TU Delft

Delft University of Technology

# Workload-Adaptive Configuration Tuning for Hierarchical Cloud Schedulers

Han, Rui; Liu, Chi Harold; Zong, Zan; Chen, Lydia Y.; Liu, Wending; Wang, Siyi; Zhan, Jianfeng

**DOI**
10.1109/TPDS.2019.2923197

**Publication date**
2019

**Document Version**
Final published version

**Published in**
IEEE Transactions on Parallel and Distributed Systems

**Citation (APA)**
Han, R., Liu, C. H., Zong, Z., Chen, L. Y., Liu, W., Wang, S., & Zhan, J. (2019). Workload-Adaptive Configuration Tuning for Hierarchical Cloud Schedulers. *IEEE Transactions on Parallel and Distributed Systems*, *30*(12), 2879-2895. Article 8741093. https://doi.org/10.1109/TPDS.2019.2923197

**Important note**
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

**Copyright**
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

**Takedown policy**
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

*This work is downloaded from Delft University of Technology.*
*For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.*

# Workload-Adaptive Configuration Tuning for Hierarchical Cloud Schedulers

Rui Han, Chi Harold Liu, *Senior Member, IEEE*, Zan Zong, Lydia Y. Chen, *Senior Member, IEEE*, Wending Liu, Siyi Wang, and Jianfeng Zhan

**Abstract**—Cluster schedulers provide flexible resource sharing mechanism for best-effort cloud jobs, which occupy a majority in modern datacenters. Properly tuning a scheduler's configurations is the key to these jobs' performance because it decides how to allocate resources among them. Today's cloud scheduling systems usually rely on cluster operators to set the configuration and thus overlook the potential performance improvement through optimally configuring the scheduler according to the heterogeneous and dynamic cloud workloads. In this paper, we introduce AdaptiveConfig, a run-time configurator for cluster schedulers that automatically adapts to the changing workload and resource status in two steps. First, a comparison approach estimates jobs' performances under different configurations and diverse scheduling scenarios. The key idea here is to transform a scheduler's resource allocation mechanism and their variable influence factors (configurations, scheduling constraints, available resources, and workload status) into business rules and facts in a rule engine, thereby reasoning about these correlated factors in job performance comparison. Second, a workload-adaptive optimizer transforms the cluster-level searching of huge configuration space into an equivalent dynamic programming problem that can be efficiently solved at scale. We implement AdaptiveConfig on the popular YARN Capacity and Fair schedulers and demonstrate its effectiveness using real-world Facebook and Google workloads, i.e., successfully finding best configurations for most of scheduling scenarios and considerably reducing latencies by a factor of two with low optimization time.

**Index Terms**—Cloud datacenter, cluster scheduler, configuration, job latency, YARN

## 1 Introduction

Today's cloud workloads are increasingly dominated by a mix of long-running production/service jobs and best-effort analysis/engineering jobs [50], [52]. The service jobs, such as storage services and web search engines, have strict latency deadlines and high priorities of using resources (usually by reservation [23]). The *best-effort* jobs, such as batch data analytic and software development/test, have different sensitivities to the latency according to their priorities. This paper focuses the latter type, which forms most of cloud jobs and tend to be short running [60]. For example, in the Google [52] and Alibaba [22] datacenters, 66.7 and 75.3 percent of jobs run less than 5 minutes. When dealing with massive jobs of diverse workload characteristics, most production cloud providers, including Google, Facebook, and Cloudera, employs *hierarchical scheduling* for resource allocation [15], [54]. Specifically, they develop hierarchical schedulers to divides users into different groups (departments) according to their organizational structure and allocates resources within each group according to job priorities. Such schedulers provide the flexibility of sharing resources among jobs, but also causes additional management complexity of the cluster [54].

*Example*. Fig. 1 shows a typical hierarchical scheduler, which employs two-level queues to share available resources among user groups (first-level queues) and jobs (second-level queues) under scheduling constraints. Its configurable parameters that are critical to job performance can be divided into two parts [15]: *cluster-level configuration* that decides the allocation of cluster resources to different groups; *group-level configuration* that controls the resource allocation among priority queues in a group and the job scheduling policy within each queue. For example, in YARN Capacity and Fair schedulers [59], the "ratio" parameters (cluster-level configurations) decide the percentages of resources assigned to the $m$ groups. For the group-level configurations, the "capacity" and "weight" parameters control the resource sharing among the second-level priority queues (e.g., $q_{11}$ and $q_{12}$); and the "scheduling policy" parameter, such as FIFO, Fair and Dominant Resource Fairness (DRF) [30], determines the resource assignment among jobs within the same queue.

*Configuration-Sensitive Scheduler*. Given a cluster scheduler, the choice of scheduling configurations leads to very different performances of jobs, as minor tweaking on such parameters can lead to changes in resource allocations and


- R. Han, C.H. Liu, and W. Liu are with the Beijing Institute of Technology, Beijing Shi 100091, P.R. China. E-mail: {hanrui, 1120152058}@bit.edu.cn, liuchi02@gmail.com.
- Z. Zong is with the Tsinghua University, Beijing Shi 100091, P.R. China. E-mail: zongz17@mails.tsinghua.edu.cn.
- L.Y. Chen is with the TU Delft, Delft 2628, CD, the Netherlands. E-mail: lydiaychen@ieee.org.
- S. Wang and J. Zhan are with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, P.R. China. E-mail: {wangsiyi, zhanjianfeng}@ict.ac.cn.


*Manuscript received 10 Dec. 2018; revised 1 May 2019; accepted 5 June 2019. Date of publication 19 June 2019; date of current version 8 Nov. 2019.*
*(Corresponding author: Chi Harold Liu.)*
*Recommended for acceptance by Y. Yang.*
*For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.*
*Digital Object Identifier no. 10.1109/TPDS.2019.2923197*

1045-9219 © 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

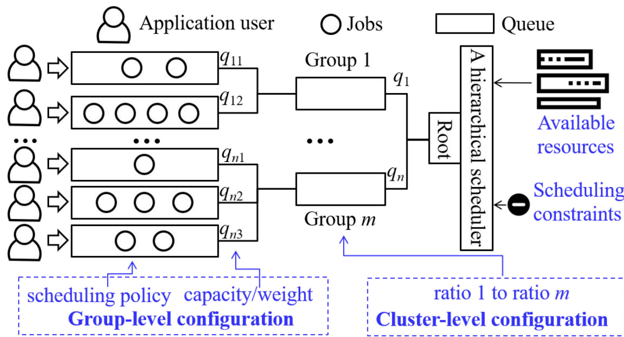



Fig. 1. An example of YARN fair scheduler in a cluster.

thus result in large discrepancies in job performance [13]. Our evaluations of Facebook and Google cloud jobs on the YARN Capacity and Fair schedulers show that setting either *best group-level* or *cluster-level* configurations outperforms other ones by achieving more than 50 percent improvement in job performance (Section 2.3). However, it is *no mean feat to gain potential performance improvement* from setting the best configuration because the performance impact of different configurations as well as the best one vary depending on a wide range of variable factors. These factors include the workload status (that is, the numbers of jobs in different queues and their workload characteristics represented by the job types and input data sizes [52]), the available resources, and the scheduling constraints (e.g., queue capacity and machine placement [31]).

*Challenges*. Most cluster schedulers for modern cloud datacenters, whether used in practice (e.g., Google Borg [60] and Hadoop YARN [7]) or described in the literature [15], [19], [47], rely on cluster administrators to manually set scheduling configurations [13] in an off-line fashion. However, today's cloud environments usually have to tackle with varying workloads, which exhibit highly heterogeneous and dynamic runtime behaviors [52]. In such environments, static configuration tuning approaches may overlook the potential improvement through on-line configuration turning, because the optimal one is highly volatile. Two major challenges in practice arise when adapting a scheduler's configuration to maximize the performance of workloads.

First, given a configuration, a scheduler's performance impact (namely jobs' performance under this configuration) is determined by both the scheduler's resource allocation mechanism and a list of variable factors such as workloads, available resources, and scheduling constraints. Hence the proposed technique needs to be applicable to diverse scheduling scenarios in the cloud.

Second, the volatile workload dynamics of best-effort jobs means the number and distribution of these jobs continuously change in the cluster (e.g., most Google and Facebook jobs complete within a few minutes), and the newly submitted jobs may have different workload characteristics. Meanwhile, today's scheduler needs to manage a cluster consisting of 10,000 and more machines, and a large number of concurrent users and jobs [60]. This gives rise to another major challenge about how to efficiently searching through the large parameter space of scheduler configuration for dynamically changing workloads at run-time.

This paper proposes a systematic approach to enable workload-adaptive tuning of scheduler configuration for large clusters. Our run-time configuration approach, called AdaptiveConfig, is designed to find a scheduler's best configuration for the latest mix of jobs via online reasoning and search according to workload and resource characteristics. Note that AdaptiveConfig differs from traditional reconfiguration techniques for VM schedulers [17], which allocate resources among VMs to improve the performance of *long-running and real-time service jobs* [63]. In contrast, the cluster schedulers studied in this work focus on *best-effort jobs*. In detail, we make the following technical contributions:

*Rule Engine Based Performance Comparator*. Based on the Drools rule engine [4], AdaptiveConfig formally describes the scheduler's resource allocation mechanisms using *business rules*. Moreover, it performs fine-grained differentiation of the factors that influence job scheduling and simultaneously processes these correlated factors as *facts* of Drools, thus constructing an comparator of configurations' performance impacts under diverse scheduling scenarios.

*Workload-Adaptive Configuration Optimizer*. AdaptiveConfig presents a run-time optimizer that turns configuration parameters automatically according to the changing workloads. To achieve on-line tuning, the optimizer implements an efficient searching method for best group-level configuration, and transforms the cluster-level configuration searching problem into an equivalent dynamic programming (DP) problem whose structures of optimal actions can be explored for efficient solutions at scale.

*Evaluation Using Testbed and Simulation*. To demonstrate the effectiveness of our approach, we implemented it on two typical YARN schedulers, namely the Capacity and the Fair schedulers, and evaluated it using real jobs derived from the Facebook [21] and Google [52] production traces. The evaluation results on a testbed (cluster) show: (i) AdaptiveConfig correctly selects the best configurations for 87.26 percent of the different mixes of jobs, while also selecting the next-best ones for the remaining jobs; (ii) our approach responds to changing workloads effectively by selecting the appropriate scheduling configuration that achieves considerable performance improvement: job latencies are reduced by an average of 2.16 times compared to the configurations without our approach. Moreover, we conducted simulations on the YARN Scheduler Load Simulator (SLS) [12], YARN's official platform for large-scale cluster and load simulations. The simulation results on a 12k-machine cluster (a typical Google [52] and Alibaba [22] cluster size) show AdaptiveConfig outperforms representative configurations by achieving 1.94 times reductions in job latency with an optimization time of less than 3 seconds.

The remainder of this paper is organized as follows: Section 2 introduces the related work and motivation of this work, Section 3 explains our approach, Section 4 evaluates it, and finally, Sections 5 summarizes the work. Portions of this work appear in a previous conference paper [36] and we have largely extended the article, by demonstrating the key challenges using concrete cases and measurement studies (see Section 2.3), formulating the best configuration searching problem (see Section 3.2) and method (Section 3.3), developing a new DP-based method to find the globally best configurations for multiple user groups (see Section 3.4), and

TABLE 1
Overview of Cluster Scheduling Techniques in the Cloud

| Categories | Representative techniques | |
|---|---|---|
| Cluster resource management systems | Production systems | Google (Borg [60], Omega [55], Kubernetes [6]), Microsoft (Autopilot [42], Apollo [16]), and Facebook Tupperware [5] |
| | Open source systems | Apache Mesos [39], Hadoop YARN [59], and Docker Swarm [3] |
| Improvement techniques for cluster schedulers | Distributed schedulers | Distributed schedulers (Sparrow [49]), hybrid schedulers (Mercury [45], Hawk [25], and Eagle [24]) |
| | Extension of resource dimensionality | I/O resources [34], [67] and GPU [58] |
| | Task-level optimization | DAG jobs [35], coordination of parallel tasks in Spark jobs [29], task queue management in nodes [51], and reduction of preemption overheads using Docker [20] |
| Scheduling techniques with constraints | MapReduce jobs | Data locality [62], [65], task dependance [66], network traffic and deadlines [18], [47] |
| | Resource contention | Contention of heterogeneous resources [26], [27]) |
| | Task placement | Fair scheduling [31], [32], geo-distributed datacenters [19] |
| | General framework | Phoenix [57] |
| Configuration tuning for individual jobs | Hadoop and Spark jobs | Hadoop jobs [14], [37], [38], [38], [56], [61], Spark jobs [33], [40] |
| | Jobs on cloud server systems | Storage systems [53], Tomcat and database jobs [68] |

extending the evaluations by adding Google workloads (see Section 4.3) and multi-group scenarios (see Section 4.5).

## 2 Background and Motivation

To motivate our focus on optimizing scheduler configurations, this section first explains existing cluster schedulers (Section 2.1) and the related techniques in the cloud (Section 2.2). By testing concrete cases using jobs in real cloud traces, it then illustrates the challenges of workload-adaptive configurations for cloud scheduler (Section 2.3).

### 2.1 Cluster Schedulers in the Cloud

In traditional HPC supercomputing centers, cluster schedulers (e.g., Maui [43]) typically use long waiting queues to achieve high resource utilizations [60]. In contrast, the next generation cluster management systems for cloud datacenters need to address more challenging scheduling problems such that most cloud jobs have heterogenous workload characteristics, multi-dimensional resource demands, and short durations [8]. Table 1 summarizes the representative cluster scheduling techniques in the cloud.

#### 2.1.1 Cluster Resource Management Systems in the Cloud

*Production Systems*. Mainstream cloud service providers usually develop their own cluster resource management systems. Google Borg [60] is a pioneer system that divides cloud jobs into high-priority service jobs and low-priority batch jobs, and schedule these jobs with consideration of multiple resource dimensionalities including CPU, memory, disk, and network. Many enterprisers develop similar systems to Borg, such as are Microsoft Autopilot [42] and Facebook Tupperware [5]. In recent years, Google launches a new system (Kubernetes [6]) for the new generation container technology (Docker [48]), and Microsoft's Apollo [16] reduces job scheduling latency using a distributed scheduling framework.

*Open Source Systems*. Before Borg, Mesos [39] is the first cluster resource management system released by UC Berkeley. Mesos increases the cluster resource utilization using a two-level scheduler, which shares resources among multiple computing frameworks (e.g., Hadoop, Spark and Storm) as well as jobs within each framework. Mesos's design philosophy has a profound impact on the following systems: YARN [59] employs hierarchal schedulers to share resources among multiple organizations and users; Google Omega [55] improves Mesos's passive resource sharing mechanism and proposes an active mechanism that allows jobs to autonomously compete for resources based on shared states; and Swarm [3] is a specialized cluster management system designed for Docker.

Existing cluster resource management systems usually provide configuration parameters in their schedulers to control the resource allocation and job scheduling process [8], and support run-time adjustment of these configurations [1], [59]. However, to the best of our knowledge, existing systems only rely on cluster operators to manually set the configuration and may loss the opportunities to improve jobs' performance through dynamically configuring their schedulers according to the changing workload and resource status in the cluster. This work is built upon the above configurable schedulers and it focuses on tuning their configurations at run-time.

### 2.2 Related Work

#### 2.2.1 Improvement Techniques for Cluster Schedulers

Many techniques have been proposed to improve cluster schedulers from different aspects.

*Distributed and Hybrid Schedulers*. Traditional centralized cluster schedulers such as Borg and YARN have the problem of long scheduling latencies when handling high throughput of jobs. To address this issue, Sparrow [49] proposes a

distributed framework for fast scheduling decisions of 100 millisecond jobs. However, distributed schedulers have a limited visibility of the whole cluster status, thus cannot make optimal scheduling decisions or strictly guarantee fairness constraints. Hence, hybrid schedulers (e.g., Mercury [45], Hawk [25], and Eagle [24]) make trade-off between high quality in centralized schedulers and low latency in distributed schedulers.

*Extension of Resource Dimensionality*. Many basic cluster schedulers only allocate resources of processor cores and memory, but ignore the constrained I/O resources such as disk and network bandwidths. Hence, some work studies the explicit allocation of network resources [34], [67]. In addition, alsched [58] maximizes the efficiency of job scheduling by considering both CPU and GPU resource constraints.

*Task-Level Optimization*. These techniques optimize the execution of jobs at the task level. Graphene scheduler [35] is designed for directed acyclic graph (DAG) jobs with complex task dependencies. It identifies straggling tasks and schedules them first. Similarly, AutoPath scheduler [29] adjusts resource allocation among parallel tasks to coordinate their completion times. In addition, Yaq-c and Yaq-d [51] focus on managing the task queue at each node such as restricting the queue length or selecting scheduling algorithms. BIG-G uses Docker to keep a task's status when it is preempted, thus decreasing its resumption overheads [20].

Our current approach is designed for standard hierarchial schedulers used by mainstream cloud providers [54], and it is orthogonal to the above scheduling improvement techniques.

#### 2.2.2 Constraint-Based Job Scheduling

In the cloud, cluster schedulers usually have various constraints listed as follows.

*MapReduce jobs* are one major type of best-effort jobs in the cloud. Early scheduling techniques usually concern two constraints in these jobs: data locality (moving tasks and their computations close to data) [62], [65] and task dependance [66]. Some recent work considers the restrictions of network traffic and deadlines [18], [47].

*Resource Contention*. In a resource-sharing cloud environment, the resource contention among co-located jobs considerably degrade service jobs' perfomance. Considering this constraint, Paragon [27] utilizes collaborative filtering to recommend suitable resources for jobs based on their running history. Similarly, Quasar [26] divides the contention of resources into different patterns including memory, cache, disk and network bandwidthes, and uses history logs to construct classifiers that decide the optimal resource allocation of jobs.

*Task Placement*. This determines the machine a task can run. Choosy scheduler [31] and Firmament scheduler [32] extend the max-min fairness algorithm and the min-cost max-flow algorithm to support this constraint. It is also studied within the context of geo-distributed datacenters and data locality [19].

*General Framework*. Phoenix [57] is a generic constraint-aware scheduling framework, which considers both heterogenous resources (e.g., GPU, FPGA and different storage devices such as solid-state drive (SSD) and hard disk drive (HDD)) and other constraints (e.g., task placement and deadlines).

The above constraints can be regarded as an influence factor when estimating a scheduler configuration's performance in our approach.

#### 2.2.3 Configuration Tuning for Individual Jobs

Some techniques are developed to automatically adjust configurations that control the running of individual jobs.

*Hadoop and Spark Jobs*. On Hadoop, the running of a MapReduce job is controlled by over 190 configuration parameters [37], [38], such as the number of map and reduce tasks, the memory allocation of each task, whether to compress data in data shuffling. All these parameters influence the job's performance. To this end, Starfish [38], MRTuner [56], FRESH [61], and RFHOC [14] profile the behaviors of jobs and develop automatic configuration tuning methods on Hadoop for optimized performance. On Spark, the memory configuration parameters have a large impact on job performance, and machine learning techniques are used to optimize these configurations [40]. Some other algorithms are developed to configuring dynamic partitioning of a Spark job to minimize its resource consumption within a user-defined latency [33].

*Jobs on Cloud Server Systems*. Similarity, some techniques are developed for single applications in cloud serving systems. ReCA studies I/O workloads in storage systems and develops a cache reconfiguration approach for a given application [53]. A configuration optimization approach is developed for a specified on Tomcat, database systems (Cassandra, MySQL, and Hive) or Spark [68].

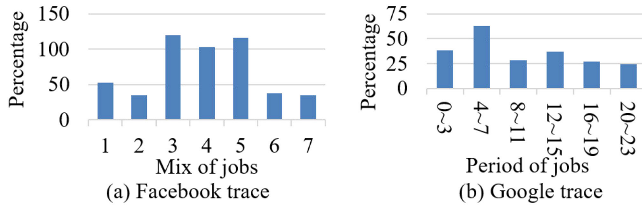
Existing configuration tuning techniques aim to minimize the *execution time* of an *individual* job, usually relying on profiling the running behaviour of the job. In contrast, this work studies the scheduler configurations that decide the resource allocation among *multiple* jobs and addresses the challenge in searching the best setting to minimize these jobs' latencies, including both *waiting times* before execution (depending on the resource allocation moments) and *execution times* (depending on the amount of allocated resources).

### 2.3 Challenges

This section motivates the challenges considered here by testing YARN Capacity and Fair schedulers on traces of real workloads.

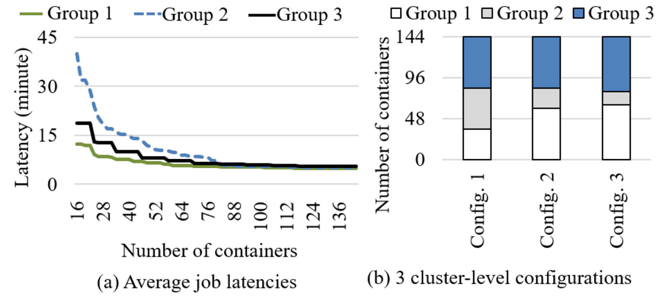
#### 2.3.1 Workloads and Schedulers

We analyzed traces from two production cloud systems, namely a month-long trace from a 12K-machine Google cluster [52] and a 7.5 month-long trace from a 3.6K-machine Facebook cluster [21], to show the workload characteristics of cloud best-effort jobs: (1) *workload heterogeneity*. These jobs are submitted by users of various application domains and hence have different application types. Each application type has a wide range of input sizes ranging from KB to TB. (2) *Workload dynamicity*. In both traces, short and medium best-effort jobs that complete within dozens of seconds or minutes account for the majority. For example, 66.7 percent of Google jobs complete less than 5 minutes, and their median duration is



(a) Facebook trace (b) Google trace

Fig. 2. Percentages of job latency increase in other configurations compared to the best ones.



(a) Average job latencies (b) 3 cluster-level configurations

Fig. 3. Evaluation of cluster-level configurations.

3 minutes; 47.07 and 98.79 percent of Facebook jobs complete within in 32 seconds and 21 minutes.

*Benchmarks*. In evaluation, the workload characteristics of a job is represented by its submission time and the resource usages of its tasks. The Facebook jobs are generated by the Statistical Workload Injector for MapReduce (SWIM) benchmark [9], which emulates the operations of reading, writing, shuffling and sorting data in MapReduce jobs. The Google jobs are generated by the CloudMix benchmark [2].

*YARN Scheduler Configurations*. We evaluate both Google and Facebook jobs using YARN Capacity and Fair schedulers. Both schedulers have dozens of configurable parameters, in which a few ones determine resource allocation within a user group and hence they are critical to job performance [13].

- In the *Capacity* scheduler, the "capacity" parameter controls the allocation of resource to different queues (two queues, called $q_A$ and $q_B$, are used in the example). We set five configuration values of this parameter in $q_A(q_B)$: $\frac{1}{6}\,(\frac{5}{6})$, $\frac{1}{3}\,(\frac{2}{3})$, $\frac{1}{2}\,(\frac{1}{2})$, $\frac{2}{3}\,(\frac{1}{3})$, and $\frac{5}{6}\,(\frac{1}{6})$.
- In the *Fair* scheduler, the "weight" parameters determines the resource sharing proportions between queues and each queue has its "schedulingPolicy" parameter. We set the same configuration values of the "weight" parameters as the previous "capacity" parameter, and 3 job scheduling policies within queues: FIFO, Fair, and dominant resource fairness (DRF).

#### 2.3.2 Challenge in Tuning Group-Level Configurations

*Scheduling Scenarios*. For the Facebook jobs, we generate 14 different mixes of jobs submitted to two cluster sizes: six and eight containers (each container has 1 CPU core and 2 GB memory). For the Google jobs, we generate 18 different mixes of jobs by considering six periods of a day (each period corresponds to four hours) and three platforms (each platform has its own machine types and resource capacities). The available resource to each mix of jobs is decided by the actual nodes assigned to these jobs in the trace. Overall, we tested 640 different cases of job scheduling.

*Evaluation of Group-Level Configurations*. We term *best configuration* as the one that results in the lowest job latency. The evaluation results show: (i) the best configuration varies when encountering either different mixes of jobs or available resources. It covers 100 and 86.67 percent of the optional configurations in the Capacity and Fair schedulers, respectively; (ii) each optional configuration experiences a similar probability to be the best one. Fig. 2 further reports the average percentage of increased latency when comparing other configurations to the best one. One can observe that these configurations considerably increase job latencies by an average of 54.95 percent.

*Challenge*. There is no "one-size-fits-all" best configuration in the above evaluations. The problem is compounded when considering resource allocation mechanisms in different schedulers and scheduling constraints. This gives rise to the key challenge about how to efficiently reason about the performance impact of different configurations under such various influence factors and scheduling scenarios.

#### 2.3.3 Challenge in Tuning Cluster-Level Configurations

*Evaluation of Cluster-Level Configurations*. We now discuss the performance impact of the cluster-level configuration, i.e., the "ratio" parameters that control the proportions of resources assigned to different user groups. Taking three groups (group 1 to 3) and the Capacity scheduler as an example, we first test how the different resource allocations affect the job performance even given the *best group-level configuration* in the previous evaluation. Fig. 3a demonstrates the fluctuations of job latencies under different numbers of containers. We can see that the resource assignment considerably affects the job latency in each group. In particular, group 2 has the highest latency and it is 7.87 times larger than the lowest one. We further consider a scenario of 144 containers that are shared across three groups under three cluster-level configurations, as shown in Fig. 3b. The evaluation result shows configuration 1 achieves the lowest job latency (8.4 minutes), which is 59.35 percent shorter than those of the other two configurations.

*Challenge*. We note that there are only three groups and three configurations in this simple example. In real cloud clusters, there exist thousands of users and dozens of groups (e.g., 50) in a cluster [60] and each group may have a list of "ratio" parameters that lead to different job performances (e.g., more than 30 in Fig. 3a's three groups). Hence at the datacenter scale, there exists a huge number (e.g., $50^{30}$) of possible combination of cluster-level configurations and how to efficiently search this configuration space for the best setting is a major challenge to be addressed.

## 3 AdaptiveConfig

In this section, we first describe the design overview of AdaptiveConfig in Section 3.1 and formulate the best configuration searching problem in Section 3.2, following by explaining its specific modules in Sections 3.3 and 3.4.

### 3.1 Overview

AdaptiveConfig aims to automatically configure a cluster scheduler according to the workload variation at run-time. It applies a periodical configuration tuning mechanism for a cluster scheduler using three steps, as shown in Fig. 4. At

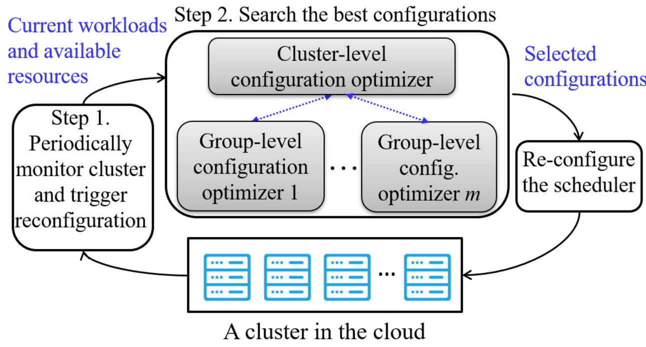



Fig. 4. Overview of AdaptiveConfig.

each tuning interval (step 1), it monitors the changes in workloads (that is, existing running and waiting jobs, and newly submitted jobs) and available resources, and decides which user groups of the cluster need reconfiguration. Specifically, a reconfiguration is triggered for a group if it has at least one newly submitted job that requires resource allocation in the current tuning interval.

At step 2, AdaptiveConfig employs two modules that work together to search the scheduler's best configuration. For each triggered group, the *group-level configuration optimizer* finds its best group-level configurations under different resource assignments, which are decided by the group's minimal and maximal resource capacities. Afterwards, the search results of all $m$ groups are forwarded to the *cluster-level configuration optimizer* to search the globally best cluster-level configuration. Finally, the scheduler is re-configured according to the found best configurations.

### 3.2 Problem Formulation

Consider a cloud cluster shared by $m$ groups, $o_i$, $1 \leq i \leq m$, and $R$ denotes its available resource for best-effort jobs. Note that the amount of resource is measured by the number of Linux containers, which are used in resource isolation and usage accounting in many mainstream cloud platforms such as Google and Alibaba. Each group $o_i$ has a set of $n$ priority queues $q_j$, $1 \leq j \leq n$. The *scheduling configurations* for $o_i$ are denoted by a triple $C_i = (c_i^{Ratio}, C_i^{Queue}, C_i^{Sche})$: $c_i^{Ratio}$ is the ratio of allocated cluster resource; $C_i^{Queue}$ is a set of parameters that decide the allocation of resources among $n$ queues; and $C_i^{Sche}$ is a set of parameters that decide the scheduling policy at each queue. Suppose $o_i$ has a set $J_i$ of jobs, its *scheduling constraints* are denoted by a pair $S_i = (S_i^{Queue}, S_i^{Job})$: $S_i^{Queue}$ is the set of constraints for its queues such as their minimum and maximum amounts of resources; and $S_i^{Job}$ is the set of constraints for its jobs such as their task placement constraints (the machines that a job's tasks can run).

*Best scheduler configuration* is defined as: given available resource $R$, jobs $J = \{J_i\}_{i=1}^m$, scheduling configurations $C=\{C_i\}_{i=1}^m$, and constraints $S = \{S_i\}_{i=1}^m$, we aim to find the configuration parameters that achieve the best job performance, $L$, over the full space $\mathbb{C}$ of parameter settings

$$C^* = \arg\min_{C \in \mathbb{C}} L(R, J, C, S). \quad (1)$$

Here, we specifically consider $L$ as the average job latency and minimize $L$. This model can be directly extended to a utility function that considers job latency, priority, and deadline [41]. We address the best configuration searching problem by decomposing $C^*$ into group-level and cluster-level best configurations.

*Group-Level Configuration Optimizer*. Given a resource assignment $R \cdot c_i^{Ratio}$ to a group $o_i$ $(1 \leq i \leq m)$, this optimizer searches the best configurations $C_i^{Queue*}$ and $C_i^{Sche*}$ that result in the lowest job latency for group $i$

$$L^*(o_i, R \cdot c_i^{Ratio}) = \min_{C_i^{Queue} \in \mathbb{C}_i^{Queue} \wedge C_i^{Sche} \in \mathbb{C}_i^{Sche}} L(R \cdot c_i^{Ratio}, J_i, C_i, S_i), \quad (2)$$

where $\mathbb{C}_i^{Queue}$ and $\mathbb{C}_i^{Sche}$ denote the subspaces of $\mathbb{C}$ consisting of only the parameters in $C_i^{Queue}$ and $C_i^{Sche}$.

*Cluster-Level Configuration Optimizer*. Based on the group-level best configurations, this optimizer finds the globally best setting of configurations $c_1^{Ratio*}$ to $c_m^{Ratio*}$ ($\sum_{i=1}^m o_i$=1) that results in the lowest job latency $l^*$ in the cluster

$$l^* = \min_{c_i^{Ratio} \in \mathbb{C}_i^{Ratio}} \sum_{i=1}^{m} L^*(o_i, R \cdot c_i^{Ratio}), \quad (3)$$

where $\mathbb{C}_i^{Ratio}$ denotes the parameter space of $c_i^{Ratio}$.

### 3.3 Group-Level Configuration Optimizer

This group-level configuration optimizer aims to efficiently search the configuration space to find the best one that minimizes the jobs' latencies. For each candidate configuration $C_i$, it employs function $L(R \cdot c_i^{Ratio}, J_i, C_i, S_i)$ to compare the job latencies with other configurations by taking the current jobs $J_i$ and the available resource $R \cdot c_i^{Ratio}$, and the scheduling constraints $S_i$ as inputs.

#### *3.3.1 Configuration-Sensitive Job Latency Comparison*

Algorithm 1 details the steps of function $L(R \cdot c_i^{Ratio}, J_i, C_i, S_i)$ in Eqn. (2). Its latency comparison process has several iterations. Each iteration corresponds to a resource allocation $r_A$ to a set of $k$ jobs in $|J_i|$ (line 4). The allocation decision depends on the scheduler's resource allocation mechanism under configuration $C_i$ and scheduling constraints $S_i$. Afterwards, the latencies of these $k$ jobs are estimated in turn (line 5 to 9). In estimation, a job $j_v$ starts running after the resource allocation and its waiting time $w_v$ is calculated by subtracting the submission time $b_v$ from the allocation moment $t_M$ (line 6). Its execution time $e_v$ and latency $l_v$ are then calculated (lines 7 and 8), and the job is added to set $J^{Run}$ of running jobs. After completing one resource allocation (line 4 to 11), the algorithm removes the job $j^*$ that finishes first in $J_i$ (line 12), resets resource allocation moment $t_M$ and available resource $r_A$ (lines 13 and 14), thereby starting the next iteration of resource allocation. Finally, the algorithm returns the average latency of all jobs (line 16).

We note here that *estimating a pending job's execution time* is a major building block of cluster schedulers in the cloud [28], [44] and the estimation error is determined by many factors such as the resource/machine type, the scheduler employed, and runtime uncertainties [50]. In contrast, function $L$ focuses on comparing job latencies across different scheduler configurations and implicitly assumes that the latencies are estimated under *the same factors*. Under this

assumption, we develop function $E(j_v, R_v^{job})$ to estimate for a multi-task job $j_v$'s execution time. The estimation is sensitive to configuration $C_i$ that decides the set $R_v^{job}$ of resource allocation to the job. Specifically, $R_v^{job}$ corresponds to one or multiple allocations and $(r, t) \in R_v^{Job}$ is a pair denoting the allocation time $t$ of resource $r$.

**Algorithm 1.** Job Latency Comparison $L(R \cdot c_i^{Ratio}, J_i, C_i, S_i)$

**Require:** $t_M$: the resource allocation moment;
$r_A$: the allocatable resources;
$R^{Job}$: the set of resource allocations to a job;
$J^{Run}$: $(t, r) \in J^{Run}$ is a pair denoting a job's completion time $t$ and released resource $r$;
$e$: A job's estimated execution time;
1. $r_A = R \times c_i^{Ratio}$;
2. $t_M = 0$;
3. **while** $(J_i \neq \phi)$ **do**
4. Allocate $r_A$ to the $k$ jobs in $J_i$ according to $C_i$ and $S_i$;
5. **for** ($v = 1$; $v \leq k$; $v$++) **do**
6. $w_v = \max\{t_M - b_v, 0\}$;
7. $e_v = E(j_v, R_v^{Job})$;
8. $l_v = w_v + e_v$;
9. $J^{Run} = J^{Run} \cup \{(b_v + l_v, R_v^{Job})\}$;
10. $J_i = J_i \backslash \{j_v\}$;
11. **end for**
12. $J^{Run} = J^{Run} \backslash \{(t^*, r^*)\}$ where $t^* = \min_{(t,r) \in J^{Run}} t$;
13. $t_M = R^{Job*}$;
14. $r_A = r^*$;
15. **end while**
16. Return the average latency of all jobs.

The steps of function $E(j_v, R_v^{job})$ are detailed in Algorithm 2. Given the set $Task_v$ of tasks in $j_v$, the algorithm simulates the assignment of these tasks to the containers in $R_v^{job}$ and this process has $|R_v^{job}|$ iterations (line 4 to 19). Each iteration corresponds to a resource allocation of $r_i$ containers $(r_i \geq 1)$, whose available moments are recorded in array $A$ as their allocation times (line 10 to 12). Suppose $n_C$ containers are allocated (line 13), the algorithm iteratively assigns tasks to containers under two conditions (line 14 to 18): (1) there exists a container whose available moment is smaller than the next resource allocation moment $t_M$; (2) there exists pending tasks. In each assignment, a pending task is allocated to the container with the earliest available moment $A[i^*]$ (line 15) and this container's available moment is updated as the time the task is completed (line 16). Finally, the algorithm calculates job $j_v$'s execution time as the interval between the first container's allocation moment and the last container's available moment (line 20). Similar to current work on job performance model [28], [46], [50], our approach estimates a task's execution time $e^{Task}$ by constructing models (e.g., regression models) based on job profiles or post running logs. The time model assumption was a homogenous cluster, in which all containers have the same instance type (represented by a container's machine type and resource capacity). This model can be extended by taking containers' instance type as input, thus supporting a heterogeneous cluster with multiple instance types.

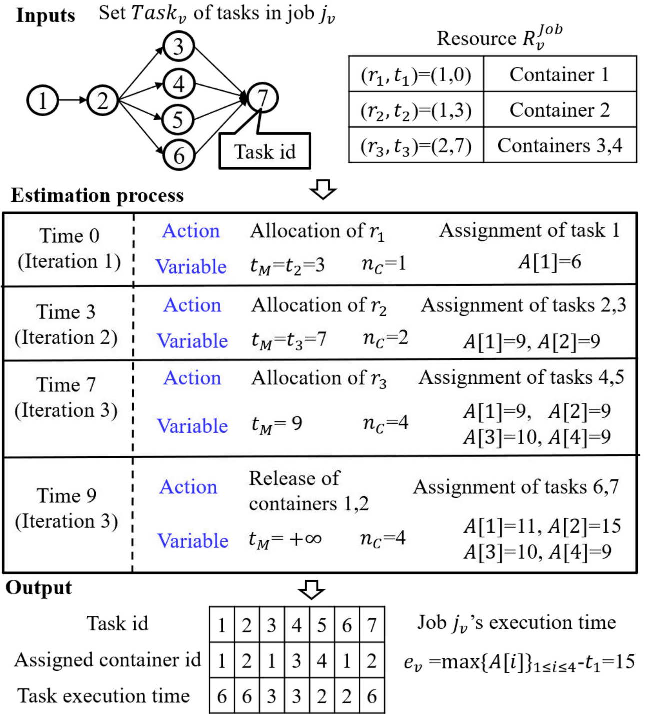



Fig. 5. An example of estimating a job's execution time and container release time under multiple resource allocations.

**Algorithm 2.** Job Execution Time Calculation $E(j_v, R_v^{job})$

**Require:** $t_M$: the next container's allocation moment;
$Task_v$: the set of tasks in job $j_v$;
$A$: An array of each container's available moment;
$e^{Task}$: a task's execution time.
1. Rank all the tasks in $Task_v$ according to their execution orders;
2. $n_C = 0$; //number of containers
3. $k = 0$; //number of assigned tasks
4. **for** ($i = 1$; $i \leq |R_v^{job}|$; $i$++) **do**
5. **if** $(i < |R_v^{job}|)$ **then**
6. $t_M = t_{i+1}$;
7. **else**
8. $t_M = +\infty$;
9. **end if**
10. **for** ($j = 1$; $j \leq r_i$; $j$++) **do**
11. $A[n_C + j] = t_i$;
12. **end for**
13. $n_C = n_C + r_i$;
14. **while** $(\max_{1 \leq i \leq n_C}\{A[i]\} < t_M$ and $k \leq |Task_v|)$ **do**
15. $i^* = \arg\min_{1 \leq i \leq n_C}\{A[i]\}$;
16. $A[i^*] = A[i^*] + e_k^{Task}$;
17. $k$++;
18. **end while**
19. **end for**
20. Return $\max_{1 \leq i \leq n_C}\{A[i]\}$-$t_1$.

Fig. 5 shows an example of estimating job $j_v$'s execution time using Algorithm 2. This job has seven tasks, in which tasks 3 to 6 are parallel tasks with the same execution order. That is, they have arbitrary assignment orders in Algorithm 2. Job $j_v$ obtains three resource allocations during its execution process and thus the estimation process consists of

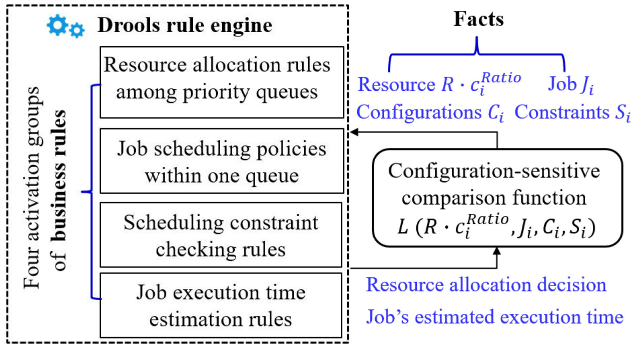



Fig. 6. Group-level configuration optimization based on Drools.

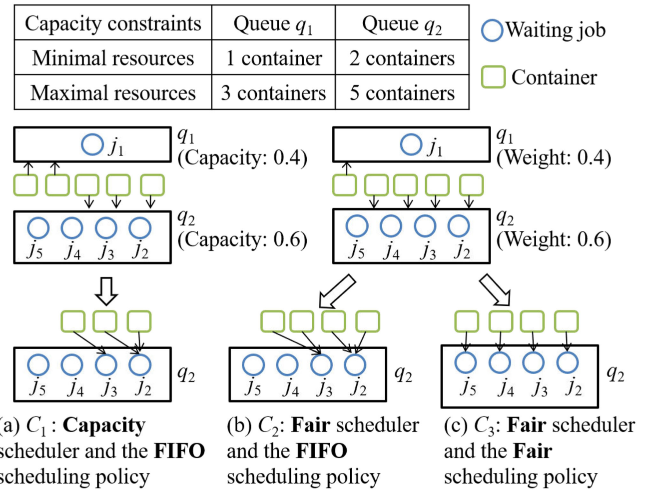



Fig. 7. Performance estimation under three scheduling configurations.

three iterations. At iteration 1, when $r_1$ is allocated at time 0, task 1 is assigned to container 1 and this container's available moment is 6. At iteration 2, when $r_2$ is allocated at time 3, only container 2 is available and task 2 is assigned to it. At iteration 3, when $r_3$ is allocated at time 7, tasks 2 and 3 are running in containers 2 and 1, so tasks 4 and 5 are assigned to the two newly released containers. Finally, at time 9, containers 1 and 2 are released and tasks 6 and 7 are assigned to them. After all seven task assignments, $j_v$'s execution time is calculated.

### 3.3.2 Rule Engine Based Implementation

To support different schedulers and diverse workload characteristics, we implement the comparison function $L$ based on the Drools Expert rule engine [4] with *two objectives*. First, it explicitly differentiates a schedulers resource allocation mechanism under different configurations and formally describes these mechanisms as *business rules* in Drools. Second, it transforms the *factors* that influence the job scheduling into facts of Drools, thus providing the ability to handle variations in workloads and available resources. Drools is used here because it provides highly efficient reasoning algorithms that scale to a large number of business rules and facts, while also offering conflict resolution strategies in reasoning. We now introduce the details of business rules and facts in the comparison function.

*Business Rules*. The Drools Expert allows convenient definition of a business rule as prerequisites ("When/If" statement) and actions ("Then") and manages different business rules using *activation groups* (the rules in the same group cannot be fired together). For example, a queue can only apply one job scheduling algorithm (e.g., FIFO or DRF), hence in Drools, each scheduling algorithm is defined as a business rule and all the algorithms are in the same activation group.

Fig. 6 shows four activation groups of business rules: (1) *Resource allocation rules among priority queues*. Example rules are the static resource allocations in the YARN Capacity scheduler and the dynamic resource sharing mechanisms in the YARN Fair scheduler. (2) *Job scheduling policies within one queue*. Example policies are FIFO, DRF, Earliest-deadline-first (EDF), and Least Remaining Time First (LRTF). (3) *Scheduling constraint checking rules*. These rules define the checking methods for scheduling constraints such as queue capacity (the minimal and maximal amounts of resources that can be allocated to a queue) and machine placement (the machines that a job can run). (4) *Job execution time estimation rules*. These rules estimate a job's execution time under a resource allocation.

*Facts*. For a user group $o_i$, the comparison function $L$ comprehensively considers the factors (available resource $R \cdot c_i^{Ratio}$, jobs $J_i$, configurations $C_i$, and constraints $S_i$) that influence job scheduling and simultaneously reasons about these correlated factors as *facts* of the rule engine. The rule engine then outputs resource allocation decision and tasks' estimated execution times to the function (Algorithm 1).

*Example*. Fig. 7 shows an example of assigning five containers (available resources) to five waiting jobs ($j_1$ to $j_5$) on the YARN Capacity and Fair schedulers. Either scheduler has two queues and each queue has its capacity constraint. The example has three configurations: $C_1$ (Capacity scheduler with FIFO queue), $C_2$ (Fair scheduler with FIFO queue), and $C_3$ (Fair scheduler with Fair queue). The resource demand is one container for $j_1$ and two containers for each of the other four jobs. Under $C_1$ (Fig. 7a), fixed amounts of resource are allocated to the two queues (two containers to $q_1$ and three containers to $q_2$). Two head-of-queue jobs, $j_2$ and $j_3$, receive the three containers in a FIFO manner. By contrast, the Fair scheduler (Figs. 7b and 7c) is supposed to allocate two (three) containers to $q_1$ ($q_2$); however $j_1$ only requests one container in $q_1$ and thus the rest of four containers are assigned to $q_2$. Under $C_2$ (Fig. 7b), the FIFO scheduling policy assigns the four containers of $q_2$ to the two first submitted jobs. Under $c_3$ (Fig. 7c), the Fair scheduling policy equally assigns the four containers to the four waiting jobs in $q_2$. Finally, the optimizer calculates all five jobs' execution times according to their resource assignments using the job execution time estimation rules.

### 3.3.3 Searching the Best Group-Level Configuration

The optimizer searches within their domains of configurations $C_i^{Queue}$ and $C_i^{Sche}$ to find the best configuration parameters that minimize the job latencies. It first utilizes the quantization technique to generate a search space. Specifically, for a configuration parameter, its domain can either be continuous (e.g., the "Capacity" parameter in the Capacity scheduler) or discrete (e.g., the parameter to select the

job scheduling algorithm can be FIFO, DRF, or EDF). Using the quantization method, this domain is equally or randomly discretized into $d$ values. When considering all $k$ parameters in $\mathbb{C}_i^{Queue}$ and $\mathbb{C}_i^{Sche}$, the space of possible settings is constructed as a grid of size $d^k$. For each setting (a point in the grid), the optimizer uses the comparison function to calculate the jobs' latencies and thus selects the setting with the lowest latency.

To search the parameter space efficiently, we develop an approximate method based on Recursive Random Search (RRS) [64]. RSS is a heuristic algorithm for black-box optimization problems and it provides probabilistic guarantees on the distance between the found best setting and the actual best one. The RSS-based method starts from the whole parameter space and searches the best solution with multiple iterations. Each iteration consists of three steps: (1) given a pre-specified confidence probability $p$, the method randomly samples the current space to find the point (configuration setting) with the lowest latency; (2) it then shrinks both the parameter space and the search granularity according to a scaling down factor $\gamma$; and (3) it samples in the space with the granularity. The methods terminates until the search granularity meets the required degree of accuracy.

**Proposition 3.1.** *The time complexity of the RRS-based optimizer is $O(\frac{ln(1-p)}{ln(1-g)} \times \log_{\gamma} \frac{g^t}{g^0})$, where $p$ represents the probability of find the best setting, $g$ is the search granularity at one iteration, $g^0$ denotes the initial/coarsest search granularity, and $g^t$ denotes the required/finest search granularity, and $\gamma$ denotes the scaling down factor at each iteration.*

**Proof.** At one iteration, let $n$ be the number of samples required to meet the confidence probability $p$, we have: $p = 1 - (1-g)^n$, where $(1-g)$ represents the probability of finding a non-optimal setting in one sample. Hence $n$ can be calculated as: $c = \frac{ln(1-p)}{ln(1-g)}$. Let $x$ be the number of iterations to meet the required search granularity $g^t$, we have: $g^t = g^0 \times c^{x-1}$, and $x$ can be calculated as: $x = log_{\gamma} \frac{g^t}{g^0} + 1$. When considering all $x$ iterations, the total time complexity of the algorithm is $O(\frac{ln(1-p)}{ln(1-g)} \times \log_{\gamma} \frac{g^t}{g^0})$. □

*Example*. Suppose the Fair scheduler has two parameters and the search granularity $g = \frac{1}{27}$ (that is, the "weight" parameter and the "scheduling policy" parameters are discretized into nine and three values, respectively). The RRS-based optimizer takes $\frac{ln(1-0.99)}{ln(1-\frac{1}{27})}$ = 122 samples to find the best setting with confidence probability $p$ = 0.99. Suppose the initial search granularity $g^0 = \frac{1}{27}$, the required granularity $g^t = \frac{1}{270}$, and the scaling factor $\gamma$ = 0.5 (that is, the search space and granularity is shrunk by half), the optimizer needs four iterations to reach the required degree of accuracy. Overall, the optimizer takes 488 samples to complete.

### 3.4 Cluster-Level Configuration Optimizer

At the cluster scale, the parameters in the cluster-level configurations have a large domain due to the large number of groups. Directly applying the enumeration technique may take prohibitively long time because the search time in Eqn. (3) *increases exponentially* with the group number. We therefore transform this optimization problem into an equivalent dynamic programming (DP) problem by: (i) defining the value of an group's resource allocation; (ii) characterizing the optimal substructure of the DP problem; and (iii) developing an algorithm that computes the value of the optimal resource allocation for all $m$ groups and searches for their best cluster-level configurations from the computed information.

*Value of Resource Allocation*. The allocated resource to a group $o$ determines the latency of its jobs and this allocation is restricted by lower and upper bounds in practice. First, given a mix of jobs, the *lower bound* $r^{Lower}$ denotes the minimum amount of resources to support their executions. For example, the YARN resource manager needs to launch an application master (using one container) for each job before allocating other containers to execute its tasks. Second, the *upper bound* $r^{Upper}$ denotes the maximum amount of resources group $o$ needs to run its jobs. That is, all jobs satisfy their resource demands under $r^{Upper}$ and the allocation of extra resources does not further decrease the job latency. Within this context, we define the value $L^R(o,r)$ as the *latency reduction* through allocating *extra* resource $r$ to group $o$ in addition to its lower bound $r^{Lower}$

$$L^R(o,r) = L^*(i, r^{Lower}) - L^*(i, r^{Lower} + r). \tag{4}$$

Note that in Eqn. (4), the calculation is based on the assumption that the best group-level configuration is applied in group $o$.

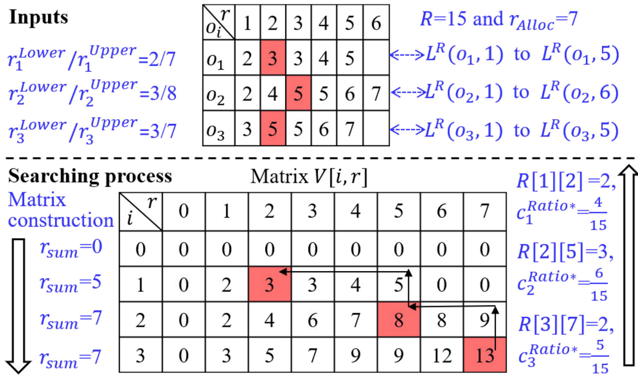
*Optimal Substructure*. Given $i$ groups $(1 \leq i \leq m)$ in the cluster, we define the optimal substructure as the maximum latency reduction $V[i,r]$ when allocating *extra* resource to these groups in addition to their low bounds

$$V[i,r] = \max\{V[i-1,r], L^R(o_i, r_i^*) + V[i-1, r-r_i^*]\}. \tag{5}$$

In Eqn. (5), the calculation of $V[i,r]$ has the *overlapping subproblems* property: it is based on the optimal substructure of ($i$-1) groups and $V[i-1,r]$ is revisited over and over again during the calculation of $V[i,r]$ ($V[0,r]=0$). Hence this calculation is a binary choice: $V[i,r]$ either equals to $V[i-1,r]$, indicating that no resource is allocated to the $i$th group; or $V[i,r]$ equals to $(L^R(o_i, r_i^*) + V[i-1, r-r_i^*])$ if this value is larger than $V[i-1,r]$, indicating that allocating resource $r_i^*$ to the $i$th group and $(r - r_i^*)$ to the other $(i-1)$ groups results in a larger latency reduction. In the second choice, $r_i^*$ = $\arg\max_{1 \leq r_i \leq r}\{L^R(o_i, r_i)$+ $V[i-1, r-r_i]\}$ denotes the optimal allocation of *extra* resources to the $i$th group.

*Algorithm* 3 details the steps of searching for the best cluster-level configuration. Given the extra resource $r_{Alloc}$ in addition to all $m$ groups' lower bounds (line 1), the algorithm first computes the maximum latency reduction $V[i,r]$ in a tabular, bottom-up manner (lines 2 to 12). Note that for $i$ groups, the summarized allocatable resources $r_{sum}$ is calculated based on these groups' lower and upper bounds, and the algorithm only considers resource allocation within $r_{sum}$ to reduce the search space (lines 6 to 11). Matrix $R[i,r]$ records the optimal solution given $i$ and $r$, and it is used to compute the best cluster-level configurations (lines 15 to 18).

**Proposition 3.2.** *The time complexity of Algorithm 3 is $O(m \times r_{sum}^2)$, where $m$ is the number of groups and $r_{sum}$ is the number of allocatable containers.*




| $o_i$ \ r | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $o_1$ | 2 | 3 | 3 | 4 | 5 | |
| $o_2$ | 2 | 4 | 5 | 5 | 6 | 7 |
| $o_3$ | 3 | 5 | 5 | 6 | 7 | |

| i \ r | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 2 | 3 | 3 | 4 | 5 | 0 | 0 |
| 2 | 0 | 2 | 4 | 6 | 7 | 8 | 8 | 9 |
| 3 | 0 | 3 | 5 | 7 | 9 | 9 | 12 | 13 |

Fig. 8. An example of searching the best cluster-level configurations.

**Proof.** The algorithm takes $m$ loops to complete the search of all $m$ groups to find the best solution. In each loop (lines 5 to 12), it takes $r_{sum}$ iterations (lines 7 and 11) to search group $i$'s best resource allocation. At iteration $r$ ($1 \leq r \leq r_{sum}$), it needs $r$ operations to find the best solution (line 8) and other operations in the iteration can be done in constant time (lines 9 and 10). Hence the algorithm takes $\sum_{r=1}^{r_{sum}} r = \frac{r_{sum}\times(r_{sum}+1)}{2} = O(r_{sum}^2)$ to complete the search of one group. The total time complexity of searching $m$ groups, therefore, is $O(m \times r_{sum}^2)$. □

**Algorithm 3.** Searching the Best Cluster-Level Configuration

**Require:** $r^{Lower}/r^{Upper}$: the lower/upper bound of resource allocation to a group;
$V[i,r]$: the maximum latency reduction of the first $i$ groups when the extra resource allocation is $r$;
$R[i,r]$: the optimal allocation of extra resource to the $i$th group when the extra resource allocation is $r$.
1. $r_{Alloc} = R - \sum_{i=1}^{m} r_i^{Lower}$;
2. Set $V[i,r] = 0$ for $0 \leq i \leq m$ and $0 \leq r \leq r_{Alloc}$;
3. Set $R[i,r] = 0$ for $0 \leq i \leq m$ and $0 \leq r \leq r_{Alloc}$;
4. $r_{sum} = 0$;
5. **for** ($i = 1; i \leq m; i$++) **do**
6. $r_{sum} = \min\{r_{sum} + r_i^{Upper} - r_i^{Lower}, r_{Alloc}\}$;
7. **for** ($r = 1; r \leq r_{sum}; r$++) **do**
8. $r_i^* = \arg\max_{1\leq r_i \leq r}\{L^R(o_i, r_i) + V[i-1, r-r_i]\}$;
9. $R[i,r] = r_i^*$;
10. $V[i,r] = \max\{V[i-1,r], L^R(o_i, r_i^*) + V[i-1, r-r_i^*]\}$;
11. **end for**
12. **end for**
13. $i = m$;
14. $r = r_{sum}$;
15. **for** ($i = m; i > 0; i = i$-1) **do**
16. $c_i^{Ratio*} = (R[i,r] + r_i^{Lower})/R$;
17. $r = r$-$R[i,r]$;
18. **end for**
19. Return $\{c_1^{Ratio*}, \ldots, c_m^{Ratio*}\}$.

*Example*. Fig. 8 shows an example of searching three groups' best cluster-level configurations using Algorithm 3. In the inputs, $r$ represents the extra resources allocated to a group in addition to its lower bound and $L^R(o_i, r)$ represents the value (reduced latency) when allocating $r$ to group $o_i$. The total available resource $R$ = 15 in cluster and thus the total extra resource to all groups is: $r_{Alloc} = R - \sum_{i=1}^{3} r_i^{Lower}$= 7. In the search process, the algorithm first constructs matrix $V[i,r]$ ($0 \leq i \leq 3$, $0 \leq r \leq 7$) by sequentially computing the maximum latency reductions when allocating resources to three groups, and $r_{sum}$ represents the amount of allocatable resources for these groups. Subsequently, the algorithm searches the three groups' best configurations by starting from the bottom-right of the matrix $V[i,r]$. Using three iterations, it stepwise calculates the optimal extra resource allocation to each group according to matrix $R[i,r]$. We can see in the best solution, group 2 is allocated the smallest ratio of resources because it brings the smallest reduction of job latency.

## 4 Evaluation

Based on the implementation of AdaptiveConfig on the YARN Capacity and Fair schedulers, our evaluation has three objectives. First, we show AdaptiveConfig is able to select best (next-best) group-level configurations for scenarios with various mixes of jobs and YARN scheduler equipped with different queues and resources (containers) (Section 4.3). Second, we present the runtime results, further highlighting the effectiveness of AdaptiveConfig in adapting to time-varying workloads (Section 4.4). The first two evaluations are conducted on real testbeds. Finally, using real-trace driven simulations, we evaluate the effectiveness of our approach in improving job performance when dealing with large clusters (Section 4.5).

### 4.1 Implementation on YARN Schedulers

AdaptiveConfig is implemented in Java and it is currently targeted for cloud jobs running in the YARN platform. Its *group-level configuration optimizer* is implemented based on open source Drools rule engine (Section 3.3.2), and it is incorporated with two typical YARN schedulers. Both ones have dozens of configurable parameters, in which *a few ones* determine the allocation of resources in job scheduling:

- Capacity scheduler [10] is designed to share resource among users in order to maximize the resource utilization. Users are allocated to different queues according to their priorities. In job scheduling, each queue is guaranteed a *capacity* of resources for its jobs, while can also access idle resources from other queues under the constraint of *maximal capacity*.
- Fair scheduler [11] also organizes users into different queues and uses *weights* to determine the fractions of resources used by different queues. For each queue, this scheduler can be configured to use one of the three *scheduling policies*: FIFO, Fair, and DRF [30].

AdaptiveConfig interacts with YARN to obtain the workload and cluster status: (i) it reads the "/history/done_intermediate" file in YARN's job history server to obtain each waiting or running job's information, including its submission time, resource requirement, and task information; (ii) it reads the "yarn-site.xml" file to get the cluster resource status, including the amount of available CPU cores and memory, and the resource granularity of a single container; (iii) after selecting the best configuration, it accesses

the "capacity-scheduler.xml" or "fair-scheduler.xml" file to re-configure the scheduler's parameters.

To make the reactive configuration tuning applicable for large clusters with massive nodes and jobs, one *group-level optimizer* is implemented for each user group with two objectives. First, an optimizer only needs to read the job and node information from one group, thus the overheads from the reading cause slight interruption to the system performance. Second, multiple optimizers run in parallel and the stall of reading information in one optimizer only influences the tuning of group-level configuration in its own group. In addition, the cluster-level optimizer can still use the previous job information in this group to search the globally best configuration. This optimizer invokes the reconfiguration of the scheduler periodically to update the scheduler's configuration file. The reconfiguration takes effect within a few seconds on YARN [59].

### 4.2 Experiment Settings

*Experiment Platform*. The testbed is a cluster of 20 nodes, each node is equipped two 6-core Intel Xeon E5645 processors and 32 GB of DRAM, and the operating system is Linux CentOS 7 3.10.0. In the YARN distribution, the versions of Hadoop and Spark are 2.7.2 and 2.0.2. The versions of JDK and Python versions are 1.7.0 and 2.7.5. On a YARN scheduler, the resource is allocated at the granularity of containers, each one has 1 CPU core and 2 GB memory. The simulation platform is YARN SLS [12].

*Workloads*. The Facebook and Google jobs are generated by the SWIM [9] and CloudMix [2] benchmarks according to the publicly available Facebook trace [21] and Google traces [52], respectively. The workload characteristics of Facebook MapReduce (Spark) jobs include submission times, job types, and input data sizes, and these jobs run on the YARN cluster. The workload characteristics of Google jobs include submission times and task resource usages (CPU and memory), and these jobs run on both YARN and YARN SLS.

*Scheduling Scenarios*. The scenario of Google workload is established according to the Google cluster trace [52], which records complete information of machine, job, and tasks. The cluster has 12.5k nodes of three heterogeneous platforms (machine types) and 10 resource capacities. The trace spans 29 days and includes about 1k user, 40k applications, 650k jobs, and 144 m tasks. Our scenario consists of four elements: (1) *Job priority*. Jobs of priorities 2 to 8 (best-effort scheduling class) are used. (2) *User group*. We category users into different groups according to the applications (denoted by the logicaljobname in the trace) they run. (3) *Available resource*. A cluster's available resource to best-effort jobs is calculated by summarizing the actual nodes assigned to these jobs. (4) *Scheduling constraints*. We consider two types of constraints: a priority queue's capacity constraint is calculated according to the maximum memory usage of its jobs; a job's task placement constraint is directly derived from the "Task constraints" table, which restricts the machines its tasks can run.

The Facebook trace [21] lacks information of the above scheduling scenario, so we only test the Facebook workload in single-group cases, randomly assign priorities (2 to 8) to jobs, set cluster sizes and queue capacity constraints: the

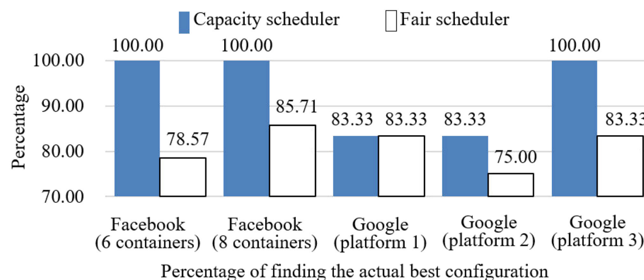



Fig. 9. The percentage of finding actual best configurations.

minimal and maximal amount of resources in each queue is 10 and 90 percent of the resource capacity.

*Metric*. When scheduling a mix of jobs on a cluster, the *job performance* is measured by the average job latency.

### 4.3 Search of the Best Group-Level Configuration

The effectiveness of AdaptiveConfig is considerably impacted by its ability to search the best group-level configurations under diverse scheduling scenarios. We define the *search effectiveness* metric as the *percentage* of increase in the average latency of mixed jobs (that is, their performance degradation), when comparing the *evaluated* configuration against the *actual best* configuration.

*Evaluation Settings*. We test different cases of job scheduling from three aspects: (1) Four *scheduler settings*, including two schedulers (Capacity and Fair) and two settings of priority queue (2 or 4) for either scheduler. In the scheduler of two queues, jobs of priorities 2$\sim$5 and 6$\sim$8 are submitted to the two queues, respectively. In the case of four queues, jobs are respectively submitted to these queues according to their priorities: 2$\sim$3, 4$\sim$5, 6$\sim$7, and 8. (2) 32 *mixes of jobs*, including 14 mixes of Facebook jobs generated for two cluster capacities (six and eight containers); and 18 mixes of Google jobs derived from three heterogeneous platforms and each platform has six periods covering 24 hours a day (each period denotes the workload of four hours); (3) 56 *group-level configurations*. Following the configuration parameter setting in Section 2.3, the Capacity and Fair schedulers of two queues have five and 15 configurations, respectively. When the number of queues increases to 4, we halve the values of the "capacity" and "weight" parameters in the previous setting and double their combinations in four queues. Hence the Capacity and Fair schedulers have nine and 27 configurations, respectively. Overall, 7,168 scheduling cases are tested.

*Evaluation Results*. Fig. 9 shows the percentages of finding the actual best configurations using AdaptiveConfig. We can see that for the Capacity and Fair schedulers respectively, the best configurations are found in 93.33 and 81.19 percent of the test cases. When considering the Facebook and Google jobs respectively, 91.07 and 84.72 percent of the best configurations are found. The results indicate that our approach can successfully compares the performance discrepancy of different configurations and identify the best ones in a majority of cases. This percentage is lower in the Fair scheduler because it has larger parameter space and hence the performance discrepancies among different configurations are smaller. Fig. 10 further shows in the remaining cases, AdaptiveConfig still finds the next-best configurations with an

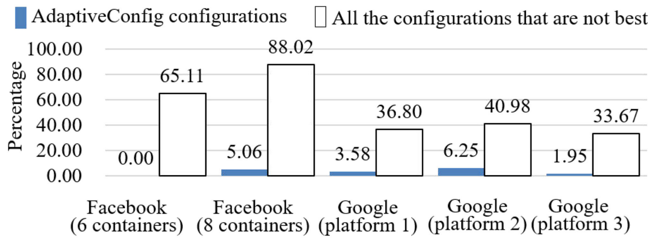



Fig. 10. Comparison of latency increase (%) in Adaptive's next-best configurations and all the configurations that are not best.

TABLE 2
18 Representative Scheduling Configurations in the Fair Scheduler of Four Priority Queues

| Configuration | $c_1$ to $c_3$ | $c_4$ to $c_6$ | $c_7$ to $c_9$ | $c_{10}$ to $c_{12}$ | $c_{13}$ to $c_{15}$ | $c_{16}$ to $c_{18}$ |
|---|---|---|---|---|---|---|
| $q_1$'s weight | 25 | 50 | 75 | 25 | 50 | 75 |
| $q_2$'s weight | 75 | 50 | 25 | 75 | 50 | 25 |
| $q_3$'s weight | 25 | 50 | 75 | 25 | 50 | 75 |
| $q_4$'s weight | 75 | 50 | 25 | 75 | 50 | 25 |
| Job scheduling policy | Configurations $c_i$, $c_{i+1}$, $c_{i+2}$ ($i$=1, 4, 7, 10, 13, 16) correspond to FIFO, Fair, and DRF job scheduling algorithms within queues, respectively | | | | | |

average of 3.37 percent increases in job latency compared to the actual best ones. In contrast, job latency is increased by 52.92 percent (15.71 times larger than our approach) when considering all the configurations that are not best.

### 4.4 Run-Time Configuration Adaptation

This section demonstrates the effectiveness of AdaptiveConfig in dynamically tuning configurations to the time-varying workloads within a user group. Following the scheduler settings of two and four queues the previous section, we test the Capacity and Fair schedulers using the Facebook workload.

*Comparison Settings*. To the best of our knowledge, AdaptiveConfig is the first system that dynamically re-configures the cluster scheduler according to the workload changes. Hence we compare against baselines with *representative configurations that lead to differential resource allocations in the configuration space*. Specifically, in the case of two priority queues, the Capacity scheduler has seven configurations, where configuration $c_i$ ($1 \leq i \leq 7$) assigns $i$ and (8-$i$) containers to the two queues, respectively and the FIFO scheduling policy is applied in both queues. The Fair scheduler has nine configurations, where either queue has three choices (25, 50, 75) of the "weight" parameter and three optional algorithms (FIFO, Fair and DRF) of the "job scheduling algorithm" parameter. In the case of four priority queues, each queue of the Capacity scheduler can reserve a capacity of one, three, or five containers, and there are 10 configurations when considering the different capacity combinations of these queues. The Fair scheduler has 18 representative configurations as listed in Table 2. In comparison, AdaptiveConfig's search space of configurations is set in a similar way: in the Capacity scheduler, the domain of the "capacity" parameter is equally discretized 7 values in each queue; in the Fair scheduler, the domain of the "weight" parameter contains three values (25, 50 and 75) and the domain of the "job scheduling algorithm" parameter also has three values: FIFO, Fair and DRF.

*Evaluation Result*. Fig. 11 shows the job latencies between the representative configurations and the dynamic AdaptiveConfig choices during a period of 20 minutes, and we report the average job latency of the mix of jobs every 2 minutes. We can see that AdaptiveConfig consistently provides lower latencies because it performs online search of the best or next-best configuration for the current mix of jobs. Along the testing time, the average job latencies of all configurations increase because jobs' queueing times become longer and longer. AdaptiveConfig suffers less from such queueing delays because it provides the lowest job latencies (that is, the best configurations) in most of the cases, thereby incurring much shorter job waiting times. In contrast, a static configuration maybe the best configuration for one interval, but causes much longer delays than those of the other configurations in the next interval, thus seriously delaying the subsequent jobs.

We further extend the above evaluation by testing four different configuration tuning intervals (1, 2, 4, and 8 minutes) in AdaptiveConfig. A shorter interval mean a finer granularity of adapting configurations to the waiting jobs in the system. The results in Fig. 12 show that in both schedulers, the 8-minute interval results in the highest job latency because this configuration tuning cannot provide timely responses to the quickly changing workload. In addition, we can observe that the 1-minute interval leads to the lowest job latency and the 2-minute interval leads to a similar latency, which indicates that 2-minute tuning interval can provide sufficiently quick responses to workload changes.

*Results. When considering different scheduling settings, AdaptiveConfig reduces Facebook job latencies by an average of 2.16 times compared to the representative settings in the configuration space, and the latency reductions are 2.22 times and 2.40 times when the tuning intervals are 2 and 1 minutes, respectively.*

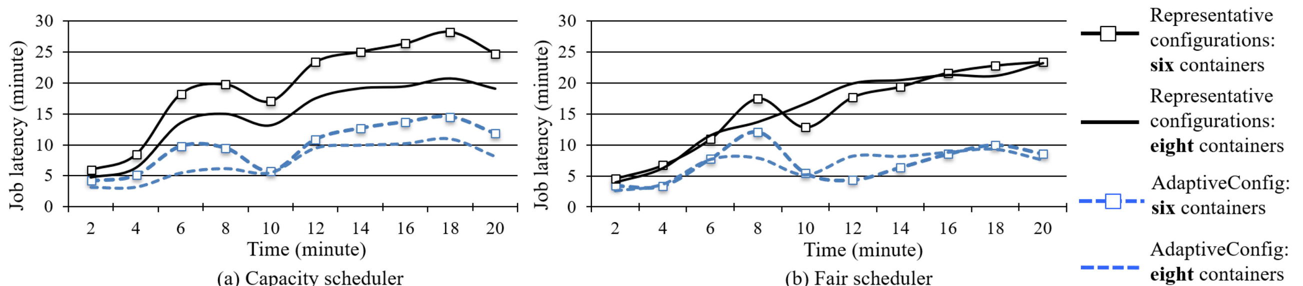



Fig. 11. Time-varying workload mixes: Comparing the average job latency of AdaptiveConfig and the representative configurations on a user group with six/eight containers.

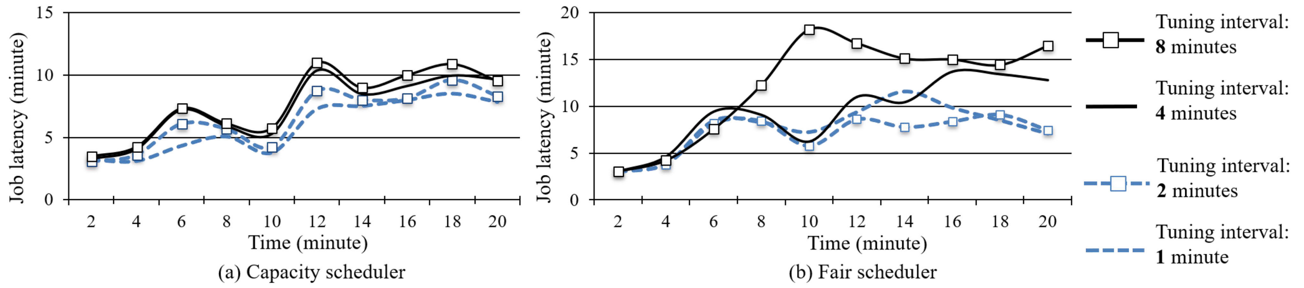



Fig. 12. Time-varying workload mixes: Comparing the average job latency of AdaptiveConfig and the representative configurations under different tuning intervals.

### 4.5 Search the Best Cluster-Scale Configurations

This section evaluates the overheads and effectiveness of AdaptiveConfig in searching large clusters to find the globally best configuration.

#### 4.5.1 *Overheads of Searching Best Configurations*

This section's evaluation tests AdaptiveConfig's search overheads consisting of two parts: (1) in each group-level optimizer, the major overhead comes from searching the group's configuration parameter space using the RRS approach, which triggers the Drools rule engine to compare the performance of different configurations; (2) based on the search results of all group-level optimizers, the overhead of the cluster-level optimizer comes from searching the whole cluster's large configuration parameter space using the DP approach.

*Evaluation Settings*. Following the scheduler setting and the group-level configurations of Section 4.3, we evaluated both the Capacity and the Fair schedulers of two and four queues. In the RSS approach of group-level optimizers, the confidence probability $p = 0.99$, the initial search granularity is 10 times larger than the required granularity, and the scaling factor $\gamma = 0.5$. In the DP approach of the cluster-level optimizer, we tested 10 cluster sizes ranging from 300 nodes to 12k nodes (each node has four containers), and the corresponding group numbers range from 10 to 500. We deployed each group in a separate virtual machine (VM) of 2 CPU cores and 1 GB memory, and the scheduling of jobs in the group was conducted/simulated on a YARN SLS. Under this evaluation setting, the scalability of the approach is tested in an actual cluster environment of up to 500 VMs. Each cluster size was tested 10 times and we report the average.

*Evaluation Results*. Fig. 13 shows that the search time gradually increases with the search scope and it is still less than 4 seconds when the cluster size reaches 12k nodes and the group number reaches 500. This time is 30 times shorter than the typical tuning interval (2 minutes) applied in our approach. This is because at the group level, both schedulers have a small number of configuration parameters and hence the RSS approach can complete within hundreds of samples. Specifically, the Capacity scheduler of two and four queues needs 83 and 175 iterations to complete respectively, and the Fair scheduler of two and four queues needs 267 and 488 iterations to complete respectively. We can also observe that although the DP approach has a polynomial time complexity $O(m \times r_{sum}^2)$ (Proposition 3.2), its actual time consumption is linearly proportional to the group size $m$. This is because when searching the best configuration in $i$ groups ($1 \leq i \leq m$), the allocatable resource $r_{sum}$ is restricted by these groups lower and upper bounds of allocatable resources, and hence $r_{sum}$ is much smaller than the total amount of resources in the cluster in most of the cases.

Fig. 14 further shows the percentages computation time of the two search parts under different evaluation settings. In AdaptiveConfig, all group-level optimizers execute in parallel and each optimizer's search time is proportional to the configuration space of the scheduler. The Fair scheduler has more configuration parameters and hence it needs higher percentages computation time than the Capacity scheduler. In addition, the percentages computation time of the cluster-level search part are proportional to the cluster

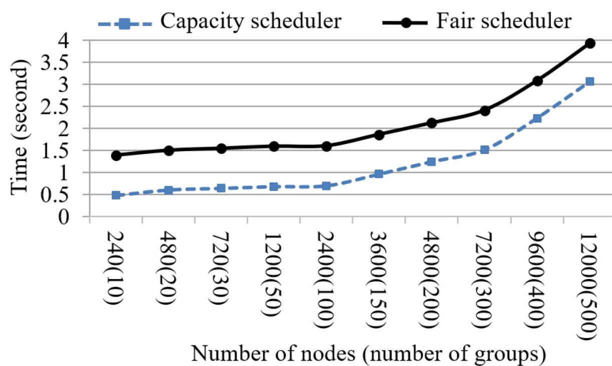



Fig. 13. Scalability of the search approaches in AdaptiveConfig.

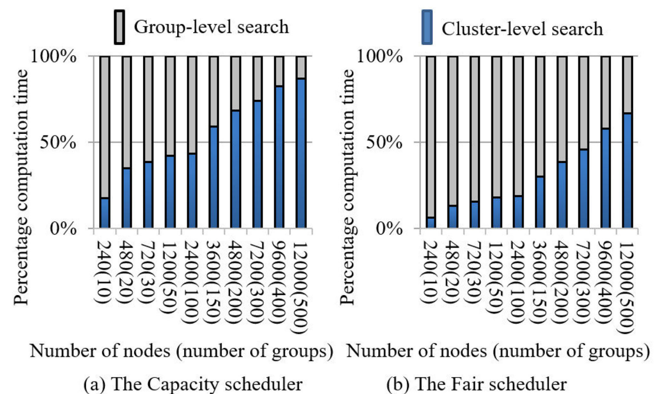



Fig. 14. Percentage computation time breakdown for group-level and cluster-level optimizers.

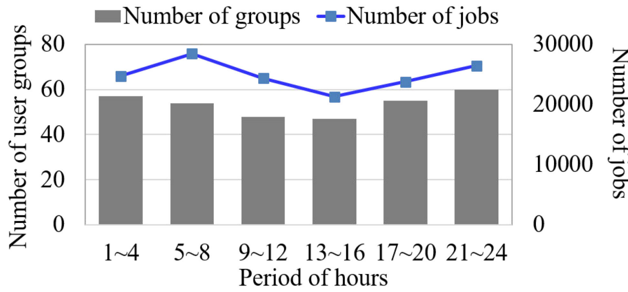



Fig. 15. Characteristics of user groups and submitted job in the six periods of Google trace.

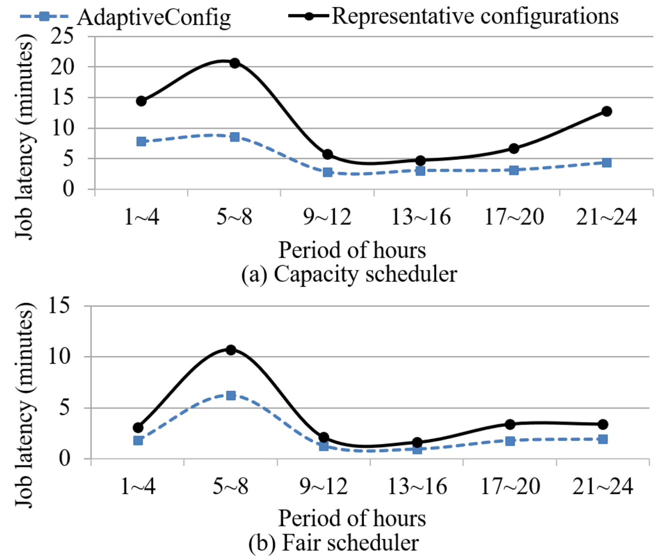



Fig. 16. Comparison of average job latency against the static configurations across six periods of the Google cluster.

size, and they range from 17.69 to 87.08 percent and 6.22 to 66.84 percent in the Capacity scheduler and the Fair scheduler, respectively. That is, our approach spends more time in searching the cluster-level best configuration as the cluster size increases.

#### *4.5.2 Comparison of Job Performance*

Next, we extend the comparison evaluation in the previous section to multiple groups in order to testing both group-level and cluster-level configurations on a 12K-node SLS cluster (a typical Google cluster size). In evaluation, the group-level configurations follow the previous comparison settings, and the cluster-level configurations, namely the ratios of resources assigned to different groups, are derived from the assigned resources to different groups in the Google trace. In addition, we test the Google jobs whose submission times span six periods of 24 hours. Fig. 15 shows the numbers of user group and submitted jobs at these periods. We can see a larger group number usually means more jobs being submitted. The period of hours 5~8 has the largest job number because the users during this period have the highest submission rate.

The comparison results in Fig. 16 shows that: (i) AdaptiveConfig consistently provides lower latencies. (ii) When the job number increases (e.g., in hours 5~8), the job latencies have apparent increases under the representative configurations, because the resources are saturated. In contrast, the latency increases in AdaptiveConfig are much smaller, which verify that our approach displays more advantages under tenser loads. (iii) When only considering group-level configurations, the average percentage of latency increase in other configurations is about 37 percent when compared to the best ones, as shown in previous evaluation (three Google platforms in Fig. 10). When considering both group-level and cluster-level configurations in this evaluation, the latency increase becomes 93.78 percent.

*Results. When testing different platforms and periods of Google jobs in a 12K-node cluster, AdaptiveConfig reduce job latencies by an average of 1.94 times compared to the representative configurations*

## 5 Conclusion

In this paper, we presented AdaptiveConfig, a run-time configuration tuning framework for cluster schedulers in the cloud. AdaptiveConfig supports diverse scheduling scenarios by comprehensively handling different factors (e.g., schedulers, workload and resource status, and scheduling constraints) based on the Drools rule engine, thus effectively comparing the performance discrepancies of different configurations. It then proposes a DP-based approach to efficiently search the best configuration adapting to the changing workload in large clusters. Our approach is implemented on two representative YARN schedulers (Capacity and Fair) and its effectiveness in significantly reducing job latencies is demonstrated using workloads in real applications.

We are currently investigating the development of a more general framework to support workload-adaptive configuration tuning for a wider class of scheduling scenarios and platforms. Developing such a framework requires investigating the different dimensions that affect job scheduling: including *the available resources*, *the used configuration optimizers*, and *the underlying resource management platforms*. Specifically, the available resources in a large cluster depends the failure probability of its nodes. The framework can mitigate the resource variance (due to failures) by means of considering such probabilities in job latency estimation and preferentially allocating newly-available resources against the re-scheduled jobs on failure nodes. In addition, this works optimizer was built upon the Drools rule engine, which requires explicitly defining a schedulers resource allocation mechanism as business rules. Our future work will leverage the emerging deep reinforcement learning (DRL) techniques to learn this resource allocation mechanism and use deep neural networks to describe the linking relationship between the scheduler's configuration and a variety of dynamics (workloads, resources and constraints) in scheduling.

In the general framework under development, we are investigating supporting reactive configuration tuning for other types of resource management platforms, including Mesos whose cluster-level scheduler shares resources among different computing frameworks and its scheduler configuration decides the orders of pushing resources to these frameworks. In order to apply our cluster-level configuration optimizer on Mesos, we need to re-define the optimal substructure of the DP problem, in which the optimal

substructure $V[j,i]$ is the lowest job latency of the first $j$ frameworks when the resources are pushed to framework $i$ in the $j$th iteration.

## Acknowledgments

We thank the anonymous reviewers for careful review of our paper. This work is supported by the National Key Research and Development Plan of China (Grant No. 2018YFB1003701 and 2018YFB1003700) and the National Natural Science Foundation of China (Grant No. 61872337).

## References

[1] Apache mesos. 2018. [Online]. Available: http://mesos.apache.org/
[2] Cloudmix, a multi-tenancy version of BigDataBench. 2017. [Online]. Available: http://prof.ict.ac.cn/multi-tenancy.html
[3] Docker swarm. 2019. [Online]. Available: https://docs.docker.com/engine/swarm/
[4] Drools expert business rules engine. 2019. [Online]. Available: https://www.drools.org/
[5] Facebook tupperware. 2014. [Online]. Available: http://www.slideshare.net/dotCloud/tupperware-containerized-deployment-at-facebook
[6] Google kubernetes. 2019. [Online]. Available: http://kubernetes.io
[7] Hadoop yarn. 2018. [Online]. Available: https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/YARN.html
[8] Mesos, omega, borg: A survey. 2015. [Online]. Available: http://umbrant.com/blog/2015/mesos_omega_borg_survey.html
[9] Statistical workload injector for MapReduce (swim). 2019. [Online]. Available: https://github.com/SWIMProjectUCB/SWIM/wiki
[10] Yarn capacity scheduler. 2016. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html
[11] Yarn fair scheduler. 2018. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html
[12] Yarn SLS. 2018. [Online]. Available: https://hadoop.apache.org/docs/r3.0.0/hadoop-sls/SchedulerLoadSimulator.html
[13] S. R. Alapati, *Expert Hadoop Administration: Managing, Tuning, and Securing Spark, YARN, and HDFS*. Reading, MA, USA: Addison-Wesley, 2016.
[14] Z. Bei, Z. Yu, H. Zhang, W. Xiong, C. Xu, L. Eeckhout, and S. Feng, "RFHOC: A random-forest approach to auto-tuning hadoop's configuration," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1470–1483, May 2016.
[15] A. A. Bhattacharya, D. Culler, et al., "Hierarchical scheduling for diverse datacenter workloads," in *Proc. ACM Symp. Cloud Comput.*, 2013, Art. no. 4.
[16] E. Boutin, J. Ekanayake, et al., "Apollo: Scalable and coordinated scheduling for cloud-scale computing," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 285–300.
[17] F. Caglar, S. Shashank, and G. Aniruddha, "iTune: Engineering the performance of Xen hypervisor via autonomous and dynamic scheduler reconfiguration," *IEEE Trans. Serv. Comput.*, vol. 11, no. 1, pp. 103–116, Jan./Feb. 2018.
[18] C. X. Cai, S. Saeed, I. Gupta, et al., "Phurti: Application and network-aware flow scheduling for multi-tenant MapReduce clusters," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2016, pp. 161–170.
[19] L. Chen, S. Liu, B. Li, and B. Li, "Scheduling jobs across geo-distributed datacenters with max-min fairness," in *Proc. IEEE INFOCOM*, 2017, pp. 46–54.
[20] W. Chen, R. Jia, and Z. Xiaobo, "Preemptive, low latency datacenter scheduling via lightweight virtualization," in *Proc. USENIX Annu. Techn. Conf.*, 2017, pp. 251–263.
[21] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of MapReduce workloads," in *Proc. VLDB Endowment*, vol. 5, pp. 1802–1813, 2012.
[22] L. Chengzhi, Y. Kejiang, X. Guoyao, X. Cheng-Zhong, and B. Tongxin, "Imbalance in the cloud: An analysis on Alibaba cluster trace," in *Proc. IEEE Int. Conf. Big Data*, 2017, pp. 2884–2892.
[23] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, "Reservation-based scheduling: If you're late don't blame us!" in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–14.
[24] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel, "Job-aware scheduling in eagle: Divide and stick to your probes," in *Proc. ACM Symp. Cloud Comput.*, 2016, pp. 497–509.
[25] P. Delgado, F. Dinu, A. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling," in *Proc. USENIX Annu. Techn. Conf.*, 2015, pp. 499–510.
[26] C. Delimitrou and K. Christos, "Quasar: Resource-efficient and QoS-aware cluster management," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2014, pp. 127–144.
[27] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2013, pp. 77–88.
[28] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed job latency in data parallel clusters," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2012, pp. 99–112.
[29] H. Gao, Y. Zhengyu, B. Janki, W. Teng, W. Jiayin, S. Bo, and M. Ningfang, "AutoPath: Harnessing parallel execution paths for efficient resource allocation in multi-stage big data frameworks," in *Proc. Int. Conf. Comput. Commun. Netw.*, 2017, pp. 1–9.
[30] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 323–336.
[31] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Choosy: Max-min fair sharing for datacenter jobs with constraints," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2013, pp. 365–378.
[32] I. Gog, S. Malte, G. Adam, N. M. Watson Robert, and H. Steven, "Firmament: Fast, centralized cluster scheduling at scale," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 99–115.
[33] A. Gounaris, G. Kougka, R. Tous, C. T. Montes, and J. Torres, "Dynamic configuration of partitioning in spark applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 7, pp. 1891–1904, Jul. 2017.
[34] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 455–466, 2014.
[35] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 81–97.
[36] R. Han, Z. Zong, L. Y. Chen, S. Wang, and J. Zhan, "AdaptiveConfig: Run-time configuration of cluster schedulers for cloud short-running jobs," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 1519–1526.
[37] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of MapReduce programs," *Proc. VLDB Endowment*, vol. 4, pp. 1111–1122, 2011.
[38] H. Herodotou, et al., "Starfish: A self-tuning system for big data analytics," in *Proc. Conf. Innovative Database Res.*, 2011, pp. 261–272.
[39] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 295–308.
[40] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss, "Resource elasticity for large-scale machine learning," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 137–152.
[41] Z. Huang, B. Balasubramanian, M. Wang, et al., "Need for speed: CORA scheduler for optimizing completion-times in the cloud," in *Proc. IEEE INFOCOM*, 2015, pp. 891–899.
[42] M. Isard, "Autopilot: Automatic data center management," *Operating Syst. Rev.*, vol. 41, no. 2, pp. 60–67, 2007.
[43] D. B. Jackson, Q. Snell, and M. J. Clement, "Core algorithms of the maui scheduler," in *Proc. Workshop Job Scheduling Strategies Parallel Process.*, 2001, pp. 87–102.
[44] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, et al., "Morpheus: Towards automated SLOs for enterprise clusters," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 117–134.
[45] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga, "Mercury: Hybrid centralized and distributed scheduling in large shared clusters," in *Proc. USENIX Annu. Techn. Conf.*, 2015, pp. 485–497.
[46] M. Khan, Y. Jin, M. Li, Y. Xiang, and C. Jiang, "Hadoop performance modeling for job estimation and resource provisioning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 2, pp. 441–454, Feb. 2016.

[47] N. Lim, S. Majumdar, and P. Ashwood-Smith, "MRCP-RM: A technique for resource allocation and scheduling of MapReduce jobs with deadlines," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 5, pp. 1375–1389, May 2017.

[48] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, 2014, Art. no. 2.

[49] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, low latency scheduling," in *Proc. ACM Symp. Operating Syst. Principles*, 2013, pp. 69–84.

[50] J. W. Park, A. Tumanov, et al., "3Sigma: Distribution-based cluster scheduling for runtime uncertainty," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2018, Art. no. 2.

[51] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao, "Efficient queue management for cluster scheduling," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2016, Art. no. 36.

[52] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. ACM Symp. Cloud Comput.*, 2012, pp. 7–19.

[53] R. Salkhordeh, S. Ebrahimi, and H. Asadi, "ReCA: An efficient reconfigurable cache architecture for storage systems with online workload characterization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 7, pp. 1605–1620, Jul. 2018.

[54] M. Schwarzkopf, "Cluster scheduling for data centers," *ACM Queue*, vol. 15, no. 5, pp. 1–12, 2017.

[55] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2013, pp. 351–364.

[56] J. Shi, J. Zou, J. Lu, Z. Cao, S. Li, and C. Wang, "MRTuner: A toolkit to enable holistic optimization for MapReduce jobs," *Proc. VLDB Endowment*, vol. 7, pp. 1319–1330, 2014.

[57] P. Thinakaran, R. G. Jashwant, S. Bikash, T. K. Mahmut, and R. D. Chita, "Phoenix: A constraint-aware scheduler for heterogeneous datacenters," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 977–987.

[58] A. Tumanov, J. Cipar, G. R. Ganger, and M. A. Kozuch, "Alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds," in *Proc. ACM Symp. Cloud Comput.*, 2012, Art. no. 25.

[59] V. K. Vavilapalli, A. C. Murthy, et al., "Apache hadoop YARN: Yet another resource negotiator," in *Proc. ACM Symp. Cloud Comput.*, 2013, Art. no. 5.

[60] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2015, Art. no. 18.

[61] J. Wang, Y. Yao, Y. Mao, B. Sheng, and N. Mi, "FRESH: Fair and efficient slot configuration and scheduling for hadoop clusters," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2014, pp. 761–768.

[62] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang, "MapTask scheduling in MapReduce with data locality: Throughput and heavy-traffic optimality," *IEEE/ACM Trans. Netw.*, vol. 24, no. 1, pp. 190–203, Feb. 2016.

[63] S. Wu, L. Zhou, H. Sun, H. Jin, and X. Shi, "Poris: A scheduler for parallel soft real-time applications in virtualized environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 3, pp. 841–854, Mar. 2016.

[64] T. Ye and S. Kalyanaraman, "A recursive random search algorithm for large-scale network parameter configuration," *Meas. Model. Comput. Syst.*, vol. 31, no. 1, pp. 196–205, 2003.

[65] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.

[66] M. Zaharia, et al., "Job scheduling for multi-user MapReduce clusters," UC Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2009–55, 2009.

[67] Z. Zhou, X. Yang, Z. Lan, P. Rich, W. Tang, V. Morozov, and N. Desai, "Improving batch scheduling on blue Gene/Q by relaxing network allocation constraints," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 11, pp. 3269–3282, Nov. 2016.

[68] Y. Zhu, L. Jianxun, G. Mengying, B. Yungang, M. Wenlong, L. Zhuoyue, S. Kunpeng, and Y. Yingchun, "BestConfig: Tapping the performance potential of systems via automatic configuration tuning," in *Proc. ACM Symp. Cloud Comput.*, 2017, pp. 338–350.

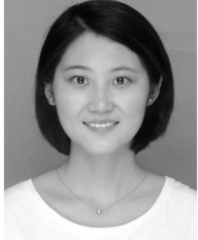
**Rui Han** received the MSc (honor) degree from Tsinghua University, China, in 2010, and the PhD degree from the Department of Computing, Imperial College London, United Kingdom, in 2014. He is an associate professor with the School of Computer Science and Technology, Beijing Institute of Technology, China. His research interests include cloud resource management and system optimization for data center workloads. He has more than 40 publications in these areas, including papers at the *IEEE Transactions on Parallel and Distributed Systems*, INFOCOM, ICDCS, ICPP, CCGrid, and CLOUD.

**Chi Harold Liu** is a full professor and vice dean with the School of Computer Science and Technology, Beijing Institute of Technology, China. He is also the director of IBM Mainframe Excellence Center (Beijing), director of IBM Big Data Technology Center, and director of the National Laboratory of Data Intelligence for China Light Industry. Before moving to academia, he joined IBM Research—China as a staff researcher and project manager, after working as a postdoctoral researcher with Deutsche Telekom Laboratories, Germany, and a visiting scholar with IBM T. J. Watson Research Center. His current research interests include the cloud computing, Internet-of-Things (IoT), mobile crowdsensing, and deep learning. He received the Distinguished Young Scholar Award in 2013, IBM First Plateau Invention Achievement Award in 2012, and IBM First Patent Application Award in 2011 and was interviewed by EEWeb.com as the featured engineer, in 2011. He has published more than 90 prestigious conference and journal papers and owned more than 14 EU/U.S./U.K./China patents, with Google Scholar H index 25. He serves as IEEE ICC'20 symposium chair on Network Generation Networking, area editor of the *KSII Transactions on Internet and Information Systems* and the book editor for six books published by Taylor & Francis Group and China Machinery Press. He also has served as the general chair of IEEE SECON'13 workshop on IoT Networking and Control, IEEE WCNC'12 workshop on IoT Enabling Technologies, and ACM UbiComp'11 Workshop on Networking and Object Memories for IoT. He served as the consultant to Asian Development Bank, Bain & Company, and KPMG, and the peer reviewer for Qatar National Research Foundation, and National Science Foundation, China. He is a senior member of the IEEE.

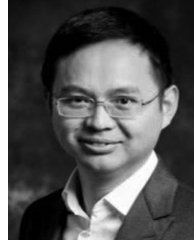
**Zan Zong** is working toward the PhD degree in the School of Software, Tsinghua University. His current research interests include resource management of cluster and cloud systems.

**Lydia Y. Chen** received the BA degree from the National Taiwan University, and the PhD degree from the Pennsylvania State University. She is an associate professor with the Department of Computer Science, Technology University Delft. Prior to joining TU Delft, she was a research staff member with the IBM Zurich Research Lab from 2007 to 2018. Her research interests center around dependability management, resource allocation and privacy enhancement for large scale data processing systems and services. She has published more than 80 papers in journals, e.g., the *IEEE Transactions on Distributed Systems*, the *IEEE Transactions on Service Computing*, and conference proceedings, e.g., INFOCOM, Sigmetrics, DSN, and Eurosys. She was a co-recipient of the best paper awards at CCgrid15 and eEnergy15. She is a senior member of the IEEE.

**Wending Liu** is working toward the undergraduate degree in the School of Computer Science and Technology, Beijing Institute of Technology. His work focuses on resource management in large-scale cloud systems.



**Siyi Wang** is working toward the graduate degree in the Software Systems Lab, Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS). Her current research interests include cloud computing and workload characteristics.



**Jianfeng Zhan** is full professor and director with Software Systems Lab, Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS) and the University of Chinese Academy of Sciences. He enjoys building new systems, and feel great interest in collaborating with researchers with different backgrounds. He recently focuses on different aspects of datacenter computing. He founded the International Symposium on Benchmarking, Measuring, and Optimizing, dedicated to benchmarking, measuring, and optimizing different complex systems. His research work is driven by interesting problems. He enjoys building new systems, and collaborating with researchers with different backgrounds.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.