Article:
Yang, R, Hu, C, Sun, X et al. (2020) Performance-Aware Speculative Resource Oversubscription for Large-Scale Clusters. IEEE Transactions on Parallel and Distributed Systems, 31 (7). pp. 1499-1517. ISSN: 1045-9219

https://doi.org/10.1109/tpds.2020.2970013

Performance-aware Speculative Resource Oversubscription for Large-scale Clusters

Renyu Yang, *Member, IEEE*, Chunming Hu, Xiaoyang Sun, Peter Garraghan, Tianyu Wo, *Member, IEEE*, Zhenyu Wen, Hao Peng, Jie Xu, *Member, IEEE*, Chao Li

Abstract—It is a long-standing challenge to achieve a high degree of resource utilization in cluster scheduling. Resource oversubscription has become a common practice in improving resource utilization and cost reduction. However, current centralized approaches to oversubscription suffer from the issue with resource mismatch and fail to take into account other performance requirements, e.g., tail latency. In this paper we present ROSE, a new resource management platform capable of conducting performance-aware resource oversubscription. ROSE allows latency-sensitive long-running applications (LRAs) to co-exist with computation-intensive batch jobs. Instead of waiting for resource allocation to be confirmed by the centralized scheduler, job managers in ROSE can independently request to launch speculative tasks within specific machines according to their suitability for oversubscription. Node agents of those machines can however avoid any excessive resource oversubscription by means of a mechanism for admission control using multi-resource threshold control and performance-aware resource throttle. Experiments show that in case of mixed co-location of batch jobs and latency-sensitive LRAs, the CPU utilization and the disk utilization can reach 56.34% and 43.49%, respectively, but the 95th percentile of read latency in YCSB workloads only increases by 5.4% against the case of executing the LRAs alone.

Index Terms—resource scheduling, oversubscription, cluster utilization, resource throttling, QoS

1 Introduction

Improving cluster resource utilization for cloud datacenters is of ever-increasing importance towards promoting return on capital of cluster operators and meeting global demand for Internet services such as web search, social networking, and machine learning applications. Modern cluster management systems [1][2][3][4] are designed to effectively allocate applications or jobs onto machines. However, production clusters still encounter issues associated with underutilization [4][5][6]. A primary cause for such low utilization is due to the disparity between requested and actual resource usage of jobs, with studies of production clusters from Twitter and Google demonstrating disparity of 53% and 40% for CPU, and memory, respectively [6][7].

In response to these issues, oversubscription (also known as overbooking) [8] has been heavily exploited at various cluster levels, spanning the kernel [9], hypervisor [10], and the cluster resource scheduler. In cluster management, this mechanism enables waiting jobs to exploit underused or idle resources currently allocated to other running jobs by launching *speculative* or *opportunistic* tasks in order to improve overall cluster resource utilization [11] [12] [13]. However, there still exist two interrelated problems:

Resource mismatch: Current approaches leverage the cluster resource states (i.e. available idle or revocable resources) to perform oversubscription decisions, and do so via piggybacking on regular cluster scheduler heartbeat messages. Speculative tasks will be sent, enqueued and launched on the given node, taking at least 3 message intervals. Given that the interval within production systems is typically configured to 3s [1][11], a total of 9s required perform oversubscription can result in substantive differences within the new cluster resource state. It is particularly devastating when considering that a large proportion of tasks only execute in seconds (e.g., MapReduce or Spark tasks). This mismatch between the acquired states and cluster resource states will generate invalid or low-quality task placement. Unsuccessful tasks have to require extra heartbeat intervals before they can be re-submitted, re-scheduled and dispatched onto a node, resulting in increased job makespan. Loosely-coupled schedulers may resolve this issue by assigning opportunistic tasks randomly [13] or on an per-application basis [14]. However, [14] is highly dependent on accurate queue delay times, and require customers who submit jobs to precisely estimate execution time which in many practical scenarios is infeasible.

QoS degradation: Tail (e.g., 95th or 99th percentile) latency has become the important and measurable attribute of quality of service (QoS) for large-scale Internet services. Existing cluster oversubscription focuses on maximizing cluster CPU utilization but omits stringent QoS requirements of interactive and latency-sensitive applications such as databases, key-value stores, etc. when such online applications are co-scheduled and co-located with computation-

- *R.Yang, X.Sun and J.Xu are with School of Computing, University of Leeds, UK. Email: {r.yang1, scxs, j.xu}@leeds.ac.uk.*
- *C.Hu, T.Wo and H.Peng are with Beihang University, China. Email:{hucm, woty, penghao}@buaa.edu.cn. C.Hu is corresponding author.*
- *P.Garraghan is with Lancaster University, UK. Email: p.garrahan@lancaster.ac.uk*
- *Z.Wen is with Newcastle University, UK. Email: zhenyu.wen@newcastle.ac.uk*
- *C.Li is with Alibaba Group, China. Email: c.li@alibaba-inc.com*

Manuscript received July 2019; revised Nov 2019

intensive batch jobs through multi-tenancy. Current cluster managers in YARN [1], Mesos [2] or Fuxi [3] are solely designed to tackle short-running tasks within batch jobs, whose performance is minimally affected when launching additional speculative tasks. Additionally, cluster managers (i.e. Resource Manager) are application-agnostic and completely unaware of runtime QoS – they are only responsible for resource allocation among applications/jobs but leave all application-specific logic to application managers. This leads to decreased responsiveness and increase tail latency for long-running applications (LRAs) which have intrinsic QoS constraints. In reality, interference is particularly destructive for these applications because their performance is extremely sensitive to the allocated resources and the contention for shared resources among co-located workloads may lead to performance unpredictability [15]. This problem will be further compounded in oversubscription framework, as even a slightly overdue resource reclaim from speculative tasks tends to violate the QoS substantially.

In this paper we propose ROSE, a resource management platform capable of conducting performance-aware resource oversubscription. Our approach maximizes cluster resource utilization whilst minimizing LRA's QoS degradation when performing oversubscription. The framework decouples oversubscription decision making from the centralized manager so that each job manager can independently leverage idle resources directly from the node agent situated within each machine to create speculative tasks without waiting for available resources to be released by the centralized resource manager. Our approach comprises two coherent stages to provision performance-aware oversubscription: *Placement* and *Execution*. At the placement stage, multi-phase machine filtering is the primary decision-making procedure that is used to determine machine suitability to launch speculative tasks by considering estimated load, correlative workload performance, and queue states, instead of relying on the estimation of task execution time. To minimize the incurred overheads of message flooding, the updated information are incrementally synchronized to each job manager. At the execution stage, a runtime agent is designed to launch and throttle speculative tasks in order to avoid QoS violation from excessive oversubscription. To achieve this, we develop a mechanism for admission control that temporarily queues tasks and decides whether waiting tasks can be launched according to multi-resource threshold restriction and a performance-aware resource throttling. This depends on quantifying machine-level performance variability and job-level responsiveness by employing streamized hardware counters (cycles-per-instruction and cache misses) in real time. Task upgrading and rescheduling are also conducted in job managers minimize head-of-line blocking and straggler manifestation [16].

We implemented and evaluated ROSE based on Alibaba's Fuxi [3], a multi-layered cluster management system. Experiments show that the average time of machine selection is in an order of hundreds of milliseconds and ROSE can roughly double CPU utilization in batch-only jobs. In case of the mixed co-location of batch jobs and latency-sensitive LRAs, the CPU and disk utilization can also reach 56.34% and 43.49%, and the 95th percentile of read latency in YCSB workloads only increases by 5.4% against executing LRAs alone. Particularly, our contributions can be summarized as follows:

- A distributed resource oversubscription architecture for cluster management that enables idle resources to be leveraged by creating speculative tasks. Two different oversubscription schemes are now used for adoption to different co-location scenarios.
- A multi-phase machine filtering process that considers diverse factors (e.g., task-level rating and machine states) to locate suitable machines for speculative task placement. Job managers can independently manage task life-cycle with task upgrading and rescheduling.
- A runtime mechanism for admission control that can exploit multi-resource threshold check and performance-aware resource throttle to dynamically adjust speculative task numbers and restrict resources that speculative tasks can use, thereby preventing excessive oversubscription.

Organization. Section 2 outlines our research motivation and Section 3 presents design principles and architecture overview. Key techniques are mainly depicted in Section 4 to 6. Section 7 discusses the evaluation followed by related work in Section 8. We conclude the paper in Section 9.

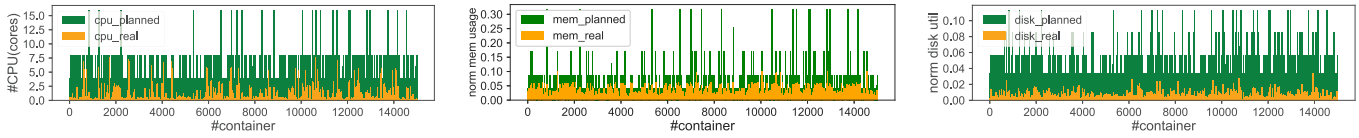
2 Motivation

2.1 Background

Resource Management. Modern scheduling systems typically decouple the resource management layer from the job-level logical execution plans to enhance system scalability, availability and framework flexibility. For instance, YARN[17] and Fuxi[3] share the following components: *Resource Manager (RM)* is the centralized resource controller, tracking resource usage, node aliveness, enforcing allocation invariants, and arbitrating contention among tenants. The component is also responsible for negotiation between available resources within the infrastructure and resource requests from Application Masters. *Application Master (AM)*, also termed job manager, is an application-level scheduler which coordinates the logical plan of a single job by requesting resources from the RM, generating a plan from received resources, and coordinating task execution. *Node Manager (NM)*, also termed Node Agent, is a daemon process within each machine responsible for managing local tasks (including launch, suspend, kill, etc.) and monitoring machine information.

Workload Characteristics. In production clusters, the cluster resources are usually consumed by batch jobs and LRAs. Various workloads that exhibit diversity in task scale and resource heterogeneity are physically co-scheduled and compete for the same underlying resources by either time multiplexing [18] or fair sharing according to fixed or dynamic quota on a node basis [19][20].

- **Batch Job**. Batch analytic jobs are big data processing applications that are insensitive to latency [3][21]. In reality, they are measured by the end-to-end completion time, and thus deadline-constrained, for their results to be consumed by downstream jobs or services. A job can be typically segmented into a large number of short-lived tasks with only subsecond or seconds duration.
- **Long-running Application (LRA)**. Long running applications (LRAs) encompass transaction analytics, online web



(a) CPU (cores) (b) Normalized Memory Usage ([0,1]) (c) Normalized Disk Usage ([0,1])

Fig. 1. Resource disparity in Alibaba's workload co-location cluster

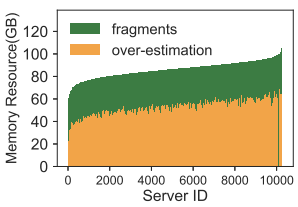


Fig. 2. Underused memory

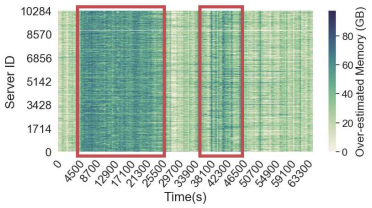


Fig. 3. Overestimated mem heatmap

services, or database services (e.g., HBase, Memcached, MongoDB, etc). Those containers are allocated and used for durations ranging from hours to months, and are typically latency-sensitive. Due to this characteristic, response latency and throughput are the key performance indicators and applications must meet strict QoS or Service Level Objectives (SLOs). Hence, the QoS-centric objective specifies a guarantee of a certain level of performance to those latency-sensitive applications.

Therefore, an imperative requirement of cluster resource management for co-located workloads is to improve cluster utilization without sacrificing both LRAs responsiveness and batch jobs performance. Meanwhile, when different types of applications co-exist in a system, different or adaptive oversubscription strategies will be required.

2.2 Cluster Utilization Issues

To ascertain a comprehensive understanding of resource characteristics within a production-level workload co-location environment, we developed a data collection and analysis pipeline based on (i) a recently released Alibaba tracelog (T_r) [22] that encompasses 24-hours traces from a production cluster which consists 1,300 machines and 23,000 jobs (millions of containers) and (ii) tracelog collected from an Alibaba's production cluster (T_i) that contains a consecutive 5-day's period over 10,000 machines.

Cluster resource usage is very low in production-level clusters. Logical resource utilization is a metric often studied to measure scheduler performance. Higher utilization implies more efficient scheduler decision making and faster job completion. Fig. 1 demonstrates an obvious disparity between real usage and the requested one in terms of CPU cores, normalized value of memory and disk of all running containers in the trace T_r. Particularly, those containers severely under-utilize resources: the real usage of CPU and disk on average is no more than 16% of the requested value. This finding can be also corroborated from other works [5][6] indicating low resource usage is a systemic issue in large-scale clusters.

Under-utilization is resultant of resource fragmentation and user overestimation. Within T_i, at a given time (i.e., at 27,500s), we collected the total amount reserved by all running jobs and calculated the fragmentation on each machine across the cluster. In this context, fragmentation is the resources that exists in the machines but cannot be

TABLE 1
Centralized oversubscription

#Spec	mQL	#Unq	#Resched	#Killed	#Timeout	#Succ
2,000	10	857	856	72	0	215
	15	804	890	79	0	227
	20	248	1,440	79	0	234
	50	24	1,664	78	0	237
	70	20	1,685	75	0	253
	avg	19.5%	65.3%	3.8%	0	11.7%
5,000	10	4,087	260	72	0	215
	15	4,111	283	79	0	227
	20	760	3,653	61	70	456
	50	9	4,398	51	32	510
	70	8	4,410	58	71	453
	avg	35.9%	52.0%	1.3%	0.7%	7.4%

leveraged due to the currently served jobs that have requested larger amount of resources. Fragmentation will not only under-utilize the cluster, but lead to delayed scheduling and reduced throughput. Additionally, users typically request excessive amounts of resources to handle workload bursting to avoid SLO violation. Fig. 2 depicts statistics ordered by the resource amount, showing that more than 80GB memory could be reused within most machines (with 132GB memory per machine). This phenomenon is also non-trivial among different machines over time. For example, we evaluate the overestimated memory changes in temporal-spatial heatmap. As shown in Fig. 3, the darker area indicates a larger value of overestimated memory and the overall range is shown in the side color-bar. We find out over 43.08% (marked in red squares) of the total time period (i.e, from 0 to 63,300s), the over-estimated memory surpasses 60GB which should be fully utilized. This indicates a huge potential of leveraging the underused resources via the use of *resource oversubscription*.

2.3 Limitations of Centralized Oversubscription

Centralized resource schedulers such as YARN and Mesos perform oversubscription decision making [11] [12] through a central manager. There are two obvious defficiencies in centralized oversubscription framework: resource mismatch and lack of QoS-centric consideration.

Centralized oversubscription approach leads to resource mismatch and inefficient resource utilization. Centralized oversubscription scheduling (e.g., Apache YARN 3.0 [11]) has been proposed to reuse such idle resources. It transmits the updated information in terms of idle or revocable resources from the central RM to a job via piggybacking on regular heartbeat messages within the cluster. Resource allocations for speculative tasks are also conducted by RM. However, the decision and heartbeat piggybacking mechanism will cause that task launching is lagged behind by at least 3 heartbeat intervals. The mismatch of the latest resource usage during heartbeat intervals will lead to a great number of task suboptimal placement or re-distributions. To demonstrate the consequential limitations, we conduct experiments to submit speculative tasks into a 32-machine Hadoop 3.0 cluster (each machine with 6-core Intel(R)

Xeon(R) CPU E5-2630 processors, 82GB RAM) and observe the numbers of unqueued, rescheduled, killed, enqueued but timing-out and launched tasks varying the number of tasks (*#Spec*) and customizable max queue length (*mQL*) on NMs. As shown in Table 1, when we submitted 2,000 speculative tasks, 19.5% of submitted tasks on average are excluded from the queue and 65.3% have to be re-scheduled even if they are allowed to enqueue. Merely 11.7% on average can be successfully launched. Additionally, when we increase the max queue length on each NM, the number of launched tasks will not ideally increase even if more tasks are allowed into the queue. This is because resource capacities are deterministic and the redundant tasks will be rescheduled instead. What is worse is that due to the inherent workflow in centralized resource scheduling, unsuccessful tasks require several heartbeat intervals before they can be re-submitted, re-scheduled and dispatched onto a new NM. For instance, when the task submission number grows to 5,000, the overall success ratio drops by 4.3% irrespective of the *mQL* increase. This reschedule-loop will result in an increased trend of rescheduling and eventual task timeout. Thus, the inefficiency greatly degrades the performance of resource oversubscription and job execution.

Current oversubscription approaches lack QoS-awareness mechanisms for LRAs. According to the de-coupled design in YARN and Mesos, the centralized scheduler (Resource Manager) is responsible for managing global resources and allocating resources to applications. However, the manager is completely agnostic with regard to both applications and job frameworks. This intuitively makes both regular resource allocation and resource oversubscription totally unaware of any information such as job progress or runtime QoS. There are interference-aware schedulers [7][23] we can potentially leverage to ensure a controllable QoS management for LRAs. However, they mainly infer the expected interference level of a given co-location, and heavily require automatic estimation of resource preferences and interference sensitivity. As this can only be done through offline profilings in as many co-locations as possible based on recurring job execution, it is prohibitively infeasible to conduct in case of stochastic combinations of a great number of batch and speculative tasks with LRAs. This necessitates a feedback-based controller integrated with current multi-layered resource management systems such that it can dynamically steer resource allocations for LRAs using generic application-independent performance monitoring and effective resource throttling.

3 System Design

3.1 Design Principles

System efficiency is typically characterized by its utilization and performance, for both resource providers and end users. Balancing system utilization and application performance (particularly LRA responsiveness) is therefore an important consideration. ROSE is a resource oversubscription framework that aims to seamlessly use two different oversubscription schemes for adoption to two distinct scenarios:

- **Batch only co-location**: The cluster only executes batch jobs, and we aim to adopt *utilization-prioritized (UP) oversubscription* scheme to aggressively oversubscribe resources and improve system efficiency.
- **Batch-LRA mixed co-location**: The cluster comprises both batch jobs and latency-sensitive LRAs. Towards a QoS-centric objective, we aim to use *responsiveness-prioritized (RP) oversubscription* scheme to assure the desired performance of latency-sensitive applications before improving cluster utilization.

Thus in order to improve utilization whilst constantly guaranteeing QoS, our roadmap follows three design principles encompassing architectural evolution, speculative task placement, and runtime execution management.

- **Distributed resource oversubscription:** We need an approach to effectively create speculative tasks by individual job managers in a distributed way. Such an approach should overcome issues associated with cluster resource inefficiencies (Section 2.2) and centralized oversubscription limitations (Section 2.3) for heterogeneous workloads [6][24]. To underpin QoS-centric awareness in resource oversubscription, we also highly require runtime performance monitoring other than machine states and loads collection. The distributed design is coherently divided into two key stages: machine selection for task placement and runtime management for task execution.
- **Task placement stage:** We need to determine the most suitable placement for speculative tasks, particularly bypassing machines that are likely to incur QoS violation of existing workloads. The procedure of selection should thoroughly exploit cluster diversity in terms of heterogeneous resources and dynamic resource usage; and monitor and aggregate both application-level and multi-dimensional system information for optimally exploiting fragmented resources and allocated (yet idle) resources. The ultimate goal is to raise the success rate of oversubscription (i.e., reducing rescheduling or evictions of speculative tasks).
- **Task execution stage:** Runtime management is essential to navigate the manipulations during the whole life-cycle of speculative tasks. Additionally, to achieve responsiveness-prioritized oversubscription, it is imperative to leverage a feedback-control based throttling for dynamically conforming to the QoS requirements of LRAs at anytime.

3.2 Distributed Oversubscription: Core Concept

The procedure of resource oversubscription of speculative tasks will be decoupled from the centralized resource manager and dispersed into distributed Job AppMasters (short for Job Master or JM). JM will independently make decisions of speculative task placement as shown in Fig. 4.

A job requests resources from the RM (step1). Once total resources in the cluster have been allocated, no further regular resources are assigned to jobs (step2). Instead of waiting for the emergence of available resources released by the RM, JM will individually attempt to request additional resources directly from per-machine Runtime Agents in a speculative manner (step3). The job then simultaneously requests to launch speculative tasks in machines that are most suitable for oversubscribing resources. To manage speculative tasks at node side, ROSE Runtime Agent will maintain a queue of submitted speculative tasks and decide whether the incoming speculative tasks should be accepted or rejected on

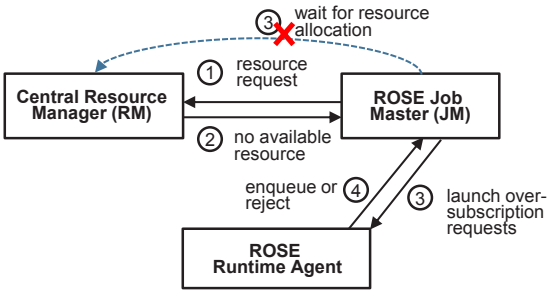


Fig. 4. Basic Idea of Speculative Oversubscription

the basis of current queue length. Once accepted, the tasks can be dispatched onto the physical machine and enqueued (step4). If rejected, corresponding JMs will be notified and new attempts will continue periodically.

At runtime, in the life-cycle of speculative tasks, ROSE Runtime Agent determines the timing of launching speculative tasks from task queue and limits the number of oversubscribing resources and speculative tasks. To prevent excessive resource oversubscription and the consequent QoS degradation, the admission controller will also throttle the resources that can be re-used by speculative tasks according to real-time LRA performance. To prioritize regular tasks, speculative tasks will run at lower priorities and are preemptable by other co-running regular tasks and LRAs.

ROSE is designed to be complementary and compatible to existing protocols between JMs and RM, and thus can enhance existing two-layered scheduling systems. In reality, RM does not perceive speculative tasks and their utilized resources. The idle resources are actually taken and re-used by the speculative tasks without being detecting by regular tasks. The fragment or idle resources therefore can be fully utilized by the speculative tasks, and the task instance can be immediately scheduled to corresponding machines.

3.3 Architecture Overview

Fig. 5 illustrates the architecture overview. In general, LRA's AM inherently follows the traditional two-tier scheduling protocols where AM requests resources from the centralized resource manager. To achieve the aforementioned goals, key system components encompass the Distributed Fingerprinting that collects real-time metrics and forms data streaming; the Cluster Aggregator (CA) that consumes the holistic metrics and provides machine suitability; the JM that is responsible for task scheduling and speculative task placement; and the Runtime Agent that manages task runtime execution and resource admission control.

Distributed Fingerprinting. For tracking system states and workload performance, we need to collect system fingerprints including machine-level, workload-level live metrics and generalized performance counters. For instance, in per machine runtime agent, we use open-source technique such as Fluentd [25] in *Metric Publisher* to locally collect and store high resolution metrics featuring hundreds of data points per cycle. To deal with back pressure and network latency, we use Apache Kafka [26] as an intermediate metrics buffer and publish-subscribe (pub-sub) messaging subsystem, thereby establishing flexible and robust data streaming. The produced metrics will be subscribed by

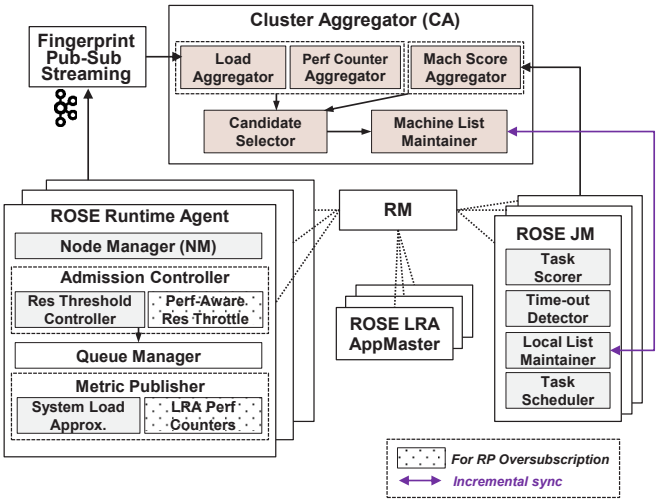


Fig. 5. ROSE architecture and design

metric consumers (such as CA and ROSE Runtime itself) according to specified topics (see details in Section 4).

Cluster Aggregator. It is the key component that is decoupled from RM and aggregates runtime information and task-level status. We use Telegraf ingest to consume metrics from the upstream data pipeline and write to local time-series database such as InfluxDB. Specifically, CA collects machine loads and queue states of each machine and machine scores given by each *JM*. For RP oversubscription, machine-level performance counters are also included in the aggregation. Based on these information, the Candidate Selector will periodically exploit a multi-phase machine filtering mechanism to select and rank machine candidates prior to oversubscribing resources. Machine List Maintainer will incrementally synchronized the list to all JMs, instructing them to launch the speculative tasks on proper machines. We will present the details in Section 5.

ROSE JobMaster. JM is a specific *AM* that leverages the proposed oversubscription mechanism to compensate underutilized resource requests. Local List Maintainer locally holds a replica of candidate machine collection that is incrementally synchronized from CA. *Task Scheduler* can therefore dispatch speculative tasks by exploiting these machine candidates in a random or round-robin way. Meanwhile, it adaptively coordinates the task upgrading between speculative tasks and regular tasks when resources are granted. Due to variations in cluster states, the previous task placement might become sub-optimal, resulting in the head of line blocking. Timeout Detector will timely trigger a timing-out re-scheduling to mitigate the task starvation (see details in Section 6.3). Additionally, to help reflect the latest machine status, *Task Scorer* is used to rate machines based on the internal state and the state transitions of all tasks within the job. The task-level scoring, therefore, becomes a valuable criterion during the machine filtering process within the CA.

ROSE Runtime Agent. Runtime Agent is the daemon running on each machine and responsible for task execution. The native NM is responsible for regular task management, and *Queue Manager* deals with the incoming tasks from different JMs. Tasks in the queue will wait for being launched by the *Admission Controller* that determines whether the node capacity and current performance status allow for further resource oversubscription based on real-time ma-

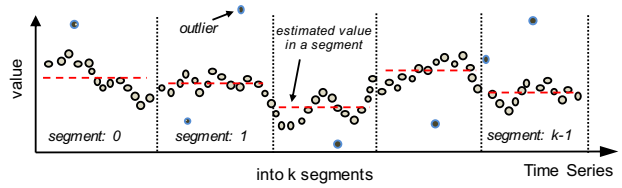


Fig. 6. Depiction of quick load approximation

chine loads, LRA's performance indicators, etc. This is primarily performed by multi-resource threshold controller (see details in Section 6.1) and performance-aware resource throttling (see details in Section 6.2) to avoid excessive oversubscription and influence on LRAs' performance.

UP/RP Global Switch. It is worth noting that we have a global co-location configuration (which is not marked in Fig. 5 for clarity) for cluster administrators to specify the target scenario (either batch-only or latency-sensitive mixed co-location). Correspondingly, a global flag in the system will be set to switch between the UP and RP oversubscription schemes for scenario adaption. ROSE provides bespoke resource saturation control inside each node agent. The performance-aware resource throttling is an exclusive procedure that adheres to RP oversubscription and targets the latency-sensitive batch-LRA scenario. By contrast, in batch-only scenario, the throttle will be turned off, thereby allowing for aggressive oversubscription as long as other conditions of resource thresholds are satisfied.

4 Distributed Fingerprinting

To underpin precise machine selection while measuring the resource utilization and cluster health, we track system fingerprints comprised of system loads and performance indicators in a distributed way and Metric Publisher will populate the fingerprints into the data stream. We further adopt load approximation to mitigate the overheads in CA.

4.1 System Load Monitor and Approximation

We track machine-level resource utilization (*cpu_util*, *memory_util*, *disk_util*, and *network received/transmit*) and the waiting/running numbers for both regular and speculative containers in each machine. We obtain these information during a fixed time interval and publish them into the Kafka data stream. Considering the rapidly-produced metrics across machines, at the side of consumer, CA timely digests the incoming data and approximates the load of each machine within a sliding time window. Meanwhile, CA only needs to store a small amount of latest metric data used for approximating and the resultant estimation values in the local time-series database, instead of storing the huge amount of all history raw data. Given the observations that load data are amendable to an approximate, the load approximation can substantially reduce the stored data but maintain high-fidelity, thereby reducing the resource overheads and mitigating the potential bottleneck in CA.

We leverage a piecewise-based approximation and de-noising load acquisition to accelerate the runtime load estimation without substantial precision degradation. Alg. 1 and Fig. 6 depict the main process. We delimit the basic calculation unit with fixed number of data points conducted over a slide window. We further divide the entire time

Algorithm 1 Load Level Approximation (LLA)

Input: D – a set containing continuous tracedata of a machine metric;
k – the pre-defined granularity of accuracy (default value is 5);
Output: v: a predicted tendency value

1: **if** D is monotonous **then**
2: **return** the last element of D
3: **elseif** $|D| < k$ **then**
4: $T \leftarrow$ eliminate outliers via Alg. 2
5: **return** $mean(T)$
6: **endif**
7:
8: **for** $i \in [1..k]$ **do**
9: **let** $S_i \leftarrow \varnothing$
10: **endfor**
11: **for** each $d \in D$ **do**
12: $S_i \leftarrow S_i \cup \{d\}$
13: **if** $|S_i| \geq \lceil\ |D|/k\ \rceil$ **then**
14: $i = i + 1$
15: **endif**
16: **endfor**
17:
18: **let** $D' \leftarrow \varnothing$
19: **in parallel** each S_i **do**
20: $T \leftarrow$ eliminate outliers via Alg. 2
21: $D' \leftarrow D' \cup mean(T)$
22: **endparallel**
23:
24: **if** D' is monotonous **then**
25: **return** the last element of D'
26: **else**
27: $T \leftarrow$ eliminate outliers via Alg. 2
28: **return** $mean(T)$
29: **endif**

Algorithm 2 Outlier Elimination

Input: S_i – a subset containing continuous tracedata of a machine metric;
Output: T: a set eliminated outliers from S_i

1: $Q_1 \leftarrow$ lower quartile of S_i
2: $Q_3 \leftarrow$ upper quartile of S_i
3: $IQR \leftarrow Q_3 - Q_1$
4: $T \leftarrow \{d | d \in [Q_1 - p * IQR, Q_3 + p * IQR]\}$ (customarily p is 1.5)
return T

period into k segments alongside the timeline (Lines 8-16). It is worth noting that all segments conduct the average evaluation in parallel (Lines 18-22). In each segment, we calculate the average load by de-noising the sample data based on Tukey's boxplot [27][28] to pinpoint and eliminate potential outliers (see Alg. 2). ROSE is compatible with other statistical methods of outlier detection [29][30] to handle data with asymmetric distribution. After the local calculation, we can obtain the value set D' containing k average values. Subsequently if the elements are monotonous, we can easily determine the target value according to the tendency (Lines 24-25). Otherwise, we regard the mean value with eliminating outliers of D' as the estimated result (Lines 26-28). CA employs LLA to approximate latest load level while periodically (e.g., every 10s) triggering a machine selection to update the suitable machine list. For further optimization, we can implement the LLA on a per machine basis with Kafka processors before the intermediate results are sent to the CA, thereby further reducing the time consumption of load aggregation within CA.

4.2 LRA Performance Monitor

To early detect machine-level performance variability and performance interference among co-running workloads, we

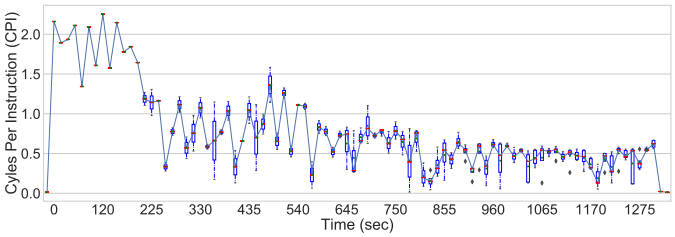


Fig. 7. CPI fluctuation over LRAs and time on a specific machine

use a non-invasive fine-grained collector to capture runtime performance of machines and LRAs. To reduce the tight dependency upon specific workloads, we define and use generalized performance indicators as important criteron for RP oversubscription.

Hardware Event Based Performance Indicators. We rely on Clock Cycles per Instruction (CPI) and Cache Miss Per Thousand Instructions (MPKI) collected from hardware events at both application-level and machine-level to indicate the responsiveness of running LRAs and the overall machine performance. In reality, CPI is the average number of cycles used by each instruction for a given execution of a given program. MPKI is a statistic that describes the number of misses out of total cache accesses, which reflects how workload behaves on a specific machine with given cache hierarchy. It has been demonstrated that compute-intensive application behavior is positively correlative to the monitored (CPI) changes and most latency-sensitive applications have fairly consistent CPIs [31]. Intuitively, higher CPI and MPKI indicate slower execution and frequent cache misses. We mainly focus on CPU interference derived from core sharing and LLC (Last Level Cache) contention in this paper. Other techniques [32][33] can be added to detect and handle IO or disk interference.

Metric Collection. CPI and MPKI can be easily measured without application level input and thus are quickly enough to capture application behavior. Statistic CPI can be derived from hardware counters periodically by using *perf stat* tool. When the *perf_event* subsystem is attached to a *cgroup* hierarchy, all cgroups under the hierarchy can be tracked to indicate behaviors of the pertaining processes and threads using the *perf* tool. In practice, we collect such metrics per container and per machine by *docker uuid/process pid* and the *root cgroup* respectively. As indicated in[31], the CPU overhead is less than 0.1% and no visible latency will be incurred to end-users. Fig. 7 shows an example of temporal variability of CPI on a specific machine where only latency-sensitive LRAs are hosted. The boxplot shows that the deviation among different LRAs fluctuates over time. Some of them experience higher CPI values due to the continuous resource contention. In principle, if LLC misses increase, the workload is highly likely to acquire more CPU cycles to load the missed cache and thereby causing interference to others. By exploring hardware counters like CPI and MPKI from different machines, we can also infer and differentiate the variability among different machines.

5 Multi-Phase Machine Filtering

In this section, we demonstrate how to use multi-phase filtering to calculate the most suitable machine candidates for speculative task placement.

Algorithm 3 Task Behavior Aware Machine Rating(TBAMR)

Input: JM – terminated jobs' masters in a specific time period, JM_i represents the i^{th} job's master;
$Task$ – all tasks, $Task_{ij}$ represents the j^{th} task of the i^{th} job consisting n_i tasks, where $j \in \{1, 2, \ldots, n_i\}$;
Output: $Score$ – a set consisting synthetic scores for all machines;

1: **in parallel** each $JM_i \in JM$ **do**
2: **let** $penaltyScore \leftarrow \varnothing$
3: **for** each $Task_{ij} \in Task_i$ **do**
4: **if** $Task_{ij}$'s status $\in \{failed, crashed, tailed, killed\}$ **then**
5: $m \leftarrow getHostID(Task_{ij})$
6: $penaltyScore_m \leftarrow penaltyScore_m + 1$
7: **endif**
8: **endfor**
9:
10: $penaltyScore' \leftarrow topK(penaltyScore)$
11: **for** each $ps_i \in penaltyScore'$ **do**
12: $Score_i \leftarrow Score_i - ps_i$
13: **endfor**
14: **endparallel**
15: **return** $Score$

5.1 Multi-Phase Filtering Mechanism

Task Behavior Aware Machine Rating (TBAMR). To accurately reflect the state of task execution, each machine is assigned a satisfaction level reflecting their ability to successfully execute speculative tasks, calculated through the use of historical job execution data. Specifically, any action that negatively impacts task execution such as long-tail, failure, task kill and eviction, is regarded as negative behavior that reduces the machine satisfaction score. This machine rating is calculated for each machine by all JMs (as shown in Alg. 3). Eventually, each JM will produce perceived machine scores into the data stream and subsequently consumed and aggregated by CA.

Multi-phase Machine Filtering. To select the most suitable machine set where speculative tasks should be placed, ROSE adopts a multi-phase machine filtering mechanism. This mechanism considers runtime load, task-level machine performance, and the queue status of each machine. With these phases, we can take advantage of both the load-balanced and minimized-queuing, thereby mitigating the head of line blocking problem. Alg. 4 depicts the core phases:

a) Blacklisting from perspective of task-level assessment, time-out machine detection and runtime resource alerting. The cluster aggregator will converge the evaluation scores of all machines from completed jobs. The lowest K machines will be marked as weak performance machines which have a higher likelihood of being removed from future scheduling (Line 5 and Line 12). In addition to low scoring machines, temporarily timing-out machines are also eliminated from task placement (Line 13). Additionally, the overloaded machines are filtered out once any monitoring dimension (such as system loads, max-queue-length, max-container-number, etc.) surpasses the pre-defined threshold (Line 14). This is steered by the multi-resource threshold controller in the Runtime Agent (see details in Section 6.1).

b) Blacklisting machines with high runtime CPI and MPKI. According to the global configuration, if RP oversubscription scheme is enabled, we will consider runtime performance counters in machine filtering. We will eliminate those machines that have frequent cache misses and higher clock cycles per instruction, and thus are likely to slow down latency-sensitive applications (Lines 7-10).

TABLE 2
Parameters in Algorithms

Parameter	Meanings
M	the cluster machine collection where M_i represents the i^{th} machine and $i \in [1, n]$
$\boldsymbol{LM_{m\times n}}$	a matrix of monitored load information. i.e., $[\overrightarrow{LM_1}, \overrightarrow{LM_2}, ..., \overrightarrow{LM_n}]$
$\overrightarrow{LM_i}$	m-dimension metrics of machine M_i. i.e., $[LM_{i1}, LM_{i2}, ..., LM_{im}]^\intercal$. $\overrightarrow{LM_i}$ can be divided into the load-relevant part $\overrightarrow{LM_i^l}$ and queue-relevant part $\overrightarrow{LM_i^q}$. Namely, $\overrightarrow{LM_i} = (\overrightarrow{LM_i^q}, \overrightarrow{LM_i^q})$
LM_{ij}	the j^{th} dimension of monitored information of M_i
$\overrightarrow{LM_i^l}$	load-relevant dimensions in $\overrightarrow{LM_i}$
$\overrightarrow{LM_i^q}$	queue-relevant dimensions in $\overrightarrow{LM_i}$. i.e., $[rc\_am, oc\_w\_am, oc\_r\_am]^\intercal$
CPI	Machines' avg CPI collected in a sliding time window
$MPKI$	Machine's avg MPKI collected in a sliding time window
L_{max}	a bespoken maximum amount of candidate machines
$\overrightarrow{queueFilter}$	$[rc\_am, oc\_w\_am, oc\_r\_am]$, a weighted filter vector for queue-relevant dimensions. The nil element means that the dimension is excluded from the selection
$\overrightarrow{loadFilter}$	$[cpu\_ut, mem\_ut, load\_avg\_1, disk\_s\_ut, disk\_ut, disk\_usage, net\_rec\_ut, net\_tra\_ut]$, a weighted filter vector for load-relevant dimensions

c) Machine selection based on approximated loads across multi-resources. Heuristic algorithm in [34] uses dot product to reduce the alignment of different tasks to a machine across multi-dimensions to one-dimension case. Inversely, ROSE intends to match a suitable set of machines with given tasks according to the approximated load level. For dimensionality reduction, we consider the dot product of the scheduler's weighted preference for each resource dimension and the used amount of each dimension. Herein, the vector $\overrightarrow{loadFilter}$ represents the configurable weight values showing the dominant degree of the given dimension in the resource management system. The dot product value of runtime load information and the weighted filter $\overrightarrow{loadFilter}$ for a given machine can be regarded as the approximated *load index* to imply the accumulated level over the corresponding dimensions ($\overrightarrow{LM_i^l}$). Notably, within the procedure of implementation, to ensure the same numerical range, we normalize the available resources and loads of different machines into a uniform value by the maximum machine capacity in the cluster and record the values of each dimension of all machines into $\boldsymbol{LM}$ (Lines 1-4). In reality, the dot product prefers to place tasks onto light-loaded machines whose available resource is much more sufficient than others. The overall resource utilization can be promoted accordingly. In this context, we order the load index of all machines (Line 19) and filter out a set of machines which are most fit for oversubscription (Line 24).

d) Machine selection based on queue states. Under similar load circumstance, the capability of launching tasks as soon as possible is significantly important to shorten the job execution time. Thus, another factor to consider in the mechanism is the length of queue state. It can indirectly indicate how soon the waiting tasks can be allocated onto a given machine. The queue relevant statistics $\overrightarrow{LM_i^q}$ such as waiting and running task numbers are also profiled periodically. For example, apart from regular tasks, we record the number of running and waiting speculative tasks in the $\boldsymbol{LM}$. Similarly, an index of queuing status is calculated by the dot product operator (Line 20). In this manner, the 3-tuple information for each machine $(M_i, lIndex, qIndex)$ can be obtained and aggregated (Line 21). On the basis of the load pre-filtering, the final selection phase is to determine the L_{max} machines that is prepared for those waiting JMs to accurately leverage idle resources (Lines 25-30).

Algorithm 4 Multi-phase Machine Filtering (MMF)

Input: $(M, JM, Task, CPI, MPKI, L_{max})$
Output: C – candidate machines fit for oversubscribed resources;

1: **let** $\boldsymbol{LM} \leftarrow \varnothing$
2: **for** each $M_i \in M$ **do**
3: $\quad \boldsymbol{LM} \leftarrow \boldsymbol{LM} \cup \{\textbf{LLA}(M_i).normalize()\}$
4: **endfor**
5: **let** $MachScore \leftarrow \textbf{TBAMR}(JM, Task)$
6:
7: **if** *the type is RP oversubscription* **then**
8: $\quad B_0 \leftarrow \{M_i \mid PS_i \in topK_{descend}(CPI)\}$
9: $\quad B_0 \leftarrow B_0 \cup \{M_i \mid PS_i \in topK_{descend}(MPKI)\}$
10: **endif**
11:
12: $B_1 \leftarrow \{M_i \mid MS_i \in topK_{ascend}(MachScore)\}$
13: $B_2 \leftarrow \{M_i \mid M_i \in M$ and M_i is disconnected$\}$
14: $B_3 \leftarrow \{M_i \mid LM_{ij}$ is over the j^{th} threshold $\}$
15: **let** $M' \leftarrow M - B_0 \cup B_1 \cup B_2 \cup B_3$
16:
17: **let** $candidateInfo \leftarrow \varnothing$
18: **for** each $M_i \in M'$ **do**
19: $\quad lIndex \leftarrow \overrightarrow{LM_i^l} \cdot \overrightarrow{loadFilter}$
20: $\quad qIndex \leftarrow \overrightarrow{LM_i^q} \cdot \overrightarrow{queueFilter}$
21: $\quad candidateInfo \leftarrow candidateInfo \cup \{(M_i, lIndex, qIndex)\}$
22: **endfor**
23:
24: $\mathcal{C}' \leftarrow candidateInfo.ascendSortBy(lIndex).topK(d * L_{max})$
25: $\mathcal{C}'' \leftarrow \mathcal{C}'.ascendSortBy(qIndex).topK(L_{max})$
26: **let** $C \leftarrow \varnothing$
27: **for** each $c \in \mathcal{C}''$ **do**
28: $\quad m \leftarrow extractCandidateMachine(c)$
29: $\quad C \leftarrow C \cup \{m\}$
30: **endfor**
31: **return** C

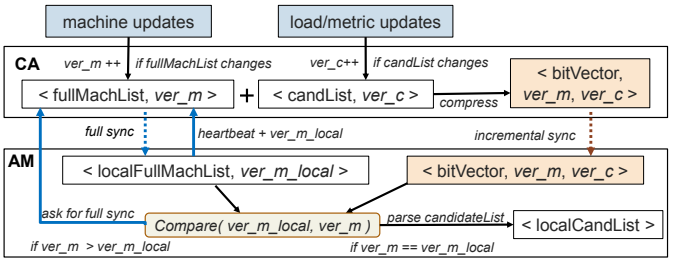


Fig. 8. Incremental transmission optimization

5.2 Incremental Information Synchronization

Considering the scalability and huge overheads of flooding communication with an increased number of jobs and machines, we optimize the transmission by incremental synchronization to keep a consistent view of machine candidates calculated by the MMF procedure. Essentially, we adopt a version control mechanism to label the different generation and candidate changes. We compare the locally cached version with the versionID recently sent by CA to decide if we need to synchronize the candidates.

Fig. 8 shows the basic pipeline. The CA maintains the full machine list with its version *ver_m*. Once machines join or leave the cluster, CA will acquire the full list updated with an added *ver_m*. MMF algorithm (Alg. 4) will be periodically performed, producing the latest candidate list and its updated version *ver_c*. Instead of transmitting the full candidate list, CA quickly generates a bit vector where each bit represents if the corresponding machine is in the candidates. The bit-vector and the above two version information will be periodically passed to JMs.

In each JM, the local maintainer keeps track of a local replica of full machine list and a local version *ver_m_local*. This local version will be piggybacked by heartbeat to the CA. If the carried version is older than CA's *ver_m* indicating the inconsistency, CA will trigger a mandatory response to synchronize the full machine list to the JM. To revive the candidate machines that CA determines the best fit for oversubscription, JM will compare the *ver_m_local* and the *ver_m* in the bit-vector information. If they match, the list will be locally cached and later used in the job scheduling. Otherwise, JM will request a full synchronization to catch up with the latest machine information. By leveraging this decoupled maintenance, JM can independently conduct oversubscription in a distributed manner.

5.3 Algorithm Complexity Analysis

Complexity Analysis. The overall time complexity of Alg. 1 is $O(d)$ where d denotes the number of elements in D. The step of dividing elements of D into subsets requires $O(d)$ time. The core operation is to determine lower and upper quartiles in each subset S_i with at most $\lceil \frac{d}{k} \rceil$ elements. Randomized Selection Algorithm is a classic means to solve top-K selection on a set with n elements, and the expected time is $\Theta(n)$ [35]. As selecting quartiles is a special case, the above operation will require $O(\lceil \frac{d}{k} \rceil)$ time. Hence the complexity is $O(d + \lceil \frac{d}{k} \rceil) = O(d)$ assuming the number of pre-defined granularity of accuracy k is a constant. The overall time complexity of Alg. 3 is $O(n_t + n_p)$ where n_t denotes the task number in JM_i and n_p denotes the number of penalized machines. Specifically, the calculation of penalty scores takes $O(n_t)$ time as it depends on each task's status shown in JM_i's execution report. The following-up operation is to merge top-K penalized scores, which traverses all penalized machine scores and thus takes $O(n_p)$ time.

The overall time complexity of Alg. 4 is $O(nd)$ where n denotes the number of machines and d represents the number of sampling points. Alg. 4 is initially dependent on Alg. 1 and 3. Since performing LLA on each machine requires $O(d)$, approximating the loads for all machines takes $O(nd)$ time. As Alg. 3 is executed in each individual JM in parallel, obtaining all these scores merely consumes $O(1)$ time. Subsequently, blacklisting unwanted machines is virtually implemented by top-K selection, resulting in $O(n)$ time complexity. Moreover, generating load and queue indexes for all machines takes $O(n)$ time and the final machine filtering via top-K takes additional $O(n)$. This results in an overall time complexity of $O(nd + n) = O(nd)$. To validate the theoretical analysis, we also measure the time consumption of the placement decision in our experiment. Nevertheless, the complexity of Alg. 4 can be reduced to $O(n)$ if LLA is dispersed onto different machines. In effect, if LLA is calculated in parallel at the machine end or integrated in Kafka processor before populating into the data stream, CA can directly use the result without taking $O(nd)$ time to calcuate. Certainly, this will increase the computation and communication costs in the data stream.

Comparison. We also compare Alg. 4 against other existing schemes including random-based (RB), system load based (SLB) and queue length based machine selection (QLB) that are explicitly defined and used in Section 7.1. Due to the intrinsic nature of load ranking, SLB algorithm is also dependent upon Alg. 1 executed on each machine and selects top-K machines according to the sampled loads, resulting in the same $O(nd)$ time complexity. By contrast, QLB requires $O(n)$ time to pick top-K machines with shortest queue because it merely requires the length of the queue in each machine which can be directly obtained. RB is the quickest algorithm as it is independent of any input of information and thus performs $O(1)$ complexity. Purely from the time complexity perspective, our approach is no better than other approaches. However, it is still acceptable considering the satisfactory decision time (hundreds of milliseconds) and substantially improved quality of speculative task.

6 ROSE Runtime Management

ROSE runtime is composed of two key components including (i) oversubscription admission control that determines the resource saturation and oversubscription degree, and (ii) job level task scheduling that manages the whole life-cycle of the pertaining tasks. Admission control encompasses a general speculative task restriction using multiple resource threshold and a resource throttling procedure especially towards RP oversubscription.

6.1 Multi-Resource Threshold Based Control

Multi-resource Threshold Control. Based on runtime system information, the admission controller of task execution primarily manages the whole life-cycle of speculative tasks including task enqueue permission, execution start time, resource allocation, task preemption with priorities, etc. We firstly employ a resource threshold controller to determine the timeliness of task execution. A speculative task can be popped from the queue and started if all resource metrics are within the limits of upper bounds and the overall oversubscription quota does not surpass the maximum threshold. Once any dimension is beyond the corresponding threshold value, the oversubscription will be temporarily blocked until the launching condition is satisfied. This also creates an opportunity to configure a certain resource requirement according to user or system administrator demand and machine heterogeneity. When the regular tasks request to revoke resources, task preemption will occur. As a result, delicate machine selection can facilitate the speculative tasks running on reasonable machines where such preemption infrequently occurs.

Resource Isolation. In general, we leverage *cgroup* [36] and its sub-systems such as *blkio, cpu, cpusets, memory, net_cls* to enforce the IO, CPU, memory and network isolation and control for oversubscription. We generate a tree-based structure for each resource dimension and the root node of the hierarchical structure describes and controls the resource

isolation. We define two node types to serve – the regular group and oversubscribed group. As the immediate children of the root, the nodes represent the parent for task containers. We separate tasks into two sub-tree collections through placing speculative task and regular task within the oversubscribed and regular groups, respectively. Each group manages its own priority and tasks within the oversubscribed group have lower priorities than those in the regular group. This mechanism ensures that the speculative tasks are preempted by higher prioritized tasks. To enhance QoS management for LRAs, we further adopt CPU pinning, LLC and memory bandwidth isolation to protect LRAs from excessive oversubscription (detailed in Section 6.2).

6.2 Performance-aware Resource Throttling

To implement RP oversubscription for coherently enabling latency-sensitive co-location, we integrate a performance-aware resource throttle with the admission controller. Since CPU is the dominating bottleneck resource for most LRAs, the mechanism elaborately restricts the oversubscribable resources (particularly CPU cores and LLC) to make sure LRAs have constantly sufficient resources and normal performance indications. Using LRA's responsiveness as feedback signal, the core idea of feedback-control is to temporarily curtail the amount of resources shared by speculative tasks and dynamically restrict the launching number of speculative tasks for calibrating corresponding performance within acceptable boundaries. Fig. 9 briefly depicts the overall pipeline of resource throttling. Specifically, the procedure encompasses the following steps:

Protective Resource Isolation for LRAs. We enforce CPU and LLC isolation to prioritize LRA performance. We restricts which CPU cores speculative tasks can use and ensure LRAs have slack cores for thread switch wake-up or switch within LRAs. We specify all CPU cores in a machine into two distinct classes using *cgroup cpuset*: the 1^{st} class cores consisting of the amount requested by LRAs and the number of slack cores are prioritized for LRAs. The remaining cores in the machine are 2^{nd} class that are unrestricted for launching any regular batch tasks and affiliated speculative tasks. In this context, incoming speculative tasks will be preferably restricted to the 2^{nd} class cores to reduce the interference to running LRAs. Naturally, the eviction order will be reversed when resources have to be mandatorily reclaimed to regular tasks and LRAs. Additionally, to tackle LLC and memory bandwidth isolation, we mainly rely on Cache Allocation Technology (CAT) and Memory Bandwidth Allocation (MBA) powered by Intel Resource Director Technology (Intel RDT) [37] to dynamically tune and direct the cache ways and memory bandwidth for LRAs.

Early Awareness of Performance Anomaly. In aforementioned Section 4.2, the CPI and MPKI values are measured and analyzed at both machine level and container level. For periodical operations, the values of performance indicators exhibit low variability given that a given application should have similar instructions or queries to execute and stable resource usage pattern. Thus if there is a significant metric deviation from the normal behavior at runtime, the victim application tends to violate its service objective. To rescue this performance degradation, the oversubscribed resources should be throttled down and instantly reclaimed so that the victim application's performance can rebound to the norm. To achieve this, we will profile the CPI and MPKI distribution for the LRAs in our system. Fig. 10 depicts an example of measured distribution of a specific database application – the skewed form indicates a long-tail CPI distribution where low performance containers have a relatively higher CPI values. Through statistical analysis, we conclude that gamma and chi-square distribution are the best fits for CPI and MPKI respectively. We use the 3σ criteria to determine an outlier if the value surpasses the 3σ boundary on the best-fit distribution.

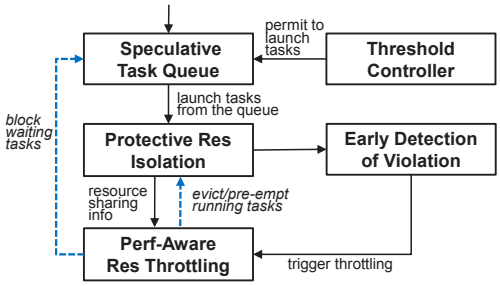


Fig. 9. Flowchart of resource throttling scheme

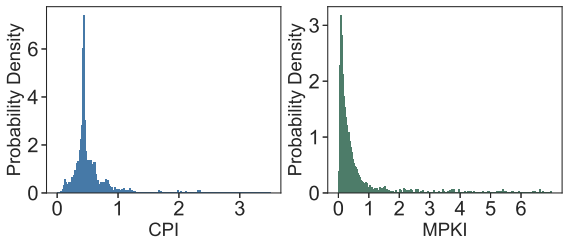


Fig. 10. Histogram of CPI and MPKI

Runtime Throttling. If we find an performance outlier on a node, we attempt to throttle the oversubscription degree by temporarily blocking more tasks from being launched from the queue and pre-empting running oversubscription tasks. Namely, all pending and existing oversubscription tasks will be postponed to give ways to the victim applications. We prefer to firstly preempt speculative tasks pertaining to 1^{st} class cores and having least execution progress to mitigate the degradation to LRAs. To precisely determine the amount of resources to be spared for rescuing the performance (equivalently how many speculative tasks will be affected), we further explore the quantitative relationship between allocated resource and consequent workload performance through learning the sensitivity of application performance to multiple resources. By using this model, we could infer the proper slice of resources for rapid performance recovery. Due to the limited space, we omit the details in this paper but they can be found in [38].

6.3 Runtime Task Scheduling

Task Upgrading. In the event of unfulfilled resource requests due to insufficient resource, the JM will proactively dispatch several speculative tasks onto NMs (Alg.5 Lines 1-3) according to the latest result of MMF procedure. The JM also tracks all launched tasks until their completion. Once the waiting resource is approved and the corresponding request can be further fulfilled, the JM determines whether a speculative task has been launched and which machine it is executing within. If the available resources are assigned

Algorithm 5 Task Scheduling in Job Master

```
1: if resource requests cannot be satisfied then
2:    dispatchSpeculativeTask()
3: endif
4:
5: if waiting resource has been approved then
6:    specTaskLoc, resLoc ← where(specTask, resource)
7:    if specTask is in queue or specTaskLoc is null then
8:       cancelSpeculativeTask(specTtask)
9:       startRegularTask(resource)
10:   elseif isSamePlace(specTaskloc, resloc) then
11:      upgradeSpeculativeTask(specTask)
12:   elseif specTask progress is more than τ then
13:      keepSpeculativeTask(specTask)
14:      reserveRes(resource)
15:   else
16:      killSpeculativeTask(specTask)
17:      startRegularTask(resource)
18:   endif
19: endif
```

to a resource request that is served by an oversubscribed resource, the speculative tasks should be transformed to regular tasks. In effect, the task upgrade is achieved according to the task location and execution progress. Ideally, we preferably expect to alter the speculative task on the same machine because there is no additional initialization time to reschedule and launch. In our implementation, if speculative tasks have not yet started (either staying in the queue or failing to find destinations), we will cancel them and directly start the regular tasks (Lines 7-9). Otherwise, if the launched task has already been executed on the same machine, we will directly re-label it as a regular task (Lines 10-11). ROSE judges whether it is cost effective to evict the speculative task based on execution progress. If the progress surpasses a predefined threshold (e.g., 60%)(indicating a near-completed task), we retain the task and also reserve the recently approved resource in the event of speculative task failure (Lines 12-14). If the task is started with a progress no greater than the threshold, the task is killed directly and launched using new resources (Lines 15-17). The resource reservation is seemingly contradictory to the improvement of resource utilization. However, the reserved resource can be oversubscribed again to other tasks, thereby reducing overall eviction and shortening the job makespan.

Task Rescheduling. Task starvation is also a possible occurrence in speculative tasks since the previous decision might become sub-optimal and unreasonable considering the variation of cluster states. In order to avoid this scenario, the JM adopts a time-out detection to determine how long the speculative tasks are waiting within the NM queue. If the waiting time is over a finite timing-out bound, the task will be re-dispatched to machines by using the latest MMF result. This strategy can prevent the starvation and head-of-line blocking, resulting in a better utilization and load balancing among different queues. The occurrence of task stragglers can also be mitigated.

6.4 Discussion

Safeguards for Robust Oversubscription. On each node, determining a safe oversubscription ratio (OR) and a load threshold is important to regulate the degree of resource oversubscription in the cluster. For example, if the memory capacity of a machine is 10GB, we can specify a 40% OR to ensure that at most 4GB can be oversubscribed. In fact, the ratio can be customized according to their conservative strategy and preference. Suitable system parameter configuration is challenging because they are usually associated with performance-sensitive operations. One common practice based on our large-scale engineering experience is to initially set safe and conservative parameters for profiling and validation in a small-scale test system that has the same hardware configurations before deploying the system to larger-scale production. This procedure can significantly help towards understanding system behavior in a controlled manner. For example, we can start from oversubscribing 10% memory of the node capacity and allowing for at most 60% disk utilization in the threshold control. Subsequently, through weekly regression tests and tracelog collections, we can analyze the observability of performance slowdowns or system failures such as out-of-memory (OOM) or out-of-disk (OOD), etc. This procedure can help us gradually revise the configuration with a small step until all regression tests deliver stable outputs and acceptable performance.

We are also aware that configuration auto-adjustment is well-studied by a rich body of work in software engineering and system communities enabled by either control theory (CT) or machine learning (ML) [39][40]. Specifically, feedback-control provides a general set of mechanisms for ensuring that systems can achieve the desired effects in dynamic and runtime environments. ML-based approaches aim at finding optimal configurations based on exploiting historical records and exploring complex configuration space. Reinforcement learning (RL), combining both CT and ML also offers us advances and opportunities in the future to replace current conservative profiling and tuning approach. This research is currently beyond the scope of this paper and will be left for future work.

Overheads. Overall, the overheads are generally low, mainly generated from collecting and retaining per-workload and per-machine states. Workload states include performance counters and task-level state information while machine states consist of different types of load information, queue states, and the information on scheduled workloads, etc. Main overheads encompass: *(i) Disk cost:* For timely collecting and storing these states, a local InfluxDB component temporarily stores the raw time-series data. The space used for saving raw data of those states per machine per day is approximately 4.8MB, and the total space required by a cluster will scale linearly with the total machine number in the cluster. In our testbed with 210 machines, for example, the disk space used per day is roughly 1.02GB, which is acceptable in typical cluster systems. *(ii) Memory and CPU cost:* Runtime memory consumption is typically determined by the number of workloads and machines involved in the resource scheduling. ROSE keeps all the updated states at runtime and consumes roughly 128B per workload and 256B per machine. The total memory cost scales linearly with the number of workloads and machines. In our experimental example with 210 machines and 120 submitted workloads, the total memory consumption is no more than 200MB, that is, a no more than 5% increase as compared with the original cluster without ROSE. The CPU overhead is less than 3% of the computation cost imposed by the original resource management system without ROSE.

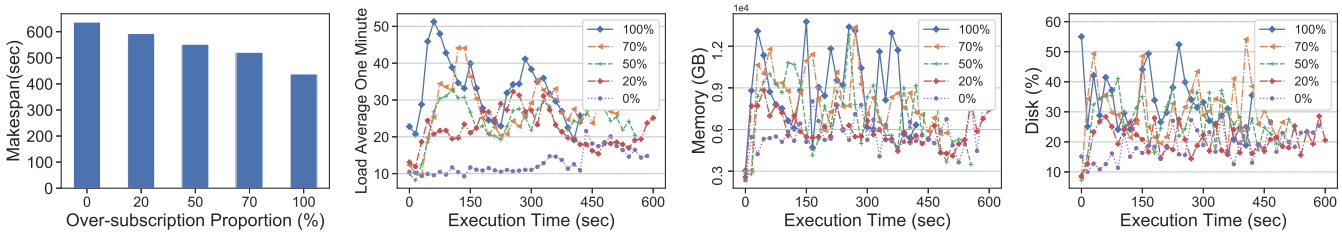


Fig. 11. The impact of varying oversubscription parameter

7 Evaluation

7.1 Experimental Setup

Environment. We implemented ROSE within Fuxi and developed a light-weight prototype of the distributed oversubscription patched in YARN. YARN evaluation was performed using a 32-machine cluster, while other evaluations were performed in a 210-machine cluster. Each machine consists of two 6-core Intel(R) Xeon(R) CPU E5-2630 processors, 82GB RAM, 12*1.9TB disk drives, and 10Gbps network.

Methodology and Baseline. The experiments are four-fold: (i) We evaluate that the distributed oversubscription architecture can effectively resolve resource mismatch in centralized oversubscription frameworks. As our proposed mechanisms in this paper are coupled with Fuxi system that is independent of open-source YARN, to carry out necessary comparisons we implemented a light-weight prototype of distributed oversubscription YARN-r that is decoupled from the centralized RM: the running AMs are developed to own oversubscription functionality and can obtain a machine list that is calculated by CA through machine filtering (detailed in Section 5.1). By these modifications, we can avoid the central RM participating in the oversubscription and speculative task scheduling. We then compare YARN-r against native YARN (YARN-n) and the centralized oversubscription method in YARN (YARN-o) [11].

(ii) We perform several micro-benchmarks to demonstrate the impact of oversubscription ratio, the efficiency/overheads of machine selection and incremental information synchronization mechanism.

(iii) We demonstrate the effectiveness of our machine selection and its impact on improved utilization and job execution particularly for batch-only job co-location. Specifically, we compare the **ROSE** method against non-oversubscription and three representative selection strategies adopted in other systems:

- **RB** (Random Based Selection): Assigns tasks by round robin and FIFO queue management. This is the comparable method of which Sparrow [41] and Apollo [13] perform resource oversubscription;
- **SLB** (System Load Based Selection): Considers real-time resource utilization when selecting candidate machines;
- **QLB** (Queue Length Based Selection): Primarily measures the queue length or waiting container size of each machine, and is adopted within Mercury [14].

(iv) We evaluate the oversubscription effectiveness in mixed co-location scenario where batch jobs and latency-sensitive LRAs co-exist. We especially measure the benefit of improved LRA responsiveness and the corresponding impacts on system utilization and batch job execution. To demonstrate the wider effects, we comprehensively compare how *utilization-prioritized (UP)* and *responsiveness-prioritized (RP)* *oversubscription* work with different machine selection and the following task placement strategies, denoted by [*strategy*]-UP and [*strategy*]-RP, e.g., ROSE-UP, SLB-RP, etc. We compare the discrepancy between *-UP and *-RP to imply the sacrifice of resource utilization and speculative amount for the guarantee of LRA performance.

TABLE 3
Comparisons – YARN-r, YARN-n and YARN-o

Group	CPUUtil	MemUtil	Makespan	#Unqueued	#SuccSpec
YARN-n	21.3%	18.5%	609s	–	–
YARN-o	38.6%	27.1%	587s	169	58
YARN-r	59.4%	49.3%	481s	0	493

Workloads. To measure the performance of batch processing workload, experiments were conducted using the mixture of jobs including *WordCount* and *TeraSort* [42] on 10GB data and a Bayesian classification algorithm in the Mahout library [43] on 1.6GB set of Wikipedia pages jobs. These are well established to benchmark cluster scheduling performance [3][44][45]. While we conduct experiments with different workload in MapReduce and Fuxi DAG [3], the concept of speculative tasks through oversubscribing resources is readily applicable to other types of jobs (e.g., Tez and Spark) and no difference will manifest if the AM is correspondingly implemented adhering to the ROSE protocol and interfaces. We use Yahoo Cloud Serving Benchmark (YCSB) benchmarks [46] to create different levels of data operations on MongoDB, a document-oriented NoSQL data store [47]. The database and benchmarks are all continuously run in docker containers. We also submit sysbench benchmarks (e.g., CPU and Memory) [48] to represent more general-purpose long-running applications.

Metrics. We mainly consider the following metrics:

- **Cluster Resource Utilization.** CPU utilization, System load avg per 1 minute[1], memory usage, and disk utilization etc. on a cluster-level and machine-level basis.
- **Job Completion Time (JCT).** End-to-end completion time for a single job, recorded from the start of job's AM execution and finished at the termination of all tasks.
- **Workload Makespan.** Holistic span-time for a batch of jobs, consisting of the accumulation of all submitted jobs.
- **Evicted and Started Task Number.** The launched tasks or evictions due to preemption.
- **LRA Performance.** Indicators such as Events per Seconds (EPS), read/write latency, throughput, etc.

7.2 YARN-r vs. YARN-o and native YARN

We submit 40 Apache Mahout ML jobs in our YARN cluster. As shown in Table 3, the average cluster CPU utilization of

1. system load avg is the average number of processes that either in a runnable or uninterruptable/waiting state. It can be collected via linux uptime command

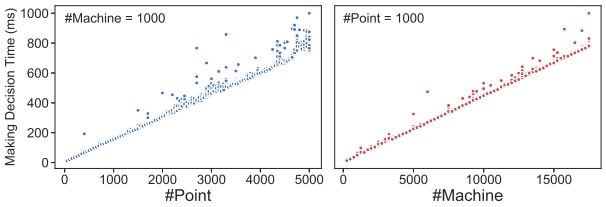


Fig. 12. Decision making time in machine selection

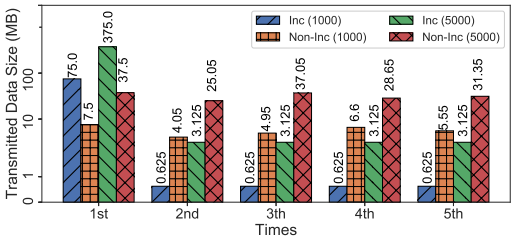


Fig. 13. Incremental vs. Non-Incremental

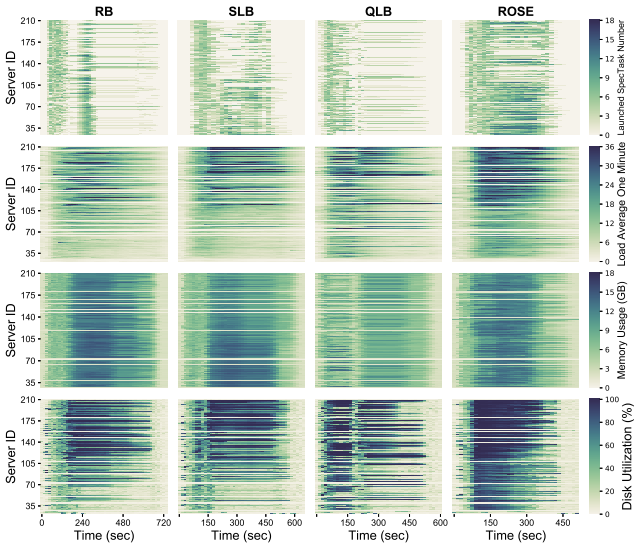


Fig. 14. Cluster resources utilization using oversubscription strategies. (Row#1: *Launched speculative task number;* Row#2: *Load Average One Minute;* Row#3: *Memory usage;* Row#4: *Disk utilization;*)

YARN-r can be increased to 59.4% in comparison to YARN-n (21.3%) and YARN-o (38.6%). Memory utilization also sees similar improvements when using YARN-r, increasing to 49.3% compared with 18.5% and 27.1%. Moreover, the overall makespan of submitted jobs can be shortened using YARN-r by 21.0% and 18.1% compared with YARN-n and YARN-o respectively. YARN-r can successfully launch 8.5× more speculative tasks than YARN-o without any exclusion from the NM's local queue. By contrast, in YARN-o, a large portion of the created speculative tasks fail to enqueue. This is because YARN-r is equipped with distributed oversubscription enabled in individual AMs and effective machine filtering mechanism, thereby effectively removing the lagged messaging in the centralized oversubscription managed by the central RM. This indicates that decoupling resource oversubscription from the centralized resource management can efficiently improve the overall system utilization and speed up the job execution.

7.3 Microbenchmark

Oversubscription Parameter Impact. We investigate the impact of oversubscription parameters that may influence the system and job performance. In a ROSE job, we can customize the proportion of tasks that can speculatively request oversubscribed resources. We adjust the ratio within a job from 0 % to 100% (where 0% indicates that no tasks within the job are allowed to use oversubscribed resources). Fig.11 demonstrates the full oversubscription (with 100% task oversubscription capability) can achieve more than 31.24% (from 637.06s to 438.04s) improvement in overall execution efficiency. Additionally, when the ratio value increases, the utilization of all resource dimensions also rises. Therefore, we preset the ratio as 100% in all experiments. In particular, when diverse workloads within different constraints are submitted by multi-tenants, the configuration can provide sufficient flexibility to satisfy various service requirements.

Machine Selection Time. As the decision making is vital for system turnover rate, we measure the overall time consumption of Alg. 4 through a large number of repetitions using our large-scale stress-testing tool. Fig. 12 demonstrates the consumed time when varying the number of machines in the cluster and the number of sampling points within a load estimation window (defined in Alg. 1). We can observe that the time is hundreds of milliseconds and is almost linearly correlative to the number of machines and sampled points, which cohere to our theoretical analysis in Section 5.3. In fact, even when the cluster size grows up to 15,000 machines, 700ms decision making time can be achieved with minor deviations, which is still acceptable in large-scale cluster management. A larger deviation begins to manifest when the number of involved sampling points surpasses 4,000, because Randomized Selection[35] used for top-K operations is more likely to experience worse cases if more sampling points are included.

TABLE 4
Median Statistics

Dimensions	RB	SLB	QLB	ROSE
CPU Load	9.739	10.704	9.263	13.902
Mem Usage(GB)	10.040	9.610	7.294	10.457
Disk Util(%)	32.727	33.870	31.814	50.956
#co-running spec task	1.64	1.87	1.31	3.18
#co-running spec task/cluster	344	392	275	668

Incremental Synchronization Impact. To demonstrate the mitigation effectiveness of flooding messages, we measure the amount of data transmission between CA and JMs after they are initialized. Due to the periodical synchronization, we mainly record the first five times synchronization under two different cluster scales (with 1,000 and 5,000 machines respectively), and compare the proposed incremental approach against the non-incremental approach. We submitted 5,000 jobs into the system and 10% of the total machines will be selected as candidates. As shown in Fig. 13, incremental synchronization needs to transmit more data for the first time but the amount will be significantly reduced in the follow-ups. For instance, for a 1,000-machines cluster, the amount can be reduced to merely 0.83% (from 75MB to 625KB). By contrast, non-incremental approach naturally maintains a stably large data transmission. We can obtain increasing benefits from our incremental design when the cluster size grows.

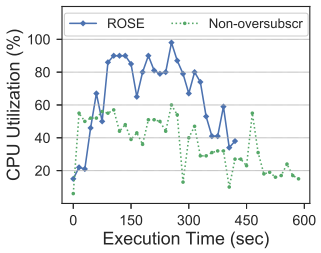


Fig. 15. Cluster CPU utilization

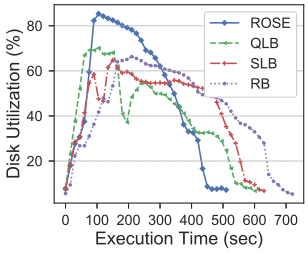
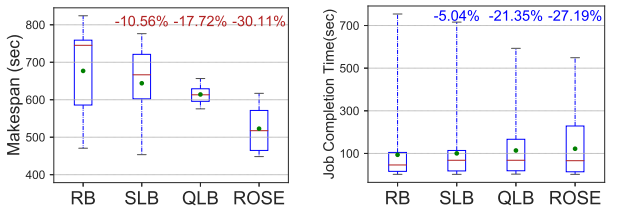


Fig. 16. Cluster disk utilization



(a) Makespan comparison (b) JCT comparison

Fig. 17. Workload execution time under different policies

7.4 Batch Only Co-location Evaluation

To emulate production workloads, we submitted 60 jobs (equal numbers for each of those three types) in each experiment run, forming 214,880 tasks in total. Each job type was configured consisting of various task scales (10*10, 100*10, 100*100, 1000*100, 1000*1000, 8000*2000) where $m*r$ represents m mappers and r reducers per job.

Resource Utilization. Fig. 15 shows that ROSE increases the cluster-level CPU utilization, achieving 65.10% on average versus 36.37% with the non-oversubscription method. Fig. 14 and Table 4 depict a heatmap and median value summary of per-machine resource utilization and the number of co-running speculative task using different oversubscription strategies. ROSE can achieve a higher CPU and disk utilization across the entire cluster. For instance, the load can be increased by 42.75% and the number of speculative task launched per machine even doubles compared with RB. Since the workloads executing within the experiment are IO-intensive, an increase in disk utilization is the most important dimension to consider. In particular, as shown in Fig. 16, ROSE can achieve an 18.23% disk utilization improvement. The memory usage does not exhibit apparent growth as the CPU and disk thresholds are firstly reached which imposes a restriction on the memory oversubscription.

All these improvements are resultant of additional speculative tasks that utilize the oversubscribed resources. In reality, by using the multiple phase machine selection, the speculative tasks within ROSE can be precisely dispatched onto machines that have sufficient capacity to execute additional workloads. Consequently, the number of running speculative tasks is almost 1.94x and 1.70x times that of RB and SLB. Accordingly, finish time is shortened due to the increased efficiency of cluster packing, as well as the reduced waiting time for tasks to be assigned oversubscribed resources. By contrast, the RB method neglects runtime resource variation and thus lacks optimal task placement. In fact, other than IO intensive workloads, ROSE naturally facilitates other types of workloads. In particular, the CPU-intensive and short tasks can be significantly enhanced by the proposed efficient resource oversubscription. This is because tasks with shorter duration will have less likelihood of eviction occurrence during their execution, resulting in efficient resource fragmentation recycling. The CPU isolation and sharing mechanism within *cgroups* can also provision more flexible approaches that complement our solution.

Job Completion Time. To accurately determine the effect of job execution, we repeatedly submit workloads 20 times. Fig. 17(a) shows the statistics of the workload execution with the maximum, 75th percentile, average(green circle), median(red line), 25th percentile and the minimum execution times depicted. In terms of the median value of all submission rounds, it is observable that SLB and QLB reduce the workload makespan by approximately 10.56% and 17.72% compared with the random-based method, while the reduction can even reach 30.11% by ROSE. Additionally, the fluctuation of workload makespan in ROSE can also be diminished compared with other approaches. Fig. 17(b) illustrates the per-job execution time by synthesizing all submitted jobs. It is observable that the execution time varies as the number of mapper and reducer changes and consequently, the submitted jobs exhibit diverse time ranges. Nevertheless, we can observe that there is an improvement for effectiveness for JCT. For example, the 75th percentile box border of ROSE increases, indicating that job execution times with small configurations (e.g. 10 mappers and 10 reducers) is shortened and aggregated within 230s. Compared against the RB method, the overall distribution of completion time shifts smaller. Specifically, the maximum of job execution time in ROSE can be reduced by approximately 27.19%. By contrast, SLB and QLB methods only result in a 5.04% and 21.35% reduction respectively. This substantial reduction within ROSE is predominantly derived from rapid task launching with oversubscribed resource and efficient machine filtering process. The threshold controller can also provision the best destination for speculative tasks, avoiding potential interference or eviction.

Rescheduling and Task Eviction Reduction. To evaluate the effect of the rescheduling approach on job execution, comparisons are conducted between RB (with rescheduling) *vs.* RB (without rescheduling) and ROSE (with rescheduling) *vs.* ROSE (without rescheduling). Fig.18 shows that RB and ROSE are capable of reducing the median makespan by approximately 12%. To evaluate the task eviction occurrence during oversubscription, Table 5 demonstrates that ROSE achieves a substantially increases speculative task number by 37.78% and 43.54% compared with RB and QLB, respectively due to more speculative tasks can be accurately launched within specific machines. Despite a large increase in speculative tasks, the eviction rate only slightly increases due to the machine filtering and threshold controller.

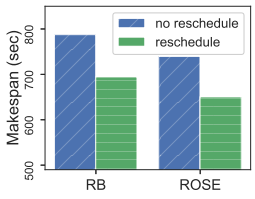


Fig. 18. Rescheduling approach

TABLE 5
Eviction Number

	#Evicted	#Started
RB	532	85411
SLB	556	111915
QLB	680	81982
ROSE	591	117676

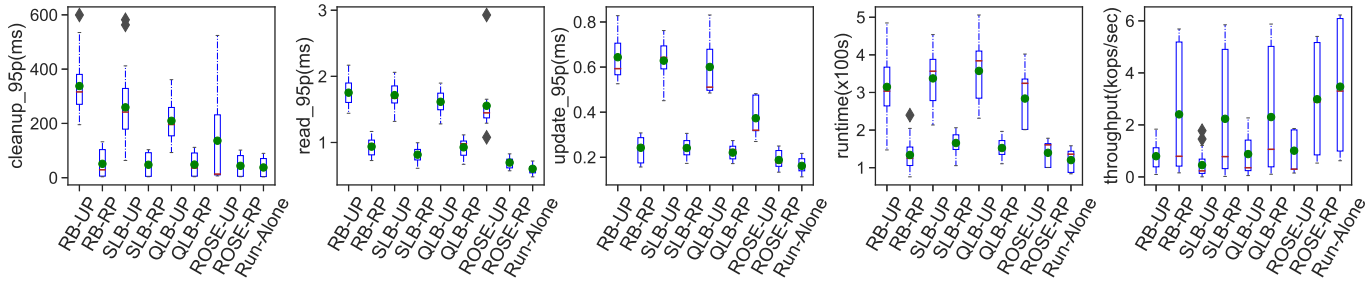


Fig. 19. Performance comparison of YCSB workloads on MongoDB

TABLE 6
*-UP vs. *-RP Comparisons (*Diff=(*-RP - *-UP)/ *-UP*)

Dimensions	RB-UP	RB-RP	*Diff*	SLB-UP	SLB-RP	*Diff*	QLB-UP	QLB-RP	*Diff*	ROSE-UP	ROSE-RP	*Diff*
avgClusterCPUUtil(%)	44.20	38.52	-12.85%	47.23	39.53	-16.30%	44.12	38.66	-12.38%	63.02	56.34	-10.60%
avgClusterDiskUtil(%)	33.32	25.93	22.18%	32.19	25.82	20.11%	30.17	24.98	17.20%	53.21	43.49	18.27%
avg #Co-running spec/node	1.44	1.25	-12.91%	1.53	1.28	-16.51%	1.24	1.10	-11.15%	2.48	2.15	-15.52%
avg #Co-running spec/all	302	263	-12.91%	321	268	-16.51%	260	231	-11.15%	521	451	-15.52%
med CPU Load	9.45	8.12	-14.07%	10.11	8.43	-16.62%	9.42	8.21	-12.85%	15.93	13.31	-19.68%
med Mem Usage(G)	10.27	8.21	-20.06%	8.69	7.19	-17.26%	7.19	6.02	-16.27%	10.92	9.21	-18.57%
med Disk Util(%)	31.79	24.41	-23.21%	31.91	24.97	-21.75%	28.49	23.6	-17.16%	49.13	40.59	-21.04%
makespan(s)	1128.7	1329.2	17.76%	1013.43	1186.1	17.04%	909.65	1081.39	18.88%	791.02	922.27	14.23%
#Evicted	402	498	23.88%	429	521	21.45%	489	561	14.72%	453	534	15.17%
#Started	64060	57232	-10.66%	77152	60378	-21.74%	61489	54393	-11.54%	79494	64995	-22.31%

7.5 Latency-sensitive Co-location Evaluation

To emulate a representative DB application, we deploy 20 instances of MongoDB cluster and an instance consists of 3 slaves and a master (each of them is run in a container). 40 YCSB containers are submitted into the system to generate query stress. MongoDB is configured to have a constant replication factor of two replicas per shard, meeting the minimum recommended production settings. A range of YCSB configurations covering different types of data operations, e.g., read-heavy, write-heavy, 95/5% breakdown of read/write and 95/5% breakdown of write/read workloads are used to stress the data stores. Record count and operation count are set to be 4M and 400k respectively. Furthermore, we also use Sysbench to represent other comprehensive workloads. Particularly, we submit a bunch of 1,200 sysbench containers and each one calculates the prime 20,000 with 4 threads, requesting for a CPU core by default. We still submit the group of 60 jobs used in Section 7.4.

Performance Guarantee for LRA. Boxplots in Fig. 19 firstly illustrate the measured performance of YCSB under different UP and RP schemes. Observably, all operational latency and running time of YCSB benchmark in RP scheme can be significantly reduced against UP scheme in which performance awareness and resource throttle are turn off. Compared with the results when the LRA runs alone, RP oversubscription only experiences slight performance degradation. For example, the median value of 95p read latency of ROSE-RP increases merely by 5.4% compared with running the LRA alone, while in ROSE-UP the value will be 2.58x times that of the running-alone LRA. Equivalently, the read latency in ROSE-RP has a 55.3% reduction compared with ROSE-UP. This is because effective resource throttling can guarantee sufficient resource provisioning to LRA and adaptively adjust the speculative tasks to avoid excessive oversubscription. We can draw similar conclusions from other 95p cleanup and update latency.

Accordingly, the median throughput of ROSE-RP can reach roughly 2.87 kops/sec, with only 12.8% performance degradation compared against running-alone LRA. By contrast, if we use ROSE-UP without prioritizing LRA's performance, the throughput will be decreased to only 9% of the baseline throughput, which is no longer tolerable. More comparisons can be easily observed from Fig. 19. We can conclude that the performance-aware RP oversubscription can effectively maintain LRA's responsiveness, and thus is very suitable for resource oversubscription among latency-sensitive workloads.

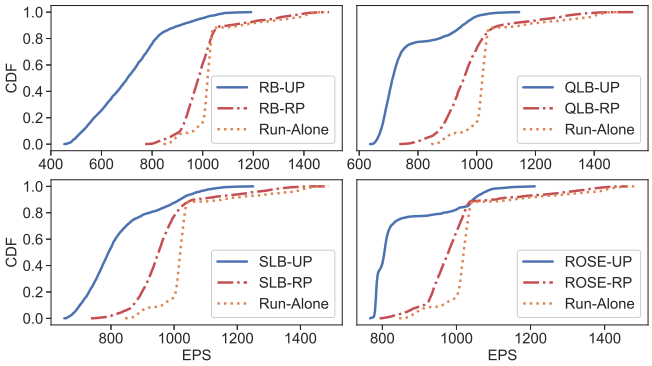


Fig. 20. Throughput comparison of sysbench containers

Fig. 20 shows the CDF of sysbench performance, measured by the built-in events per second (EPS). It is obvious that all RP schemes outperforms UP schemes. Explicitly, the curve of RP schemes is much closer to the baseline when LRAs run alone. However, the overall EPS of UP curve is far lower than the other two. More specifically, the median EPS of ROSE-RP can achieve 978.26, only with 4% degradation compared with the running-alone baseline (1019.47). However, the EPS of adopting the ROSE-UP oversubscription will drop to 805.37, which has been degraded over 20%. More specifically, roughly 85% container's EPS in ROSE-RP are between 850 to 1050, while in ROSE-UP approximately 75% container's EPS are no great than 850. All these results demonstrate the proposed performance-aware monitor and runtime resource throttling can effectively protect LRAs

from being aggressively oversubscribed by other batch jobs.

Overall Utilization and Performance Impact on Batch Jobs. Since Section 7.4 has demonstrated the utilization increase of adopting UP oversubscription against non-oversubscription, we will focus on the comparisons between UP and RP schemes, and further examine their impact on the performance of batch jobs. Table 6 outlines all comparative statistics. The overall utilization of *-UP can be maintained the same level as that of batch-only co-location. For instance, the median disk utilization per machine is 49.13%, only have a 1.83% absolute discrepancy against that (50.956%) in the batch-only test. This is because the fixed cluster capacity determines the similar saturation in the same consolidated system. Nevertheless, as demonstrated above, the LRA's responsiveness has been severely affected. Furthermore, by adopting RP oversubscription, the utilization of CPU and disk can still reach 56.34% and 43.49%, separately. Although all system metrics of *-RP exhibit decreased phenomenon against *-UP, this is not unexpected since resource oversubscription are preservably conducted in order to guarantee performance of LRAs. This indicates the necessity of adopting RP oversubscription for LRA, despite that it will slightly diminish the utilization.

Regarding the total number of launched speculative tasks, ROSE-UP and ROSE-RP can be found 32.45% and 44.77% reduction respectively compared against that in batch-only co-location. Against ROSE-UP, the number that ROSE-RP can launch is reduced by 22.31% and the evicted task number is also slightly increased by 15.17% as more speculative tasks are forced to give ways to LRAs when specific LRAs are detected suffering from performance degradation. Consequently, the makespan will be extended to 791.02s and 922.27s in ROSE-UP and ROSE-RP compared against the same jobs in batch-only co-location (517.4s). Against ROSE-UP, the makespan of ROSE-RP is extended by 4.23%. In reality, the extension in ROSE-RP can be attributed to resource contention incurred in mixed co-location, less launched speculative tasks and higher eviction frequency for giving way to LRAs. As the responsiveness protection (*-RP) can be enforced onto any other placement approaches, experiments also showcase similar comparison results.

8 Related work

Cluster scheduling. Resource management systems in shared clusters are proposed [1][2][3][4] to underpin diverse workload through resource negotiations in centralized manager. Capacity Scheduling [49] or Fairness Scheduling [50][51] are proposed to fulfill an efficient quota-based resource sharing among multiple jobs. The objective is the enforcement of scheduling invariants for heterogeneous applications, with policing/security utilized to prevent excessive resource occupation. It sacrifices efficient resource utilization for the assurance of scheduling fairness and job execution performance.

Oversubscription in virtualized environments. Overcommitting memory or CPU is a long-standing concept in the Linux kernel [9]. It allows for allocation of more resources than available, based on the assumption that processes often do not utilize applied resources fully. Conservative systems frequently leverage this concept to avoid the dangers of OOM killing and evictions. In fact, virtual resources (e.g., vCPU) are actually visible to the resource manager. The underlying over-committing mechanism in the standalone is orthogonal to the proposed cluster scheduling policy. In virtualized Cloud datacenters, resource oversubscription is a widely-used technique [8][10][52] due to the same principle. Different VMs might have differentiated QoS, resulting in the different tolerance level of being overbooked from its own resources. Available physical capacity can be dynamically adjusted for VMs that need the resource. A feedback-control approach is adopted to steer and ensure the resource re-distribution among co-located VMs depending on their tolerated oversubscription level. However, this requires proactive exploitation of per-task resource patterns, and thus cannot be directly applied into the computation-intensive scenario consisting of millions of running tasks. ROSE compacts the resource utilization by aggressively dispatching latency-agnostic tasks and reactively coordinates shared resources through QoS-aware throttling.

Oversubscription in Big Data environments. Centralized oversubscription in YARN [11], Mesos [12] and Datom [53] re-uses the resource allocation strategy used in the centralized scheduling pipeline. All oversubscription decisions are handled in the central manager (YARN RM or MesosMaster) according to current machine loads that places considerable strain on the cluster scheduler. The message including machine load level, where to launch opportunistic tasks is only piggy-backed by the heartbeat between NM, RM and AM. Afterwards, opportunistic tasks will be sent, enqueued and launched on the given NM. The the whole procedure take at least 3 heartbeat intervals, and the time will be several times longer considering the task retry and re-submission. However, resource usage changes during this time period and resource mismatch will result in the inaccurate speculative task distributing, long-time queuing, cancelling, as well as longer job completion times. By contrast, ROSE harnesses the decentralized, heartbeat-decoupled method to filter proper nodes for oversubscription, and uses the threshold control to flexibly create and control speculative tasks at any possible moment. Within decentralized schedulers, Apollo [13] introduces opportunistic scheduling to take advantage of idle resources. However, randomly selected tasks can only fill the spare capacity of compute slots and may lead to blind task dispatching. Mercury [14] adopts a hybrid scheduling to enhance cluster throughput and reduce feedback delays. These approaches rely on the precise queue delay estimation. However, many factors including code logic, processed data size and runtime resource allocation will make the execution time difficult to predict. Sparrow [41] and Hawk [20] perform random-based probing to assign tasks. However, due to limited visibility of entire cluster resources, it sacrifices scheduling quality for low-latency and is unlikely to ascertain an appropriate destination machine under high load. In comparison, ROSE can comprehensively reuse all idle resources with accurate machine filtering and task packing.

9 Conclusions and Future Work

This paper proposes ROSE, a resource management platform capable of conducting performance-aware resource oversubscription in a distributed manner. The approach

takes into account specifically latency-sensitive applications, due to the increasing importance of Quality of Services (QoS) assurance, and investigates the potential implication of co-locating them with batch jobs. Main conclusions can be summarized as follows:

Judiciously oversubscribed resources for speculative task execution often increase substantially the gain of resource efficiency. Although individual speculative task may experience a slight slowdown due to interference, the overall job completion time will be significantly reduced owing to a much higher task concurrency. The distributed and loosely-coordinated scheduling mode within each Job Manager to re-use idle resources is also becoming a necessity when handling heterogeneous workloads with diverse characteristics.

Tackling workload co-location plays an increasingly important role in resource management and job scheduling. Improving the resource utilization and guaranteeing the QoS of running applications has become a severe dilemma which requires constant efforts to balance. It is still challenging to realize innovations in terms of QoS modelling, interference-aware job scheduling and fine-grained resource management in uncertain and extra-dynamic environments.

Analysing and quantifying the elasticity and plasticity in the cloud environments can facilitate the design of the next-stage resource scheduling system. Exclusive resource plasticity indicates more stringent throttling of resource allocation to batch tasks, resulting in improved effects on QoS assurance of long running applications. However, the limitations are the break-down of resource elasticity and the resultant lower utilization. Therefore, only elaborate planning for resource re-usability and resource reservation can achieve a high-quality resource sharing with performance isolated.

Dealing with imbalance among multi-dimensional resources is another important but open research challenge in the context of resource oversbuscription. Such a problem is often formalized as a multi-objective combinatorial optimization problem. Multiple objectives would have to include maximizing the balance among different dimensions, minimizing job execution time, and maximizing resource utilization subject to constraints such as machine capacity, machine affinity preference, etc. Speculative task placement could be further formulated as an integer linear program (ILP), to be solved as an online optimization problem. However, theoretically optimal solutions are often difficult to achieve in practice due to various factors including resource reservation for load spike, and dynamic request changes in real use cases. Therefore, oversubscription-based scheduling with multi-dimensional resources in large-scale clusters will require further research effort and additional implementations.

We plan to futher study GPU oversubscription mechanism in ROSE and analyze historical tracelogs of different LRAs before employing reinforcement learning technique to automate and optimize oversubscription configurations.

ACKNOWLEDGMENT

We would like to thank Alibaba Group, particularly colleagues from Fuxi scheduling team for their collaboration. This work is supported by National Key R&D Program of China (2016YFB1000503), NSFC (61421003), the EPSRC (EP/T01461X/1) and Beijing Advanced Innovation Center for Big Data and Brain Computing (BDBC).

REFERENCES

[1] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *ACM SoCC*, 2013.

[2] 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 *USENIX NSDI*, 2011.

[3] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu, "Fuxi: a fault-tolerant resource management and job scheduling system at internet scale," in *VLDB*, 2014.

[4] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *ACM EuroSys*, 2015.

[5] Q. Liu and Z. Yu, "The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from alibaba trace," in *ACM SoCC*, 2018.

[6] 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 *ACM SoCC*, 2012.

[7] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," in *ACM ASPLOS*, 2014.

[8] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource overbooking and application profiling in shared hosting platforms," *ACM OSDI*, 2002.

[9] Overcommit. [Online]. Available: https://www.kernel.org/doc/Documentation/vm/overcommit-accounting

[10] L. Tomás and J. Tordsson, "Improving cloud infrastructure utilization through overbooking," in *ACM ICAC*, 2013.

[11] YARN Oversubscription. [Online]. Available: https://issues.apache.org/jira/browse/YARN-1011

[12] Mesos Oversubscription. [Online]. Available: https://issues.apache.org/jira/browse/MESOS-354

[13] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: scalable and coordinated scheduling for cloud-scale computing," in *USENIX OSDI*, 2014.

[14] 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 *USENIX ATC*, 2015.

[15] S. K. Barker and P. Shenoy, "Empirical evaluation of latency-sensitive application performance in the cloud," in *ACM SIGMM*, 2010.

[16] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu, "Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters," *IEEE Trans. on Services Computing*, 2016.

[17] Apache Hadoop. [Online]. Available: http://hadoop.apache.org

[18] S. Das, V. R. Narasayya, F. Li, and M. Syamala, "Cpu sharing techniques for performance isolation in multi-tenant relational database-as-a-service," *VLDB*, 2013.

[19] P. Delgado, D. Didona *et al.*, "Job-aware scheduling in eagle: Divide and stick to your probes," in *ACM SoCC*, 2016.

[20] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling," in *USENIX ATC*, 2015.

[21] Apache Tez. [Online]. Available: http://tez.apache.org/

[22] Alibaba Cluster Trace. [Online]. Available: https://github.com/alibaba/clusterdata

[23] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ACM ASPLOS*, 2013.

[24] I. S. Moreno, P. Garraghan, P. Townend, and J. Xu, "Analysis, modeling and simulation of workload patterns in a large-scale utility cloud," *IEEE Trans. on Cloud Computing*, 2014.

[25] Fluentd. [Online]. Available: https://www.fluentd.org/

[26] Apache kafka. [Online]. Available: https://kafka.apache.org

[27] J. W. Tukey, "Exploratory data analysis," 1977.

[28] M. Frigge, D. C. Hoaglin, and B. Iglewicz, "Some implementations of the boxplot," *The American Statistician*, 1989.

[29] M. Hubert and E. Vandervieren, "An adjusted boxplot for skewed distributions," *Computational Statistics & Data Analysis*, 2008.

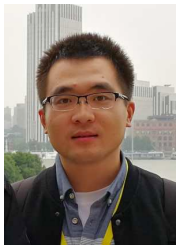
[30] P. J. Rousseeuw and M. Hubert, "Robust statistics for outlier detection," *Wiley Data Mining and Knowledge Discovery*, 2011.

[31] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi 2: Cpu performance isolation for shared compute clusters," in *ACM Eurosys*, 2013.

[32] S. Wu, S. Tao, X. Ling, H. Fan, H. Jin, and S. Ibrahim, "ishare: Balancing i/o performance isolation and disk i/o efficiency in virtualized environments," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 2, pp. 386–399, 2016.
[33] C. A. Lai, Q. Wang, J. Kimball, J. Li, J. Park, and C. Pu, "Io performance interference among consolidated n-tier applications: Sharing is better than isolation for disks," in *IEEE Cloud*, 2014.
[34] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM*, 2015.
[35] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.
[36] Linux Control Groups. [Online]. Available: http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt
[37] Intel RDT Software Package. [Online]. Available: https://github.com/intel/intel-cmt-cat
[38] J. Zhu, R. Yang, C. Hu *et al.*, "Perphon: a ml-based agent for workload co-location via performance prediction and resource inference," in *ACM SoCC*, 2019.
[39] C. T. Karamanolis, M. Karlsson, and X. Zhu, "Designing controllable computer systems." in *HotOS*, 2005.
[40] S. Wang, C. Li, H. Hoffmann, S. Lu, W. Sentosa, and A. I. Kistijantoro, "Understanding and auto-adjusting performance-sensitive configurations," in *ACM ASPLOS*, 2018.
[41] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *ACM SOSP*, 2013.
[42] O. OMalley, "Terabyte sort on apache hadoop," *Yahoo, available online at: http://sortbenchmark. org/Yahoo-Hadoop. pdf*, 2008.
[43] Apache Mahout. [Online]. Available: http://mahout.apache.org/
[44] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *ACM ISCA*, 2013.
[45] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang *et al.*, "Bigdatabench: A big data benchmark suite from internet services," in *IEEE HPCA*, 2014.
[46] Ycsb. [Online]. Available: https://github.com/brianfrankcooper/YCSB
[47] Mongodb. [Online]. Available: https://www.mongodb.com
[48] Sysbench. [Online]. Available: https://wiki.gentoo.org/wiki/Sysbench
[49] YARN Capacity Scheduler. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html
[50] YARN Fair Scheduler. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html
[51] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job scheduling for multi-user mapreduce clusters," Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, Tech. Rep., 2009.
[52] L. Tomás, E. B. Lakew, and E. Elmroth, "Service level and performance aware dynamic resource allocation in overbooked data centers," in *ACM/IEEE CCGrid*, 2016.
[53] J. Chen, C. Cao, Y. Zhang, X. Ma, H. Zhou, and C. Yang, "Improving cluster resource efficiency with oversubscription," in *IEEE COMPSAC*, 2018.

Renyu Yang is a postdoc researcher with University of Leeds and Edgetic Ltd. UK. He received BSc and PhD degree from Beihang University in 2011 and 2017. He was previously with Alibaba Group and had industrial experience in building large-scale resource scheduling systems. His research interests include reliable resource management, distributed systems and data analytics. He is a member of IEEE.

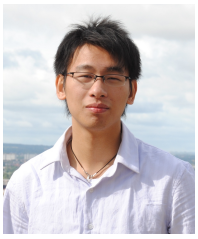
Chunming Hu is an Associate Professor and Vice Dean of the School of Computer Science and Engineering, Beihang University. He received PhD degree from Beihang University in 2006. His current research interests include distributed systems, system virtualization, data management and processing systems.



Xiaoyang Sun is currently a PhD student with University of Leeds, UK. He received MSc degree from Beihang University, China in 2018. He was previously a software engineer in Alibaba Group, working on real-time systems and large-scale cluster scheduling. His research interests include distributed systems and data analytics, etc.

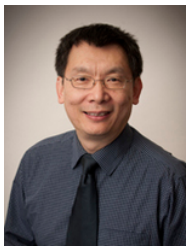
Peter Garraghan is a Lecturer in the School of Computing & Communications, Lancaster University. He has industrial experience building large-scale systems and his research interests include distributed systems, cloud datacenters, dependability, data analytics and energy-efficient computing.

Tianyu Wo is an Associate Professor with the School of Computing at Beihang University. He received his BEng and PhD Degrees both in computer science from Beihang University in 2001 and 2008 respectively. His current research interests include distributed systems, network operation systems and IoV systems. He is a member of IEEE.



Zhenyu Wen is currently a postdoc researcher at the School of Computing, Newcastle University, UK. He received M.S and Ph.D. degrees in computer science from Newcastle University, Newcastle Upon Tyne, UK in 2011 and 2015 respectively. His current research interests include multi-objects optimisation, big data processing and cloud computing.

Hao Peng is currently a Ph.D. candidate at the State Key Laboratory of Software Development Environment, and Beijing Advanced Innovation Center for Big Data and Brain Computing in Beihang University. His research interests include distributed machine learning, big data processing, urban computing, etc.



Jie Xu is Chair Professor of Computing at University of Leeds, Director of UK EPSRC WRG e-Science Centre and Chief Scientist of BDBC, Beihang University, China. He has worked in the field of dependable distributed computing for over 30 years. He is a Steering/Executive Committee member for numerous IEEE conferences including SRDS, ISORC, HASE, SOSE and is a co-founder for IEEE IC2E. He has led or co-led many research projects to the value of over $30M, and published in excess of 300 academic papers, book chapters and edited books. He is a member of IEEE.

Chao Li is Director of Engineering at Alibaba Group. He leads the teams of Cloud resource scheduling infrastructure. His research interest is distributed systems and large scale data processing techniques.