Dynamically Scheduling a Component-Based Framework in Clusters

Aleksandra Kuzmanovska, Rudolf H. Mak, and Dick Epema

Dept. of Mathematics and Computer Science, Eindhoven University of Technology,
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands,
{a.kuzmanovska, r.h.mak, d.h.j.epema}@tue.nl

Abstract. In many clusters and datacenters, application frameworks are used that offer programming models such as Dryad and MapReduce, and jobs submitted to the clusters or datacenters may be targeted at specific instances of these frameworks, for example because of the presence of certain data. An important question that then arises is how to allocate resources to framework instances that may have highly fluctuating workloads over their lifetimes. Static resource allocation, a traditional approach for scheduling jobs, may result in inefficient resource allocation because of poor resource utilization during off-peak hours. We address this issue with a strategy for the dynamic deployment of a component-based framework by extending a resource manager responsible for scheduling jobs in multi-cluster environments. This extension allows scheduling multiple concurrent instances of the framework as long-running utility jobs that share computational resources of the cluster. In order to accommodate the fluctuating resource demands of frameworks, we consider two provisioning policies for dynamic resource allocation: OnDemand and Proactive provisioning. We evaluate the effectiveness of both policies by comparing them with static resource allocation on the DAS4 multi-cluster system. Our results shows that dynamic resource allocation gives at least 30% improvement over the static resource allocation with respect to both the utilization of the resources and the reject rate of the applications within the framework.

Keywords: cluster, datacenter, framework, scheduling, dynamic deployment, resource utilization.

1 Introduction

The growing demand for computational resources has resulted in an increased popularity of clusters, grids, clouds and other data center environments. Various frameworks have been developed for these systems to accommodate domain-specific applications such as MapReduce [1] and Dryad [2] for parallel data-intensive applications, Pregel [3] for large-scale graph processing, and various component-based frameworks for specific application domains such as video processing [4]. Once installed in a cluster or datacenter, these frameworks act as

utilities to which users can submit jobs that adhere to the programming models of the frameworks. The immediate question that arises is how many resources to allocate to framework instances in the face of time-varying workloads. In this paper, we address this question with the design, the implementation, and the analysis of a dynamic resource allocation mechanism for scheduling component-based frameworks in clusters.

There are different reasons for having the schedulers in clusters and datacenters schedule framework instances rather than separate, single jobs, and leave the scheduling of the single jobs to the frameworks themselves. First, it relieves schedulers of large clusters and datacenters of a potentially very high load of scheduling decisions. Secondly, it may be difficult to teach the schedulers about all the intricacies of potentially many frameworks that may influence the quality of scheduling decisions. Thirdly, frameworks typically require their own configuration and deployment steps of variable complexity. For instance, some frameworks require a distributed file system to be set up with a certain replication factor (e.g., HDFS [5] for MapReduce), whereas others may need name servers or component repositories to be installed. Although these frameworks may require complex and potentially time-consuming deployment, once deployed, they act as long-running utilities serving large numbers of users who may submit highly fluctuating workloads and the cost of their deployment can be amortized across many jobs.

A common approach for allocating resources to framework instances is static resource allocation, where each framework instance runs on a fixed number of resources over its lifetime. However, even though many frameworks have their own resource management, static allocation may lead to periods of over- and under-utilization of the allotted resources and is therefore not a suitable solution. In contrast, dynamic resource allocation reflects changing resource requirements of a framework instance by changing the fraction of resources allotted to the framework during its lifetime. In this approach, each framework instance is allotted a minimum number of resources, sufficient for its initial deployment. As the load submitted to the framework changes over time, its resource allocation is continuously adapted, in order to achieve continuously a high utilization.

There are several challenges in using dynamic resource allocation for allocating cluster resources to frameworks. First, framework extendibility is essential in the context of dynamic resource allocation, but unfortunately, not all frameworks are extendible. Secondly, most frameworks are developed independently and their local resource managers are not capable of communicating with external resource managers. Thirdly, resource provisioning policies at the cluster side have to meet the fluctuating resource demands of all competing frameworks.

In previous work, we have designed and implemented the KOALA [6] resource manager for multi-cluster systems such as the DAS4 [7]. The original purpose of KOALA was to support co-allocation, i.e., the allocation of processors in multiple clusters to single parallel (MPI) applications. Later we have incorporated support for scheduling various application types into KOALA, e.g., Bags-of-Tasks [8], workflows [9], and malleable applications [10]. In all of these cases, the jobs

submitted to KOALA are single applications. In contrast, previous work on support in KOALA for scheduling of MapReduce clusters [11] addresses scheduling of multiple jobs as a part of single MapReduce instances.

The purpose of this paper is to present the design, the implementation, and the analysis of an extension of the KOALA resource manager for the dynamic deployment of the FLUENT framework [4, 12] as long-running utility jobs. In our case, the "jobs" scheduled by KOALA are instances of the FLUENT framework rather than single jobs. Our extension of KOALA provides two-level resource management. At the first level, KOALA allocates resources to the frameworks, and at the second level, local resource managers within each framework instance use the allotted resources for the deployment of jobs submitted to them. Furthermore, these local resource managers can negotiate resource allocation with KOALA: additional resources may be requested or unused resources may be released. Our final aim is to create a *generic* extension to KOALA that allows a wide range of frameworks to be scheduled dynamically in cluster and datacenter environments. The research reported in this paper contributes towards that goal by:

- An extension of the KOALA resource manager for the dynamic deployment of the FLUENT framework as a long-running utility (Sections 2 and 3).
- The introduction of two provisioning policies, OnDemand and Proactive, for the dynamic resizing of FLUENT framework instances (Section 3).
- The experimental evaluation of the proposed extension including the policies by means of synthetic workloads in a real cluster environment (Section 4).

2 The FLUENT Framework

FLUENT is a distributed component framework for run-time composition of component-based applications [4, 12]. Figure 1 visualizes the building blocks of the framework, called framework entities, in terms of a client-server architecture. From a logical point of view, the server side of the framework is organized in three layers: the *Master* layer, the *Orchestrator* layer, and the *Runtime* layer. Besides the framework entities divided across the three layers, the server side of the framework consists of two types of file-based storage for storing re-usable applications and components: a *Global repository* and a *Local repository*. From a deployment point of view, the framework entities which are deployed on a physical node define the node's role as either a *client*, a *master*, an *orchestrator* or a *worker node*. A single physical node can have multiple roles, with the only restriction that two worker nodes cannot be placed on a single physical node.

The client side of the framework comprises client nodes with *GUITool* entities deployed on them. The *GUITool* is a user interface for managing components and applications which are stored and deployed on the server side of the framework. For that purpose, this entity offers a set of interfaces that cover various aspects of application management such as discovery of available components, composition of applications, deployment of composed applications, and monitoring and

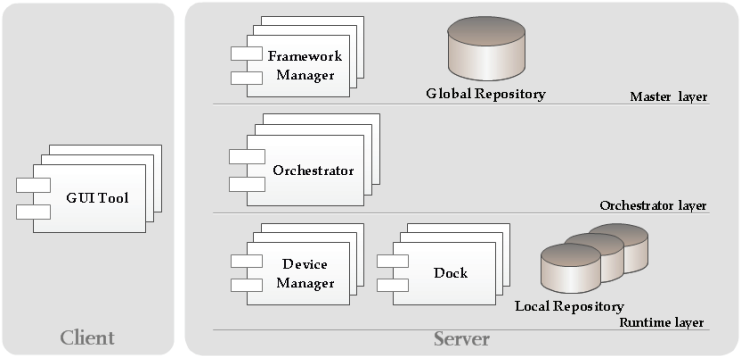


Fig. 1. The client-server architecture of the FLUENT framework.

dynamic reconfiguration framework entities. A single *GUITool* corresponds to a single user of the framework, but a single client node may contain multiple *GUITools*.

The server side of the framework comprises master, orchestrator and worker nodes distributed across the *Master*, the *Orchestrator*, and the *Runtime* layers, respectively. A master node is reserved for the *FrameworkManager* entity and the *Global repository*. The *FrameworkManager* entity is the central part of the FLUENT framework that provides a registry-based entity subscription and entity discovery service to the rest of the framework. The *FrameworkManager* entity manages information where other entities are hosted and allows dynamic configuration of all framework entities. The *Global repository* is a file-based storage that holds two types of data: available components in the form of shared libraries and applications composed of these components in the form of description files. A single master node exists within the FLUENT framework with a single *FrameworkManager* installed on it.

An orchestrator node is used for deployment of *Orchestrator* entities which provide key functionality to compose, deploy, and monitor an application. An *Orchestrator* entity acts as an application manager which enables placement of application components on worker nodes. A single instance of it manages a single application at a time. The number of possible orchestrator nodes within the framework depends on the number of applications running concurrently within the framework. A single orchestrator node may host multiple *Orchestrators*.

A worker node has a *Local repository* and a *DeviceManager* entity deployed on it. The *DeviceManager* entity is the basic processing unit in the framework which is responsible for application execution and monitoring the resource usage by the application. Components of deployed applications are isolated in separate containers within the *DeviceManager* called *Docks*. A *Dock* entity is a wrapper for application components that manages the connections between them. A subset of the *Global repository* is installed on a worker node in the form of a *Local repository* which holds components and applications available locally. A common deployment of the framework comprises multiple worker nodes with a single *DeviceManager* installed on each of them.

An application running in the framework involves a single *Orchestrator* that orchestrates deployment of its components on a single or across multiple *Docks* according to the deployment specification in the application description file.

Since the numbers of nodes employed by the two of the layers, *Orchestrator* and *Runtime*, depend on the current load in the framework, both layers need to be dynamic in order to enable dynamic deployment of the framework in cluster environments. As part of the *FrameworkManager*, there is a resource manager capable of handling dynamic changes of physical resources, but it does not handle scheduling of applications over worker nodes in the framework. We address this issue by extending the resource manager with a scheduler that is responsible for scheduling applications submitted to it. This scheduler places application components on worker nodes according to the FCFS scheduling policy with preference to reuse partially busy nodes before using the idle nodes.

3 KOALA Extension for Dynamic Scheduling of Frameworks

In this section, we describe the mechanisms and policies for dynamic resource allocation by KOALA to the FLUENT framework, as a representative of component-based frameworks. First, we describe the KOALA resource manager used for scheduling jobs in multi-cluster environments. Then we present the additional components that are needed and how they should work together so that KOALA is able to achieve dynamic deployment of frameworks such as FLUENT. Finally, we discuss the resizing mechanisms and the provisioning policies used for dynamic allocation of resources.

3.1 The KOALA Resource Manager

KOALA [6] is a resource manager for scheduling jobs in multi-cluster environments, where each cluster consists of a number of compute nodes used for computations only, and a single head node used as an access point to the cluster. The kernel part of the resource manager is the scheduler that schedules jobs by placing them on suitable cluster sites according to its placement policies. Once the compute nodes are allocated to the job, the actual job submission to those nodes is done by specialized interfaces called *runners* that provide the ability to submit and monitor jobs of different application types. In the past we have implemented runners for rigid parallel applications [6], cycle scavenging jobs [8], workflows [9], malleable applications [10], and map-reduce jobs [11].

3.2 System Architecture

In this paper, the "jobs" scheduled by the KOALA resource manager are fully functional FLUENT framework instances. Scheduling such a framework instance involves the deployment of the three server-side layers of the FLUENT framework on the cluster nodes allocated by the KOALA scheduler in such a way that each

node executes framework entities from a single layer only. The framework is deployed on exactly one node dedicated to the *Master* layer, at least one node dedicated to the *Orchestrator* layer, and multiple nodes dedicated to the *Runtime* layer. We distinguish two types of phases in dynamic scheduling of the framework-based job: the initial and the resizing phase. The initial phase covers the initial deployment of the framework on a minimal number of nodes required for the job execution, which is given as an input to the job in the form of a job description file (jdf). Required nodes are allocated by KOALA, and distributed among the three layers. In the resizing phase, the numbers of nodes dedicated to the *Orchestrator* and the *Runtime* layers are changed based on the load submitted to the framework. The latter changes are negotiated with KOALA where nodes are the unit of resource allocation.

To add support to the KOALA scheduler for scheduling framework-based jobs on a multi-cluster system, we have extended the original KOALA architecture with two components: a runner called the *FrameworkRunner (FR)*, and a global job manager called the *FrameworkJobManager (FJM)*, which keeps track of all running FLUENT instances. Figure 2 provides a high level overview of the newly introduced KOALA components and their iterations involved in scheduling a single FLUENT framework on a multi-cluster system.

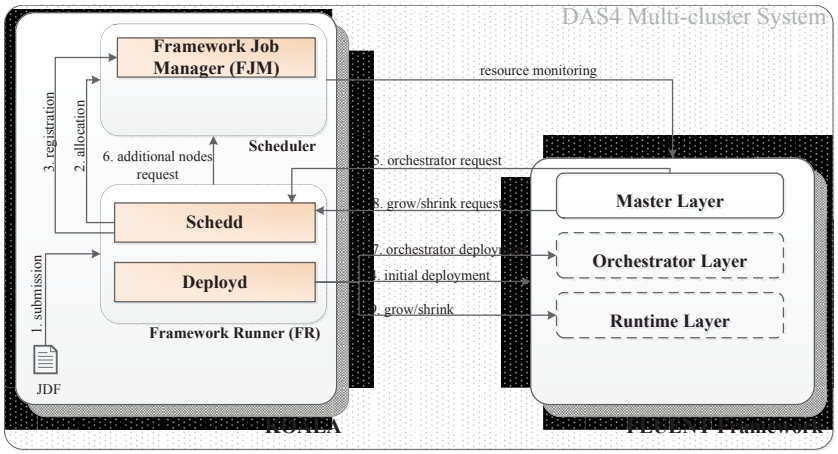


Fig. 2. The sequence of steps involved in scheduling a single FLUENT framework by the KOALA scheduler.

The *FJM* has been added to the scheduler part of the KOALA resource manager and supports the deployment of multiple frameworks. For that purpose, it maintains connections of every framework instance deployed through KOALA and their metadata such as a unique framework identifier, the location of the configuration files, the deployed sites, and the location of the master node. Whenever the *GUITool* needs access to the deployed framework, it can obtain this information from the *FJM*.

The *FR* is used for scheduling framework-based jobs which requires a job description file as an input. This job submission interface consists of two components: a schedule demon *Schedd* that interfaces with the KOALA scheduler, and a deployment daemon *Deployd* that interacts with the FLUENT framework deployed through the *FR*. *Schedd* communicates with the KOALA scheduler in order to provide the desired number of nodes to the framework scheduled by it, whereas *Deployd* deploys the framework entities on the nodes allotted by the KOALA scheduler. *Deployd* handles the communication with the deployed framework, and is responsible for all changes in the deployment of the FLUENT instances. A single *FR* corresponds to a single FLUENT instance scheduled for deployment.

The interaction between these components and a framework instance scheduled for deployment starts with the job submission (step 1 in Figure 2). The *Schedd* component of the *FR* processes the jdf, received as an input from a job submission side, and subsequently requests (step 2) the desired number of compute nodes from the KOALA scheduler. After the successful allocation of these nodes, *Schedd* registers (step 3) the submitted framework with the newly introduced *FJM*. Subsequently, the control is transferred to the *Deployd* component which interacts with local resource managers of the clusters, e.g., SGE, to deploy the framework entities and to install the file system which contains the repositories of the framework (step 4). These four steps capture the interactions in the initial phase.

In the second phase, two events may cause dynamic resizing of the framework: the submission of a new job to the framework or the completion of a running job application by the framework. In response to a submission of a new job, the *FrameworkManager* requests deployment of *Orchestrator* entities (step 5), for all applications in the job, from the *FR*, for which either an additional node may be requested (step 6) from the KOALA scheduler or an already allocated orchestrator node can be used (step 7). Based on the resizing mechanisms described in the next section, the *FR* resizes the *Orchestrator* layer.

Once a novel *Orchestrator* entity has been deployed by *Deployd*, it subsequently takes care of deploying the job applications in FLUENT. This is not depicted in Figure 2 since it is an intra-framework activity. KOALA is not aware of the application deployment until the framework detects a lack of suitable worker nodes and requests additional ones from the *FR* (step 8). As a response to such a request, *Schedd* may either request additional nodes from the KOALA scheduler (step 5), which are subsequently deployed as worker nodes by *Deployd* (step 9), or may reject the request based on the provisioning policies described in the next section.

3.3 Resizing Mechanisms and Provisioning Policies

In addition to allocating compute nodes for the initial deployment of a framework, the *FR* dynamically resizes two layers of a deployed framework: the *Orchestrator* and the *Runtime* layers. For each layer, we introduce a resizing mechanism to handle changes in the number of compute nodes allocated to the framework.

The mechanism for resizing the *Orchestrator* layer is based on the current number of *Orchestrators* deployed on orchestrator nodes. The *Orchestrator* layer of a framework instance is extended by an additional node when the average number of deployed *Orchestrators* exceeds a threshold. The threshold value is empirically chosen based on the performance analysis described in Section 4. When an orchestrator node is idle during a period of time, which means there are no *Orchestrators* deployed on it, the node is removed from the *Orchestrator* layer and return to KOALA. The resizing mechanism is such that at least one orchestrator node is always available for *Orchestrator* deployment.

The mechanism for resizing the *Runtime* layer is regulated by one of two provisioning policies called the **OnDemand** and the **Proactive** policy. In the **OnDemand** policy, FLUENT takes the initiative for resizing the *Runtime* layer by following the pattern of job submissions to a framework instance. In contrast, in the **Proactive** policy it is the KOALA resource manager that takes initiative for resizing by keeping the utilization of a framework instance within certain bounds without knowing any details about the framework activities. In both policies, KOALA sets the maximum number of worker nodes allowed per framework instance to a value F_{max} which is a general static value applied to all instances. FLUENT does not support application migration but it does allow partially busy worker nodes. When FLUENT deploys new applications, it tries to pack them on partially busy, rather than idle, worker nodes.

The **OnDemand** policy resizes the *Runtime* layer based on framework requests of two types called grow requests and shrink requests. This policy allows the initial deployment of the framework with the minimal number of nodes, a single master and a single orchestrator node, whereas the worker nodes are deployed dynamically as part of the resizing. When a FLUENT instance does not have sufficient idle worker nodes for the deployment of a newly submitted job, it sends a grow request for the number of additional worker nodes it needs for the job. We assume the framework knows how many nodes are required for job execution, and we will show how FLUENT calculates the number of nodes needed for our example applications in Section 4. The *FR* adds the requested number of worker nodes to the *Runtime* layer of the framework, unless the value F_{max} is exceeded or KOALA does not have free resources. A shrink request is sent when the framework instance has worker nodes that have been idle for a time period of length at least t_{idle}, to which the *FR* responses by removing these idle worker nodes from the *Runtime* layer. The value of the parameter t_{idle} is empirically chosen with the performance analysis described in Section 4.

The **Proactive** policy, on the other hand, resizes the *Runtime* layer in response to requests by the KOALA scheduler. This policy does require worker nodes as part of the initial deployment of the framework; this number can be changed because of resizing, but will never go below the number of nodes used in the initial deployment. By adding to and removing nodes from the *Runtime* layer, the policy tries to keep the average CPU utilization of the *Runtime* layer between two threshold values U_{min} and U_{max}, which are specified on the scheduler side. Based on monitoring information, the KOALA scheduler expands the

Runtime layer by the same number of worker nodes as in the initial deployment when the average CPU utilization of the *Runtime* layer exceeds U_{max}, and contracts it by the number of idle worker nodes when the average CPU utilization of the *Runtime* layer drops below U_{min}. Again, when KOALA wants to shrink the *Runtime* layer, it only removes nodes that have been idle for at least t_{idle}.

When KOALA cannot meet the framework requirements and rejects grow requests, depending on the application type, the framework will either queue the submitted application until the current worker nodes can deploy them or reject the application, e.g., in a video surveillance case (see Section 4).

4 Performance Evaluation

In this section, we present a performance evaluation of dynamic resource allocation to the FLUENT framework as a utility in the DAS system. First, we describe the experimental setup and the types of applications supported by the FLUENT framework with an emphasis on the applications we use in the experiments. Then we describe the conducted experiments and the workloads used in the experiments. Finally, we analyze the obtained results.

4.1 Experimental Setup

For the purpose of the evaluation, we use the DAS multi-cluster system as the experimental environment. The DAS4 [7] is the fourth generation of this system which is distributed accross research institutes and organizations in the Netherlands. The system consists of six clusters and comprises roughly 200 compute nodes with properties as shown in Table 1. The Sun Grid Engine(SGE) to which KOALA interfaces, operates as the local resource manager on each of the DAS clusters.

Table 1. Specification of compute nodes in the DAS multi-cluster system

Processor	Dual quad-core Intel E5620 at 2.4 GHz
Memory	24 GB RAM
Network	10 Gbit/s Infiniband, 1 Gbit/s Ethernet
Disk	2 ATA OCZ Z-Drive R2 with 2 TB (RAID0)
OS	Linux CentOS-6
JVM	jdk 1.6.0_27

The experiments were performed within a single cluster with 32 compute nodes and a single head node. The initial deployment of the FLUENT framework comprises a single master node, a single orchestrator node, and multiple worker nodes. The number of worker nodes used for the initial deployment of the framework depends on the experiment and the provisioning policy. The components

available for an application composition are placed in the *Local Repositories* installed on each worker node, whereas the *Global Repository* exist as an union of the *Local Repositories* .The Infiniband network is used for inter-framework communication among the framework entities due its low latency in data transmission. The clients are deployed on the head node.

The worker-node idle time parameter t_{idle} and the utilization threshold values U_{min} and U_{max} used in the provisioning policies are determined as part of the calibration experiment. Since the **Proactive** policy requires data about CPU utilization of the allocated nodes, we collect the CPU utilization statistics of every node with a sampling interval of 40 second using the open-source "audria" utility tool [13] and the standard Linux monitoring tool "pidstat".

4.2 FLUENT Applications

FLUENT applications are component-based applications, and they are composed from fully independent components with well-defined interfaces and specified behavior. Components are reused across multiple applications and can be dynamically orchestrated to build various applications. A FLUENT application is represented by an application description file in which the used components are defined, together with the bindings between them and deployment information.

The FLUENT framework has been conceived as a general-purpose framework, but was originally used as a framework for video processing multimedia applications in the area of surveillance and transport logistics in the scope of the ViCoMo project [14]. Therefore, the framework comes with libraries of components that provide video encoding/decoding, streaming, and customized support typically used in video processing applications. The surveillance applications are computationally intensive applications with fluctuating resource requirements over their long lifetimes, and require short deployment time (response time).

Both long running computationally intensive and parallel data-intensive applications are conveniently supported by the FLUENT framework. In our experiments, we use two applications: an application from the video processing domain as a representative of the computationally intensive applications, and a more general application that performs a word count on a file as a representative of data-intensive applications. Each of these applications uses its own library of components described below.

The video processing library used in a *RemoteLaplace (RL)* application consists of two components. Both components have one interface and communicate to each other by using a buffer or overwrite channels. The applications provided by this library have a simple producer-consumer architecture. The first component performs image sharpening by applying a Laplacian filter on the input video stream which has been smoothed to remove noise. The component generates two outputs shown in real-time, the original video and the transformed video. The video transformation is based on two parameters that are provided remotely by the second component which simulates user input by generating new values for the parameters every s seconds. Therefore the computational load changes every s seconds.

The library used in a *StreamingWordcount (SWC)* application consists of five components, three core components and two auxiliary components that allow applications to be structured with a variable number of core components. As opposed to the communication between mappers and reducers by means of files in MapReduce, these components communicate with each other by using buffer channels. The main computing component of the library is a *mapper*, which emits key-value pairs for each word of the input block of text. The *reader* component provides the *mapper* with input by reading and splitting the given input file in multiple blocks, and emitting each of them separately. The counting part is performed in the *counter* component which is responsible for generating the output file. These three core components allow applications with only one *mapper* computing component. In order to support parallel processing in data-intensive applications, the auxiliary components are used for composing applications with multiple *mappers*. The *multiplexer* auxiliary component splits up and redirects its input across two outputs, whereas the *demultiplexer* component redirects two received inputs to one output. Multiple levels of *multiplexer* and *demultiplexer* components can be used, and the number of *mappers* in an application can be 2^n, for $n = 1, 2, \ldots$ when using n levels of auxiliary components.

4.3 Experiments and workloads

We perform two types of experiments and classify them as either micro- or macro-experiments. By means of two micro-experiments, we investigate the characteristics of the operation of the framework and of the execution of single jobs. In the first micro-experiment we asses the time required to install an instance of the FLUENT framework, the time required to process a grow request, and the time required to deploy a job submitted to a running instance of FLUENT. In the second micro-experiment, we examine the CPU utilization of the *Orchestrator* and the *Runtime* layers for both our application types to find out how many applications fit on one node in both layers. Within this experiment, we deploy the two applications separately.

The *RL* application has two components and its runtime is restricted to 300 sec; it operates on a video with a playing time of 300 sec at a frame rate of 60 fps and with a frame resolution of 320x239 pixels. The filter parameter s is set to 30 s, which means that the transformation is performed 10 times during the application life time. The *SWC* application has three components, a *reader*, a single *mapper*, and a *counter* component. It processes a 10 MB file by reading data blocks of 100 KB that are processed in sequence by the single *mapper*. When deployed using a one-to-one mapping between components and cores, its runtime is 44 minutes. In both the *SWC* and *RL* applications, the application components are deployed within a single worker node.

By means of the macro-experiments, we evaluate the performance benefit of dynamic resource allocation with the provisioning policies over static resource allocation by submitting workloads consisting of many jobs. In the first macro-experiment, we perform a sensitivity analysis of the parameters t_{idle}, U_{min} and U_{max} of the dynamic provisioning policies. In the second macro-experiment,

we assess the performance of the dynamic allocation of resources with our two provisioning policies (OnDemand and Proactive) versus static allocation.

In this assessment we use a synthetic workload W_{rl} that consists of jobs each running a number 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. Therefore, the arrivals of the applications in the workload are modeled by a batch arrival process with the jobs (batches of applications) arriving according to a Poisson process with rate $\lambda = 0.0278$ per second. The size of the batches has a geometric distribution over the interval $[1, 10]$ with mean 2.5. We choose the arrival rate such that the framework with 20 worker nodes has utilization of approximately 50%. Since every the *RL* video application requires immediate deployment and sufficient resources over its lifetime, in order to evaluate the performance, we use the *reject rate*; jobs can be partially accepted and rejected if only several but not all of the video applications in its batch fit on the available resources, and we define the reject rate as the percentage of all applications across all jobs in the workload that are rejected. In addition we use the utilization as a metric, defined as the ratio of actually used and the total number of (statically or dynamically) allocated resources. In this experiment, the maximum number of worker nodes per framework, denoted by F_{max}, is set to 20 worker nodes.

4.4 Experimental results

The results from the first micro-experiment in which we investigate the overheads of the FLUENT framework, show that the time needed for the initial deployment of the framework is affected by the type (but not the number) of nodes involved in the deployment. When the **OnDemand** policy is used, the initial deployment of the framework involves only master and orchestrator nodes, and takes on average 11 s. In case of static allocation or dynamic allocation with the **Proactive** policy, besides the master and orchestrator nodes, workers nodes are also involved in the initial deployment of the framework, and it takes on average 45 s. The difference between the initial deployment times in the two cases is due to the additional time needed for the worker nodes to install the *Local Repositories*, which includes the transfer of application components from the *Global Repository*.

For the deployment of a job, a running framework instance needs on average 62 ms when all worker nodes needed for deployment are available; this is the time from the job submission on the client side until the applications included in the job start operating on worker nodes, including the scheduling and the placement time of their application components. The average overhead in the framework for handling a grow request, denoted by O_{grow}, which includes node allocation by KOALA and the deployment of the suitable entities on them by the *FR*, is approximately 30 s.

As to the results of the second micro-experiment, in which we investigate the number of applications that can be hosted on a single node, Figure 3 shows that the utilization of the *Orchestrator* layer is neither application-type nor application-size specific. During the application deployment, around 20% of the CPU is

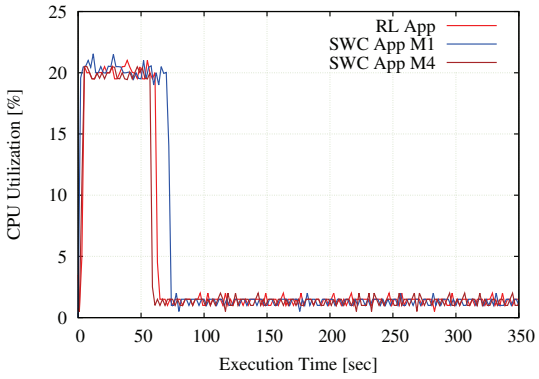
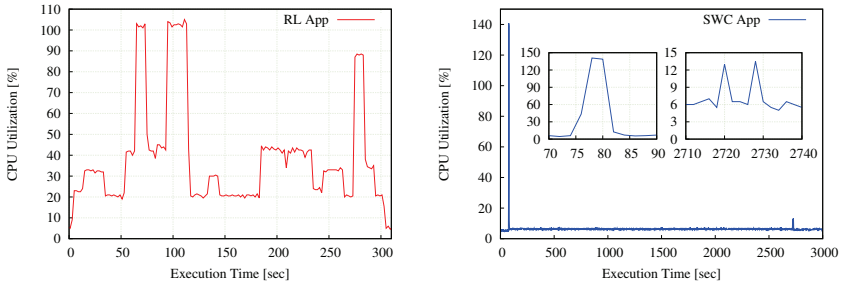


Fig. 3. The CPU utilization of a single node in the *Orchestrator* layer (SWC App M1, M4 indicates the SWC application with 1 or 4 mappers).

needed at the beginning. This peak covers the deployment of the orchestrator entity itself, and after it around 2% is needed for orchestrating an application. The period of high CPU utilization lasts for 70s and it occurs only once per *Orchestrator* lifetime.

We can conclude that a single orchestrator node can host multiple *Orchestrators* with a potential delay of $70N/5$ sec when N applications are submitted simultaneously. As the jobs in the macro-experiment consist of at most 10 applications, the expected delay is at most 140 seconds. Based on these results, we fix the threshold used in the resizing mechanism of the *Orchestrator* layer to 45. When the number of *Orchestrators* deployed on an orchestrator node exceeds this value, KOALA introduces a new node in the *Orchestrator* layer.



(a) *RL* application (b) *SWC* application

Fig. 4. CPU utilization of a worker node in the *Runtime* layer for different applications.

With respect to the *Runtime* layer, Figure 4 (a) shows that the *RL* application is computationally intensive with a fluctuating CPU utilization pattern over its execution time. The fluctuating CPU utilization leads to the conclusion that only a single application can be placed on a single node without overloading. On

the other hand, the *SWC* application (Figure 4 (b)) has very low CPU utilization, around 5% during the *mapper* execution, with two short peaks at the beginning and at the end of the application execution. These two peaks correspond to the I/O operation for reading the input file and writing the results back to the disk in the *reader* and *counter* components, respectively. Therefore, a single worker node can host an *SWC* application with 20 *mappers* without overloading. As a conclusion we can say the utilization of the *Runtime* layer varies depending on the type of deployed applications.

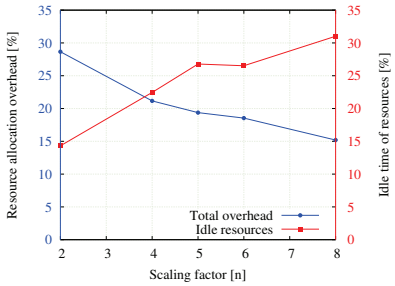


Fig. 5. The resource allocation overhead due to grow requests and the fraction of idle resources depending on the worker-node idle time $t_{idle} = n \cdot O_{grow}$ in the OnDemand policy.

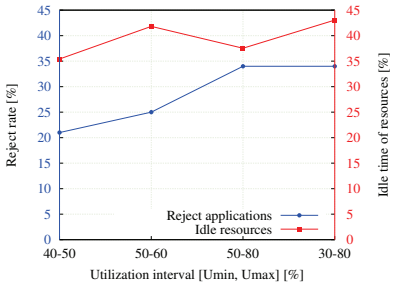
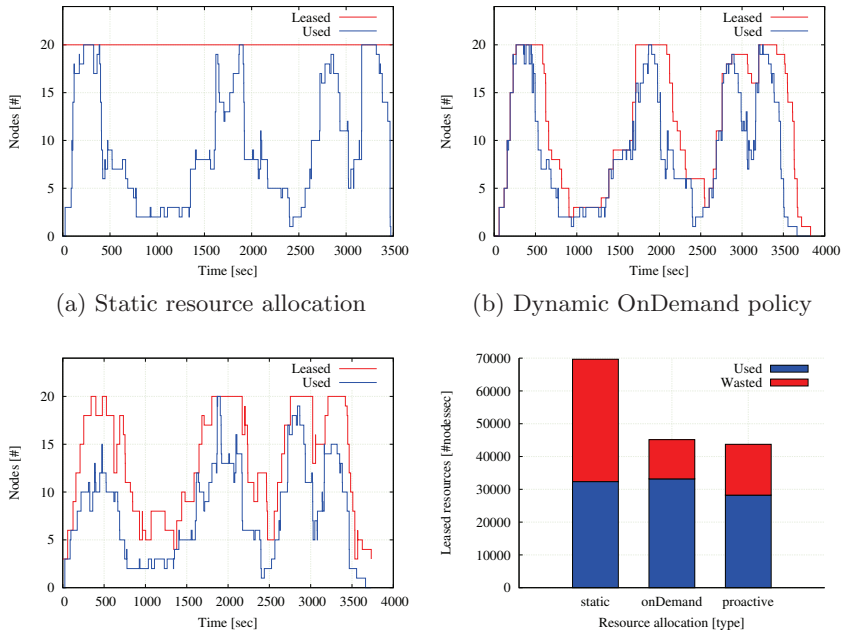


Fig. 6. The reject rate of applications and the fraction of idle resources vs. different ranges of U_{min} - U_{max} in the Proactive policy.

In the first macro-experiment, we conduct a sensitivity analysis of the parameters t_{idle}, U_{min} and U_{max} of the provisioning policies. First we consider t_{idle}, the time worker nodes can remain idle before being released. We relate t_{idle} to the average overhead for handling a grow request O_{grow}, and we assess effect of the performance of scaling O_{grow} by a factor n on the resource allocation overhead and the resource idle time t_{idle},so by setting $t_{idle} = n \cdot O_{grow}$.

Figure 5 shows the behavior of the framework when processing the workload W_{rl} for different values of the scaling factor n in terms of the total overhead due to grow requests and the idle time of the allocated resources. The higher the scaling factor, the lower the number of grow requests and so the less overhead for resource allocation, but the higher the fraction of idle resources. In the **Proactive** policy, where the value t_{idle} is used as a control mechanism to ensure that resources are not needlessly kept in the framework, we want to release the nodes as soon as they become idle, so we fix the scaling factor at $n = 2$. In the **OnDemand** policy, we fix the scaling factor at $n = 6$ for the *RL* application.

Figure 6 shows the behavior of the framework when processing the workload W_{rl} for different values of the parameters U_{min} - U_{max} in terms of the reject rate of applications and the idle time of the allocated resources. Since the total overhead due to grow requests cannot be measured on the framework side, we use the application reject rate as a metric to determine the values of U_{min} and



(a) Static resource allocation (b) Dynamic OnDemand policy

(c) Dynamic Proactive policy (d) The total used and wasted resources

Fig. 7. The amounts of resources allocated and used with static and dynamic allocation for the workload with the *RL* application (over time and total).

U_{max}. As we can see in Figure 6, setting the utilization levels U_{min} and U_{max} to 40% and 50%, respectively, gives the lowest number of rejected applications and the smallest fraction of idle resources.

The results from the second macro-experiment show that both our policies for dynamic allocation of resources improve the performance over static resource allocation in terms of the resource utilization and the application reject rate. In Figure 7 we show the amount of allocated and used resources over time, and the total amount of leased resources (used and wasted) during the experiments with the video application workload. Clearly, with the two dynamic policies, the amounts of allocated resources follow the patterns of the used resources pretty well, and the dynamic policies waste significantly less resources than static allocation (Figure 7 (d)).

Table 2. The reject rate and the utilization with the *RL* application vs. the allocation policy (all values are in %.)

Policy	Reject Rate	Utilization
Static	13%	46%
OnDemand	13%	73%
Proactive	21%	65%

In Table 2, we show the performance metrics for each of the policies when processing the W_{rl} workload in terms of the reject rate of applications and the the actual utilization of the allocated resources as the ratio of used and allocated resources from Figure 7 (d). We find that the **Proactive** policy improves resource utilization by approximately 30%, but it is not as good as the **OnDemand** policy, which improves utilization by 37%. For the **Proactive** policy, the improvement comes at the price of a higher reject rate, as shown in Table 2. As a conclusion, we can say the **OnDemand** policy is more suitable for applications with a batch arrival pattern such as the *RL* jobs, because it follows the pattern of resource usages in the framework and in the same time keeps the reject rate on the same level as the static allocation.

5 Related Work

The system that is closest to ours is Mesos [15], which provides a two-level scheduling mechanism for sharing cluster resources across multiple frameworks, and in particular shares data among the frameworks. Mesos periodically does resource offers to individual frameworks that can either accept or reject them, and so it is the global scheduler that takes the initiative. In contrast, we design, implement and compare mechanisms in which either the frameworks explicitly express their requirements and the global scheduler, KOALA, allocates the requested resources, or the initiative lies with KOALA. As another difference, Mesos acts as the owner of the cluster resources while KOALA does not own resources but is built on top of, and interfaces to, the local cluster schedulers. The advantage of this way of operation is that our way of supporting frameworks does not in any way entail any change in the setup or deployment of the clusters.

YARN [16] is a resource manager that explicitly has multi-framework support, but it interfaces to application managers rather than framework managers as KOALA does and so in fact, it still provides a single-level scheduling mechanism. However, as opposed to Mesos, it is request-based, like our OnDemand policy. Omega [17] follows the interesting decentralized approach of *shared-state scheduling* by having the schedulers of multiple frameworks compete for the resources of the complete cluster without a central authority—optimistic concurrency control is employed to mediate between conflicting allocation decisions of the separate schedulers.

Other related work mostly covers automatic resource management and scheduling of jobs in clusters and grids. Cluster and grid resource managers such as Torque [18], Condor [19, 20], and Quincy [21], address jobs that require static resource allocations during their execution.

6 Conclusion

In this paper we have presented an extension to the KOALA resource manager that enables dynamic resource allocation to instances of the FLUENT framework. We have designed and implemented two policies for provisioning resources to

frameworks, and we we have assessed the performance of the dynamic allocation of resources with our two provisioning policies (OnDemand and Proactive) versus static allocation. Our results show that both policies for dynamic allocation of resources improve the performance over static resource allocation by at least 30%, with respect to both the utilization of the resources and the reject rate of the applications within the framework.

As future work, we are planning to design a mechanism in KOALA for dynamically allocating resources in clusters and datacenters to frameworks that is as generic as possible and that can accommodate many different frameworks. In addition, we will refine our policies for doing the dynamic allocations and we will analyze their performance with mixes of different frameworks in a single cluster and other datacenter environments.

References

1. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM **51**(1) (January 2008) 107–113
2. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed data-parallel programs from sequential building blocks. SIGOPS Oper. Syst. Rev. **41**(3) (March 2007) 59–72
3. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: 2010 ACM SIGMOD International Conference on Management of Data. (2010) 135–146
4. Orlic, B., David, I., Mak, R.H., Lukkien, J.J.: Dynamically reconfigurable resource-aware component framework: architecture and concepts. In: 5th European Conference on Software architecture. ECSA'11, Berlin, Heidelberg, Springer-Verlag (2011) 212–215
5. White, T.: Hadoop: The Definitive Guide. 1st edn. O'Reilly Media, Inc. (2009)
6. Mohamed, H., Epema, D.: Koala: A co-allocating grid scheduler. Concurrency and Computation: Practice and Experience **20**(16) (November 2008) 1851–1876
7. [Online]: The distributed asci supercomputer 4. http://www.cs.vu.nl/das4/
8. Sonmez, O.O., Grundeken, B., Mohamed, H.H., Iosup, A., Epema, D.H.J.: Scheduling strategies for cycle scavenging in multicluster grid systems. In: 9th IEEE/ACM Int'l Symp. on Cluster Computing and the Grid. CCGRID '09 (2009) 12–19
9. Sonmez, O.O., Yigitbasi, N., Abrishami, S., Iosup, A., Epema, D.H.J.: Performance analysis of dynamic workflow scheduling in multicluster grids. In: ACM Symp. on High-Performance Parallel and Distributed Computing (HPDC). (2010) 49–60
10. Buisson, J., Sonmez, O.O., Mohamed, H.H., Lammers, W., Epema, D.H.J.: Scheduling malleable applications in multicluster systems. In: IEEE Int'l Conf. on Cluster Computing. (2007) 372–381
11. Ghit, B., Yigitbasi, N., Epema, D.: Resource management for dynamic mapreduce clusters in multicluster systems. In: Proceedings of the 5th Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS) co-located with Supercomputing (SC), IEEE (2012)
12. David, I., Orlic, B., Mak, R.H., Lukkien, J.J.: Towards resource-aware runtime reconfigurable component-based systems. In: 2010 6th World Congress on Services. SERVICES '10, Washington, DC, USA, IEEE Computer Society (2010) 465–466

13. [Online]: A utility for detailed resource inspection of applications. https://github.com/scaidermern/audria
14. [Online]: Vicomo website. http://www.vicomo.org
15. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R., Shenker, S., Stoica, I.: Mesos: a platform for fine-grained resource sharing in the data center. In: 8th USENIX Conference on Networked Systems Design and Implementation. NSDI'11 (2011) 22–22
16. Vavilapalli, V.K.: Apache hadoop yarn: Yet another resource negotiator. In: ACM Symp. on Cloud Computing. (2013)
17. M. Schwarzkopf, A. Konwinski, M.A.E.M., Wilkes, J.: Omega: flexible, scalable schedulers for large compute clusters. In: Eurosys'13. (2013)
18. Staples, G.: Torque resource manager. In: 2006 ACM/IEEE conference on Supercomputing (SC'06). (2006)
19. Raman, R., Livny, M., Solomon, M.: Matchmaking: An extensible framework for distributed resource management. Cluster Computing **2**(2) (April 1999) 129–138
20. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the condor experience. Concurrency and Computation: Practice and Experience **17**(2-4) (February 2005) 323–356
21. Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., Goldberg, A.: Quincy: fair scheduling for distributed computing clusters. In: 22nd ACM Symposium on Operating Systems Principles. SOSP '09 (2009) 261–276