ROSE: Cluster Resource Scheduling via Speculative Over-subscription

Xiaoyang Sun[1], Chunming Hu[1], Renyu Yang[21*], Peter Garraghan[3], Tianyu Wo[1], Jie Xu[21], Jianyong Zhu[1], Chao Li[4]
[1]Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, China
[2]School of Computing, University of Leeds, UK
[3]School of Computing and Communications, Lancaster University, UK
[4]Alibaba Group, China
{*sunxy,hucm,woty,zhujy*}*@buaa.edu.cn;*{*r.yang1,j.xu*}*@leeds.ac.uk; p.garraghan@lancaster.ac.uk; chao.li@alibaba-inc.com*

***Abstract*—A long-standing challenge in cluster scheduling is to achieve a high degree of utilization of heterogeneous resources in a cluster. In practice there exists a substantial disparity between perceived and actual resource utilization. A scheduler might regard a cluster as fully utilized if a large resource request queue is present, but the actual resource utilization of the cluster can be in fact very low. This disparity results in the formation of idle resources, leading to inefficient resource usage and incurring high operational costs and an inability to provision services. In this paper we present a new cluster scheduling system, ROSE, that is based on a multi-layered scheduling architecture with an ability to over-subscribe idle resources to accommodate unfulfilled resource requests. ROSE books idle resources in a speculative manner: instead of waiting for resource allocation to be confirmed by the centralized scheduler, it requests intelligently to launch tasks within machines according to their suitability to oversubscribe resources. A threshold control with timely task rescheduling ensures fully-utilized cluster resources without generating potential task stragglers. Experimental results show that ROSE can almost double the average CPU utilization, from 36.37% to 65.10%, compared with a centralized scheduling scheme, and reduce the workload makespan by 30.11%, with an 8.23% disk utilization improvement over other scheduling strategies.**

***Index Terms*—cluster scheduling, resource management, over-subscription**

I. Introduction

Improving the resource utilization of clusters is a long-standing issue that has become increasingly important to satisfy global demand for Internet services such as web search, social networking, and machine learning applications. Various applications often exhibit diverse workload characteristics in terms of task scale and resource heterogeneity. To cope with such diverse characteristics, modern cluster management systems, e.g. those in [9][10][11][12], are designed to effectively allocate jobs onto machines, and manage various resource requirements as a unified pool of underlying resources. However, there exists a substantial disparity between resource usage requested by jobs and the actual cluster resource utilization. For instance, studies of production clusters from Twitter and Google show that typical disparities are around 53% and 40% for CPU and memory respectively [13]. Therefore, the actual CPU utilization is between 20% and 35% and the memory utilization is from 20% to 40% [14]. A scheduler might consider a cluster as being fully utilized if a large resource request queue is present, even when the actual resource utilization of cluster machines is in fact very low. This disparity results in the formation of idle resources, producing inefficient cluster resource usage and incurring higher operational costs and an inability to provision service. Additionally, computation-intensive batch DAG jobs are increasingly common within data-parallel clusters and should be handled properly. These batch jobs are typically segmented into a large number of tasks which only last sub-second or seconds duration [11][15].

The resource utilization of a cluster may be improved through over-subscription (also known as overbooking) [16], that enables jobs to be executed within a cluster by leveraging resources from existing jobs that are presently underused or idle. This technique has been heavily exploited at different levels of a cluster, including the kernel [8], the hypervisor [17], and the cluster resource scheduler. For example, the resource scheduler would launch *speculative* tasks for existing jobs, where the tasks use the over-subscribed resources and run in a best-effort manner.

A centralized resource scheduler, such as YARN and Mesos, performs over-subscription decision making [4] [5] through a central manager that transmits the information about updated idle or revocable resources to a job via piggybacking on regular heartbeat messages within the cluster. However, several heartbeat messages are required for transmission of load information, resource scheduling, and dispatching of speculative tasks. The duration of this transmission process becomes even longer when task retry and re-submission are needed. As heartbeats are typically configured to be sent at 3s intervals [9], it is likely that these instructions will no longer reflect nor match the current cluster resource usage. The reason for this mismatch in cluster resource usage is primarily due to creation of potentially thousands of new tasks, diverse resource consumption patterns of existing tasks, and tasks that have second or even sub-second completion times. This consequently results in sub-optimal reduction of overall job makespan that could be significantly improved. A decentralized scheduler may resolve some of the issues by assigning speculative tasks randomly [18] or on per application

* Dr. Renyu Yang is the corresponding author.

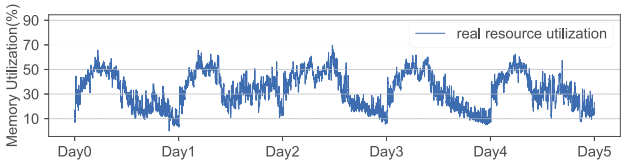


Fig. 1. Daily memory utilization of a 10,000 node cluster

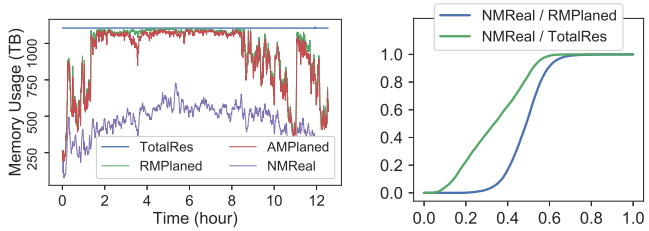
basis [19]. However, this approach is highly dependent on accurate queue delay times, which requires customers to provide precise job completion times. It is unfortunately infeasible in practice when considering the lack of customer knowledge and unknown job types. What is needed is an over-subscription approach that is capable of overcoming the late delivery of idle resources for speculative tasks, as well as exploiting cluster diversity in terms of heterogeneous resources and dynamic resource usage.

In this paper we propose ROSE, a resource over-subscription framework that provides an efficient and speculative Resource Over-subscribing SchedulEr for cluster resource management, which improves utilization whilst reducing job end-to-end time. A ROSE job leverages idle resources directly from the node controller daemon situated within every machine and creates speculative tasks, instead of waiting for resources to be made available from the centralized resource manager. These speculative tasks are then launched within machines that are determined to be most suitable for resource over-subscription through a multi-phase filtering process, considering estimated load, correlative workload performance, and queue states. Such information is incrementally and timely synchronized to the application-level scheduler. A threshold control is used to determine and control launching speculative tasks, thereby allowing for globally maximizing the reusability of cluster resources. Furthermore, task rescheduling is also employed to reduce the head of line blocking and resultant straggler manifestation [20] within speculative tasks. ROSE can be integrated into any multi-layer cluster system, and is capable of over-subscribing multi-dimensional resources (e.g., CPU, memory, network, etc.). We implemented and evaluated ROSE within the open-source resource management system Yarn[1] and Alibaba's cluster management system – Fuxi [11] – in order to study improvements against comparative over-subscribing strategies. The main contributions of this work are:

- A general resource over-subscribing architecture for cluster resource management that enables idle resources to be leveraged to create speculative tasks.
- A multi-phase machine filtering process that considers diverse factors (e.g., task-level rating and machine states) to locate optimal machines for speculative execution.
- A runtime threshold control with timely task rescheduling that provides speculative task management.

II. Resource Utilization Issues

Centralized scheduling framework. Modern resource 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[1] and Fuxi[11] share the following components: *Resource Manager (RM)* is the centralized resource controller; tracks resource usage, node liveness, enforces allocation invariants; and arbitrates contention among tenants. The component is also responsible for negotiation between the available resources within the infrastructure and the resource requests from Application Masters. *Application Master(AM)* is a 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 the execution. *Node Manager(NM)* is a daemon process within each machine and responsible for managing local tasks (including launch, suspend, kill, etc.).



(a) Cluster memory usage (b) Cluster memory CDF

Fig. 2. Breakdown of total cluster memory usage

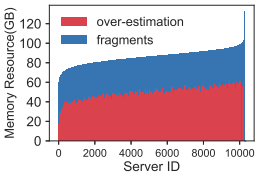


Fig. 3. Idle resources

TABLE I
Centralized over-subscription

mQL	#Unqueued	#Resched	#Succ
10	857	856	215
15	804	890	227
20	248	1440	234
50	24	1664	237
70	20	1685	253
avg	390.6	1307	233.2
%	19.5%	65.3%	11.7%

Production-level cluster resource usage. Logical resource utilization is a metric often studied to measure scheduler performance. Higher utilization implies more efficient scheduler decision making and faster job completion. We profiled these characteristics by exploring the resource behavior of a production-level cluster to ascertain idle resource occurrence. Due to cluster's cyclical behavior, we randomly select a consecutive 5-days period and analyze over 10,000 machines at Alibaba to study daily usage patterns. The metrics captured include *TotalResource* that represents the total available resource (referring to memory for demonstration), *RMPlanned* indicating the total amount of assigned memory to all AMs (job managers) after resource assignment, *AMPlanned* representing the amount that is obtained and used by all AMs, and *NMReal* showing the total resource consumed by all containers at runtime.

As shown in Fig. 1, the mean memory utilization of the cluster is 30%, with a deviation of 14.2%. Furthermore, a daily temporal usage pattern can be visually observed in the cluster due to conducting large-scale batch processing between 0:00 to 08:00 (CST). There exist sufficient idle cluster resources that can be reused. Fig. 2(a) shows the memory consumption of the entire cluster within a 12-hour period. Approximately 97.1% of memory on average can be reached by *RMPlanned*,

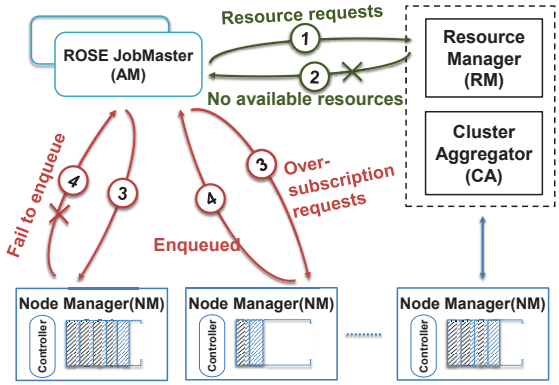


Fig. 4. Workflow and inter-component interaction

indicating that most memory is visible and initially utilized by the scheduler. Moreover, *AMPlanned* is extremely close to *RMPlanned*, indicating that the majority of job requests within the cluster are eventually confirmed with corresponding resources granted. Nevertheless, a large disparity between the *RMPlanned* and *NMReal* can be perceived, resulting in actual resource utilization being very low. Fig. 2(b) reveals that over 80% sampled data points experience no more than 55% actual memory usage compared with total assigned memory.

Overestimation & fragmentation. We further discover that this under-utilization is resultant of user overestimation and resource fragmentation. For example, at time point (27,500s), we collected the total amount reserved by all running jobs and calculated the fragmentation on each machine across the cluster. Fig. 3 depicts statistics ordered by the resource amount, showing that more than 80GB memory could be reused within most machines (with 132GB memory per server). In fact, users typically request excessive amounts of resources to handle workload bursting to avoid SLA (Service Level Agreement) violation. Fragmentation is also a common occurrence in clusters, especially during periods of high resource requests. From our analysis, there exists a significant gap between actual and predicted resource allocation that remains unused.

Resource mismatch. To reuse such idle resources, centralized over-subscription scheduling (e.g., Hadoop 3.0 [4]) is proposed. To demonstrate the consequential limitations, we submit 2,000 speculative tasks to a Hadoop 3.0 cluster and observe the launching number, varying the customizable max queue length(*mQL*) on NMs. However, the decision and heartbeat piggybacking mechanism will result in the delayed message delivery and mismatch of the latest resource usage during heartbeat intervals and lead to a great number of task redistributions. For example, on average 19.5% of submitted tasks are excluded from the queue and 65.3% have to be rescheduled even if they are allowed to enqueue. Merely 11.7% on average can be successfully launched. Additionally, 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. Such inefficiency greatly degrades the performance of resource over-subscription and job execution.

Challenges & requirements. A challenge for scheduling computation-intensive batch jobs is reducing the probability of over-subscription violation and the consequent compensation (such as rescheduling or evictions of speculative tasks) in order to improve job end-to-end performance and system utilization. Another challenge is how to timely detect and exploit cluster diversity in terms of heterogeneous resources and dynamic resource usage, and how to realize appropriate speculative task execution using idle resources optimally. To address these challenges, the first objective is **[R1]** to design a generalized resource over-subscribing mechanism that can effectively create speculative tasks compatible with established centralized resource management pipeline; The mechanism should effectively find and timely deliver the idle resource information for launching such tasks. Additionally, the system should **[R2]** fully exploit the dynamicity of the cluster, as well as workload/server heterogeneity [13][21] to reduce task eviction occurrence, and harness idle cluster resources including fragmented resources and allocated (yet idle) resources. To underpin the efficient resource discovery and optimal determination of task dispatching, resource scheduling should also **[R3]** aggregate and consider both application-level and multi-dimensional system information; and flexibly manage the life-cycle of speculative tasks on specific machines.

III. Resource-Oversubscription Based Scheduling

A. Oversubscription-based Scheduling

To improve the scalability and deal with the heartbeat-dependent issues in the centralized scheduling, decisions for over-subscription and speculative task launching are made in each job application master independently. We enable speculative tasks to be launched and executed via leveraging idle resources through the following steps:

(Step 1) A ROSE job requests resources from the RM. (Step 2) Once the resources in the cluster have been allocated, no further regular resources are assigned to jobs. (Step 3) A job attempts to request additional resources directly from NMs in a speculative manner, rather than waiting for the emergence of available resources released by the RM. The job then requests to launch speculative tasks in machines that are determined by the Cluster Aggregator (CA) to be most suitable to oversubscribe resources. (Step 4) To avoid inter-task performance interference, speculative tasks run at lower priorities and are preemptable compared to currently executing tasks in the machine. NM maintains a local waiting queue to maintain the order of submitted speculative tasks while the NM judges whether the speculative tasks can be accepted through a controller according to runtime system information. For example, as long as the maximal utilized resource does not surpass the upper-bound threshold, additional workload can be launched onto the physical machine. (Step 4) Once accepted, the speculative tasks will be enqueued and wait to be scheduled by the threshold controller, otherwise, these attempts will continue periodically.

It is worth noting that the procedure of resource over-subscription to speculative tasks is decoupled from the central-

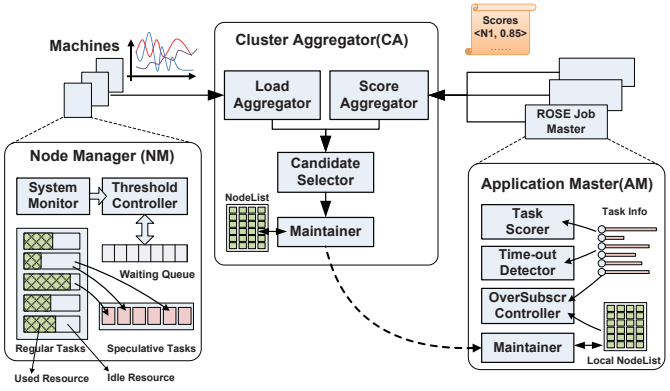


Fig. 5. Architecture and design

ized resource manager and dispersed into separate job masters. In reality, the 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 can be fully utilized by the speculative task, and the task instance can be immediately scheduled to execution machines. Consequently, these instances can run in advance, greatly accelerating the running process. Although there is a risk that such tasks may be killed, holistically it is still more effective at shortening job duration as well as utilizing idle cluster resources.

B. ROSE Architecture Overview

To achieve **[R1]**, we propose the architecture to not only guarantee that sufficient resources can be obtained by specific jobs through centralized resource management, but also reuses uncollected resources by executing speculative tasks. ROSE is designed to be complementary and compatible to existing protocols between the AM and RM, and thus can enhance existing two-layer scheduling systems. Generally speaking, to prevent excessive resource over-subscription and the resultant performance degradation, we design a multi-level threshold controller at runtime within each NM to determine the timing of launching speculative tasks and to limit the over-subscription amount and running number of speculative tasks. To maximize the effectiveness of resource over-subscription, ROSE leverages a machine selection mechanism to select candidate destinations for tasks.

Fig. 5 illustrates an overview of the ROSE system architecture, which contains of two scheduling modes in *AM*. The first mode is to request resources from the centralized resource manager and then launch relevant tasks or make a resource re-allocation, which can guarantee that dispatched tasks can be executed immediately without resource conflicts. The second mode is the over-subscription mode to proactively negotiate with one *NM* whether it can accept speculative tasks. To support this, *AM* locally maintains a replica of candidate machine collection that is periodically coordinated and incrementally synchronized by the maintainer in the Cluster Aggregator. Even in case of long task starvation due to queuing or eviction, AM in ROSE can instantly re-submit the speculative tasks according to the maintained machine collection. AM adapts

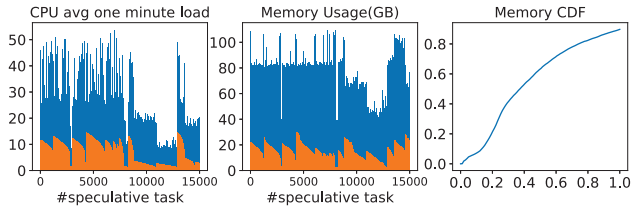


Fig. 6. For each speculative task: the load on the current scheduled machine and the minimum load(defined as optimal) in the cluster at that moment

a selection process to realize task placement in the event of a large task number centering in a minority of machines. The ROSE client can decide whether to turn on distributed scheduling mode and customize the ratio of tasks that are allowed for over-subscription per application.

To achieve **[R2]** and **[R3]**, *Cluster Aggregator (CA)* is designed to be a module decoupled with RM, to monitor runtime information and aggregate the task-level status. CA collects machine load from each *NM* and aggregates each machine score based on the estimation from each *AM* (Section IV-A). Based on this information, CA can determine a set of candidate machines in a certain time period and then AM launches speculative tasks onto machines according to the results. In CA, we use a multi-phase machine filtering mechanism to select and rank machine candidates prior to oversubscribing resources, taking into consideration machine load, correlative workload performance, and queue states for decision-making (Section IV-B).

ROSE Job Master is a specific *AM* that leverages the proposed over-subscription mechanism to compensate under-utilized resource requests. First, the Task Scorer component is used to rate machines based on the internal state and the state transitions of all tasks within the job. Therefore, the task-level scoring becomes a valuable criterion during the machine filtering process within the CA. Additionally, as the main actor and beneficiary of the resource over-subscription, AM coordinates between the regular resource scheduling and over-subscription strategy and adaptively upgrades the resource priority/level via the Over-subscription Controller when resources are granted or preempted. Due to variations in cluster states, it is possible that the scheduling decision making might become sub-optimal. For example, some tasks have already been dispatched to a specific machine but experience head of line blocking. To this end, a timing-out re-scheduling might be triggered to mitigate the task starvation by the Timeout Detector.

Apart from the regular task management, the queue management to order and control speculative tasks must be imposed in the NM. Tasks from different AMs are enqueued and wait to be launched. This is triggered by the node controller that determines if the node capacity allows for further oversubscribing and which speculative task can be launched based on real-time queue status and machine loads. This is performed by: a) component monitoring to capture real-time metrics and report to the CA (Section IV-A), and b) threshold controller responsible for process management and runtime resource isolation (Section V-B).

IV. Load Aggregation and Multi-Phase Filtering

We firstly demonstrate that the dispatched speculative tasks can improve cluster resource utilization efficiency. We conduct experiments to investigate whether speculative tasks are assigned to machines in an optimal manner. Each task is expected to be assigned onto machines with the least load for the sake of load-balancing. To quantify this desired outcome, we profile all speculative tasks by contrasting the utilization of the current machine where the task is running against that of the lowest utilized machine across the cluster in the same measurement interval. Fig. 6 shows the optimal and actual value of different load dimensions (CPU, memory, load and disk utilization), with the ideal CPU value (orange segment) merely accounting for no more than 50% of tasks are optimally allocated to machines in terms of utilization. In terms of memory, this gap is even larger, indicating that the task allocation strategy can be improved substantially. The CDF shown in Fig. 6 also demonstrates that more than 62% tasks can be placed onto better machines with at least half less loads. Therefore, selecting suitable machines is highly preferable to enhance cluster utilization efficiency.

A. Runtime Load Monitor and Aggregation

Load Monitor. To support precise machine selection at runtime while measuring the resource utilization and cluster health, a load monitor is proposed by capturing several machine metrics for each machine. For instance, we monitor the resource utilization(*cpu_util*, *memory_util*, *disk_util*, and network received/transmit) and waiting/running container numbers for both regular and speculative containers. The monitor is currently implemented in the NM as a submodule and can be deployed as a underlying service for operational management. Considering the data volume of the monitored data and the resultant network pressure, we aggregate the load information during a fixed time interval X seconds and periodically collect and transmit it every Y seconds wherein X and Y indicate the tradeoff between the data precision and monitor overhead. Users can manually configure these parameters according to machine bandwidth or other requirements. We also have to depict the approximate load level based on the accumulated historical data and the value will be utilized as a metric in the decision making.

Scalability Considerations. The rate of data generation will result in scalability issues for data processing and subsequent task allocation. For example, if we set X to be 2s, there will be 30 records within a minute, and over 300,000 pieces of data in a 10,000-node cluster that must be collected and processed to perform scheduling. It is still extremely challenging to timely provide an estimated approximation of load for each machine during a specific time frame, particularly when the fluctuations of demand and workload frequently changes over time. To solve this problem, we propose a piecewise-based algorithm and de-noising load acquisition method to accelerate the runtime load evaluation without substantial precision degradation. The NM will locally conduct the calculation before sending the intermediate results to the CA. The stress of transmission and calculations can be significantly mitigated by the approximated estimation and decentralized design.

Algorithm 1 Load Level Approximation (LLA)

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

1: **if** D is monotonous **then** :
2: **return** the last element of D
3: **if** $card(D) < k$ **then** :
4: **return** $eliminateOutlierMean(D)$
5: **let** $S_i \leftarrow \varnothing, \forall i \in [1, k]$
6: **for** each $d \in D$ **do**
7: $S_i \leftarrow S_i \cup \{d\}$
8: **if** $card(S_i) \geq \lceil card(D)/k \rceil$
9: $i = i + 1$
10: $D' \leftarrow \{eliminateOutlierMean(Si) | \forall i \in [1, k]\}$
11: **if** D' is monotonous **then** :
12: **return** the last element of D'
13: **else** :
14: **return** $eliminateOutlierMean(D')$

Function : $eliminateOutlierMean(I)$
Input: I – several continuous sample data;
Output: m – mean with eliminating outliers;

1: Q_1 and Q_3 are the lower and the upper quartiles respectively
2: $T \leftarrow \{d | d \in [Q_1 - k(Q_3 - Q_1), Q_3 + k(Q_3 - Q_1)]\}$ (customarily with k is 1.5)
3: **return** $m \leftarrow \frac{\sum T}{card(T)}$

Load Estimation. Algorithm 1 depicts the load level estimation based on the prior monitoring information and de-noising process. We define the basic calculation unit as conducted over a slide window with fixed number of data points. To accelerate the estimation, we divide the entire time period into k segments alongside the timeline (Line 1-5) and the calculation can be parallelized on different segments (Line 6). For each segment, we calculate the average load by de-noising the sample data through a calculation ($eliminateOutlierMean$) based on Tukey's boxplot [22] [23] to pinpoint and eliminate potential outliers described in Alg. 1. Moreover, ROSE can be compatible with other statistical methods of outlier detection[24] [25] to handle the data with asymmetric distribution. It is noteworthy that all segments can conduct the average evaluation in parallel. After the local calculation, we can obtain the value set D' containing the k average values. Subsequently if the elements are monotonous, we can easily determine the target value according to the tendency (Line 12). Otherwise, we regard the mean value with eliminating outliers of D' as the estimated result (Line 14). Due to parallelism and approximation, the load can be efficiently calculated. Compared with the decision making based on loads at a single measured instance, load estimation based on recent slide window can more precisely capture the load variation and obtain suitable machines.

B. Multi-Phase Machine 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

Algorithm 2 Task Behavior Aware Machine Rating (TBAMR)

Input: AM – terminated jobs in a specific time period, AM_i represents the i-th job's application master ;
$Task_{ij}$ – a set of all tasks, $j \in \{1, \dots N_i\}$ represents j^{th} tasks of AM_i;
Output: $Score$ – a synthetic score for all machines

1: **for** each $AM_i \in AM$ **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: $penaltyScore' \leftarrow topK(penaltyScore)$
8: **for** each $ps_i \in penaltyScore'$ **do**
9: $Score_i \leftarrow Score_i - ps_i$
10: **return** $Score$

Algorithm 3 Multi-phase Machine Filtering (MMF)

Input: $(M, AM, Task, mL)$
Output: C – candidate machines fit for oversubscribed resources;

1: $\boldsymbol{LM} \leftarrow \{\mathbf{LLA}(M_i).normalize() | \forall M_i \in M\}$
2: $MachScore \leftarrow \mathbf{TBAMR}(AM, Task).asendSortByScores$
3: $B_1 \leftarrow \{M_i | MS_i \in topK(MachScore)\ \}$
4: $B_2 \leftarrow \{M_i | M_i \in M \text{ and } M_i \text{ is disconnected}\}$
5: $B_3 \leftarrow \{M_i | (\exists j)(\ LM_{ij} \geq \text{the } j^{th} threshold)\}$
6: $M' \leftarrow M - B_1 \cup B_2 \cup B_3$
7: **let** $candidateInfo \leftarrow \varnothing$
8: **for** each $M_i \in M'$ **do**
9: $lIndex \leftarrow \overrightarrow{LM_i^l} \cdot \overrightarrow{loadFilter}$
10: $qIndex \leftarrow \overrightarrow{LM_i^q} \cdot \overrightarrow{queueFilter}$
11: $candidateInfo \leftarrow candidateInfo \cup (M_i, lIndex, qIndex)$
12: $\mathcal{C}' \leftarrow candidateInfo.ascendSortBy(lIndex).topK(d * mL)$
13: $\mathcal{C}'' \leftarrow \mathcal{C}'.ascendSortBy(qIndex).topK(mL)$
14: $C = \{m | \exists(l) \exists(q)(m, l, q) \in \mathcal{C}''\}$
15: **return** C

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 AMs. Eventually, each AM reports the perceived machine scores to the CA, using the detailed pseudo-code as shown in Alg. 2. The procedure is an important step that will be integrated into the machine selection (Alg. 3, Line 2).

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. In particular, the mechanism considers runtime load, task-level machine performance, and the queue status of each machine. With these multiple phases, we can take advantage of both the load-balanced and minimized-queuing, thereby mitigating the head of line blocking problem. Alg. 3 depicts the core phases:

a) Blacklisting from perspective of task-level assessment, timing-out machine detection and runtime resource alerting. The cluster aggregator will converge the evaluation scores of all machines from completed jobs by utilizing Alg. 2. The lowest K machines will be marked as weak performance machines which have a higher likelihood of being removed from future scheduling (Line 2-3). In addition to the low scoring machines, temporarily timing-out machines are also eliminated from task placement (Line 4). Additionally, the latest machine load status is also gathered with overloaded machines filtered out once any monitoring dimension(such as system loads, max-queue-length, max-container-number, etc.) surpasses the pre-defined threshold (Line 5). This is performed through the multi-level threshold controller in NM(Section V-B).

b) Machine selection based on multi-resource load. In order to comprehensively consider various resource dimensions, ROSE leverages a modified tetris algorithm [26] to calculate where speculative tasks should be packed. The method predominantly comprises task packing as an optimal projecting problem within a multi-dimensional euclidean space. The objective is to maximize the packing efficiency when considering the scheduler's weighted preference for each resource dimension and the used resource amount (i.e. current system load) of each candidate machine. 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 $\overrightarrow{loadFilter}$ can be regarded as a *load index* to imply the accumulated level over the corresponding dimensions($\overrightarrow{LM_i^l}$) and the approximate load level according to the system's administration.

Specifically, within the procedure of implementation, to ensure the same numerical range, we have to 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 $\boldsymbol{LM}$ (Line 1). In reality, the dot product prefers to place tasks onto light-loaded machines whose available resource is much more sufficient than others. Compared to the state-of-the-art DRF [27] multi-resource selection approach, the proposed method can enable more compacted workload packing especially when the tasks' requirements are heterogeneous. The overall resource utilization can be promoted accordingly. In this context, we calculate and order the load index of all

TABLE II
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 $\overrightarrow{LM_i^q}$ 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., $[rc\_am, oc\_w\_am, oc\_r\_am]^\intercal$
mL	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

machines (Line 8-9) and filter out a set of machines which most fit for over-subscription (Line 12).

c) 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, running task numbers are also profiled periodically. For example, apart from the regularly running task containers, the currently running and waiting number of the speculative tasks are recorded in the $\boldsymbol{LM}$. Similarly, an index of queuing status is calculated by the dot product operator (Line 10). In this manner, the 3-tuple information for each machine $(M_i, lIndex, qIndex)$ can be obtained and aggregated (Line 11). On the basis of the load pre-filtering, the final selection phase is to determine the mL machines that is prepared for those waiting AMs to efficiently and accurately leverage idle resources (Line 13-14).

C. Parameter Setting-up

The interval of load collection X highly depends on the performance monitor techniques adopted in the distributed platform. A larger X will result in greater precision loss of load estimation. Thus the number should be minimized as long as the monitor overhead is affordable. At present, we configure X to be 1s or 2s since the paralleled load approximation can effectively cope with the scalability issues.

The interval of load aggregation Y from each NM to CA is another parameter that determines the frequency of mutiphase machine filtering in CA and triggers the dispatching of speculative tasks in AM. If Y grows too large, the machine filtering will exhibit random selection, which dramatically decreases the ROSE's effectiveness. In fact, the basic principle of setting Y is to cover the trend of load change and ensure the timely update of machine list in CA. We also discovered that this value positively correlates to the turnover and throughput of speculative tasks. Therefore, according to our experience, Y was configured to be 10s, and can be dynamically adjusted when workloads change.

V. ROSE Job Scheduling and Threshold Control

A. ROSE Job Scheduling

Job Scheduling with Task Upgrade. In the event of unfulfilled resource requests due to insufficient resource, the AM will proactively dispatch several speculative tasks onto NMs according to the latest result of MMF procedure (Alg.4 Line 1-2). The AM also tracks all launched tasks until their completion. Once the waiting resource is approved and the corresponding request can be further fulfilled, the AM determines whether a speculative task has been launched and which machine it is executing within. If the available resources are assigned to a resource request that is served by an over-subscribed 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. Thus, if the launched task has already been executed on the same machine, ROSE will directly relabel it as a regular task (Line 5-6). Otherwise, ROSE judge if the it is cost effective to evict the speculative task based on the execution progress. If the task is started with a progress no greater than a provider defined threshold(e.g., 60%), the task is killed and launched using new resources (Line 7-9). If the progress surpasses the threshold (indicating a near-completed task), we retain the task and also reserve the recently approved resource just in case of the speculative task failure (Line 11-12). 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 cluster task eviction and shortening overall job makespan.

Algorithm 4 ROSE Job Scheduling

```
1: if requests cannot be satisfied then
2:    launchSpeculativeTask(MMF(M, AM, Task, mL))
3: if waiting resource is approved then
4:    taskLoc, resLoc ← where(task, resource)
5:    if isSamePlace(taskloc, resloc) then
6:         upgradeTask(task)
7:    else if task progress is less than τ then
8:         killSpeculativeTask(task)
9:         startRegularTask(resource)
10:   else
11:        keepSpeculativeTask(task)
12:        reserveRes(resource)
```

Task Rescheduling. Task starvation is also a common occurrence in the 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 AM 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.

B. Runtime Threshold Control

Threshold Control. We extend NM to realize the threshold control of physical resource and queue management. Firstly, multi-resource restrictions are imposed during the execution of speculative tasks. An over-subscription ratio (OR) is used to regulate the degree of resource over-subscription in the cluster to avoid using excessive resources. 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 over-subscribed. In fact, the ratio can be tuned by system administrators according to their aggressive or conservative strategy and preference. The value can also be learned from experiences based on continuous monitoring and profiling. In this manner, the over-estimated and fragmented resources can be aggregated and reused for

launching tasks. It can also avoid the performance interference with regular tasks as much as possible in the scheduling. Secondly, based on runtime system information, a controller manages the whole life-cycle of speculative tasks including task enqueue permission, execution start time, resource allocation, task preemption with priorities, etc. For instance, the controller determines if a speculative task can be started if all resource metrics are within the limits of upper bounds and the overall over-subscription quota does not surpass the maximum threshold. Once any dimension is beyond the corresponding threshold value, the over-subscription will be temporarily blocked until the launching condition is satisfied. This also creates the opportunity to configure a certain resource requirement according to user's or system administrator's demand. The administrator should also have a specified configuration to consider machine heterogeneity. When the regular tasks request to revoke resources, task preemption will occur. As a result, our proposed machine selection can facilitate the speculative tasks running on reasonable machines where such preemption infrequently occurs.

Resource Isolation. In current process management, we leverage *cgroup* [6] and its sub-systems such as *blkio, cpu, memory, net_cls* to enforce the IO, CPU, memory and network isolation and control for over-subscription. 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 over-subscription 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 over-subscribed and regular group, respectively. Each group manages its own priority and tasks within the over-subscribed group have lower priorities than those in the regular group. This mechanism can ensure the speculative tasks are preempted by higher prioritized tasks. In regards to applicability, the architecture and approaches of ROSE have been designed so that it can accommodate LXC[7], Docker[28] which can provide further isolation.

C. Discussion

ROSE is particularly effective and suitable for under-utilized computing systems where offline(such as batch processing) jobs are the dominant type of workload. ROSE is designed on the assumption that idle resources can be heavily exploited as long as the resource thresholds are not met. As batch processing workloads are less sensitive to QoS(e.g., response latency), they can tolerate a transient reduction in resource requirements and can be speculatively used by co-located tasks due to resource sharing or evicted tasks due to resouce preemption and return. However, for online jobs, resources should be strictly over-provisioned to ensure sufficient resources in the event of any burst-type behavior. Due to the lack of a kernel-level feedback control mechanism to capture transient QoS fluctuation, ROSE cannot fully exploit the idle resources of such online jobs whilst providing a high-precision control of end-to-end latency. Therefore how to realize the QoS control and corrections in a dynamic environment remains a challenging topic in ROSE, as well as other resource managers. We are currently extending ROSE to deal with safe co-location of both computation-intensive and latency-sensitive workloads by using QoS profiling and prediction, and runtime PiPo [29] with rapid failover [30] to precisely control the batch execution without causing severe QoS degradation. The threshold controller can carry out corrections from the QoS Controller based on the performance of running tasks.

TABLE III
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

VI. EVALUATION

A. Experimental Setup

Environment. To illustrate the general applicability, we implemented ROSE within both Fuxi and Yarn to demonstrate improvements in cluster resource utilization and job makespan compared to other resource over-subscription strategies. Evaluation of ROSE's effectiveness in Yarn was performed using a 32-machine cluster, while comparisons against other over-subscription strategies 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. Furthermore, we also implemented ROSE into a 4,600-machine cluster in Alibaba.

Baselines and Methodology. We compare the ROSE mechanism in Yarn (Yarn-r) against native Yarn (Yarn-n) and the centralized over-subscription method in Yarn (Yarn-o). Furthermore, we also compare ROSE against non-over-subscription and three over-subscription scheduling strategies adopted in other systems: 1) **RB** (Random Based method): Assigns tasks by round robin and FIFO queue management. This is the comparable method of which Sparrow[31] and Apollo[18] perform resource over-subscription; 2) **SLB** (System Load Based method): Considers real-time resource utilization when selecting candidate machines; 3) **QLB** (Queue Length Based method): Primarily measures the queue length or waiting container size of each machine, and is adopted within Mercury[19]. A series of micro-benchmarks were performed to demonstrate the detailed benefits gained from adopting the ROSE design. These metrics consider:

- **Cluster Resource Utilization.** Average CPU utilization, CPU load per minute, memory usage, and disk utilization etc. on a cluster-level and node-level basis.
- **Job Completion Time (JCT).** End-to-end completion time for a single job, recorded from the start of job AM execution and finished at the termination of all tasks.
- **Workload Makespan.** The holistic span-time for a batch of jobs, consisting of the accumulation of all submitted jobs.
- **Task Eviction Number.** Evictions due to preemption.

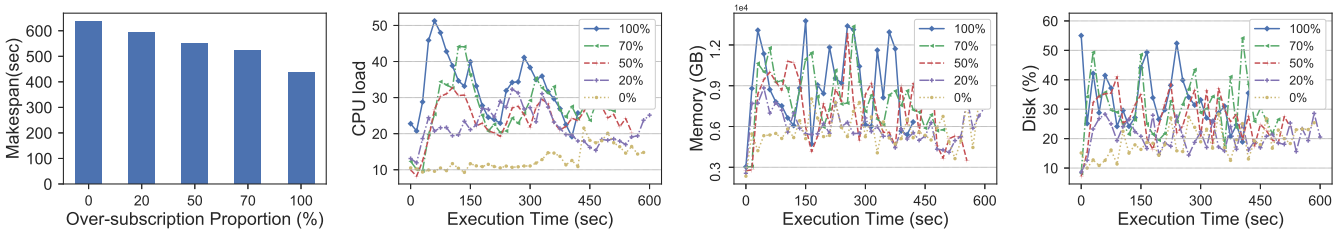


Fig. 7. The impact of varying over-subscription parameter

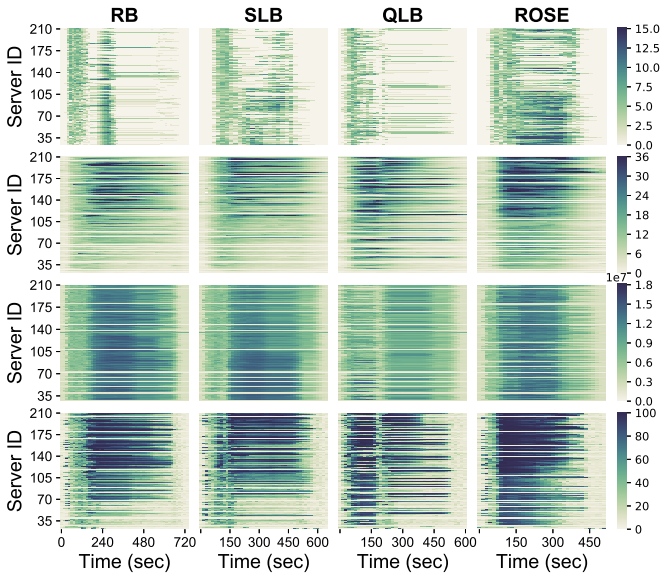


Fig. 8. Cluster resources utilization using over-subscription strategies. (Row#1: *Launched speculative task number;* Row#2: *CPU average load in one minute;* Row#3: *Memory usage;* Row#4: *Disk utilization;*)

TABLE IV
USAGE STATISTICS

Dimensions	RB	SLB	QLB	ROSE
Avg Task Num/node	1.637	1.868	1.308	3.176
Avg 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

Workloads. Experiments were conducted using the mixture of jobs including *WordCount*, *TeraSort*[32] on 10GB data and a Bayesian classification algorithm in the Mahout library[33] on 1.6GB set of Wikipedia pages jobs. These are well established to benchmark cluster scheduling performance[11][29][34]. While we conduct experiments with different workload in MapReduce, the concept of speculative tasks through over-subscribing 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.

B. ROSE vs. Centralized Over-subscription in Yarn

We evaluated the effectiveness of applying ROSE in Yarn(Yarn-r) by submitting 40 Mahout ML jobs into a 32-machine Yarn cluster. As shown in Table III, the average cluster CPU utilization 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 ROSE, increasing to 49.3% compared to 18.5% and 27.1%. Moreover, the overall makespan of submitted jobs can be shortened using ROSE-based Yarn-r by 21.0% and 18.1% compared with Yarn-n and Yarn-o. ROSE can successfully launch 8.5× more speculative tasks than Yarn-o without any exclusion from the NM's local queue. By contrast, in the centralized over-subscription method Yarn-o, a large portion of the created speculative tasks fail to enqueue. This is because ROSE is capable of launching speculative tasks more efficiently and accurately into machines with higher suitability due to removing delayed message piggybacking and adopting multi-phase machine selection.

C. Benchmarking

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.

Over-subscription Parameter Impact. Herein, we investigate the impact of over-subscription parameters that may influence the system and job performance. In a ROSE job, we can customize the proportion of tasks that can speculatively requests resources that can be over-subscribed. This ratio is used to balance between resource utilization and cross-job fairness within a multi-tenant environment. We adjust the ratio within a job from 0 % to 100% (where 0% indicates that no tasks within the job are allowed to use over-subscribed resources). Fig.7 demonstrates the full over-subscription (with 100% task over-subscription 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 increase. Therefore, we preset the ratio as 100% in all experiments. In particular, when diverse workloads within different constraints are submitted by multiple tenants, the ratio configuration can provide sufficient flexibility to satisfy various service requirements.

Resource Utilization. Fig. 9 shows ROSE increases CPU utilization, achieving 65.10% on average versus 36.37% with the non-over-subscription method. Fig. 8 and Table IV depict a heatmap of per-node resource utilization and the number of speculative tasks launched using different over-subscription strategies. ROSE can achieve a higher CPU and disk utility across the entire cluster. For instance, the average load can be increased by 42.75% and the number of speculative task launched per node even doubles compared with other strategies. 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. 10,

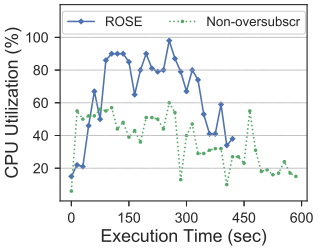


Fig. 9. CPU utilization

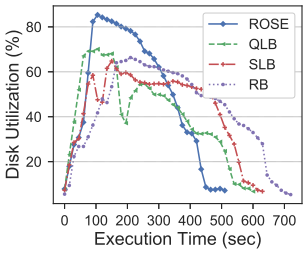
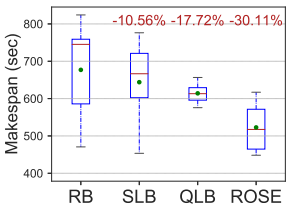
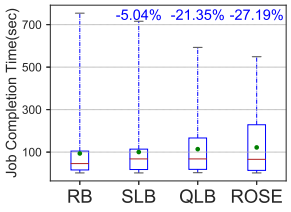


Fig. 10. Disk utilization



(a) Makespan comparison



(b) JCT comparison

Fig. 11. Workload execution time under different policies

ROSE can achieve an 18.23% disk utilization improvement. The memory usage does not exhibit an obvious grow as the CPU and disk thresholds are firstly reached which imposes a restriction on the memory over-subscription.

All these improvements are resultant of additional speculative tasks that utilize the over-subscribed 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, the 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 does not consider the runtime resource variation and thus lacks the 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 over-subscription. This is because the shorter task duration is, the less likelihood of task eviction occurrence during it's 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 the workloads for 20 rounds. Fig. 11(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.

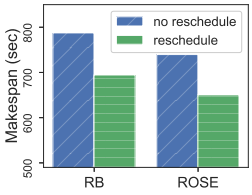


Fig. 12. Rescheduling approach

TABLE V
EVICTION NUMBER

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

Fig. 11(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.12 shows that RB and ROSE are capable of reducing the median makespan by approximately 12%. We also evaluate the task eviction occurrence during over-subscription. Table V demonstrates that ROSE achieves a substantially increases speculative task number by 37.78% and 43.54% compared to 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 in ROSE.

D. ROSE at Scale

We implemented ROSE into the Fuxi scheduler used within a large-scale test cluster at Alibaba to demonstrate it's effectiveness at scale. This system is formed by 4,300 servers, 100k jobs and over 200k tasks, containing total 42 millions parallel instances. Figure 13 shows cluster resource utilization when ROSE over-subscription is activated for 3 hours. It is observable that ROSE was able to consistently achieve a doubled CPU increment, with the average cluster CPU utilization increasing from 31.7% to 60.2% compared with no over-subscription activated. This improvement results in a 23.3% reduction of the overall job makespan. The evaluation

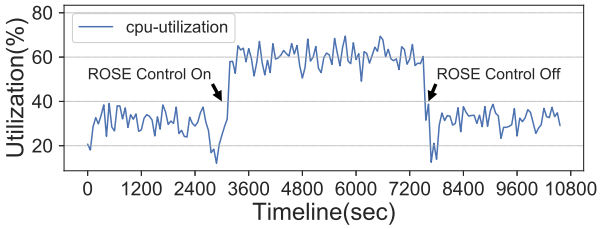


Fig. 13. ROSE in large scale testbed (4,300 servers)

demonstrates that ROSE can significantly accelerate the job execution and increase the system throughput.

VII. Related work

Cluster scheduling. Resource management systems in shared clusters are proposed[9][10][11][12] to underpin diverse workload through resource negotiations through a central resource manager. DRF[27], capacity scheduling[2] or fairness scheduling[3][35] 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. Over-subscription slightly violates the fairness criterion when best-effort tasks are launched.

Over-subscription in virtualized environments. Over-committing memory or CPU is a long-standing concept in the Linux kernel [8]. 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 over-subscription is a widely-used technique [16] [17] [36] 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 over-subscription level. However, this over-subscription mechanism cannot be directly applied into the computation-intensive scenario as batch processing jobs due to resource mismatch described within Section II, and unsuited to handle millions of running tasks in virtualized or latency-sensitive application platforms [29]. ROSE is to compact the resource utilization in a more black-boxing manner for those latency-agnostic tasks.

Over-subscription in big data environments. Centralized over-subscription in YARN[4] and Mesos[5] re-uses the resource allocation strategy used in the centralized scheduling pipeline. All over-subscription 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 interval, 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 over-subscription, and uses the threshold control to flexibly create and control speculative tasks at any possible moment. Within decentralized schedulers, Apollo[18] 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[19] adopts a hybrid scheduling to enhance cluster throughput and reduce feedback delays. However, both these approaches rely on the precise queue delay estimation which limit its applicability when considering volatile workloads behavior. Sparrow[31] and Hawk[37] 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.

VIII. Conclusion and Future Work

We have proposed in this paper a cluster scheduling system, called ROSE, that uses a multi-layered scheduling architecture to manage and oversubscribe idle resources in a speculative fashion. The ROSE system requests intelligently to launch speculative tasks within machines according to their effectiveness to oversubscribe resources. We have demonstrated that ROSE has several desirable advantages, including increased resource utilization and reduced workload makespan. Judiciously oversubscribed resources for speculative task execution often increase substantially the gain of resource efficiency. It is however important to notice that trade-off should be made elaborately between the job execution with fairness guaranteed and the effectiveness of resource scheduling to applications. Quota-based admission control may be enforced for multiple applications so that those oversubscribed resources can be fairly shared. Additionally, the distributed and loosely-coordinated scheduling mode within each AM to reuse idle resources is becoming a necessity when handling heterogeneous workloads with diverse characteristics. We plan to enhance further the effectiveness of ROSE in high resource contention environments, and at the same time reduce the preemption overhead.

Acknowledgment

This work is supported by the National Key Research and Development Program of China (2016YFB1000503), NSFC (61421003), and the EPSRC (EP/P031617/1).

REFERENCES

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

[2] Capacity scheduler. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html

[3] Fair scheduler. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html

[4] Yarn-oversubscription. [Online]. Available: https://issues.apache.org/jira/browse/YARN-1011

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

[6] Linux control groups. [Online]. Available: http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

[7] Linux containers. [Online]. Available: http://lxc.sourceforge.net

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "G: Packing and dependency-aware scheduling for data-parallel clusters," in *USENIX OSDI*, 2016.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

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

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

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

[26] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM*, 2015.

[27] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *USENIX NSDI*, 2011.

[28] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, 2014.

[29] 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.

[30] R. Yang, Y. Zhang, P. Garraghan, Y. Feng, J. Ouyang, J. Xu, Z. Zhang, and C. Li, "Reliable computing service in massive-scale systems through rapid low-cost failover," *IEEE Trans. on Services Computing*, 2017.

[31] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *ACM SOSP*, 2013.

[32] O. OMalley, "Terabyte sort on apache hadoop," *Yahoo, available online at: http://sortbenchmark. org/Yahoo-Hadoop. pdf,(May)*, pp. 1–3, 2008.

[33] Apache mahout. [Online]. Available: http://mahout.apache.org/

[34] 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.

[35] 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.

[36] 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.

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