# MROrchestrator: A Fine-Grained Resource Orchestration Framework for Hadoop MapReduce

Bikash Sharma, Ramya Prabhakar, Seung-Hwan Lim, Mahmut T. Kandemir and Chita R. Das
Department of Computer Science and Engineering
The Pennsylvania State University
{bikash, rap244, seulim, kandemir, das}@cse.psu.edu
**Technical Report CSE-12-001, January 2012**

***Abstract*—Efficient resource management in data centers and clouds running large distributed data processing frameworks like Hadoop is crucial for enhancing the performance of hosted MapReduce applications, and boosting the resource utilization. However, existing resource scheduling schemes in Hadoop allocate resources at the granularity of fixed-size, static portions of the nodes, called *slots*. A slot represents a multi-dimensional resource slice, consisting of CPU, memory and disk on a machine. 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 the nodes, prevents dynamic slot reconfiguration and leads to resource contention. Towards this end, we perform a detailed experimental analysis of the performance implications of slot-based resource scheduling in Hadoop. Based on the insights, we propose the design and implementation of *MROrchestrator*, a MapReduce resource Orchestrator framework, that can dynamically identify the 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 suite of representative MapReduce benchmarks demonstrate up to 38% improvement in reducing job completion times, and up to 25% increase in resource utilization. We further show how popular resource managers like NGM and Mesos when augmented with MROrchestrator can boost their performance.**

## I. INTRODUCTION

MapReduce [10] has emerged as an important programming model for big-scale data processing in large distributed environments. Several academic and commercial organizations use Apache Hadoop [7], an open source implementation of MapReduce. In cloud computing environments like Amazon Web Services [4], Hadoop is gaining prominence with services such as Amazon Elastic MapReduce [5], for providing the required backbone for Internet-scale data analytics.

Figure 1 shows a generic Hadoop framework. It consists of two main components – a MapReduce engine and a Hadoop Distributed File System (HDFS). It adopts a master/slave architecture, where a single 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). In a typical MapReduce job, the framework divides the input data 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 a shuffle step, in which the intermediate map output is copied across the network, followed by a sort step, and finally the reduce step.

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. The top portion of Figure 1 shows the conceptual view of a slot. At any given time, only one map/reduce task can run per slot. The primary advantage of slots is the ease of implementation of the MapReduce programming paradigm. Slot offers a simple but coarse abstraction of the available resources on a machine. It provides a means to cap the maximum degree of parallelization.

The slot-based resource allocation in Hadoop has three main disadvantages. The first downside is related to the fixed-size and static definition of slot. Hadoop is configured with a fixed number of slots per machine, which are statically[1] estimated in an ad-hoc manner irrespective of the machine's dynamically varying resource capacities. Slot-level uniform and fixed-size allocation could lead to scenarios, where some of the resources are under-utilized, while others become bottlenecks. A slot is too coarse an allocation unit to represent the actual resource demand of a task. Analysis on a 2000-node Hadoop cluster at Facebook [11] has shown both the under and over utilization of resources due to significant disparity between tasks resource demands and slot resources.

There is an implicit assumption in Hadoop that the tasks running in the slots have similar resource characteristics. However, in a shared Hadoop MapReduce cluster, jobs from multiple users have widely varying resource demands [11]. Besides, the constituent tasks of the jobs have different resource usage profiles across the map and reduce phases (*e.g.,* some may be CPU-intensive, while others I/O intensive). Hadoop currently lacks abilities to dynamically configure the number of slots based on the run-time resource footprints of constituent tasks. Since the slots are configured statically, the dynamic

[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.

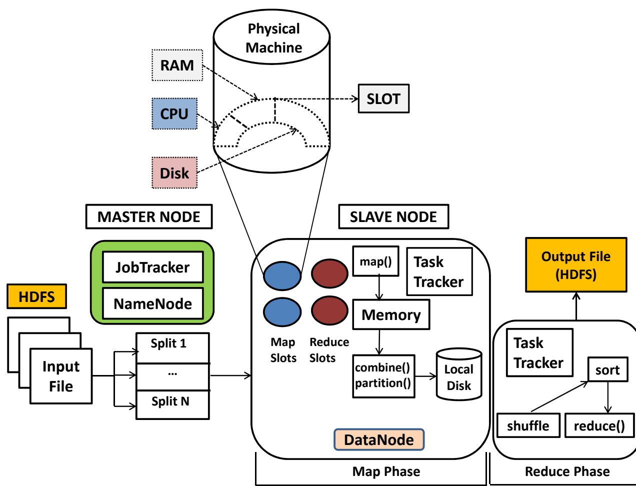



Fig. 1: Hadoop Framework.

variations in the run-time resource characteristics of tasks are not well captured. Thus, the fixed-size and static slot model precludes a fine-grained control over resource allocation and leads to wastage of 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 [14], but there is no implicit partitioning or isolation of resources between jobs. In MapReduce clusters, presence of stragglers or outliers [10] prolong the executions of MapReduce jobs. A recent analysis from a Microsoft production MapReduce cluster indicates that contention for dynamic machine resources like CPU and memory among tasks contributes significantly towards the prominence of stragglers [6]. Further, multiple nodes running different tasks are agnostic of their resource demands and contentions. Lack of global coordination in the management of multiple resources across the nodes, could 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 because of resource contention. These inefficiencies in Hadoop make resource scheduling more challenging and different from traditional cluster scheduling [1]. Very recently, Apache Hadoop released the Next Generation MapReduce architecture [15] that has also acknowledged the above mentioned negative impacts of the slot-level allocation.

Towards this end, we make the following **contributions**:

- Through detailed experimental analysis, we highlight the shortcomings associated with the current slot-based resource allocation in Hadoop. Based on the insights, we advocate the need for the transition from slot-based resource scheduling to a more flexible and run-time resource proportional allocation model.
- We present the design and implementation of the new resource management framework, *MROrchestrator*, that provides fine-grained, dynamic and coordinated allocation of resources to jobs. Based on the run-time resource profiles of tasks, MROrchestrator builds resource estimation models to provide on-demand allocations. It dynamically detects resource bottlenecks and reallocates resources among jobs to fairly share the bottleneck. MROrchestrator assumes no changes to the Hadoop framework, and optimizes the usage of the resources.
- Detailed measurement based analysis on two 24-node native and virtualized Hadoop clusters demonstrate the benefits of MROrchestrator in terms of reducing the job completion times and boosting resource utilization. Specifically, MROrchestrator can achieve up to 38% reduction in job completions and up to 25% increase in resource utilization over the base case of slot-based fair scheduling [14].
- The proposed MROrchestrator is complementary to the contemporary resource scheduling managers like Mesos [16] and Next Generation MapReduce (NGM) [15]. It can be augmented with these frameworks to achieve further overall system performance benefits. Results from the integration of MROrchestrator with Mesos and NGM demonstrate a percentage reduction of up to 17% and 23.1%, in the job completion times, respectively. In terms of resource utilization, there is an increase of 12% (CPU), 8.5% (memory); 19% (CPU), 13% (memory) corresponding to the integration of MROrchestrator with Mesos and NGM, respectively.

The rest of this paper is organized as follows: Section II outlines motivation for the problem addressed. The design and implementation details of MROrchestrator is described in Section III. Section IV presents the experimental evaluations, followed by related literatures in Section V. The conclusions are summarized in Section VI.

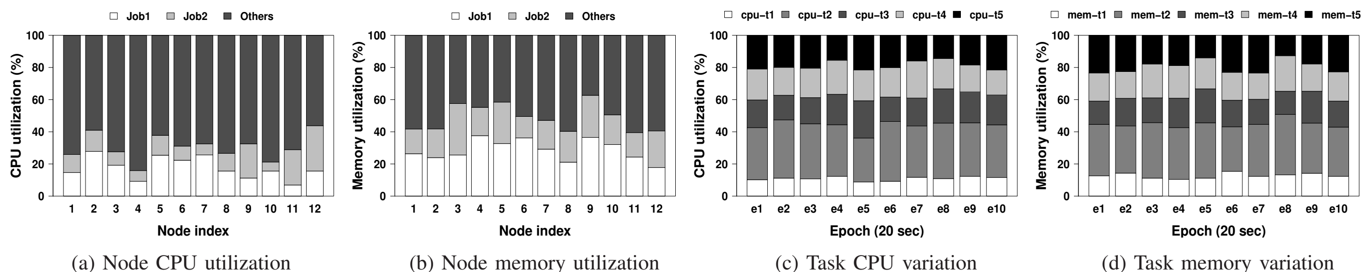
## II. MOTIVATION

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

Hadoop MapReduce jobs from multiple users have widely varying resource usage characteristics. 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 different map/reduce phases, and statically allocates resources in a coarse grained slot unit. This leads to wastage of resources since not all resources contained in a slot are utilized in the same proportion.

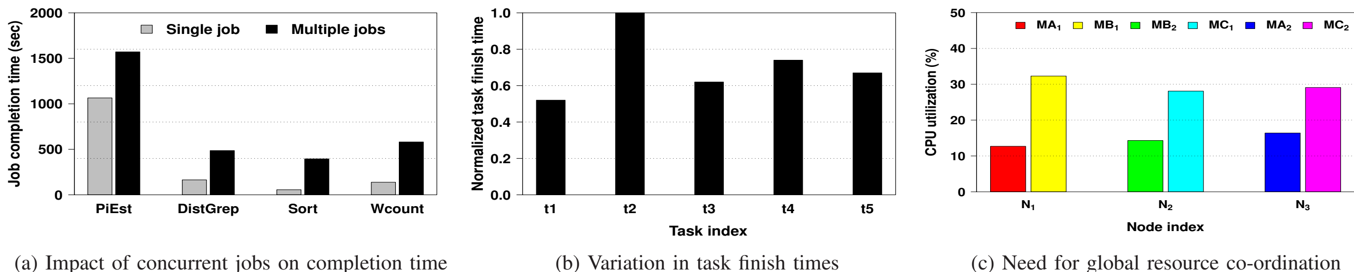
We provide some empirical evidences for the problems related to the slot-based resource allocation with two simple experiments. In the first experiment, we concurrently run a group of four MapReduce jobs – PiEst, DistGrep, Sort and Wcount (Section IV contains details of the experimental setup). Here, we focus on the performance of Sort (*Job1*) and Wcount (*Job2*). We group the other two jobs (PiEst and DistGrep) into the *Others* category. We first measure the average CPU and memory utilization[2] when each of Job1, Job2, is run

[2]CPU utilization is the percentage of total CPU time; memory utilization is the percentage used of the total memory. These are obtained using *sar* utility in Linux.




(a) Node CPU utilization (b) Node memory utilization (c) Task CPU variation (d) Task memory variation

Fig. 2: Illustration of resource contention. Experimental results on a 24-node Hadoop cluster with 15 GB of text input for Job1 (Sort), 10 GB text input for Job2 (Wcount), and randomly chosen data for Others (mix of DistGrep and PiEst).




(a) Impact of concurrent jobs on completion time (b) Variation in task finish times (c) Need for global resource co-ordination

Fig. 3: Resource profile variation and global in-coordination in Hadoop MapReduce.

separately. We notice CPU utilization of 42.2% and 28.6%; memory utilization of 68.8% and 79.8%, for Job1 and Job2, respectively. We then run Job1, Job2 and Others together. Figures 2(a) and 2(b) show the average CPU and memory utilization of the jobs across 12 nodes. The average CPU utilization is 17.4% and 12.9%; average memory utilization is 28.6% and 20.1% for Job1 and Job2, respectively. We clearly see the reduction in resource utilization of Job1, Job2, in the presence of other jobs, due to potential resource contention. Furthermore, resource utilization is non-uniform across the jobs. For example, in Figure 2(b), for node 11, the memory utilization due to Job1, Job2 and Others is around 24%, 15% and 61%, respectively. Figure 3(a) shows the corresponding job completion times of these four jobs when run in isolation (*Single job*) as well as in combination (*Multiple jobs*). On an average, we find *2.9X* increase in the completion times with *Multiple jobs* over *Single job*. The above scenarios occur due to the fact that slot-based resource allocation does not provide implicit resource isolation between jobs, leading to resource contention and performance deterioration.

In the second experiment, we demonstrate variation in resource utilization and finish times across the constituent map/reduce tasks of a single MapReduce job. Figures 2(c) and 2(d) show the CPU and memory utilization variation for five concurrently running reduce tasks of Sort (Job1) on one 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 3(b)) when compared with the other concurrently executing tasks. This observation stems from a variety of reasons – node heterogeneity, data skewness and network traffic, which are prevalent in typical MapReduce environment. Further, due to an inherent barrier between the map and reduce phase [10], such variation in resource usage and finish times is disadvantageous since the end-to-end completion time of a MapReduce job is determined by the finish time of the slowest task. In such scenarios, the system resources are wasted when allocation is done in the form of fixed-size static slots, which fails to account for the run-time dynamism in the resource usage of tasks.

### B. Need for a global resource coordination

Cluster nodes hosting MapReduce applications when unaware of the run-time resource usage 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 in $N_1$ (based on local information) will only hurt B's performance without improving A's performance. The above scenario is the result of a simple experiment, where we run Sort (A), Wcount (B) and DistGrep (C) jobs on a 24-node Hadoop cluster (details in Section III-C1). Figure 3(c) depicts the corresponding results. Such inefficiencies can be avoided with proper global coordination among all the nodes.

## III. Design and Implementation of MROrchestrator

In this section, we describe the design and implementation of our proposed resource management framework, *MROrches-*

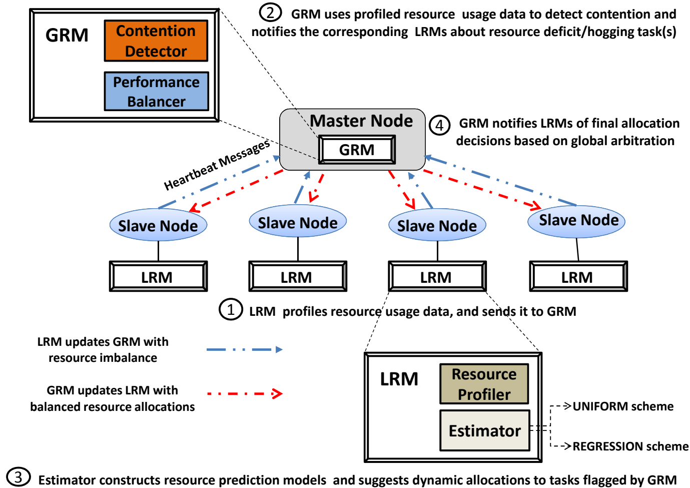



Fig. 4: Architecture of MROrchestrator.

*trator*. Figure 4 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 the cause of resource contention and identifies both resource straggling 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 LRM, based on a 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 task performance at run-time as a function of its resource allocations. The Estimator can have different performance models. As with other resource managers like Mesos, we currently focus on CPU and memory, as the two major resources for dynamic allocation and isolation. We plan to address disk and network bandwidth isolation in the near future.

Figure 5 shows the position of the MROrchestrator framework in a system stack hosting Hadoop. MROrchestrator is a software resource management layer that runs on top of a cluster infrastructure. In some cases, there is a resource manager software layer like Mesos [16], that runs on top of the infrastructure, which in turn can host MROrchestrator. An important aspect to note from Figure 5 is that MROrchestrator only operates on/controls the underlying infrastructure resources in a fine-grained manner to meet its objectives. It assumes or requires no changes to the overlying Hadoop framework, making it a simple, flexible and generic scheme that can be ported to any platform. MROrchestrator performs two main functions (i) Detect resource bottleneck and (ii) Perform dynamic resolution of resource contention across tasks. We describe these functions in detail below.

**Resource Bottleneck Detection**: The functionalities of this phase can be summarized in two parts (denoted as steps 1 and 2 in Figure 4): **(1)** At regular intervals, the Resource Profiler in each LRM monitors and collects run-time resource utilization and allocation information of each task using the *Hadoop profiler* [8]. This data is sent back to the Contention Detector module of GRM at JobTracker, piggy-backed with the heartbeat messages[3]. We modify this interval to suit our epoch duration for performing dynamic allocations via MROrchestrator (see Section IV-D). **(2)** On receiving these heartbeat messages from all the LRMs, the GRM can identify *which task* is experiencing a 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 same operations (map or reduce function) on similar input size, thus requiring identical amount of shared resources (CPU, memory, disk or network bandwidth). However, as discussed in Section II, 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%, GRM, based on its global view of all the nodes, should be able to deduce that the particular task is memory deficit.

This approach will not work properly in some cases. For example, some outlier jobs 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 [10] based on their progress score [26]. Essentially, there are four main reasons for the presence of stragglers in a MapReduce cluster – (i) resource contention; (ii) data skewness; (iii) network traffic; and (iv) faulty machines (for more details, refer [6]). Among these, resource contention is a leading cause for stragglers [6], [11]. The GRM uses this feature to 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 3–4, as shown in Figure 4. **(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 task completion time (see Section III-A and Section III-B). The Estimator also maintains mappings between task execution time and run-time resource allocations.

[3] In the default Hadoop implementation, each slave node sends heartbeat messages to the master node every 5 seconds [7]. These heartbeat messages contain information about the status of slots on each node, besides other information like task failures.

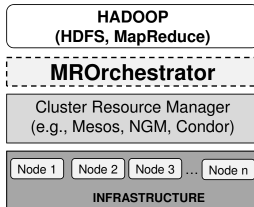



Fig. 5: Position of MROrchestrator in the system stack.

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. The Performance Balancer module in GRM on receiving the resource imbalances (if any) from every LRM executes this 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 concurrently 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 the LRM triggers the dynamic resource allocation for the task. Further details of the adopted methodology are described in the following sub-sections.

### 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 represents the fraction of the intermediate data processed by a reduce task. All tasks in a map or reduce phase perform similar operations, thus the amount of resources consumed by a task is assumed to be proportional to the amount of 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, on varying the resource allocated to a task, the remaining finish time, $\hat{T}$ of a task may change. Thus, $\hat{T}$ can be represented as

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

Depending on $\alpha$ (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 that the task resource usage/allocation is balanced. Thus, we may want to adjust $\alpha$ depending on the relative progress of each task. In order to control the resource scaling factor $\alpha$, we propose two choices – REGRESSION and UNIFORM schemes, which are described next.

### B. Estimator

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 here with MROrchestrator. The first scheme is called *REGRESSION*, and it uses statistical regression models to get the estimated resource allocation for the next 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 outputs this value as the predicted resource allocation for the next epoch. Details of each 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 [6]. 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 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 predicts the estimated 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 did not turn out to be a good fit, 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 recent time windows in past. 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 windows 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 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 the training. From empirical analysis, we found 10% of the total number of past windows as a

**Algorithm 1** Dynamic allocation of resources to tasks (at GRM).

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

**Algorithm 2** Computation of the estimated resource (at LRM).

**Input:** Time-series of past resource usage $TS_{usage}$ of concurrent running tasks, time-series of progress scores $TS_{progress}$ for a task *tid*, resource *R*, current resource allocation ($R_{cur}$), Estimator scheme (UNIFORM or REGRESSION).
1: Split $TS_{usage}$ and $TS_{progress}$ 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 ($R_{mean}$) 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 regression model.
8: **end if**
9: **return** Resource scaling factor, $\alpha$.
10: **end for**

good estimate for the number of recent past windows to use for training. This threshold is adaptive, and depends on the prediction accuracy of the resulting memory usage model. Algorithm 1 and Algorithm 2 describe the pseudo-code for the dynamic allocation of resources to tasks based on the above explanations. Specifically, Algorithm 2 describes how the run-time estimated resources for a task can be computed. Algorithm 1 uses Algorithm 2 to perform the on-demand resource allocation.

*2) UNIFORM scheme:* This scheme is based on the intuitive notion of fair allocation of resources to each map/reduce task in order to reduce the run-time resource usage variation and ensure approximately equal finish time across tasks. In this scheme, each map/reduce task's resource entitlement at a given point in time is set equal to the *mean* of the resource allocations across all tasks in the respective map and reduce phases of the same job. However, in practice, due to factors like machine heterogeneity, resource contention and workload imbalance, this scheme based on the assumption of homogeneity 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, when plugged with MROrchestrator, 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:* In this option, 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* in the paper). We use Linux control groups (LXCs) [9] for fine-grained resource allocation in Hadoop. LXCs are Linux kernel-based features for *resource isolation*. Using LXCs, we can at run-time increase/decrease the CPU and memory allocation at the granularity of individual map/reduce task (which is a process in Linux). MROrchestrator using LXCs can provide on-demand and fine-grained control of CPU and memory for map/reduce tasks. LXCs are light-weight with little performance overheads [9].

Out of the total 24 nodes, one node is configured as the master and the remaining 23 nodes are set up as slaves. The master runs the GRM component and manages Contention Detector and Performance Balancer modules. Each slave node runs the LRM module and implements the functionalities of Resource Profiler and Estimator. Hadoop profiler [8] is used to collect and profile the resource usage data. It allows TaskTrackers running on slave nodes to report their run-time CPU and memory usage to JobTracker on master. The dynamic resource allocation to tasks is performed using Algorithm 1 and Algorithm 2.

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

We allocate 2 virtual machines per node on a total of 12 machines (of the same native Hadoop cluster (Section III-C1)) to create an equivalent 24-node virtualized cluster, running Hadoop v0.20.203.0 on top of Xen [3] hypervisor. Xen offers advanced resource management techniques (like *xm* tool) for dynamic resource management of the overlying virtual machines. We configure Hadoop to run one task per virtual machine. This establishes a one-to-one equivalence between a task and a virtual machine, giving the flexibility to dynamically control the resource allocation to a virtual machine (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

| Workload | Resource sensitivity |
|---|---|
| Sort | CPU+I/O |
| Wcount | CPU+Memory |
| PiEst | CPU |
| DistGrep | CPU+I/O |
| Twitter | CPU+Memory |
| K-means | CPU+I/O |

TABLE I: Resource sensitiveness of workloads.

functionalities of Resource Profiler and Estimator are implemented in DomUs. Similar to the case with native Hadoop, the TaskTrackers on DomUs collect, profile resource usage data (using Hadoop profiler), and send it to the JobTracker at Dom0 using the Heartbeat messages. Using Algorithm 1 and Algorithm 2, the dynamic resource allocation to tasks is performed via *xm* utility.

## 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 [12] provided random text writer.
- *Wcount:* computes the frequencies of words in the 20GB of Wikipedia text articles [2].
- *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 on 20 GB of Wikipedia text articles [2].
- *Twitter:* uses the 25GB twitter data set [17], 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 MapReduce benchmarks, with diverse resource mix. The first four jobs are standard benchmarks available with Hadoop distribution [7], while the last two are in-house MapReduce implementations. Table I shows the resource sensitivity characteristics of these 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 [14].

### A. Results for native Hadoop cluster

Figure 6(a) shows the percentage reduction in the execution times 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 the three control modes. Each result is reported 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 reduction in job completion time varies at different scales for the three control modes. Across all the 6 jobs, *CPU+Memory* mode tends to yield the maximum reduction in job completion times, 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 completion times when MROrchestrator controls both CPU and memory allocations (*CPU+Memory*). 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 job completion times for other five jobs varies depending on their resource footprints and sensitiveness (Table I). It is to be noted that larger jobs (w.r.t. both bigger input size and longer finish time) like Twitter, Sort 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 based resource orchestration.

We next analyze the performance of MROrchestrator with the UNIFORM scheme. Figure 6(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 an intuitive but naive scheme to allocate resources. However, it achieves reasonable performance improvements, suggesting the intuitive benefits and generality of MROrchestrator with different plug-in resource estimation techniques.

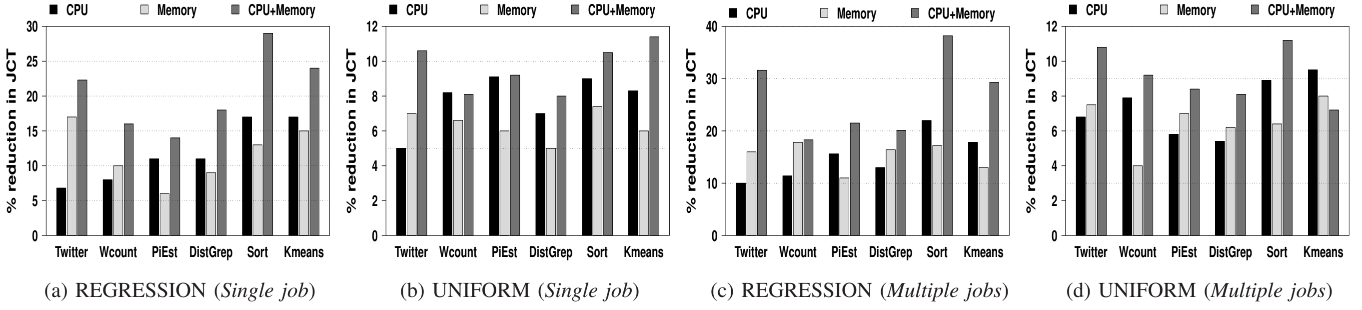
Next, we analyze the completion times of individual jobs in the presence of other concurrent jobs (*Multiple jobs*). For this, we run all the six jobs together and note the finish time of each. Figure 6(c) shows the percentage reduction in the completion times of the 6 jobs with the REGRESSION scheme.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

Figures 7(a) and 7(b) show the percentage increase in the CPU and memory utilization for *Multiple jobs* scenario. With the REGRESSION scheme, we see an increase of 15.2% (average) and 24.3% (maximum) for CPU; 14.5% (average) and 18.7% (maximum) for memory, with *CPU+Memory*, respectively . We also observe an average increase of 7% and 8.5% in CPU and memory utilization, respectively for *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).

There are two important points to note here. First, the benefits seen with MROrchestrator are higher for environments with *Multiple jobs*. This is due to the better isolation, dynamic allocation and global coordination provided by MROrchestrator. Second, we can observe that control of both CPU and memory (*CPU+Memory*) yields the maximum benefits.

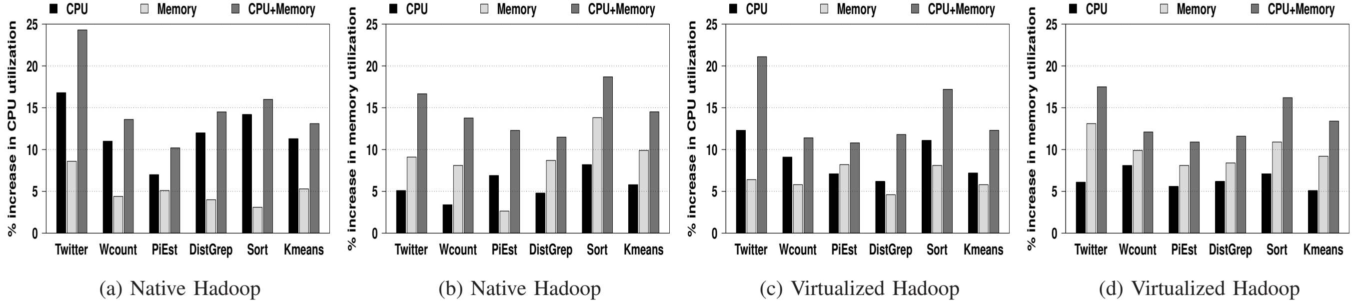
### B. Results for virtualized Hadoop cluster

Figure 8(a) and Figure 8(b) show the percentage reduction in job completions for *Single job* case with REGRESSION and UNIFORM schemes, respectively. We can notice a reduction



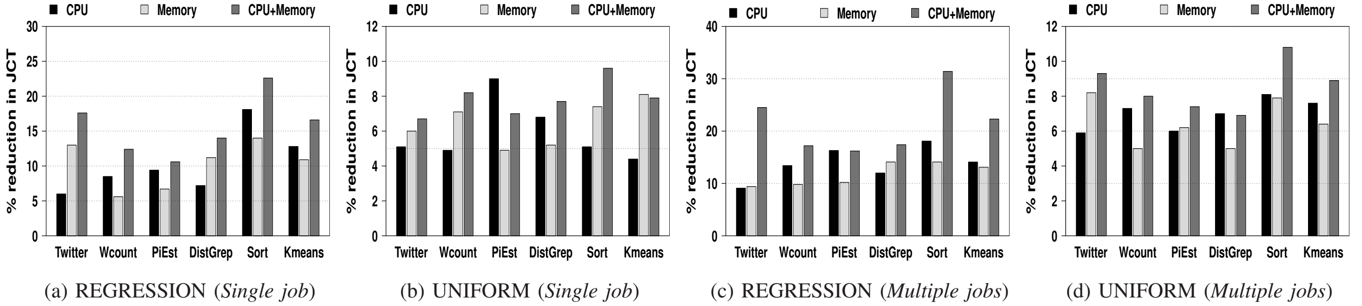

(a) REGRESSION (*Single job*) (b) UNIFORM (*Single job*) (c) REGRESSION (*Multiple jobs*) (d) UNIFORM (*Multiple jobs*)

Fig. 6: Reduction in Job Completion Time (JCT) for a *Single job* and *Multiple jobs* cases in native Hadoop cluster.



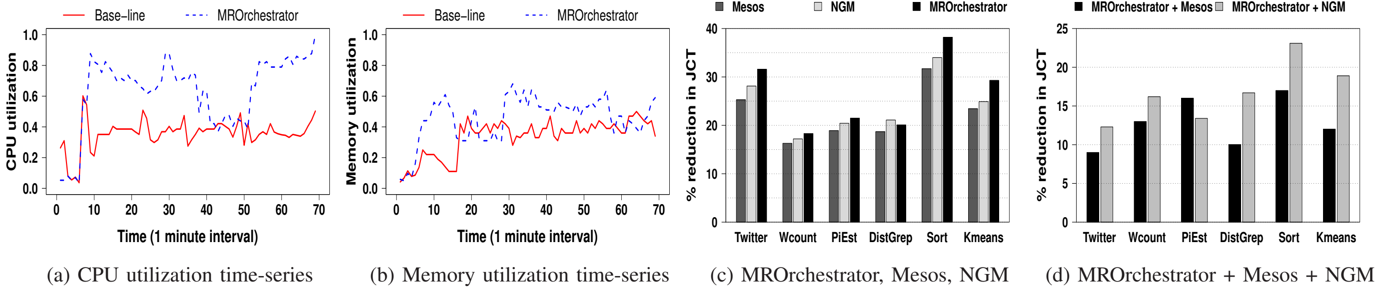

(a) Native Hadoop (b) Native Hadoop (c) Virtualized Hadoop (d) Virtualized Hadoop

Fig. 7: Improvement in CPU and memory utilization in native and virtualized Hadoop clusters with *Multiple jobs*.




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

Fig. 8: Reduction in Job Completion Time (JCT) for a *Single job* and *Multiple jobs* cases in virtualized Hadoop cluster.




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

Fig. 9: (a) and (b) demonstrate the dynamics of MROrchestrator. (c) shows the performance benefits with Mesos, NGM and MROrchestrator. (d) shows the comparison of MROrchestrator's integration with Mesos and NGM, respectively.

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 8(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 7(c) and Figure 7(d). We observe an increase in the utilization of 14.1% (average) and 21.1% (maximum) for CPU (Figure 7(c)); increase of 13.1% (average) and 17.5% (maximum) for memory (Figure 7(d)). These numbers correspond to REGRESSION scheme with *CPU+Memory* allocation mode of MROrchestrator. For UNIFORM scheme, the percentage increase in CPU and memory utilization is less than 10%.

To illustrate the dynamics of MROrchestrator, we plot Figure 9(a) and Figure 9(b), which show the snapshots of

CPU and memory utilization of the 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.

From the above analysis, we can observe that the magnitude of performance benefits obtained from the implementation of MROrchestrator on native Hadoop is more than the corresponding implementation on virtualized Hadoop. This might be related to the CPU, memory or I/O performance overheads associated with Xen virtualization [19]. 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, coupled with advancements in virtualization, the difference would shrink in the near future.

### C. MROrchestrator with Mesos and NGM

There are two contemporary resource scheduling managers – Mesos [16] and Next Generation MapReduce (NGM) [15] that provide better resource management in Hadoop. Mesos provides efficient resource isolation and sharing across distributed frameworks like Hadoop and MPI. It provides multi-resource (memory and CPU aware) scheduling with isolation between tasks with the use of Linux containers. Recently, Ghodsi et al. [11] proposed a multi-resource scheduling scheme, called Dominant Resource Fairness (DRF), that provides fair allocation of resources, and is implemented in Mesos. NGM is the new resource management architecture for Hadoop MapReduce, which was very recently proposed. It includes a generic resource model for efficient scheduling of cluster resources. NGM replaces the default fixed-size slot with another basic unit of resource allocation called *resource container*, which is more dynamic, as it can allocate resources in generic units like container X = (2 GB RAM, 1 CPU). We believe MROrchestrator framework is complementary to these systems, and thus, we integrate it with Mesos with DRF and NGM to observe prospective benefits.

For the integration of MROrchestrator with Mesos and NGM, we separately installed Mesos (with DRF) and NGM (which is supported in the latest Hadoop version, v0.23.0) on the 24-node native Hadoop cluster. We run the same workload suite of 6 MapReduce jobs for evaluations.

We first separately compare the performance of Mesos, NGM and MROrchestrator, *normalized* over the base case of default Hadoop with fair scheduling. Figure 9(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 observed with MROrchestrator is 17.5% and 8.4% more than the corresponding reduction seen with Mesos and NGM, respectively. We can observe that NGM has better performance than Mesos with DRF scheme. One possible reason is the replacement of slot with the resource container unit in NGM, which provides more flexibility and fine granularity in resource allocations.

We next demonstrate the benefits from the integration of MROrchestrator on top of Mesos (*MROrchestrator+Mesos*) and NGM (*MROrchestrator+NGM*), respectively. The results are shown in Figure 9(d). We can observe an average and a maximum reduction of 12.8% and 17% in job completions across all jobs, for *MROrchestrator+Mesos*. For *MROrchestrator+NGM*, the average and maximum decrease in job completions 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 in the *MROrchestrator + Mesos* case. 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 limit).

There are two main reasons for the observed better performance with MROrchestrator's integration. First, irrespective of the allocation units (a slot is replaced with *resource container* in NGM, or the same slot is used in Mesos), their static characteristics still persist. On the other hand, MROrchestrator dynamically controls and provides on-demand allocation of resources based on the run-time tasks resource profiles. Second, the inefficiency in resource allocation that arises due to the lack of global coordination (Section II-B) has yet not been addressed both in Mesos and NGM. MROrchestrator's architecture incorporates global coordination.

### D. Performance overheads of MROrchestrator

There are some important design choices that determine 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* (which is four times the default Hadoop heartbeat message frequency) was chosen as the epoch duration. Second, we discuss the expected delay in observing and resolving resource bottlenecks in MROrchestrator. It consists of four parts (overhead for each is with respect to the 20 seconds epoch duration): (i) the resource usage measurement data is collected and profiled every *5 seconds* (same as the interval of heartbeat messages). The associated time delay is negligible because of the use of light-weight, built-in Hadoop profiler. (ii) the time taken in detecting resource bottleneck by the Contention Detector module in GRM. It is of the order of 10s of milliseconds; (iii) the time taken to build the predictive models by the Estimator module in LRM. It is less than 1% and 10% for UNIFORM and REGRESSION schemes, respectively; and (iv) time taken to resolve contention by the Performance Builder in GRM is less than 4%.

## V. RELATED WORK

**Scheduling and resource management in Hadoop:** In the context of Hadoop, techniques for dynamic resource allocation

and isolation have been recently addressed. Polt et al. [20] proposed a task scheduler for Hadoop that performs dynamic resource adjustments to jobs, based on their estimated completion times. Qin et al. [21] used kernel-level virtualization techniques to reduce the resource contention among concurrent MapReduce jobs. Technique for assigning jobs to separate job-queues based on their resource demands was discussed in [23]. ARIA [24] resource manager for Hadoop estimates the amount of resources in terms of the number of map and reduce slots required to meet given SLOs. Another category of work has proposed different resource scheduling policies for Hadoop [13], [14], [22], [25]. The main focus of these schemes is to assign equal resource shares to jobs to maximize resource utilization and system throughput.

However, all the above solutions do not address the fundamental cause of performance bottlenecks in Hadoop, which is related to the static and fixed-size slot-level resource allocation. An important aspect where this paper differs from existing systems is that we try to estimate the requirements of each job (and its constituent tasks) at run-time rather than assuming that they are known a priori. The current Hadoop schedulers [13], [14] are not resource-aware of running tasks. We also advocate the need to resort to traditional scheduling in Hadoop, which is based on the generic resource concept instead of slot abstraction.

**Fine-grained resource management in Hadoop:** The closest works to ours are Mesos [16] and Next Generation MapReduce (NGM) [15]. Mesos is a resource scheduling manager [16] that provides fair share of resources across diverse cluster computing frameworks like Hadoop and MPI. Ghodsi et al. [11] recently proposed a Dominant Resource Fairness (DRF) scheduling algorithm to provide fair allocation of slots to jobs, and is implemented in Mesos. Next Generation MapReduce (NGM) [15] is the latest architecture of Hadoop MapReduce, that was very recently proposed. It includes a generic resource model for efficient scheduling of cluster resources. NGM replaces the default fixed-size slot with another basic unit of resource allocation called resource container. Condor [18] is another resource scheduling manager [18] that can potentially host Hadoop frameworks.

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

Efficient resource management is a critical but difficult task in a cloud environment. 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 suite, 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 achieve higher benefits in terms of reduction in job executions and increase in resource utilization. Specifically, we observed a reduction of up to 17% and 23.1%, in the completion times for the case of MROrchestrator's integration with Mesos and NGM, respectively. In terms of resource utilization, there was a corresponding increase of 11.7% (CPU), 8.2% (memory); and 19.2% (CPU), 13.1% (memory), respectively.

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 in a public cloud environment like Amazon EC2, with a larger experimental test-bed.

## REFERENCES

[1] Torque resource manager. http://tinyurl.com/torque-manager.
[2] Wikitrends. http://trendingtopics.org.
[3] Xen hypervisor. http://www.xen.org.
[4] Amazon. Aws. http://aws.amazon.com.
[5] Amazon. Mapreduce. http://aws.amazon.com/elasticmapreduce/.
[6] G. Ananthanarayanan and et al. Reining in the Outliers in MapReduce Clusters using Mantri. In *USENIX OSDI*, 2010.
[7] Apache. Hadoop. http://hadoop.apache.org.
[8] Apache. Hadoop profiler: Collecting cpu and memory usage for mapreduce tasks. https://issues.apache.org/jira/browse/MAPREDUCE-220.
[9] cgroups. Linux control groups. http://tinyurl.com/linuxcgroups.
[10] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *ACM Comm.*, 2008.
[11] A. Ghodsi and et al. Dominant resource fairness: fair allocation of multiple resource types. In NSDI, 2011.
[12] Gridmix2. http://tinyurl.com/gridmix2.
[13] Hadoop. Capacity scheduler. http://hadoop.apache.org/common/docs/r0.19.2/capacity_scheduler.html.
[14] Hadoop. Fair scheduler. http://hadoop.apache.org/common/docs/r0.20.2/fair_scheduler.html.
[15] Hadoop. Next generation mapreduce scheduler. http://goo.gl/GACMM.
[16] B. Hindman and et al. Mesos: A platform for fine-grained resource sharing in the data center. Technical Report UCB/EECS-2010-87, UC Berkeley.
[17] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.
[18] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for High Throughput Computing. *SPEEDUP Journal*, 1997.
[19] A. Menon and et al. Diagnosing performance overheads in the xen virtual machine environment. In *USENIX VEE*, 2005.
[20] J. Polo and et al. Performance-driven task co-scheduling for mapreduce environments. IEEE NOMS, 2010.
[21] A. Qin, D. Tu, C. Shu, and C. Gao. XConveryer: Guarantee Hadoop Throughput via Lightweight OS-Level Virtualization. GCC, 2009.
[22] T. Sandholm and K. Lai. Dynamic proportional share scheduling in hadoop. In *Job scheduling strategies for parallel processing*, 2010.
[23] Chao T., Haojie Z., Yongqiang H., and Li Z. A Dynamic MapReduce Scheduler for Heterogeneous Workloads. In *GCC*, 2009.
[24] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: Automatic Resource Inference and Allocation for Mapreduce environments. ICAC, 2011.
[25] M. Zaharia and et al. Job Scheduling for Multi-User MapReduce Clusters. Technical Report UCB/EECS-2009-55, UC Berkeley.
[26] Matei Zaharia and et al. Improving mapreduce performance in heterogeneous environments. USENIX OSDI, 2008.