Improving Utilization and Parallelism of Hadoop Cluster by Elastic Containers

Yinggen Xu[*], Wei Chen[†], Shaoqi Wang[†], Xiaobo Zhou[†] and Changjun Jiang[*]
[*]The Key Lab of Embedded System and Service Computing, Tongji University, Shanghai, China
[†]Department of Computer Science, University of Colorado, Colorado Springs, Colorado, USA

***Abstract*—Modern datacenter schedulers apply a static policy to partition resources among different tasks. The amount of allocated resource won't get changed during a task's lifetime. However, we found that resource usage during a task's runtime demonstrates high dynamics and it only reaches full usage at few moments. Therefore, the static allocation policy doesn't exploit the dynamic nature of resource usage, leading to low system resource utilization. To address this hard problem, a recently proposed task-consolidation approach packs as many tasks as possible on the same node based on real-time resource demands. However, this approach may cause resource over-allocation and harm application performance.**

In this paper, we propose and develop *ECS*, an elastic container based scheduler that leverages resource usage variation within the task lifetime to exploit the potential utilization and parallelism. The key idea is to proactively select and shift tasks backward so that the inherent paralleled tasks can be identified without over-allocation. We formulate the scheduling scheme as an online optimization problem and solves it using a resource leveling algorithm. We have implemented *ECS* in Apache Yarn and performed evaluations with various MapReduce benchmarks in a cluster. Experimental results show that *ECS* can efficiently utilize resource and achieves up to 29% reduction on average job completion time while increasing CPU utilization by 25%, compared to stock Yarn.

I. Introduction

Many frameworks have been developed and are widely used to simplify the complexity of data-parallel processing in datacenter clusters. Prominent examples are MapReduce [9], Dryad [17], Spark [24] and Storm [3]. These frameworks generate a job and divide it into multiple execution tasks (e.g., map or reduce task in MapReduce) to perform data-parallel processing. To realize the parallelism, a resource management framework, such as Apache Yarn [21], schedules tasks and distribute them in clusters. Currently, each task runs within a container assigned with static resource.

However, as discussed in recent studies [13], [22], [25] and industry talk [4], a task's resource usage varies over time. A common practice to dealing with the time-varying demand is to provision the container with resource according to tasks' peak demand [13], [4]. This prevents tasks from suffering performance degradation, but causes low resource utilization.

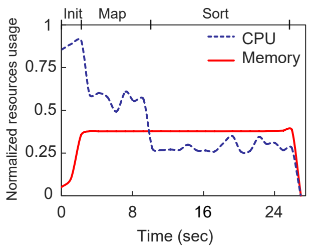
To satisfy the dynamic resource demand, the task resource usage should be known prior to its runtime. However, the full picture of the demand per task can only be collected when the task is completed. To overcome this problem, recent studies proposed to profile task resource usage and execution time from the history logs to estimate the resource demand [13], [7] for the future runs, since jobs are often executed routinely in production environments [18]. Although these methods provide some flexibility in task scheduling, resource assigned to a container still remains static during runtime.

For higher cluster utilization and better parallelism, we find it is necessary to assign a container with dynamic resource capacity in order to run the task with time-varying resource demand. To do it, we utilize a new resource abstraction with dynamic resource capacity, called *elastic container*, to replace the native static container. However, schedulers using task consolidation mechanism are not suitable to schedule elastic containers since they are not aware of such elasticity, leading to harmful resource over-allocation.

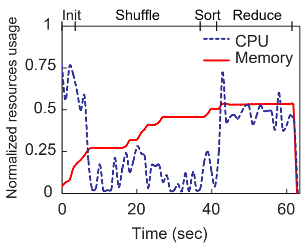
In this work, we exploit the dynamics of task resource demand to improve cluster utilization, parallelism, and application performance. We enable the elastic container using Linux containers (LXC) based on cgroups. To avoid the over-allocation, we design and develop an elastic container-based scheduler, namely *ECS* that cuts the high resource demand by shifting task's context backward. In a nutshell, we make the following contributions in this paper.

- We exploit the dynamics of task resource demand and propose to use elastic containers to run tasks. Compared to a static native container, an elastic container allows the task to acquire resources based on its time-varying demand.
- Based on time-varying resource usage, we design and develop an elastic container-based scheduler, namely *ECS*, to enable resource overcommitment. To avoid resource over-allocation, *ECS* delays the execution for some tasks and resumes their execution when the cluster resource becomes available.
- We formulate the scheduling scheme as an optimization problem based on the predicted task execution profile (time and resource demand) and the available resource. To solve the problem, we design an efficient resource leveling algorithm to derive the optimal scheduling solution.

We have implemented *ECS* on Apache Yarn for enabling elastic containers, and evaluated the approach with various MapReduce benchmark workloads. Experimental results in a Hadoop cluster show that *ECS* can efficiently allocate and utilize cluster resource, achieving up to 29% reduction on average job completion time and 25% improvement on CPU utilization compared to stock Yarn.



(a) Map task.



(b) Reduce task.

Fig. 1. Resource usages of *wordcount* benchmark.

TABLE I
THE DEVIATIONS OF RESOURCE USAGES IN BENCHMARKS.

	WCount	TSort	Kmeans	Bayes	PRank
Map	0.22	0.21	0.14	0.12	0.19
Reduce	0.29	0.28	0.27	0.25	0.33

II. Motivation

A. Resource Usage Analysis and Elastic Container

To understand the potential performance loss due to static resource allocation, we analyze the resource demand of five MapReduce benchmarks in three types of resource consumption. Among the benchmarks shown in Table I, *wordcount* and *terasort* are I/O intensive, *k-means* and *bayes* are CPU intensive, and *pagerank* is a hybrid. We created a 7-node cluster. Each node is configured with dual eight-core Xeon CPU, 128 GB RAM, 6 TB RAID disk drive, and 10Gbps NIC. We deployed Apache Hadoop 2.7.1 and modified NodeManager to collect resource usage traces from containers.

In case studies, each task runs in an isolated container with 2 CPU cores and 2 GB memory. Figure 1 shows the normalized CPU and memory consumptions of map and reduce tasks respectively for *wordcount*. It is obvious that both CPU and memory usages fluctuate significantly. The CPU usage of map or reduce task stays low for more than half of task lifetime. For instance, the CPU stays idle during much of the shuffle phase since both map and reduce tasks are dominated by network IO. However, the peak demand is still very high, i.e., around 0.75 at the initial phase of reduce task. Table I lists the standard deviation in resource usage for tasks in five benchmarks. The higher the deviation is, the more variable is the demand.

In the face of the dynamic usage, configuring resource of a container according to the peak demand can lead to serious wastage. But if resource is allocated based on the low curve, it would become a bottleneck in the high demand phase.

To meet the fluctuant task resource demands, we propose to use elastic containers with dynamic resource capacities implemented through Linux Containers (LXC) via cgroups. Allocated resources within elastic container are dynamically adjusted according to the task resource demand at the phase level. However, container elasticity also brings challenging issues to task scheduling. Next, we conduct a case study to show how an existing scheduling strategy may cancel out the gain by elastic containers, and how we tackle it.

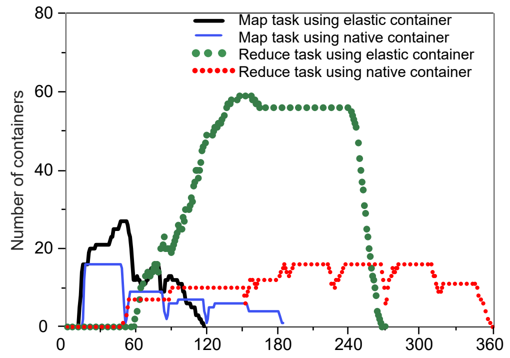


Fig. 2. Number of containers profile for a slave node.

B. Case Study

To illustrate the issues brought up by using an elastic container, we ran two *pagerank* jobs using native containers and elastic containers. The rest experiment environments of two jobs were configured identically to default settings. For the sake of clarity, we only use CPU resource in this case study. Note that the fluctuation of CPU usage in *pagerank* is similar to that of *wordcount* shown in Figure 1.

Figure 2 shows the number of containers for map and reduce tasks respectively. Figure 3 shows the aggregated resource demand of co-hosted tasks over time, which is represented in the percentage of one slave node. From 1st second to 20th second, map tasks start and enter into Init phase. We can see that the numbers of two kinds of containers are similar since the resource usage in an elastic container reaches its peak demand. From the 20th second to 50th second, map tasks using elastic container gradually enter into the map/sort phases with decreased resource usage. Thus, more elastic containers (see Figure 2) can be assigned to enable overcommitment and run more map tasks in parallel. During this period, the resource utilization is improved as shown in Figure 3.

After the 50th second, both kinds of containers start reduce tasks. From to 60th second to about 160th second, more elastic containers are assigned to run reduce tasks. This is due to fact that reduce tasks enter into shuffle phase with low resource usage. Thus, overcommitment is enabled to improve resource utilization. However, when these tasks enter into reduce phase from the 160th second to 270th second, their resource demands increase, leading to resource over-allocation as shown in Figure 3. Compared to the approach with elastic containers, an approach with native containers reserves sufficient resources based on task peak demand statically. However, this results in resource fragmentation and performance degradation. Despite the over-allocation, the approach by elastic containers still outperforms that by native containers.

The following results further illustrate the over-allocation problem due to the default schedulers. Figure 4 shows the details of elastic container-based resource scheduling for a slave node during a short time window. For clarity, 32 tasks were grouped into multiple task sets (i.e., **A** with 5 map tasks, **B** with 6 map tasks, **C** with 7 reduce tasks, **D** with

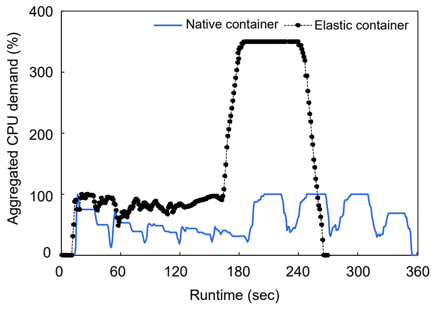
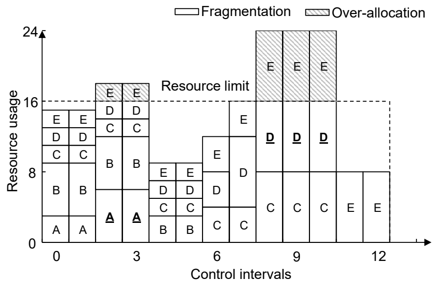
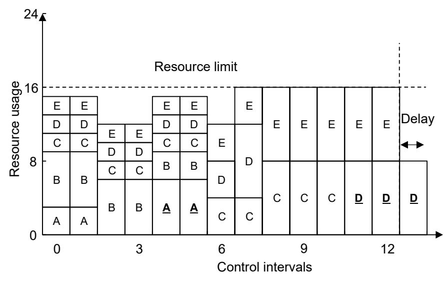


Fig. 3. Resource profile for a slave node.



(a) Default scheduler.



(b) Ideal scheduler.

Fig. 4. Scheduling elastic containers by the default scheduler and by an ideal scheduler.

7 reduce tasks and **E** with 7 reduce tasks). Only the combined resource demand profile of each task set is shown in the figure. The resource profile corresponding to the default schedulers, e.g., FIFO, Fair and Capacity schedulers, is shown in Figure 4(a). The default schedulers just greedily exploit the unused resource to launch new tasks. However, when task set **A** increases its resource demand in the 2nd to 3th intervals (shown as **A**), undesirable resource over-allocation is triggered. The similar situation happens in the 8th to 10th intervals.

The ideal scheduler, as shown in Figure 4(b), can detect the future resource fluctuations and find available resources to avoid over-allocation. For example, it uses the "freezing" function (described in Section IV) implemented in elastic container to suspend task sets **A** and **D**, then restarts them in the 4th to 5th and 11th to 13th intervals shown as **A** and **D**, respectively. The appropriate task shift effectively leveled resource usage on the slave node.

[***Summary***] We have two important observations. First, the default schedulers are inefficient in scheduling elastic containers at runtime. Second, shifting appropriate tasks is critical to avoid over-allocation and to minimize resource fragmentation. However, task shifting poses significant challenges. For example, shifting critical tasks, i.e., a job's last map and reduce tasks, will result in delay to the whole job execution time. Moreover, we need to consider the inter-task dependency between map and reduce tasks. For instance, we should shift map task set **A** instead of the last four intervals of map task set **B** into the 4th to 7th intervals, since reduce task set **D** needs to start reduce phase on the 7th interval. Hence, it is important to consider all these factors when we design an efficient resource scheduling.

III. Problem Formulation

In this section, we formulate the *ECS* task scheduling using elastic containers as an optimization problem that minimizes resource fragmentation and avoids resource over-allocation. The formulation is based on the prediction of task execution and the detection of resource fluctuation. To achieve fine-grained control of task shift, *ECS* divides the execution window of slave node into multiple intervals and builds performance models for task execution estimation.

A. Task Shift

Consider a detected resource fluctuation with duration T. Eeach task has an estimated duration and resource demand at the phase level (e.g., map, sort, shuffle, reduce). The tasks hosted in the same slave node are distinguished as c critical tasks and n noncritical tasks. Each task k has total float TF_k, phase-level duration D_k and resource demand R_k. The noncritical tasks are shifted within the extent of the tasks' float. Since any shift of a critical task will increase the job completion time, the critical tasks are not shifted.

We introduce 0-1 binary vector λ to represent task shift solution. If critical task i is running during interval t, $\lambda_{it} = 1$ and 0 otherwise, where $t = 1, 2, \cdots, D_i$. Similarly, if noncritical task j is running during interval t, $\sigma_{jt} = 1$ and 0 otherwise, where $t = 1, 2, \cdots, D_j + TF_j$. Thus, all possible shift solutions of each noncritical task can be expressed as a 0-1 binary vector. For example, a decision vector σ, expressed as $\sigma_{j1} = 1$, $\sigma_{j2} = 1$, $\sigma_{j3} = 0$, $\sigma_{j4} = 1$, $\sigma_{j5} = 1$ and $\sigma_{j6} = 0$, represent that the noncritical task j having 4 intervals of duration and 2 intervals of total float, is suspended on 3th interval and restarted on 4th interval (see Figure 5). The aggregated resource demand for the slave node in interval t, AR_t, can be written as

$$AR_t = \sum_{i=1}^{c} \lambda_{it} R_{it} + \sum_{j=1}^{n} \sigma_{jt} R_{jt}; \quad t = 1, 2, \cdots, T \tag{1}$$

where R_{it} and R_{jt} are the resource demand of critical task i and noncritical task j in interval t, respectively.

The objective of the optimization model is to minimize the sum of the deviation between the aggregated resource demand AR_t and the resource capacity of slave node C. The objective function is formulated as follows:

$$Minimize\ Z = \sum_{t=1}^{T} |AR_t - C| \tag{2}$$

This can result in a combined reduction in both resource fragmentation and over-allocation with the same weight. Since there is an exponential increase of task execution time with over-allocation in practice [18], a higher weight should be given to"over-allocation" term. Moreover, the absolute value as part of Eq. (2) means that the objective is not linear. Due to these reasons, we reformulate Eq. (2) into a linear objective

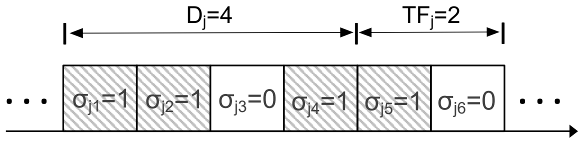


Fig. 5. A shift solution of noncritical task j.

function with additional variables and linear constraints. An equivalent form [11] can be represented as

$$Minimize\, Z = \sum_{t=1}^{T} [\eta F_t + (1-\eta)\, O_t] \tag{3}$$

subject to

$$AR_t - C + F_t - O_t = 0 \tag{4}$$

$$F_t, O_t \geqslant 0 \tag{5}$$

where $t = 1, 2, \cdots, T$. If $C \geqslant AR_t$ then $F_t = C - AR_t$ and $O_t = 0$. If $C \leqslant AR_t$ then $O_t = AR_t - C$ and $F_t = 0$. F_t and O_t are the amount of resource fragmentation or over-allocation in interval t. The weight value η is used to achieve a balance between F_t and O_t.

1) Task Shift Cost: Although task shift is able to effectively level the aggregated resource demand for a slave node, performance deterioration can be caused by two reasons: suspension overhead and shift delay.

Overhead due to task suspension. Suspension overhead occurs when the state data of a suspended task is swapped out from memory to disk by the operating system [5]. The shift solution, which produces optimal objective function value, may include frequent suspension operations and lead to serious system overhead. We use S_j to express the number of suspensions for noncritical task j. Thus, the total overhead of suspending non-critical tasks is represented as follows:

$$\sum_{j=1}^{n} \alpha S_j; \quad j = 1, 2, \cdots, n \tag{6}$$

where α represents the suspension overhead, and S_j can be expressed as function of the binary variable σ_{jt}. When $\sigma_{jt} = 0$ and $\sigma_{j(t+1)} = 1$, task j is restarted in interval $t+1$, meaning that it suffered a suspension. Thus, S_j can be written as

$$S_j = \sum_{t=1}^{D_j + TF_j - 1} max(\sigma_{j(t+1)} - \sigma_{jt}, 0) \tag{7}$$

As shown in Figure 5, for noncritical task j, S_j is equal to 1 due to $\sigma_{j3} = 0$ and $\sigma_{j4} = 1$ when $t = 3$.

Unnecessary cost due to shift delay. Different shift solutions for a task may produce the same objective function value. So the model may select the solution with later finishing times, resulting in an unnecessary shift delay in the task execution. We use L_j to express the shift delay for noncritical task j. Thus, the total delay of shifting noncritical tasks is represented as follows:

$$\sum_{j=1}^{n} \beta L_j; \quad j = 1, 2, \cdots, n \tag{8}$$

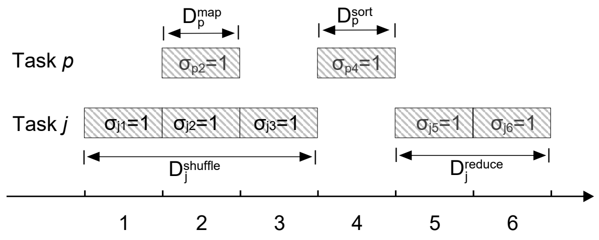


Fig. 6. Example for inter-task dependency.

where β is introduced to emphasize shift to earlier finishing times, and L_j can be written as

$$L_j = max\{t\sigma_{jt} : t = D_j, D_j + TF_j\} - D_j \tag{9}$$

Consider the noncritical task j in Figure 5, where $D_j = 4$ and $TF_j = 2$. L_j is equal to 1 due to the max value of $t\sigma_{jt}$ is 5, which occurs when $t = 5$.

2) Problem Constraints: The decision vectors should maintain two types of constraints for the optimization model. One is duration constraint, and the other is relationship constraint.

Duration constraint. Each task has an estimated execution time. The decision vector must satisfy the duration constraint. Therefore, the sum of running intervals for a noncritical task j must equal to its duration D_j

$$\sum_{t=1}^{D_j + TF_j} \sigma_{jt} = D_j; \quad j = 1, 2, \cdots, n \tag{10}$$

Relationship constraint. The dependency between execution tasks of some frameworks, such as map and reduce task of MapReduce, cannot change. In order to preserve this, a relationship constraint is required as shown in Figure 6. The following constraint guarantees that the reduce phase of reduce task j is started after the sort phase of map task p is finished,

$$S_j^{reduce} \geq F_p^{sort} + 1 \tag{11}$$

where S_j^{reduce} and F_p^{sort} are the reduce phase start time of reduce task j and the sort phase finish time of map task p, respectively. They can be written as

$$S_j^{reduce} = T - max\{(T-t)\sigma_{jt} : t = 1 + D_j^{shuffle}, 1 + D_j^{shuffle} + TF_j\}$$

$$and \sum_{t=1}^{D_j^{shuffle}} \sigma_{jt} = D_j^{shuffle} \tag{12}$$

$$F_p^{sort} = max\{t\sigma_{pt} : t = D_p, D_p + TF_p\} \tag{13}$$

Note that Eq. (12) gives a simple derivation for S_j^{reduce} under the constraint of no shift delay in shuffle phase. $D_j^{shuffle}$ denotes the duration of shuffle phase.

Using the above shift costs and constraints, a complete optimization model for the resource fluctuation problem is shown here

$$Minimize \sum_{t=1}^{T} [\eta F_t + (1-\eta)\, O_t] + \sum_{j=1}^{n} \alpha S_j + \sum_{j=1}^{n} \beta L_j \tag{14}$$

subject to

$$Eq.\ (4),\ (5),\ (10)\ and\ (11)$$

B. Detecting Resource Fluctuation

We use a simple detection algorithm to detect the future resource fluctuation in a node. The resource fluctuation is measured based on the profile of aggregated resource demand of tasks running in a node. For instance, a phase of first task starts at the time t_0, will finish at the time t_1 and requires R_1 amount of resource. A phase of second task starts at the time t_0, will finish at the time t_2 and requires R_2 amount of resource. Let $t_1 < t_2$, then the demand stages of aggregated resource demand are (t_0, t_1) and (t_1, t_2), in which the combined resource demands are R_1 and $R_1 + R_2$, respectively.

To generally formulate the resource demand profile, we introduce some notations as shown in Table II. Let $R_k(s)$ denotes the resource demand from task k between start time $T_i^{s,start}$ and end time $T_i^{s,end}$. The aggregated resource demand for node i at demand stage s, AR_i^s, can be written as

$$AR_i^s = R_1(s) + R_2(s) + \cdots + R_k(s). \tag{15}$$

For each task k, if there exists a phase p that $t_k^{p,start} \leqslant T_i^{s,start}$ and $T_i^{s,end} \leqslant t_k^{p,end}$, then $R_k(s) = R_k^p$; otherwise, $R_k(s) = 0$.

To measure the resource fluctuation, we accumulate the resource fragmentation and over-allocation of every demand stage. That is,

$$RF_i = \sum_s \lambda^s * \left[|AR_i^s - C_i| * (T_i^{s,end} - T_i^{s,start}) \right] \tag{16}$$

where λ is a positive number smaller than one. This parameter gives larger weights on more recent resource fragmentation and over-allocation since their need for leveling is more urgent. As the RF_i be known, we can use a threshold value H_{RF} to judge whether resource fluctuation is detected or not.

C. Estimating Task Execution Time

Tasks' execution can be affected by a number of factors, e.g., data skew in tasks and unpredictable interference from other applications. As such, we exploit a self-adaptive fuzzy model for MapReduce execution estimation based on fuzzy MIMO modeling technique. It is used to predict a task's phase-level durations based on its input data size and resource allocation in each phase [7]. Compared to existing prediction methods [8], MIMO modeling is an ideal choice for capturing the complex relationship between resource allocations and a task's execution, even in the face of interferences.

IV. Design and Implementation

Figure 7 shows the architecture of *ECS*. We implemented the *scheduling optimizer* in Yarn's ResourceManager. The task shift is performed by the scheduling optimizer only when a significant resource fluctuation is detected. It solves the resource fluctuation problem using integer programming and outputs a sequence of task shift solutions that minimize the future resource fragmentation and over-allocation. Then, *ECS* delivers the task shift solutions to each application master via a modified RM-AM RPC protocol.

Job scheduling. *ECS* scheduling (described in Algorithm 1) performs the resource fluctuation-aware scheduling operations on the top of multi-resource fair scheduling. The key improvement by *ECS* is that it will execute overcommitment to consolidate as many tasks as possible on a node as long as the task's demand can be met by the node's current available resource (line 5 $\sim$ line 10).

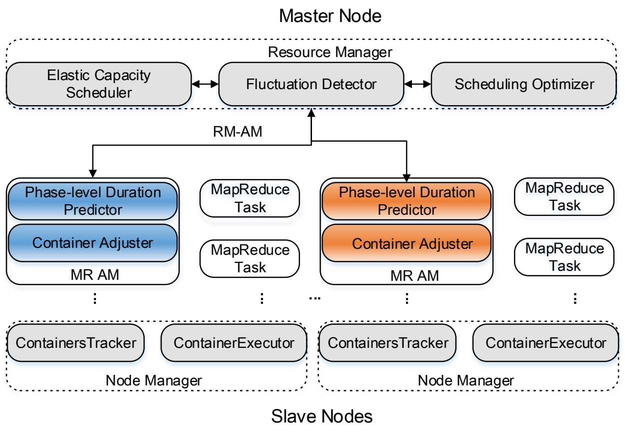


Fig. 7. *ECS* architecture (in gray the Yarn modifications, and in blue and orange two application masters.)

TABLE II
NOTATIONS IN THE FLUCTUATION DETECTION ALGORITHM.

C_i	Resource capacity of node i
R_k^p	Resource demand of task k at its phase p
$t_k^{p,start}$, $t_k^{p,end}$	Sart and end times of phase p at task k
$T_i^{s,start}$, $T_i^{s,end}$	Start and end times of stage s at node i
AR_i^s	Aggregated demand on node i at stage s

However, there is no guarantee that the task's future resource demand at the following phases can be satisfied. To this end, the *fluctuation detector* will check whether aggregated resource demand of tasks on the node exceeds the node capacity (use AR_i^s). If the possibility for over-allocation is identified (line 13), taking the phase-level resource demands and estimated durations (based on history data) of tasks as inputs, the fluctuation detector returns a quantitative value to evaluate the resource fluctuation (line 14). When the measured resource fluctuation is larger than the threshold value H_{RF}, the scheduling optimizer will be performed and output a sequence of task shift solutions (line 16).

Task management. *ECS* maintains two separate queues for the running tasks and suspended tasks. The scheduling optimizer determines which task is to be moved to the suspended queue according to the solution of the resource leveling problem. For example, the execution of noncritical tasks may be shifted, i.e., $S_j > 0$. These suspended tasks wait until their computed restart time is reached. Since the stages of aggregated resource demand are determined by the start and end times of all tasks' phases, the short tasks are more likely to cause more fine-grained demand stages. The number of noncritical tasks and demand stages are two major sources of overhead for scheduling optimizer. To this end, we perform

TABLE III
THE CONTAINER MANAGEMENT INTERFACE.

Function	Class	Functionality
`trackContainer(ContainerId id)`	ContainersTracker	Monitor resource usage for a specific container
`setContainerCapacity(ContainerId id, Res res)`	ContainerExecutor	Set resource capacity of a specific container
`suspendExecutor(ContainerId id)`	ContainerExecutor	Suspend the running task of a specific container
`restartExecutor(ContainerId id)`	ContainerExecutor	Restart the suspended task of a specific container
`adjustContainer(ContainerId id, Res newRes)`	ContainerAdjuster	Pass adjustment command to ContainerExecutor
`suspendContainer(ContainerId id)`	ContainerAdjuster	Pass suspension command to ContainerExecutor
`restartContainer(ContainerId id)`	ContainerAdjuster	Pass restart command to ContainerExecutor

Algorithm 1 *ECS* Scheduling.

```
1: Variables: Demand Stages $DS$; Task information $T_{info}$; resource
   requirement of task $k$ of job $j$ at first phase $R_k^1$;
2: When a heartbeat is received from a free node $i$:
3: Sort jobs in $JQ$ according to the domain fairness;
4: calculate aggregated demand of all tasks $AR_i^s$;
5: for each job $j$ in $JQ$ do
6:     if $C_i - AR_i^1 \geq R_k^1$ then
7:         assign task $k$ to the node $i$;
8:     else
9:         skip current job $j$ and process the next one;
10:    end if
11:    update aggregated demand $AR_i^s$;
12:    for each $s$ in $DS$ do
13:        if $AR_i^s > C_i$ then
14:            $RF_i = FluctuationDetector(T_{info})$;
15:            if $RF_i > H_{RF}$ then
16:                $SchedulingOptimzer(T_{info})$;
17:            end if
18:            break the loop;
19:        end if
20:    end for
21: end for
```

a k-means clustering algorithm on the durations of tasks. The clustering algorithm divides tasks into multiple clusters. The tasks in the cluster with smaller durations are considered as short task set. We do not shift the executions of short tasks (i.e., serve as critical task set), and allocate resource according to their peak resource requirement. Accordingly, the rest of task sets may be shifted so as to smooth resource fluctuation.

Implementation details. We implemented a series of container runtime management APIs based on cgroups subsystem, as listed in Table III. The `ContainersTracker` is used to track resource usage of containers. It reads `cpuacct.usage` file of the `cpuacct` subsystem to accumulate task's CPU usage, and accesses `memory.usage_in_bytes` file located in the `memory` subsystem to collect memory usage. The above information is periodically communicated to task's `ApplicationMaster` and used to estimate resource demand and predict duration for each phase of tasks.

Additionally, we implemented elastic containers by modifying class `LinuxContainerExecutor`. We added a new function `setContainerCapacity()`, which supports changing a container's capacity on the fly. For dynamic adjustment of CPU resource, we rewrite the container's `cpu.shares`, `cpu.cfs_period_us` and `cpu.cfs_quota_us` files through the `cpu` subsystem. The memory size is adjusted by rewriting `memory.limit_in_bytes` and `memory.soft_limit_in_bytes` files of the `memory` subsystem. We have also employed the `freezer` subsystem in cgroups to implement two "freezing" functions `suspendExecutor()` and `restartExecutor()`.

TABLE IV
RESOURCE CONFIGURATIONS OF BENCHMARKS.

Phase	I/O	CPU	Hybrid
Map	1core, 1.5GB	1core, 1.5GB	1core, 1.5GB
Sort	0.5core, 2GB	0.5core, 2GB	0.5core, 2GB
Shuffle	0.3core, 1GB	0.2core, 1GB	0.3core, 1GB
Sort	0.5core, 1.5GB	0.5core, 1.5GB	0.5core, 1.5GB
Reduce	1core, 2GB	1core, 2GB	1core, 2GB

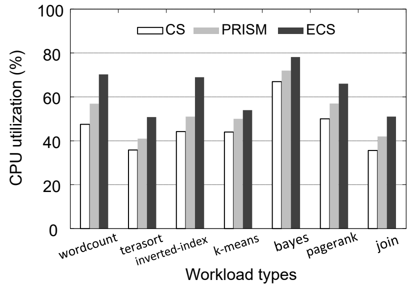
To solve the optimization problem in Eq. (14), we transformed its internal maximum functions as linear inequality constraints for inclusion in the model. We used a 0-1 integer programming solver coded in LINGO to find the optimal task shift solution. The programming solver is integrated into the scheduling optimizer through the Java Native Interface (JNI). Based on the observations, we empirically set the weight α, β as 0.3, 0.05 and η as 0.4 for the optimization model.

V. EVALUATION

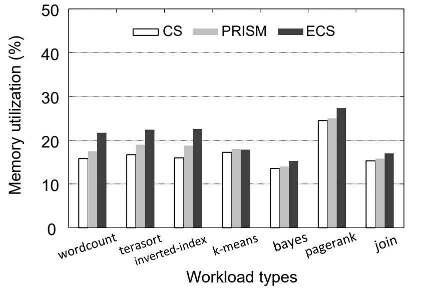
A. Testbed Setup

We performed evaluations of *ECS* on a cluster composed of 7 Sugon I620-G20 (16-core CPUs and 128 GB RAM) and 9 ThinkServer RD650 (20-core CPUs and 128 GB RAM). Each machine has 6 TB hard disk, and are connected with 10 Gbps Ethernet. We deployed *ECS* in the Hadoop version 2.7.1 and each server ran Ubuntu Linux with kernel 4.2.0. Two servers were configured as the `ResourceManager` and `NameNode`, respectively. The rest 14 servers ran as slave nodes for HDFS and task execution. The HDFS block size is set to 128MB and the number of replicas of HDFS is set to 3 in our experiments.

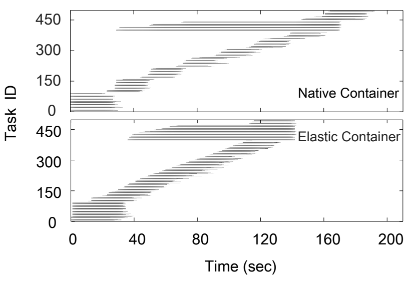
For comparison, we also implemented a recently proposed fine-grained resource-aware MapReduce scheduler PRISM [25]. PRISM improves resource utilization and reduces job completion time by dividing tasks into phases and performing resource scheduling at phase level. However, each task has multiple running phases in the execution progress. Simply



(a) CPU.

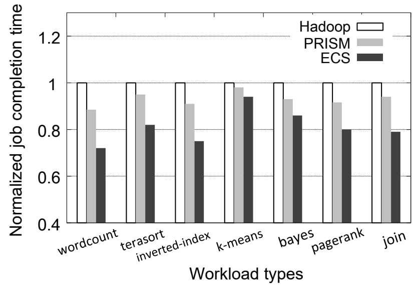


(b) Memory.

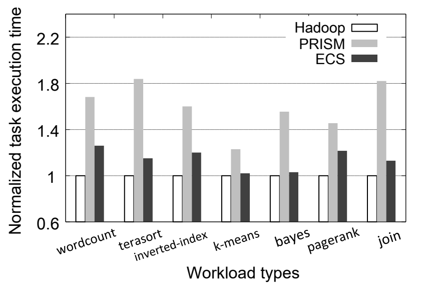


(c) Task execution timeline.

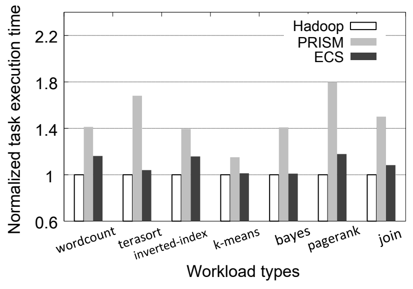
Fig. 8. The cluster resource utilization and task parallelism by Capacity Scheduler, PRISM and *ECS*.



(a) Job completion time.



(b) Map task execution time.



(c) Reduce task execution time.

Fig. 9. The normalized job completion and task execution time by Capacity Scheduler, PRISM and *ECS*.

scheduling each phase of tasks incurs large scheduling delays and leads to serious suspension overheads. Since serious suspension overhead can slow down tasks, resulting in more resources fragmentation. Furthermore such fine-grained approach also significantly increases the system complexity.

B. Workloads

To evaluate the effectiveness of *ECS*, we used a set of representative MapReduce benchmarks from PUMA [2] and HiBench [16]. These benchmarks can be grouped into three major categories: I/O-intensive, CPU-intensive, and hybrid, according to resource consumption characteristics. The I/O intensive benchmarks include: *wordcount*, *terasort* and *inverted-index*. As resource demands are highly variable during task execution, such benchmarks are sensitive to resource over-commitment. The CPU intensive benchmarks include *k-means* and *bayes*, which have a relatively stable resource usage over time. For hybrid benchmarks, we choose *pagerank* and *join* for evaluation. As shown in Table IV, we configured phase-level resource demands for these MapReduce benchmarks based on previous studies [19], [14] and our experimental experiences.

C. *Effectiveness of* ECS

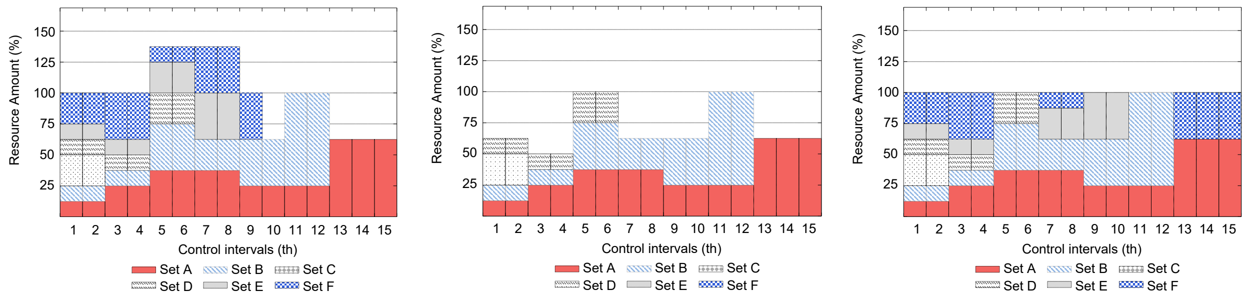
Improving resource utilization and parallelism. Figure 8 compares the cluster resource utilization and task parallelism for Capacity Scheduler, PRISM, and *ECS*. We measure the utilization as the ratio between the aggregated resource usage of co-hosted tasks and the server's capacity. Figures 8(a) and 8(b) shows the CPU and memory usage for individual benchmarks, respectively. The results show that *ECS* outperforms Capacity Scheduler by 18% and 4% in CPU and memory utilization, respectively; and outperforms PRISM by 12% and 3% respectively. The underutilized resources due to dynamic demand were used to launch pending tasks in parallel, as the result the queuing delay for tasks are reduced. Note that with different job characteristics, the improvement in resource utilizations is different. For *k-means* and *bayes* benchmarks which have stable resource usage, the utilization improvement is marginal. However, For *wordcount* and *inverted-index* benchmarks, *ECS* results in roughly 25% higher CPU utilization compared to Capacity Scheduler. But due to serious scheduling delays and suspension overheads, PRISM brings less improvement over resource utilization than *ECS* does.

To understand the potential benefits of improving resource utilization by *ECS*, we also compare detailed executions of two *wordcount* jobs (i.e., with native container and elastic

TABLE V
SCHEDULE DATA FOR TASK SETS IN FIGURE 10.

Task Set	Duration	ES	EF	LS	LF	Total Float
A	15	1	15	1	15	0
B	12	1	12	4	15	3
C	2	1	2	14	15	13
D	6	1	6	10	15	9
E	8	1	8	8	15	7
F	9	1	9	7	15	6

Note: ES=earliest start time; EF=earliest finish time.



(a) Capacity Scheduler. (b) Contention-aware Scheduler. (c) *ECS*.

Fig. 10. The resource allocations by Capacity Scheduler, Contention-aware Scheduler and *ECS*.

container). Figure 8(c) shows the tasks execution timelines for the two different job instances. Apparently, *ECS* significantly reduces the number of task waves for the job by using elastic container and speeds up the job completion. This is due to the fact that more unused resource is allocated for pending tasks compared to the stock Hadoop with native container. In this case, we find the degree of parallelism achieved by *ECS* is almost 1.5 times higher than that of stock Hadoop while they complete with the same number of tasks.

Reducing job completion time. We have shown that *ECS* is effective in improving resource utilization and task parallelism. Here, we study how does the improvement help reduce overall job completion time. We use the job completion time in the stock Hadoop as the baseline and compare it with the normalized job completion time of *ECS* and PRISM. Figure 9(a) shows that for I/O intensive benchmarks, e.g., *wordcount*, *terasort* and *inverted-index*, *ECS* outperforms stock Hadoop by 29%, 19% and 26%, respectively. *ECS* also outperforms PRISM by 17%, 14%, and 17% in these benchmarks. For hybrid benchmarks, such as *pagerank* and *join*, *ECS* achieved 20% and 21% shorter job completion time than the stock Hadoop did, respectively. Relatively, *ECS* was less effective for CPU intensive benchmarks, e.g., *k-means* and *bayes*. This is due to the fact that for those benchmarks, especially *k-means*, *ECS* only had marginal improvement on the resource utilization and task parallelism. Thus, such benchmarks are not favored by the optimization. Figures 9(b) and 9(c) show that the average task execution time increases slightly by 11% when applying elastic container to shrink resource allocations for tasks. However, the performance deterioration of individual tasks contributes more opportunities to run new tasks in parallel. It also reveals that PRISM significantly delays tasks completion due to its fine-grained scheduling.

D. *Effectiveness of Resource Leveling*

Consider the schedule data in Table V, which shows the relevant information of five task sets. Task set **A** is the critical task set and its position is fixed in order to maintain the original window duration. We compare *ECS* with Hadoop Capacity Scheduler. As discussed in previous experiment, when applied to elastic container, default schedulers are more likely to incur serious resource over-allocation. In this subsection, we also implement a variation of Capacity Scheduler: contention-aware scheduler. Capacity Scheduler greedily schedules tasks to a node without regard to its future resource availability, as long as the current residual capacity of that node is sufficient to serve at least one task. The basic idea of contention-aware scheduler is taking into account the knowledge of a node's future varying capacity so as to avoid over-allocation. Thus, if sufficient resources are not available in the near future, then the node will not be assigned tasks.

Figure 10 depicts the interval resource profiles for a node when running task sets **A-F** by different approaches. Figure 10(a) shows that Capacity Scheduler greedily allocated resources to all waiting task sets in the 1st to 4th interval. However, its obliviousness of future resource availability on the node caused significant resource fluctuation, which led to serious resource fragmentation and over-allocation in the 5rd to 15th interval. Figure 10(b) shows that contention-aware scheduler rejected task sets **E** and **F** since there are not enough resources available in the 5rd to 6th and 11rd to 12th intervals. Applying the contention-aware approach efficiently eliminated over-allocation, but additionally increased resource fragmentation by 15%. In contrast, Figure 10(c) shows that the executions of task sets **E** and **F** were suspended and restarted flexibly according to the future resource fragmentation and over-allocation. This is due to the fact that *ECS* scheduler shifted more task executions from the peak value interval to the valley value interval. Thus, we conclude that *ECS* effectively leveled resource usage on the node.

E. *Overhead and Scalability*

The overhead of *ECS* mainly comes from three sources: (1) the time required to activate resource adjustment according to allocated elastic container when task progresses from one phase to another (around 50ms); (2) the time required to suspend and restart tasks (around 0.7s); (3) the time required to perform scheduling optimizer (around 2.1s). We also found this overhead is sensitive to control interval and task set size.

VI. Related Work

Resource management: Yarn [21] is the second generation of Hadoop. Its major contributions are the container resource abstraction and the decoupling of resource management and programming framework. Corona [1] is Facebook's in house scheduling solution. It scales more easily and offers better cluster resource utilization based on the actual resource demands of workloads. Mesos [15] abstracts CPU, memory, storage, and other resources away from machines, enabling fault-tolerant and elastic distributed systems to easily be built and run effectively. All of these resource managers assume that resource usage of execution instances from various computing frameworks keep relatively stable over their lifetime.

Shared use of non-isolated resources: In existing resource management platforms, it is allowed only to reserve and isolate CPU and memory usage for each task [21], [15]. A few studies start to address the potential resource contention on other shared resources such as network and disk. Tetris [13] packs tasks with heterogeneous demands to servers based on their peak requirements along multiple resources, to avoid resource fragmentation and over-allocation. Quasar [10] use classification techniques to co-locate tasks of complementary network and disk demand for the best performance. Prophet [22] focuses on time-varying network and disk resource demand in Spark. Our work differs from these efforts in that we consider the dynamic CPU and memory allocation based on elastic container, so as to satisfy the time-varying demands of executor instances from various computing frameworks.

Cluster schedulers: Many studies have shown that applications performance in large-scale data-parallel systems can be significantly improved by various scheduling techniques [6], [9], [17]. Dominant Resource Fairness (DRF) [12] uses a max-min fair allocation policy to maximize the share of critical resource allocated to a user. PRISM [25], a fine-grained MapReduce scheduler, perform task scheduling at the phase level to achieve higher execution parallelism and resource utilization. Bubble-flux [23] and Heracles [20] adopt a task consolidation mechanism to enable latency-critical workloads to be colocated with "safe" batch jobs without SLO. BIG-C [5] firstly proposed to leverage lightweight containers to enable preemptive and low latency scheduling in clusters with heterogeneous workloads. But these schedulers are not applicable to scheduling elastic containers with dynamic resource capacities.

VII. CONCLUSIONS

Typical cluster schedulers assume that resource usage for each task remains relative stable and thus are oblivious of the dynamics of task resource demand at runtime. In this paper, we have proposed and developed *ECS*, an elastic container-based scheduler, that adjusts the resource capacity of elastic containers at runtime and avoid resource over-allocation. We have implemented *ECS* in Apache Yarn and evaluated its effectiveness with various MapReduce benchmarks. *ECS* is able to achieve up to 29% reduction on job completion time and 25% improvement on CPU utilization, compared to stock Yarn. It is effective for all benchmarks, and most efficient for I/O intensive benchmarks.

In future work, we will deploy more computing frameworks (e.g., Spark and Storm) on Apache Yarn equipped with *ECS*.

ACKNOWLEDGEMENT

This research was supported in part by Shanghai Science and Technology Research Project #17511104502 and U.S. NSF grant CNS-1422119.

REFERENCES

[1] Hadoop corona: the next version of mapreduce. https://github.com/facebookarchive/hadoop-20/tree/master/src/contrib/corona.

[2] PUMA: Purdue mapreduce benchmark suite. https://engineering.purdue.edu/~puma/pumabenchmarks.htm.

[3] Storm. http://storm-project.net/.

[4] The Next Generation of Apache Hadoop. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kambatla/.

[5] W. Chen, J. Rao, and X. Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *Proc. of USENIX ATC*, 2017.

[6] D. Cheng, Y. Chen, X. Zhou, D. Gmach, and D. Milljichic. Adaptive scheduling of parallel jobs in spark streaming. In *Proc. of IEEE INFOCOM*, 2017.

[7] D. Cheng, X. Zhou, P. Lama, J. Wu, and C. Jiang. Cross-platform resource scheduling for spark and mapreduce on yarn. *IEEE Transactions on Computers*, 66:1341–1353, 2017.

[8] E. Coppa and I. Finocchi. On data skewness, stragglers, and mapreduce progress indicators. In *Proc. of ACM SoCC*, 2015.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, 2008.

[10] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proc. of ACM ASPLOS*, 2014.

[11] S. M. Easa. Resource leveling in construction by optimization. *Journal of construction engineering and management*, 115:302–316, 1989.

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

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

[14] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer. Online parameter optimization for elastic data stream processing. In *Proc. of ACM SoCC*, 2015.

[15] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. of USENIX NSDI*, 2011.

[16] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Proc. of IEEE ICDEW*, 2010.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of ACM EuroSys*, 2007.

[18] P. Lama and X. Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proc. of ACM ICAC*, 2012.

[19] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller. Mronline: Mapreduce online performance tuning. In *Proc. of ACM HPDC*, 2014.

[20] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: improving resource efficiency at scale. In *Proc. of ACM/IEEE ISCA*, 2015.

[21] 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 *Proc. of ACM SoCC*, 2013.

[22] G. Xu, C.-Z. Xu, and S. Jiang. Prophet: Scheduling executors with time-varying resource demands on data-parallel computation frameworks. In *Proc. of IEEE ICAC*, 2016.

[23] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proc. of ACM/IEEE ISCA*, 2013.

[24] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *Proc. of USENIX HotCloud*, 2010.

[25] Q. Zhang, M. F. Zhani, Y. Yang, R. Boutaba, and B. Wong. Prism: fine-grained resource-aware scheduling for mapreduce. *IEEE Transactions on Cloud Computing*, 3:182–194, 2015.