DOI 10.12694/scpe.v19i1.1376
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 19, Number 1, pp. 53–67. http://www.scpe.org
© 2018 SCPE

AUTOADMIN: AUTOMATIC AND DYNAMIC RESOURCE RESERVATION ADMISSION CONTROL IN HADOOP YARN CLUSTERS

ZHENGYU YANG, JANKI BHIMANI, YI YAO, CHO-HSIEN LIN,[*] JIAYIN WANG,[†] NINGFANG MI,[‡] AND BO SHENG[§]

Abstract. Hadoop YARN is an Apache Software Foundation's open project that provides a resource management framework for large scale parallel data processing, such as MapReduce jobs. Fair scheduler is a dispatcher which has been widely used in YARN to assign resources fairly and equally to applications. However, there exists a problem of the Fair scheduler when the resource requisition of applications is beyond the amount that the cluster can provide. In such a case, the YARN system will be halted if all resources are occupied by ApplicationMasters, a special task of each job that negotiates resources for processing tasks and coordinates job execution. To solve this problem, we propose an automatic and dynamic admission control mechanism to prevent the ceasing situation happened when the requested amount of resources exceeds the cluster's resource capacity, and dynamically reserve resources for processing tasks in order to obtain good performance, e.g., reducing makespans of MapReduce jobs. After collecting resource usage information of each work node, our mechanism dynamically predicts the amount of reserved resources for processing tasks and automatically controls running jobs based on the prediction. We implement the new mechanism in Hadoop YARN and evaluate it with representative MapReduce benchmarks. The experimental results show the effectiveness and robustness of this mechanism under both homogeneous and heterogeneous workloads.

Key words: Cloud Computing, MapReduce, Hadoop, YARN, Scheduling, Resource Management, Admission Control, Big Data, Scalable Computing, Cluster Computing

AMS subject classifications. 68M14, 68P05

1. Introduction. Large scale data analysis is of great importance in a variety of research and industrial areas during the age of data explosion and cloud computing. MapReduce [11] becomes one of the most popular programing paradigms in recent years. Its open source implementation Hadoop [5] has been widely adopted as the primary platform for parallel data processing [1]. Recently, the Hadoop MapReduce ecosystem is evolving into its next generation, called Hadoop YARN (Yet Another Resource Negotiator) [19], which adopts fine-grained resource management for job scheduling.

The architecture of YARN is shown in Figure 1.1. Similar to the traditional Hadoop, a YARN system often consists of one centralized manager node running the ResourceManager (RM) daemon and multiple work nodes running the NodeManager (NM) daemons. However, there are two major differences between YARN and traditional Hadoop. First, the RM in YARN no longer monitors and coordinates job execution as the JobTracker of traditional Hadoop does. Alternatively, an ApplicationMaster(AM) is generated for each application in YARN to coordinate all processing tasks (e.g., map/reduce tasks) from that application. Therefore, the RM in YARN is more scalable than the JobTracker in traditional Hadoop. Secondly, YARN abandons the previous coarse grained slot configuration used by TaskTrackers in traditional Hadoop. Instead, NMs in YARN consider fine-grained resource management for managing various resources. e.g., CPU and memory, in the cluster.

On the other hand, YARN uses the same scheduling mechanisms as traditional Hadoop and supports the existing scheduling policies (such as FIFO, Fair and Capacity) as the default schedulers. However, we found that a resource (or "container") starvation problem exists in the present YARN scheduling under Fair and Capacity. As mentioned above, for each application in YARN, an ApplicationMaster is first generated to coordinate its processing tasks. Such an ApplicationMaster is indeed a special task in the YARN system, which has a higher priority to get resources (or containers) and stays alive without releasing resources till all processing tasks of that application finish. Consequently, when the amount of concurrently running jobs becomes too high, for example, a burst of jobs arrived, it is highly likely that system resources are fully occupied by ApplicationMasters of

[*] Department of Electrical & Computer Engineering, Northeastern University, 360 Huntington Ave., Boston, MA, USA, 02115 (yangzy1988@coe.neu.edu, bhimani@ece.neu.edu, yyao@ece.neu.edu, jacks953107@ece.neu.edu).

[†] Computer Science Department, Montclair State University, 1 Normal Ave, Montclair, NJ, USA, 07043 (wangji@montclair.edu)

[‡] Department of Electrical & Computer Engineering, Northeastern University, 360 Huntington Ave., Boston, MA, USA, 02115 (ningfang@ece.neu.edu).

[§] Deptartment of Computer Science, University of Massachusetts Boston, 100 Morrissey Boulevard, Boston, MA, USA 02125 (shengbo@cs.umb.edu).

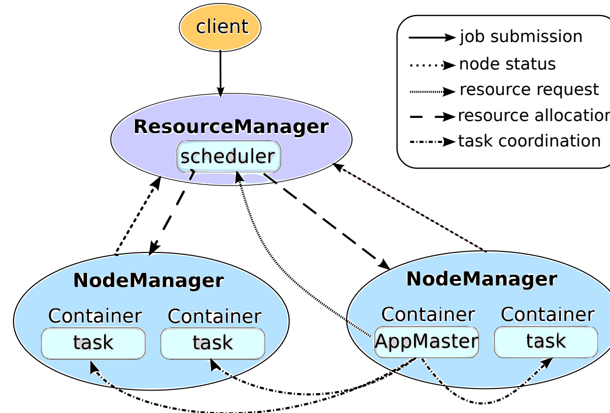

Fig. 1.1. *YARN Architecture. When a client submits a job to the ResourceManager (RM), the RM communicates with a NodeManager (NM) to launch an ApplicationMaster(AM) for running that job. The AM is responsible for submitting resource requests to the RM and negotiating with a set of NMs to launch containers for processing the tasks of the job.*

these running jobs. A *resource deadlock* thus happens such that each ApplicationMaster is waiting for other ApplicationMasters to release resources for running their application tasks.

To solve this problem, one could kill one or multiple jobs and their ApplicationMasters to break the deadlock. A preventative solution is to apply the admission control mechanism to control the number of concurrently running jobs (or ApplicationMasters) in the system and thus reserve resources to run processing tasks. By this way, the previous deadlock of resource waiting can be avoided. However, choosing a good admission control mechanism (e.g., how many jobs admitted in the system?) is difficult when efficiency of the system is also an important consideration. If we reserve too many resources for running processing tasks, then the concurrency of jobs will be sacrificed because the resources for starting ApplicationMasters are limited. In contrast, the running jobs may take too much time to wait for resources for running their tasks and thus be delayed dramatically if we admit too many jobs in the system. Furthermore, MapReduce applications in real systems are often heterogeneous, with job sizes varying from a few tasks to thousands of tasks and different submission rates [7]. A static and fixed admission control mechanism thus cannot work well.

Therefore, the objective of this work is to design a new admission control mechanism which can automatically and dynamically decide the number of concurrently running jobs, with the goal of avoiding the resource waiting deadlock and meanwhile preserving good system performance. The main performance metric we take into consideration is the makespan (i.e., total completion length) of a given set of Mapreduce jobs. There are three main components of our admission control mechanism.

- **Resource Information Collector** (**RIC**): this module periodically records the resource usage information of the running jobs of each worker node.
- **Reserved Resource Predictor** (**RRP**): this is the major component of this mechanism. RRP resides in the RM and uses the information provided by the **RIC** module to decide the amount of resources that need to be reserved for regular tasks.
- **Application Resource Controller** (**ARC**): this module controls the admission of incoming applications based on the current amount of reserved resources determined by the **RRP** module. All applications that are not allowed to enter the system for running will be queued.

We implement these components in a YARM platform (e.g., Hadoop YARN) and evaluate our new admission control mechanism with a suite of representative MapReduce benchmarks. The experimental results demonstrate the effectiveness of the proposed mechanism under both simple and complex workloads.

The remainder of the paper is organized as follows. In Section 2, we introduce the deadlock problem of YARN systems and show our preliminary investigation on static admission control. Our proposed mechanism is proposed in Section 3. Evaluation of this mechanism is presented in Section 4. We describe the related works in Section 5 and conclude in Section 6.

2. Motivation.

2.1. Background. In a YARN system, each application needs to specify the resource requirements of all their tasks, such as ApplicationMasters, map tasks and reduce tasks. The resource requirement is a bundle of physical resources with the format as <2 CPU, 1GB RAM>, for example. Then, a task from that application will only be executed in a container whose resource capacity (e.g., 2 CPU and 1GB RAM) is equal to that task's resource demand. Meanwhile, the NodeManager (NM) keeps monitoring the actual resource usage of that container and could kill that container if the actual usage exceeds the granted resource amount.

The ResourceManager (RM) acts as a central arbitrator, dispatching available containers to various competing applications in light of each application's resource demand and the scheduling policy. NMs residing in the work machines (i.e., slave nodes) associate with the RM to manage the resource; they periodically report the information of resource and containers usage to the RM through the *heartbeat* message in YARN. The scheduler in the RM then schedules the queuing jobs based on the waiting resource requests of applications and the current residual resource amount on each slave node.

When an application is submitted, the RM needs to first allocate resources for launching the ApplicationMaster container on a work node according to the resource requirement specified by the client. The launched AM will then send resource requests for processing remaining tasks of that application to the RM. The RM responses with resource allocations on NMs and the AM then communicate with the NMs for launching containers and executing tasks. The AM is also responsible for monitoring and coordinating the execution of other processing tasks, e.g., re-submitting failed tasks, gradually submitting reduce tasks according to the progress of map tasks, etc.. Therefore, the AM only terminates and releases the resources its container occupies after all the remaining tasks of its application are finished.

Moreover, the AM is the head of a job and it must be launched firstly when processing the job. Since AM needs to harness some resource to running the job's tasks, it will issue requests to RM for allocating the Task Container (TC) to process the tasks. While RM accepts the requests then it will response to AM that which nodes have the available resource. The AM then continuously negotiates with the NMs of those nodes to running the tasks. Finally, while all the tasks are processed the AM can be finished and the RM will release the AMC. In conclusion, the RM and NM starts running when the YARN system is set up; however, the AM launched upon a container is based on the condition that a job is submitted to the RM and the RM accepts to processing it; then, the AM can start its negotiating mission, relative state [17] is as Fig. 2.1.

2.2. The Deadlock Problem of Fair Scheduler in YARN. Since an ApplicationMaster is the first launched task of each job and will not release its resource until the associated job finishes, the number of AM tasks running in a YARN system is always equal to the number of concurrently running jobs. It is not a problem if a small amount of jobs are concurrently running in the system. However, it is possible that these AMs use up most or even all of the resources during a busy period, for example, when a large amount of users submit many ad-hoc MapReduce query jobs. In such a case, the YARN system faces a *deadlock* problem, i.e., none of these AMs could get resources for running their map/reduce tasks. Consequently, overall efficiency of the YARN system is severely degraded because the AMs are occupying most resources yet waiting for others to release resources.

Therefore, our goal in this work is to design a new mechanism that can automatically control the resource reservation in the YARN system such that the deadlock problem can be avoided even under busy conditions when resource demands of AMs exceed the entire cluster capacity. Moreover, this new mechanism can dynamically adjust the strategy for resource reservation according to the present workloads in order to improve the efficiency of the YARN system, e.g., minimizing the makespan (i.e., the overall completion length) of a set of MapReduce jobs.

2.3. Preliminary Investigations. The basic solution of preventing the deadlock situation is to reserve a fixed amount of resources for non-AM processing tasks. However, as we discussed before, the system performance could be degraded if the reservation number is not carefully chosen. We first modify the Fair scheduler to support the static resource reservation and investigate the impact of resource reservation on the performance. Two sets of experiments are conducted to evaluate the performance of resource reservations. For the sake of simplicity, we only consider the cpu resource in these experiments (i.e., all jobs are cpu intensive). The YARN cluster we

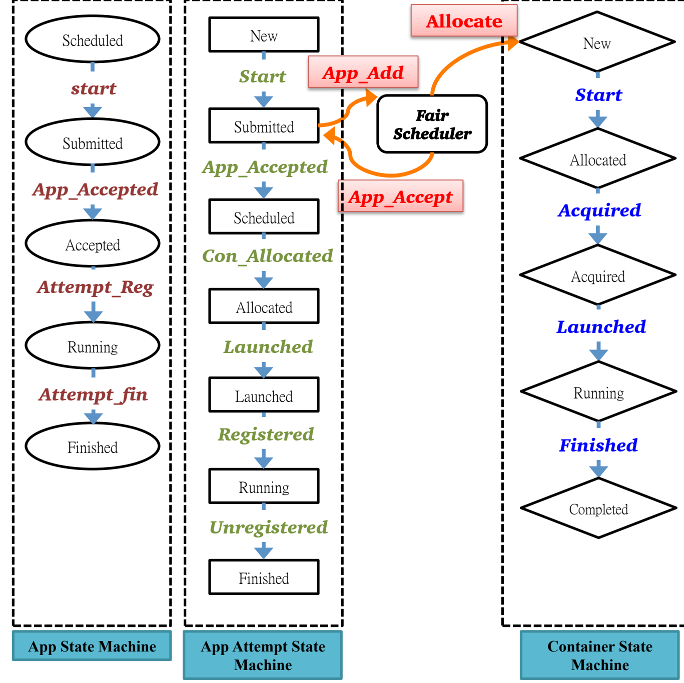

Fig. 2.1. *Job Processing State Graph in YARN. When a job is submitting, the scheduler (in this case is Fair Scheduler) will depend on the resource usage situation to decide to accept the application and allocate the AM container or not.*

used has the resource pool of 64 vcores.

Case 1: We submit a batch of 66 *terasort* jobs to the YARN cluster. Each job will process 100 MB input data generated through *teragen*. In this case, we fix the amount of resource requirement of each job's map and reduce tasks to be equal to 4 vcores in all the experiments. At the same time, we tune the resource requirements of each job's AM from 1 vcore to 4 vcores in different sets of experiments. The makespans of these jobs under different resource requirement settings and different static resource reservations are shown in Figure 2.2. As observed from the figure, the number of reserved resources has great impacts on the makespan of jobs and the optimal configuration of resource reservation is changing as the resource requirement of AMs changes, see the lower plot in Figure 2.2.

Case 2: In this case, we submit the same 66 *terasort* jobs to the cluster. We fix the resource requirement of each job's AM to be equal to 4 vcores and tune the resource-requisition of map and reduce tasks of each job from 1 vcore to 4 vcores in different sets of experiments. The makespans of these jobs under different resource requirement settings and different static resource reservations are shown in Fig. 2.3. Similar to case 1, we can observe that the optimal reservation point (i.e., the number of reserved vcores) is changing as shown in the lower plot.

Based on the above observations, we conclude that the optimal amount of reserved resources is related with the characteristic of workloads. Generally, the best reservation number (R_R) is inversely proportional to the resource requirement of AM (AMC_R), but proportional to the resource demands of other tasks (TC_R), i.e., $R_R \propto \frac{TC_R}{AMC_R}$.

3. Algorithm Design. As shown in Section 2.3, the number of reserved resources has a great impact on system efficiency, and the optimal value could change under different workloads. Therefore, we design a new admission control mechanism for YARN that can dynamically predict the optimal tuning point (i.e., the amount

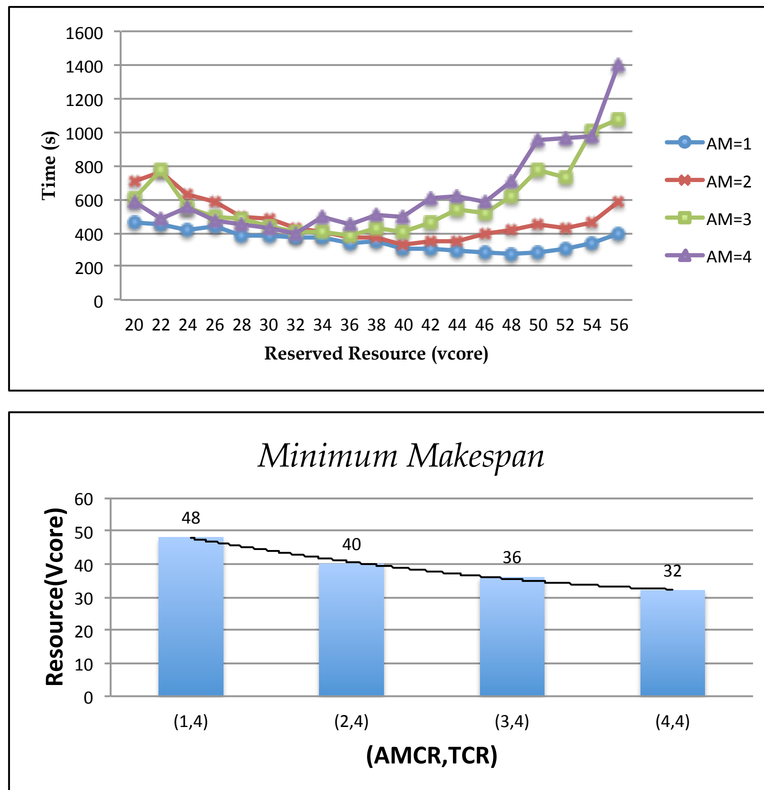

Fig. 2.2. *Makespans of 66* terasort *jobs under different resource requirement settings on AM and different static resource reservations.*

of resources reserved for non-AM tasks) and perform the admission control on incoming YARN jobs. There are three main components in our mechanism:

- RIC (Resource Information Collector): collect the resource and container's information from each node through heartbeat messages.
- RRP (Reserved Resource Predictor): decide how many resources should be reserved based on the information collected by from RIC.
- ARC (Application Resource Controller): leverage the predicted value of RRP to manipulate an application's running.

We will describe the design of these components in details in the following sections.

3.1. Resource Information Collector. The key function of this component is to record the number of currently running ApplicationMasters and normal processing tasks as well as the resources that have been occupied by these two kinds of containers on each worker node. A map data structure is maintained to record the information and will be updated through each heartbeat message between NodeManager and ResourceManager. The algorithm of RIC is shown in Alg. 1.

3.2. Reserved Resource Predictor. The purpose of the RRP component is to find out the amount of reserved resources that can not only preserve high throughput of the system but also prevent the deadlock problem. As shown in Section 2.3, the optimal value of reservation is varying under different workloads. Therefore, the RRP component should be frequently triggered to recalculate the desired reservation level in order to adapt to dynamic workload changes. The overhead of this component thus becomes a primary concern when the workload changes too frequently. In this paper, we propose a simple, heuristic approach to decide an appropriate reservation level according to the information of workload characteristics provided by the RIC component.

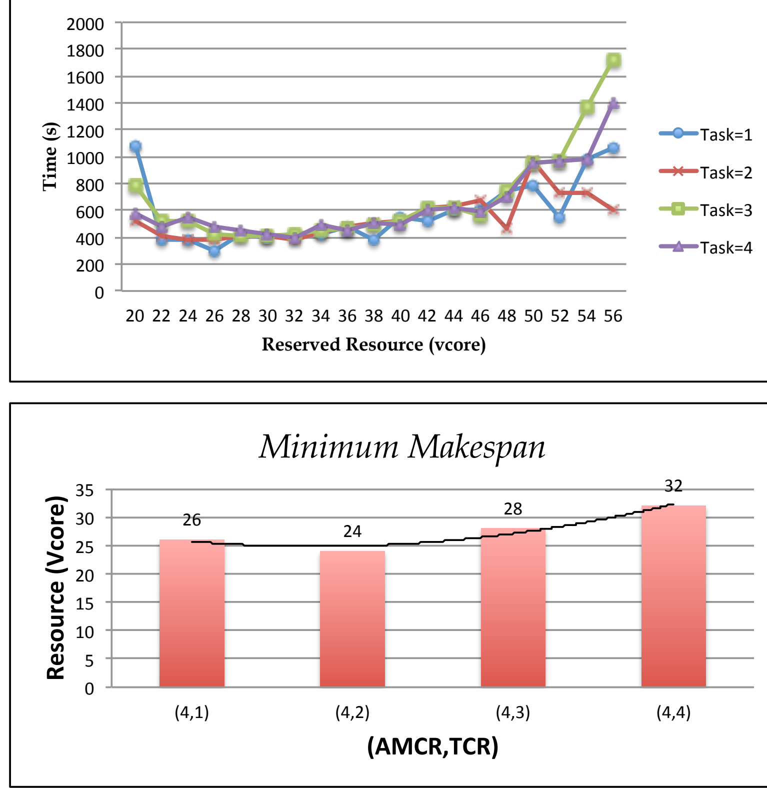

Fig. 2.3. *Makespans of 66* terasort *jobs under different resource requirement settings on map/reduce tasks and different static resource reservations.*

Algorithm 1: Resource Information Collector

Data:
Number of AM containers in node N_i: $N_{AMC_{N_i}}$;
Number of task containers in node N_i: $N_{TC_{N_i}}$;
Total resource of AMC in node N_i: Total_AMC_{N_i};
Total resource of TC in node N_i: Total_TC_{N_i};
HashMap that records each node's AMC information: Map_AMC;
HashMap that records each node's TC information: Map_TC;

1 **while** *each node N_i doing NodeUpdate* **do**
2 Record or update N_i: $N_{AMC_{N_i}}$ and Total_AMC_{N_i} into Map_AMC;
3 Record or update N_i: $N_{TC_{N_i}}$ and Total_TC_{N_i} into Map_TC;
4 Update $AvgAMC_R$ and $AvgTC_R$;

In Section 2.3, we also find that the optimal amount of reserved resources (R_R) seems to be proportional and inversely proportional to the resource requisition of non-AM task's (TC_R) and AM task's (AMC_R) containers, i.e., $R_R \propto \frac{TC_R}{AMC_R}$, under static workloads. To get a more clear understanding of this relationship, we conduct 16 sets of experiments. In each set of experiments, we run a batch of jobs with the same resource requirements, i.e., static workloads, repeatedly under different static resource reservations. The optimal resource reservation amounts in terms of makespan are shown in Figure 3.1. We can observe a linear relationship between the optimal reservation and the resource requirements of AM and tasks of jobs. Thus, we have the function:

$$R_R = C_R \cdot \frac{TC_R}{AMC_R + TC_R}, \tag{3.1}$$

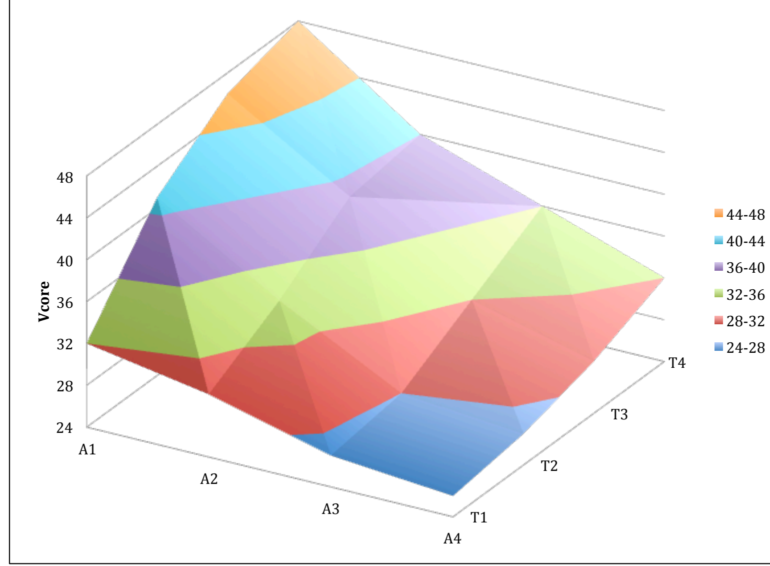

Fig. 3.1. *3D diagram of 16 sets of experiments under static workloads, where x-axis shows the resource requirements of an AM task (from 1 vcore to 4 vcores), y-axis shows the resource requirements of a regular task (from 1 vcore to 4 vcores), and z-axis gives the amount of reserved resources (i.e., number of vcores). Different colors in 3D surface indicate different makespans of the given set of jobs.*

Table 3.1
Optimal and predicted resource reservation, i.e., the number of reserved vcores. Predicted values are shown in the parenthesis.

	AMC=1 EXP(PDI)	AMC=2 EXP(PDI)	AMC=3 EXP(PDI)	AMC=4 EXP(PDI)
TC=1	32(32)	30(22)	27(16)	26(13)
TC=2	41(43)	34(32)	28(26)	27(22)
TC=3	46(48)	39(39)	32(32)	29(31)
TC=4	48(52)	40(43)	36(37)	32(32)

that fits such a linear relationship shown in the figure. The intuition behind this relationship is that we need to reserve enough resources for each running application in order to run at least one processing task to avoid severe resource contention.

The comparison between the optimal reservation results obtained through experiments and the prediction ones is shown in Table 3.1. As we can observe, the predicted values (i.e., the number of reserved vcores, shown in the parenthesis) are close to the optimal reservation in most cases. However, the amount of resources that need to be reserved is underestimated when the resource requirement of AM is much larger than the resource requirement of normal tasks. Under such a case, the benefit of accelerating each job's execution, i.e., reserving more resources for processing tasks, clearly surpasses the benefit of allowing higher concurrency between jobs, i.e., allowing AMs to occupy more resources. That is because jobs release a large amount of resources that are occupied by their AMs more quickly when they can finish their execution faster. To improve the accuracy of prediction under the extreme cases, we simply set a lower bound for resource reservation as 40% of the total cluster resources according to our observations. New prediction values are shown in Table 3.2.

Furthermore, we find that such a linear relationship still holds under the mixed workloads where jobs can have different resource requirements if we use the average resource requirements of running AMs and processing tasks to calculate the amount of reserved resources, i.e.,

$$R_R = C_R \cdot \frac{AvgTC_R}{AvgAMC_R + AvgTC_R}, \tag{3.2}$$

Table 3.2
Compare the predicted and actual (Experiment result) optimal vcore reservation number.

	AMC=1 EXP(PDI)	AMC=2 EXP(PDI)	AMC=3 EXP(PDI)	AMC=4 EXP(PDI)
TC=1	32(32)	30(26)	27(26)	26(26)
TC=2	41(43)	34(32)	28(26)	27(26)
TC=3	46(48)	39(39)	32(32)	29(31)
TC=4	48(52)	40(43)	36(37)	32(32)

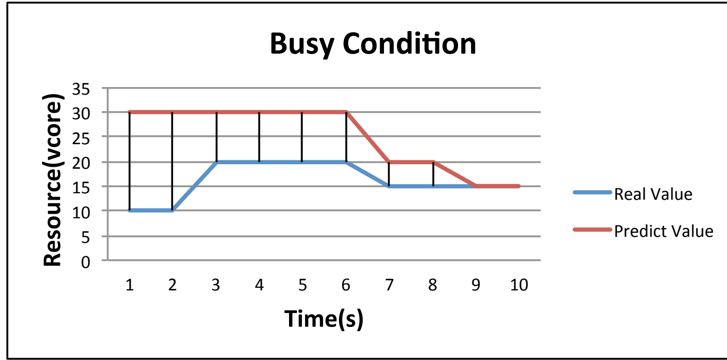

Fig. 3.2. *Compare the predicted reservation resources and the actual available resources.*

where $AvgAMC_R$ and $AvgTC_R$ indicate the average resource requirements of running AMs and processing tasks, respectively. Under the mixed workloads, the prediction value may change over time. For example, the reservation may increase when a new job's AM is submitted for running. However, such an increasing of reservation usually cannot be performed immediately since the resources that are already occupied by AMs cannot be released quickly.

For example, as shown in Figure 3.2, the desired resource reservation for processing tasks is 30 vcores at time period 1. However, the actual amount of remaining resources in the system, i.e., resources that are not occupied by AMs, is 10 vcores. The predicted reservation cannot be achieved until time period 10.

To mitigate the impact of this situation, we further scale up the amount of reserved resources to speed up the execution of processing tasks in the next time period, i.e.,

$$R_R = R_R \cdot \frac{R_R + TotalAMC_R}{C_R}, \tag{3.3}$$

where $TotalAMC_R$ indicates the total amount of resources occupied by AMs. The scale-up is triggered when the summation of resources that are currently occupied by AMs and are needed to be reserved for processing tasks exceeds the resource capacity of the cluster. The algorithm for predicting the reservation amount is shown in Alg. 2, where the scale-up approach is performed in lines 5-6. Figure 3.3 illustrates the summation of resources that are currently occupied by AMs and the desired resource reservation for tasks. The scale-up is triggered when this summation exceeds the capacity of the cluster.

3.3. Application Resource Controller (ARC). The ARC component manages two job queues, i.e., running queue and waiting queue. Each submitted job is inserted into one of these two queues according to the reservation policy. Figure 3.4 illustrates the mechanism of this component. As shown Figure 3.4, when RRP decides to reserve 21 vcores for processing tasks, 6 remaining vcores can then be assigned to AM tasks. The AMs of currently running jobs, i.e., job1 and job2, have already occupied all 6 vcores. Therefore, job3 needs to be inserted into the waiting queue and waits for available resources to start its execution. The algorithm of the ARC component is shown in Alg. 3. In lines 1-6, when a job is submitted to YARN, the ARC component decides if this job can enter the running queue. If the amount of resources for running processing tasks is more

Algorithm 2: Reserved Resource Predictor

Input: Cluster resource: C_R;
Output: Predicted reserved resource: R_R;
1 **while** *free resource available on node* **do**
2 Get the information from RIC;
3 **if** $N_{AMC} \neq 0$ *and* $N_{TC} \neq 0$ **then**
4 $R_R = C_R \cdot \frac{AvgTC_R}{AvgAMC_R + AvgTC_R}$;
5 **if** $R_R > C_R - TotalAMC_R$ **then**
6 $R_R = R_R \cdot \frac{R_R + TotalAMC_R}{C_R}$;
7 **if** $R_R < C_R \cdot 40\%$ **then**
8 $R_R = C_R \cdot 40\%$;
9 **if** $R_R > C_R - AvgAMC_R$ **then**
10 $R_R = C_R - AvgAMC_R$;
11 **else**
12 $R_R = C_R \cdot 40\%$;

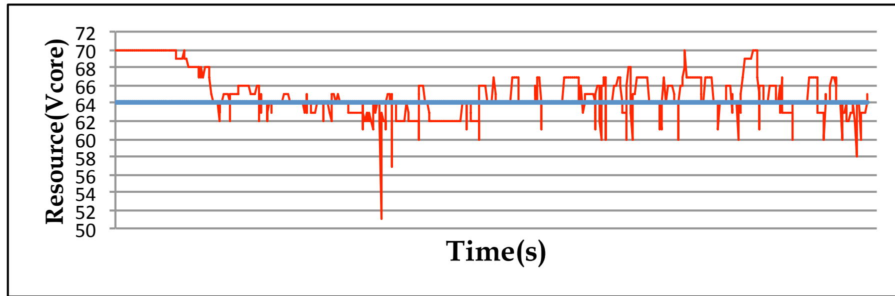

Fig. 3.3. *Illustrations of the total amount of resource (i.e., the resources occupied by AMs and the desired resource reservation for tasks). When the curve is above the overall capacity (i.e., 64 vcores), the reserved resources are then scaled up.*

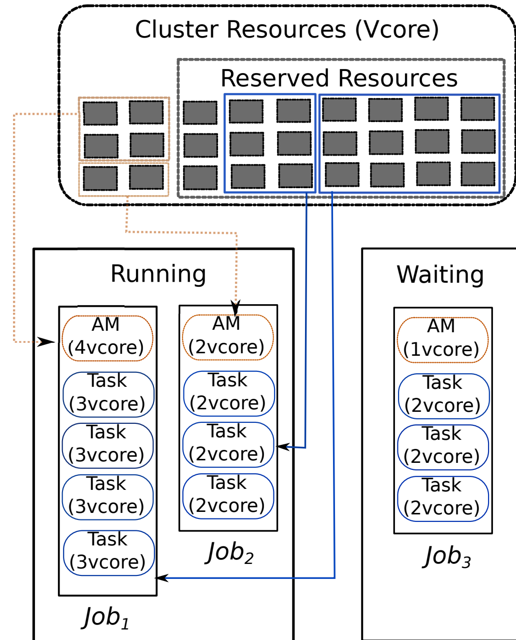

Fig. 3.4. *Application Resource Controller. The controller holds a specific amount of resources to run processing tasks. If there is no resource for processing, then the controller puts incoming jobs into the waiting queue.*

Algorithm 3: Application Resource Controller

```
   Input:
   Cluster resource: C_R;
   Reserved resource: R_R;
   Current occupied resource: TotalApp_R;
1  if job J submitted then
2  |  if C_R − TotalApp_R − AMC_{R_J} ≥ R_R then
3  |  |  Push J into APPLIST_RUN;
4  |  else
5  |  |  Push J into APPLIST_WAIT;
6  while each node doing NodeUpdate do
7  |  Get existing available resource in Node_i : AVA_R_i;
8  |  Get the necessary resource to run next waiting job's AM: WaitingJobAM_R;
9  |  AVA_R = C_R − TotalAPP_R − WaitingJobAM_R;
10 |  if AVA_R ≥ R_R and AVA_R_i ≥ WaitingJobAM_R then
11 |  |  Pop out job J from APPLIST_WAIT;
12 |  |  submit ApplicationMaster of J on Node_i;
13 |  |  Push J into APPLIST_RUN;
```

than the desired reservation, then the submitted job enters the running queue such that this job's AM can be launched immediately. Furthermore, once work nodes send heartbeat messages to the ResourceManager, the ARC component re-calculates the amount of available resources (AVA_R) and re-submits jobs that are currently waiting in the queue if available resources are enough to run additional AMs, see lines 7-12.

4. Evaluation.

4.1. Experiment Settings. We implement our new admission control mechanism in the Fair scheduler of Hadoop YARN, and build the YARN platform on a local cluster with one master node and 8 worker nodes. Each worker node is configured with 8 vcores and 12GB memory, such that the YARN cluster has the resource capacity of 64 vcores and 96GB memory.

In our experiments, we choose the following four classic MapReduce benchmarks for evaluation.

- **terasort:** a MapReduce implementation of quick sort.
- **wordcount:** a MapReduce program that counts the occurrence times of each word in input files.
- **wordmean:** a MapReduce program that records the average length of words in input files.
- **pi:** a MapReduce program that estimates π value using the Monte Carlo method.

The input files for **terasort** are generated through the `teragen` program, while the input files for **wordcount** and **wordmean** are generated through `randomtextwriter`.

For better understanding how well our new mechanism works, we calculate the relative performance score as follows:

$$Perf.\ Score = \left(1 - \frac{Makespan - Min_Makespan}{Min_Makespan}\right) \times 100\%, \tag{4.1}$$

where *Makespan* is the measured makespan (i.e., total completion length) of a batch of MapReduce jobs under our dynamic admission control mechanism, and *Min_Makespan* represents the makespan under the optimal static admission control setting.

4.2. Results Analysis. To better evaluate the robustness of the proposed mechanism, we conduct our experimental evaluation under two sets of workloads, i.e., homogeneous workloads with the same MapReduce jobs and heterogeneous workloads mixing with different MapReduce jobs.

4.2.1. Homogeneous Workloads. In this set of experiments, we submit a batch of 72 **terasort** jobs in each round. All the jobs in the same round have the same resource requirement setting, and each job processes a 100MB randomly generated input file. We further change the resource requirements for jobs in different rounds,

Table 4.1
Perf. Scores *under homogeneous workloads.*

	AMC=1	AMC=2	AMC=3	AMC=4
TC=1	90.4%	99.6%	89.3%	91.4%
TC=2	99.3%	95.6%	99.4%	91.7%
TC=3	97.6%	99.2%	96.9%	97.4%
TC=4	88.5%	90.5%	99.7%	98.5%

i.e., from 1 vcore to 4 vcores for both ApplicationMasters and processing tasks. Therefore, there are totally 16 rounds with different resource requirement combinations. The performance (e.g., makespans) under our admission control mechanism which dynamically sets the resource reservations (see red lines) as well as different static reservation configurations (see blue lines) is depicted in Figure 4.1. The corresponding *Perf. Score*s of our new mechanism are also shown in Table 4.1.

We first observe that different resource requirements need different amounts of reserved resources (e.g., numbers of vcores) for running tasks in order to achieve the best performance, i.e., the minimum makespan, see blue curves in Figure 4.1. However, it is inherently difficult to statically find such an optimal reservation level, especially if resource requirements are not fixed. While, by dynamically tuning the resource reservation level, our admission control mechanism always obtains the best performance compared to the results under the static resource reservations under almost all resource requirement configurations, see red lines in the figure. Table 4.1 further demonstrates that our new mechanism achieves high *Perf. Score*s, e.g., under more than half of the cases, *Perf. Score*s are greater than 95%.

4.2.2. Heterogeneous. Now, we turn to evaluate our new mechanism under a more challenging scenario, where we consider heterogeneous workloads which are mixed with different MapReduce jobs. Specifically, we choose more than two MapReduce benchmarks and submit a batch of jobs from these benchmarks which are configured with different resource requirements for running their ApplicationMasters and tasks. We totally conduct 6 sets of experiments with different combinations of MapReduce jobs. Table 4.2 shows the detailed configurations for each set of experiments.

The experimental results (e.g., the makespans and the *Perf. Score*s) under heterogeneous workloads are shown in Figure 4.2 and Table 4.3, respectively. Consistent with the case of homogeneous workloads, we can see that our proposed mechanism can still work well under various heterogeneous MapReduce workloads, which achieves the performance close to those under optimal static resource reservations. On the other hand, the settings for optimal static resource reservations are varying across different heterogeneous workloads. An inappropriate configuration could dramatically degrade the performance. Therefore, our mechanism which dynamically and automatically controls the admission of applications (i.e., reserving resources for running regular tasks) is important for achieving competitive performance of YARN.

5. Related Works. While the original Hadoop MapReduce framework has been extensively studied in recent years, Hadoop YARN is relatively new and has not been widely studied yet. A few preliminary works have been presented to improve the scheduling in Hadoop YARN systems. Dominant resource fairness [14] was proposed to improve the fairness between users when multiple types of resources are required by MapReduce jobs as in YARN framework. Some previous studies [13] [6] designed modified frameworks to handle iterative jobs. These works attempt to achieve performance improvements on share-based scheduling in Hadoop YARN. However, the deadlock problem caused by ApplicationMasters has not been considered yet.

Simple admission control policy that could help avoid deadlocks is supported in Fair [3] and Capacity [2] schedulers of YARN. For example, users can specify the maximum number of jobs for each queue under the Fair scheduler. By manually specifying a small number of jobs that are allowed in each queue concurrently, the ApplicationMasters of these jobs will not occupy all system resources. However, it is difficult for administrators to manually choose a good configuration. Moreover, static configurations cannot work well, especially under dynamic workloads.

Admission control in cloud computing has also been well studied in different aspects. Wu et al. [23] proposed admission control policies that aimed to maximize the SaaS provider's profits based on user SLA

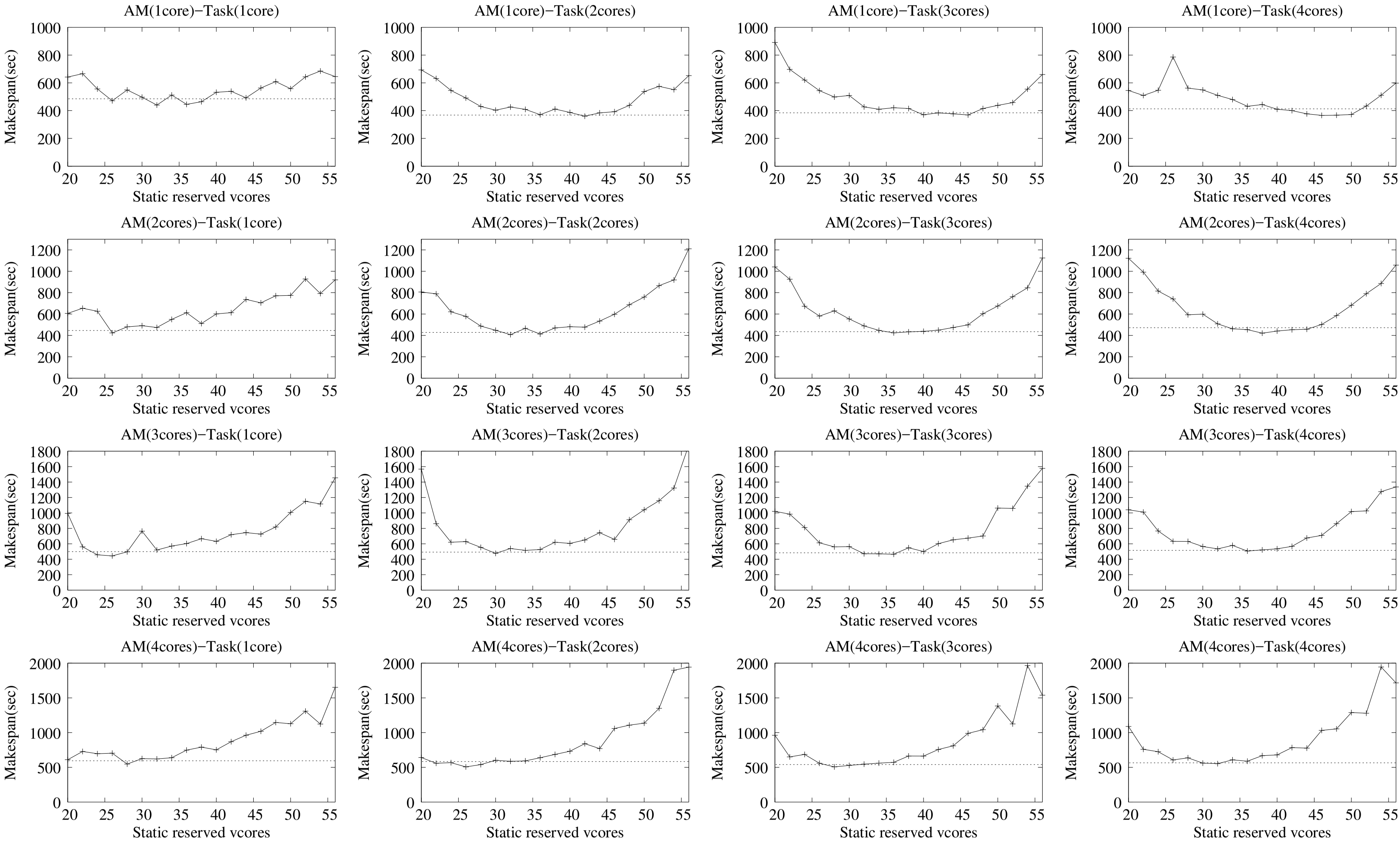

FIG. 4.1. *Makespans under homogeneous workloads, where the solid curves show the results under different static reservation configurations and the red dashed-lines present the results under our mechanism which dynamically sets the resource reservations.*

Table 4.2
Experimental setup for heterogeneous workloads.

	Benchmark	Job number	AMC_R	TC_R (Map)	TC_R (Reduce)	Input size
Exp1	Terasort	15	1	2	1	100M
	Wordmean	15	2	1	2	400M
	Wordcount	15	3	3	2	400M
	Pi	15	1	4	1	N/A
Exp2	Terasort	15	3	4	2	400M
	Wordmean	15	1	3	2	200M
	Wordcount	15	2	3	2	200M
	Pi	15	4	2	3	N/A
Exp3	Terasort	15	4	1	3	400M
	Wordmean	15	3	2	4	400M
	Wordcount	15	2	4	1	200M
	Pi	15	1	2	2	N/A
Exp4	Terasort	30	2	3	1	200M
	Wordcount	30	1	3	4	800M
Exp5	Terasort	15	3	1	1	100M
	Wordmean	15	4	1	2	100M
	Wordcount	15	4	1	1	100M
	Pi	15	2	1	1	N/A
Exp6	Terasort	20	1	1	2	200M
	Wordmean	20	2	3	3	400M
	Pi	20	3	2	4	N/A

Table 4.3
Perf. Scores *under heterogeneous workloads.*

	Exp1	Exp2	Exp3	Exp4	Exp5	Exp6
Perf. Score	100%	85.9%	94.2%	99.1%	90.6%	95.8%

and IaaS provider SLA. Machine learning based admission control for MapReduce jobs was proposed in [12] to meet job deadlines. Our previous work [24] also proposed a similar admission control mechanism but it does not consider heterogeneous workload use cases. In this work, our primary goal is to avoid the resource deadlocks and meanwhile to improve the makespan of MapReduce jobs [22]. Furthermore, our admission control mechanism only delays jobs instead of declining jobs.

Fine-grained resource management was also well studied for Hadoop systems. ThroughputScheduler [15] was proposed to improve the performance of heterogeneous Hadoop cluster. An explore stage was proposed to learn the resource requirement of tasks and the capabilities of nodes, and the best node was then selected to assign tasks in the scheduler. [18] leveraged job profiling information to dynamically adjust the number of slots on each node, as well as workload placement across nodes, to maximize the resource utilization of the Hadoop cluster. [21] is the first comprehensive study of intermediate data for YARN with Lustre and RDMA. [4] mathematically investigates the scheduling problem which is assigning inputs with various sizes to a set of reducers with capacity. POPI [8] is a lightweight algorithm targeting for efficient processing large outer joins under Hadoop. In order to improve the performance in large distributed environments, a new framework called CCF [9] is proposed to co-optimize application-level data movement and network-level data communications for distributed operators. Rayon [10] is proposed to reserve resources for production jobs and best-effort jobs such that the SLAs for production jobs can be guaranteed and meanwhile the execution time of best-effort jobs can be reduced. We note that our scheduler mainly focuses on how to allocate the reserved resources for best-effort jobs, which is complementary to Rayon in [10]. Zaharia et al. [25] proposed a delay scheduling policy

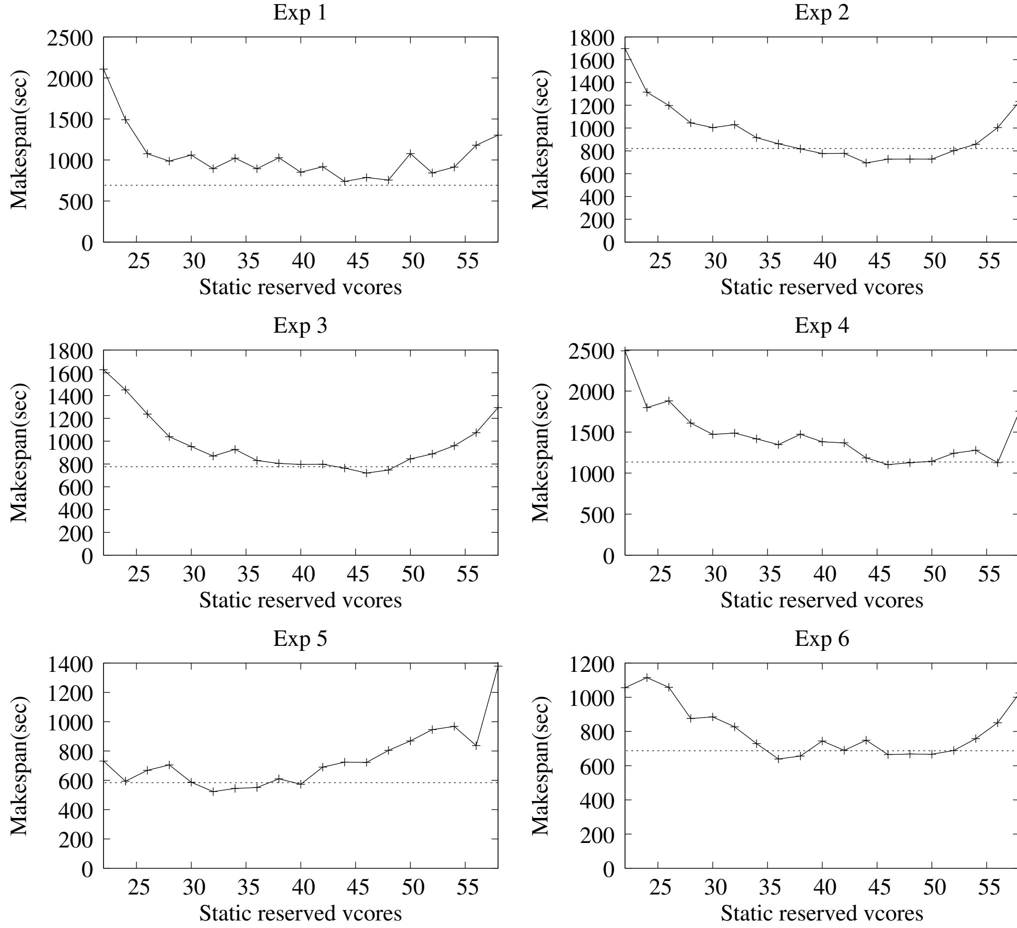

Fig. 4.2. *Makespans under heterogeneous workloads, where the blue curves show the results under different static reservation configurations and the red dashed-lines present the results under our mechanism which dynamically sets the resource reservations.*

to improve the performance of Fair scheduler by increasing the data locality of Hadoop, which is compatible with both Fair scheduler and our proposed scheduling policies. Quincy [16] formulated the scheduling problem in Hadoop as a minimum flow network problem, and decided the slots assignment that obeys the fairness and locality constraints by solving the minimum flow network problem. However, the complexity of this scheduler is high and it was designed for slot based scheduling in the first generation Hadoop. Verma et al. [20] introduced a heuristic method to minimize the makespan of a set of independent MapReduce jobs by applying the classic Johnson's algorithm. However, their evaluation is based on simulation only without real implementation in Hadoop.

6. Conclusions. In this paper, we presented a novel admission control mechanism that integrates with the existing Fair scheduler of Hadoop YARN. The main objective of our work is to automatically and dynamically reserve a specific amount of resources for processing tasks in YARN such that the deadlock problem caused by ApplicationMasters can be avoided. In addition, we aim to achieve better performance by controlling the concurrency level of jobs in the cluster. To meet this goal, the mechanism collects the resource usage information from each work node and leverages this information to predict the optimal amount of reserved resources for processing tasks. A waiting queue is further maintained to hold delayed jobs that will be resubmitted when there are available resources. We implemented our proposed mechanism in Hadoop YARN v2.2.0 and evaluated it with a suite of representative MapReduce benchmarks. The experimental results demonstrate that our mechanism can achieve the near optimal performance. The effectiveness and robustness of this new mechanism

are validated under both homogeneous and heterogeneous workloads. In the future, we will investigate the relationship between job concurrency and system throughput in a YARN cluster and further extend our work to other cloud computing platforms.

REFERENCES

[1] *Apache hadoop users.*

[2] *Hadoop mapreduce next generation - capacity scheduler.*

[3] *Hadoop mapreduce next generation - fair scheduler.*

[4] F. Afrati, S. Dolev, E. Korach, S. Sharma, and J. D. Ullman, *Assignment problems of different-sized inputs in mapreduce*, ACM Transactions on Knowledge Discovery from Data (TKDD), 11 (2016), p. 18.

[5] Apache, *Apache hadoop nextgen mapreduce (yarn).*

[6] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, *Haloop: efficient iterative data processing on large clusters*, Proceedings of the VLDB Endowment, 3 (2010), pp. 285–296.

[7] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, *The case for evaluating mapreduce performance using workload suites*, in Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on, IEEE, 2011, pp. 390–399.

[8] L. Cheng and S. Kotoulas, *Efficient large outer joins over mapreduce*, in European Conference on Parallel Processing, Springer, 2016, pp. 334–346.

[9] L. Cheng, Y. Wang, Y. Pei, and D. Epema, *A coflow-based co-optimization framework for high-performance data analytics*, in Parallel Processing (ICPP), 2017 46th International Conference on, IEEE, 2017, pp. 392–401.

[10] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, *Reservation-based scheduling: If you're late don't blame us!*, in Proceedings of the ACM Symposium on Cloud Computing, ACM, 2014, pp. 1–14.

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

[12] J. Dhok, N. Maheshwari, and V. Varma, *Learning based opportunistic admission control algorithm for mapreduce as a service*, in Proceedings of the 3rd India software engineering conference, ACM, 2010, pp. 153–160.

[13] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, *Twister: a runtime for iterative mapreduce*, in Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, ACM, 2010, pp. 810–818.

[14] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, *Dominant resource fairness: Fair allocation of multiple resource types.*, in NSDI, vol. 11, 2011, pp. 24–24.

[15] S. Gupta, C. Fritz, B. Price, R. Hoover, J. de Kleer, and C. Witteveen, *Throughputscheduler: Learning to schedule on heterogeneous hadoop clusters*, in Proceedings 10th ACM International Conference on Autonomic Computing (ICAC'13), ACM.

[16] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, *Quincy: fair scheduling for distributed computing clusters*, in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ACM, 2009, pp. 261–276.

[17] A. Murthy, *Apache hadoop yarn - concepts and applications.*

[18] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, and E. Ayguadé, *Resource-aware adaptive scheduling for mapreduce clusters*, in Middleware 2011, Springer, 2011, pp. 187–207.

[19] V. K. Vavilapalli, A. C. Murthy, C. Douglas, et al., *Apache hadoop yarn: Yet another resource negotiator*, in Proceedings of the 4th annual Symposium on Cloud Computing, ACM, 2013.

[20] A. Verma, L. Cherkasova, and R. H. Campbell, *Two sides of a coin: Optimizing the schedule of mapreduce jobs to minimize their makespan and improve cluster performance*, in Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on, IEEE, 2012, pp. 11–18.

[21] M. Wasi-ur Rahman, N. S. Islam, X. Lu, and D. K. D. Panda, *A comprehensive study of mapreduce over lustre for intermediate data placement and shuffle strategies on hpc clusters*, IEEE Transactions on Parallel and Distributed Systems, 28 (2017), pp. 633–646.

[22] M. Wojnowicz, D. Nguyen, L. Li, and X. Zhao, *Lazy stochastic principal component analysis*, in IEEE International Conference on Data Mining Workshop, 2017.

[23] L. Wu, S. Kumar Garg, and R. Buyya, *Sla-based admission control for a software-as-a-service provider in cloud computing environments*, Journal of Computer and System Sciences, 78 (2012), pp. 1280–1299.

[24] Y. Yao, J. Lin, J. Wang, N. Mi, and B. Sheng, *Admission control in yarn clusters based on dynamic resource reservation*, in Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on, IEEE, 2015, pp. 838–841.

[25] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, *Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling*, in Proceedings of the 5th European conference on Computer systems, ACM, 2010, pp. 265–278.

Edited by: Pradeep Reddy CH
Received: Jun 7, 2017
Accepted: Feb 26, 2018