Admission Control in YARN Clusters Based on Dynamic Resource Reservation

Yi Yao* yyao@ece.neu.edu
Jason Lin* jacks953107@ece.neu.edu
Jiayin Wang† jane@cs.umb.edu
Ningfang Mi* ningfang@ece.neu.edu
Bo Sheng† shengbo@cs.umb.edu

*Department of Electrical and Computer Engineering, Northeastern University, 360 Huntington Ave., Boston, MA 02115
†Department of Computer Science, University of Massachusetts Boston, 100 Morrissey Boulevard, Boston, MA 02125

***Abstract*—Hadoop YARN is an open project developed by the Apache Software Foundation to provide a resource management framework for large scale parallel data processing. However, there exists a resource waiting deadlock under 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. Therefore, we develop a new admission control mechanism which dynamically reserves resources for processing tasks in order to avoid resource waiting deadlocks and meanwhile obtain good performance. We implement and evaluate our new mechanism in Hadoop YARN v2.2.0. The experimental results show the effectiveness of this mechanism under MapReduce benchmarks.**

I. 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 [1] becomes one of the most popular programing paradigms in recent years. Its open source implementation Hadoop [2] has been widely adopted as the primary platform for parallel data processing [3]. Recently, the Hadoop MapReduce ecosystem is evolving into its next generation, called Hadoop YARN (Yet Another Resource Negotiator) [4], which adopts fine-grained resource management for job scheduling. 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.

This work was partially supported by the AFOSR grant FA9550-14-1-0160, Northeastern University FY14 Tier-1 grant, and the AWS in Education Research Grant.

On the other hand, YARN uses the classic 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. For each application in YARN, an ApplicationMaster is first generated to coordinate its processing tasks. Such an ApplicationMaster is indeed a special task in YARN, which has 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 these running jobs. A *resource waiting deadlock* thus happens such that each ApplicationMaster is waiting for other ApplicationMasters to release resources for running their processing tasks.

To solve this problem, one could kill/terminate running jobs and their AMs to break the deadlock. An alternative solution is to apply an admission control mechanism to control the number of jobs concurrently running in the system. Admission control in cloud computing has also been well studied in different aspects. Wu et al. [5] proposed admission control policies that aim to maximize the SaaS provider's profits based on users' and IaaS providers' SLAs (Service Level Agreements). Machine learning based admission control for MapReduce jobs was proposed in [6] to meet job deadlines. In this work, we consider to control the number of concurrently running jobs (or ApplicationMasters) by reserving resources to run processing tasks. By this way, the 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 system efficiency 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, running jobs might take a long time to receive resources for running their processing 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. We implement and evaluate our new admission control mechanism in a YARN platform (e.g., Hadoop YARN v2.2.0). The experimental results demonstrate that our new mechanism achieves the near optimal performance by leveraging the collected information of workloads and resource usages to decide the amount of resources that need to be reserved for regular tasks.

The remainder of the paper is organized as follows. In Section II, we present our new admission control mechanism. Evaluation of this mechanism is presented in Section III. We draw our conclusion in Section IV.

II. Methodology

The basic solution of preventing the deadlock situation is to reserve resources for running processing tasks. However, the amount of reserved resources has a great impact on system efficiency. A bad choice might degrade the system performance and the optimal amount often changes under different workloads. Therefore, we design a new admission control mechanism that can dynamically determine the optimal tuning point (i.e., the amount of resources reserved for processing tasks) and perform 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 RIC.
- ARC (Application Resource Controller): leverage the predicted value of RRP to manipulate an application's running.

A. Resource Information Collector

The key function of this component is to record the number of currently running ApplicationMasters and processing tasks as well as the amount of 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 NodeManagers and the ResourceManager.

B. 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 the optimal value of reservation is varying under different workloads, the RRP component should be frequently triggered 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.

To better understand the relationship between the optimal amount of reserved resources (R_R) and the resource requisition of processing task's (TC_R) and AM's (AMC_R) containers, we conduct 16 sets of experiments, in each of which a batch of jobs with the same resource requirements is launched repeatedly under different static resource reservations. The optimal amounts of reserved resources (which achieve the best makespans) are shown in Figure 1 as a function of different (TC_R, AMC_R) pairs. A linear relationship can be observed between the optimal reservation (see z-axis) and the resource requirements of AM (x-axis) and processing tasks (y-axis). We thus derive the following function:

$$R_R = C_R \cdot \frac{TC_R}{AMC_R + TC_R}, \tag{1}$$

where C_R gives the total resource capacity. The intuition behind Eq.(1) 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.

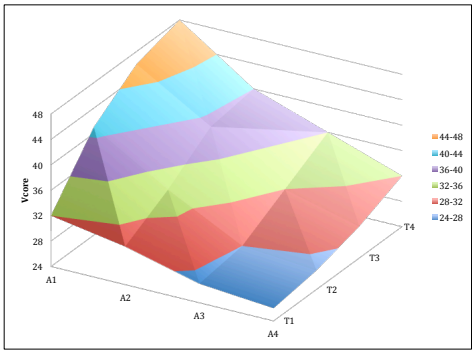

Fig. 1. The optimal amounts of reserved resources (i.e., number of vcores) under 16 sets of experiments, as a function of resource requirements of an AM task (from 1 vcore to 4 vcores, see x-axis) and a processing task (from 1 vcore to 4 vcores, see y-axis). Different colors in the surface further indicate different ranges of makespans of the given set of jobs.

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.,

$$R_R = C_R \cdot \frac{AvgTC_R}{AvgAMC_R + AvgTC_R}, \tag{2}$$

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 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.,

$$R_R = R_R \cdot \frac{R_R + TotalAMC_R}{C_R}, \tag{3}$$

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.

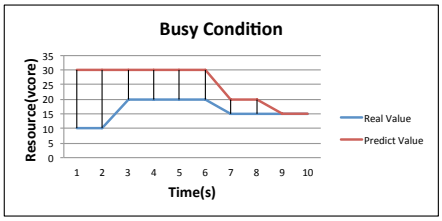

Fig. 2. Deficit between actual available and to-be-reserved resources.

C. 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. 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 than the desired reservation, then the submitted job enters the running queue such that this job's AM can be launched immediately. Furthermore, once a work node sends a heartbeat message to the ResourceManager, the ARC component re-calculates the amount of available resources and re-submits jobs that are currently waiting in the queue if available resources are enough to run additional AMs. For example, as shown Figure 3, 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.

III. Evaluation

A. Experiment Settings

We implement our new admission control mechanism in the Fair scheduler of Hadoop YARN v2.2.0, 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.

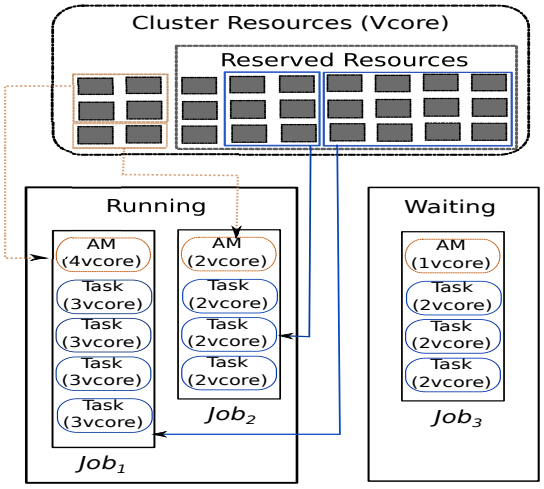

Fig. 3. Example of the ARC component.

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

$$Perf.\,Score = \left(1 - \frac{Makespan - Min\_Makespan}{Min\_Makespan}\right) \times 100\%, \tag{4}$$

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.

B. Results Analysis

In our experiments, we submit a batch of 72 **terasort** jobs in each round. The **terasort** benchmark is a MapReduce implementation of quick sort and its input files are generated through the *teragen* program. All jobs in the same round have the same resource requirements and each job processes a 100MB randomly generated input file. We further change the resource requirements for jobs in different rounds, 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.

Figure 4 depicts the makespans under our admission control mechanism which dynamically sets the resource reservations (see dashed lines) as well as different static reservation configurations (see solid lines). The corresponding *Perf. Score*s of our new mechanism are also shown in Table I.

TABLE I
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%

We first observe that different resource requirements need different amounts of reserved resources (e.g., numbers of

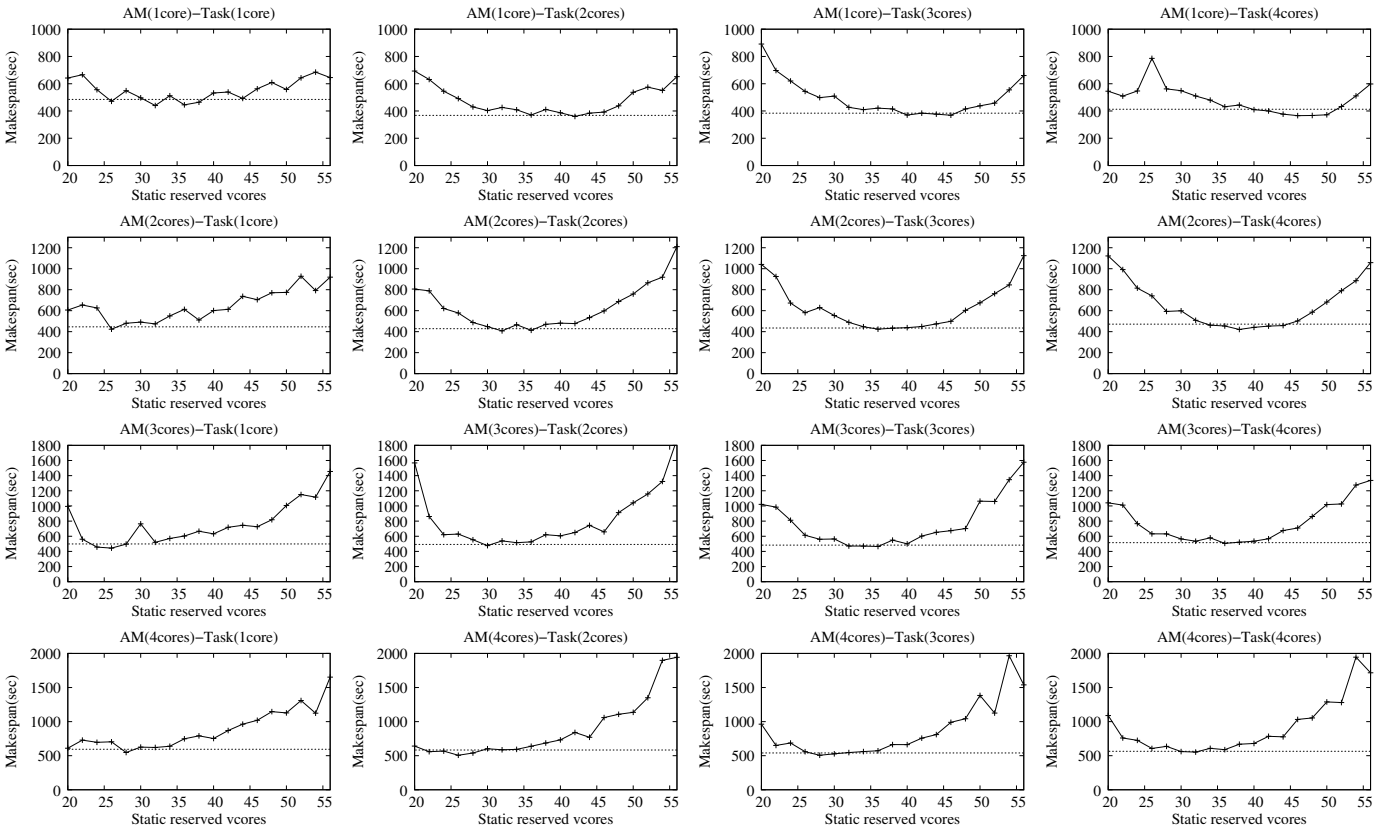

Fig. 4. Makespans of a batch of **terasort** jobs, where the solid lines show the results under different static reservation configurations and the dashed lines present the results under our mechanism which dynamically sets the resource reservations.

vcores) for running tasks in order to achieve the best performance, i.e., the minimum makespan, see solid lines in Figure 4. 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 (see dashed lines in the figure) under most resource requirement configurations. Table I 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%.

IV. 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 and evaluated our proposed mechanism in Hadoop YARN v2.2.0. The experimental results demonstrate that our mechanism can achieve the near optimal performance. 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] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] Apache, "Apache hadoop nextgen mapreduce (yarn)." [Online]. Available: http://hadoop.apache.org/docs/r2.2.0/hadoop-yarn/hadoop-yarn-site/YARN.html

[3] "Apache hadoop users." [Online]. Available: https://wiki.apache.org/hadoop/PoweredBy

[4] 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.

[5] 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*, vol. 78, no. 5, pp. 1280–1299, 2012.

[6] 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.

[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.