2018

New YARN Non-Exclusive Resource Management Scheme through Opportunistic Idle Resource Assignment

Zhengyu Yang, *Northeastern University*
Yi Yao, *Northeastern University*
Han Gao, *Northeastern University*
Jiayin Wang, *Montclair State University*
Ningfang Mi, *Northeastern University*, et al.

New YARN Non-Exclusive Resource Management Scheme through Opportunistic Idle Resource Assignment

Zhengyu Yang*, Yi Yao*, Han Gao*, Jiayin Wang‡, Ningfang Mi*, and Bo Sheng†
* Dept. of Electrical & Computer Engineering, Northeastern University, 360 Huntington Ave., Boston, MA 02115
† Dept. of Computer Science, University of Massachusetts Boston, 100 Morrissey Boulevard, Boston, MA 02125
‡ Computer Science Dept., Montclair State University, 1 Normal Ave, Montclair, NJ 07043

***Abstract*—Efficiently managing resources and improving throughput in a large-scale cluster has become a crucial problem with the explosion of data processing applications in recent years. Hadoop YARN and Mesos, as two universal resource management platforms, have been widely adopted in the commodity cluster for co-deploying multiple data processing frameworks, such as Hadoop MapReduce and Apache Spark. However, in the existing resource management, a certain amount of resources are *exclusively* allocated to a running task and can only be re-assigned after that task is completed. This exclusive mode unfortunately leads to a potential problem that may under-utilize the cluster resources and degrade system performance. To address this issue, we propose a novel opportunistic and efficient resource allocation scheme, named OPERA, which breaks the barriers among the encapsulated resource containers by leveraging the knowledge of actual runtime resource utilizations to re-assign opportunistic available resources to the pending tasks. OPERA avoids incurring severe performance interference to active tasks by further using two approaches to efficiently balances the starvations of reserved tasks and normal queued tasks. We implement and evaluate OPERA in Hadoop YARN v2.5. Our experimental results show that OPERA significantly reduces the average job execution time and increases the resource (CPU and memory) utilizations.**

Keywords: Resource Allocation, MapReduce Scheduling, Hadoop YARN, Spark, Opportunistic, Starvation, Reservation.

I. INTRODUCTION

With the rise of big data analytics and cloud computing, cluster-based large-scale data processing becomes a common paradigm in many applications and services. Many cluster computing frameworks have been developed to simplify distributed data processing on clusters of commodity servers in the past decades. For example, Hadoop MapReduce [1], [2], as one of the prominent frameworks, has been widely adopted in both academia and industry for un-structured data processing [3]. As data sources become more diverse and parallel data processing algorithms become more complex, it is not possible for a single framework to be optimal for all applications. New frameworks are emerging in recent years and thriving to address different large-scale data processing problems. For example, Apache Spark [4], [5] was introduced to optimize iterative data processing and Apache Storm [6] was proposed to deal with streaming data.

A fundamental research issue in the field is that given the available computing resources in a cluster, how to efficiently manage the executions of a large volume of jobs. The emerging data processing systems as well as their resource management frameworks possess unique features compared to the traditional cluster computing systems. To better accommodate diverse data processing requirements, the common practice is to co-deploy multiple frameworks in the same cluster and choose the most suitable ones for different applications. A centralized resource management service is deployed to allocate a certain amount of resources to form a *resource container* at one of the servers to execute a task.Two popular and representative resource management platforms are Hadoop YARN [7] and Apache Mesos [8], which share the similar designs with centralized resource allocation and fine-grained resource representation. When the cluster is initially launched, each node declares its resource capacities, e.g., the number of CPU cores and the memory size. Meanwhile, applications from different frameworks send resource requests for their tasks to the centralized resource manager. The resource management tracks the available resources when allocating the containers, and guarantees that the resources occupied by all the containers on a host do not exceed its capacities.

While providing easy management and performance isolation, the existing *exclusive* mode of resource container leads to a potential problem that may underutilize the cluster resources and degrade system performance significantly. For example, a production cluster at Twitter managed by Mesos has reported to have its aggregated CPU utilization lower than 20% [9] when reservations reach up to 80%. Similarly, Google's Borg system has reported an aggregated CPU utilization of 25-35% while reserved CPU resources exceed 70% [10]. The major reason of such a low utilization is that the resources (e.g., CPU cores and memory) occupied by a task will not be released until that task is finished. However, tasks from many data processing applications often exhibit fluctuating resource usage patterns. These tasks may not fully use all the resources throughout their executions. As an example, a reduce task in MapReduce usually has low CPU utilization during its shuffle stage but demands more CPU resources once all intermediate data are received. Another example is an interactive Spark job. The resource usage of its tasks can be extremely low during a user's thinking time but significantly increases upon the arrival of a user request.

To solve this problem, we present a new opportunistic and efficient resource allocation scheme, named OPERA, which aims to break the barriers among the encapsulated resource containers by sharing their occupied resources. Rather than developing one-off solutions for a specific system like Hadoop, our goal is to develop general techniques that can be integrated into a unified resource management framework such that the

cluster resources can be shared by multiple data processing paradigms to increase resource utilization and cost efficiency. The main idea of our approach is to leverage the knowledge of actual runtime resource utilizations as well as future resource availability for task assignments. When a task becomes idle or is not fully utilizing its reserved resources, OPERA re-assigns the idle resources to other pending tasks for execution.

In particular, when a running task is observed to be idle or not fully utilize its assigned resources, OPERA aggressively assigns the spare resources to waiting tasks. There are two key problems that we need to consider in the design of this new approach. First, resource usage can dynamically change across time. Second, a server node can be overloaded due to resource over-provisioning, which may incur performance interference and degrade the performance of all active tasks. Therefore, in this paper, our solution presents the following contributions to solve these problems:

- dynamically monitors the runtime resource utilization;
- classifies the pending tasks to determine if each task is eligible for the new opportunistic resource allocation;
- efficiently assigns the idle resources occupied by the running tasks to other eligible pending tasks, and integrates the new approach with the existing resource allocation;
- mitigates severe resource contentions caused by opportunistic resource allocation;
- effectively avoids the starvation of both reserved tasks and normal queued tasks.

We implement OPERA in Hadoop YARN and evaluate its performance with a set of representative MapReduce and Spark applications. Our experimental results show that our OPERA with three resource releasing schemes can significantly reduce the average job execution time and increase the resource (memory and CPU) utilizations. We also show that OPERA is able to make better use of the opportunistic resources when Spark or interactive jobs are waiting for intermediate data or user input and further improve the overall performance. Three schemes for releasing resources when severe resource contentions happen have been investigated in our evaluation as well. We find that although the conservative scheme does not aggressively utilize idle resources, this scheme reduces both the number of killed tasks and the amount of wasted work and thus mitigates the performance degradation incurred by resource contention.

The organization of this paper is as follows. We present our understandings of task resource usage patterns and the intuition of opportunistic resource allocation in Sec. II. Sec. III describes details of our new opportunistic resource allocation scheme. The performance of this new scheme is evaluated under the workloads mixed with MapReduce and Spark jobs in Sec. IV. The related work is presented in Sec. V. We finally give our conclusion in Sec. VI.

II. MOTIVATION

A. Understanding Task Resource Usage Patterns

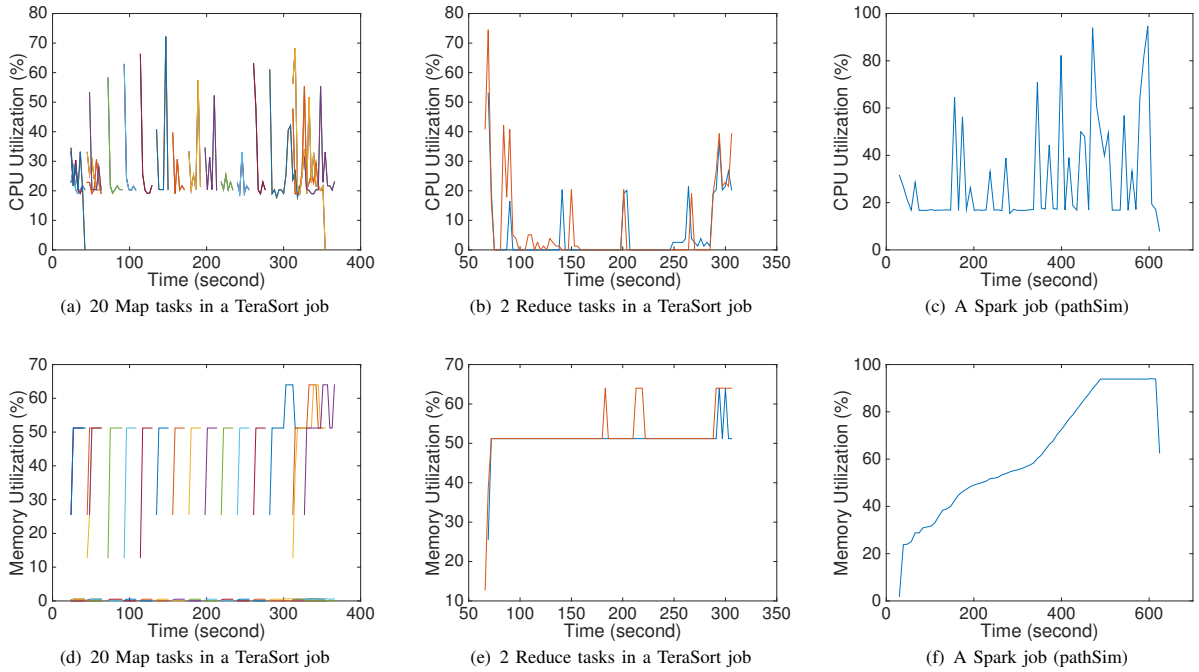
In the current resource management, a certain amount of resources are *exclusively* allocated to each running task, and will be recycled (i.e., re-assigned) only after the task is completed. This mechanism works well with short-lived, fine-grained tasks that usually process a consistent workload throughout their executions. When the cluster resources are repeatedly assigned to serve this type of tasks, the entire system can consistently keep a high resource utilization. However, if data processing jobs include tasks with long life cycles, the current resource management may not work efficiently. Long tasks usually consist of multiple internal stages and may have different resource usage patterns in each stage. For example, reduce tasks include two stages: data transfer/shuffling and reduce. Network bandwidth is the main resource consumed in the former stage and CPU resources are mainly used in the latter stage. The current framework allocates a fixed amount of resources throughout the life time of the task often leading to low resource utilization. In the above example of reduce tasks, the CPU utilization is low in the first stage of shuffling.

To explore this issue, we conduct experiments in a YARN cluster of 20 slave nodes (8 CPU cores and 16GB memory per node) to better understand the task resource usage patterns. In these experiments, we launch a *TeraSort* job (sorting a randomly generated 50GB input data) on a MapReduce Hadoop platform and a *pathSim* job (a data mining algorithm to analyze the similarity between authors using the academic paper submission records [11]) on a Spark platform. Fig. 1 shows the measured CPU and memory utilizations during the execution of different tasks. We observe that resource usages of all tasks, especially CPU utilizations, are fluctuating over the time. For example, the CPU utilizations of reduce tasks (see Fig. 1(b)) are extremely low for a long period (e.g., 100s $\sim$ 140s) because reduce tasks are waiting for the output of map tasks. Similarly, the resource usages of a Spark job change across time, see Fig. 1(c) and (f). The CPU utilizations of that Spark job range from 20% to 96%. Meanwhile, as Spark is a memory processing framework, we request 9GB memory for each task of that Spark job. However, we find that the assigned memory (i.e., 9 GB) is not always fully utilized by each task, especially at the beginning of the processing, as shown in Fig. 1(f).

B. Impact of Opportunistic Resource Allocation

One possible solution to avoid low resource utilizations is to use time series data (such as actual resource demands) for resource allocation. However, this solution is barely practical. It is difficult to accurately predict actual resource usages of tasks running in many frameworks. It can also dramatically increase the complexity of resource allocation even if a precise prior knowledge is available. Therefore, we consider an alternative solution that attempts to improve resource utilizations by opportunistically allocating resources for tasks, i.e., reassigning the occupied but idle resources to other pending tasks.

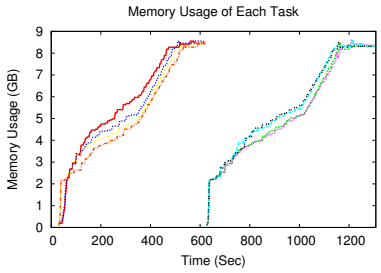
We here investigate the impact of such an opportunistic scheduling on data processing performance in order to understand how to best utilize system resources without causing severe performance interference. We execute Spark jobs in a YARN cluster under both the existing resource allocation scheme (i.e., the current scheme which reserves resources



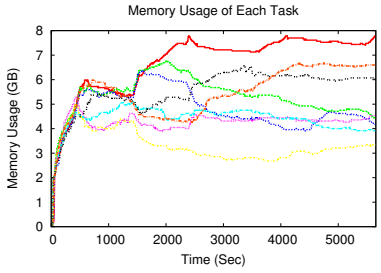
(a) 20 Map tasks in a TeraSort job (b) 2 Reduce tasks in a TeraSort job (c) A Spark job (pathSim)

(d) 20 Map tasks in a TeraSort job (e) 2 Reduce tasks in a TeraSort job (f) A Spark job (pathSim)

Fig. 1: The CPU and memory utilizations of the tasks from a MapReduce *TeraSort* job and a Spark *pathSim* job.



(a) Existing resource allocation



(b) Opportunistic resource allocation

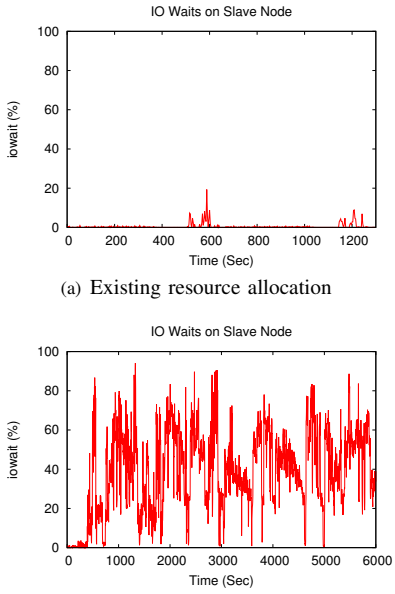
Fig. 2: Runtime memory usages on a node. Plot (a) shows the results under the existing resource allocation and Plot (b) shows the results under a simple opportunistic resource allocation scheme.

during the lifetime of a task) and a simple opportunistic resource allocation scheme. In these experiments, we set two slave nodes with the capacity of 12 CPU cores and 44GB memory each and submit 2 Spark jobs, each of which has 8 executor tasks.

Fig. 2 depicts the runtime memory usages of each executor task running on a slave node, and Fig. 3 shows the runtime IOwait ratios of that node. We observe that when we request 9GB memory for each task, only 4 tasks (or executors) can be launched on each node under the existing scheme, see Fig. 2(a). On each node, 36GB memory in total has been reserved for the first job while the remaining 8GB memory is not enough to run the tasks from the second job. In such a case, the second job has to wait till the first one completes and releases its resources, i.e., around time 600 seconds in Fig. 2(a). Obviously, memory resources are not always fully utilized across time under the existing scheme. On the other hand, low IOwait ratios are observed in Fig. 3(a), which indicates that no resource contention happens since each task is guaranteed to maintain its requested 9 GB memory during the entire period of its execution.

To make better use of the resources, we consider a simple opportunistic resource allocation solution that simply enforces both jobs to be executed concurrently by ignoring the resource capacity limit. The memory utilization is significantly improved as shown in Fig. 2(b). However, we find that the total completion length (i.e., makespan) of two jobs is increased. This is due to the fact that the total effective demand of memory eventually exceeds the capacity, which causes a severe memory contention when the running tasks from both jobs start to request more memory resources. Such memory contention incurs more IOs for swapping (e.g., the high IOwait ratios in Fig. 3(d)), which further degrades overall performance.

The above results give us some implications that opportunistic resource allocation does have potentials to improve the resource utilization. However, it comes with the risk of severe resource contention. Simply reassigning idle resources to a random pending task may not work well in practice. Motivated by theses findings, we develop a new approach, OPERA, that can identify an appropriate set of idle resources and assign them to suitable pending tasks.



(a) Existing resource allocation

(b) Opportunistic resource allocation

Fig. 3: Runtime IOwait ratios on a node. Plot (a) shows the results under the existing resource allocation and Plot (b) shows the results under a simple opportunistic resource allocation scheme.

III. The Design of OpERA

In this section, we present a new opportunistic resource allocation approach, named OpERA, which leverages the knowledge of the actual runtime resource utilizations to determine the availability of system resources and dynamically re-assigns idle resources to the waiting tasks. The primary goal of this design is to break the barriers among the encapsulated resource containers and share the reserved resources among different tasks (or containers) such that the overall resource utilization and system throughput can be improved.

A. Sketch of OpERA

As a resource management scheme, OpERA's goal is to assign the available resources in the system to the pending tasks. However, the definition of "available resources" in OpERA is different from that in the traditional systems. They include not only the resources that have not yet been assigned to any tasks, but also the occupied resources that are idle at the runtime. Therefore, OpERA includes two types of resource allocations, *normal resource allocation* and *opportunistic resource allocation*, referring to assigning the former and the latter types of available resources, respectively. Then, the basic design of OpERA boils down to two tasks, identifying the available resources under the new definition, and selecting a candidate pending task for execution.

We first define some terms that will be used in our design:

- **Opportunistic/guaranteed available resources**: When assigning *guaranteed available resources* to a task, our OpERA system always guarantees that those resources are available throughout that task's lifetime. On the other hand, if a task is assigned with *opportunistic available resources*, it might lose these resources and get terminated during its execution.
- **Opportunistic/normal tasks**: The tasks that are served with/without opportunistic available resources.

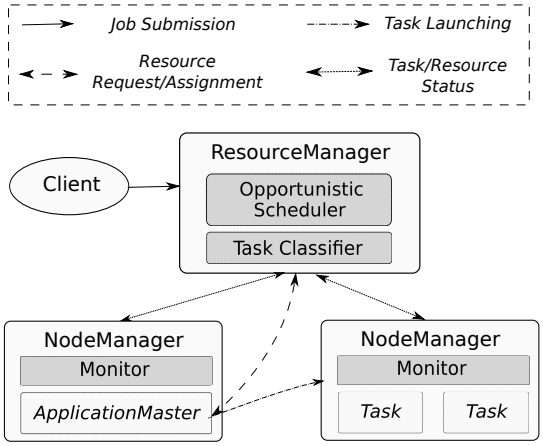


Fig. 4: The architecture of the OpERA-based YARN framework. The modified and newly designed components are marked in grey.

In particular, we develop OpERA on the top of the existing Hadoop YARN framework. The architecture is illustrated in Fig. 4. We develop the following three major components (the grey parts in Fig. 4).

Task Classifier: The goal of this component is to identify the eligible pending tasks for opportunistic resource allocation. As we discussed in Sec. II, opportunistic resource allocation is not suitable for all tasks. It is based on resource over-provisioning, and could cause severe resource contentions. In our design, only *short* tasks are eligible for opportunistic resource allocation because longer tasks are more likely to cause resource contentions. However, estimating the execution time of task is challenging in practice. This component is developed to dynamically and accurately classify all the pending tasks into two categories, i.e., short and long.

NodeManager Monitor: This component runs on each cluster node and mainly provides two functions. First, it monitors the dynamic usage of the occupied resources, i.e., the CPU and memory serving the active tasks on the node, and periodically reports the status to the ResourceManager. This function helps the ResourceManager estimate the available resources on each node for opportunistic resource allocation. Accurate reports in a timely fashion are crucial to the system performance. The second function of this component is to mitigate severe resource contentions when the total effective resource utilization is close to or over 100%. Multiple strategies are adopted in this component.

Opportunistic Scheduler: The last component is the core in our system. This component manages both normal and opportunistic resource allocations. When applying opportunistic resource allocation, this component identifies the available resources in the system based on the actual resource usage collected by *NodeManager Monitor*, and allocates them to the eligible pending tasks determined by *Task Classifier*.

The details of the above components are discussed in the following subsections. Table I lists a summary of notations used in this section.

TABLE I: Notations in this section

Notation	Description
n_i / t	node i / task t
r	a general resource $r \in$ {CPU, memory}
$C_i(r)$	resource capacity (r) of n_i
NT_i / OT_i	normal / opportunistic task set on n_i
RT_i	set of all running tasks on n_i, $RT_i = NT_i \cup OT_i$
$D_t(r)$	resource demand (r) of task t
$NU_i(r)$	total resource usage (r) on n_i
$TU_t(r)$	resource usage (r) of task t, $NU_i(r) = \sum_{t \in RT_i} TU_t$
$GA_i(r)$	guaranteed available resource (r) on n_i
$OA_i(r)$	opportunistic available resource (r) on n_i
tr_i	reserved task on node i

B. Task Classifier

Task Classifier is expected to classify all the pending tasks into two categories (i.e., short and long), indicating whether they are eligible for the opportunistic resource allocation. Our design basically includes two steps: (1) estimate the execution time of a task; (2) compare it to a threshold to determine the category it belongs to. The classification accuracy is the major concern in our design.

The execution time of a task is hard to be profiled before the execution because it depends on some runtime parameters such as the hardware of the processing cluster. Prior work [12], [13] attempted to use historic statistics of the same type of tasks in the same job to estimate the execution time. We adopt the same approach and extend it to consider the historic information from other applications and even other frameworks. The intuition is that the tasks from the same processing stage of the same type of applications usually have an identical function, process the input data with similar size, and thus have similar execution time. For example, in a MapReduce system, the input data size of each map task is configured by a system parameter. Consider a cluster processing two *wordcount* applications such that one with 10GB input data and the other with 100GB input data. Different input data sizes only yield different numbers of map tasks. All map tasks in these two applications will process similar size of input files and have similar execution time.

Obviously, the historic task execution information is helpful. However, the challenge here is to estimate the execution time when there is no exactly matching historic information. For example, the first batch of map tasks of a new type of MapReduce application and reduce tasks in general MapReduce jobs (usually there are only a few reduce tasks and they may not become effective references to each other). Our solution aims to derive an estimation based on the information from different processing stages, applications, and frameworks.

In OPERA, we adopt the naive Bayes classifier [14] to identify tasks as *short* or *long*. It has been widely used in text classification and spam filtering due to its high accuracy and low overhead on both storage and computation. And we address the challenges of task classification by presenting a new hierarchy approach that considers the following five properties of each task t.

- F_t: the framework of task t, e.g., MapReduce, Spark, and Storm;
- A_t: the application name of task t, e.g., wordcount and sort;
- S_t: the processing stage that task t belongs to;
- P_t: the progress of the application that task t belongs to;
- D_t: the resource demands of task t.

The first three properties ($\{F_t, A_t, S_t\}$) identify the computation process of each task t, e.g., { MapReduce, WordCount, Map }, and { Spark, PageRank, Stage 3 }. Their values represent the computation details with different granularities. The last two properties are runtime parameters configured by the users. Apparently, with different resource demands (D_t), the execution time of the same task could vary. The progress of the application (P_t) is another implicit factor that may affect the execution time. For example, the user can configure a MapReduce application such that its reduce tasks will start after all its map tasks are finished or the reduce tasks can start when half of the map tasks are completed. In either of these configurations, given different progress values (e.g., all or half of map tasks finish) of the job, the execution time of a reduce task will be different. We then define the features of each task t as a tuple using Eq. 1.

$$\begin{aligned} \mathcal{F}_t \;=\; & \{\ \{F_t\},\ \{F_t, A_t\},\ \{F_t, A_t, S_t\}, \\ & \{F_t, A_t, S_t, P_t\},\ \{F_t, A_t, S_t, P_t, D_t\}\ \} \end{aligned} \quad (1)$$

We find that combining these task properties together to form such hierarchy features provides more meaningful hints for a better prediction accuracy. In fact, considering each of these properties individually does not provide useful information for classification. For example, the map tasks (same S_t) in different MapReduce applications may yield different execution times; the tasks from the same application "sort" (same A_t) in MapReduce and Spark frameworks may not have the same execution time. However, on the other extreme side, if we classify tasks only based on the historic information from the tasks with the same values of all properties, there will be a lack of information for many tasks and we will miss the correlation between the tasks that share a subset of the properties. Therefore, we decide to combine different task properties in a hierarchical structure in order to explore the correlation between "similar" tasks and confine our estimation in a reasonable scope to accurately classify the tasks.

Once a task t's features ($\mathcal{F}_t$) are determined, we calculate the posterior probability ($P(C|\mathcal{F}_t)$) of its category (C_t) using Eq. 2–3 as follows.

$$P(C_t|\mathcal{F}_t) \;\propto\; P(C_t) \cdot P(\mathcal{F}_t|C_t), \quad (2)$$

$$P(\mathcal{F}_t|C_t) \;=\; \prod_i P(\mathcal{F}_t^i|C_t), \quad (3)$$

where $C_t \in \{short, long\}$ and $\mathcal{F}_t^i$ represents the ith element of the feature tuple $\mathcal{F}_t$. Task t is then classified to one of the two categories which yields a higher posterior probability. Probabilities, e.g., $P(C_t), P(\mathcal{F}_t^i|C_t)$ used in Eq. 2–3, are online learned and updated upon the completion of tasks. We determine the category (short or long) of the finished tasks

by checking if their execution times are less than a threshold (e.g., 1 minute) and update all the related probabilities with tasks' features and category information. There is a special case when an application with all new features is submitted, i.e., no historical information can be referred. In our solution, we opt to conservatively classify the task as a *long* one.

C. NodeManager Monitor

The key idea of our new opportunistic scheduling scheme is to assign idle resources to the pending tasks based on the actual runtime resource usages. Therefore, we develop a monitor module on NodeManagers to (1) keep tracking both CPU and memory usages of the running tasks and sending the collected usage information to ResourceManager through heartbeat messages; and (2) detect and solve performance interferences caused by resource contentions when the resources on a node have been over provisioned and the overall resources occupied by the running tasks exceed the node's capacity.

Algorithm 1: Node Monitoring

```
   Data: C_i(r), POLICY, BR_i, BT_i
1  Procedure Monitoring()
2      while TRUE do
3          NU_i(r) ← 0, op ← false, c ←"NONE";
4          foreach t in RT_i (the set of running tasks) do
5              NU_i(r) ← NU_i(r) + CurrentUsage(r);
6              if t is an opportunistic task then
7                  op ← true;
8          if op then
9              c = CRes(NU_i);
10         RelieveContention(c, POLICY);
11         SLEEP MonitorInterval;
12 Procedure CRes(NU_i)
13     if NU_i(mem) > ρ * C_i(mem) then
14         return "Memory";
15     if NU_i(CPU) > ρ * C_i(CPU) then
16         return "CPU";
17     return "NONE";
18 Procedure RelieveContention(c, PO)
19     if PO = AGGRESSIVE and c = Memory then
20         kill the most recently launched opportunistic task;
21     else if PO = NEUTRAL and c ≠ NONE then
22         kill the most recently launched opportunistic task;
23     else if PO = PRESERVE then
24         if c ≠ NONE then
25             kill the most recently launched opportunistic
                 task;
26             BR_i ← α · BR_i;
27             BT_i ← α · BT_i;
28             LastReliefTime = CurrentTime;
29         else
30             if CurrentTime − LastReliefTime > BT_i
                 then
31                 BR_i ← BR_i/α;
32                 BT_i ← BT_i/α;
33                 LastReliefTime = CurrentTime;
```

Alg. 1 shows the main process for monitoring resource utilization, detecting and mitigating resource contention on a working node which consists of three modules. In particular, the first module (lines 1–11) periodically collects the CPU and memory usages of the running tasks. $NU_i(CPU)$ and $NU_i(mem)$ represent the CPU and memory usage on node n_i, respectively. We use op to check if there exists any opportunistic task on the node (lines 6-7). If op is false, there will be no resource contention caused by the opportunistic resource allocation. Otherwise, resource contention is possible, and we call the function $CRes$ (line 9) to check the effective resource utilization and return the type of contented resource indicated by variable c. Eventually, the algorithm calls $RelieveContention$ function to mitigate the resource contention. The arguments passed to the function are the type of the resource that causes the contention ("CPU", "Memory", or "NONE"), and the user-specified policy for handling the contention.

The second module, $CRes$ (lines 12-17), simply compares the resource usage $NU_i(CPU)$ and $NU_i(mem)$ with a predefined threshold. If the threshold has been exceeded, the algorithm determines that there exist resource contentions, and reports the type of the contended resource to the main monitoring module. In the algorithm, we set the threshold as $\rho * C_i$, where $C_i(CPU)$ and $C_i(mem)$ are the CPU and memory capacity of node n_i. ρ is an adjusting parameter that can tune the performance of our scheme. By default, we set ρ to 0.95. Note that if both CPU and memory have contentions, this module returns "Memory" as we give memory contention a higher priority to be mitigated.

Finally, in the third module, $RelieveContention$ (lines 18–33), offers the following policies to solve the problem of performance interference caused by resource contentions.

- AGGRESSIVE: this policy kills the most recently launched opportunistic task only when the monitor detects contention on memory resources.
- NEUTRAL: this policy kills the most recently launched opportunistic task under either CPU or memory contention.
- PRESERVE: this policy applies the same behaviors as NEUTRAL. It further blocks some opportunistic available resources on the node for a period of time.

In all three policies, when resource contentions are detected, the NodeManager attempts to kill the most recently launched opportunistic task to reduce the resource consumption.

AGGRESSIVE policy (lines 19–20) only checks memory contentions. This policy ignores the CPU contention because it is usually less harmful and does not lead to task failures as memory contention does. As opportunistic tasks are relatively short and can release the occupied resources quickly, AGGRESSIVE policy tends to aggressively keep these opportunistic tasks running even under CPU contentions for achieving a better overall performance. On the other hand, the drawback of this policy is that the normally reserved resources cannot always be guaranteed especially during the periods of system overloading. In contrast, NEUTRAL is a conservative policy (lines 21–22) that kills opportunistic tasks under both CPU and memory resource contentions. Clearly, this policy can guarantee the reserved resources but might incur frequent task terminations, especially when resource utilizations of the running tasks are oscillating.

To guarantee the reserved resources without killing too many tasks, we further present a PRESERVE policy for

contention mitigation, by introducing the concepts of blocked resource (BR_i) and block time (BT_i), see lines 23-33 of Alg. 1. Besides killing opportunistic tasks, this policy further blocks a certain amount (BR_i) of opportunistic available resources of node n_i for a time window (BT_i). Under the PRESERVE policy, the opportunistic scheduler estimates opportunistic available resources (OA_i) by considering both the actual resource usage of running tasks (TU_t) and the amount of blocked resources (BR_i), as shown in Eq. 4.

$$OA_i(r) = C_i(r) - \sum_{t \in RT_i} TU_t(r) - BR_i(r). \quad (4)$$

The values of BR_i and BT_i are adjusted exponentially in our solution. As shown in lines 26 – 27, We double the values of BR_i (i.e., to reserve more blocked resources) and BT_i (i.e., to decrease the availability of opportunistic assignment) to be more conservative if a new resource contention is detected within the current block time window. Similarly, the values of BR_i and BT_i are decreased exponentially by a factor of α, if no resource contention has been detected in the past time window BT_i, see line 31 to 32.

D. Opportunistic Scheduler

Next, we present the last component in this subsection. As discussed in Sec. II, the ResourceManager under the current YARN framework considers each resource container exclusively allocated for a single task. When assigning resources to a pending task, the ResourceManager checks the *available resources* on each node as follows,

$$C_i(r) - \sum_{t \in RT_i} D_t(r). \quad (5)$$

However, in practice, the tasks do not always fully utilize their assigned resources during their executions. The traditional resource allocation often leads to a low resource utilization.

To address this issue, we develop the opportunistic scheduler, which considers both *guaranteed available resources* and *opportunistic available resources*. The key difference between these two types of resource allocations is the calculation of the available resources. The guaranteed available resources ($GA_i(r)$) are defined in Eq. 6, which equal to the differences between the resource capacities and the total resource demands of the normal tasks on node n_i. When calculating the opportunistic available resources ($OA_i(r)$), we consider the runtime resource usages of the running tasks rather than their resource demands, see Eq. 7.

$$GA_i(r) = C_i(r) - \sum_{t \in NT_i} D_t(r), \quad (6)$$

$$OA_i(r) = C_i(r) - \sum_{t \in RT_i} TU_t(r), \quad (7)$$

where $C_i(r)$ represents the capacity of resource r of node n_i and $D_t(r)$ and $TU_t(r)$ represent task t's resource demand and resource usage, respectively.

Alg. 2 presents the high level idea of our scheduling scheme. When allocating available resources, the opportunistic scheduler always first tries to assign guaranteed available resources,

Algorithm 2: Task Assignment

```
   Data: n_i, GA_i, OA_i, tr_i, D_t
 1 Procedure Assign(t, D_t)
 2     if NormalAssign(t, D_t) then
 3         return true;
 4     else if OpportunisticAssign(t, D_t) then
 5         return true;
 6     return false;

 7 Procedure NormalAssign(t, D_t)
 8     if D_t(r) < GA_i(r) then
 9         GA_i(r) ← GA_i(r) − D_t(r);
10         OA_i(r) ← OA_i(r) − D_t(r);
11         Assign task t to node n_i;
12         return true;
13     return false;

14 Procedure OpportunisticAssign(t, D_t)
15     if t is eligible for opportunistic resource allocation then
16         if D_t(r) < OA_i(r) then
17             OA_i(r) ← OA_i(r) − D_t(r);
18             Assign task t to node n_i;
19             return true;
20     return false;
```

i.e., normal resource assignment (lines 2–3). If the resource demand of the task cannot be fulfilled, the scheduler then attempts to allocate opportunistic available resources (lines 4–5).

E. Task Reservation

The last problem of task assignment is how to address the "insufficient resource" scenario for both normal and opportunistic assignment. That is, neither guaranteed resources nor opportunistic resources are sufficient for task and the $Assign$ function in Alg. 2 returns false. Here, we present two approaches to handle this insufficiency scenario.

1) Strict Reservation Approach: Our first approach is to *strictly* follow the order of tasks in the task queue (i.e., *strict* reservation). When an "insufficiency" event happens (i.e., the task is not eligible for opportunistic scheduling or still cannot fit into the opportunistic available resources on the node), the scheduler reserves this task on the node and stops the assignment process until receiving the next heartbeat message. Our scheduler treats the selected task as a reserved one which has a higher priority to receive the available resources in the future.

Alg. 3 shows the main procedure of this *strictly* approach for node update. As discussed in Sec. III-A, the NodeManager of each working node periodically sends heartbeat messages to the ResourceManager, which includes the node's health status and runtime resource utilizations of each running task. Once receiving a heartbeat message from one working node, our scheduler updates both guaranteed and opportunistic available resources of that node using Eq. 6 and Eq. 7 (line 2) , and then tries to fulfill the previous reserved task (tr_i) on the particular node if there exists one (line 5 to 9). Otherwise, the scheduler sorts the task queue based on the Fair queuing discipline (i.e., the deficit between their deserved fair share of cluster resources and actual occupied resources) to choose a

Algorithm 3: Node Update (Strict Approach)

```
   Data: n_i, GA_i, OA_i, tr_i, D_t
1  Procedure NodeUpdate(n_i)
2      Update GA_i, OA_i;
3      if GA_i < MinTaskDemand and
         OA_i < MinTaskDemand then
4          return;
5      tr_i ← GetReservedTask(n_i);
6      if tr_i ≠ NULL then
7          if Assign(tr_i, D_{tr_i}) then
8              tr_i = NULL;
9          return;
10     else
11         t ← GetNextTaskFromQueue();
12         if Assign(t, D_t)==false then
13             tr_i = t;
14         return;
```

Algorithm 4: Node Update (Multi-chance Approach)

```
   Data: n_i, GA_i, OA_i, tr_i, D_t
1  Procedure NodeUpdate(n_i)
2      Update GA_i, OA_i;
3      if GA_i < MinTaskDemand and
         OA_i < MinTaskDemand then
4          return;
5      tr⃗_i ← GetReservedTaskQueue(n_i);
6      if tr⃗_i ≠ ∅ then
7          for tr_ij ∈ tr⃗_i do
8              if Assign(tr_ij, D_{tr_ij}) then
9                  tr⃗_i −= tr_ij;
10             else
11                 skipCount[tr_ij] += 1;
12                 if skipCount[tr_ij] > ε_s then
13                     return;
14     while |tr⃗_i| ≤ ε_r and maxSkip(tr⃗_i) ≤ ε_s do
15         t ← GetNextTaskFromQueue();
16         if Assign(t, D_t) then
17             for each tr_ij ∈ tr⃗_i do
18                 skipCount[tr_ij] += 1;
19         else
20             tr⃗_i += t;
21             skipCount[t] = 1;
22     return;
```

task t for assigning the available resources to, see in lines 10 to 14. As discussed, this task will be reserved as tr_i line 13, if the scheduler cannot assign either guaranteed or opportunistic available resources for the task t. The scheduler will return and assign the available resource to this reserved task in the next round.

2) Multi-chance Reservation Approach: The *strict* reservation approach does not assign resources to other tasks if there is a reserved one, which is effective for preventing the starvation of the reserved task. However, this approach may cause the starvation on other waiting tasks, if the reserved task's resource demand is not easy to be satisfied. Therefore, we present the second approach called "multi-chance reservation", which aims to avoid the starvation of both reserved and queued tasks by considering more than one tasks to be reserved and allowing those reserved tasks to be "skippable" for other tasks that are eligible to be assigned during runtime.

As shown in Alg. 4, a reservation queue with a fixed size (i.e., vector $\vec{tr_i}$) is used to store those reserved tasks. To avoid starvations, we set up two thresholds – the maximum length of the reservation queue (i.e., ε_r), and the maximum time that each reserved task can be "skipped" by *any other* tasks (i.e., ε_s). In detail, the scheduler first attempts to assign resources to tasks in the reservation queue based on the FIFO policy (see line 6 to 13). If the $Assign$ function returns true, the reserved task tr_{ij} will be dequeued from the reservation queue $\vec{tr_i}$ and run with the assigned resources, seeline 8 and 9. If a reserved task tr_{ij} is not able to be assigned to the current node due to insufficient resource, and it is eligible to be "skipped" (line 12), then the scheduler increases tr_{ij}'s $skipCount$ by 1 (line 11), and then moves on to check the next reserved task in $\vec{tr_i}$. Otherwise, if that reserved task tr_{ij} has reached the preset skip counter threshold ε_s, the scheduler will return and wait for the next round to assign resources to task tr_{ij}.

Furthermore, if the current length of reservation queue is less than ε_r, and the maximum skip counter of the reservation queue is less than ε_s, then the scheduler can further select a normal task for assign resource by calling function $GetNextTaskFromQueue$ in line 15.

Once that normal task is assigned with available resources, the skip counters for all reserved tasks will be increased by 1 (line 18). Similarly, if that normal task cannot get enough resources, then that task will be queued in the reservation queue with its skip counter initialed as 1 (line 20–21).

IV. Evaluation

We implement the proposed opportunistic resource allocation scheme, OpERA, on Hadoop YARN `v2.5`. Specifically, we modify the scheduler component in the ResourceManager of YARN (on top of the Fair scheduler) to include a task classifier for separating the opportunistic task assignment from the normal task assignment. We note that OpERA can also be integrated with any other scheduling algorithms. In the NodeManager, we enable the mechanisms of runtime resource monitoring/reporting as well as contention detection and integrate these mechanisms in the ContainerMonitor component. The communication protocols and messages among the ResourceManager, NodeManagers, and ApplicationMasters are also modified to convey the actual resource usage and the assignment type (i.e., normal or opportunistic) information of tasks. We evaluate OpERA in a real YARN cluster with different data processing workloads which include mixed sets of representative MapReduce and Spark jobs.

A. Experiment Settings

We conduct our experiments in a YARN cluster which is deployed in a cloud environment provided by CloudLab [15]. This YARN cluster is configured with one master node and 20 working nodes, each of which has 8 physical cores. We configure 2 virtual cores for each physical core such that there are 16 vCores in total on each working node. Among those 16 vCores, we use one vCore for NodeManager and the HDFS usage, and the remaining 15 vCores for running cluster

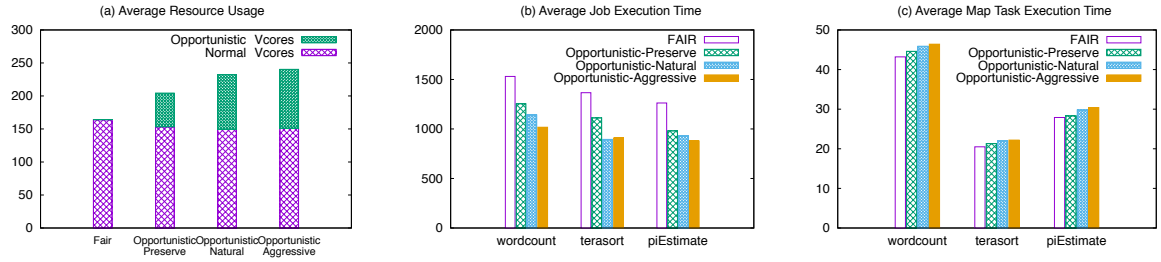


Fig. 5: The overall performance in the experiments with 6 MapReduce jobs

computing applications. Each node is configured with memory capacity of 12 GB. Thus, the total resource capacity of this cluster is $< 300vCores, 240GB >$ for CPU vCores and memory. The CloudLab provides 2x10Gbps network interfacesto every node via software-defined networking. Besides, we use multiple 500GB SATA drives (Intel DC S3500) in the cluster for HDFS.

The following four benchmarks are considered in our experiments:

- *pathSim*: a Spark application [11] that computes the meta path based on the similarity among academic paper authors. The input data contains 1.2 million paper submission records.
- *terasort*: a MapReduce application that sorts the input records. We use 50GB input data generated by *teragen*.
- *wordcount*: a MapReduce application that counts the occurrences of each word in the input files. Each input file with 50GB data is generated through *randomTextWriter*.
- *piEstimate*: a MapReduce application that estimates the value of π using the quasi-Monte Carlo method. Each task processes 300 million data points.

Table II shows the configurations of each application, including task numbers and task resource requests. By default, we configure each task's resource request according to their actual resource usage. The CPU demand of a task is equal to 75% of its peak CPU usage and the memory requirement is set to that task's maximum memory usage. These applications thus have various resource requirements. For example, the tasks from Spark applications are memory intensive while MapReduce tasks are mainly CPU intensive.

TABLE II: Task Configurations of Applications

Framework	Application	Task Num.	Task Resource Req.
Spark	*pathSim*	10 executors	$< 4vCores, 9GB >$
MapReduce	*terasort*	374 mappers	$< 4vCores, 1GB >$
		100 reducers	$< 2vCores, 1GB >$
	wordcount	414 mappers	$< 4vCores, 1GB >$
		50 reducers	$< 2vCores, 1GB >$
	piEstimate	500 mappers	$< 3vCores, 1GB >$
		1 reducers	$< 2vCores, 1GB >$

In our experiments, we set the results under the original YARN framework with the Fair scheduler as a baseline for comparison. The major performance metrics we consider for evaluating OPERA include resource utilizations and job execution times.

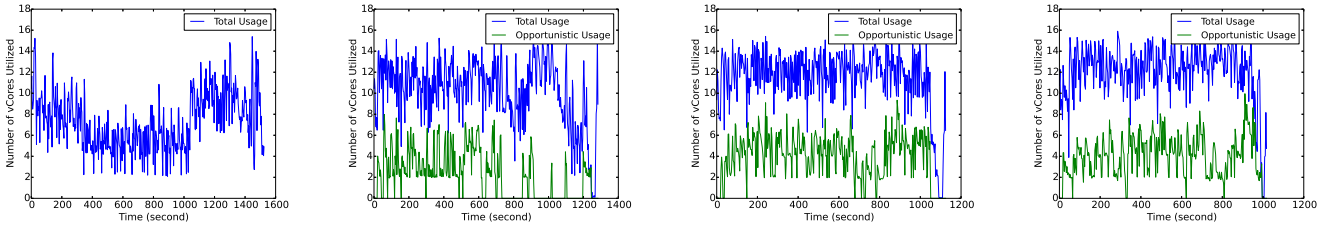
B. Workloads with MapReduce Jobs Only

In the first set of experiments, we generate a workload of 6 MapReduce jobs, where each MapReduce application listed in Table II launches 2 jobs. We execute this workload under both the traditional YARN resource allocation scheme with the Fair scheduler and with our OPERA scheme with three mitigation policies, i.e., AGGRESSIVE, NEUTRAL, and PRESERVE.

Fig. 5 presents the average cluster resource usages (i.e., the number of used vCores) and average job execution times (in seconds). We observe that our OPERA scheduler is able to more efficiently utilize cluster resources as shown in Fig. 5(a). As mentioned in Sec. IV-A, the total CPU capacity in our YARN cluster is $300\,vCores$, i.e., $15\,vCores \cdot 20\,nodes$. We can see that the original YARN system with Fair only uses about 50% (164) of vCores in average. OPERA increases the number of actually used vCores up to 240 (e.g., with the AGGRESSIVE scheme), which accelerates the overall processing by using more CPU resources. As a result, compared to Fair, our OPERA significantly improves the average job execution times for each MapReduce application, with relative improvements of 19.7%, 28.7%, and 32.4%, respectively, see Fig. 5(b).

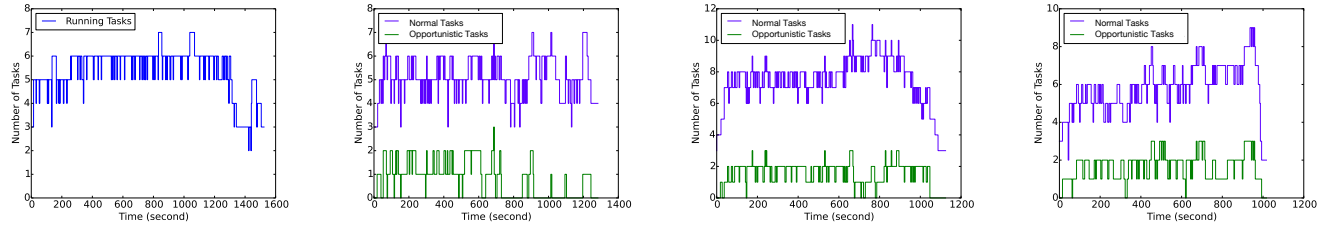
While improving resource utilization and job throughput, opportunistic scheduling may compromise average the task execution times. For example, Fig. 5(c) presents the average map task execution times of different applications. The reason we focus on investigating map task execution times is that the reduce task execution times highly depend on how long they overlap with map tasks and thus are more random. It is clear that task execution time degrades differently under different resource contention relief policies. If we assign opportunistic tasks more aggressively and do not kill tasks under CPU contention, i.e., AGGRESSIVE policy, then tasks experience longer execution time due to more frequent performance interference, but average job execution times are improved because of higher task concurrency in the cluster. On the other hand, the PRESERVE policy can provide better task performance isolation, but less improvement in system resource utilization and job throughputs. Compared to the Fair scheduler, average task execution time of our proposed three policies degraded only by 2.8%, 6.8%, and 8.2%, respectively. This is because those opportunistically assigned tasks are very short and do not introduce severe resource contentions.

To better understand how OPERA and Fair work, we also present the CPU vCore usages, the number of normal tasks, and the number of opportunistic tasks across time on a single cluster node in Fig. 6 and Fig. 7. Obviously, under the Fair



(a) Fair (b) OPERA-PRESERVE (c) OPERA-NEUTRAL (d) OPERA-AGGRESSIVE

Fig. 6: Runtime CPU usages on a single cluster node under the workload with 6 MapReduce jobs



(a) Fair (b) OPERA-PRESERVE (c) OPERA-NEUTRAL (d) OPERA-AGGRESSIVE

Fig. 7: The numbers of normal tasks and opportunistic tasks on a single cluster node under the workload with 6 MapReduce jobs

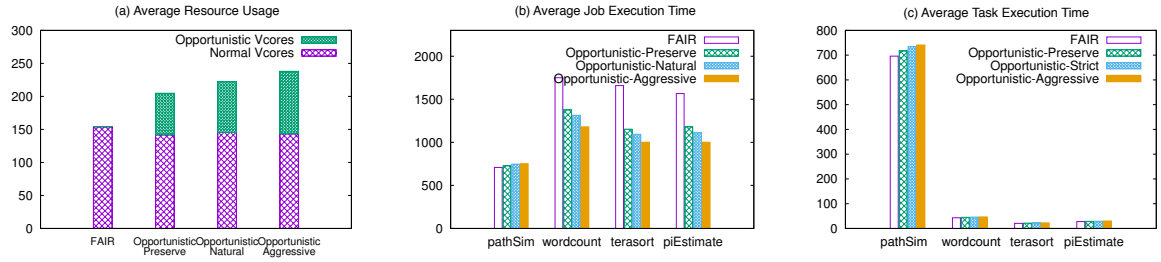


Fig. 8: The overall performance in the experiments with MapReduce and Spark jobs

policy, the number of vCores that are actually used is low and fluctuating across time, see Fig. 6(a). Moreover, the number of normal tasks does not change much under Fair (see Fig. 7(a)), which further indicates that these tasks running under the traditional resource allocation with the Fair scheduler yield varying CPU usage patterns. On the other hand, through the opportunistic resource allocation, the system resources (e.g., CPU vCores) are better utilized because more tasks are scheduled to run in a node when we detect underutilized resources on that node, as shown in Fig. 6(b)-(d). Particularly, OPERA with AGGRESSIVE or NEUTRAL always keeps CPU resources fully utilized (i.e., around 15 vCores in use per node), see plots (c) and (d) in Fig. 6.

Table III further shows the prediction accuracy of our task classifier which adopts the hierarchy feature-based naive Bayes classification algorithm as described in Sec. III-B. In this table, the "Fact" column shows the total number of actual short (resp. long) tasks, while the "Classification" column presents the predicted results, i.e., the numbers of tasks that are predicted to be short and long the predication accuracy.

TABLE III: Task classification.

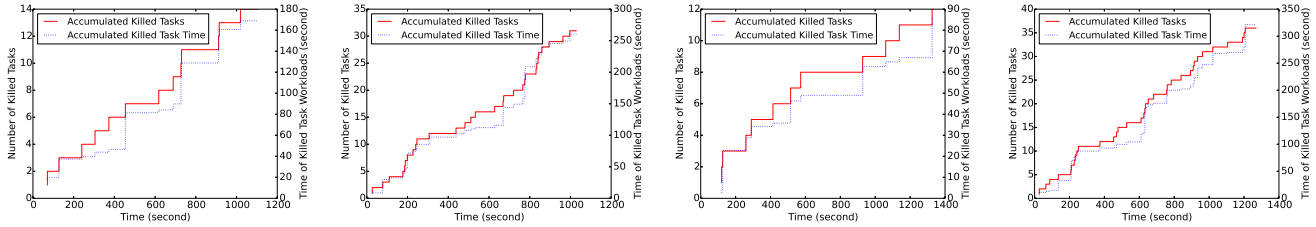
	Fact	Classification		
	Task Number	Short	Long	Pred. Accuracy
Short	2777	2417	360	87.0%
Long	107	2	105	98.1%

We observe that our task classifier is able to accurately categorize most of short and long tasks with high prediction accuracy ratios, i.e., 87% and 98%, respectively. More importantly, our classifier successfully avoids the false positives (i.e., predicting a long task as short) that can incur severe resource contention and other harmful consequences. As shown in Table III, only 1.9% (i.e., 2 out of 107) of the long tasks are classified as short ones. On the other hand, we notice that the false negative (i.e., predicting a short task as long) ratio is slightly high, i.e., 13%. However, it is still in a low range and only prevents us from assigning opportunistic available resources to those tasks, which in general does not degrade the overall performance.

C. Workloads with MapReduce and Spark Jobs

In the second set of experiments, we launch two Spark jobs, i.e., *pathSim* [11], together with 6 MapReduce jobs that are the same as we have in the first set of experiments. Here, each Spark executor occupies 10GB (i.e., 9GB executor memory request and 1GB overhead) memory resource on a cluster node.

Fig. 8 shows the experimental results, including the average job and task execution times and the average vCore usage under different scheduling policies. Obviously, all MapReduce jobs receive a significant performance improvement under OPERA. As shown in Fig. 8(b) and (c), the average job



(a) PRESERVE (b) NEUTRAL (c) PRESERVE (d) NEUTRAL

Fig. 9: Accumulated numbers of killed tasks and accumulated amounts of wasted execution (in seconds) under the workloads (a)(b) with MapReduce jobs, and (c)(d) with MapReduce and Spark jobs.

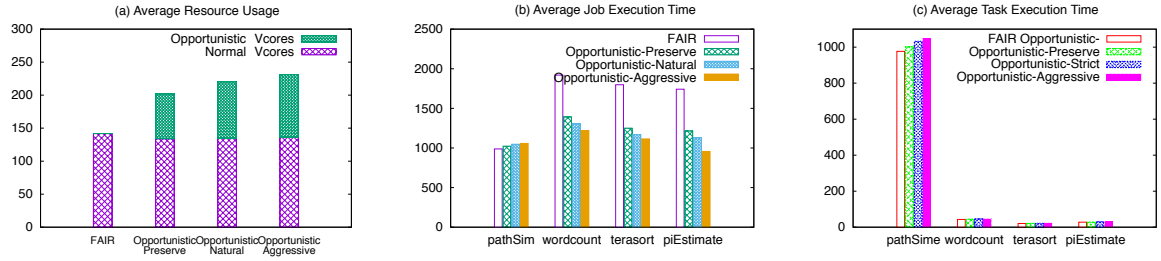


Fig. 10: The overall performance in the experiments with MapReduce and Spark jobs that have 5-minute interactive periods.

execution time of all MapReduce jobs is reduced by 25.5%, 29.3%, and 36.1% under PRESERVE, NEUTRAL, and AGGRESSIVE, respectively. On the other hand, the two Spark jobs (i.e., *pathSim*) do not benefit from our opportunistic resource allocation. The reason is that all tasks in *pathSim* are launched together in a single wave. The parallelism of these Spark jobs thus cannot be further improved through opportunistic resource allocation. Moreover, the performance of two Spark jobs becomes slightly worse under our opportunistic resource allocation due to the resource contention caused by other opportunistically scheduled MapReduce tasks.

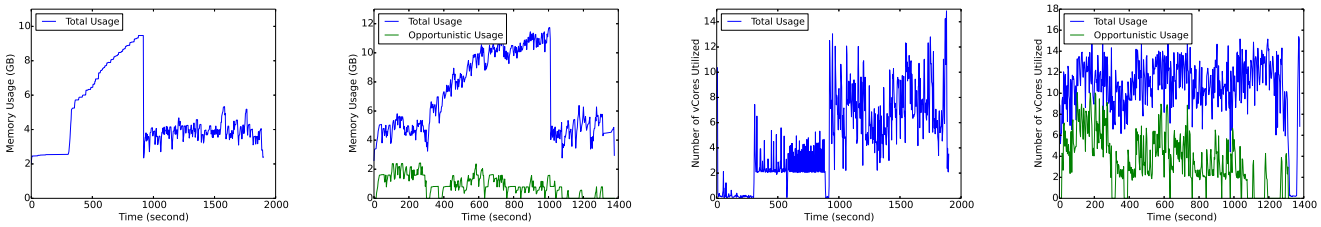
Consistent to the experiments in Sec. IV-B, the average resource utilization becomes much higher under OPERA than that under the Fair policy. As shown in Fig. 8(a), such an improvement comes from the increasing in the number of vCores (see the green parts) that are used through the opportunistic resource allocation. Furthermore, although having the lower resource utilization, the PRESERVE scheme achieves better performance isolation for normally assigned tasks (e.g., tasks from Spark jobs) compared with the other two resource release schemes.

D. Analysis on Resource Contention Relief Schemes

We now present the evaluation of different resource contention relief schemes as introduced in Alg. III-C. Firstly, under the workloads with MapReduce jobs, we notice that the CPU utilization becomes slightly low during some time periods under OPERA with the PRESERVE scheme. We look closely at PRESERVE and find that this scheme conservatively performs opportunistic resource allocation by blocking available opportunistic resources when resource contention happens frequently. Fig. 12 depicts the amount of blocked opportunistic CPU resources under this scheme. When severe resource contention happens at time 700, PRESERVE increases the blocked opportunistic CPU vCores from 2 to 7. As a result, this scheme misses some opportunities for serving more tasks and improving overall resource utilizations. However, PRESERVE can successfully avoid severe performance interference during busy periods, e.g., spikes in CPU usages between 900 seconds and 1,100 seconds in Fig. 6(b).

Fig. 9 further shows the number of killed opportunistically launched tasks across time under two workloads (i.e., without and with Spark jobs), when we use the PRESERVE and NEUTRAL schemes. As described in Sec. III-C, PRESERVE blocks some resources for opportunistic scheduling once it detects resource contention. Since tasks have more oscillating CPU usage patterns at around 700 seconds, see Fig. 6(b), NodeManager detects resource contention and kills opportunistic scheduled tasks more frequently, see Fig. 9(a). Moreover, the PRESERVE policy then tries to avoid more contentions by increasing the blocked vCores at 700 seconds, see Fig. 12, which results in fewer opportunistic scheduled tasks in the following time period, see Fig. 7(b). Compared with the NEUTRAL policy (see Fig. 9(b)) which simply kills opportunistic tasks when resource contention happens, PRESERVE can significantly reduce the number of killed tasks as well as the amount of waste work. For example, the total number of killed tasks in the entire cluster is reduced from 632 to 248, and the amount of wasted work is reduced from 4,575 seconds to 2,400 seconds by using the PRESERVE policy.

Consistently, PRESERVE terminates much less running opportunistic tasks than the NEUTRAL scheme when we have both MapReduce and Spark jobs in the workload. Fig. 9(c) and Fig. 9(d) show the accumulated number of killed tasks as well as the accumulated amount of wasted work on a single cluster node under both the PRESERVE and NEUTRAL schemes. We can see that PRESERVE significantly reduces the number of killed tasks and the amount of wasted work from 36 and 318 seconds under NEUTRAL to 12 and 90 seconds for each cluster node.



(a) Fair - Memory (b) OPERA-PRESERVE- Memory (c) Fair - vCore (d) OPERA-PRESERVE- vCore

Fig. 11: Illustrating actual memory and CPU vCore usages across time on a cluster node under Fair and OPERA-PRESERVE.

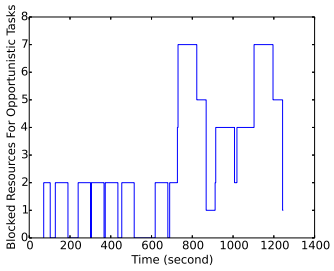


Fig. 12: The amount of blocked opportunistic CPU resources across time under the workload with MapReduce jobs when running the PRESERVE policy.

E. Workloads with Interactive Jobs

Lastly, we investigate how our proposed scheduler with task reservation enabled feature achieves better performance improvement. In order to see the difference, we hereby focus on a more realistic "interactive" scenario, where heterogeneous workloads comes and have long idles periods when they are waiting for user inputs or incoming data streams. To emulate such a scenario, we generate a workload mixed with two Spark jobs and six MapReduce jobs (the same as those in the second set of experiments), but further introduce sleeping periods (e.g., 5 minutes) into the two Spark jobs (i.e., *pathSim*) after parsing the author information but before calculating the author similarities. Such a 5-minute sleeping time is then used to simulate the user thinking time in interactive jobs. Fig. 10 presents the experimental results (e.g. average job execution time and average vCore usage) under this workload.

We first observe that the average execution time of *pathSim* jobs is increased by about 300 seconds because of the additionally injected sleeping periods. Moreover, all the other MapReduce jobs also experience longer execution times under the Fair scheduler, compared to the experiments when there are no sleeping periods, see Fig. 10(a) and Fig. 8(a). In contrast, our OPERA makes better use of idle resources during those sleeping periods by opportunistically scheduling available resources to pending tasks. As a result, compared to Fair, all MapReduce jobs under OPERA experience even better performance improvement, with a reduction of 29.1%, 33.9%, and 39.8% in job execution times when we have PRESERVE, NEUTRAL, and AGGRESSIVE, respectively.

Fig. 11 further shows the actual resource (memory and CPU vCores) usages across time under the Fair and OPERA-PRESERVE policies. As shown in Fig. 11(a) and (c), both memory and CPU resources are under-utilized during the first 5 minutes when we have the Fair scheduler. This is because the two *pathSim* jobs are in the sleep mode during these 5 minutes such that their occupied resources become idle and cannot be assigned to the other waiting tasks. While, our proposed scheduler addresses this issue through the opportunistic resource assignment (see the green curves in Fig. 11(b) and (d)) and thus significantly increases the total cluster resource utilization especially during those sleeping periods. The overall task throughputs are improved as well under the opportunistic resource allocation. Furthermore, by carefully selecting tasks that have short execution times for opportunistic scheduling, we can minimize the side effect, e.g., increase of task execution time, caused by increasing the chance of resource contentions.

V. RELATED WORK

Improving resource efficiency and throughput of cluster computing platforms was extensively studied in recent years. Our previous works [16]–[18] focus on the first generation Hadoop system which adopts coarse-grained resource management. For fine-grained resource management, we proposed a scheduler HaSTE in Hadoop YARN [19] that improves resource utilization using more efficient task packing according to diverse resource requirements of tasks on different resource types and dependencies between tasks. However, HaSTE only considers the task dependencies in the MapReduce framework and assigns resources according to task requests without considering real time resource usages. DynMR [20] presents that reduce tasks in the MapReduce framework bundle multiple phases and have changing resource utilization, and proposes to assemble multiple reduce tasks into a progressive queue for management and backfill map tasks to fully utilize system resources. Their work is closely bounded with the MapReduce framework, and involves complex task management that cannot be easily extended to other frameworks. Quasar [9] designs resource efficient and QoS-aware cluster management. Classification techniques are used to find appropriate resource allocations to applications in order to fulfill their QoS requirements and maximize system resource utilization. Resource assignment in their work is to assign one or multiple nodes to the application, which is different from task assignment in cluster computing. MR-SPS [21] designs a scalable parallel scheduling algorithm which improves scalability and performance of a cluster by managing workload and data locality. Studies [22]–[24] further

investigate storage-related resource management problems, in order to improve the system performance bottlenecked by I/Os. BGMRS [25] is a MapReduce Scheduler based on the Bipartite Graph model. BGMRS reaches the optimal solution of the deadline-constrained scheduling problem by transforming the problem into the "minimum weighted bipartite matching" problem. Study [26] proposes an alternative approach to solve the problem by spiting each job into tasks using an appropriate splitting ratio, and assigns tasks to slave servers based on server processing performance and network resource availability. The previous studies mainly address the inefficient resource utilization caused by the gap between user specified application resource requirements and actual resource usages of applications. While, our work mainly addresses the issue of resource underutilization that is caused by the fluctuating resource usage patterns of tasks.

Resource utilization is of greater importance in large enterprise data centers since increasing utilization by few percentages can save a lot in a large-scale cluster. Recent published works reveal some technique details of Google's Borg [27] and Microsoft's Apollo [28] systems. They both have the design ideas similar to our work, i.e., improving cluster resource utilization by exploring the actual resource usage of running jobs and assigning idle resources to pending tasks. Borg classifies jobs into the categories of high priority and low priority, monitors the resource usage of high priority tasks, and predicts their future resource usage by adding safety margins. If high priority tasks are not using all their reserved resources, resource manager can reclaim these resources and assign to low priority tasks. Low priority tasks may be killed or throttled when high priority tasks require more resources. Apollo starts opportunistic scheduling after all available resource tokens have been assigned to regular tasks. Fairness is achieved by assigning a maximum amount of opportunistic tasks for each job and randomly selecting opportunistic tasks from waiting jobs when scheduling.

Although sharing the similar idea, we differentiate our work from the other designs (e.g., Borg and Apollo) in the following aspects. First, instead of using user-defined task priorities or scheduling the fixed amount of opportunistic tasks for each job that is proportional to that job's resource tokens, we automatically classify tasks according to their estimated execution time and choose short tasks only to get the opportunistic resources in order to avoid severe resource contention as well as the waste of work when regular tasks need resources. As a result, the interference introduced by opportunistic scheduling and the penalty of killing unfinished opportunistic tasks can be minimized under our proposed approach. Second, in our approach, high priority tasks can also benefit from opportunistic scheduling as long as these tasks are classified as short ones by our task classifier, which is different from those under the Borg system. Furthermore, the amount of opportunistic tasks of each job is not fixed (like Apollo does) but instead is determined based on task resource requirements, estimated task execution time, and available opportunistic resource capacity. Finally, our approach uses three resource release schemes that consider different degrees of aggressiveness of opportunistic scheduling. We show that the selection of these resource release schemes can be made based on the consideration of resource utilization, task properties, performance interference, etc.

We also remark that our opportunistic scheduler is complementary with any resource sharing based scheduling algorithms. That is, each job can still get their required resources that are allocated according to a particular scheduling algorithm. Our OpERA further re-assigns those allocated but unused resources to other pending tasks, without any non-negligible impacts on regular scheduling.

VI. Conclusion

In this paper, we developed a novel resource management scheme, named OpERA, to enable the sharing of occupied resources among different tasks (or resource containers). The main objective of OpERA is to improve the overall resource utilization and reduce the executions time for data processing jobs. To meet this goal, OpERA leverages the knowledge of actual runtime resource utilizations to detect underutilized resources and opportunistically re-assigns these resources to other pending tasks. We further classify pending tasks according to their expected running time and only allow the expected short tasks for opportunistic scheduling. By this way, we can guarantee that performance interference can be minimized and killing opportunistically launched tasks does not lead to a significant waste of work. Three different approaches are considered by OpERA to release opportunistic resources when severe resource contention happens. We further develop two approaches to avoid the starvation of both reserved and queued tasks. We implemented OpERA on the top of Hadoop YARN v2.5 and evaluated our proposed scheduler in a cloud environment provided by CloudLab. Diverse workloads mixed with MapReduce and Spark jobs have been produced to evaluate OpERA under different scenarios We also investigate the performance of OpERA under the workload with interactive jobs. The experimental results show that our OpERA is able to achieve up to 39.8% reduction in average job execution time and 30% increase in resource utilizations. In the future, we plan to extend our opportunistic scheduler for streaming data processing. Stream data is often processed in batches or slide windows, e.g. in the Spark Streaming platform. We will develop new method to estimate task lengths using historical data given the batch size or window size is known.

Acknowledgments

This work was partially supported by National Science Foundation Career Award CNS-1452751, National Science Foundation grant CNS-1552525 and AFOSR grant FA9550-14-1-0160.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] T. White, *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[3] (2014) Hadoop Users. [Online]. Available: https://wiki.apache.org/hadoop/PoweredBy

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 10–10.

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[6] N. Marz, "A storm is coming: more details and plans for release," *Twitter Engineering*, vol. 42, 2011.

[7] 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 *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[8] 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 *NSDI*, vol. 11, 2011, pp. 22–22.

[9] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[10] 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 *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 7.

[11] Y. Sun, J. Han, X. Yan, P. S. Yu, and T. Wu, "Pathsim: Meta path-based top-k similarity search in heterogeneous information networks," *VLDB11*, 2011.

[12] A. Verma, Ludmila Cherkasova, and R. H. Campbell, "Aria: Automatic resource inference and allocation for mapreduce environments," in *ICAC'11*, 2011, pp. 235–244.

[13] J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguadé, M. Steinder, and I. Whalley, "Performance-driven task co-scheduling for mapreduce environments," in *Network Operations and Management Symposium (NOMS), 2010 IEEE*. IEEE, 2010, pp. 373–380.

[14] S. J. Russell, P. Norvig, and S. Chakrabarti, "Artificial intelligence: a modern approach."

[15] Cloudlab. [Online]. Available: http://cloudlab.us/

[16] Y. Yao, J. Wang, B. Sheng, C. Tan, and N. Mi, "Self-adjusting slot configurations for homogeneous and heterogeneous hadoop clusters," *Cloud Computing, IEEE Transactions on*, no. 99, p. 1, 2015.

[17] J. Wang, Y. Yao, Y. Mao, B. Sheng, and N. Mi, "Fresh: Fair and efficient slot configuration and scheduling for hadoop clusters," in *Cloud Computing (CLOUD)*, 2014.

[18] Z. Yang, J. Bhimani, Y. Yao, C.-H. Lin, J. Wang, N. Mi, and B. Sheng, "AutoAdmin: Admission Control in YARN Clusters Based on Dynamic Resource Reservation," in *Scalable Computing: Practice and Experience, Special Issue on Advances in Emerging Wireless Communications and Networking*, vol. 19, 2018, pp. 53–67.

[19] Y. Yao, J. Wang, B. Sheng, J. Lin, and N. Mi, "Haste: Hadoop yarn scheduling based on task-dependency and resource-demand," in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*. IEEE, 2014, pp. 184–191.

[20] J. Tan, A. Chin, Z. Z. Hu, Y. Hu, S. Meng, X. Meng, and L. Zhang, "Dynmr: Dynamic mapreduce with reducetask interleaving and maptask backfilling," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 2.

[21] R. Mennour, M. Batouche, and O. Hannache, "Mr-sps: Scalable parallel scheduler for yarn/mapreduce platform," in *Service Operations And Logistics, And Informatics (SOLI), 2015 IEEE International Conference on*. IEEE, 2015, pp. 199–204.

[22] Z. Yang, D. Jia, S. Ioannidis, N. Mi, and B. Sheng, "Intermediate Data Caching Optimization for Multi-Stage and Parallel Big Data Frameworks," in *IEEE International Conference on Cloud Computing*. IEEE, 2018.

[23] Z. Yang, Y. Wang, J. Bhimani, C. C. Tan, and N. Mi, "EAD: Elasticity Aware Deduplication Manager for Datacenters with Multi-tier Storage Systems," in *Cluster Computing*, 2018.

[24] J. Bhimani, Z. Yang, N. Mi, J. Yang, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan, "Docker Container Scheduler for I/O Intensive Applications running on NVMe SSDs," in *IEEE Transactions on Multi-Scale Computing Systems (TMSCS)*, 2018.

[25] C.-H. Chen, J.-W. Lin, and S.-Y. Kuo, "Mapreduce scheduling for deadline-constrained jobs in heterogeneous cloud computing systems."

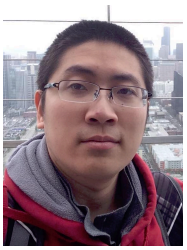
[26] T. Matsuno, B. C. Chatterjee, E. Oki, M. Veeraraghavan, S. Okamoto, and N. Yamanaka, "Task allocation scheme based on computational and network resources for heterogeneous hadoop clusters," in *High Performance Switching and Routing (HPSR), 2016 IEEE 17th International Conference on*. IEEE, 2016, pp. 200–205.

[27] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 18.

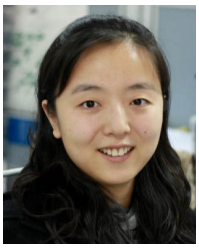
[28] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: scalable and coordinated scheduling for cloud-scale computing," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 2014, pp. 285–300.

Zhengyu Yang received the Ph.D. degree at the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA. He graduated from the Hong Kong University of Science and Technology with a M.S. in Telecommunication in 2011, and he obtained his B.Eng. in Communication Engineering from Tongji University in China. His current research area is mainly on caching algorithm, cloud computing, deduplication, and performance simulations.

Yi Yao is software engineering at VMWare Inc. He received the Ph.D. degree at the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA. He received the B.S. and M.S. degrees in Computer Science from the Southeast University, China, in 2007 and 2010. His research interests include resource management, scheduling, and cloud computing.

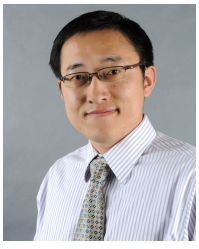


Han Gao is a M.S. student at Northeastern University, majoring in Computer Engineering. He also has a M.S. degree graduated from Penn State University, majoring in Electrical Engineering. He got his B.E. degree in Nankai University, China. His research interests include resource management, scheduling policy, performance evaluation, system modeling, simulation and cloud computing



Jiayin Wang is currently an assistant Professor in Computer Science Department at Montclair State University. She received her Ph.D. degree from University of Massachusetts Boston in 2017. She received her Bachelor degree in Electrical Engineering from Xidian University, China in 2005. Her research interests include cloud computing and wireless networks.

Ningfang Mi is an associate professor at Northeastern University, Department of Electrical and Computer Engineering, Boston. She received her Ph.D. degree in Computer Science from the College of William and Mary, VA in 2009. She received her M.S. in Computer Science from the University of Texas at Dallas, TX in 2004 and her B.S. in Computer Science from Nanjing University, China, in 2000. Her current research interests are capacity planning, MapReduce/Hadoop scheduling, cloud computing, and resource management.



Bo Sheng is an associate professor in Computer Science Department at University of Massachusetts Boston. He received his Ph.D. from College of William and Mary in 2010 (under the supervision of Prof. Li) and my B.S. from Nanjing University (China) in 2000, both in Computer Science. His research interests include mobile computing, big data, cloud computing, cyber security, and wireless networks.