2017 IEEE 6th International Congress on Big Data

Resource Management for Parallel Processing Frameworks with Load Awareness at Worker Side

Son-Hai Ha
Orange Labs & EURECOM
sonhai.ha@orange.com

Patrick Brown
Orange Labs
patrick.brown@orange.com

Pietro Michiardi
EURECOM
pietro.michiardi@eurecom.fr

***Abstract*—Many resource management systems and large-scale data processing frameworks use a reservation-based model for managing resources and scheduling tasks. We observe from the reported traces of Facebook and Google that this model leads to resource being wasted because the tasks do not use effectively the allocated resources. We confirm the problem with a trace of our production cluster. We propose an algorithm to estimate the resource usage at worker nodes. This estimation is used as an input for the scheduler at the resource manager. We verify the stability of the new system in a simulator and develop a prototype of this approach for YARN. Our results in the simulator show that the new model can flexibly match the actual demand of the workload to the capacity of the cluster avoiding resources over-reserved by users. Comparing the worst scenario of our management model and the best scenario of the reservation model, we obtain almost the same performance and comparable system stability. In practice, our prototype for YARN completes jobs faster from 23% to 44%.**

***Index Terms*—big data; cloud computing; resource usage slack; scheduling; feedback**

I. Introduction

To analyze large datasets, it has become typical to use clusters of machines to execute jobs consisting of many tasks. The types of computations people run have diversified from MapReduce batch jobs to a very rich type of jobs, including interactive analytics, stream and graph processing, iterative machine learning, MPI-style computations, and complex pipelines. Recent resource management systems such as Mesos [1], Omega [2], Hadoop YARN [3], Kubernetes [4], and Corona [5] are popular frameworks to run these workloads.

We observe that these cluster management frameworks use reservation-based model for scheduling tasks. However, from our trace and from data of different studies [6], [7], [8], we see that this model is not an optimal choice for scheduling since it leads to resource being wasted. A reservation-based model uses a resource vector $< resourceA, resourceB, ... >$ to describe the resource demand of task. The resource management system reserves a resource segment on its managed nodes with fundamental requirement: resource demand cannot exceed the capacity that the node can provide at the time. In practice, there are also other parameters for a scheduler to place a task on a node such as data-locality, fairness,... In this paper we focus on the resource demand vector and the node capacity when considering a location for placing task. The reservation model assumes that users know in advance the resource that their jobs need, but in practice jobs at different phases have different resource demands; different tasks in the same phase may also have different resource demands; and even the same tasks at different moments require different type of resources. A static resource configuration for a job or for a task is not an optimal approach.

Our motivation is to enhance the existing resource management infrastructures by scheduling tasks with the reference to the current utilization at each worker node in the cluster. In the reservation model, a popular approach is to learn the workload resource in advance to determine the optimized reservation of each jobs [7], [9]. Another is to over-commit server resources to cope with the variability of workload resource usage [10]. While the former needs to accurately track resource usage of identical or similar workloads in advance, the latter consists in allocating applications more server resources than the servers can actually provide. This reduces system stability during peak time and requires detailed observation on the workloads.

In this paper, we introduce a method to estimate the resource usage on worker nodes and determine the available resource that we can use for placing tasks for the parallel processing frameworks without sacrificing the system stability and without the necessity to learn exactly the resource consumption of each job in the workload in advance.

With the new approach, we prove that there is a better way to manage the cluster resources for parallel processing frameworks. When we apply this approach to the real system - in our case Hadoop YARN - we show that cluster performance improves. Using Hive Testbench [11], which is a standard large scale system benchmarking for Hive, we observe jobs complete 23% - 44% faster in average.

In addition, we provide a tracing tool [1] to track the cluster utilization on YARN, which can study various workloads and learn job resource demand. We also introduce a simulator to simulate the system and replay trace with information on real usage of jobs, enabling new ideas for scheduling or to study the response of different monitoring parameters when running production workloads. We also present a prototype implementation [2] of the proposed method for YARN which considers the current resource utilization at node for scheduling, to improve the job performance.

In section III, we present our problem statement which is verified by our tracing tool on a production cluster at Orange (France Telecom). We propose a new algorithm to improve resource usage efficiency in section IV and then develop a simulator to replay the trace and to verify the stability of our approach in section V. In section VI, we show how our prototype performs on a YARN cluster. Even though we only

1. Yarn Tracking is published at https://bitbucket.org/hasonhai/yarntracking
2. Yarn Prototype is published at https://github.com/hasonhai/hadoop

apply these ideas to YARN - a Hadoop resource management system, we believe that our approach can be applied to other distributed platforms as well.

II. Related works

Cluster are generally under-utilized because of either *resource usage slack* or *allocation slack*. The resource usage slack represents the resource that is allocated to jobs but not used by jobs. In some contexts, this is also called over-allocation. The allocation slack is the available resource that is not allocated because of some system constraints or because of a lightweight workload.

In order to increase resource utilization, jobs and tasks are profiled to study resource demands or to estimate jobs and tasks runtime. So that identical jobs can be scheduled more efficiently [9], [12], [13], [14] or tasks are placed on node more optimized [7]. The authors in [7] state that the current scheduling algorithms which are only based on one or two types of resource (memory and CPU) are not optimal. They introduces Tetris which schedules jobs based on the 4 kinds of task demand: disk, network, CPU, and memory. Tasks are profiled on resource demands and packed on worker node to optimize for job makespan, solve the resource fragmentation and over-allocation problem. For makespan, they reduce 29% comparing to the Capacity Scheduler and 27.5% comparing to the Dominant Resource FairShare scheduler[6]. In [14] Curino et al. claim that the lack of predictability and time-awareness is the key limitation for modern frameworks. With *Rayon*, they attempt to increase the utilization of the cluster by providing a declarative resource demand language (RDL) so that job submitters can declare all of their demands at submission time. By using a global view on the resource needs of all jobs, the cluster management system can reserve resources ahead and ensure predictable resource-allocation for production jobs and also minimal latency for best-effort jobs.

Another approach is resource over-subscription. While static over-subscription enables conservative and predictable over-subscription, dynamic over-subscriptions shown to make over-subscription more robust and precise for utilizing the actual slack resources. Dynamic over-subscription has been deployed in Borg [10] and Mesos [15]. Borg [10] gives the tasks a fixed amount of resources in the beginning and slowly reduces the task reservations to the actual usage with a safety margin. The reduced amount is advertised as free resource and allowed for other jobs to use. They claim that the reclaimed resources can be used for 20% more of workload while the system stability is maintained. Mesos takes similar approach but only reclaims resources from best-effort workloads. While Borg and Mesos manage the prediction for reservation per task, we estimate the resource usage per node.

III. Problem statement

In one study of Google [8], we see that even if cluster resource are fully allocated to jobs or applications, the real resource utilization is around 40 to 50% of the allocated one. Another study on Facebook MapReduce workload trace [6] shows that resource is allocated to job but jobs cannot use all of them. We claim that the problem happens because of the reservation model.

A. Tracing tool

To analyse the efficiency of the reservation/allocation model in the context of YARN, we develop a light-weight application that keeps track of the cluster utilization at our production cluster including CPU, memory, storage I/O, and network I/O. The profiling service is composed of an observer and several trackers. A tracker is installed at each worker node. It polls the NodeManager to know if there are containers assigned to the NodeManager and tracks the resources used by such containers during their execution. A container in YARN is a Java process that runs the task. Resource used by these Java processes is limited by the user's reservation. The profiling service logs the CPU utilization, memory usage, network I/O, and disk I/O during the container lifetime. After the containers complete their tasks, these logs are sent to a collected node which also runs the observer. The observer polls the YARN Timeline Server, which is a logging service of YARN, to track the creation and completion of new jobs. When there is new job finished on the cluster, the observer collects the job information and the container metadata, acquires the logs of the containers running the tasks, and merges them with the resource utilization which is produced by the tracker. The merged information is stored in JSON format for analyzing.

B. Cluster Resource Usage

The cluster has 3 master nodes which are installed with master services and 6 worker nodes to run the tasks. Nodes in this cluster have 24 cores and 64GBs of memory. YARN is configured to used 144 cores and 324GBs of memory for container allocation and execution. Job types are varied from interactive jobs to batch jobs but the majority of them are short batch jobs. Users submit jobs through hive, pig, oozie, and Spark. We perform our profiling service during one week and log around 8000 jobs in which there are 110000 containers.

We define the allocation efficiency as the ratio of total resource used to the resource allocated for container during its lifetime.

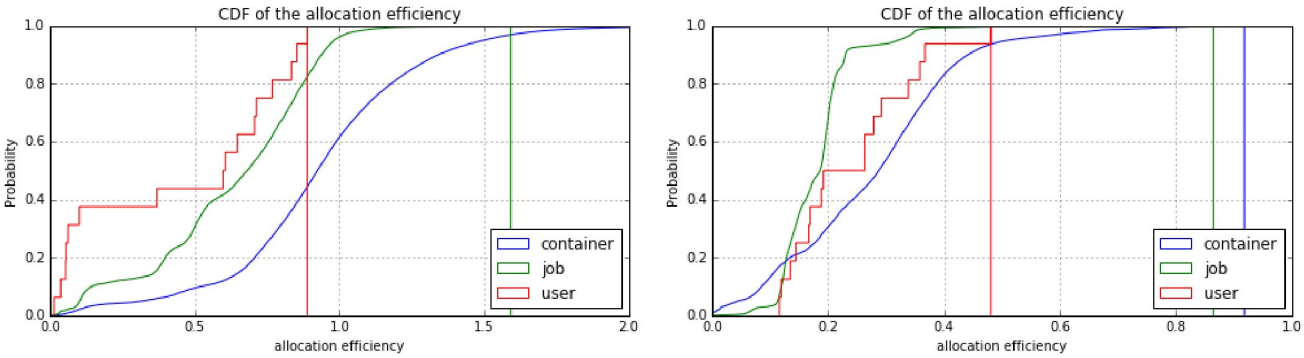
$$E_c = \frac{\sum_{i=0}^{n} u_i}{\sum_{i=0}^{n} r_i}$$

where

- E_c is the allocation efficiency of the container c
- n is the duration of the container
- u_i is the resource used at time i
- r_i is the resource reserved at time i

Figure 1 shows the cumulative distribution of the allocation efficiency. We present the container, job, and user allocation efficiency. Container allocation efficiency is the ratio of the total resource used and the total resource allocated to a container. Job is a group of containers. Job allocation efficiency is the ratio between the total resource used and the total resource allocated by all containers that belong to the jobs. Similarly, user executes multiple jobs on a cluster. We define the user allocation efficiency as the ratio of the total resource used and the total resource allocated by all containers that belong to all the user's jobs.

Concerning CPU usage, 60% of the containers use less than their allocation. However, they can use the resource upto 2 times their allocation. CPU over-utilization in some jobs can be explained by two reasons. *i)* The default setting of YARN



(a) CPU *(b) Memory*

Figure 1: Cumulative distribution function of container's allocation efficiency

does not isolate the CPU used between containers. Since each application's thread can use entirely one CPU core, containers with many threads can use more CPU than their allocation. *ii)* The majority of our observed jobs are short duration batch jobs which are CPU hungry [16]. We believe when the cluster has more interactive jobs which requires user input with long idle time, or when the cluster was configured with *cgroups* for CPU isolation, resource usage slack will appear clearer.

Concerning memory usage, each container's memory is isolated inside each JVM. YARN is able to control the memory that a container uses. 90% of the containers use less than 50% of the memory that is reserved. 90% of the applications use less than 24% of the memory that they reserve. 90% of the users use less than 35% of the memory that they reserves. This may be a result of bad configuration. For example, we see that all the containers that run the application masters only occupy at most 35% of the memory. Our first question is if there is a way to avoid this mis-configuration? Mappers and Reducers do not have the same load, so there is no common size for memory that they need. Our second question is that if we can find a dynamic value to represent the reservation of mappers and reducers? We present our approach to answer these questions in section IV.

Our results indicate that we need a better way to allocate the resource on the cluster and preferably an automatic way which does not require human intervention and tunning. If the scheduler is aware of the actual load on each worker node, it can decide to push more container to that node without hurting the performance of other containers running on the same node. Increasing concurrency level of job can reduce job makespan. This is the seed for our idea in the next section.

IV. Estimation-based scheduling

Nodes in a cluster generally do not have the same load simultaneously. Reservation-based scheduling treating all nodes as the same is clearly not optimal. Scheduling based on real utilization at worker nodes (WNs) can be an alternative way to effectively utilize the resources. It allows the system to dynamically adapt to the current state of each WN in the whole cluster. However, as we have learned so far, tasks and resources allocation cannot only be based on real resource utilization. We take two examples, one with memory and one with CPU. Task needs a few seconds to load data to memory, or longer if data is retrieved remotely via the network. Scheduling based on the instant value of memory utilization may lead us to assign too many tasks because some of the tasks are at the start-up time. Similarly, the CPU utilization varies erratically during job execution time. If we consider only the instant load of the CPU which can be a short decline at peak time, we may assign too many tasks to a heavy loaded node which will introduce the costly context switching.

To get the advantages of both paradigms (reservation and real usage), we propose a mechanism which takes into account both the reservation and the real utilization at node. The main idea is doing an estimation on the resource that the task can use by *i)* taking the reference on the actual usage at WNs by averaging (eq. 1), this averaging method also prevents the estimation to vary too fast and with high amplitude, *ii)* preparing for the low usage of the resource at the start-up time of the tasks (eq. 2) *iii)* correcting the estimation when it goes wrong because of sudden utilization spikes (eq. 3).

Assume that we have a cluster with one resource manager (RM) and several worker nodes for executing the tasks. Denote by $U_{R(n)}$, the utilization of resource R measured at time n and sent by a worker node to the RM. We assume the time is discretized. The nodes of the cluster periodically send to the RM the current level of utilization of the resources. The resource manager updates its estimate, $E_{U_R(n)}$, of the resource R with the measurement $U_{R(n)}$ sent by the node to which R is attached, and its previous estimate, $E_{U_R(n-1)}$, with an averaging formula:

$$E_{U_R(n)} = (1-\alpha)E_{U_R(n-1)} + \alpha U_{R(n)} \quad (1)$$

where α is a damping factor. As an important advantage compared to today's RM solutions, this method allows adjusting the estimate of the level of utilization of resource R with a correcting factor composed of the measured utilization level at the node. This allows for the RM to allocate supplementary jobs if the utilization level is lower than the reservation, or on the contrary to avoid allocating more jobs if the utilization level is too high.

Denote by RR_J the resource requirement for a resource of the same type as resource R declared by a job J. When the RM decides to allocate RR_J of resource R to job J at time

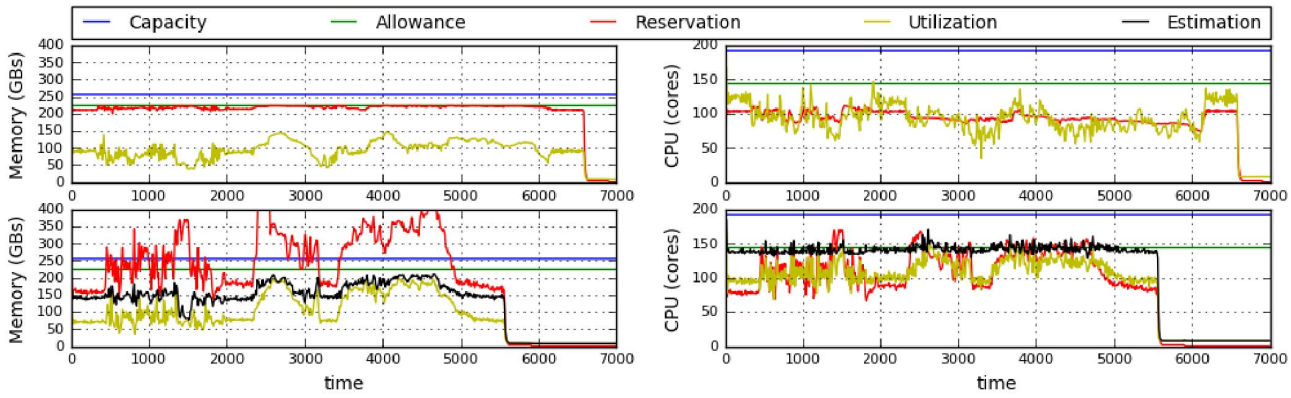


Figure 2: Memory and CPU utilization of the cluster when reservation model is used (top figures) and when the estimation model is used (bottom figures).

n, the RM instantly updates the current value of its estimate of that resource usage, $E_{U_R(n)}$, in the following way:

$$E_{U_R(n)} \leftarrow E_{U_R(n)} + RR_J \quad (2)$$

This allows for the RM to allocate at least as many jobs using R as it would basing only on reserved resources. The RM can take instant decisions without having to wait to "learn" how much resource the job will effectively use. So equations 1 and 2 allow simultaneously to "learn" how much resources the jobs are using by taking into account the resource usage measurements (cf. equation 1) and to take instantly into account the resource requested reservation (cf. equation 2). A user submitting a job may now not worry about a precise estimation of the resources that her job will consume. A resource requirement for her job are only used as a gross estimate to know if there is enough space for it to run. Even if it is overestimated or mis-configured, the update performed by equation 1 will allow the over-estimated resources to not be blocked indefinitely until the job stops. If we apply these to the cases where jobs are idle in an intermittent manner such as interactive jobs waiting for user inputs, or real time services at low load moment, the blocked resources by user's reservation can be reclaimed for other jobs during these idle periods.

The current resource usage at a node may increase suddenly due to external load by other applications, by a sudden input from user of interactive jobs, or by the arriving at peak time of data processed by real time services. We correct the resource estimation with equation 3 to prevent later scheduling decision overloading the node.

$$E_{U_R(n)} = U_{R(n)} \quad \text{if } E_{U_R(n)} < U_{R(n)} \quad (3)$$

The damping factor can control how fast the estimation adapts to the real utilization at the node. When the damping factor is too small, it takes long time for the estimation to converge to the real utilization and leave resource wasted. Let n_J be the execution time of task J, we apply equation 4 when task J finishes to remove the residual estimation for task J from the estimation for the node.

$$E_{U_R(n)} \leftarrow E_{U_R(n)} - (1-\alpha)^{n_J} RR_J \quad (4)$$

V. Evaluation on simulator

To study the performance of the new scheduling model, we develop a simulator to replay the trace that we collected with our profiling tool. The simulator simulates a cluster with many nodes. Each node is defined with a number of cores and an amount of memory. When assigning a task to a node, the simulator priories the node that has more resources. Input tasks are parsed from the trace that we analyzed in section III. They are queued to run in FIFO order based on the creation time.

We classified resources into two different types: *compressible* and *non-compressible* resources. Compressible resources are rate-based like processing power, network I/O bandwidth, and disk I/O bandwidth. If these resources are overloaded, the task will not crash but slow down. In our simulator, when all the tasks demand the CPU resource more than the capacity of the node, some of the tasks will be served only a part of their demands, the remains will be served in the next second. This is to simulate the context switching penalty.

On the other hand, non-compressible ones are like memory or disk space which will cause the task to crash if it is over-used. If the crashed task is an important task like the *application master* in YARN which manages the whole job or a critical task in the middle of a complex data flow, the penalty of crashing is worse. In our simulator, when memory demanded reaches the node capacity, all the demanding task will break and send back to the scheduler for later execution. Our simulator treats all tasks equally so there is no dependence between tasks. Therefore, if one task fails, only that task will be re-scheduled.

A. Avoiding the reservation bottleneck

The goal of our first experiment is to show if the new model can avoid resource wastage because of user over-demand and avoid the reservation bottleneck. In this experiment, the simulated cluster composes of 8 worker nodes. Each node is set with 32 GBs of memory and 24 cores. In each node, 28 GBs of memory and 18 cores are configured for the parallel processing framework to use. The workload are parsed from a one-day trace on our production cluster which contains about 18000

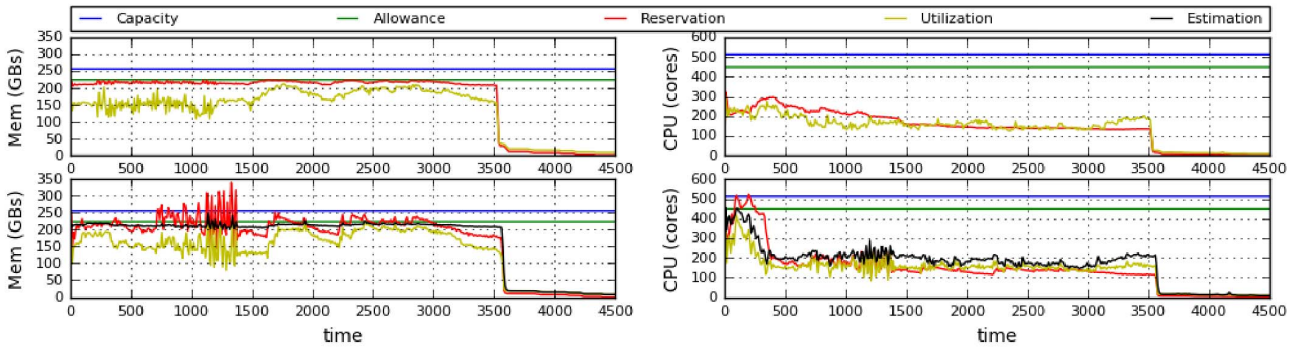


Figure 3: Resource utilization for the reservation model (top strip) and the estimation model (bottom strip) shows almost identical performance. The reservation model in its ideal scenario where it knows the maximum resource usage of task. The reservation model in its worst scenario to avoid tasks crash where it schedule based on the past estimation in a memory-constraint cluster with tied reservation from user

tasks. We compare the resource utilization of the reservation model and our estimation model.

Figure 2 shows the utilization for the whole cluster which is a summation of all node at the same point in time. In the legends, the *capacity* is the total capacity of the cluster. The *allowance* is the desired maximum utilization that we would like to have. In the reservation model, the *allowance* is the configured node capacity that the administrator configures at each node. The *estimation* is our current estimation on the resource usage of the cluster. The gap between the *allowance* and the *estimation* is the available resources that we can allocate tasks. The gap between the *allowance* and the *capacity* can be used as a safety margin for our estimation algorithm since sometimes the *estimation* and the real utilization can exceed the *allowance* and cause tasks to break. The reservation is the total resource reserved by all executing tasks at the current time. The reservation model allocates tasks based on the reservation while the estimation model allocates them based on the current *estimation*.

We show 16% faster completion times of all tasks in the trace when our estimation algorithm is applied. The top strip shows that the reservation model gets bottleneck because users over-demand for memory resource. With the traditional reservation scheduling, even if we always have tasks to execute, the node will only use around 50% of the available memory. This leads to longer time to finish the workload. With the estimation algorithm (the bottom strip in the two figures), the system learns the resource usage by the workload and automatically shift the bottleneck from memory resource to CPU resource which the best response that the system should have. We observe that there is zero task failed because of the memory consumption in the test.

B. Avoiding memory saturation

Because of the resource usage slack, our scheduling model likely assigns more concurrent tasks to the node than the reservation model. If disk swapping is disabled, tasks will crash when the memory of the system saturates. We would like to confirm the response of our scheduling model when memory is the main constraint on the cluster.

To simulate the case where memory is the bottleneck, we create one simulated cluster with eight slave nodes. Each node has 32 GBs of memory and 64 cores. Nodes were configured to allow the utilization of memory to go up to 28 GBs, and of CPU to go up to 58 cores when executing tasks. Note that one core and one GB of memory are assigned by default to the OS and the other applications.

We replay also the trace of one day of our production cluster. The task durations fall in between several seconds and 1000 seconds, but the majority of them have duration from 10 to 100 seconds. In this experiment, we assume that users know in advance the memory that their task needs. So the resource reservation for each task is the exact maximum memory and CPU used. This setting is called *Extreme Fit Reservation*. To implement this setting, we replace the old reservation value of the memory in the trace with the maximum memory used by the task. This setting is the ideal case for the reservation model because: *i)* it minimizes the resource usage slack so all the tasks will complete fastest. *ii)* it guaranties that, at any moment, the total consumption of memory of all tasks will not be higher than the allowance. Therefore there is no task failed because of insufficient memory and every task has its resource ready for use. However, applying this workload to our estimation algorithm will place it in the worst situation to avoid memory saturation and container crashes.

Figure 3 is our simulation results where the top strip is for the reservation model, and the bottom strip is for the estimation model. We select damping factor of 0.125 (see the next section for a discussion on the sensitivity of the damping factor). Even though this is an ideal scenario for the reservation model and our estimation model in its most disadvantaged scenario, the performance of the two models is almost similar. Through several test runs, we observe a failure rate of 0,37% (or 64/17385 failed tasks) which does not impact the job performance.

C. Damping Factor Sensitivity

The only parameter in our algorithm is the damping factor in equation 1. The damping factor controls how fast the estimation adapts to the real utilization at worker nodes. The damping factor has an impact on the stability of the system.

When the damping factor is zero, our model acts like a reservation model which is the most stable but also more resource wasted. When it is one, the system depends only on the current load at the worker nodes, which wastes less resource but is also less stable.

We define the stability of the system as the failure count of the executed tasks (assuming disk swapping is isolated). Large clusters focus on resource isolation to keep system stability, while small clusters often squeeze for every bit of performance from the hardware. The reason for reservation model to be popular in many resource management systems is that it is simple and it provides resource isolation protection for tasks. That means job j reserves an amount of resource r for task t, that resource r will be kept until task t finishes. The best scenario happens when users know in advance the demand of theirs tasks. This is why there are two approaches to improve resource utilization for reservation model: *i)* learn the exact demand of tasks so that in later runs the system can prepare resource better or the user can reserve better. The benefit of this method is that the stability of the system is preserved because the total consumption of resource never exceeds the capacity of the system. However it requires to learn the resource usage of identical tasks. Even though it is proved that this method can improve the performance[7], applying the resource usage learnt from identical tasks to different data may result in different resource usage, which may give unexpected performance. *ii)* over-subscribe the cluster. The benefit of this method is that it just needs the administrators to observe the global load of cluster and configure the resource and the workload to best fit their cluster. This requires human intervention some stability studies for the workload (like borg [10] has done to choose safety margin when doing resource reclaiming for over-subscribing).

We repeat the experiment in V-B with different settings for the damping factor to study the stability of the system. We compare the impact of the damping factor for the test case with the Resource Requested by Users (RRU) and for the test case with Extreme Fit Reservation (EFR). In figure 4 we presents the fraction of crashed tasks to the total submitted tasks and the completion times of the workloads for different damping factors. For EFR, failure count starts to raise when damping factor is 0.1 while it does so for RRU when the damping factor is 0.175. We observe that completion times for the workloads on the system decrease until the damping factor reaches 0.3 (and stay constant after) even though we have more crashed tasks and we have to re-schedule them. Note that in our simulator as with Map and Reduce tasks, tasks are not depended on each other so a failed task will not cause other tasks to fail. However, in practice, tasks may have dependency and the penalty of task crashing can be critical. It is preferable to select the damping factor with low value. We care for system stability so we select the damping factor of 0.125 for all of our tests in the simulator and in the real implementation.

VI. Evaluation on real system

We implement our solution for YARN as a prototype to study the performance that we can get in practice.

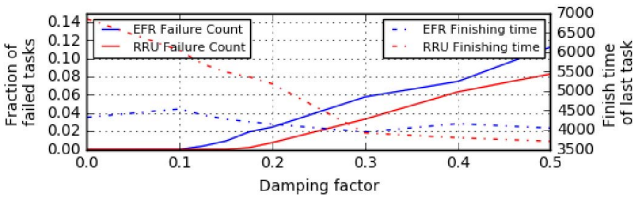


Figure 4: Fraction of crashed tasks to the total submitted tasks because of memory over-demanded

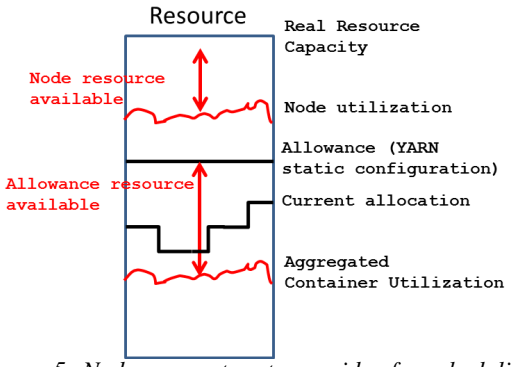


Figure 5: Node parameters to consider for scheduling

A. New YARN implementation

In YARN, the term *container* often means a task or a group of tasks executed inside a Java Virtual Machine (JVM). When discussing about the container, we means the task or the process that run inside the JVM. In this paper, we only consider the case of one task per JVM. Figure 5 indicates the parameters that we consider when allocating resource to node. While the original YARN only takes into account the *allowance* (the static resource capacity of node configured by administrators) and the current *allocation* (total reservation by all users), we consider also the real capability of the node, the current node utilization and the aggregated utilization of all containers. The node utilization is the current utilization of all applications and OS on the whole node. The aggregated container utilization is the total resource utilized by only the YARN containers.

Described in figure 6 is our prototype for YARN. At node registering process, worker nodes report to the Resource Manager their real resource capacity together with the configured capacity set by administrators which we call the "*allowance*". A resource monitoring service is added to track the resource usage of node. This service can run the estimation described in section IV at a worker node to shift the overhead from the Resource Manager RM to the NodeManager (NM). Running the estimation at worker side will introduce a small amount of allocating latency at maximum of two heartbeats due to RM waiting for the resource update from NM. Running the estimation at master side increases the heartbeat processing time which is critical for big clusters with thousands of nodes. For small clusters, we prefer to calculate at the master side, while for big clusters, we prefer doing it at worker side. The dash line is the feedback flow when we run the estimation at

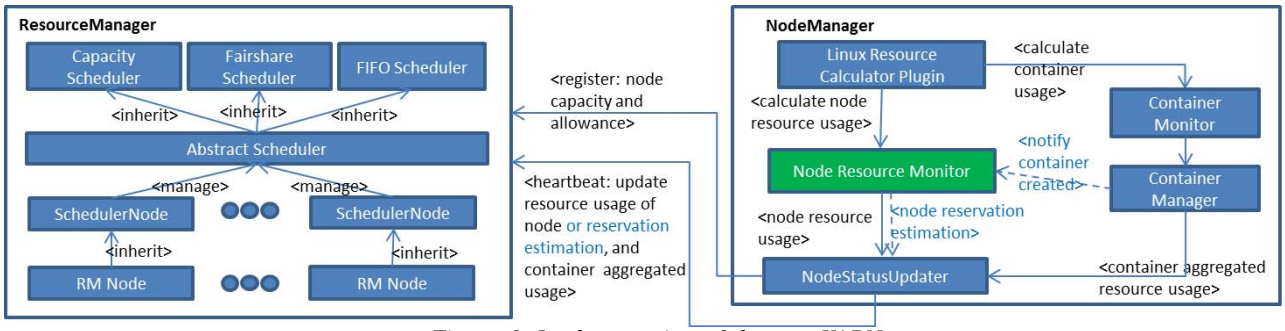


Figure 6: Implementation of the new YARN

NM.

Finding the available memory of the system is a delicate task. Memory in Linux is managed by pages and each application has a virtual memory space which has each page mapped to another page in the physical memory space. Because of the optimization of the OS, the unused memory can be used for page caching to increase application performance and the system will dump the inactive page cache if needed to satisfy allocation requests. Therefore the free memory reported in */proc/meminfo* file is alway low. It is not a good measure for the *available memory* because not all cache is needed and useful for improving application performance. Recent Linux kernels built from 2016 provide an estimation on how much memory is available for starting new applications without pushing the system into swap. This is a good estimation of the space available for containers. However, this information is not available in older kernels. For those kernels, we estimate the available memory by the free memory size and half of the inactive page size.

We extend the heartbeat content of NM to include the node utilization and the aggregated utilization of current containers. These heartbeats are sent to RM every second or few seconds depending on the size of the cluster. Upon receiving the heartbeat, the RM calculates the availability of the resources of each node as follow (see figure 5):

Denote by AR the allowed resource on node A. AR is the desired level of utilization that we would like node A to achieve when executing tasks on that node. Let ACU be the current aggregated resource utilization of all tasks executed on node A. We define the available configured resource by: $available\_configured\_resource = AR - ACU$ (5)

Denote by RU the Resource Usage Estimation that we calculated by applying the algorithm on section IV and by NC the capacity of the node. We find the available node resource by: $available\_node\_resource = NC - RU$ (6)

The *available resource* is the *min()* of eq. 5 and eq. 6.

In our prototype for YARN, we only consider CPU and memory. In order not to completely redesign the YARN scheduling system, our ”available resource” is only used to rescale the node total resource capacity as presented to Yarn scheduling. We scale up the total resource configured for the node if the available resource is positive and scale down if the available is negative. In the class "RMNode" which is managed by the main RM service, we add the information about the real capacity of node. In the class "SchedulingNode", we store the information of the current resource usage of node. We mostly modified the ”abstract scheduler” class which is the base of all YARN schedulers. In this class, we ”refresh” or "rescale" the capacity of the worker node and update the total capacity of the cluster after each estimation is received. Other decision making process and scheduling policies of FairShare, Capacity, and FIFO scheduler are kept as is.

B. System under test

To evaluate our new scheduler, we deploy a cluster with 6 Virtual Machines (VM) on OpenStack, each of them is configured with 8 virtual cores, 8 GBs of memory, and 160 GBs HDD. VMs are installed with Ubuntu LTS 16.04, Java OpenJDK 1.8, Hadoop 2.7.1, and Hive 2.1.0. We take the result of original YARN (version 2.7.1) as the base to compare with the improvement made by the modified YARN.

The original Yarn is configured with the minimum allocation set to 1 core and 1 GBs of memory. YARN is allowed to use at most: 7 cores and 7 GBs of memory. We leave the remaining resources to the OS and other utilities. For the prototype, the allowance is set similarly. For doing the estimation of node resource usage, we set the damping factor α to 0.125 as the in previous section.

C. Workloads

In order to make the evaluation repeatable, we use the standard TPC-DS benchmark which is de-facto industry standard benchmark for measuring the performance of decision support solutions. This TPC-DS benchmark is re-implemented for Hive which is published as Hive Testbench [11] by HortonWorks. The test was run with 200GB of data input (scale factor 200), partitioned by day. In this experiment, we focus on memory and CPU resources. Therefore, before each query is executed, we balance the blocks of HDFS files across the cluster once to maximize data-locality and to minimize the chance that the network becomes the bottleneck. A total of 60 queries were run as-is with no additional hinting, and there was no special tuning used for any of the queries. Some of the queries can be complex queries involving multiple tables and generating large intermediate datasets. Others can be queries that return large amounts of data for further processing by other tools.

Our algorithm is just a method to estimate how much resource the node does have and it do not change any scheduling policies of the schedulers like the Capacity scheduler, the FairShare scheduder, and the FIFO scheduler of YARN.

Therefore, we run the query serially and only make sure that the query can fill-up the cluster with parallel tasks on each node. Queries were executed in 3 times on MapReduce engine. In this setting, a query is a chain of MapReduce jobs, one MR job's output is another MR job's input. We notice that some of the jobs in the chain may not occupy the whole cluster, but the majority of them are able fill up the cluster. We also note that the containers are generated by the same query, so the containers are rather homogeneous, which is rather a pessimistic assumption for our algorithm.

D. Results

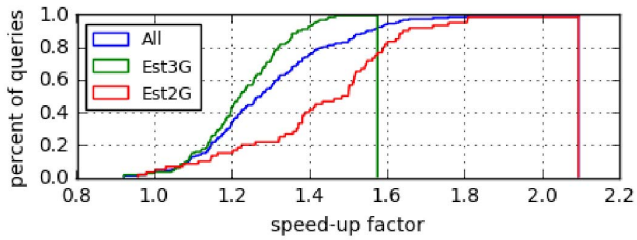


Figure 7: CDF for speed-up factor of all queries in the tests when comparing to original YARN as base

The speed-up factor is measured as the ratio of the completion time of the query on new system to the completion time of query on the original system. We report in figure 7 the cumulative distribution function of the speedup factors for all 60 executed queries. Queries having value for speedup factor larger than one completed faster than their version executed in standard Yarn. We reported the results of 2 different settings: *Est_3G* and *Est_2G*. In "Est_3G" and "Est_2G" users request 3 GBs and 2GBs of memory for each task, respectively. Almost all of the queries improve their completion time. A few of them show similar or longer completion times. As we investigate, these queries have many small stages in the data processing pipeline which cannot take the advantages of parallelism. In average, the queries in the "Est_3G" test complete 23% faster, and the queries in the "Est_2G" test complete 44% faster. "Est_2G" completes faster than "Est_3G" since the actual resource demand of tasks is less than 2 GBs of memory, the system needs less time to converge to the real utilization of nodes. We also ran tests with "Est_1G" but we did not presented in the figure because tasks often failed for both reservation model and estimation model because the memory usage of the task exceeded the user reservation for the task. We also confirm that during *Est_3G* and *Est_2G* tests very few tasks failed.

VII. Conclusion

The reservation model is simple and is a popular model for current resource management frameworks. However, it will waste resources and needs careful monitoring by cluster administrators. In this work, we aim to introduce a new and simple model for resource management which is based on resource usage feedback from worker nodes. This model helps to avoid reservation bottleneck by users over estimating their needs and to effectively utilize the resources without sacrificing for system stability or without the necessity to learn the workload resource usage.

In addition, we introduce a tracing tool to collect the job utilization, an algorithm to improve scheduling decision, a simulator to replay the trace and to study the stability of the system, and a new improved YARN that is aware of the resource usage at worker nodes.

With the tracing tool, we show that the reservation model leaves a large potential space for improving resource utilization and avoiding wasting resources. With the simulator, we prove that there is a better model to allocate resource to tasks without sacrificing system stability. With our modified YARN, we show that we can reduce the completion time for the jobs from 23% to 44% on the diversified Hive Testbench.

A thorough test on the real system with heterogeneous worker nodes would be the extension of our paper. We mainly focused on batch tasks. It would be interesting in a next step to take into account interactive tasks.

Acknowledgment

The research leading to these results has received funding from Orange SA and the EU commission in call H2020-644182, project "IOStack".

References

[1] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center.," in *NSDI*, vol. 11, pp. 22–22, 2011.

[2] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proc. 8th European Conf. on Comp. Systems*, pp. 351–364, ACM, 2013.

[3] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 5, ACM, 2013.

[4] D. K. Rensin, "Kubernetes-scheduling the future at cloud scale," 2015.

[5] Facebook, "Scheduling MapReduce jobs more efficiently with Corona." https://goo.gl/13Va9z, 2012. [Online; accessed 30-August-2016].

[6] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of heterogeneous resources in datacenters," in *Proc. of NSDI*, 2010.

[7] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 455–466, ACM, 2014.

[8] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. 3rd ACM Symp. on Cloud Computing*, p. 7, ACM, 2012.

[9] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao, "Efficient queue management for cluster scheduling," in *Proc. 11th European Conf. on Computer Systems*, p. 36, ACM, 2016.

[10] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proc. 10th European Conf. on Computer Systems*, p. 18, ACM, 2015.

[11] C. Shanklin, "Testbench for Apache Hive at any data scale.." https://goo.gl/FCthvV, 2017. [Online; accessed 11-January-2017].

[12] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pp. 499–510, 2015.

[13] P. Delgado, F. Dinu, D. Didona, and W. Zwaenepoel, "Eagle: A better hybrid data center scheduler," tech. rep., 2016.

[14] 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 *Proc. ACM Symp. on Cloud Computing*, pp. 1–14, ACM, 2014.

[15] N. Nielsen, "Turbocharging your Mesos cluster with oversubscription." https://mesosphere.com/blog/2015/08/26/turbocharging-your-mesos-cluster-with-oversubscription/, 2015. [Online; accessed 30-August-2016].

[16] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, "Clash of the titans: Mapreduce vs. spark for large scale data analytics," *Proc. of the VLDB Endowment*, vol. 8, no. 13, 2015.