A Reference Architecture for Datacenter Scheduler Programming Abstractions: Design and Experiments (Work In Progress Paper)

Aratz Manterola Lasa
Vrije Universiteit Amsterdam
The Netherlands
A.M.Lasa@atlarge-research.com

Sacheendra Talluri
Vrije Universiteit Amsterdam
The Netherlands
s.talluri@vu.nl

Alexandru Iosup
Vrije Universiteit Amsterdam
The Netherlands
a.iosup@vu.nl

ABSTRACT

Datacenters are the backbone of our digital society, used by the industry, academic researchers, public institutions, etc. To manage resources, data centers make use of sophisticated schedulers. Each scheduler offers a different set of capabilities and users make use of them through the APIs they offer. However, there is not a clear understanding of what programming abstractions they offer, nor why they offer some and not others. Consequently, it is difficult to understand the differences between them and the performance costs that are imposed by their APIs. In this work, we study the programming abstractions offered by industrial schedulers, their shortcomings, and the performance costs of the shortcomings. We propose a general reference architecture for scheduler programming abstractions. Specifically, we analyze the programming abstractions of five popular industrial schedulers, we analyze the differences in their APIs, we identify the missing abstractions, and finally, we carry out an exemplary experiment to demonstrate that schedulers sacrifice performance by under-implementing programming abstractions. In the experiments, we demonstrate that an API extension can improve task runtime by up to 23%. This work allows schedulers to identify their shortcomings and points of improvement in their APIs, but most importantly, provides a reference architecture for existing and future schedulers.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **General and reference → Design**.

KEYWORDS

scheduler, API, design, performance

ACM Reference Format:
Aratz Manterola Lasa, Sacheendra Talluri, and Alexandru Iosup. 2023. A Reference Architecture for Datacenter Scheduler Programming Abstractions: Design and Experiments (Work In Progress Paper). In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3578245.3585035

1 INTRODUCTION

Datacenters are used by industry, academic researchers, public institutions, etc. to deploy their services and products. They have become the main infrastructure of our digital society [13, 27]. To manage resources, data centers make use of sophisticated schedulers. In order to improve the scheduling, the community has designed many different schedulers over the years, such as Omega [29], Mesos [37], Condor [33], Protean [16], etc. All these schedulers share a common reference architecture of the scheduling process [2]. However, regarding the interface they provide to the users, that is to say, their Application Programming Interfaces (APIs), there has been little work. A conceptual model that aims to capture the programming abstractions across different types of schedulers (data analysis, datacenter resource management, containers orchestration, etc.) has never been designed. Such a conceptual model would be beneficial to generate a base knowledge on how to design and build the systems, as well as ease the understanding of how schedulers work and what functionalities they provide [7, 12, 23]. It would also allow us to identify shortcomings in existing schedulers and provide a framework for comparing different designs. On the other hand, the lack of a conceptual model for scheduling programming abstractions can be costly. Ill-defined abstractions make it hard to port scheduling innovations to established schedulers. Hence, established schedulers get stuck with old designs. Often, they require a significant redesign to accommodate new innovations, as has been the case with Condor [33] and Borg [6].

There is not a clear understanding of why a scheduler does or does not offer a certain API to the user. Instead, the concept has always been reduced to (1) *the richer the API, the higher the performance of the user applications is* [29] and (2) *the more limited the API, the higher the simplicity, security, and provider control is* [37]. This is probably why schedulers limit the APIs, in exchange for greater control and simplicity. However, it is not clear to existing schedulers what performance benefits are sacrificed, due to simple API designs [1, 28, 35]. Therefore, it is necessary to study the costs imposed by the existing programming interfaces. This will allow vendors to realize the importance of the programming abstractions, and quite possibly update them to increase performance for their customers.

How can we model scheduler programming abstractions?, *Are there shortcomings in the programming abstractions of industrial schedulers?* and *What are the costs imposed by the shortcomings?* are the main research questions we address in this work. For building schedulers, it is necessary to have a clear understanding of what programming abstractions the scheduler could expose to their users. In other words, what is the potential functionality of a scheduler is. For that, we design a reference architecture for scheduling programming abstractions. Once the reference architecture is designed, we use it to analyze the industrial schedulers and identify shortcomings of their current APIs. Lastly, we hypothesize that existing schedulers sacrifice performance in exchange for simplicity, by limiting their programming interfaces. Therefore, we carry out experiments to prove that by extending current scheduler APIs, performance is gained. For that, we take an existing programming interface, identify missing features based on the reference architecture, and we experiment by implementing the missing abstraction and comparing it to the original design.

The contribution of this work is three-fold:

(1) We design a reference architecture for scheduling programming abstractions (Section 3).
(2) We analyze existing industrial scheduler APIs by mapping them to the reference architecture (Section 4).
(3) We evaluate, using simulation, the performance cost of missing programming abstractions in industrial schedulers (Section 5).

2 BACKGROUND

For encapsulating the context and explaining the central concepts of this work, we present a set of scheduling system models that we explain below.

The workload is what is executed using the resources that the scheduler assigns to the user. In this work, we assume they fit the morphology of a workflow: a stream of jobs that are made up of one or several tasks, and there are dependencies in the precedence between the tasks.

The scheduling resources are the resources the scheduler manages and what workloads are executed on top of. Resources are physical machines, in a data center with several hosts each, and each host virtualizes its resources in VMs or containers, and they are a combination of CPU, memory RAM, and storage.

The scheduler is the central component users submit the workload, in order to make use of the resources. It takes care of several tasks: finding resources to assign to the user workload, transferring the workload to the resources, starting the execution of the workload, managing the workload through its lifecycle, and notifying to the user about lifecycle events.

The programming abstractions are the API offered by schedulers and is the language by which the user submits workloads and modifies the workload's requirements during the workload's life-cycle.

3 REFERENCE ARCHITECTURE DESIGN

In this section, we design the reference architecture. Below we specify the methodology we follow.

(1) **Analysis of requirements and design principles**. First, we identify the requirements of the reference architecture and the design principles by which we guide and evaluate the design.
(2) **Model real-world schedulers**. Next, we model the programming abstractions of five real-world schedulers. For that, we identify five popular schedulers in the industry, and we analyze their APIs. Consulting experts in the field we select the following schedulers: Kubernetes [24], SLURM [22], Spark [36], Condor [33], and Apache Airflow [31].
(3) **Model emerging concepts from academia**. Then, we model scheduler designs from emerging fields, such as IoT/Edge, energy efficiency, etc. For that, we carry out a literature survey. The schedulers we identify in the literature survey are: [5, 8–11, 14, 19, 21, 25, 26, 30, 32, 34, 35, 38].
(4) **Unify real-world and emerging concepts from academia**. After modeling real-world and emerging scheduler designs, we extract, filter, generalize, and unify them into a reference architecture.

In this work, we omit the intermediate steps and the specification of the methodology for selecting the academic schedulers, and we only present the requirements analysis and the final result of the reference architecture.

3.1 Requirements

To design a reference architecture it is necessary to identify which are the requirements that must be met. We list all the requirements below.

R1 Comprehensibility. The reader should not make a great effort to understand the different components that make up the reference architecture, how they relate to each other, or what their high-level meaning is.
R2 Actionable. The main driver of the reference architecture is to be used in the real world. Therefore, the design must take into account whether the resulting work is actionable.

3.2 Design principles

For the design of the scheduling programming abstractions reference architecture, we identify the following design principles.

P1 Separation of Objects from Actions. We distinguish between the actions that can be performed and the objects that are used as input to the actions. This separation facilitates comprehension.
P2 Grouping of related actions. There may be several actions that are related to each other. Therefore, to facilitate comprehension, related actions are grouped together.
P3 Avoidance of concrete technologies in objects. For avoiding to commit or couple to a specific technology, we keep the objects as high-level as possible.
P4 Naming relationships between actions and objects. The reference architecture must relate the objects to the actions, through named relationships.

3.3 Reference architecture

The reference architecture is found in Figure 1. The high-level approach of the model is based on listing the actions and objects that the APIs are composed of, and how objects relate to actions. The objects are the input that the actions receive. Each action must have three types of relationships: WHAT, WHEN, and WHERE, and for each relationship there can be one or more objects. This way, programming abstractions can be understood through the following syntactic structure: `<action> <object> IN <object>`

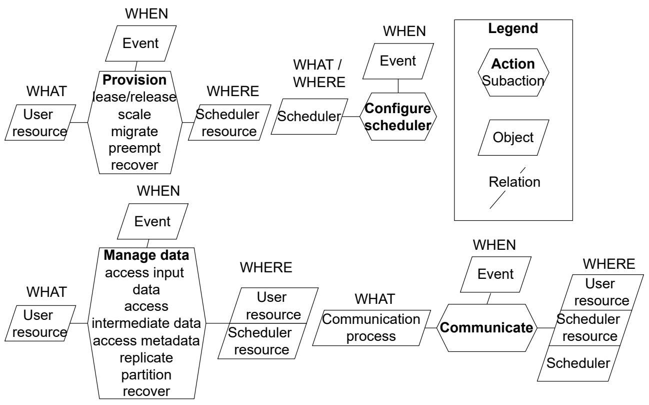

Figure 1: Reference architecture for scheduling programming abstractions.

Table 1: Full overview of programming abstraction $\boxed{\text{actions}}$ of schedulers mapped to the reference architecture.

Action	Subaction	Schedulers				
		Kubernetes	SLURM	Spark	Condor	Airflow
Provision	lease / release	✓	✓	✓	✓	✓
	scale	✓		~		
	migrate					
	preempt	~	✓		✓	
	recover	✓	~	✓	~	~
Configure scheduler		✓	~	✓	✓	✓
Manage data	access input data	✓	~	✓	✓	✓
	access intermediate data			~		
	access metadata					
	replicate					
	partition			✓		
	recover	~		✓	✓	
Communicate		~	✓	~	~	~

Legend: ✓/~ /() = *full*/*partial*/*nomatch*.

Table 2: Full overview of programming abstraction $\boxed{\text{objects}}$ of schedulers mapped to the reference architecture.

Action	Object	Schedulers				
		Kubernetes	SLURM	Spark	Condor	Airflow
Provision	user resource	✓	~	~	✓	✓
	event	✓	~	~	~	~
	scheduler resource	✓	✓	~	✓	✓
Configure scheduler	scheduler	✓	~	✓	✓	✓
	event	✓	~	~	~	~
Manage data	user resource	~	~	✓	~	✓
	event	~		~		~
	scheduler resource	~	~	~	~	~
Communicate	communication process	✓	~	~	~	✓
	event	✓	~	✓		✓
	user resource	✓	~	✓	~	✓
	scheduler resource	~	~	✓		~
	scheduler	~	~	~		~

Legend: ✓/~ /() = *full*/*partial*/*nomatch*.

`WHEN <object>`, where the objects and actions are filled using the reference architecture. For example:

- `Provision:Lease UserResource<type:job, runtime:5 days> IN SchedulerResource<type:vm, cpu:2.4Ghz, memory:16Gb> WHEN Event<day:31, month:12, year:2022>`.
- `Provision:Scale UserResource<type: application> IN SchedulerResource<type:vm, cpu:2.4Ghz, memory:16Gb> WHEN Event<cpu-utilization:> 80%>`

Next we define each of the objects and actions of the reference architecture.

The objects are the following:

- **Event**: objects in time or instantiations of properties in objects. Such as concrete date-times (00:00 of 31st of December 2022) or an instantiation of a property like a metric reaching a numeric value (CPU utilization is greater than 80%).
- **User resource**: representation of any kind of input from the user. This includes execution units like a job, task, etc. but also data as a file, environment variable, etc.
- **Scheduler resource**: Representation of resources owned and managed by the scheduler. Resources can be virtual machines, containers, storage systems, databases, etc.
- **Communication process**: Representation of the process of communication, such as a signal, message, callback, etc.

The actions can be main actions or sub-actions, this allows the grouping of actions by theme and facilitates understanding. The actions are the following:

- **Provision**: Provisioning of resources.
  - **Lease / release**: Activation and assignment of a user resource to a scheduler resource.
  - **Scale**: Addition or reduction of already provisioned user resources.
  - **Migrate**: Migration of a user resource to a different scheduler resource.
  - **Preempt**: Abortion of execution or assignment of a user resource, putting it back in the scheduler queue.
  - **Recover**: Recovery of a user resource after a failure, restarting the execution, or putting it back into the scheduler queue.
- **Configure scheduler**: Configuration of the behavior of the scheduler.
- **Manage data**: Management of the user data.
  - **Access input data**: Access to data that user jobs take as input.
  - **Access intermediate data**: Access to data that user jobs generate during their runtime.
  - **Access metadata**: Access to the information about the user data.
  - **Replicate**: Replication of the user data.
  - **Partition**: Partitioning of the user data, so that subset of the data is placed in different scheduler resources.
  - **Recover**: Recovery of the user data after the failure of execution or the storage system.
- **Communicate**: Communication with the user resources, scheduler resources, or even the scheduler, such as setting a callback for getting notified about scheduling events.

4 ANALYSIS OF INDUSTRIAL SCHEDULERS

Using the reference architecture we analyze the shortcomings of the selected group of five industrial schedulers. Currently, it is not known when nor why you should use some schedulers and not others. It is also not clear if any scheduler has a clear missing gap, nor how to fill those gaps. For that, it is necessary to analyze the scheduling APIs. We map their APIs into the reference architecture and we aggregate the results in two tables. In Table 1 we map the actions and in Table 2 the objects. For each action and object we specify if it is a full, partial, or no match. We label it as *full match* if we can find the component and the API is flexible enough to accept any input the user comes up with. We label it as *partial match* if we can find the component but the inputs are limited to a specific subset. The remaining components we label them as *no match*. Lastly, from here on out, words with bounding boxes are components of the reference architecture

The results indicate that industrial schedulers have several shortcomings. There are several actions that are under-implemented. There is a very clear pattern, where most schedulers fully implement four actions: $\boxed{\text{lease / release}}$, $\boxed{\text{configure scheduler}}$, $\boxed{\text{access input data}}$ and $\boxed{\text{communicate}}$. All others, in most cases, are either partially implemented or not implemented at all. The biggest shortcoming is found in $\boxed{\text{manage data}}$ action and its objects, where most of the sub-actions and objects are not implemented. That is, industrial schedulers are not designed to manage and schedule data. This means that users have less control over the data, and consequently less chance to optimize performance. For example, if the user has several unordered data items to process, consulting the metadata and obtaining information about the placement and requests load of the storage systems where the data is stored, could optimize how and when the data is processed. Secondly, the $\boxed{\text{communicate}}$ action, except in SLURM, in all other cases is partially implemented. Similarly, most communication objects are partial matches. This implies a lower performance since it does not allow the user to inform the scheduler during runtime about application-level insights, nor vice versa, the scheduler to inform the user about scheduling-level insights. Moreover, partial matches imply that actions and objects are limited to a particular subset, and therefore do not allow the user to specify arbitrary inputs. For example, the Condor API provides communication actions, but only with user jobs, not with the scheduler. Therefore, the user can dynamically inform about application-level insights to their jobs, but not to the scheduler, reducing the scope of potential performance improvements.

In conclusion, there are several shortcomings in the programming abstractions of industrial schedulers. Many objects have partial or no matches, meaning their APIs are under-implemented, and consequently, they reduce the ability and scope of the user to optimize the performance of their jobs. The main shortcomings are found in $\boxed{\text{manage data}}$ action and its objects. But also to a lesser extent in $\boxed{\text{communicate}}$ actions and their objects.

5 EVALUATING THE PERFORMANCE COSTS OF SIMPLE SCHEDULING ABSTRACTIONS

We state that schedulers prioritize simplicity in their programming models, and thus, they limit the APIs. We hypothesize this simplicity has costs that mainly translate into the lower performance of user applications. We aim to prove that it is necessary for schedulers to revise their APIs, in order to improve user-applications performance.

To support the hypothesis, we choose a scheduling API shortcoming that we found in the previous section, and we identify a concrete use case in which the under-implemented programming abstractions are required. Then, we carry out experiments, by extending the programmability of the scheduler to include the missing abstractions.

5.1 Selection of an API shortcoming

In this section, we choose an under-implemented programming abstraction in the scheduler APIs that we will use to perform the experiments. The shortcomings are obtained from the mapping carried out in the previous section.

Experiment name: Reducing VM total times using user-level migrations.

Use-case: When there are interferences in a VM, the scheduler requests the user to migrate or reduce part of the workload to another VM through a callback.

Ideal scheduler: User performs a $\boxed{\text{communicate}}$ action specifying a $\boxed{\text{communication process}}$ which is a callback, an $\boxed{\text{event}}$ which identifies when there is interference, and a $\boxed{\text{scheduler}}$ identifying the scheduler that uses the callback. This way, when there are interferences, the scheduler activates the callback, and the user migrates the workload.

Industrial scheduler shortcomings: Condor cannot perform this operation since the $\boxed{\text{communicate}}$ action does not support $\boxed{\text{communication process}}$ objects of callback type, nor does it implement $\boxed{\text{event}}$ objects.

Extension based in the reference architecture: Condor needs to extend its $\boxed{\text{communicate}}$ action to accept any type of $\boxed{\text{communication process}}$ object to receiving callbacks, and also accept $\boxed{\text{event}}$ objects so that the communication can be done at a specific scheduling event.

5.2 System model

In a data center, there are multiple tenants that lease and release virtual machines, through a scheduler that has equivalent API shortcomings to Condor's. Each tenant deploys an arbitrarily sized Kubernetes cluster on top of the leased virtual machines, and on each VM, one or more batch tasks called pods are executed.

Kubernetes clusters lead to under-utilization of resources at times, and consequently, the provider oversubscribes resources. So, on high load spikes, tenants sharing physical machines may suffer interferences between them. When that happens, the scheduler tries to migrate VMs to other physical machines to reduce interferences.

The API offered by the data center to users can be simplified as:

- `lease(requirements): vm`: the user passes a list of resource requirements, the provider boots up a virtual machine with those requirements, and returns the machine to the user.

5.3 Model extension

We extend the model to include the ability to be callbacked by the Condor equivalent data center scheduler when a machine is oversubscribed. For submitting callbacks, the scheduler provides a communicate action that accepts: 1) a communication process by which the users submit callbacks, 2) a scheduler that specifies the scheduler that uses the callback, and 3) an event by which users specify when the VM is oversubscribed. The callback that users submit is named `requestMigration`. When calling it, Kubernetes is requested to migrate pods to another node. This way, we expect to increase the performance of the tasks that are executed, by reducing the size of migrations, since pods are smaller than VMs. Thus, we expect machines to get better packing and consequently less interference and greater performance. The extended programming model offered by the scheduler is the following:

- `communicate(communicationProcess, scheduler, event)`: the user specifies a communicationProcess that contains the callback, a scheduler that will use the callback, and the event at which the callback is activated.
- `requestMigration(vm, cpuCapacity)`: `cpuCapacity`: the scheduler specifies the oversubscribed VM and the CPU capacity on Gigahertz (GHz).to be migrated. The user returns the CPU capacity. that will migrate.

In Figure 2 we show the diagram of the system and the extension.

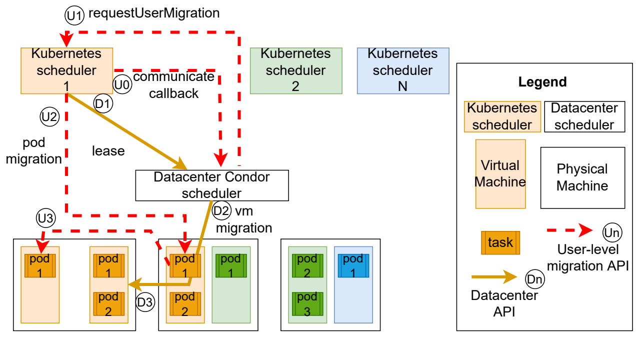

Figure 2: User-level migrations experiment system model. The number beside the API action denotes the order of API calls.

5.4 Execution, Configuration, and design of the experiment

We run the experiments on a personal laptop with an Apple M1 Max chip, 1TB SSD storage, and 32 GB memory. The experiment configurations are composed of a combination of three dimensions: trace, user-level migrations, and oversubscription ratio. We experiment with three oversubscription ratios: 3, 4, and 5. Regarding user-level migrations, we experiment if it is used or not through the callback API extension. Lastly, we use three real-world anonymized trace workloads from private and public cloud environments. The chosen traces are Bitbrains Azure and Google. While Bitbrains and Azure traces are VM requests, Google traces are task requests. In Azure, we sample 1829 VMs from the original trace and in Google, we sample 79820 requests from the original trace in 2.5 days. Lastly, Bitbrains is a 1250 VMs trace of a dutch private cloud provider

The artifacts used in the experiment are available in https://github.com/aratz-lasa/opendc.

5.5 Results

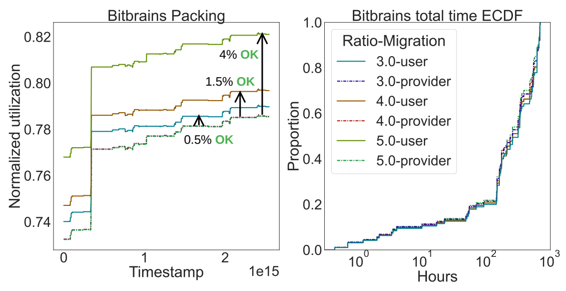

Figure 3: Tasks packing and total times ECDF of Bitbrains trace.

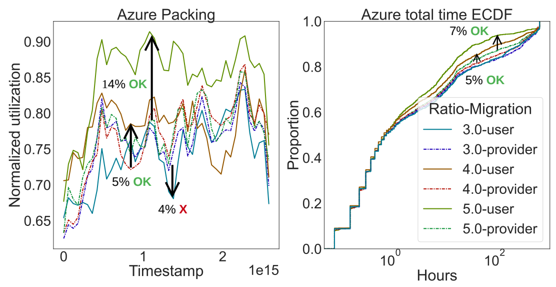

Figure 4: Tasks packing and total times ECDF of azure trace.

In Figures 3, 4, 5 we show the results of the Bitbrains, Azure and Google traces, for each combination of oversubscription ratio and the activation (or not) of the callback API for user-level migrations. On the left, we present the aggregated utilization along the data center's physical machines. This represents the packing that is obtained in each configuration of the experiment. On the right, we show the ECDF of the total time of each configuration. The total time is the sum of the waiting time and the execution time. That is because the user requests the provisioning until it finishes running the task.

In the packing graphs, it is clearly seen that in Bitbrains and Azure the configurations that use the API obtain better packing, around 3% and 10% higher utilization, respectively. However, while in Azure this translates into shorter times, in Bitbrains, all configurations get a similar result. This is because Bitbrain's 3% higher utilization doesn't make much of a difference, while the 10 % of Azure does.

The Google trace, unlike the other traces, has almost no differences found in the packing of different configurations. However,

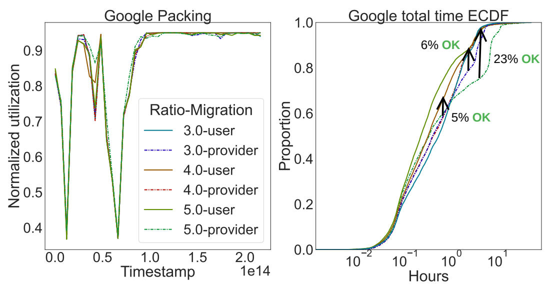

Figure 5: Tasks packing and total times ECDF of google trace.

when using the API, shorter times are obtained. This is because the Google trace tasks are small and they use a single CPU core. This means that the differences in the packing cannot be perceived, because the tasks last so little that the space is rapidly freed up. Nonetheless, there is a significant performance improvement in times, where the 5.0 ratio has 23% more tasks at lower times.

5.6 Discussion

The main findings from this experiment are:

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

MF2 The highest oversubscription ratio of 5.0 obtains the highest performance improvement using user-level migrations.

MF3 The main benefit of migrations is greater packing, that is, greater utilization of resources.

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

The objective of this experiment is to demonstrate that schedulers may be sacrificing performance in exchange for simplicity if they do not offer callbacks to their users. The most important takeaway from the results of this experiment is that in all configurations, except for Bitbrains, performance is increased using the extended API. Depending on the workload and oversubscription ratio, the performance improvement is different. The highest oversubscription ratio of 5.0 obtains the highest performance improvements. In addition, in Bitbrains no improvements are found. So user-level migrations should not be always used.

In the experiment, we not only demonstrate that making use of user-level migrations improves performance, but also that it is necessary to offer it as a programming abstraction. This is because the user does not always have a second layer of scheduling such as a Kubernetes cluster, nor do the schedulers have the business logic knowledge to decide what tasks to migrate and where. Therefore, the scheduler cannot internally implement the user-level migration logic, without exposing programmability to the user.

6 CRITICAL DISCUSSION

The work, in its current form, has 3 main weaknesses. First, we only evaluate a concrete case, and it is not enough to fully validate the hypothesis that the existing scheduling APIs sacrifice performance for being under-implemented. Second, we have not yet validated the reference architecture, by mapping to it real-world, well-known, and state-of-the-art schedulers. Third, a reference architecture is always limited in that it is kept at a sufficiently high abstraction layer to map and represent all kinds of schedulers, consequently, it is not capable of representing all the low-level details that allow differentiating one scheduling API from another.

The main limitations of the design are found in the *objects*. In the reference architecture, there are only 5 distinct objects, and for each of them, we do not specify any sub-objects. For example, one of the objects is the $\boxed{\text{Scheduler Resource}}$, which does not differentiate between an API that offers VMs or Edge mobile devices. This is a limitation, but it is made on purpose to be future-proof, since if there is one thing certain it is that the type of resource is constantly changing. That is why instead of differentiating objects by their content, we differentiate them by what they represent in the highest level of abstraction.

7 RELATED WORK

We are not aware of any other work designing a reference architecture for scheduling programming abstractions. On the one hand, there are several models that focus on modeling the scheduling process instead of the APIs. Among these works is Schopf's multi-stage model of the grid scheduling process [18], the consequent work in Global Grid Forum [15], and the datacenter scheduler reference architecture [3]. However, these works are not replacements but rather complementary to our model, since they model the internal process of a scheduler, while we model the external interface offered to users.

There are also conceptual models of APIs but of specialized systems. Among the most relevant is the reference architecture of grid computing such as the work of Foster et al. [12]. Similarly, models for cloud computing have also been proposed by the National Institute of Standards and Technology (NIST) [23]. However, all these works are specialized in specific environments and therefore are not applicable to a large part of schedulers like Spark and Kubernetes.

Lastly, several works focus on developing systems trying to combine various scheduling abstractions into a single scheduler, which they generalize the APIs so that their system models can be compared to our work [4, 17, 20]. For example, Ghost offers a model for delegating kernel scheduling decisions to the users [17], and ESCHER presents a model for letting users express arbitrary scheduling constraints as resource requirements [4]. However, these schedulers are too general like Ghost and they cannot map schedulers, or else, they only focus on affecting a subset of the scheduling functionalities, like ESCHER, which only focuses on offering provisioning constraints. None of them aims to identify and unify all the programming abstractions that a scheduler can potentially provide.

8 CONCLUSION

With the increasing digitization of society, efficient management of compute resources is important, and consequently an efficient design of the schedulers. Currently, it is difficult to understand and compare the API features offered by schedulers. To address this problem we design a reference architecture for scheduling programming abstractions. In addition, with this reference architecture, we identify that the existing industrial schedulers have several shortcomings, among others, to manage and schedule data, as well as communicating between the components that interact in scheduling. Lastly, through experimentation, we demonstrate that these shortcomings can suppose a relevant cost in the performance of the users.

REFERENCES

[1] Isam Mashhour Al Jawarneh, Paolo Bellavista, Filippo Bosi, Luca Foschini, Giuseppe Martuscelli, Rebecca Montanari, and Amedeo Palopoli. 2019. Container orchestration engines: A thorough functional and performance comparison. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, 1–6.

[2] Georgios Andreadis, Laurens Versluis, Fabian Mastenbroek, and Alexandru Iosup. 2018. A reference architecture for datacenter scheduling: design, validation, and experiments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE / ACM, 37:1–37:15. http://dl.acm.org/citation.cfm?id=3291706

[3] Georgios Andreadis, Laurens Versluis, Fabian Mastenbroek, and Alexandru Iosup. 2018. A reference architecture for datacenter scheduling: design, validation, and experiments. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 478–492.

[4] Romil Bhardwaj, Alexey Tumanov, Stephanie Wang, Richard Liaw, Philipp Moritz, Robert Nishihara, and Ion Stoica. 2022. ESCHER: expressive scheduling with ephemeral resources. In *Proceedings of the 13th Symposium on Cloud Computing*. 47–62.

[5] Luiz F Bittencourt, Javier Diaz-Montes, Rajkumar Buyya, Omer F Rana, and Manish Parashar. 2017. Mobility-aware application scheduling in fog computing. *IEEE Cloud Computing* 4, 2 (2017), 26–35.

[6] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, omega, and kubernetes. *Commun. ACM* 59, 5 (2016), 50–57.

[7] Wo L Chang, David Boyd, Orit Levin, et al. 2019. NIST Big Data Interoperability Framework: Volume 6, Reference Architecture. (2019).

[8] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. 2000. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of network and computer applications* 23, 3 (2000), 187–200.

[9] Carlo Curino, Djellel E Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. 2014. Reservation-based scheduling: If you're late don't blame us!. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–14.

[10] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices* 48, 4 (2013), 77–88.

[11] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices* 49, 4 (2014), 127–144.

[12] Ian Foster, Carl Kesselman, and Steven Tuecke. 2001. The anatomy of the grid: Enabling scalable virtual organizations. *The International Journal of High Performance Computing Applications* 15, 3 (2001), 200–222.

[13] F Gens. 2014. Worldwide and Regional Public IT Cloud Services.

[14] Robert Grandl, Arjun Singhvi, Raajay Viswanathan, and Aditya Akella. 2021. Whiz:{Data-Driven} Analytics Execution. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 407–423.

[15] Christian Grimme, Joachim Lepping, Alexander Papaspyrou, Philipp Wieder, Ramin Yahyapour, Ariel Oleksiak, Oliver Wäldrich, and Wolfgang Ziegler. 2008. Towards a standards-based grid scheduling architecture. In *Grid Computing*. Springer, 147–158.

[16] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E. Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. 2020. Protean: VM Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 845–861. https://www.usenix.org/conference/osdi20/presentation/hadary

[17] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 588–604.

[18] M Schopf Jennifer. 2004. Ten Actions When Grid Scheduling: The User as a Grid Scheduler. *Grid Resource Management: State of the Art and Future Trends, Norwell, MA, USA, Kluwer Academic Publishers* (2004), 15–24.

[19] Fredy Juarez, Jorge Ejarque, and Rosa M Badia. 2018. Dynamic energy-aware scheduling for parallel task-based application in cloud computing. *Future Generation Computer Systems* 78 (2018), 257–271.

[20] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. 2021. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 605–620.

[21] Nakku Kim, Jungwook Cho, and Euiseong Seo. 2014. Energy-credit scheduler: an energy-aware virtual machine scheduler for cloud systems. *Future Generation Computer Systems* 32 (2014), 128–137.

[22] Don Lipari. 2012. The slurm scheduler design. *SLURM User Group. http://slurm. schedmd. com/slurm_ug_2012/SUG-2012-Scheduling. pdf* (2012).

[23] Fang Liu, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badger, Dawn Leaf, et al. 2011. NIST cloud computing reference architecture. *NIST special publication* 500, 2011 (2011), 1–28.

[24] Marko Luksa. 2017. *Kubernetes in action*. Simon and Schuster.

[25] Maciej Malawski, Gideon Juve, Ewa Deelman, and Jarek Nabrzyski. 2015. Algorithms for cost-and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds. *Future Generation Computer Systems* 48 (2015), 1–18.

[26] Kavitha Ranganathan and Ian Foster. 2002. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings 11th IEEE International Symposium on High Performance Distributed Computing*. IEEE, 352–358.

[27] Tech. Rep. 2022. *2022 Leadership Vision for Infrastructure Operations*. Gartner.

[28] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, et al. 2016. Scheduler technologies in support of high performance data analysis. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[29] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek (Eds.). ACM, 351–364. https://doi.org/10.1145/2465351.2465386

[30] Siqi Shen, Alexandru Iosup, Assaf Israel, Walfredo Cirne, Danny Raz, and Dick Epema. 2015. An availability-on-demand mechanism for datacenters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 495–504.

[31] Pramod Singh. 2019. Airflow. In *Learn PySpark*. Springer, 67–84.

[32] Karnam Sreenu and M Sreelatha. 2019. W-Scheduler: whale optimization for task scheduling in cloud computing. *Cluster Computing* 22, 1 (2019), 1087–1098.

[33] Douglas Thain, Todd Tannenbaum, and Miron Livny. 2005. Distributed computing in practice: the Condor experience. *Concurr. Pract. Exp.* 17, 2-4 (2005), 323–356. https://doi.org/10.1002/cpe.938

[34] Alexey Tumanov, James Cipar, Gregory R Ganger, and Michael A Kozuch. 2012. alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proceedings of the third ACM Symposium on Cloud Computing*. 1–7.

[35] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

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

[37] Chao Zheng, Ben Tovar, and Douglas Thain. 2017. Deploying High Throughput Scientific Workflows on Container Schedulers with Makeflow and Mesos. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14-17, 2017*. IEEE Computer Society / ACM, 130–139. https://doi.org/10.1109/CCGRID.2017.9

[38] Qiang Zheng, Kan Zheng, Haijun Zhang, and Victor CM Leung. 2016. Delay-optimal virtualized radio resource scheduling in software-defined vehicular networks via stochastic learning. *IEEE Transactions on Vehicular Technology* 65, 10 (2016), 7857–7867.