KOALA-F: A Resource Manager for Scheduling Frameworks

Kuzmanovska, Aleksandra; Mak, Rudolf; Epema, Dick

DOI
10.1109/CCGrid.2016.60

Publication date
2016

Document Version
Accepted author manuscript

Published in
16th IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing (CCGrid)

Citation (APA)

Important note
To cite this publication, please use the final published version (if applicable). Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights. We will remove access to the work immediately and investigate your claim.
KOALA-F: A Resource Manager for Scheduling Frameworks in Clusters

Aleksandra Kuzmanovska, Rudolf H. Mak, Dick Epema
Dept. of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands
{a.kuzmanovska, r.h.mak, d.h.j.epema}@tue.nl

Abstract—Due to the diversity in the applications that run in clusters and datacenters, many different application frameworks have been developed, such as MapReduce for data-intensive batch jobs and Spark for interactive data analytics. A framework is first deployed in a cluster, and then starts executing a large set of jobs that are submitted over time. When multiple such frameworks with time-varying resource demands are present in a single cluster, static allocation of resources on a per-framework basis leads to low system utilization and resource fragmentation. In this paper, we present KOALA-F, a resource manager that dynamically provides resources to frameworks by employing a feedback loop to collect their possibly different performance metrics. Frameworks periodically—not necessarily with the same frequency—report the values of their performance metrics to KOALA-F, which then either rebalances their resources individually against the idle resource pool, or, when the latter is empty, rebalances their resources amongst them. We demonstrate the effectiveness of KOALA-F with experiments in a real system.

I. INTRODUCTION

Data-center environments such as clusters and clouds are generic computing platforms that can run multiple application frameworks simultaneously. While frameworks encapsulate the complexity of distributed computing and significantly simplify job management, there is no one-size-fits-all framework to accommodate all job types that are typically run in a single data-center environment. Therefore, consolidation of multiple frameworks is a common way to accommodate all job types and to achieve high system utilizations at the same time. In such consolidated environments, a major challenge is to share the physical resources among a diverse set of frameworks according to the variations in their workloads over time. In this paper, we address this challenge with the design, the implementation, and the analysis of a thin resource manager called KOALA-F for sharing resources across multiple frameworks in large clusters.

Many frameworks have been developed and are widely used for applications that share a common high-level programming model. Examples of such frameworks are MapReduce [3] and Dryad [9] for parallel data-intensive applications, Pregel [13] for large-scale graph processing, Spark [29] for iterative machine-learning applications and interactive analytics, and various component-based frameworks for specific application domains such as video processing [18]. There are some efforts to develop general-purpose frameworks that unify a wide variety of high-level programming models, e.g., Naiad [15], but sofar they have not become very popular. The different frameworks diverge in terms of the programming models they offer, the complexity and details of their deployment, the resource requirements of the applications they support, and, very important for this paper, the metrics that capture their performance.

Sharing resources across frameworks rather than across individual jobs relieves resource managers of large clusters of a potentially very high load of scheduling decisions, and of the complex logic needed to understand the requirements of various job types to obtain high-quality scheduling decisions. The resources of clusters can be shared by using either static or dynamic resource allocation. Static resource allocation with fixed numbers of resources assigned to frameworks over their lifetimes may lead to resource fragmentation and periods of over- and under-utilization. In contrast, dynamic resource allocation responds to the changing resource requirements of frameworks by changing the resource shares allotted to them according to the variations in their workloads over time. Therefore, the latter is the approach of choice for sharing resources among frameworks that process fluctuating workloads.

Resource management systems that use dynamic resource allocation mostly leverage fine-grained control over the resources they manage [7], [26], which requires very intensive communication between the cluster resource manager and the frameworks. As a result, these resource managers suffer from large overheads which is inappropriate for frameworks with low-latency requirements such as interactive query processing.

To overcome these two limitations, we design KOALA-F, a thin resource manager for sharing resources in large clusters by using the high-level abstraction of frameworks. KOALA-F is an extension of the KOALA multi-cluster scheduler [14] that we have previously designed for scheduling parallel applications, bags-of-tasks, workflows, etc. Whereas KOALA schedules each job separately, KOALA-F only schedules complete frameworks—frameworks are the “jobs” of KOALA-F—leaving the scheduling of the resources allocated to a framework to the internal framework scheduler. Frameworks periodically express their resource status, codified in an operational mode, to KOALA-F through a low-overhead feedback mechanism. Based on the operational modes of the concurrent frameworks and the idle cluster resources, KOALA-F recalculates the fraction of resources allocated to each framework and dynamically adapts them. With this level of abstraction, we employ multi-tenancy not at the level of jobs, or users, but at the level of frameworks.

To demonstrate the effectiveness of our thin resource manager, we have ported three frameworks to KOALA-F, one
for each targeted job type: MapReduce Hadoop [27] for batch-oriented data-intensive jobs, Spark [29] for interactive query processing, and FLUENT [2], [18] for service jobs that require immediate deployment upon arrival. Each job type requires its own performance metric that expresses the current performance level of the frameworks. To that end, we explore multiple performance metrics such as job slowdown, temporal job slowdown, response time, reject rate of applications, and utilization in the framework.

This paper makes the following contributions:

- We propose a provisioning mechanism for dynamic resource allocation across multiple frameworks based on their current performance levels (Section III).
- We provide a comprehensive set of experiments in a real multi-cluster system, and analyze the effect of the provisioning mechanism on the performance of the frameworks (Sections IV and V).

II. BACKGROUND

In this section, we introduce the foundation of our system, the KOALA resource manager, and the three frameworks that we have selected as case studies to highlight the features of KOALA-F and to assess its performance. These frameworks diverge in terms of the characteristics of the jobs they support and their performance goals. The selected frameworks all have their own local framework scheduler that schedules jobs on the resources that have been allocated to them by KOALA-F.

A. The KOALA Resource Manager

KOALA [14] is a multi-cluster resource manager that operates on physical infrastructures organized in clusters by interfacing to their local resource managers. The KOALA kernel is a centralized scheduler that has its own information service for finding out the status of all clusters to which it interfaces, and that schedules jobs by submitting them to suitable cluster sites according to its placement policies. The actual job submission to the allocated nodes is done by specialized interfaces called runners that provide the ability to submit and monitor jobs of different application types.

KOALA was originally designed to support co-allocation, i.e., the allocation of processors in multiple clusters to single parallel (MPI) applications [14]. It was later extended to other application types such as cycle scavenging jobs [22], workflows [23], and malleable applications [1]. Furthermore, we have previously extended KOALA for scheduling the Hadoop MapReduce [5] and FLUENT [11] frameworks. As a result, the unit of scheduling is complete frameworks that through KOALA and the local resource managers get allocated resource shares of clusters that can dynamically change over time based on various provisioning policies. The latest extension to KOALA is KOALA-C [4], which enables uniform job management across cluster and cloud environments by provisioning resources from both infrastructures.

B. Frameworks

Hadoop [27] is one of the first open-source implementations of Google’s MapReduce [3] programming model for large-scale data processing. It provides a distributed file system, HDFS [21], and an execution engine. HDFS splits up large files into fixed-size data chunks (of 64 or 128 MB) and replicates them on multiple compute nodes. In the execution engine, the highest-level unit of computation is a job, which is partitioned into multiple map and reduce tasks. A master node schedules tasks in FIFO order for parallel execution by assigning the tasks of jobs to containers close to the location of their input data chunks. Hadoop provides support for a variety of use cases. In this paper, we use Hadoop on a use case for which was originally developed, which is batch processing.

Spark [29] is an execution engine initially developed for iterative MapReduce jobs, such as machine learning algorithms in which a distributed data set is reused across multiple iterations during the job runtime. Today, it is a general purpose execution engine optimized to support a variety of use cases such as SQL and structured data processing, machine learning, graph processing, and stream processing. The highest-level unit of computation is an “application”, which may run a single batch job, a sequence of jobs, or an interactive session for running multiple jobs simultaneously. An application consists of a driver program and a set of executors. The driver divides jobs into tasks grouped in stages that depend on each other, and schedules these tasks in FIFO order for parallel execution in containers called executors. An executor runs multiple tasks concurrently or successively, which is enabled by in-memory data storage called resilient distributed data sets (RDD). This abstraction represents a fault-tolerant collection of objects partitioned across cluster nodes that can be rebuilt if a partition is lost, that can be explicitly cached in memory across machines, and that can be reused by multiple parallel jobs. In this paper, we use Spark to interactively analyze large data sets stripped across multiple nodes by running ad-hoc exploratory queries through the SparkSQL interface.

Fluent [2], [18] is a component-based framework for the run-time composition of video processing multimedia applications in the area of surveillance and transport logistics. The surveillance applications are computationally intensive with fluctuating resource requirements over their variable lifetimes, and they require strict admission deadlines. The highest-level unit of computation is a job, which is a batch of applications each possibly composed of multiple components. FLUENT performs scheduling at the level of applications in FIFO order without queuing—upon arrival of a job, it is immediately checked whether it can be admitted. A job can be completely accepted or rejected, but it can also be partially rejected if only several but not all of the applications in its batch fit on the available resources.

III. PROVISIONING MECHANISM

KOALA-F is an extension of the KOALA resource manager for dynamically scheduling heterogeneous frameworks. KOALA itself interfaces to the local resource managers of clusters or datacenters, from which it acquires a fixed set of nodes for a certain duration (potentially days or weeks) in order to schedule frameworks. Then it is KOALA-F that accepts new framework deployments, and uses the provisioning mechanism that we will explain in this section to distribute
those resources across the frameworks present. So the “jobs” scheduled by KOALA-F are actually frameworks that run the intra-framework jobs (henceforth simply called “jobs”) that are submitted directly to them through their own internal framework schedulers. KOALA-F treats the frameworks as black boxes and is agnostic to the intra-framework jobs.

A. The Idle Pool

In order to accommodate newly arriving frameworks, KOALA-F maintains an idle pool of spare resources. It admits a new framework to the system when this pool contains an amount of resources at least equal to the initial size, and rejects a framework otherwise. If admitted, a framework is initially deployed on an amount of resources equal to the initial size, which are removed from the idle pool. The initial size is a global system parameter set by a system administrator, which depends on the total amount of resources in the system and the expected number of concurrent frameworks. KOALA-F dynamically changes the amount of resources allocated to a framework by adding and removing resources, but it never decreases the amount of resources of a framework below the initial size. KOALA-F changes the amount of resources allocated to a framework based on the framework’s performance, and it employs a feedback loop to collect performance information from frameworks. A framework is removed from the system when it has been without jobs for a certain amount of time, which is also a system parameter. The resources of a removed framework are returned to the idle pool.

B. Operational Modes

At every point in time, frameworks operate in one of three operational modes, each identified by a color: green (G), yellow (Y), and red (R). The color G indicates that a framework performs in a satisfactory way with enough resources to accommodate more jobs, but may have too many resources. The color Y indicates that a framework still performs very well, but without resources for additional jobs. The color R indicates that a framework operates poorly and needs more resources. KOALA-F interprets the boundaries between the colors G and Y, and between the colors Y and R as a low and a high watermark, respectively, of a performance metric between which a framework is allowed to operate. Its goals are to operate all frameworks in the color Y and if possible, to keep spare resources for new frameworks. Since KOALA-F treats frameworks as black boxes, frameworks have to translate the values of their performance metrics into the corresponding operational modes. For example, for a framework for interactive processing that uses response time as a metric to express its performance, the watermarks can be set to 0.5s and 1s. Then, an average response time below 0.5s is translated to G, an average response time above 1s is translated to R, and every value between these two numbers is translated to Y.

The values of the watermarks of a framework may be set in two ways. First, the watermark values may be set to desired arbitrary values that KOALA-F should reach. Depending on the actual values and the workloads submitted to the frameworks, such a setting may stress KOALA-F to the point that it is impossible to keep all frameworks in the color Y. Secondly, the watermark values may be set based on the range of the performance metric extracted from an operational framework, or based on the experience with and knowledge of the type of framework. In this paper, we employ the second approach. Since we do not have operational frameworks ourselves, we perform calibration experiments in which we run the target frameworks in isolation under various loads with fixed resource allocations. Then, we inspect the performance metrics to arrive at reasonable watermark values for each of the frameworks. This method can also be used in production systems by first monitoring the execution of a framework for some period of time under the expected loads before setting the watermarks.

C. Reporting

In order to dynamically manage the amounts of resources allocated to the frameworks currently present in the system, they have to report periodically the colors of their current operational modes to KOALA-F. Since different frameworks may operate on different time scales, they can have different window lengths over which they compute their performance metrics, which is called the reporting interval length. In addition, we let each framework have its own reporting frequency, which is the number of reports per unit of time. The reporting interval length and the reporting frequency of a framework do not have to correspond, the reporting frequency may be higher (but not lower) than indicated by the reporting interval length. That is, a framework may for instance report every 3 minutes its performance over the last 20 minutes, thus having a large overlap between successive reporting intervals. The reason for doing so is to give KOALA-F a more stable picture of the performance of the frameworks.

D. Asynchronous and Synchronous Mode

KOALA-F supports both a synchronous and an asynchronous mode for making decisions about the resource (de-)allocation to frameworks, and it switches between them based on the status of the idle pool. When this pool is not empty, it operates in the asynchronous mode, and otherwise in the synchronous mode. In both modes, the frameworks report their colors in a non-synchronized fashion according to their own reporting interval length and reporting frequency. In this paper we assume that the unit A of resource (de-)allocation to a framework in terms of a number of nodes is always equal to a fixed fraction of the nodes of the cluster that is set in proportion to the cluster size (e.g., 5% of the cluster size). The two modes of operation of the mechanism employed by KOALA-F for dynamically provisioning resources to the frameworks currently present in the system, which we will explain in detail below, are illustrated in Figure 1.

In the asynchronous mode, KOALA-F makes (re-)allocation decisions for each framework separately, in response to the reporting by a framework of its color. It adds A nodes to a framework that is in color R from the idle pool, and removes A nodes from a framework in color G, which are then returned to the idle pool. KOALA-F does not make any changes in the
resource allocation of a framework with color Y. The amount of resources allocated to a framework over time is determined by its reporting frequency. In this way, we can control how reactive KOALA-F is to a framework (e.g., reporting every 5 min on a reporting interval of 20 min allows a maximum of 4 changes per reporting interval).

In the synchronous mode, KOALA-F synchronizes its allocation decision to one of the frameworks with the highest reporting frequency. It makes allocation decisions for all frameworks at once, taking into consideration the last reported colors of the frameworks. The lack of spare resources indicates that the cluster is partitioned among frameworks, and the only thing KOALA-F can do is try to balance the resources among the frameworks. When all frameworks are in color R, or they are all in color Y, there is nothing KOALA-F can do. When both colors Y and R occur, KOALA-F selects as many pairs of frameworks in colors Y and R, respectively, and moves A nodes from the former to the latter, in the hope that the frameworks in color Y remain in Y, and the frameworks in color R shift to Y. The appearance of frameworks in color G means switching to the asynchronous mode.

IV. EXPERIMENTAL SETUP

In this section, we describe the infrastructure, the configurations of the selected frameworks, and for each framework, the applications, workloads and metrics used in our experimental evaluation of the provisioning mechanism of KOALA-F.

A. System Configuration

We evaluate KOALA-F on the Dutch DAS-4 multi-cluster system [17], which consists of six clusters and comprises roughly 200 dual-quad-core compute nodes with 24 GB memory and a local storage of 2 TB per node. The clusters have 1 Gbit/s Ethernet and 20 Gbps QDR InfiniBand networks; the wide-area network between the clusters is based on dedicated 10 Gbit/s light paths. The Sun Grid Engine (SGE), to which KOALA-F interfaces, operates as the local resource manager on each of the six clusters.

We perform experiments in a cluster with 30 nodes, where we can run a maximum of three concurrent frameworks with the unit A of resource (de-)allocation fixed to a single node, and an initial framework size five nodes. Currently, KOALA-F supports the following versions of the three frameworks described in Section II: Hadoop-1.2.1, a standalone mode of Spark-1.3.0, and Fluent-0.9.0. In all three frameworks, a local framework scheduler schedules jobs in FIFO order. After its deployment on the initial nodes, we do not resize or reconfigure the HDFS in order to prevent the reconfiguration overhead due to data re-balancing. KOALA-F provisions only transient nodes to Hadoop framework instances, which may execute tasks and store intermediate data locally, but do not store any part of the input data sets. HDFS is configured to store data in blocks of 128 MB with a default replication factor 3. With 8 cores per node, without hyper-threading, we configure Hadoop with 6 map and 2 reduce slots, and Spark workers with 8 slots. Speculative task execution is turned off.

B. Applications

The use cases we consider for the three frameworks are large-scale batch processing for Hadoop, interactive analysis of data for Spark, and surveillance applications that require strict admission deadlines for FLUENT. We select applications for these use cases from the latest version of the MapReduce benchmark suite, HiBench [8], the AMPLab benchmark for big data processing [16], and video processing applications from our previous work [11].

1) Hadoop Applications: We use a mix of three applications as representatives of typical MapReduce applications for batch processing, namely Wordcount (WC), Sort (ST), and PageRank (PR). In order to provide jobs with different durations in our workloads, we generate data sets of two sizes for each application. WC is a compute-intensive application that extracts the number of occurrences of words from a data set, and ST is an I/O-intensive application that sorts a given data set. For both applications, we generate input data sets of 25GB and 100GB. PR is an open-source implementation of the page-rank algorithm that consists of two consecutive I/O-intensive jobs. We generate input data sets of 20M and 50M pages in which the numbers of hyper-links follow a Zipf distribution.

![Fig. 2. The 95th percentiles of the run-times (left) and the service-times (right) of the Hadoop applications across ten runs in an empty 10-node framework. Besides the application, the names indicate the size of the input data set in GB for WC and ST, and in number of pages for PR.](image-url)
In order to assess the performance of the applications, we define their run-time and service-time, which differ from each other by the level of parallelism employed. The run-time of an application is defined as the amount of time between the start of its first task and the completion of its last task, and the service-time of an application is defined as the total execution time of all of its (map and reduce) tasks together. The service-time of an application is in principle independent of the level of parallelism in the application and the circumstances in which it is executed, and therefore defines its total processing requirements. Figure 2 shows the 95th percentiles of the run-times and service-times across ten runs in an empty 10-node framework with a 5-node HDFS. We categorize the applications as either short, medium or long, based on their run-time, which is directly dictated by the input data size and the application type.

2) Spark Application: We use a single application as an interactive session for running multiple jobs simultaneously, where a job corresponds to a single query. Queries are executed in two stages: in the first, 200 tasks analyze partitions of a distributed data set in parallel, and in the second, a single task performs aggregation of the results. We use three variants of the first query from the AMPLab big data benchmark suite, a simple `select`, which represents common query types executed in popular SQL query engines in big data systems. This queries analyzes $\sim 6GB$ of page ranks per website, stored on a local disk. The data are cached in memory, and all queries read/write data directly from/to memory. The query scans the entire input data set, and filters out the subset of it based on the input parameter which dictates the size of the query output. The service-time of the query is defined as the total execution time of all of its tasks. The average query service-time across five runs is 16.6 s.

3) Fluent Application: In FLUENT, we use a synthetic application, `RemoteLaplace` (RL), from the video processing domain that performs image sharpening by applying a Laplacian filter on the input video stream which has been smoothed to remove noise. The RL application operates on a video at a frame rate of 60 fps and with a frame resolution of 320x240 pixels. The application has a variable execution time which is drawn uniformly from the interval from 5 to 10 minutes.

C. Workloads

We create three categories of synthetic workloads identified by the name of their target framework. The Hadoop Workload ($W_h$) consists of jobs of various types and run-times, each running one of the six MapReduce applications. Similar to the workloads used by others [28], in which most jobs are small, in $W_h$, the ratios between jobs from the categories short, medium and long are 6:3:1. The Spark Workload ($W_s$) consists of jobs, each executing one of the three pre-defined query variants selected with equal probability. The data set is loaded and cached in memory before the first query is submitted.

The Fluent Workload ($W_f$) consists of jobs, each running a batch of RL applications, to reflect the operation of multiple surveillance video cameras started simultaneously, e.g., for monitoring a shopping mall or a parking garage. The batch size is drawn from a geometric distribution on the interval [1, 5] with mean 1.4. Within a batch, all applications have equal execution times. Due to a technical limitation of FLUENT, a single application takes a complete node of 8 cores. So we can view a single application to represent 8 RL applications, and in fact, the jobs can be seen as representing 8–40 applications, for a mean of 11.2.

In all our workloads, jobs arrive according to a Poisson process. We fix the duration of the generated workloads, and change the job arrival rate to generate workloads that impose different loads. In order to show how KOALA-F handles unexpected load spikes in frameworks, in the workloads of some of the experiments we create bursts by increasing the arrival rate by some factor over a period of time.

D. Metrics

Assessing the performance of frameworks is a challenging task, given that it is not always clear which metrics should be used in the face of various job types submitted to the frameworks. To that end, we start with commonly used metrics for each framework, job slowdown, response time, and reject rate, to assess the performance of Hadoop, Spark and FLUENT, respectively. However, based on initial experimental results in asynchronous mode, we conclude that some of the selected metrics are not appropriate in the context of our provisioning mechanism (see Section V-B). Therefore, we also explore the metrics temporal job slowdown for Hadoop and utilization for Fluent. In our provisioning mechanism, a framework reports the average value of a metric calculated on a reporting interval.

Job slowdown (JSD) is defined as the ratio between the elapsed time (from the moment of submission until the completion) of a job in the system with contention and the job run-time on an empty reference system. Therefore, JSD can only be determined, and be included in the average value reported to KOALA-F, once a job is finished, which has two drawbacks. First, the number of values per reporting interval is generally small, which may lead to unstable values reported by the framework to KOALA-F. Secondly, the reported values may fail to capture the instantaneous system performance as they do not sufficiently reflect the execution of long jobs due to the late calculation of their JSD.

To counter these problems of JSD, we modify its definition to temporal job slowdown (T-JSD), defined at any point in time during the execution of a job as the ratio between the time from the moment of submission until that point, and its (complete) run-time on an empty reference system. Over a job’s lifetime, its T-JSD increases to its JSD, and contributes to multiple values reported to KOALA-F. Similarly as in Section IV-B, the empty reference system we use for the calculation of both JSD and T-JSD is a 10-node system with a 5-node HDFS.

The Response time (RT) of a job is defined as the time from its submission until the completion of its last task. The Reject rate (RR) is defined as the percentage of all applications across all jobs that are rejected (a job can be partially rejected if there is only room for some but not all of its applications). For FLUENT, we will switch to another metric, the utilization (U), defined as the ratio of the actually used and the total number of allocated resources.
V. PERFORMANCE EVALUATION

In this section, we present a performance evaluation of KOALA-F. First, we perform for each of the three frameworks a calibration experiment to determine the values of the watermarks of the performance metrics, of the reporting intervals, and of the reporting frequencies used in our provisioning mechanism. Then, we analyze the effect of the provisioning mechanism on the performance of the frameworks. We show how KOALA-F manages the idle pool and reacts to bursts in the load of the frameworks (asynchronous mode), and how KOALA-F balances resources among frameworks when a cluster is overloaded (synchronous mode).

A. Calibration

To determine the values of the watermarks of the performance metrics used in the provisioning mechanism of KOALA-F, we consider the distributions of these metrics under various loads over a three-hour period. We statically deploy the frameworks on 10 nodes and we submit workloads without bursts that impose loads of 0.5, 0.7 and 0.9. Table I summarizes the workload characteristics for each framework under an imposed load of 0.7.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Arrival rate</th>
<th>No. of jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>W_h</td>
<td>0.0031</td>
<td>33</td>
</tr>
<tr>
<td>W_s</td>
<td>3.3800</td>
<td>36,106</td>
</tr>
<tr>
<td>W_f</td>
<td>0.0110</td>
<td>121</td>
</tr>
</tbody>
</table>

Rather than considering the distributions of the performance metrics on a per-job basis, in this section we consider the distributions of the metrics on a per-reporting-interval basis by taking their averages in each reporting interval during the experiment. As a matter of fact, this is how the metrics are reported to KOALA-F. By abuse of terminology, we use the performance metric to refer to its average calculated on a reporting interval, e.g., JSD refers to the average value of JSD of all jobs that finished in such an interval. Based on extensive experimentation with different values, we fix the reporting interval to 20, 1 and 3 minutes, and the reporting frequency to once every 5, 1, and 3 minutes for Hadoop, Spark and FLUENT, respectively.

Figures 3 and 4 show the results of the calibration experiments, where the whiskers depict the 5th and 95th percentiles, the box depicts the 25th and 75th percentiles, the bold line in the box indicates the median, and the red dot indicates the mean of a performance metric.

Figure 3 shows that JSD and T-JSD have similar distributions. We exclude the results under a load of 0.9 from our decision how to set the watermarks of these metrics, because the cluster cannot deal with an imposed load of 0.9, and a rather large backlog of jobs still remained at the end of our experiment. Only 60% of the jobs were actually completed during the experiment, and Hadoop needed an additional hour to complete these jobs. Since the median of JSD at a load of 0.5 is slightly above 1, we fix the low watermark value of JSD to 1. We fix the high watermark value of JSD to 3, which is equal to the third quartile of JSD at a load of 0.7, and includes all values of JSD at a load of 0.5. Given that the distributions of T-JSD are similar to the distributions of JSD, we apply the same reasoning to select watermark values for T-JSD. Therefore, we set the low and high watermarks to 1 and 2.5, respectively.

Figure 4 depicts the distributions of RT, which is used as the metric for Spark, and of RR and U, the two metrics for FLUENT. As to Spark at a load of 0.5, RT is evenly spread between 0.6s and 0.8s with a median of 0.7s, which indicates that the values below 0.6s may be reached in a relatively empty framework. Therefore, we use the 5th percentile, 0.6s, as the low watermark of RT. We set the high watermark value of RT to 0.8s in order to satisfy an appropriate notion of interactive queries, thus including 75% of the distribution at a load of 0.7.

As to FLUENT, the distributions of RR at loads of 0.5 and 0.7 show that 50% of the reporting intervals have no rejects at all, but that the remainder of the reporting intervals have a relatively high RR. At a load of 0.9, the median RR is around 6%, which looks as an appropriate value for the low watermark of RR. To find a proper value for the high watermark, we take the 95th percentile at a load of 0.7, which is about 15%. As to the watermarks of U, we might set the low watermark of U to 50% based on the median value under a load of 0.5.
However, we set it to 40% based on our previous experience with this metric [11]. As the high watermark value, we use the median of U under the load of 0.7, which is 70%. Of course, increasing the high watermark value will improve the utilization of the system, but will also increase the overall RR.

Table II summarizes the selected values for the watermarks of the performance metrics used in KOALA-F. We use these values in the rest of the experiments.

<table>
<thead>
<tr>
<th>Watermark</th>
<th>JSD</th>
<th>T-JSD</th>
<th>RT[sec]</th>
<th>RR[%]</th>
<th>U[%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>1</td>
<td>1</td>
<td>0.6</td>
<td>6</td>
<td>40</td>
</tr>
<tr>
<td>High</td>
<td>3</td>
<td>2.5</td>
<td>0.8</td>
<td>15</td>
<td>70</td>
</tr>
</tbody>
</table>

B. Asynchronous Mode

In this section, we assess the behavior of KOALA-F in the asynchronous mode in which the idle pool is not empty and there is still space to grow a framework without having to consider other frameworks, or to accommodate new frameworks. In this mode, KOALA-F handles all frameworks separately based on their reported colors. We configure KOALA-F to operate a 20-node cluster, and to serve a single framework at a time with an initial size of 5 nodes. To show how KOALA-F reacts to unexpected load in frameworks, we use workloads that would impose a load of 0.6 on a cluster of 10 nodes, and we add an additional burst period of 20 minutes in the middle of the workload.

Figure 5 depicts the performance of Hadoop over time when KOALA-F is used with (a) JSD and (b) T-JSD as metrics, and when (c) 10 nodes are statically allocated. Also in the latter case we report the color the framework operates in over time, even though these colors are not reported or used. We exclude the burst period from this workload because it increases the load above 100%, and our experience with a load of 0.9 (calibration experiment) shows that Hadoop performs poorly at high loads.

With both JSD and T-JSD, the number of nodes allocated to the framework varies between 5 (initially) and the maximum of 20 nodes, and the overall utilization of the resources allocated to the framework is 75% and 65%, respectively. The blue areas above the solid black lines indicate the amount of work Hadoop redoes as a result of lost data due to the killed tasks when a node is decommissioned. The decommissioning process lasts between 8 s and 10 minutes, and depends on the size of the intermediate data stored on a node. The amount of redone work for both JSD and T-JSD is only 4%.

Figure 5a shows that Hadoop operates in the color R for a long time, and that it rarely operates in the color Y but almost exclusively oscillates between the unstable colors G and R. Due to the late calculation of JSD once a job is finished, Hadoop operates in the color G for a long period towards the beginning of the experiment with too few nodes for the submitted workload. As a result, when jobs finish with large slowdowns, Hadoop enters the color R and hardly leaves it anymore. Since JSD fails to capture the immediate system performance, it is very common to have periods without finished jobs while the contention in the system is pretty high. As an example, between 100 and 105 min the reported color is G, but the blue area indicates that the load in the system is very high. As a result, KOALA-F removes a node from Hadoop, when it actually needs to add one. With T-JSD, Hadoop enters the unstable state and asks for resources much sooner than with JSD. Furthermore, all transitions between R and G are buffered with the color Y due to the incremental update KOALA-F has on the progress of jobs in the system.

With JSD as the metric, the average JSD over all jobs in the framework is 5.1, which is way above the high watermark. In contrast, when T-JSD is used as a metric, the average JSD is 2.83, which is close to the target value of the high watermark. When Hadoop is statically deployed on 10 nodes with only 5 data nodes, the overall JSD in the framework is 5.54, again much higher than the high watermark. Besides the advantage that we do not need to know the number of nodes in advance, our mechanism outperforms the static deployment by reducing JSD of jobs almost by a factor of 2.

Figure 6 depicts the performance of FLUENT over time when KOALA-F is used with (a) RR and (b) U as metrics, and when (c) 10 nodes are statically allocated. The burst period starts 90 min and ends 110 min after the start of the workload, during which the arrival rate is multiplied by a factor of 3. With RR as a metric, the number of nodes allocated to the framework varies between 5 (initially) and a maximum of 13...
nodes, and the overall utilization of the resources allocated to the framework is 68%. There are two extended periods when the framework operates in the color R, soon after the start of the experiment, which is expected as the framework starts with too few nodes for the submitted workload, and around the burst period. In both cases, KOALA-F nicely reacts by allocating more nodes. There are two problems with the operation of FLUENT with RR as the metric. First, it rarely operates in the color Y, but almost exclusively oscillates between the unstable colors G and R. Secondly, the overall RR is 32%, which is unacceptable as we have fixed the watermarks at 6 and 15%. We have performed the experiment for different values of reporting interval, reporting frequency, and watermarks, but after extensive experimentation, we could not get it right for any set of parameter values. Therefore, we conclude that RR is not an appropriate metric for controlling the amount of resources needed in FLUENT.

Unfortunately, we could not make sense of the results from the experiment with Spark and RT as a metric, due to the technical limitation of the framework. Spark, by nature, supports a fixed number of executors per application, and it is not possible to change their number during its lifetime. This functionality is crucial in our use case since we have a single application that serves as an interactive shell to submit exploratory queries. According to the documentation, this functionality is introduced to the standalone mode in the last version of Spark, but we have experienced a strange increase of a query response time in case of adding additional executors to the application. We have tried many different versions and parameter combinations in order to make it work, but without much success.

C. Synchronous Mode

Finally, we will assess the behavior of KOALA-F in synchronous mode in which the idle pool is empty and there is no space to accommodate new frameworks. In this mode, KOALA-F balances the resources among the frameworks based on colors reported by all frameworks. In these experiments, KOALA-F is configured to operate a 20-node cluster, and serves two instances of the same framework, each with an initial size of 5 nodes, and workloads without bursts. For each pair of frameworks, we report the total amount of allocated resources with a solid black line, the total amount of used resources with a solid blue area, and the operational modes reported by frameworks to KOALA-F with a colored bar at the bottom and at the top of each plot.

When U is used as a metric to capture the performance of FLUENT, the overall utilization of the framework is 55%, which is right in the middle of the corresponding watermarks, and the number of nodes allocated to the framework varies between 5 and 17. The curve of allocated resources has largely the same shape as with RR, but is at a higher level. The two problems identified with RR are to a large extent solved. First, the framework operates a decent fraction of the time in the stable color Y (the two largest red periods are due to the small initial allocation and to the burst). Secondly, the overall RR is 18%, which is only slightly above the high watermark.

When FLUENT is statically deployed on 10 nodes, the overall utilization is 58%. The pattern of operational colors is roughly the same at with KOALA-F and the metric U, but here FLUENT does not start with a number of nodes that is too small, and the burst is not handled as well. The overall RR in this case is 24%, which is much higher than the high watermark. We conclude that static deployment may give results as with KOALA-F and U as the metric when the workload is pretty stable, but that KOALA-F outperforms it when it comes to unpredictable loads. Of course, the big drawback of static allocation is that the number of nodes to be allocated has to be known in advance.

When F is statically deployed on 10 nodes, the overall utilization is 58%. The pattern of operational colors is roughly the same at with KOALA-F and the metric U, but here FLUENT does not start with a number of nodes that is too small, and the burst is not handled as well. The overall RR in this case is 24%, which is much higher than the high watermark. We conclude that static deployment may give results as with KOALA-F and U as the metric when the workload is pretty stable, but that KOALA-F outperforms it when it comes to unpredictable loads. Of course, the big drawback of static allocation is that the number of nodes to be allocated has to be known in advance.

When U is used as a metric to capture the performance of FLUENT, the overall utilization of the framework is 55%, which is right in the middle of the corresponding watermarks, and the number of nodes allocated to the framework varies between 5 and 17. The curve of allocated resources has largely the same shape as with RR, but is at a higher level. The two problems identified with RR are to a large extent solved. First, the framework operates a decent fraction of the time in the stable color Y (the two largest red periods are due to the small initial allocation and to the burst). Secondly, the overall RR is 18%, which is only slightly above the high watermark.

When FLUENT is statically deployed on 10 nodes, the overall utilization is 58%. The pattern of operational colors is roughly the same at with KOALA-F and the metric U, but here FLUENT does not start with a number of nodes that is too small, and the burst is not handled as well. The overall RR in this case is 24%, which is much higher than the high watermark. We conclude that static deployment may give results as with KOALA-F and U as the metric when the workload is pretty stable, but that KOALA-F outperforms it when it comes to unpredictable loads. Of course, the big drawback of static allocation is that the number of nodes to be allocated has to be known in advance.

Unfortunately, we could not make sense of the results from the experiment with Spark and RT as a metric, due to the technical limitation of the framework. Spark, by nature, supports a fixed number of executors per application, and it is not possible to change their number during its lifetime. This functionality is crucial in our use case since we have a single application that serves as an interactive shell to submit exploratory queries. According to the documentation, this functionality is introduced to the standalone mode in the last version of Spark, but we have experienced a strange increase of a query response time in case of adding additional executors to the application. We have tried many different versions and parameter combinations in order to make it work, but without much success.

When U is used as a metric to capture the performance of FLUENT, the overall utilization of the framework is 55%, which is right in the middle of the corresponding watermarks, and the number of nodes allocated to the framework varies between 5 and 17. The curve of allocated resources has largely the same shape as with RR, but is at a higher level. The two problems identified with RR are to a large extent solved. First, the framework operates a decent fraction of the time in the stable color Y (the two largest red periods are due to the small initial allocation and to the burst). Secondly, the overall RR is 18%, which is only slightly above the high watermark.

When FLUENT is statically deployed on 10 nodes, the overall utilization is 58%. The pattern of operational colors is roughly the same at with KOALA-F and the metric U, but here FLUENT does not start with a number of nodes that is too small, and the burst is not handled as well. The overall RR in this case is 24%, which is much higher than the high watermark. We conclude that static deployment may give results as with KOALA-F and U as the metric when the workload is pretty stable, but that KOALA-F outperforms it when it comes to unpredictable loads. Of course, the big drawback of static allocation is that the number of nodes to be allocated has to be known in advance.

Unfortunately, we could not make sense of the results from the experiment with Spark and RT as a metric, due to the technical limitation of the framework. Spark, by nature, supports a fixed number of executors per application, and it is not possible to change their number during its lifetime. This functionality is crucial in our use case since we have a single application that serves as an interactive shell to submit exploratory queries. According to the documentation, this functionality is introduced to the standalone mode in the last version of Spark, but we have experienced a strange increase of a query response time in case of adding additional executors to the application. We have tried many different versions and parameter combinations in order to make it work, but without much success.

C. Synchronous Mode

Finally, we will assess the behavior of KOALA-F in synchronous mode in which the idle pool is empty and there is no space to accommodate new frameworks. In this mode, KOALA-F balances the resources among the frameworks based on colors reported by all frameworks. In these experiments, KOALA-F is configured to operate a 20-node cluster, and serves two instances of the same framework, each with an initial size of 5 nodes, and workloads without bursts. For each pair of frameworks, we report the total amount of allocated resources with a solid black line, the total amount of used resources with a solid blue area, and the operational modes reported by frameworks to KOALA-F with a colored bar at the bottom and at the top of each plot.

In Figure 7, we show the performance of two instances of (a) FLUENT and (b) Hadoop, and their operational modes over a three-hour period. In the case of FLUENT, we have to overload the system in order to prevent KOALA-F from frequently switching back to asynchronous mode due to the immediate rejection of applications. Therefore, we submit a workload that imposes a load of 0.4 to the first instance, F-1, and a workload that imposes a load of 0.8 to the second instance, F-2. Both loads are calculated relative to a 20-node cluster. At the beginning of the experiments KOALA-F operates in the asynchronous mode since the initial size of both instances F-1 and F-2 is 5, and idle pool is not yet empty. After 20 minutes of operation, KOALA-F switches to synchronous mode and starts to balance resources between the two instances. F-2 operates
in color R most of the time, which is expected on the basis of the imposed load, and periodically switches to color Y by exploiting additional nodes deallocated from F-1. When F-1 reaches color R, and F-2 operates in Y, the nodes are returned back to the F-1. In both instances, the overall RR is pretty high, 31% for F-1 and 43% for F-2, which is expected because the system is overloaded. We evaluate the performance in this mode by looking at the proportion of allocated nodes. The ratio between the allocated resources over time is 2, same as the ratio of the imposed loads.

In the case of Hadoop, we go with smaller load due to the inability of Hadoop to deal with high loads. Here, we submit a workload that imposes a load of 0.2 to the first instance, F-1, and a workload that imposes a load of 0.5 to the second instance, F-2, relative to a 20-node cluster. We initialize both instances with 10 nodes in order to force KOALA-F to work in synchronous mode right from the start. F-1 operates in color Y right from the beginning due to the small imposed load, and as a result it releases nodes in favor of F-2. At the end of the experiment, we have the opposite situation where F-2 operates in color R most of the time, which is expected because the system is overloaded. We evaluate the performance in this mode by looking at the proportion of allocated nodes. The ratio between the allocated resources over time is 2.5.

VI. RELATED WORK

From the current state of the art resource management systems, Mesos [7], YARN [26] and Fawkes [6] are most similar to ours due to the common design of multiplexing cluster resources among various frameworks by means of a two-level scheduling architecture. Mesos shares cluster resources across various frameworks by following a fine-grained resource allocation model employed in existing frameworks, and delegating scheduling control to the frameworks.

A centralized global resource manager allocates resources to each framework by periodically initiating resource offers that frameworks may either accept or reject based on their own policies. Each framework separately schedules its jobs on the accepted resources by sending information about the jobs that should be execute to the global Mesos resource manager, who is responsible for job execution. Mesos targets frameworks with a fine-grained resource sharing model and preferably short jobs where efficient data sharing is a primary goal. It is not clear how it performs in the context of greedy frameworks with longer jobs. YARN explicitly supports multi-frameworks deployment, but it interfaces to per-job application managers rather than to framework schedulers as KOALA and Mesos do, and so in fact, it still provides a monolithic scheduling mechanism. However, as opposed to Mesos, YARN has a request-based centralized scheduler where application managers control the resource shares they get and the application’s execution on the allocated resources. In this setup, YARN responds to the requests based on global guarantees such as fairness or capacity. Fawkes is a centralized resource management system that balances fair resource shares among multiple MapReduce frameworks based on demand, usage, or performance. It balances resources by closely observing submitted jobs to each framework and dynamically changing the weights of the frameworks. As opposed to Mesos and YARN which target near-optimal data locality, Fawkes achieves better performance by relaxing the strict data locality assumptions. However, none of the foregoing resource managers provides support for scheduling heterogeneous frameworks with mechanisms for imposing balanced resource shares of the frameworks and without scheduling overhead itself all jobs submitted to all frameworks.

Another class of cluster management systems follows a distributed parallel architecture. Google’s Omega [12], a leader in this category, follows an alternative decentralized approach of shared-state scheduling by having the schedulers of multiple parallel frameworks compete for the resources of the complete cluster without a central authority. Omega employs lock-free optimistic concurrency control to mediate between conflicting allocation decisions of the separate framework schedulers at the potential cost of redoing work when optimistic concurrency assumptions are incorrect. Omega cannot enforce any global cluster guarantees such as fairness, but it is more a QoS-driven scheduler where a service level agreement is in form of the maximum amount of resources a framework may request. Furthermore, it may not perform well in heavily loaded clusters due to the increased conflict fraction. A distributed architecture is also followed the fine-grained task scheduler Sparrow [19]. Targeted jobs are composed of short tasks and require sub-second response time. This scheduler does not belong to the category of cluster resource managers since the provided functionality is task scheduling under the assumption that framework’s processes are already running. It can be used as high-level task scheduler on top of multiple data analytical frameworks that share a common performance metric (e.g., sub-second response time) which is opposite to the functionality provided by KOALA where the goal is to equally satisfy a mix of frameworks with potentially different performance
metrics.

Prior to these systems, state of the art resource management frameworks mainly fall into the category of high performance computing schedulers mostly optimize for large parallel jobs. Torque [24], Condor [20], [25], and Quincy [10] address non-elastic jobs that require static resource allocations during their execution and require users to declare the desired resources in advance. These schedulers typically deploy jobs onto dedicated, statically-partitioned clusters of machines, which leads to fragmentation and under-utilization of resources. In contrast, KOALA archives a high resource utilization by using dynamic resource allocation.

VII. CONCLUSION

In this paper we presented the design, implementation (as part of the KOALA multi-cluster scheduler), and analysis of a dynamic resource provisioning mechanism for multiple frameworks in datacenters based on the values of the performance metrics they periodically report. We applied this mechanism to the Hadoop, Spark, and FLUENT frameworks that have very different characteristics. We showed that our KOALA-F system can indeed result in dynamic allocations to these frameworks that lead to much better performance than static allocations, both when a datacenter is overloaded and when it is not. The added benefit of the approach is that the static allocations to each framework do not have to be guessed.

However, for each of the three frameworks we considered, a large amount of tuning and experimentation had to be done to arrive at proper values for the intervals and frequencies with which the frameworks report their performance metrics to KOALA-F, and to arrive at proper metrics to be used in the first place. As an additional complexity, we found that for Hadoop, the service times of the jobs varied significantly across different clusters and utilizations, which partly defied the operation of our mechanism. We consider this to be more of an issue of the framework than of our provisioning mechanism. The increase of service times in Hadoop, and the lack of application elasticity in Spark show that we cannot treat frameworks as black boxed, but in contrast, we need to have a good understanding of the internal operation of a framework in order to design a meaningful provisioning mechanism.

As future work, we will compare our provisioning mechanism with more fine-grained allocation mechanisms, and we will perform a more comprehensive simulation study.

REFERENCES