TPShare: A Time-Space Sharing Scheduling Abstraction for Shared Cloud via Vertical Labels

Yuzhao Wang*
yuzhao_w@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, Hubei, China

Lele Li
ll.li@siat.ac.cn
Shenzhen Institute of Advanced Technology (SIAT), CAS
Shenzhen, Guangdong, China

You Wu
youwu@usc.edu
University of Southern California
Los Angeles, California, USA

Junqing Yu
yjqing@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, Hubei, China

Zhibin Yu
zb.yu@siat.ac.cn
Shenzhen Institute of Advanced Technology (SIAT), CAS
Shenzhen, Guangdong, China

Xuehai Qian
xuehai.qian@usc.edu
University of Southern California
Los Angeles, California, USA

ABSTRACT

Current shared cloud operating systems (cloud OS) such as Mesos and YARN are based on the "de facto" two-horizontal-layer cloud platform architecture, which decouples cloud application frameworks (e.g., Apache Spark) from the underlying resource management infrastructure. Each layer normally has its own task or resource allocation scheduler, based on either time-sharing or space-sharing. As such, the schedulers in different layers are unavoidably *disconnected*, — not aware of each other, which highly likely leads to resource (e.g.,CPU) wastes. Moreover, the tail latency may even be harmed due to the performance interference on shared resources.

This paper takes the first step to *establish the critical missing connection* between the horizontal layers. We propose *TPShare*, a *time-space* sharing scheduling abstraction, using a simple but efficient *vertical label* mechanism to *coordinate* the time- or space-sharing schedulers in different layers. The vertical labels are bidirectional (i.e., up and down) message carriers which convey necessary information across two layers and are kept as small as possible. The schedulers in different layers can thus take actions according to the label messages to reduce resource wastes and improve tail latency. Moreover, the labels can be defined to support different cloud application frameworks.

We implement the label mechanism in Mesos and two popular cloud application frameworks (Apache Spark and Flink) to study the effectiveness of the time-space sharing scheduling abstraction. The label messages of TPShare reduce resource waste and performance interference due to independent time-sharing or space-sharing scheduling of different layers by enabling 1) on-demand fine-grained resource offering, 2) load-aware resource filtering, and 3) resource demand scaling with global view, eventually improving performance and tail latency.

We co-locate 13 Spark batch and 4 Flink latency-sensitive programs on a 8-node cluster managed by TPShare to evaluate the speedup, CPU and memory utilization, and tail latency. The results show that TPShare accelerates the Spark programs significantly with even lower CPU and memory utilization compared to Mesos. With higher resource utilization, the throughput of TPShare is drastically larger than that of Mesos. For the Flink programs, TPShare improves the $99th$ tail latency by 48% on average and up to 120%.

CCS CONCEPTS

• **Computer systems organization → Cloud Computing**; *Performance*; Resource Management.

KEYWORDS

Cloud Computing, Task Scheduling, Resource Management

ACM Reference Format:
Yuzhao Wang, Lele Li, You Wu, Junqing Yu, Zhibin Yu, and Xuehai Qian. 2019. TPShare: A Time-Space Sharing Scheduling Abstraction for Shared Cloud via Vertical Labels. In *ISCA '19: ACM Symposium on Computer Architecture, June 22–26, 2019, Phoenix, AZ.* ACM, New York, NY, USA, 14 pages. https: //doi.org/10.1145/3307650.3326634

1 INTRODUCTION

Traditional cloud application frameworks such as Hadoop 1.0 [47] tightly couple a specific programming model (e.g., MapReduce) and its implementation framework with the underlying resource management infrastructure. This design puts the burden of resource management on cloud application developers, causing endless scalability concerns for task schedulers [49]. To address this drawback, current cloud operating systems (cloud OS) such as Mesos [22] and YARN [49] decouple application frameworks (e.g., Apache Spark) from the underlying resource management infrastructure, — forming two abstraction layers. Such separation of concern significantly relieves the burden of resource management from application developers, and facilitates better scalability of cloud task schedulers. As a result, Mesos and YARN have been widely used in internet companies such as Twitter and Yahoo!, and such two-horizontal-layer architecture has become the "de facto" standard in shared cloud.

*Yuzhao Wang is an intern at SIAT.

In this two-layer architecture, each layer has its own scheduler to schedule tasks or allocate resources within that layer. For example, in a Mesos based platform, the scheduling policy at the cloud OS layer is dominant resource fairing (DRF) [17], and the scheduling policy in the application framework layer depends on the specific framework. For instance, Apache Spark [44, 57] employs delayed scheduling [56] while Apache Flink [1, 14] uses eager scheduling [2, 40]. In general, these scheduling policies belong to two categories: time-sharing and space-sharing. If a scheduler schedules multiple tasks to concurrently run on shared virtual or physical resources but only one task runs on the shared resources at any certain moment (*time-slicing*), we call it time-sharing scheduling. If a scheduler schedules multiple tasks to concurrently run on logically shared physical resources but the resources are physically divided into multiple zones (*space-slicing*) and different tasks run in different zones without any time-sharing, we define it as space-sharing. As such, delayed scheduling belongs to time-sharing while DRF and eager scheduling can be considered as space-sharing.

Despite the advantage of the separation concerns, a natural pitfall of such two-layer architecture is that, the schedulers in different layers are *unaware* of each other and *independently* schedule tasks according to their *own policy*. In essence, the time- or space-sharing schedulers of different layers are *disconnected* without any coordination. This may result in significant resource wastes at space dimension, and severe performance degradation at time dimension.

This paper proposes *TPShare*, a *time-space* sharing scheduling abstraction as the first step to *establish the critical missing connection* between horizontal layers. The key idea is to use *vertical label*, a simple but efficient mechanism, to enable the coordination of the time- or space-sharing schedulers from different horizontal layers. The vertical labels are bidirectional (i.e., up and down) message carriers which convey necessary information across layers and are kept as small as possible. The schedulers in different layers leverage the label messages to take proper actions, if necessary, to reduce space/time wastes, and finally improve performance and tail latency. Note that TPShare is not a new stand-alone task scheduling algorithm or policy. Instead, it is a scheduling *abstraction* with necessary interfaces to connect and coordinate the schedulers in different layers when necessary.

We name the *coordinated* time- or space-sharing scheduling as *time-space sharing scheduling*. Although Hawk [10], Mercury [26], and Eagle [9] all employ a mixed scheduling policy of time-sharing and space-sharing, they do not consider to coordinate time- or space-sharing scheduling among *different horizontal* layers. Instead, they simply employ space-sharing for short jobs and time-sharing for long ones in the *same horizontal* layer.

Distinct from prior works, our time-space sharing scheduling abstraction allows developers to define the *direction* and *content* of the vertical labels. Specifically, the vertical label can be bidirectional, which means that the label message can be passed from a lower layer to a upper layer and vice versa. The content of a label message can be defined to support different application frameworks, e.g., Apache Spark [44, 57] and Flink [14]. As a result, the time-space sharing scheduling abstraction is flexible for different situations. Although current cloud OS and application frameworks do not support the vertical label mechanism, we envision it to be *mandatory* for the next generation of shared cloud.

To demonstrate the effectiveness of time-space sharing scheduling abstraction, we implement the vertical label mechanism on Apache Mesos and two popular could application frameworks: Apache Spark and Apache Flink. We leverage the label mechanism to realize three techniques of TPShare to coordinate the schedulers from the two layers: *on-demand fine-grained resource offering* (**RO**), *load-aware resource filtering* (**RF**), and *resource demand scaling with global view* (**RS**). The three novel techniques enable resource management policies not possible in current cloud OSs. For example, Mesos packages all available resources of a slave server as a coarse-grained resource offer, and provides it to an application framework each time sequentially because Mesos does not know the amount of resources required by the application. In contrast, TPShare can leverage RO to divide the coarse-grained resource offer into multiple fine-grained resource offers according to the demands of corresponding applications, and then provide the offers to the applications *in parallel*. Moreover, TPShare employs RF to filter out servers whose load is already high when providing resource offers to applications, — not supported in Mesos. Finally, the RS technique takes cluster-wide resource status into consideration, and guides the application to adjust their resource demand to favour performance. As a result, RO and RS reduce resource wastes, whereas RF decreases performance interference.

To evaluate the performance of TPShare, we employ an 8-node cluster and co-locate 13 Spark batch and 4 Flink latency-sensitive programs on it. Compared to Mesos, the results show that TPShare can accelerate the Spark programs significantly with even lower CPU and memory utilization. It effectively makes running more programs on TPShare with comparable performance of Mesos possible. In other words, the results show that, when the resource utilization is increased, the throughput of TPShare is dramatically higher than that of Mesos. For Flink programs, TPShare improves the 99th percentile tail latency by 48% on average and up to 120% over Mesos.

To evaluate the adaptivity (i.e., whether its effectiveness depends on certain cluster) of TPShare, we use a different 4-node cluster and construct a different scenario: co-locate a Python program with infinite loops as a severe CPU disturber with 13 Spark programs. The results show that TPShare can still improve the CPU utilization by 3% on average and up to 6% compared to Mesos. Accordingly, TPShare accelerates the Spark programs by 1.37× on average and up to 9.3×. Moreover, the results show that TPShare can improve more CPU utilization and achieve more speedups with larger input datasets. We also co-run a Spark streaming program with the CPU disturber to evaluate the tail latency. The results show that TPShare improves the 99*th* percentile tail latency by 190%.

2 BACKGROUND AND MOTIVATION

In this section, we first take the Mesos-based cloud platform as an example to describe the two-layer cloud architecture. We then describe our motivation.

2.1 Two Layers and Their Schedulers in Mesos

Apache Mesos is a cloud OS built by using the same principles as the Linux kernel but at a different level of abstraction [22]. The goal of Mesos is to share commodity clusters between multiple diverse application frameworks such as Spark and Hadoop to improve cluster

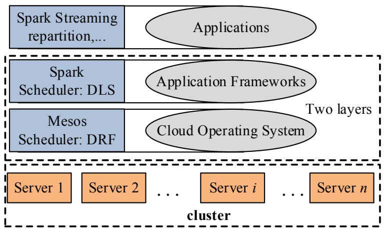


Figure 1: The two-layer architecture of Mesos clusters. DLS-Delayed Scheduling, DRF-Dominant Resource Fairness.

utilization. Figure 1 illustrates the two-layer architecture of Mesos-based cloud platforms. As we can see, besides the applications, the two system layers are cloud OS and application framework. The cloud OS is Mesos but it can be YARN or others. The application frameworks can be Apache Spark or other frameworks such as Apache Flink. In this paper, we consider Spark and Flink.

Generally, each of the two system layers has its own task scheduling policy or algorithm. As Figure 1 shows, at the cloud OS layer, although Mesos pushes task scheduling to application frameworks such as Spark, it still has its own resource allocation policy, — dominant resource fairness (DRF), which can be considered as space-sharing scheduling. At the application framework layer, the task scheduling algorithm is framework specific. For example, Apache Spark employs a time-sharing scheduling algorithm, delay scheduling (DLS) [56]; while Apache Flink uses the eager scheduling algorithm [2, 40] which can be considered as a dynamic space-sharing scheduling algorithm. While these schedulers from different layers may work well for their own corresponding layer, they are *unaware* of each other.

2.2 Disconnected Scheduling between Layers

The problem of disconnected scheduling between horizontal layers is caused by the mutual unawareness between schedulers from different layers, which may lead to significant time and space wastes in the presence of performance interference on shared resources. To understand how the disconnected scheduling affects throughput and tail latency, we design the following experiments.

We employ a 2-node cluster with 32 cores in each node and use Mesos to manage the cluster. On top of Mesos, we co-run a Spark streaming program, *repartition* and 30 Python programs with each containing an infinite loop. *repartition* is latency-critical and is vulnerable to interference. Each Python program with an infinite loop consumes up one CPU core and can be used to severely disturb the co-located program *repartition*. The Python program is called as a CPU disturber. Moreover, we allocate two cores for *repartition* and start 30 disturbers to make them consume 30 cores in total. We run these 31 programs on the clusters several times and observe the throughput and tail latency. In addition, we also co-run *wordcount* which is a batch-processing program and 30 disturbers to validate our observations.

Figure 2 (a) shows the results for *repartition* and Figure 2 (b) for *wordcount*. As we can see, some times the program *repartition* (or *wordcount*) and the 30 disturbers run on all the 32 cores of the same server and all the cores of the other server in the cluster are idle. We use `CPU_32_0` to denote this case where '32' indicates all the 32 cores of a server are used and '0' indicates that no core

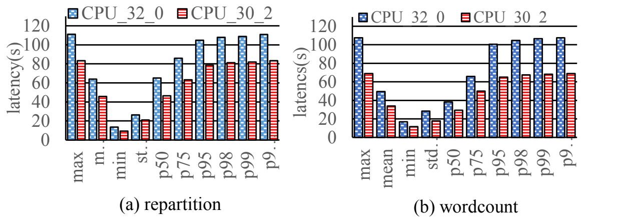


Figure 2: Latency with different scheduling strategies. pXX represents xx percentile tail latency. For example, p99 means 99th percentile tail latency. `CPU_32_0` **indicates the case where all 32 cores of a server are used by the 31 programs and no cores of the other server are used.** `CPU_30_2` **represents the case where 2 cores of a server are used by repartition or wordcount and the 30 cores of the other server are used by the 30 disturbers.**

of the other server is used. In contrast, the program *repartition* (or *wordcount*) may also run on the two cores of one server and the 30 CPU disturbers run on the 30 cores of the other server in other times, denoted by `CPU_30_2`. The tail latencies in the case of `CPU_32_0` for *repartition* and *wordcount* are longer than those in the case of `CPU_30_2` by 34.6% and 47.5% on average, respectively. This indicates scheduling tasks to different server/cores leads to significantly tail latency variance. What's worse, the case `CPU_32_0` or `CPU_30_2` *randomly* takes place when we co-run the programs on Mesos. This further indicates that the tail latency is unpredictable when programs run on shared resources, which is confirmed by previous studies on other cloud platforms [58, 59, 61].

One important reason for this issue is that the scheduler from the application framework layer (the Spark) and the one from the cloud OS layer (the Mesos) are mutual unaware of each other. At the application layer, for the latency-critical Spark program *repartition*, Spark employs DLS to send resource requests to Mesos. The disturbers request resource from Mesos according to their starting order. At the cloud OS layer, Mesos uses DRF to offer resources for the applications in the above layer. However, Mesos does not know the fact that *repartition* is latency-critical and in turn just offers it enough resources but does not consider where the resources are and whether the resources can satisfy the latency-critical requirement. This leads to the behavior that the long or short latency of the program *repartition* takes place randomly, being unpredictable.

Although this motivation example is an artificial case by co-locating a latency-critical or a batch program with an artificial CPU disturber, it represents the principle of co-locating normal programs. We confirm this by running *repartition* on the cluster first, then co-running *wordcount* on the same cluster, and vice versa. We observe that the unpredictable performance issue still holds. Moreover, the latency of *repartition* when it co-runs with *wordcount* on the same server is significantly longer than that it runs on a single server separately. Note that the total CPU consumption and memory consumption of these two programs are significantly less than the total amount of CPU and memory resources of a single server. This indicates that artificial and normal programs both suffer from unpredictable performance on a Mesos cluster.

2.3 The Need for a Vertical Label Mechanism

One may think that we can solve the aforementioned disconnected scheduling issue by using one scheduler to schedule the tasks or

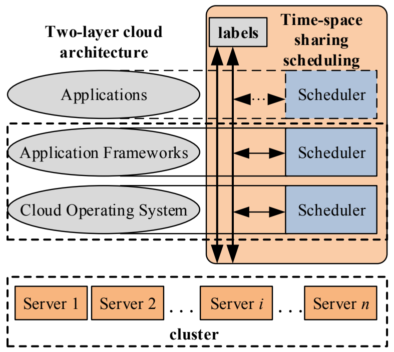


Figure 3: Time-space sharing scheduling abstraction.

allocate resources of the two horizontal layers at the same time, which we call vertical-global scheduling. However, vertical-global scheduling does not work well because it needs to tightly couple the two horizontal layers together, which is exactly what Hadoop 1.0 did before. This approach has at least two shortcomings: imposing resource management burden on application developers, and poor scalability of the scheduler. To overcome these limitations, we propose a vertical label mechanism that can still keep the two horizontal layers separated and provides label messages to convey *necessary but yet small* messages across the layers to coordinate the schedulers from different layers and meet various requirements of different layers. In the next section, we describe our time-space scheduling abstraction with a vertical label mechanism for shared cloud. We argue that the vertical label mechanism should be mandatory for the next generation of shared cloud platforms.

3 TIME-SPACE SHARING SCHEDULING

In this section, we describe our *time-space sharing* scheduling abstraction with the vertical label mechanism.

3.1 Overview

Most of current scheduling policies (algorithms) of cloud systems can be classified into either time-sharing [4, 12, 19, 41, 50] or space-sharing [2, 35] scheduling according to our definition but *not both*. This makes it extremely difficult to improve resource utilization and tail latency simultaneously in shared cloud even with many different stand-alone time-sharing or space-sharing scheduling algorithms, policies, and strategies. In fact, improving performance (e.g., execution time or throughput), resource utilization, and tail latency at the same time is indeed a dilemma based on current scheduling techniques for shared cloud. We therefore argue that the time-sharing or space-sharing schedulers from different layers in a system should work cooperatively. This leads to the *time-space sharing* scheduling.

Figure 3 shows an overview of our time-space scheduling abstraction. The key idea is to leverage a label mechanism to convey necessary information across the two layers of a cloud platform. The time- or space-sharing schedulers of different layers can use such information to take the most suitable actions if necessary to coordinate each other. With the cross-layer collaboration, the performance, resource utilization, and tail latency can be improved at the same time.

As shown in Figure 3, we design the label as a bidirectional channel which can convey information from the lower cloudOS layer to the upper application layer, and vice versa. Note that we do not consider the application scheduler layer in this work because the Spark or Flink programs themselves do not have schedulers. The tasks of them are scheduled by the schedulers in the application frameworks (e.g., Spark or Flink). By doing so, the schedulers from the lower cloudOS layer can adjust scheduling actions according to the requirements from the upper application framework layer while the ones from the upper layer can schedule tasks based on the status (e.g., resources are busy or idle) of the lower layer. Our label mechanism not only allows the bidirectional label to convey messages in one direction at a time but even allows such messages to be exchanged in both directions between the two horizontal layers *at the same time* when scheduling is performed.

In summary, the label mechanism enables our time-space sharing scheduling abstraction to be flexibly adapted to different situations and to orderly coordinate the schedulers from two horizontal layers of a shared cloud. Although some studies such as Hawk [10], Mercury [26], and Eagle [9], apply both time-sharing and space-sharing scheduling in a system, they simply employ time-sharing scheduling for long tasks and use space-sharing for short tasks, and these schedulers work independently.

3.2 Label Message

This section discuss the details of label messages and how they are generated, as well as how to act on them.

3.2.1 What are label messages? We propose three kinds of label messages to satisfy possible requirements for shared cloud: *temporal*, *spatial*, and *temporal-spatial* label messages. The temporal label messages are related to *time* and can be: the deadline of a batch job, the tolerable latency of a web-search-like operation, the response time for interactive applications, the priority of a task, and so on. Note that we use the term "deadline" only for batch jobs/tasks. In addition, "latency" is different from "response time" in our terminology because "latency" is used to represent the time between the moment a user starts a request and the one he/she gets the results, while "response time" refers to the time between the moment a user starts a request and that the system responses.

The spatial label messages are related to *space* (e.g., computing resources). Examples of spatial label messages include the number of CPU cores, the size of memories and disks, memory, disk I/O, and network bandwidth, and so on. In addition, the label message can also be a bundle of resources such as <2 CPU cores, 2GB mem> or the amount of a kind (or bundle) of resources such as CPU.

The temporal-spatial label messages are used to specify the *required performance* which is affected by *both* temporal and spatial factors. This kind of label messages include query per seconds (QPS), the maximum or minimum number of tasks needed to be processed on a given cluster in a specified amount of time, the ratio of successfully processed tasks in a given interval of time on a given cluster, and so on.

3.2.2 How to generate label messages? We propose two ways to generate the label messages: manual setting and optimal setting. Manual setting refers that the label messages are set by users, developers, or system administrators manually. Since some spatial label messages such as the number of CPU cores are already in the configuration file of a system such as Spark, we can add the label messages as configuration parameters and set them manually in the configuration file. Subsequently, the label messages can be generated automatically according to the configuration file after a program is initialized. Of course, the label messages can also be set in the source codes of a system or by command lines. However, manually setting the temporal, temporal-spatial, and spatial label messages has difficulties to some extent. In particular, manually setting spatial label messages can not achieve optimal performance or other targets although it works correctly. Optimal setting is therefore proposed to generate "ideal" label messages for a certain target. For example, a Spark task may need 3 cores for optimal performance but 1 or 2 cores may be set by manual setting. However, optimally setting *all* kinds of label messages is extremely difficult. We therefore discuss how to generate temporal and temporal-spatial label messages by manual setting and how to generate spatial label messages by optimal setting.

The temporal label messages can be provided by end users or administrators of cloud computing platforms because it is not difficult for a user to specify the time he/she wants to complete his/her submitted jobs or tasks. For example, users can specify the deadlines of their batch processing jobs based on their experience and therefore indicate the deadlines as label messages. Another example is that users can specify the priorities of their jobs or specify their jobs as latency-critical ones which have higher priority than batch jobs. As for setting the temporal-spatial label messages, it may be difficult for end users but it is still doable for the administrators of a shared cloud because they are familiar with the performance such as QPS of the cloud platform they manage. We therefore propose to let administrators set the temporal-spatial label messages if necessary. Note that the temporal or temporal-spatial label messages are related to users/administrators’ requirements which may be significantly different for the same type of jobs. It is difficult to satisfy everyone’s requirements because everyone expects the shortest latency, closest deadlines, and others of their jobs. We leave this optimization problem as a future work because this work proposes a scheduling abstraction which expects many different implementations and optimizations.

Optimally setting a spatial label message might be difficult for both end users and administrators because it is not easy to know how much resource is exactly needed by a task. Currently, some spatial label messages such as the number of cores and the size of memory needed by a task are also configuration parameters of a system such as Spark. In practice, developers either set the values of these parameters manually based on experience or just use their default values. However, although this approach works, the values are not "ideal" because they are affected by not only program characteristics but also input data. One set of default values are therefore not "ideal" for different programs and input datasets. Manually setting the spatial label messages for many program and input pairs, if possible, would be very time-consuming. To address this issue, we propose to leverage machine learning techniques to learn the resource demand of a task for a given shared cloud because researchers have employed suitable machine learning algorithms to successfully optimize the configuration parameters of Apache Spark programs and achieved significantly high performance [55]. Therefore, using machine learning techniques to optimally set spatial label messages for a given program and input pair on a given shared cloud is a viable approach.

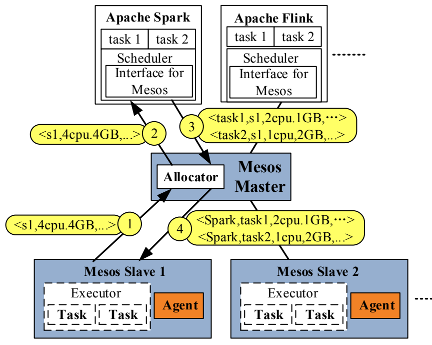


Figure 4: Mesos architecture with Apache Spark.

3.2.3 *How to act on label messages?*

The vertical label mechanism is proposed to coordinate different schedulers from different layers if necessary. It does not change the original scheduling algorithms or policies. Instead, the system takes proper actions according to the label messages on top of the original schedulers. We now discuss how to take actions on different label messages.

In a two-layer shared cloud, temporal or temporal-spatial label messages are usually generated by the upper layer and are passed to the lower layer. The scheduler on the lower layer allocates resources at suitable servers for the upper layer tasks according to the messages but still uses the original scheduling algorithm. For example, if the deadline of a task is very close to the moment when the task is submitted, the lower layer scheduler selects the slave with low-load to execute the task. On the contrary, the lower layer scheduler may select the slave with relatively high-load to run the task. The spatial label messages are related to computing resource requirements of a task. When they are passed from the upper layer to the lower layer, the scheduler of the lower layer knows how much resources the task requests exactly and in turn has the information to prepare fine-grained resources offers for the tasks. More details will be provided in Section 3.3.

Note that these are general guidelines about the label messages. More concrete methods depend on the specific cloudOS and application framework. Next, we elaborate how do we enable the label mechanism in Mesos, Apache Spark, and Apache Flink, forming our TPShare which can be treated as a labeled cloudOS. Not all the general guidelines discussed above are implemented in TPShare.

3.3 TPShare

This section describes the implementation of TPShare and the label mechanism in Mesos, Apache Spark, and Flink. We first provide an introduction for the Mesos architecture before going into the details of TPShare design.

3.3.1 *Mesos Architecture.*

Mesos is a representative cloud OS for shared cloud. As shown in Figure 4, one server in a cluster serves as the master and other servers serve as slaves. A daemon named `Allocator` in Mesos master is in charge of allocating resources for application frameworks such as Spark. In each slave, a daemon named `Agent` is responsible for managing the executors of application frameworks. There can be a large number of slaves (Agents) in a Mesos cluster. Correspondingly, Mesos can support a larger number of application frameworks to form a shared cloud platform.

Mesos employs a concept named *resource offer* to allocate resources for applications. As shown in step ① in Figure 4, **slave**

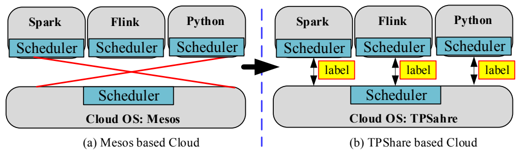


Figure 5: Mesos vs. TPShare based clouds.

1 reports *all* its available resources (e.g., `<4 CPUs, 4GB mem>`) to the master via heartbeats. The master then invokes `Allocator` to encapsulate all the resources of **slave 1** as a resource offer. In step ②, the master provides the Spark framework the resource offer which describes the amount of available resources and the location (e.g., `<s1, 4 CPUs, 4GB mem>`). Next, the Spark scheduler responses to the master with its scheduling decision: using `<1 CPU, 1GB mem>` for **task 1**, and `<1 CPU, 1GB mem>` for **task 2** [1] as shown in step ③. In step ④, the master sends the tasks to **slave 1** there the `Agent` allocates appropriate resources to Spark's executor, which subsequently launches two tasks to execute. Note that **slave 1** still has 2 CPUs and 2 GB of memory unallocated which will return to the `Allocator`. Subsequently, the `Allocator` may offer them to another framework such as Flink. In other words, before the remaining resources are returned to the `Allocator`, they are unavailable to any other frameworks.

One Spark or Flink program running on Mesos corresponds to one framework shown in Figure 4. Each program has a number of tasks and the number depends the code and its input data. Generally, the larger size of input data results in more tasks for a given program. If multiple programs are submitted to Mesos concurrently, Mesos has to provide resource offers to them *sequentially*. Moreover, Mesos packages all the available resources of a slave server into a single resource offer and provides it to a program each time. We observe that the round time between step ② and ③ is typically 200ms. Since Mesos provides resource offers to multiple programs sequentially, submitting more programs to Mesos results in longer time to wait for some programs to get their resource offers.

3.3.2 TPShare Design. As discussed earlier, the current Mesos provides resources to application frameworks using the DRF scheduling policy for resource selection. It does not consider whether performance interference between co-located programs would occur on shared resources. This is why the 2 CPUs and 2 GB of memory on **slave 1** may be offered to Flink after step ④ shown in Figure 4. Moreover, it is difficult for the current Mesos and the application framework to avoid performance interference by scheduling because the scheduler of the application framework layer is unaware of the scheduler of the Mesos layer, as shown in Figure 5 (a). To coordinate the scheduling of the application framework layer and that of Mesos, we implement the label mechanism in between Mesos and application frameworks, as shown in Figure 5 (b). As a result, the labels can pass necessary information across the two schedulers to avoid harmful factors such as performance interference.

Figure 6 shows the detailed design of TPShare based on Apache Mesos, Spark, and Flink. In the `allocator` on the master, TPShare maintains four lists: RDL (resource demand list), ARL (available resource list), AFL (active application framework list), and LPL

[1]The current Spark only allows to set the same amount of resources for each task in a program

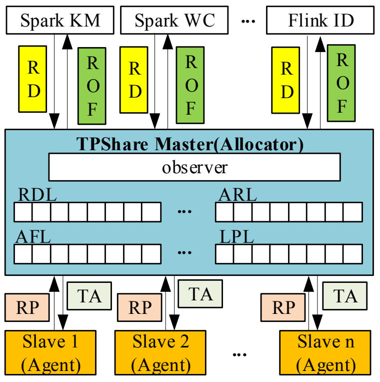


Figure 6: The detail design of TPShare `Allocator`. KM - KMeans, WC - WordCount, ID - Identity, RD - resource demand, ROF - resource offer, RDL - resource demand list, ARL - available resource list, AFL - active application framework list, LPL - latency-sensitive program list, RP - resource report, TA - task assignment.

(latency-sensitive program list). RDL is used to maintain the resource demands of all active application frameworks. Here, we refer each framework by the program running on top of it, such as Spark KMeans (KM) and Flink Identity (ID). Each element of ARL stores the available resource list of a slave. AFL maintains the registered and active programs on TPShare. LPL stores a list of latency-sensitive programs such as Flink programs. Between the lower layer (TPShare) and the upper application framework layer, we define two spatial label messages: RD (resource demand) and ROF (resource offer). RD is a down label message which contains the resource demand, the status indicating whether the resource requirement is satisfied, and the priority (e.g., latency sensitivity or urgent level) of a program. ROF is a up label message making resource offers to the programs of the upper layer. In addition, each slave reports its available resources to the master using RP label messages, and the master assigns tasks to slaves to execute with TA label messages, as shown in Figure 6. Based on these data structures and the vertical label messages, we propose three new techniques for TPShare to help scheduling: *on-demand fine-grained resource offering*(**RO**), *load-aware resource filtering* (**RF**), and *resource demand scaling with global view* (**RS**).

3.3.3 On-demand fine-grained resource offering. Unlike Mesos packages all the available resources of a slave into a single coarse-grained resource offer and provides it to upper level programs sequentially, the RO technique of TPShare divides the coarse-grained offer into n ($n > 1$) fine-grained offers and provides offers to n upper-level programs *at the same time*. This allows more programs on TPShare to get resources in shorter time than on Mesos. The number n generally depends on the RDL which is already known thanks to the down label messages, and the ARL which is known by the `allocator` of TPShare. Larger n may result in higher scheduling efficiency. However, too large n may exhaust the available resource of a server and some offers have no available resources corresponding to them. That is, m ($m < n$) of the n fine-grained offers exhaust a server's resources and in turn the $n - m$ offers have no corresponding resources. The programs getting these offers have to wait. In our implementation, we choose 3 for n considering the trade-off

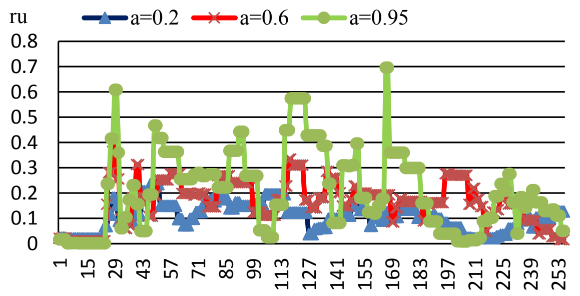


Figure 7: The variance of *ru* along with time with different values of α.

described above. In Mesos, when making multiple resource offers to the same number of user programs, dominant resource fairing (DRF) scheduling is used but it schedules the resource offers *sequentially* because one slave server can only make one resource offer to one user program at a time. In TPShare, the RO technique can help DRF schedule resource offers to multiple user programs concurrently, improving the performance.

3.3.4 *Load-aware resource filtering.*

The RF technique calculates the resource utilization rate of a slave server as follows:

$$ru = \alpha \times CPU_u + (1 - \alpha) \times mem_u \qquad (1)$$

where CPU_u is the CPU utilization rate, and mem_u is the memory utilization rate of a server. Since the sum of CPU_u and mem_u may be larger than 1, we employ a coefficient α ($0 < \alpha < 1$) to make the value of ru be between 0 and 1. CPU_u and mem_u of a server can be easily obtained by profiling. The value of α therefore determines how we represent the resource utilization rate (ru) of a server. Although we do not change the scheduling algorithm DRF at the CloudOS layer, we leverage ru to help select a proper slave to run the tasks (Note that DRF concerns the amount of dominant resources of a user but do not concern where the resources locate). If ru does not change significantly, it would not help select slaves to run the tasks for better performance. We therefore expect that the ru can change significantly in different times.

To determine the value of α, we conduct the following experiments. We co-run two programs (*wordcount* and *logistic regression*) with 5 instances each on a server to observe how ru varies along with time. As Figure 7 shows, the variance of ru increases along with the value of α. When it equals 0.2, the variance of ru is less than 10%, which is not helpful for scheduling. When *alpha* increases to 0.95, the variance of ru increases to 46% accordingly, which is helpful for selecting suitable slave servers to run the tasks. We therefore choose 0.95 as the value of α in this study.

We then use ru to represent the load of a server. The RF technique sets a threshold of ru when TPShare selects slave servers to make resource offers to the upper layer programs. When the ru of a server is higher than the threshold, this server will not be selected to provide resource offers. Otherwise, it will be selected. In this paper, we tentatively set the threshold to 0.6 for evaluation because the resource utilization of current shared cloud is typically significantly lower than 0.6. As a result, the RF technique of TPShare always selects servers with low-load to make resource offers to upper layer programs. In addition, the RF technique firstly selects the servers with the lowest load to make resource offers to the latency-sensitive

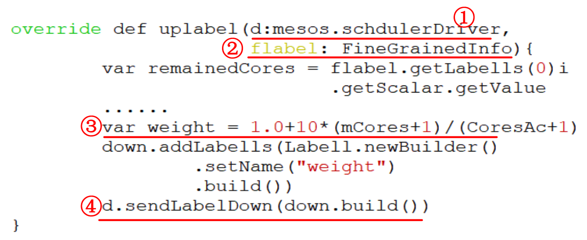

```
override def uplabel(d:mesos.schdulerDriver, ①
                   ② flabel: FineGrainedInfo){
          var remainedCores = flabel.getLabells(0)i
                              .getScalar.getValue
          ......
        ③var weight = 1.0+10*(mCores+1)/(CoresAc+1)
          down.addLabells(Labell.newBuilder()
                  .setName("weight")
                  .build())
        ④d.sendLabelDown(down.build())
}
```

Figure 8: Label message interface in Spark.

programs, and does not attempt to make two resource offers for two latency-sensitive programs from the same server.

Since the Spark scheduler on the upper layer does not know the loads of servers in the lower Mesos layer, it may accept the resource offer with enough resource from a high-load rather than low-load server according to the Spark scheduling policy — delayed scheduling. In contrast, the RF technique of TPShare makes it *impossible* to make resource offers from a high-load server to Spark programs even they still use the delayed scheduling. The same can be applied to Flink programs. In summary, RF can help the schedulers from upper layers reduce performance interference.

3.3.5 *Resource demand scaling with global view.*

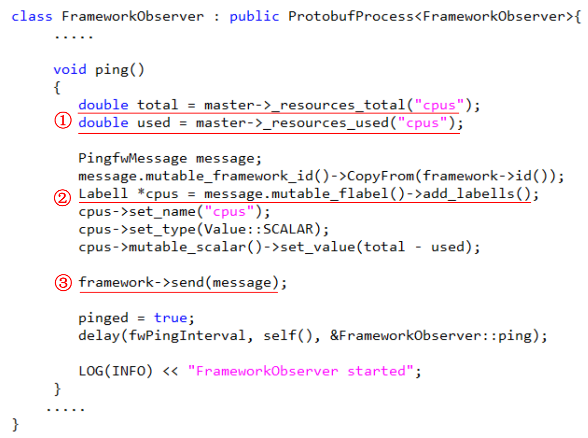
We now describe the RS technique of TPShare. The key idea is to extend additional resources for tasks besides their initially allocated resources when the resources of a cluster are sufficient, improving performance and resource utilization. To this end, we implement one module called *observer* in the `Allocator` of TPShare and one module dubbed *actuation* in Spark. As shown in Figure 6, the observer checks the total amount of used resources (TAUR) of a cluster based on per node report (passed by the RP label). If the TAUR is less than a threshold, the `Allocator` informs the *actuation* of Spark to increase resource demand. Otherwise, the Spark tasks will use the initially allocated resources.

There are two factors needing to be determined according to the workload characterization and runtime environment: the threshold and the amount of extended resources. In this study, we believe that a cluster has sufficient resources if the amount of used resources is less than 60% of the cluster’s total amount of resources. We therefore set the threshold to 0.6. As for determining the amount of extended resources, we develop a model. It takes the global resource status, program name, and dataset size as inputs, and outputs the total cores and memory size per executor required by the program.

3.3.6 *Label Message Interfaces.*

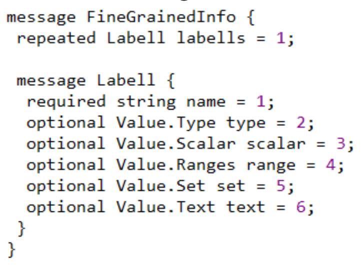
Figure 8 shows the label message interface defined in the Spark scheduler. It sends label messages down to Mesos master and slaves. As statement ① in Figure 8 shows, we leverage Mesos scheduler driver in Spark to implement the label message passing from Spark to Mesos. Statement ② illustrates a class defined by us to manage the label messages. We define this class by using Protobuffer [23], as shown in Figure 10. When sending down label messages, we first use statement ③ to add a new label message such as the weight of a task, and subsequently employ statement ④ to send down the label message to Mesos.

Figure 9 illustrates how we send label message from Mesos to Spark. First, we collect the total amount of resource and total amount of used resources in the cluster, as the statements ① show. Subsequently, we encapsulate this information as up label



```
class FrameworkObserver : public ProtobufProcess<FrameworkObserver>{
    .....

    void ping()
    {
       double total = master->_resources_total("cpus");
    ①  double used = master->_resources_used("cpus");

       PingfwMessage message;
       message.mutable_framework_id()->CopyFrom(framework->id());
    ②  Labell *cpus = message.mutable_flabel()->add_labells();
       cpus->set_name("cpus");
       cpus->set_type(Value::SCALAR);
       cpus->mutable_scalar()->set_value(total - used);

    ③  framework->send(message);

       pinged = true;
       delay(fwPingInterval, self(), &FrameworkObserver::ping);

       LOG(INFO) << "FrameworkObserver started";
    }
   .....
}
```

Figure 9: Label message interface in Mesos.



```
message FineGrainedInfo {
 repeated Labell labells = 1;

 message Labell {
  required string name = 1;
  optional Value.Type type = 2;
  optional Value.Scalar scalar = 3;
  optional Value.Ranges range = 4;
  optional Value.Set set = 5;
  optional Value.Text text = 6;
 }
}
```

Figure 10: Code example for label messages.

messages, as shown in statement ②. Finally, we leverage the `send()` interface of frameworks to send up the label message to Spark, as the statement ③ shows.

Note that we use the same approach to implement the vertical label mechanism for Apache Flink on Mesos. Therefore, our TPShare supports the label mechanism for two application frameworks: Spark and Flink.

4 EXPERIMENTAL SETUP

In this section, we describe our experimental methodology.

4.1 Experimental Platform

Our experimental platform consists of 8 DELL servers, one serves as the master node and the others serve as slave nodes. Each server is equipped with 2 Intel(R) Xeon(R) CPU E5-2630 2.40GHz 16-core processor and 64GB memory. The OS of each node is Ubuntu Server 16.04. We use Mesos 1.3.2, Spark 2.3.0, and Flink 1.5 as our experimental frameworks.

4.2 Representative Applications

We select representative programs from the Spark version of HiBench which is widely used to evaluate the Spark framework. HiBench has different kinds of real-world programs, including machine learning and web search. We choose 14 programs to evaluate the cross-layer scheduling coordination mechanism, as shown in Table 1. These programs represent a sufficiently broad set of typical Spark program behaviors. *WordCount* is CPU-intensive and map-intensive, while *TeraSort* is both CPU and memory-intensive with higher reduce intensity. As for the machine learning benchmarks, *KMeans* has good instruction locality but poor data locality while *Bayes* is the opposite. Although both performing selective shuffling, the iteration selectivity of *PageRank* is much higher compared to *KMeans*. *Join*, *Aggregation* and *Scan* take the data generated from web trace as input and perform Hive queries. As for latency-sensitive benchmark, *Repartition* takes data from Kafka as the source and changes the level of parallelism by creating more or fewer partition tests. It is used to test the efficiency of data shuffle in the streaming module. To evaluate our design, we employ three different sizes of inputs (Table 1) for the corresponding programs.

Table 1: Experimented benchmarks in this study. Intval - Interval, rods - records, recLen - record length.

Application	Abbr.	Input Data size
Terasort	TRS	3.2MB, 320MB, 3.2GB
WordCount	WCT	3.2KB, 320MB, 3.2GB
SQL Aggregartion	AGG	8KB, 512KB, 640KB
SQL Join	JON	8KB, 512KB, 640KB
SQL Scan	SCN	8KB, 512KB, 640KB
PageRank	PRK	12KB, 1.8MB, 240MB
Bayes	BYS	88MB, 105MB, 360MB
Kmeans	KMS	1.5MB, 550MB, 3.8GB
Prin..Compo..Analysis	PCA	80KB, 8MB, 30MB
GrdiantBoostTree	GBT	9.4KB, 400KB, 15MB
RandomForest	RFT	8KB 400KB 8MB
LinearRegression	LNR	3.8GB, 15GB, 45GB
SupportVectorMachine	SVM	8MB, 750MB, 18.5GB
Streaming Repartition	REP	Intval:50ms, recLen: 200

Moreover, we employ all the four latency-sensitive Flink programs from HiBench to evaluate TPSare: *fixwindow*,*identity*, *repartition*, and *wordcount*. In addition, we employ the CPU disturber to co-locate with 13 Spark programs on a 4-node cluster (we remove 4 servers from the 8-node cluster) to evaluate whether TPShare can improve the performance and tail latency of the programs, and resource utilization of the cluster in this extreme case.

5 RESULTS AND ANALYSIS

In this section, we present the experimental results comprehensively and analyze the ingishts.

5.1 Evaluation of RO, RF, and RS

We first individually evaluate the effects of the three proposed techniques, RO, RF, and RS. Figure 11 shows the effects caused by the RO technique. As can be seen, RO can significantly reduce the scheduling waiting time of a Mesos cluster. When there is only one slave node in the cluster, the scheduling waiting time can be reduced by 1.55×. When there are more slave nodes in the cluster, the scheduling waiting time reduction decreases but still achieves 40% at least. This indicates that providing fine-grained resource offers to multiple programs concurrently can significantly improve the scheduling efficiency of the upper-level application frameworks in shared cloud. Note that when there is a large number of slave nodes in the cluster, RO can still reduce the scheduling waiting time significantly when the load of the cluster is high.

Figure 12 shows the speedup produced by RF for the Spark programs. It shows the individual RF technique can accelerate most experimented programs and the speedup achieves 15.8% on average and up to 44%. We see that the speedup is not significant. This is because the RF technique is designed to select low-load servers for latency-critical applications while Spark programs in this evaluation are batch programs. In addition, we do not co-run programs in this experiment, which does not generate interference. However, the combination of RF and RS techniques significantly accelerate the

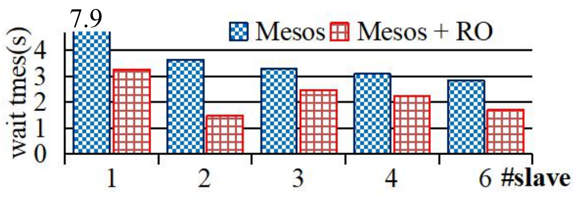


Figure 11: The scheduling waiting time reduction produced by the RO technique.

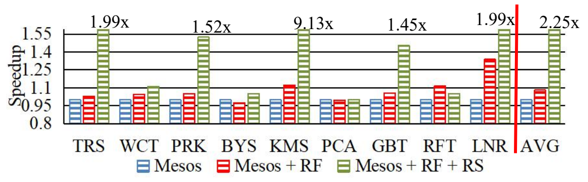


Figure 12: The speedup caused by RF and RS.

experimented Spark programs, as shown in Figure 12. The speedup of the RF and RS achieves 2.3× on average and up to 9.1×. This implies that the RO technique can significantly accelerate the Spark batch programs.

In summary, the above results show that the three proposed technique can speed up the performance of programs or improve the scheduling efficiency individually. We next evaluate the effects of TPShare which combines all the three techniques together.

5.2 Spark Programs

We co-run 13 Spark program instances (9 programs) on the 8-node cluster twice. In the first time, the cluster is managed by TPShare and the other time it is managed by Mesos. Figure 13 shows the speedups of the programs running on TPShare over running on Mesos. We see that, each program runs significantly faster on TPShare than on Mesos. The average speedup is 1.5× and the maximum speedup is 1.9×. Figure 14 shows the CPU and memory utilization of the 7 slave servers. We find that some servers have higher CPU utilization when managed by TPShare over by Mesos whereas others have higher memory utilization. On average, the CPU and memory utilization of the cluster managed by TPShare is slightly lower than the one managed by Mesos. This is a surprising result because the intuition is that faster programs usually consume more resources. We found that this result is caused by the space (resource) and time wastes in Mesos due to the unawareness of the schedulers from the two layers. The RO and RF techniques of TPShare can help coordinate the two schedulers and in turn reduce the space and time wastes. Therefore, TPShare achieves higher performance by consuming even less resources than Mesos.

These results imply that we can submit more programs to run on the cluster managed by TPShare. We did so by co-running 29 program instances concurrently on the cluster. Figure 15 and Figure 16 show the results. We see that, the programs managed by TPShare still run significantly faster than by Mesos, and the average speedup is 17%. The average CPU utilization of TPShare is now higher than that of Mesos, while the average memory utilization shows the opposite, due to the finer grain resource allocation in TPShare. Comparing Figure 14 and Figure 16, we see that, the

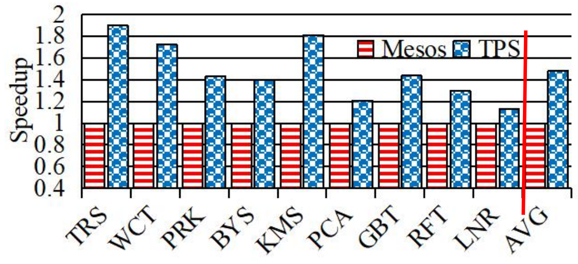


Figure 13: Speedup of TPShare over Mesos when co-running 13 Spark program instances.

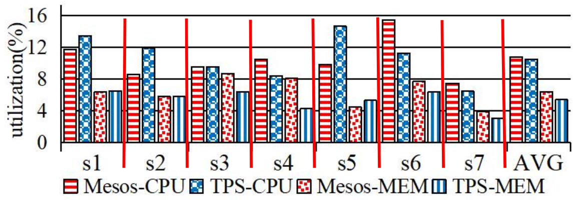


Figure 14: Resource utilization comparison between TPShare and Mesos when co-running 13 Spark program instances. s1 denotes the first server, and so on.

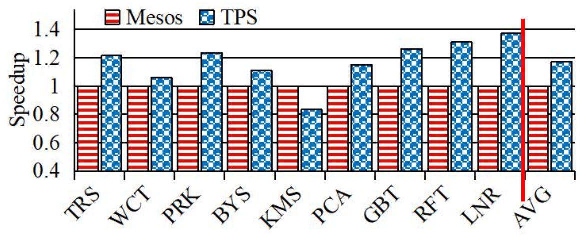


Figure 15: Speedup of TPShare over Mesos when co-running 29 Spark program instances.

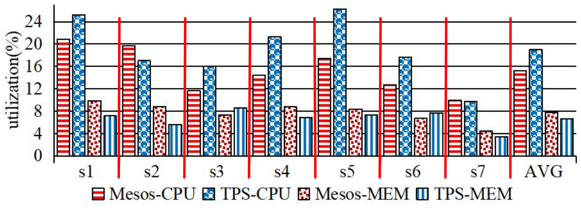


Figure 16: Resource utilization comparison between TPShare and Mesos when co-running 29 Spark program instances. s1 represents the first server, and so on.

resource utilization with 29 program instances co-running is significantly higher than that with 13 Spark program instances. It validates the known fact that the resource utilization can be improved by co-running more programs, — and it is true for both TPShare and Mesos. We do see that TPShare increases the resource utilization more than Mesos does. This indicates that TPShare can better leverage the resources to improve performance.

5.3 Flink and Spark Programs

To evaluate how does TPSare reduce the performance interference, we co-run each of the four Flink programs with the 13 Spark program instances on the cluster twice, once managed by TPSare and the other managed Mesos. Figure 17 shows that co-running Spark

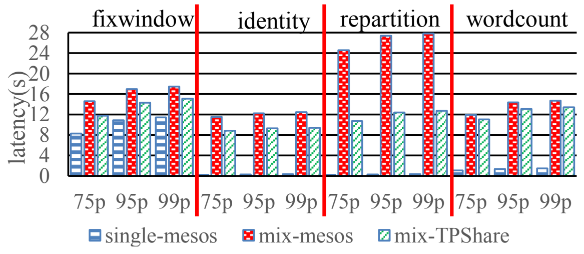


Figure 17: Tail latency comparison between TPShare and Mesos for the 4 Flink programs.

programs with Flink programs significantly increase the tail latency of the Flink programs. Fortunately, TPShare can significantly reduce the tail latency of the Flink programs with the presence of interference. Specifically, TPShare improves the tail latency by 48% on average and up to 120% over Mesos. This is because that, the RF technique of TPShare can always choose the servers with lowest load to run the latency-sensitive Flink programs, thereby reduces the performance interference.

5.4 CPU Disturber and Spark Programs

In this section, we evaluate TPShare in an extreme condition.

5.4.1 Tail Latency. To evaluate whether TPShare can improve tail latency in an extreme case, we co-run a latency-critical (Spark streaming) program - *repartition* with three different inputs: *tiny*, *small*, and *large* which are 50K, 100K, 160K records per second, respectively. The CPU disturber is the one used in Section 2.2. To quantify the efficacy of TPShare, we define relative improvements of tail latency as follows.

$$TL_{ri} = |(TL_m - TL_{tp})|/TL_{tp} \times 100\% \tag{2}$$

with TL_{ri} the relative tail latency improvement, TL_m the tail latency on Mesos, and TL_{tp} the tail latency on TPShare. Figure 18 compares the tail latencies on Mesos and on TPShare. We observe a number of interesting observations.

First, our TPShare significantly improves the tail latency of program *repartition* with the presence of interference caused by the disturber. For example, as Figure 18 (a) shows, the TL_{ri}s for 75th, 80th, 95th, and 99th percentiles are 135%, 255%, 215%, and 190%, respectively. The average TL_{ri} is 200%. Moreover, we use other three latency-critical benchmarks (*SQL Aggregation*, *SQL join*, *SQL scan*) to evaluate TPShare. We observed that TPShare can similarly improve tail latency with different degrees but we cannot include all results due to space limitation. In summary, the experimental results indicate that our label mechanism can indeed significantly reduce the performance interference caused by co-located programs on shared resources and in turn improve tail latency drastically.

Second, TPShare achieves more tail latency improvements for programs with larger inputs over Mesos, see Figure 18 (a), (b), and (c). More importantly, TPShare achieves significantly more improvement on 99th percentile tail latency (which is the one people are most interested in) over Mesos when programs run with larger sizes of input data than with smaller sizes of data, as shown in Figure 18. This is because smaller inputs of a Spark program result in smaller number of tasks. On a given cluster, a smaller number of tasks are able to obtain more resources to run, and in turn achieve

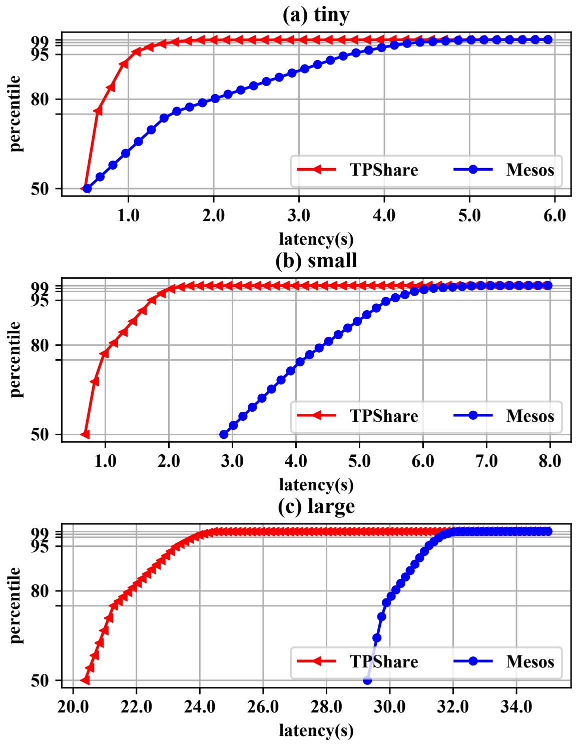


Figure 18: Tail-latency comparison of benchmark *repartition* between Mesos and TPShare.

shorter latency. This result indicates that TPShare has more potential to benefit big data programs because they typically process large inputs, and the input is expected to keep increasing in future.

5.4.2 CPU Utilization. We now evaluate the speedups and CPU utilization of the programs considering four cases: (1) a program runs on Mesos; (2) a program runs on TPShare; (3) a program and the disturber co-run on Mesos; and (4) a program and the disturber co-run on TPShare. We use large inputs for all the experimented programs — 10 batch workloads and 3 latency-critical ones (SQL-related workloads), as shown in Table 1.

Figure 19 shows the CPU utilization for all the cases with *large* input datasets. We make several interesting observations. First, for each program, the average CPU utilization for the two cases with TPShare is higher than that for the two cases with Mesos. This indicates that the our label mechanism can indeed improve the overall CPU utilization of a shared cloud.

Second, we observe that TPShare improves the CPU utilization over Mesos for most experimented programs when there are no disturbs, i.e., a single program runs on the cluster at a time. This is somewhat counter-intuitive because TPShare is designed to improve the CPU utilization when there are significant interference when more than one programs co-run on the cluster. We explain it as follows. There are two aspects affecting CPU utilization. First, the label mechanism incurs overhead for creating, passing, and leveraging the label messages to help scheduling, which may reduce the number tasks running on the cluster. This will in turn reduce the CPU utilization although the label mechanism consumes a little amount of CPU. Second, due to the label mechanism to pass

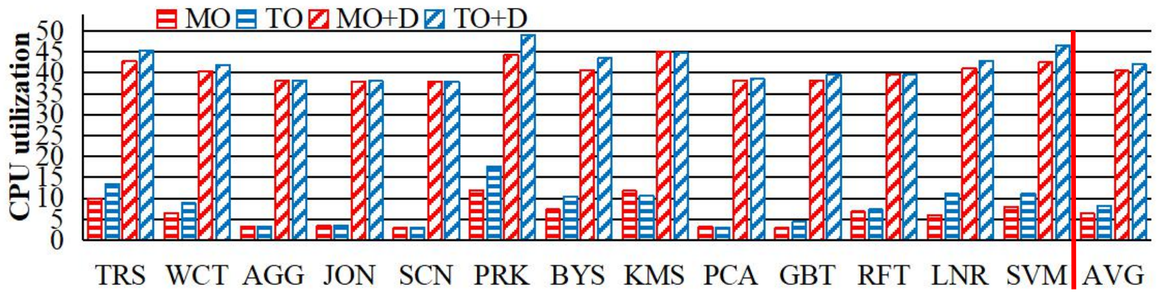


Figure 19: CPU utilization of the experimented benchmarks with large inputs. MO - a program runs on Mesos, TO - a program runs on TPShare, MO+D - a program and the disturber co-run on Mesos, TO+D - a program and the disturber co-run on TPShare.

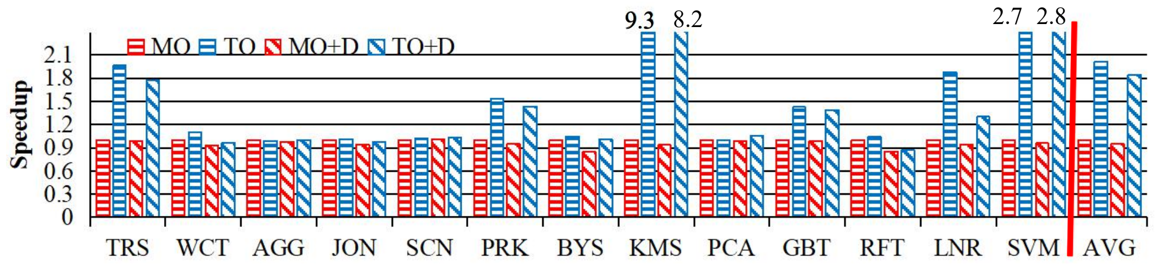


Figure 20: Speedups of the experimented benchmarks with large inputs running on TPShare over Mesos. MO - a program runs on Mesos, TO - a program runs on TPShare, MO+D - a program and the disturber co-run on Mesos, TO+D - a program and the disturber co-run on TPShare.

information on resources availability from lower layers (e.g., Mesos layers) to upper layers (e.g., Spark framework layer), the Spark scheduler of TPShare is aware of the total amount of available resources in a given cluster. In contrast, the scheduler of Spark running on Mesos does not have this information. It will lead to more CPU utilization increase in TPShare.

5.4.3 Speedup. We now evaluate the speedup made by TPShare over Mesos for all the experimented programs in two cases: without/with disturber. We define speedup as follows:

$$speedup = ET_m/ET_{tp} \tag{3}$$

with ET_m the execution time of a program running on Mesos and ET_{tp} the execution time on TPShare.

# Without Disturber. As shown in Figure20, TPShare accelerates almost all the programs with three input datasets for each over Mesos . For the three SQL-related programs (AGG, SCN, and JON), the speedups are negligible. This is because these programs are latency-critical ones and the execution time (latency) is very short. In contrast, the speedups for the batch programs are significant. Especially, TPShare speeds up KMS by 9.3×. This is because KMS has good instruction locality but poor data locality with massive data parallelism. Thanks to the label mechanism, the Spark scheduler of TPShare can issue more tasks to execute concurrently. In this case, the average speedups achieved by TPShare over Mesos for all the programs 2×. This indicates that TPShare accelerates programs more with larger datasets. Thus, the improved CPU utilization improves program performance.

# With Disturber. It is more difficult to accelerate programs co-running with disturbers compared to without disturbers. However, Figure 20 show that TPShare still significantly accelerates most programs. In this case, the speedup is 1.77× on average and up to 8.2×. This indicates that our label mechanism can indeed help to reduce interference and in turn improve performance of programs running on shared resources.

6 RELATED WORK

The task scheduling of shared cloud can be classified into three categories: central scheduling, distributed scheduling, and hybrid scheduling. We summarize them in this section.

6.1 Central Scheduling

Many central scheduling schemes have been proposed [8, 11, 12, 16, 20, 21, 25, 28–31, 45, 46, 50, 53]. Borg is a central architecture with applications gathered in front of a scheduler waiting for time-sharing the resource pool [50]. Furthermore, it makes several duplications of the scheduler to increase scalability and reduce waiting time. The scheduling modules in system Paragon[11] and Quasar[12] both take fine-grained information such as quantified interference and low-level heterogeneity into consideration to select resources. Firmament[20] formalizes task scheduling as a maximum flow minimum cost (MFMC) problem regarding the resource states and computes the scheduling plan periodically. [8] adopts a mixed-integer linear programming method to enable resource reservation planing.

Apart form scheduling logic optimization, Studies [5, 21, 31, 45, 53, 59] try to capture the time, source, and degree of resource contention, and leverage this information to help the central scheduler to distribute tasks efficiently. For example, [36, 45, 46, 51, 54] propose to reduce the negative effects of sharing either memory subsystem resources or core resources via arbitrating co-located applications. [24, 30, 38, 48] quantify the application's resource sensitivity in cloud, which is used to guide scheduling decision.

A variant of centralized scheduling is the two-level scheduling such as the algorithm used in both Mesos[22] and Yarn[49]. Recently, [41] proposes shared-state scheduling between multiple schedulers using lock-free optimistic concurrency control. But the conflicts between schedulers is a bottleneck. In summary, our time-space sharing scheduling is different from these time-sharing algorithms because our approach focuses on the coordination between schedulers from different layers.

6.2 Distributed Scheduling

In contrast to central scheduling, distributed scheduling can achieve shorter scheduling delay [4, 13, 35]. [35] proposes a decentralized and randomized sampling approach to avoid the limitations of a centralized scheduling. Although it achieves a good performance, optimal decision is rare due to the sampling method's uncertainty and lack of global information. Apollo [4] utilizes global cluster information via a horizontally coordinated mechanism to schedule millions of tasks efficiently. In contrast, we propose a vertically coordinated scheduling solution.

Another line of distributed scheduling studies focus on particular workloads, such as the data analytic applications [34] [18] [37]and AI applications [32]. [34] proposes a new system building guideline that each resource should has its own scheduler to schedule tasks which only use a single resource (Monotasks). [37] offloads task scheduling to each worker to increase throughput. Similarly, [32] builds a two-level hierarchical scheduler consisting of a global scheduler and a per-node local scheduler. By considering the load per node, the global scheduler coordinates with the local scheduler. However, similar to the centralized scheduling, these distributed scheduling policies still do not consider the coordination of schedulers from different horizontal layers while our TPShare does.

Akin to [35], [42] designed a distributed OS model for a disaggregated data center[15], which needs a set of monitors to manage each hardware component. Different monitors are loosely-coupled through a customized RDMA-based network. The distributed OS architecture employs a similar spirt,but targets at clusters with commodity servers. Moreover, inspired by this spirt, multi-kernel OSes [3, 7, 33, 52] run a kernel at each processor, and each communicates through message passing over local buses. Despite these efforts, as argued in [27, 39], improving the per-node efficiency in cloud via new scheduling abstraction is still urgently needed. TPShare works toward this end.

6.3 Hybrid Scheduling

From the above description, we learn that centralized scheduling can achieve high quality of scheduling decision but with long scheduling delay while distributed scheduling does the opposite. In order to achieve high quality of scheduling decision and short scheduling delay at the same time, hybrid scheduling is proposed. For example, Hawk[10] adopts a centralized/distributed scheduling structure, with the former enforcing scheduling quality and the latter reducing scheduling delay. However, a part of the cluster resources are dedicated to centralized schedulers. In [9], an enhanced hybrid scheduling policy is proposed by leveraging an application-aware task placement algorithm to choose the appropriate scheduler between the centralized and distributed ones. Moreover, [26] adopts a down-up scheduling policy similar to [32], and the tasks are routed to either centralized or distributed schedulers based on the resource type specified by users. These scheduling polices also do not consider the coordination of schedulers from different layers, which is different from TPShare.

The closest works to ours are [43], [60], and [6]. In [43], the authors try to address the double CPU scheduling problem between the hyper-visor and guest operating system via vCPU ballooning. Similarly, [60] points out that the unawareness between the VM host- and guest-level schedulers hinders the timeliness guarantees. This paper attempts to address it via key scheduling information and scheduling decision coordination. Both [43] and [60] address the scheduling issues in virtualization systems while [6] tries to narrow the semantic gap between OS and database engines. In contrast, we address the scheduling disorder issue between cloud OS and application frameworks in shared cloud.

7 CONCLUSION

We propose TPShare, a time-space sharing scheduling abstraction, using a simple but efficient vertical label mechanism to coordinate the time-sharing or space-sharing schedulers in different layers. The vertical labels are bidirectional message carriers which convey necessary information across two layers and are kept as small as possible. The schedulers in different layers can thus take actions according to the label messages to reduce resource wastes and improve tail latency. Moreover, the labels can be defined to support different cloud application frameworks. We implement the label mechanism in Mesos and two popular cloud application frameworks (Apache Spark and Flink) to study the effectiveness of the time-space sharing scheduling abstraction. We co-locate 13 Spark batch and 4 Flink latency-sensitive programs on a 8-node cluster managed by TPShare to evaluate the speedup, CPU and memory utilization, and tail latency. Compared to Mesos, the results show that TPShare accelerates the Spark programs significantly with even lower CPU and memory utilization. With higher resource utilization, the throughput of TPShare is drastically larger than that of Mesos. For the Flink programs, TPShare improves the 99*th* tail latency by 48% on average and up to 120%.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments. This work is supported by the national key research and development program under Grant No. 2016YFB1000204, outstanding technical talent program of CAS, and NSFC under (Grant No. 61672511, 61702495, 61572211, and 61802384), Post-doctoral Science Foundation of China under No. 2017M622830, NSF of Guangdong province under grant No. 2017A030310350. Zhibin Yu and Junqing Yu are the corresponding authors. Contact: zb.yu@siat.ac.cn.

REFERENCES

[1] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heises, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinl ander, Mathias J. Sax, Sebastian Schelter, Mareike H oger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal* 23, 6 (Dec. 2014). https://doi.org/10.1007/s00778-014-0357-y

[2] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. 1996. Charlotte: Metacomputing on the Web. *Future Gener. Comput. Syst.* 15, 5-6 (Oct. 1996). https://doi.org/10.1016/S0167-739X(99)00009-6

[3] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Sch upbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM Press, New York, NY, 29–44. https://doi.org/10.1145/1629575.1629579

[4] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for Cloud-scale Computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, 285–300. http://dl.acm.org/citation.cfm?id=2685048.2685071

[5] Xiangping Bu, Jia Rao, and Cheng zhong Xu. 2013. Interference and Locality-aware Task Scheduling for MapReduce Applications in Virtual Clusters. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing (HPDC '13)*. ACM Press, New York, NY, 227–238. https://doi.org/10.1145/2493123.2462904

[6] JANA GI CEVA. 2016. *Database/Operating System Co-Design*. Ph.D. Dissertation. Swiss Federal Institute of Technology Zurich, Zurich, Swiss.

[7] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. 1995. Hive: Fault Containment for Shared-memory Multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM Press, New York, NY, 12–25. https://doi.org/10.1145/224056.224059

[8] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. 2014. Reservation-based Scheduling: If YouŔe Late DonŤ Blame Us!. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '14)*. ACM Press, New York, NY, 1–14. https://doi.org/10.1145/2670979.2670981

[9] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. 2016. Job-aware Scheduling in Eagle: Divide and Stick to Your Probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. ACM Press, New York, NY, 497–509. https://doi.org/10.1145/2987550.2987563

[10] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. 2015. Hawk: Hybrid Datacenter Scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 499–510. https://doi.org/10.1145/2493123.2462904

[11] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM Press, New York, NY, 77–88. https://doi.org/10.1145/2451116.2451125

[12] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM Press, New York, NY, 127–144. https://doi.org/10.1145/2541940.2541941

[13] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. 2015. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM Press, New York, NY, 97–110. https://doi.org/10.1145/2806777.2806779

[14] flink team. 2018. Apache Flink - Stateful Computations over Data Streams. Retrieved March 30, 2019 from https://flink.apache.org/

[15] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, Berkeley, CA, 249–264. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gao

[16] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. 2018. MEDEA: Scheduling of Long Running Applications in Shared Production Clusters. In *Proceedings of the 13rd European Conference on Computer Systems(EuroSys) (EuroSys '18)*. ACM Press, New York, NY, 131–143. https://doi.org/10.1145/3190508.3190549

[17] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI '11)*. USENIX Association, Berkeley, CA, 323–336. http://dl.acm.org/citation.cfm?id=1972457.1972490

[18] Andrey Goder, Alexey Spiridonov, and Yin Wang. 2015. Bistro: Scheduling Data-parallel Jobs Against Live Production Systems. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, Berkeley, CA, 459–471. https://doi.org/10.1145/2987550.2987576

[19] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, 99–115. http://dl.acm.org/citation.cfm?id=3026877.3026886

[20] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, Berkeley, CA, 99–115. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gog

[21] Jaeung Han, Seungheun Jeon, Young ri Choi, and Jaehyuk Huh. 2016. Interference Management for Distributed Parallel Applications in Consolidated Clusters. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM Press, New York, NY, 443–456. https://doi.org/10.1145/2872362.2872388

[22] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Berkeley, CA, 295–308. http://dl.acm.org/citation.cfm?id=1972457.1972488

[23] Google Inc. 2012. Protocol Buffers - Google's data interchange format. Retrieved Match 30, 2019 from https://github.com/protocolbuffers/protobuf

[24] Pawel Janus and Krzysztof Rzadca. 2017. SLO-aware Colocation of Data Center Tasks Based on Instantaneous Processor Requirements. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. ACM Press, New York, NY, 256–268. https://doi.org/10.1145/3127479.3132244

[25] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. 2013. EyeQ: Practical Network Performance Isolation at the Edge. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI '13)*. ACM Press, New York, NY, 297–311. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/jeyakumar

[26] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. 2015. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'15)*. USENIX Association, Berkeley, CA, 485–497. https://www.usenix.org/conference/atc15/technical-session/presentation/karanasos

[27] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling High Server Utilization and Sub-millisecond Quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM Press, New York, NY, Article 4, 14 pages.

[28] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM Press, New York, NY, Article 1, 16 pages.

[29] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. 2015. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI) (NSDI '15)*. USENIX Association, Berkeley, CA, USA, 589–603. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/mace

[30] Jason Mars and Lingjia Tang. 2013. Whare-map: Heterogeneity in Homogeneous Warehouse-scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM Press, New York, NY, 619–630. https://doi.org/10.1145/2485922.2485975

[31] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM Press, New York, NY, 248–259. https://doi.org/10.1145/2155620.2155650

[32] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, 561–577. http://dl.acm.org/citation.cfm?id=3291168.3291210

[33] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. 2009. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM Press, New York, NY, 221–234. https://doi.org/10.1145/1629575.1629597

[34] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM Press, New York, NY, 184–200. https://doi.org/10.1145/3132747.3132766

[35] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM Press, New York, NY, 69–84. https://doi.org/10.1145/2517349.2522716

[36] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-aware Thread Management. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX

Association, Berkeley, CA, 145–160. http://dl.acm.org/citation.cfm?id=3291168.3291180

[37] Hang Qu, Omid Mashayekhi, David Terei, and Philip Levis. 2016. Canary: A Scheduling Architecture for High Performance Cloud Computing. *CoRR* abs/1602.01412 (2016).

[38] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. 2016. PerfOrator: Eloquent Performance Models for Resource Optimization. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. ACM Press, New York, NY, 415–427. https://doi.org/10.1145/2987550.2987566

[39] Barret Rhoden, Kevin Klues, David Zhu, and Eric Brewer. 2011. Improving Per-node Efficiency in the Datacenter with New OS Abstractions. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing (SOCC '11)*. ACM Press, New York, NY, 25:1–25:8.

[40] Luis F. G. Sarmenta and Satoshi Hirano. 1998. Building and Studying Web-Based Volunteer Computing Systems Using Java. *Future Generation Computer Systems, Special Issue on Metacomputing* 15, 5 (Nov. 1998).

[41] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM Press, New York, NY, 351–364. https://doi.org/10.1145/2465351.2465386

[42] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, 69–87. http://dl.acm.org/citation.cfm?id=3291168.3291175

[43] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. 2013. Schedule Processes, Not VCPUs. In *Proceedings of the 4th Asia-Pacific Workshop on Systems (APSys '13)*. ACM Press, New York, NY, Article 1, 7 pages.

[44] spark team. 2018. Apache Spark: a unified analytics engine for large-scale data processing. Retrieved March 30, 2019 from https://spark.apache.org

[45] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. 2015. The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM Press, New York, NY, 62–75. https://doi.org/10.1145/2830772.2830803

[46] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. 2011. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM Press, New York, NY, 283–294. https://doi.org/10.1145/2000064.2000099

[47] Apache Hadoop Team. 2012. Hadoop 1.0.4 Release Notes. http://hadoop.apache.org/docs/r1.0.4/releasenotes.html

[48] Priyanka Tembey, Ada Gavrilovska, and Karsten Schwan. 2014. Merlin: Application- and Platform-aware Resource Allocation in Consolidated Server Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. ACM Press, New York, NY, Article 14, 14 pages.

[49] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen OḾalley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM Press, New York, NY, Article 5, 16 pages.

[50] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM Press, New York, NY, Article 18, 17 pages.

[51] Jingjing Wang and Magdalena Balazinska. 2017. Elastic Memory Management for Cloud Data Analytics. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Berkeley, CA, 745–758. https://doi.org/10.1145/2987550.2987576

[52] David Wentzlaff, Charles III Gruenwald, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. 2010. An Operating System for Multicore and Clouds: Mechanisms and Implementation. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM Press, New York, NY, 3–14. https://doi.org/10.1145/1807128.1807132

[53] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM Press, New York, NY, 607–618. https://doi.org/10.1145/2485922.2485974

[54] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. 2016. Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading. In *2016 USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 309–322. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yang

[55] Zhibin Yu, Zhendong Bei, and Xuehai Qian. 2018. Datasize-Aware High Dimensional Configurations Auto-tuning of In-Memory Cluster Computing. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (ASPLOS '18)*. ACM Press, New York, NY, 564–577. https://doi.org/10.1145/3173162.3173187

[56] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. ACM Press, New York, NY, 265–278. https://doi.org/10.1145/1755913.1755940

[57] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*. USENIX Association, Berkeley, CA, 15–28. https://doi.org/10.1145/2493123.2462904

[58] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM Press, New York, NY, 450–462. https://doi.org/10.1145/2465351.2465388

[59] Yunqi Zhang, Michael A. Laurenzano, Jason Mars, and Lingjia Tang. 2014. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 406–418. https://doi.org/10.1109/MICRO.2014.53

[60] Ming Zhao and Jorge Cabrera. 2018. RTVirt: Enabling Time-sensitive Computing on Virtualized Systems Through Cross-layer CPU Scheduling. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM Press, New York, NY, Article 27, 13 pages.

[61] Haishan Zhu and Mattan Erez. 2016. Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM Press, New York, NY, 33–47. https://doi.org/10.1145/2872362.2872394