VMs are migrated when disabled. If interference persists despite pod migrations, the datacenter initiates subsequent node migrations to mitigate performance degradation. By exploring different migration strategies, we gain insights into their respective impacts on system performance.

Metrics In this experiment, our objective is to enhance performance through migrations, as we anticipate that migrations will improve resource packing. To evaluate the effectiveness of migrations, we need to gather three types of metrics: time improvement metrics, resource packing metrics, and migration-related metrics.

The time improvement metrics enable us to analyze the performance enhancement in terms of execution time and waiting time. By examining these metrics, we can assess the impact of migrations on reducing task completion time and overall job waiting time.

The resource packing metrics allow us to understand whether the performance improvements are a result of better resource utilization and allocation. These metrics provide insights into how effectively resources are utilized and how well the system is able to pack tasks onto available resources.

Additionally, the migration-related metrics enable us to investigate the effectiveness of migrations in achieving better resource packing. These metrics help us analyze the number and frequency of migrations performed, as well as the impact of migrations on resource allocation and interference reduction.

While OpenDC provides a wide range of available metrics for analyzing simulation results, we have selected a subset of metrics specifically tailored to our experiment. The chosen metrics, presented in Table 3, provide us with the necessary data to conduct a comprehensive analysis of the experiment and evaluate the effectiveness of container migrations in improving performance.

Name	Unit	Description
vm.id	-	Unique identifier of the VM
vm.provision time	Epoch (ms)	The instant at which the server was enqueued for the scheduler
vm.boot time	Epoch (ms)	The instant at which the server booted
vm.stop time	Epoch (ms)	The instant at which the server stopped
vm.timestamp	Epoch (ms)	The timestamp of the current VM metric entry
machine.id	-	Unique identifier of the physical machine of the datacenter
machine.cpu utilization	-	The CPU utilization of the machine
machine.cpu count	-	The number of logical processor cores available for this machine
machine.timestamp	Epoch (ms)	The timestamp of the current physical machine metric entry
migrations.success	-	The number of migrations that successfully reduced oversubscription
migrations.failure	-	The number of migrations that are not able to reduce oversubscription
migrations.improvement	-	The amount of CPU capacity that is causing interference is migrated
migrations.penalty	Time duration (ms)	The amount of penalty in task durations migrations cause

Table 3: The metrics that are recorded for the experiment results

10 Implementation of a Software Prototype

In this section, we provide an overview of the software prototype we developed for conducting the experiment. The prototype is built upon OpenDC, an open-source datacenter discrete event simulator created by AtLarge with several years of development and operation. To incorporate the necessary functionalities for our experiment, we extended OpenDC to support a second layer of scheduling, perform VM migrations, and execute container migrations within the second layer. The original code of the extension can be found at `https://github.com/aratz-lasa/opendc/tree/master/opendc-experiments/studying-apis/migrations`.

Second Layer of Scheduling To simulate a second layer of scheduling, similar to Kubernetes running on top of a datacenter, we made modifications and additions to various components of OpenDC.

Firstly, we expanded the internal representation of virtual machines (VMs) to include metadata about the Kubernetes pods running on them. This enhancement enables us to model second-layer tasks and perform scheduling across VMs. Subsequently, we extended the `ComputeService` component to manage this metadata effectively, allowing for operations such as adding, deleting, and moving second-layer pods. Furthermore, we implemented new schedulers that ensure pods are exclusively executed on nodes in their corresponding Kubernetes cluster. To achieve this, we extended the metadata and schedulers to consider resource consumption per Kubernetes node rather than per VM. Additionally, we developed a modified version of the `ComputeServiceHelper` component, which determines when tasks are submitted to the datacenter and when they are removed, thereby establishing the underlying physical topology.

Oversubscription Detection and Migrations The most complex part of our software development efforts was implementing the mechanisms for detecting oversubscription and performing migrations at both the VM and pod levels.

To detect oversubscription, we extended the metadata to track the tasks running on each machine and their respective CPU capacity consumption. We integrated this detection mechanism into the scheduling process, so that whenever a new task is scheduled, oversubscription is analyzed, and migrations are initiated if necessary.

For simulating migrations, we implemented the logic to suspend task execution on one VM and resume it on another. This involved stopping the work of a task and launching it on a new VM. We also developed a customized version of the `ComputeService` component, which incorporated all the necessary logic for executing both container and VM migrations. The migration procedures for containers and VMs are similar.

To provide a clear understanding of the implementation details, Algorithm 1 presents the complete pseudocode for resolving any implementation-related doubts.

Secondary Optimizations and Bug Fixes In addition to implementing migrations, we introduced several optimizations and bug fixes to enhance the functionality and accuracy of OpenDC:

- We addressed a bug related to simulating interferences between virtual machines. We discovered that the rate was not being updated correctly, causing tasks to continue running at their initially requested rate. To rectify this, we made the necessary adjustments.

- We improved the implementation of the `ComputeServiceHelper` component. The default implementation in OpenDC launches tasks as VMs and waits for completion. However, instead of waiting for VM completion, we modified the component to calculate the task's ideal runtime and stop the machine after that specific duration. As a result, we developed a custom `ComputeServiceHelper` that sets a listener on the machine and waits until it has finished executing the workload.

- We added new metrics to ensure accurate calculations of time and migration-related statistics. Although OpenDC already provides metrics for boot time and provisioning time, it

Algorithm 1: Migrations simulation algorithm

```
1  Function Migrate(host, oversubscription):
2      if isPodMigrationsActivated then
3          migrated = migratePods(host, oversubscription)
4          oversubscription -= migrated
5      migrated = migrateNodes(host, oversubscription)
6      return oversubscription − migrated
7
8  Function migratePods(host, oversubscription):
9      migrated = 0
10     nodes = nodesSortedByCpuCount(host)
11     for node in nodes do
12         migrated += requestUserMigration(node, overubscription-migrated)
13         if migrated ≥ oversubscription then
14             return migrated
15     return migrated
16
17 Function requestUserMigration(node, oversubscription):
18     migrated = 0
19     pods = podsSortedByCpuCount(node)
20     for pod in pods do
21         nodes = getClusterNodes()
22         for node in sortByLessRemainingCpus(nodes) do
23             machine = nodeToMachine[node]
24             if !isOversubscribed(machine) then
25                 migrated += migratePod(pod, node)
26                 if migrated ≥ oversubscription then
27                     return migrated
28     return migrated
29
30 Function migrateNodes(host, oversubscription):
31     migrated = 0
32     nodes = nodesSortedByCpuCount(host)
33     for node in nodes do
34         machines = getCandidateMachines(node)
35         for machine in sortByLessRemainingCpus(machines) do
36             if !isOversubscribed(machine) then
37                 migrated += migrateNode(node, machine)
38                 if migrated ≥ oversubscription then
39                     return migrated
40     return migrated
41
```

lacks a metric for recording stop time. Therefore, we introduced this metric. Additionally, for our experiment, we included additional metrics to measure the number of migrations performed and evaluate the resulting improvements or penalties.

11 Assumptions

When conducting experiments of this nature, it is essential to make certain assumptions to manage the experiment's complexity effectively. Below, we outline the key assumptions made in this experiment:

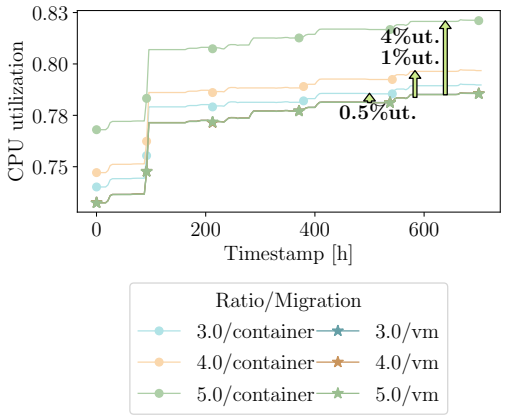
- Each task is assigned a specific CPU count, capacity, and utilization percentage for the requested resources. However, it is assumed that the workload for each task remains constant throughout the entire execution.
- Migrations are associated with penalties to simulate the time required to migrate a VM from one machine to another. The penalty duration is determined based on the amount of requested memory, with an extension of 1 minute for every 4 GB of memory requested.
- The results of the experiments are limited to the steady state of the traces. This is necessary because the traces have a predefined time limit. If the graphs were plotted from the beginning to the end of the execution, there would be a tail at the end where tasks continue to be executed, but no new tasks are submitted. This tail does not accurately represent a real-world scenario, as tasks continuously arrive in an actual system. To address this, we focus on the steady state of the obtained results. The steady state is defined as the time range between the submission of the first task and the submission of the last task, plus a delta. In this experiment, we set the delta as 5% of the average task duration.

12 Results

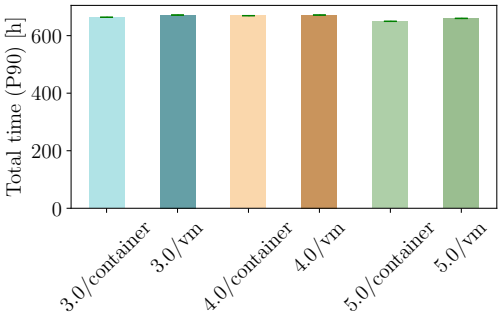
In Figures 2, 4, and 6, we present the results of the Bitbrains, Azure, and Google traces, respectively, for each combination of the oversubscription ratio and the activation (or deactivation) of the container migrations API. The purpose is to highlight the differences in scheduling performance between configurations that utilize the API and those that do not.

On the left side of the figures, we display the utilization of the physical machines within the datacenter. This metric represents the achieved resource packing efficiency for each experiment configuration. Higher utilization indicates better resource utilization and packing. On the right side, we present the empirical cumulative distribution function (ECDF) of the total time for each configuration. The total time encompasses both the waiting time and the execution time, providing an overall measure of task completion time.

Additionally, in Figures 3, 5, and 7, we provide data related to the migrations, aiming to understand their impact on the ECDF results. On the left side, we depict the cumulative number of successful migrations. This represents how many times, when facing



(a) Packing efficiency



(b) Tasks Total Times P90

Figure 2: Packing efficiency (left) and tasks Total Times P90 (right) of Bitbrains trace. Each line and bar represents a different `<Oversubscription ratio>/<Migrations API>` configuration.

oversubscription, the system was able to perform enough migrations to completely alleviate oversubscription. On the right side, we illustrate the cumulative penalty generated by the migrations, which reflects the additional time required for the execution of migrated tasks.

Bitbrains. Figure 2 illustrates the results for the Bitbrains trace. The packing graph demonstrates that configurations utilizing the API achieve better resource utilization, with up to a 4% higher utilization compared to configurations without the API. However, as time progresses, all configurations converge to similar utilization levels, indicating that the initial packing advantage diminishes over time.

The migrations graph in Figure 2 reveals that configurations utilizing the API experience a higher number of successful migrations, approximately 170,000 (93%) more migrations in total. However, it is important to note that these migrations come at a cost. The cumulative penalty associated with the migrations is also higher with the API, resulting in a total penalty difference of 41.6 hours (88%). The impact of these penalties may overshadow the benefits of the 3% higher resource packing achieved through container migrations, ultimately leading to limited improvements in task execution times.

Azure. Figure 4 presents the results for the Azure trace. The packing graph demonstrates that configurations utilizing the API achieve better resource packing. The difference in packing efficiency becomes more pronounced with higher oversubscription ratios. Specifically, at a ratio of 5.0, the API yields a 15% higher utilization, while at a ratio of 4.0, the difference is around 8%. However, at a ratio of 3.0, using the API results in worse packing compared to not using the API, with a difference of approximately 6%.

Certainly! Here's the modified code to align the subfigures in the second figure at the top:

Regarding task execution times, the API consistently yields shorter times, indicating

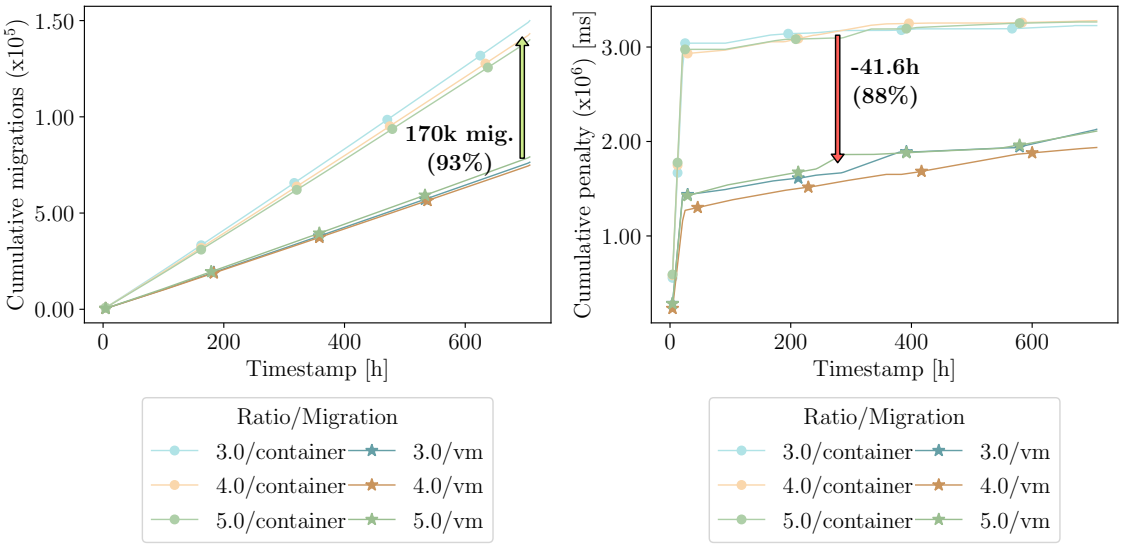


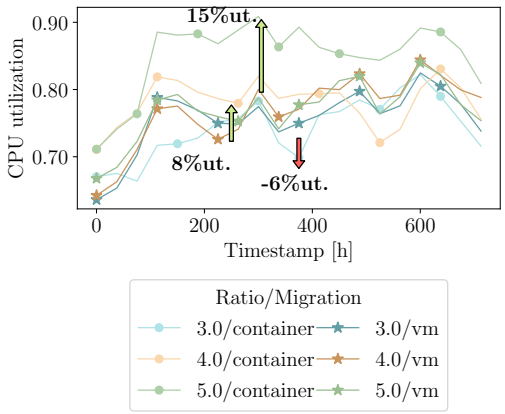
Figure 3: Migrations cumulative amount (left) and migrations cumulative penalty (right) of Bitbrains trace. Each line, bar, or boxplot represents a different `<Oversubscription ratio>/<Migrations API>` configuration.

higher performance. The greatest differences in performance are observed at the oversubscription ratios of 5.0 and 4.0. Using the API with a ratio of 5.0, the 90th percentile experiences a time reduction of 260 hours (81%), while at a ratio of 4.0, the reduction is 180 hours (13%). For a ratio of 3.0, the API results in a time reduction of around 45 hours (13%).

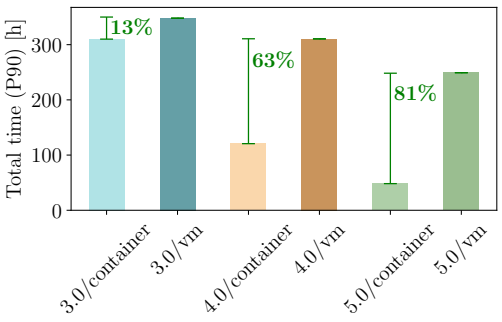
In Figure 5, we observe that using the API for migrations leads to a higher number of successful migrations. Specifically, for oversubscription ratios of 3, 4, and 5, the number of successful migrations increases by 400 (25%), 1300 (61%), and 1000 (23%) respectively. Additionally, there is a penalty reduction of 1.6 (25%) and 2.5 (25%) minutes for oversubscription ratios of 3 and 4, respectively, while an increase of 8.8 minutes is observed for an oversubscription ratio of 5.

Google. The packing graph in Figure 6 for the Google trace shows minimal differences among the different configurations, regardless of the oversubscription ratio and API usage. However, when utilizing the API, shorter task execution times and higher performance are observed. This can be attributed to the unique nature of the Google trace, where tasks are extremely small and utilize a single CPU core. Although the packing differences may not be noticeable due to the short task durations, significant improvements are seen in the 90th percentile total times. Notably, using the API with a ratio of 5.0 results in a 66% reduction (4 hours) in the 90th percentile times, while ratios of 3.0 and 4.0 achieve 21% (1 hour) and 3% (8 minutes) lower times, respectively.

In Figure 7, the use of the API leads to a significant increase in successful migrations, with approximately 35,000 more migrations compared to when the API is not used. Additionally, in this trace, no penalties are observed as the requested memory of the tasks



(a) Packing efficiency



(b) Tasks Total Times P90

Figure 4: Packing efficiency (left) and tasks Total Times P90 (right) of Azure trace. Each line and bar represents a different `<Oversubscription ratio>/<Migrations API>` configuration.

are small enough to avoid any penalties.

13 Discussion

Our main findings from this experiment are:

MF1 In all traces except for Bitbrains, the performance is improved by using the extended API for container migrations.

MF2 The highest oversubscription ratio of 5.0 obtains the highest performance improvement using container migrations.

MF3 The main benefit of migrations is greater packing, i.e., greater utilization of resources. However, when the tasks are very small, the benefits of migrations cannot be appreciated through packing.

MF4 It is complex to explain performance improvements through migration metrics, and it is necessary to generate more metrics and deeper analysis to have a complete picture.

This experiment highlights the trade-off between simplicity and performance in schedulers that do not offer callbacks to users. The results demonstrate that providing user-level migrations as a programming abstraction significantly improves performance in terms of total times, except for the Bitbrains trace. The highest oversubscription ratio of 5.0 achieves the greatest performance improvement, up to 260 hours lower total times for the 90th percentile. However, there are cases where using the lowest oversubscription ratio of 3.0 with user-level migrations results in worse performance, indicating the need for further research to understand the underlying causes.

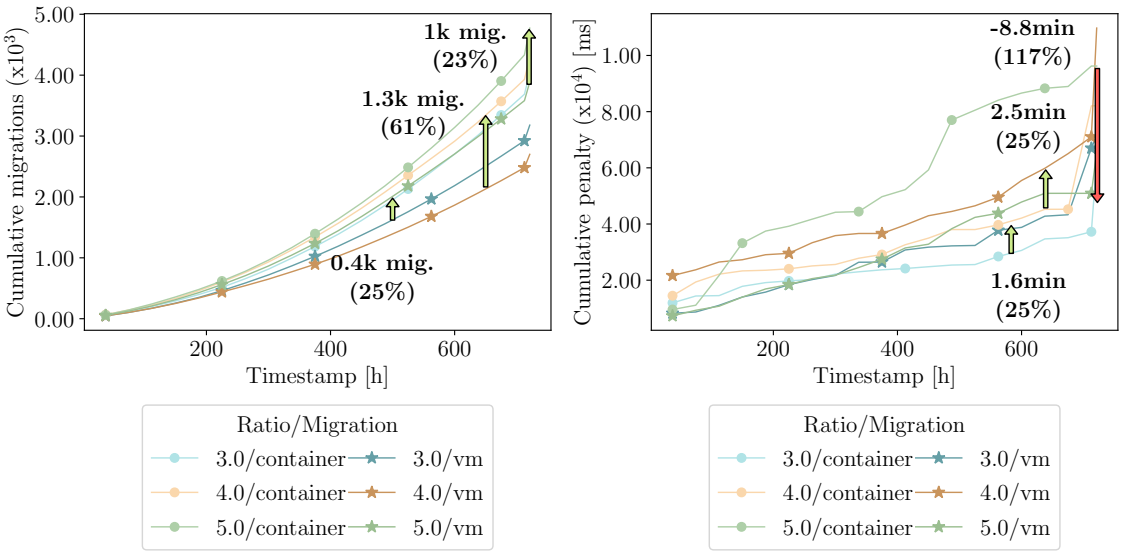


Figure 5: Migrations cumulative amount (left) and migrations cumulative penalty (right) of Azure trace. Each line, bar, or boxplot represents a different `<Oversubscription ratio>/<Migrations API>` configuration.

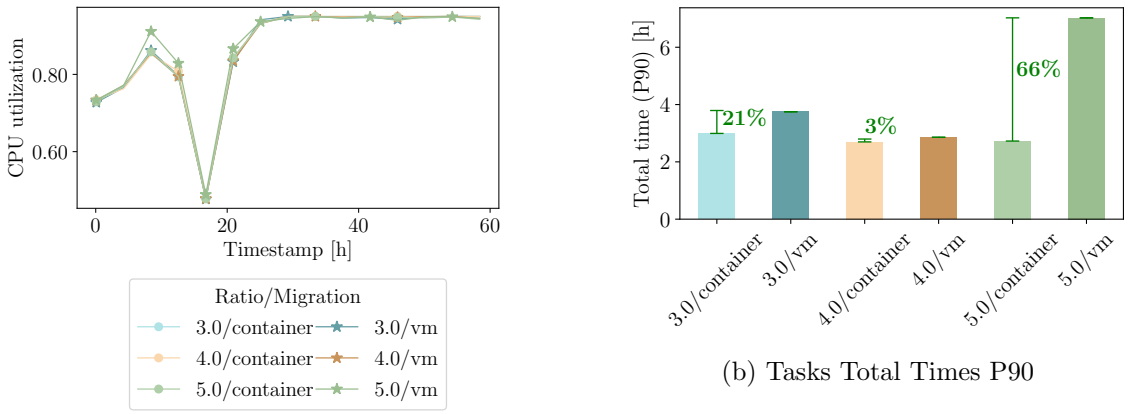
Furthermore, in this experiment, we demonstrate that container migrations improve performance and that offering it as a programming abstraction is necessary. The user does not always have a second scheduling layer like a Kubernetes cluster. Moreover, the datacenter schedulers do not have the business logic knowledge to decide which tasks to migrate and where. Therefore, the scheduler cannot internally implement the container migration logic without exposing programmability to the user. Our experiments simulate these neutral cases when we deactivate the container migrations.

Lastly, to understand how migrations affect performance, we also show graphs on the number of migrations and their penalties. This helps to understand how the Bitbrains trace, despite having a lot of migrations, does not show almost any improvement in performance because it also presents the highest penalties. However, it is not straightforward to understand how the penalties and migrations interplay. This creates an opportunity for future research to better understand how migrations affect task scheduling performance.

14 Conclusion

In this work, we conducted a comprehensive analysis to evaluate the impact of container migrations on scheduling performance in datacenter environments. First we analyzed five real industrial schedulers, namely Kubernetes, SLURM, Spark, Condor, and Airflow, and we aimed to understand the extent to which these schedulers can implement the container migrations API and whether they can effectively leverage its benefits.

By analyzing each scheduler, we found that Kubernetes and SLURM provide native support for callbacks and can readily incorporate the container migrations API. Spark and Airflow offer callback functionality but require an extension to include migration-specific



(a) Packing efficiency

(b) Tasks Total Times P90

Figure 6: Packing efficiency (left) and tasks Total Times P90 (right) of Google trace. Each line and bar represents a different `<Oversubscription ratio>/<Migrations API>` configuration.

methods. Condor, on the other hand, does not currently implement callbacks and, thus, cannot directly support container migrations based on our proposed extension. These findings highlight the varying capabilities and limitations of existing industrial schedulers in implementing the container migrations API.

Second, we proceeded with a series of experiments to assess the performance improvements achieved through container migrations. To facilitate this analysis, we extended OpenDC, a renowned open-source datacenter discrete event simulator developed by At-Large. This extension enabled us to simulate a second layer of scheduling, perform VM migrations, and incorporate container migrations on the second layer. To ensure transparency and foster further research in the field, we have made our implementation publicly available, including all the relevant code and data utilized in our experiments at `https://github.com/aratz-lasa/opendc`. By sharing these resources, we aim to facilitate reproducibility and encourage the advancement of research in this domain.

Our experiment yielded significant results regarding the impact of container migrations on scheduling performance. In the Bitbrains trace, the use of the container migrations API resulted in a 4% higher utilization of resources. However, due to associated penalties, this increase in packing did not translate into noticeable performance improvements in terms of execution times. In the Azure trace, we observed that using the API with a ratio of 5.0 led to a 260-hour reduction (81%) in the 90th percentile total times compared to configurations without the API. The Google trace demonstrated that although packing differences were minimal, the API consistently achieved shorter total times, with a 66% reduction (4 hours) in the 90th percentile times when using a ratio of 5.0.

Our findings reinforce the importance of providing container migrations as a programming abstraction in schedulers. The experiment demonstrated that leveraging the container migrations API significantly improved scheduling performance, resulting in better resource utilization and shorter execution times. Furthermore, it emphasized the trade-off between simplicity and performance, indicating that schedulers lacking callback functionality need to expose programmability to users for effective implementation of container

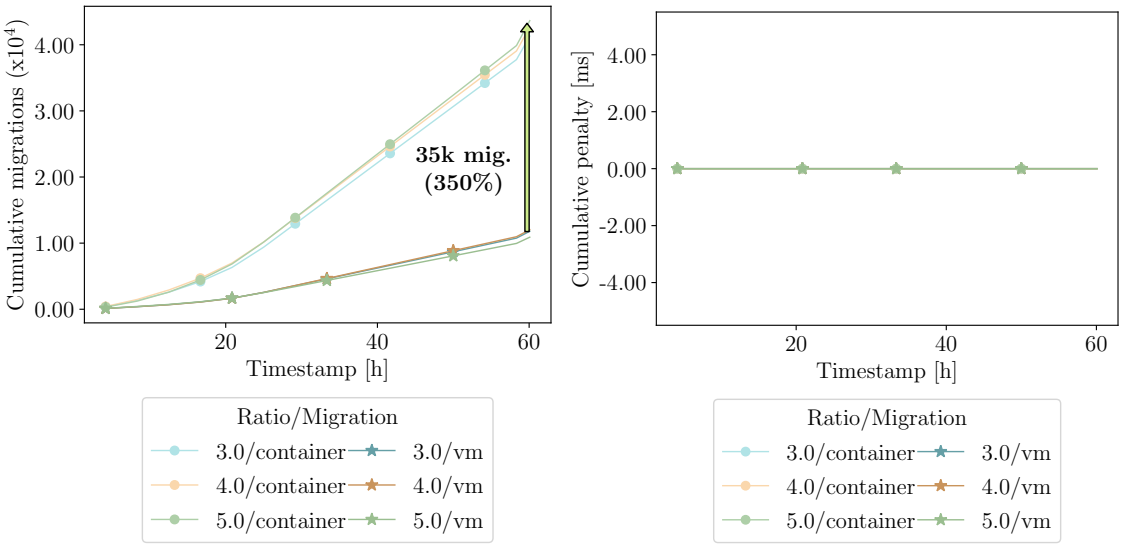


Figure 7: Migrations cumulative amount (left) and migrations cumulative penalty (right) of Google trace. Each line, bar, or boxplot represents a different `<Oversubscription ratio>/<Migrations API>` configuration.

migrations.

Lastly, this work is part of a broader research project on programming abstractions in datacenter scheduling. Specifically, we assessed the capabilities of existing industrial schedulers in implementing advanced abstractions, focusing on container migrations. Through detailed examination and extensive experiments, we gained valuable insights into the limitations of current abstractions and highlighted the potential performance improvements with a dedicated migration API.

References

[1] Apache airflow. `https://github.com/apache/airflow`, 2023.

[2] Kubernetes. `https://github.com/kubernetes/kubernetes`, 2023.

[3] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 153–167. ACM, 2017.

[4] Tomi Dufva and Mikko Dufva. Grasping the future of the digital society. *Futures*, 107:17–28, 2019.

[5] F Gens. Worldwide and regional public it cloud services, 2014.

[6] Michael Haenlein and Andreas Kaplan. A brief history of artificial intelligence: On the past, present, and future of artificial intelligence. *California management review*, 61(4):5–14, 2019.

[7] Rachel Householder, Scott Arnold, and Robert Green. Simulating the effects of cloud-based oversubscription on datacenter revenues and performance in single and multi-class service levels. In *2014 IEEE 7th International Conference on Cloud Computing, Anchorage, AK, USA, June 27 - July 2, 2014*, pages 562–569. IEEE Computer Society, 2014.

[8] Paulo Souza Junior, Daniele Miorandi, and Guillaume Pierre. Good shepherds care for their cattle: Seamless pod migration in geo-distributed kubernetes. In Lena Mashayekhy, Stefan Schulte, Valeria Cardellini, Burak Kantarci, Yogesh Simmhan, and Blesson Varghese, editors, *6th IEEE International Conference on Fog and Edge Computing, ICFEC 2022, Messina, Italy, May 16-19, 2022*, pages 26–33. IEEE, 2022.

[9] Ram Srivatsa Kannan, Animesh Jain, Michael A. Laurenzano, Lingjia Tang, and Jason Mars. Proctor: Detecting and investigating interference in shared datacenters. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2018, Belfast, United Kingdom, April 2-4, 2018*, pages 76–86. IEEE Computer Society, 2018.

[10] Don Lipari. The slurm scheduler design. In *SLURM User Group Meeting, Oct*, volume 9, page 52, 2012.

[11] Daniel S. Marcon and Marinho P. Barcellos. Packer: Minimizing multi-resource fragmentation and performance interference in datacenters. In *2017 IFIP Networking Conference, IFIP Networking 2017 and Workshops, Stockholm, Sweden, June 12-16, 2017*, pages 1–9. IEEE Computer Society, 2017.

[12] Fabian Mastenbroek, Georgios Andreadis, Soufiane Jounaid, Wenchen Lai, Jacob Burley, Jaro Bosch, Erwin Van Eyk, Laurens Versluis, Vincent van Beek, and Alexandru Iosup. Opendc 2.0: Convenient modeling and simulation of emerging technologies in cloud datacenters. In Laurent Lefèvre, Stacy Patterson, Young Choon Lee, Haiying Shen, Shashikant Ilager, Mohammad Goudarzi, Adel Nadjaran Toosi, and Rajkumar Buyya, editors, *21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2021, Melbourne, Australia, May 10-13, 2021*, pages 455–464. IEEE, 2021.

[13] Pascale Minet, Eric Renault, Ines Khoufi, and Selma Boumerdassi. Analyzing traces from a google data center. In *14th International Wireless Communications & Mobile Computing Conference, IWCMC 2018, Limassol, Cyprus, June 25-29, 2018*, pages 1167–1172. IEEE, 2018.

[14] Saloua El Motaki, Ali Yahyaouy, and Hamid Gualous. A prediction-based model for virtual machine live migration monitoring in a cloud datacenter. *Computing*, 103(11):2711–2735, 2021.

[15] Tech. Rep. *2022 Leadership Vision for Infrastructure and Operations*. Gartner, 2022.

[16] Haiying Shen and Liuhua Chen. A resource usage intensity aware load balancing method for virtual machine migration in cloud datacenters. *IEEE Trans. Cloud Comput.*, 8(1):17–31, 2020.

[17] Siqi Shen, Vincent van Beek, and Alexandru Iosup. Statistical characterization of business-critical workloads hosted in cloud datacenters. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015*, pages 465–474. IEEE Computer Society, 2015.

[18] Mirsaeid Hosseini Shirvani, Amir Masoud Rahmani, and Amir Sahafi. A survey study on virtual machine migration and server consolidation techniques in dvfs-enabled cloud datacenter: Taxonomy and challenges. *J. King Saud Univ. Comput. Inf. Sci.*, 32(3):267–286, 2020.

[19] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurr. Pract. Exp.*, 17(2-4):323–356, 2005.

[20] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 30:1–30:14. ACM, 2020.

[21] Xuan-Tuong Vu, Minh-Ngoc Tran, and Young-Han Kim. Service migration over edge computing infrastructure. *Proceedings of the Korean Institute of Communication Sciences Annual Conference*, pages 138–139, 2021.

[22] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.