MROrchestrator: A Fine-Grained Resource Orchestration Framework for MapReduce Clusters

Bikash Sharma[*], Ramya Prabhakar[*], Seung-Hwan Lim[†], Mahmut T. Kandemir[*], Chita R. Das[*]
[*]Department of Computer Science and Engineering, Pennsylvania State University
{bikash, rap244, kandemir, das}@cse.psu.edu
[†]Oak Ridge National Laboratory
lims1@ornl.gov

***Abstract*—Efficient resource management in data centers and clouds running large distributed data processing frameworks like MapReduce is crucial for enhancing the performance of hosted applications and increasing resource utilization. However, existing resource scheduling schemes in Hadoop MapReduce allocate resources at the granularity of fixed-size, static portions of nodes, called *slots*. In this work, we show that MapReduce jobs have widely varying demands for multiple resources, making the static and fixed-size slot-level resource allocation a poor choice both from the performance and resource utilization standpoints. Furthermore, lack of coordination in the management of multiple resources across nodes prevents dynamic slot reconfiguration, and leads to resource contention. Motivated by this, we propose *MROrchestrator*, a MapReduce resource Orchestrator framework, which can dynamically identify resource bottlenecks, and resolve them through fine-grained, coordinated, and on-demand resource allocations. We have implemented MROrchestrator on two 24-node native and virtualized Hadoop clusters. Experimental results with a suite of representative MapReduce benchmarks demonstrate up to 38% reduction in job completion times, and up to 25% increase in resource utilization. We further demonstrate the performance boost in existing resource managers like NGM and Mesos, when augmented with MROrchestrator.**

***Keywords*—Cloud, MapReduce, Resource Scheduling**

I. Introduction

Google MapReduce [11] has emerged as an important cloud activity for massive distributed data processing. Several academic and commercial organizations like Facebook, Microsoft and Yahoo! [1] use an open source implementation called Hadoop MapReduce [8]. In utility clouds like Amazon EC2, MapReduce is gaining prominence with customized services such as Elastic MapReduce [6] for providing the desired backbone for efficient Internet-scale data analytics.

MapReduce framework consists of two main components – a MapReduce engine and a Hadoop Distributed File System (HDFS). A master node runs the software daemons, *JobTracker* (MapReduce master) and *Namenode* (HDFS master), and multiple slave nodes run *TaskTracker* (MapReduce slave) and *Datanode* (HDFS slave). The input data is divided into multiple splits, which are processed in parallel by map tasks. The output of each map task is stored on the corresponding TaskTracker's local disk. This is followed by shuffle, sort and reduce steps (refer [23] for illustrative figure and description).

Currently, resource allocation in Hadoop MapReduce is done at the level of fixed-size resource splits of the nodes, called *slots*. A slot is a basic unit of resource allocation, representing a fixed proportion of multiple shared resources on a physical machine. Only one map/reduce task can run per slot at a time. The primary advantage of a slot is its simplicity and ease of implementation of the MapReduce paradigm.

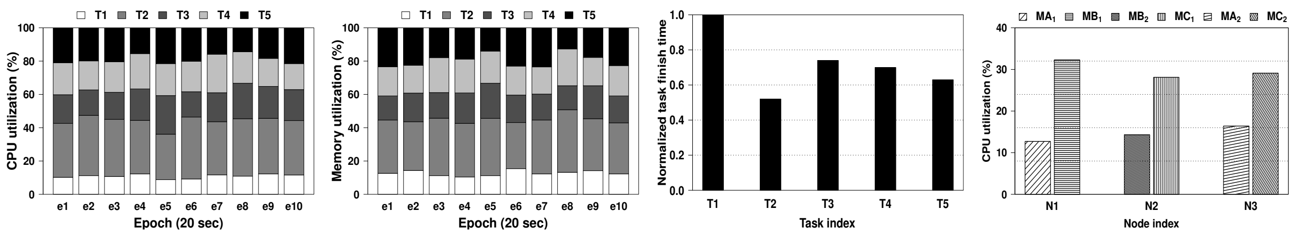
The slot-based resource allocation in Hadoop has three main disadvantages. The first downside is related to the fixed-size and static definition of a slot. Currently, in Hadoop framework, a node is configured with a fixed number of slots, which are statically [1] estimated in an ad-hoc manner irrespective of the machine's dynamically varying resource capacities. A slot is too coarse an allocation unit to represent the actual resource demand of a task, leading to wastage of individual resources when multiple of those are paired together in a slot. We observed in our experiments that slot-level allocations can lead to scenarios, where some of the resources are under-utilized, while others become bottlenecks. Likewise, analysis on a 2000-node Hadoop cluster at Facebook [12] has shown both the under and over cluster utilization due to significant disparity between tasks resource demands and slot resources.

The second and third problems are attributed to the lack of isolation and uncoordinated allocation of resources across the nodes of a Hadoop cluster, respectively. Although there is a form of 'slot-level' sharing of resources across jobs that is enforced by the Hadoop fair scheduler [15], there is no implicit partitioning of resources across jobs. This can lead to negative interference of resources among jobs. For example, a recent study on a Microsoft production MapReduce cluster indicates high contention for dynamic resources like CPU and memory [7]. Further, multiple nodes running different jobs are agnostic of their resource demands and contentions. Lack of global coordination in the management of multiple resources across the nodes can lead to situations, where local resource management decisions contradict with each other. Thus, when there are multiple concurrently executing MapReduce jobs, lack of isolation and uncoordinated sharing of resources can lead to poor performance.

Towards this end, we make the following contributions:

- We present the design and implementation of a novel resource management framework, *MROrchestrator*, that

[1]Based on the Hadoop code base, the number of slots per node is implemented as the minimum amount of the resource in the tuple {C-1, (M-2)/2, D/50}; where C = number of cores, M = memory in GB, D = disk space in GB, per node.



(a) Task CPU variation (b) Task memory variation (c) Task finish time variation (d) Global resource coordination

Fig. 1: Illustration of the variation in the resource usage and finish time of the constituent tasks of a *Sort* MapReduce job [(a)-(c)], and global coordination problem [(d)]. Y-axis in plot (c) is normalized w.r.t. maximum value.

provides fine-grained, dynamic and coordinated allocation of resources to MapReduce jobs. Based on the run-time resource profiles of tasks, MROrchestrator builds on-line resource estimation models for on-demand allocations. MROrchestrator is a software layer that assumes or requires no changes to the Hadoop framework, making it a simple, flexible, portable and platform generic design architecture.

- Detailed experimental analysis on two 24-node native and virtualized Hadoop clusters demonstrate the benefits of MROrchestrator in terms of reducing jobs finish times and boosting resource utilization. Specifically, MROrchestrator can achieve up to 38% reduction in job completion times and up to 25% increase in resource utilization, compared to slot-based resource allocations.
- MROrchestrator is complementary to the contemporary resource scheduling managers like Mesos [17] and Next Generation MapReduce (NGM) [16]. It can be augmented with these frameworks to boost system performance. Results from the integration of MROrchestrator with Mesos and NGM demonstrate up to 17% and 23.1% reduction in the job completion times, respectively. In terms of resource utilization, there is a corresponding increase of 12% (CPU), 8.5% (memory); 19% (CPU), 13% (memory), respectively.

II. MOTIVATION

A. *Need for dynamic allocation and isolation of resources*

As stated above, slot-level resource allocation, which does not provide any implicit resource partitioning and isolation, leads to high resource contentions. Furthermore, Hadoop framework is agnostic to the dynamic variation in the run-time resource profiles of tasks across map and reduce phases, and statically allocates resources in a coarse grained slot unit. This leads to wastage of resources since not all contained in a slot are proportionally utilized.

We provide some empirical evidences to the aforesaid problems related to the slot-based resource allocations. We run a *Sort* MapReduce job with 20 GB text input on a 24-node Hadoop cluster (experimental details are described in Section III-C1). We observe variation in resource utilization and finish times across the constituent map/reduce tasks of this job. Figures 1(a) and 1(b) show the CPU and memory utilization variation for five concurrently running reduce tasks of Sort job on a node across 10 randomly selected epochs of their total run-time. From these figures, we can observe that multiple tasks have different resource utilization across these epochs. Some task (*t2*) gets more CPU and memory entitlements and consequently finish faster (see Figure 1(c)) when compared with the other concurrently executing tasks. This observation can stem from a variety of reasons – node heterogeneity, data skewness and network traffic, which are prevalent in a MapReduce environment [7]. Here, the variation in disk utilization of these tasks is low since most of the data resides in the memory of each node (20 GB data split across 24 nodes). Further, due to an inherent barrier between the map and reduce phase [11], such variation in resource usage and finish times is disadvantageous since the completion time of a MapReduce job is constrained by the slowest task.

B. *Need for global resource coordination*

Cluster nodes hosting MapReduce jobs when unaware of the run-time resource profiles of constituent tasks can lead to poor system performance. We illustrate this aspect through an example. Consider 3 nodes N_1, N_2, N_3, executing 3 jobs A, B and C, with 2 map tasks each. That is, N_1 is shared by a map task of A and B, denoted as MA_1 and MB_1, N_2 is shared by a map task of B and C, denoted as MB_2 and MC_1, and N_3 is shared by a map task of A and C, denoted as MA_2 and MC_2. In this scenario, if we detect that MB_1 is hogging CPU allocation of MA_1, and change the CPU allocations between MB_1 and MA_1, we may not be able to improve job A's performance, because MC_2 may be contending with MA_2 for CPU. Also, MC_1 may be contending with MB_2 for CPU. Therefore, reducing MB_1's CPU allocation at N_1 (based on the local information) will only hurt B's performance without improving A's performance. The above scenario was observed in a simple experiment, where we ran Sort (A), Wcount (B) and DistGrep (C) jobs on a 24-node Hadoop cluster (details in Section III-C1). Figure 1(d) depicts the corresponding results. Such inefficiencies can be avoided with proper global coordination among all the cluster nodes.

III. DESIGN AND IMPLEMENTATION OF MRORCHESTRATOR

Figure 2 shows the architecture of *MROrchestrator*. Its main components include a *Global Resource Manager (GRM)*, running on the JobTracker, and a *Local Resource Manager (LRM)*, running on each TaskTracker. The GRM consists of two sub-components: (i) a *Contention Detector* that dissects

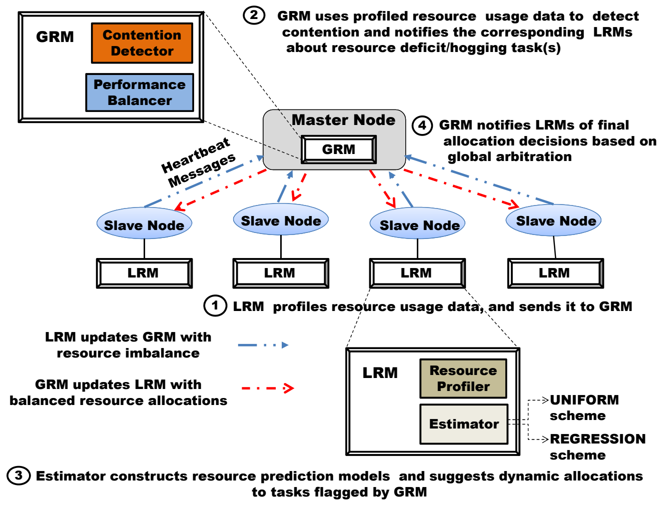

GRM
Contention Detector
Performance Balancer
2 GRM uses profiled resource usage data to detect contention and notifies the corresponding LRMs about resource deficit/hogging task(s)
Master Node
GRM
4 GRM notifies LRMs of final allocation decisions based on global arbitration
Heartbeat Messages
Slave Node
Slave Node
Slave Node
Slave Node
LRM
LRM
LRM
LRM
1 LRM profiles resource usage data, and sends it to GRM
LRM updates GRM with resource imbalance
GRM updates LRM with balanced resource allocations
LRM
Resource Profiler
Estimator
UNIFORM scheme
REGRESSION scheme
3 Estimator constructs resource prediction models and suggests dynamic allocations to tasks flagged by GRM

Fig. 2: Architecture of MROrchestrator.

the cause of resource contention and identifies both resource-deficit and resource-hogging tasks and (ii) a *Performance Balancer* that leverages the run-time resource estimations from each LRM to suggest the resource adjustments to each, based on the global coordinated view of all tasks running on TaskTrackers. The LRM also consists of two sub-components: (i) a *Resource Profiler* that collects and profiles the run-time resource usage/allocation of tasks at each *TaskTracker* and (ii) an *Estimator* that constructs statistical estimation models of a task's run-time performance as a function of its resource allocations. The Estimator can have different performance estimation models. As with other resource managers like Mesos, we currently focus on CPU and memory as the two major resources [7] for isolation and dynamic allocation. We plan to address disk and network resources in the near future.

MROrchestrator performs the following two main functions: (i) Detecting resource bottleneck and (ii) Performing dynamic resolution of resource contention across tasks and nodes.

Resource Bottleneck Detection: The functionalities of this phase can be summarized in two parts (denoted as steps **1** and **2** in Figure 2): **(1)** At regular intervals, the Resource Profiler in each LRM monitors and collects the run-time resource utilization and allocation information of each task using the *Hadoop profiler* [9]. This data is sent back to the Contention Detector module of GRM at JobTracker, piggy-backed with the heartbeat messages [8]. **(2)** On receiving these heartbeat messages from all the LRMs, the GRM can identify *which task* is experiencing bottleneck for *which resource*, on *which TaskTracker*, based on the following rationale. When different jobs, each with multiple map/reduce tasks, concurrently execute on a shared cluster, we expect similar resource usage profiles across tasks within each job. This assumption stems from the fact that tasks operating in either the map or reduce phase typically perform the same operations (map or reduce function) on similar input size, thus requiring identical amount of shared resources (CPU, memory, disk or network bandwidth). Due to practical factors like node heterogeneity, data skewness and cross-rack traffic, there is wide variation in the run-time resource profiles of tasks due to slot-based resource allocations in Hadoop. We exploit these characteristics here to identify potential resource bottlenecks and the affected tasks. For example, if a job is executing 6 map tasks across 6 nodes, and we see that the memory utilization of 5 of the 6 tasks on nodes 1–5 is close to 60%, but the memory utilization of only one of the tasks on node 6 is less than 25%, the GRM, based on its global view of all the nodes, should be able to deduce that the particular task is potentially memory deficit.

This approach might not work properly in some cases. For example, some outlier tasks may have very different resource demands and usage behavior, or because of workload imbalance, some tasks may get more share of input than the others. Thus, the resource profiles of such tasks may be quite deviant (although normal), and could be misinterpreted in this approach. In such scenarios, we adopt an alternative approach– we leverage the functionality of the JobTracker that can identify the straggler tasks [11] based on their progress score [26]. Since, resource contention is a leading cause for stragglers [7], [12], the GRM based on these two features can explicitly identify the potential resource contention induced straggling tasks from others.

Resource Bottleneck Mitigation: The functionalities of this phase can be explained using the remaining steps **3** and **4**, as shown in Figure 2. **(3)** After getting the information about both the resource deficit and resource hogging tasks from the GRM, the LRM at each TaskTracker invokes its Estimator module to estimate the tasks completion times (see Section III-A and Section III-B). The Estimator also maintains mappings between task execution time and run-time resource allocations. The Estimator builds predictive models for the finish time of a task as a function of its resource usage/allocation. The difference between the estimated and the current resource allocation is the resource imbalance that has to be dynamically allocated to the resource-deficit task. This information is piggy-backed in the heartbeat messages from the respective LRM to the GRM. After receiving the resource imbalances (if any) from every LRM, the Performance Balancer module in GRM executes the following simple heuristic: the GRM at JobTracker uses the global view of all running tasks on TaskTrackers to determine if the requested adjustment in resource allocation is in *sync* (see Section II-B) with other concurrent tasks (of the same phase and job) on other TaskTrackers. **(4)** Based on this global coordinated view, the GRM notifies each LRM of its approval of suggested resource adjustment, following which each LRM triggers the dynamic resource allocation.

A. *Resource allocation and progress of tasks*

In Hadoop MapReduce, a metric called *progress score* is used to monitor the run-time progress of each task. For the map phase, it represents the fraction of the input data read by a map task. For the reduce phase, it denotes the fraction of the intermediate data processed by a reduce task. Since, all the tasks in a map or reduce phase perform similar operations, the amount of resources consumed by a task is assumed to be proportional to the input data. In turn, the amount of consumed resources will be proportional to the progress score of each task. Therefore, as proposed in [26], in the absence of any change in the allocated resource for a task, the predicted task finish time, $\hat{T}$ can be expressed as

$$\hat{T} = \frac{1 - ProgressScore}{ProgressScore} T \,, \tag{1}$$

where T represents the elapsed time of a task. However, by varying the resource allocated to a task, the remaining finish time, $\hat{T}$ of a task may change. $\hat{T}$ can thus be represented as

$$\hat{T} = \alpha \frac{1 - ProgressScore}{ProgressScore} T \,. \tag{2}$$

Depending on α (which represents the resource scaling factor), the estimated finish time can increase ($\alpha > 1$, indicating resource deficiency) or decrease ($\alpha < 1$, indicating excess resource). $\alpha = 1$ signifies balanced task resource allocation. In order to control α, we propose two choices for the Estimator.

B. Estimator Design

The Estimator module in LRM is responsible for building the predictive models for tasks run-time resource usage/allocations. It can plug-in a variety of resource allocation schemes. We demonstrate two such schemes for MROrchestrator. The first scheme is called *REGRESSION*, which uses statistical regression models to obtain the estimated resource allocation for the next run epoch. The second scheme is called *UNIFORM*. It collects the past resource usage of all tasks in a phase, computes the *mean* usage across all these tasks and uses this value as the predicted resource allocation for the next epoch. Details of each scheme are described below:

1) REGRESSION scheme: This scheme determines the amount of resources (CPU and memory) to be allocated to each task at run-time, while considering the various practical scenarios that may arise in typical MapReduce clusters like node heterogeneity, workload imbalance, network congestion and faulty nodes [7]. It consists of two stages. The first stage inputs a time-series of progress scores of a given task and outputs a time-series of estimated finish times corresponding to multiple epochs in its life time. The finish time is estimated using Equation 1. The second stage takes as input a time-series of past resource usage/allocations across multiple epochs of a task's run-time (from its start time till the point of estimation). It outputs a statistical regression model that yields the estimated resource allocation for the next epoch as a function of its cumulative run-time. Thus, at any epoch in the life-time of a task, its predicted finish time is computed from the first model, and then the second model estimates the resource allocation required to meet the target finish time.

Separate estimation models are built for the CPU and memory profiles of a task. For the CPU profile, based on our experiments, we observed that the choice of a linear model achieves a good fit. For the memory profile, however, a linear model based on the training data from entire past resource usage history does not fit well, possibly due to the high dynamism in the memory usage of tasks. We use the following variant to better capture the memory footprints of the task– instead of using the entire history of memory usage, we only select training data over some recent time windows. The intuition is that a task has different memory usage across its map and reduce phases, and also within a phase. Thus, training data corresponding to recent time history is more representative of a task's memory profile. The memory usage is then captured by a linear regression model over the recent past time windows. The number of recent time windows

Algorithm 1 Computation of the estimated resource (at LRM).

```
Input: Time-series of past resource usage $TS_{\text{usage}}$ of concurrent
    running tasks, time-series of progress scores $TS_{\text{progress}}$ for a
    task $tid$, resource $R$, current resource allocation $R_{\text{cur}}$, Estimator
    scheme (UNIFORM or REGRESSION).
1: Split $TS_{\text{usage}}$ and $TS_{\text{progress}}$ into equally-sized multiple time-
   windows, ($W = \{w_1, w_2, ... , w_t\}$).
2: for each $w_i$ in $W_t$ do
3:   if allocation-scheme = UNIFORM then
4:     Compute the mean ($R_{\text{mean}}$) of the resource usage across
       all tasks of the same phase and job in the previous time-
       windows ($w_1 ... w_{i-1}$).
5:   else if allocation-scheme = REGRESSION then
6:     Get the expected finish time for task $tid$ using the progress-
       score based model (see Section III-B).
7:     Compute the expected resource allocation for the next
       epoch using the linear regression model.
8:   end if
9:   return Resource scaling factor, $\alpha$.
10: end for
```

Algorithm 2 Dynamic allocation of resources to tasks (at GRM).

```
Input: Resource type $R$ (CPU or memory), current allocation ($R_{\text{cur}}$),
    task ID ($tid$), current epoch ($e_{\text{cur}}$), resource scaling factor ($\alpha$).
1: Compute the estimated allocation ($R_{\text{est}}$) using Algorithm 1.
2: if $R_{\text{est}} > R_{\text{cur}}$ then
3:   Dynamically increase (e.g., $\alpha > 1$) the amount of resource
     allocation $R$ to task $tid$ by a factor ($R_{\text{est}}$ - $R_{\text{cur}}$).
4: else if $R_{\text{est}} < R_{\text{cur}}$ then
5:   Dynamically decrease (e.g., $\alpha < 1$) the amount of resource
     allocation $R$ to task $tid$ by a factor ($R_{\text{cur}}$ - $R_{\text{est}}$).
6: else
7:   Continue with the current allocation $R_{\text{cur}}$ in epoch $e_{\text{cur}}$.
8: end if
```

across which the training data is collected is determined by this simple rule. We collect data over as many past time windows till the memory prediction from the resulting estimation model is not worse than the average memory usage across all tasks (*e.g.,* UNIFORM scheme). This serves as a threshold for the maximum number of past time windows to be used for training. From empirical analysis, we found that 10% of the total number of past windows serves as a good estimate for the number of recent past windows. This threshold is adaptive, and depends on the prediction accuracy of the resulting memory usage model. Note that, the parameters of this simple linear model dynamically vary across a task's run-time epochs, and thus not shown here. Algorithm 1 describes the run-time estimated resource computations for a task. Algorithm 2 uses Algorithm 1 to perform the on-demand resource allocation.

2) UNIFORM scheme: This scheme is based on the intuitive notion of fair allocation of resources to tasks in order to reduce the run-time resource usage variation, and ensure nearly equal finish times across tasks. Here, the resource entitlement of each map/reduce task is set equal to the *mean* of the resource allocations across all tasks in the respective map and reduce phase of the same job. However, in practice, due to factors like machine heterogeneity, resource contention and workload imbalance, this scheme might not work well. We are demonstrating this scheme here primarily because of two reasons. First, it is a very simple scheme to implement with negligible performance overheads. Second, it highlights the fact that even a naive but intuitive technique like UNIFORM can achieve reasonable performance benefits.

C. Implementation options for MROrchestrator

We describe two approaches for the implementation of MROrchestrator based on the underlying infrastructure.

1) Implementation on a native Hadoop cluster: Here, we implement MROrchestrator on a 24-node Linux cluster, running Hadoop v0.20.203.0. Each node in the cluster has a dual 64-bit, 2.4 GHz AMD Opteron processor, 4GB RAM, and Ultra320 SCSI disk drives, connected with 1-Gigabit Ethernet. Each node runs on bare hardware without any virtualization layer (referred as *native Hadoop*). We use Linux control groups (LXCs) [10] for fine-grained resource allocation in Hadoop. LXCs are Linux kernel-based features for resource isolation, used for similar purposes as in Mesos [17].

2) Implementation on a virtualized Hadoop cluster: Motivated by the growing trend of deploying MapReduce applications on virtualized cloud environments [5], [18], we provide the second implementation of MROrchestrator on a virtualized Hadoop cluster in order to demonstrate its platform independence, portability and other potential benefits.

We allocate 1 virtual machine (VM) per node on a total of 24 machines (using same above native Hadoop cluster) to create an equivalent 24-node virtualized cluster, running Hadoop v0.20.203.0 on top of Xen [4] hypervisor. Xen offers advanced resource management techniques (like *xm* tool) for dynamic resource management of the overlying VMs. We configure Hadoop to run one task per VM. This establishes a one-to-one equivalence between a task and a VM, giving the flexibility to dynamically control the resource allocation to a VM (using *xm* utility), implying the control of resources at the granularity of individual task.

xm can provide resource isolation, similar to LXCs on native Linux. The core functionalities of GRM, namely Contention Detector and Performance Balancer, are implemented in Dom0. The LRM modules containing the functionalities of Resource Profiler and Estimator are implemented in DomUs. Similar to the native Hadoop case, the TaskTrackers on DomUs collect, profile resource usage data (using Hadoop profiler [9]), and send it to the JobTracker at Dom0 using the Heartbeat messages. We plan to explore the use of LXCs in the virtual cluster, and compare the benefits/trade-offs of these two implementation alternatives as part of our future work.

IV. Evaluation

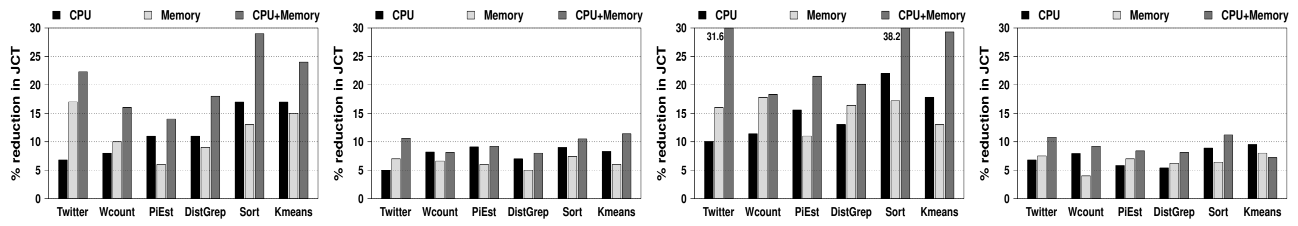
We use a workload suite consisting of the following six representative MapReduce jobs.

- *Sort:* sorts 20 GB of text data generated using Gridmix2 [13] provided random text writer.
- *Wcount:* computes the frequencies of words in the 20 GB of Wikipedia text articles [3].
- *PiEst:* estimates the value of Pi using quasi-Monte Carlo method that uses 10 million input data points.
- *DistGrep:* finds match of randomly chosen regular expressions over 20 GB of Wikipedia text articles [3].
- *Twitter:* uses the 25 GB twitter data set [2], and ranks users based on their followers and tweets.
- *Kmeans:* constructs clusters in 10 GB worth data points.

These jobs are chosen based on their popularity and being representative of real MapReduce workloads, with a diverse resource mix. The first four jobs are standard benchmarks available with Hadoop distribution [8], while the last two are in-house MapReduce implementations. The Sort, DistGrep, Twitter, and K-means are primarily CPU and I/O intensive benchmarks, while Wcount and PiEst are CPU bound jobs. Our primary metrics of interest are reduction in *job completion time* and increase in *resource utilization*. The base case corresponds to the slot-level sharing of resources with Hadoop fair scheduler [15]. Two map/reduce slots per node are configured for the baseline Hadoop.

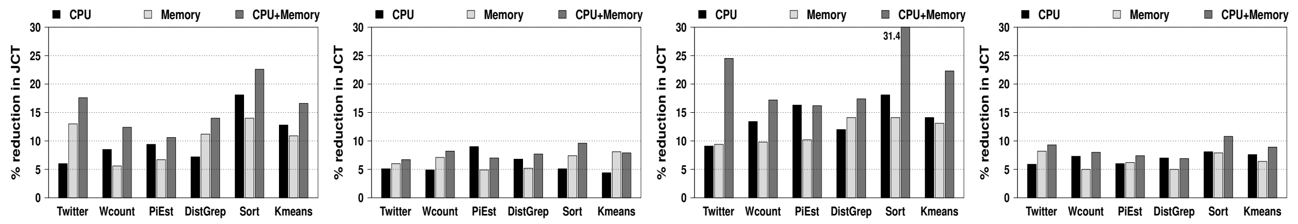
A. Results for native Hadoop cluster

Figure 3(a) shows the percentage reduction in the job completion times (JCTs) (which are in the order of tens of minutes) of the six MapReduce jobs, when each one is run in isolation (*Single job*), with the REGRESSION scheme. MROrchestrator is executed in three control modes: (a) MROrchestrator controls only CPU (*CPU*); (b) MROrchestrator controls only memory (*Memory*); and (c) MROrchestrator controls both CPU and memory (*CPU+Memory*) allocations. We show results separately for these three control modes. We summarize (in words, not shown in figures) each result with two values – *average* (mean value across all the 6 jobs) and *maximum* (highest value across all the 6 jobs). We can observe that the magnitude of decrease in JCTs varies at different scales for the three control modes. Across all the 6 jobs, *CPU+Memory* mode tends to yield the maximum reduction in JCTs, while the magnitude of reduction for *CPU* and *Memory* modes varies depending on the job resource usage characteristic. Specifically, we can notice an average and a maximum reduction of 20.5% and 29%, respectively, in the JCTs with *CPU+Memory* mode of MROrchestrator. Further, we see that Sort job makes extensive use of CPU (map phase), memory and I/O (reduce phase), and benefits the most. The percentage reduction in the JCTs for other five jobs varies depending on their resource sensitiveness. It is to be noted that larger jobs (w.r.t. both bigger input size and longer finish time) like Twitter, Sort



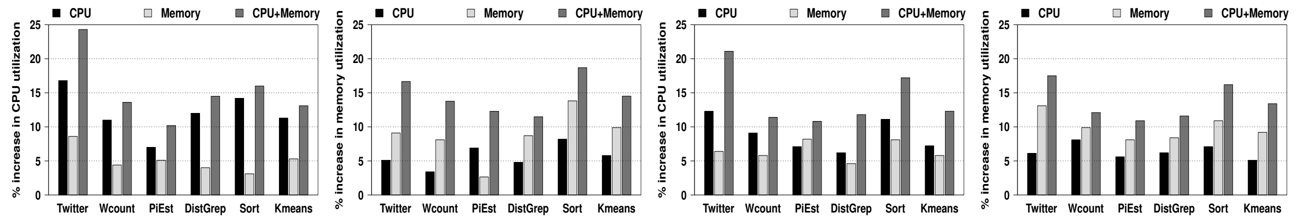
(a) REGRESSION (*Single job*) (b) UNIFORM (*Single job*) (c) REGRESSION (*Multiple jobs*) (d) UNIFORM (*Multiple jobs*)

Fig. 3: Reduction in Job Completion Time (JCT) for a *Single job* and *Multiple jobs* cases in native Hadoop cluster.



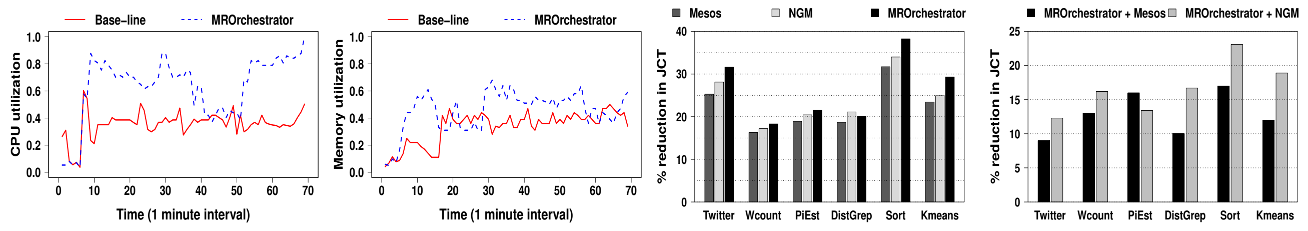
(a) REGRESSION (*Single job*) (b) UNIFORM (*Single job*) (c) REGRESSION (*Multiple jobs*) (d) UNIFORM (*Multiple jobs*)

Fig. 4: Reduction in Job Completion Time (JCT) for a *Single job* and *Multiple jobs* cases in virtualized Hadoop cluster.



(a) Native Hadoop (b) Native Hadoop (c) Virtualized Hadoop (d) Virtualized Hadoop

Fig. 5: Improvement in CPU and memory utilization in native and virtualized Hadoop clusters with *Multiple jobs*.



(a) CPU utilization time-series (b) Memory utilization time-series (c) MROrchestrator, Mesos, NGM (d) MROrchestrator + Mesos + NGM

Fig. 6: (a) and (b) demonstrate the dynamics of MROrchestrator in improving the cluster utilization. (c) shows the performance benefits with Mesos, NGM and MROrchestrator. (d) shows the comparison of MROrchestrator's integration with Mesos and NGM, respectively.

and Kmeans tend to benefit more with MROrchestrator, when compared to other relatively shorter jobs (PiEst, DistGrep, Wcount). The reason being that larger jobs run in multiple map/reduce phases, and thus can benefit more from the higher utilization achieved by MROrchestrator.

We next analyze the performance of MROrchestrator with the UNIFORM scheme. Figure 3(b) shows the results. We can observe an average and a maximum reduction of 9.6% and 11.4%, respectively, in the completion times. As discussed in Section III-B2, UNIFORM is a simple but naive scheme to allocate resources. However, it achieves reasonable performance improvements, suggesting the generality of MROrchestrator.

Next, we analyze the completion times of individual jobs in the presence of other concurrent jobs (*Multiple jobs*). For this, we run all the six jobs together. Figure 3(c) shows the percentage reduction in the completion times of the 6 jobs with the REGRESSION scheme. We observe an average and a maximum reduction of 26.5% and 38.2%, respectively, in the finish times with *CPU+Memory* mode. With the UNIFORM scheme and *CPU+Memory* (Figure 3(d)), we observe a 9.2% (average) and 12.1% (maximum) reduction in JCTs.

Figures 5(a) and 5(b) show the percentage increase in the CPU and memory utilization for *Multiple jobs* scenario. With the REGRESSION scheme and *CPU+Memory*, we see an increase of 15.2% (average) and 24.3% (maximum) for CPU; 14.5% (average) and 18.7% (maximum) for memory. We observe an average increase of 7% and 8.5% in CPU and memory utilization, respectively for the *Single job* case with *CPU+Memory* mode. With the UNIFORM scheme, the percentage increase seen in CPU and memory is within 10% both for *Single job* and *Multiple jobs* cases (corresponding plots not shown due to space constraint). Note that first, the benefits seen with MROrchestrator are higher for environments with *Multiple jobs*. This is due to better isolation, dynamic allocation and global coordination provided by MROrchestrator. Second, the control of both CPU and memory (*CPU+Memory*) with MROrchestrator yields the maximum benefits.

B. Results for virtualized Hadoop cluster

Figure 4(a) and Figure 4(b) show the percentage reduction in job completions for the *Single job* case with REGRESSION and UNIFORM schemes, respectively. We can notice a reduction of 15.6% (average) and 22.6% (maximum) with the REGRESSION scheme and *CPU+Memory* mode. With the UNIFORM scheme, the corresponding reduction is 7.8% (average) and 9.6% (maximum). When all the jobs are run concurrently, the percentage reduction is 21.5% (average) and 31.4% (maximum) for the REGRESSION scheme (Figure 4(c)). With the UNIFORM scheme, the percentage decrease in finish time is 8.5% (average) and 10.8% (maximum), respectively.

The resource utilization improvements are plotted in Figure 5(c) and Figure 5(d). We observe an increase in the utilization of 14.1% (average) and 21.1% (maximum) for CPU (Figure 5(c)); increase of 13.1% (average) and 17.5% (maximum) for memory (Figure 5(d)). These numbers correspond to REGRESSION scheme with *CPU+Memory*. For UNIFORM scheme, the percentage increase in CPU and memory utilization is less than 10%.

The performance benefits obtained from the implementation of MROrchestrator on a native Hadoop is marginally better than corresponding implementation on a virtualized Hadoop. There are two possible reasons for this. First, it is due to the CPU, memory or I/O (in particular) performance overheads associated with the Xen virtualization [21]. Second, configuring one task per virtual machine to achieve one-to-one correspondence between them seems to inhibit the degree of parallelization. However, the difference is not much, and we believe with the growing popularity of MapReduce in virtualized cloud environments [6], coupled with advancements in virtualization, the difference would shrink in near future.

To illustrate the dynamics of MROrchestrator, we plot Figure 6(a) and Figure 6(b), which show the snapshots of CPU and memory utilization of the native Hadoop cluster with and without MROrchestrator (*base-line*), and running the same workload (all 6 MapReduce jobs running concurrently). We can observe that MROrchestrator provides higher utilization compared to the *base-line*, since it is able to provide better resource allocation aligned with task resource demands.

C. MROrchestrator with Mesos and NGM

There are two contemporary resource scheduling managers – Mesos [17] and Next Generation MapReduce (NGM) [16] that provide better resource management in Hadoop. We believe MROrchestrator is complementary to both Mesos and NGM, and thus, integrate them to derive added benefits.

We first separately compare the performance of Mesos, NGM and MROrchestrator, *normalized* over the base case of default Hadoop with fair scheduling. Figure 6(c) shows the results. We can notice that the performance of MROrchestrator is better than both Mesos and NGM for all but one job (*PiEst*) (possibly because *PiEst* is a relatively shorter job, operating mostly in single map/reduce phase). The average (across all the 6 jobs) percentage reduction in JCT observed with MROrchestrator is 17.5% and 8.4% higher than the corresponding reduction seen with Mesos and NGM, respectively. We can observe that NGM has better performance than Mesos with the DRF scheme (a multi-resource scheduling scheme recently proposed for Mesos in [12]). One possible reason is the replacement of slot with the resource container unit in NGM, which provides more flexibility and finer granularity in resource allocations [16].

We next demonstrate the benefits from the integration of MROrchestrator with Mesos (*MROrchestrator+Mesos*) and NGM (*MROrchestrator+NGM*), respectively. The results are shown in Figure 6(d). We observe an average and a maximum reduction of 12.8% and 17% in JCTs for *MROrchestrator+Mesos*. For *MROrchestrator+NGM*, the average and maximum decrease in JCTs is around 16.6% and 23.1%, respectively. Further, we observe an increase of 11.7% and 8.2% in CPU and memory utilization, respectively for *MROrchestrator + Mesos*. For *MROrchestrator+NGM*, the corresponding increase in CPU and memory utilization is around 19.2% and 13.1%, respectively (plots not shown due to space constraint).

There are two main reasons for the observed better performance with MROrchestrator's integration. First, irrespective of the allocation units, the static characteristics in Mesos and NGM still persist. On the other hand, MROrchestrator dynamically controls and provides on-demand allocation of resources based on the run-time resource profiles. Second, the inefficiencies that arise due to the lack of global coordination among nodes have yet not been addressed both in Mesos and NGM. MROrchestrator incorporates such global coordination.

D. Performance overheads of MROrchestrator

There are some important design choices concerning the performance overheads of MROrchestrator. First, the frequency (or epoch duration) at which LRMs communicate with GRM is an important performance parameter. We performed detailed sensitivity analysis to determine the optimal frequency value by taking into consideration the trade-offs between prediction accuracy and performance overheads due to message exchanges. Based on the analysis, *20 seconds* (four times the default Hadoop heartbeat message frequency) is chosen as the epoch duration. Second, we discuss the expected delay in detecting and resolving resource bottlenecks in MROrchestrator. It consists of four major parts (overhead for each is with respect to the 20 seconds epoch duration): (i) resource usage measurements are collected and profiled every 5 seconds (= heartbeat message interval). The associated delay is negligible because of the use of a light-weight, built-in Hadoop profiler. (ii) time taken in resource bottleneck detection by Contention Detector is of the order of 10s of milliseconds; (iii) time overhead to build the predictive models by Estimator is less than 1% and 10% for UNIFORM and REGRESSION schemes, respectively; and (iv) time overhead to resolve contention by Performance Builder is within 4%.

Model Accuracy: With the REGRESSION scheme, 90% of predictions are within 10% and 19% accuracy for CPU and memory usage models, respectively. For UNIFORM scheme, the percentage error in resource estimations is within 35%.

Scalability: MROrchestrator can scale well with larger systems since the core modules (Estimator and Performance Builder) work on a fixed amount of recent history data, and thus, are largely independent of the increase in scale.

V. RELATED WORK

Resource Scheduling in Hadoop: Scheduling techniques for dynamic resource/slot adjustment for MapReduce jobs based on their estimated completion times have been recently addressed in [19], [20], [24]. Different resource scheduling policies for Hadoop MapReduce have also been lately proposed [14], [15], [22], [25].

However, these works do not address the fundamental cause of performance bottlenecks in Hadoop, which is related to the static and fixed-size slot-level resource allocation. The current Hadoop schedulers [14], [15] are not resource-aware of running jobs. Our proposed solution builds upon these shortcomings. Another important aspect where this paper differs from existing systems is that we estimate the requirements of each job at run-time (through predictive models) rather than assuming that they are known a priori.

Fine-grained resource management in Hadoop: The closest works to ours are Mesos [17] and Next Generation MapReduce (NGM) (a.k.a YARN) [16]. Mesos is a resource scheduling manager that provides fair share of resources across diverse cluster computing frameworks like Hadoop and MPI. NGM is the latest architecture of Hadoop MapReduce. However, the dynamic run-time resource management is yet not considered in both Mesos and NGM. MROrchestrator specifically addresses this missing piece.

We believe our current work is complementary to these systems in that we share the same motivations and final goals, but we attempt to provide a different approach to handle the same problem, with a coordinated, fine-grained and dynamic resource management framework, called MROrchestrator.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we analyzed the disadvantages of fixed-size and static slot-based resource allocation in Hadoop MapReduce. Based on the insights, we proposed the design and implementation of a flexible, fine-grained, dynamic and coordinated resource management framework, called *MROrchestrator*, that can efficiently manage the cluster resources. Results from the implementations of MROrchestrator on two 24-node physical and virtualized Hadoop clusters, with representative workload suites, demonstrate that up to 38% reduction in job completion times, and up to 25% increase in resource utilization can be achieved. We further show how contemporary resource scheduling managers like Mesos and NGM, when augmented with MROrchestrator can boost their system's performance.

We are pursuing two extensions to this work. First, we plan to extend MROrchestrator with control of other resources like disk and network bandwidth. Second, we plan to evaluate MROrchestrator on a large cloud environment.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their comments towards improving this paper. This work is supported in part by National Science Foundation (NSF) grants (CNS-1152479, CCF-1147388), and a research grant from Google Inc.

REFERENCES

[1] Powered-by-hadoop. http://wiki.apache.org/hadoop/PoweredBy.
[2] Twitter traces. http://an.kaist.ac.kr/traces/WWW2010.html.
[3] Wikitrends. http://trendingtopics.org.
[4] Xen hypervisor. http://www.xen.org.
[5] Amazon. Aws. http://aws.amazon.com.
[6] Amazon. Mapreduce. http://aws.amazon.com/elasticmapreduce/.
[7] G. Ananthanarayanan, S. Kandula, and et al. Reining in the Outliers in MapReduce Clusters using Mantri. In *USENIX OSDI*, 2010.
[8] Apache. Hadoop. http://hadoop.apache.org.
[9] Apache. Hadoop profiler: Collecting cpu and memory usage for mapreduce tasks. https://issues.apache.org/jira/browse/MAPREDUCE-220.
[10] cgroups. Linux Control Groups. http://en.wikipedia.org/wiki/Cgroups.
[11] J Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI*, 2004.
[12] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *USENIX NSDI*, 2011.
[13] Gridmix2. http://hadoop.apache.org/mapreduce/docs/current/gridmix.
[14] Hadoop. Capacity Scheduler. http://hadoop.apache.org/common/docs/r0.19.2/capacity_scheduler.html.
[15] Hadoop. Fair Scheduler. http://hadoop.apache.org/common/docs/r0.20.2/fair_scheduler.html.
[16] Hadoop. Next Generation MapReduce Scheduler. http://developer.yahoo.com/blogs/hadoop/posts/2011/03/mapreduce-nextgen-scheduler/.
[17] B. Hindman, A. Konwinski, and et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *USENIX NSDI*, 2011.
[18] S. Ibrahim, H. Jin, L. Lu, L. Qi, S. Wu, and X. Shi. Evaluating Mapreduce on Virtual Machines: The Hadoop Case. In *IEEE CloudCom*, 2009.
[19] P. Jorda, C. Claris, and et al. Resource-Aware Adaptive Scheduling for MapReduce Clusters. In *ACM/USENIX/IFIP Middleware*, 2011.
[20] P. Jorda, C. David, B. Yolanda, S. Malgorzata, and W. Ian. Performance-Driven Task Co-Scheduling for Mapreduce Environments. In *IEEE/IFIP NOMS*, 2010.
[21] A. Menon, J. Santos, Y. Turner, J. Janakiraman, and W. Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *ACM/USENIX VEE*, 2005.
[22] T. Sandholm and K. Lai. Dynamic Proportional Share Scheduling in Hadoop. In *Job scheduling strategies for parallel processing*, 2010.
[23] B. Sharma, R. Prabhakar, S. Lim, M. Kandemir, and C. Das. MROrchestrator: A Fine-Grained Resource Orchestration Framework for Hadoop MapReduce. Technical Report CSE-12-001, Penn State University.
[24] A. Verma, L. Cherkasova, and R. Campbell. ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments. In *ICAC*, 2011.
[25] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job Scheduling for Multi-User MapReduce Clusters. Technical Report UCB/EECS-2009-55, UC Berkeley.
[26] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving Mapreduce Performance in Heterogeneous Environments. In *USENIX OSDI*, 2008.