HORIZON 2020 FRAMEWORK PROGRAMME

IOStack

(H2020-644182)

Software-Defined Storage for Big Data on top of the OpenStack platform

D5.3 Consolidated System Monitoring and Deployment Tools

Due date of deliverable: 31-12-2017
Actual submission date: 31-12-2017

Start date of project: 01-01-2015 Duration: 36 months

Summary of the document

Document Type	Deliverable
Dissemination level	Public
State	v1.2
Number of pages	29
WP/Task related to this document	WP5
WP/Task responsible	EURECOM
Leader	Pietro Michiardi (EUR)
Technical Manager	Francesco Pace (EUR)
Quality Manager	Ramon Nou (BSC)
Author(s)	Francesco Pace (EUR), Daniele Venzano (EUR), Pietro Michiardi (EUR)
Partner(s) Contributing	EUR, URV, GDP, BSC, IBM
Document ID	IOStack_D5.3_Public.pdf
Abstract	Nowadays, data-centers are largely under-utilized because resource allocation is based on reservation mechanisms which ignore actual resource utilization. Indeed, while the amount of resources used by applications is not constant throughout their execution, it is common practice to provision (and reserve) resources for peak demand, which may occur only for a small portion of the application life time. As a consequence, cluster resources are reserved but often go under-utilized. A solution is to give resource schedulers the possibility to take decisions based on the resource utilization rather than reservation. However, a direct application of this solution can exacerbate resource contention, which ultimately can result in application failures. In this work, we propose a mechanism that improves cluster utilization, thus decreasing the average turnaround time, while preventing application failures due to contention. Our approach, which monitors resource utilization and relies on resource demand forecasting, is able to reduce the turnaround time by more than one order of magnitude while minimizing application failures. Thus, tenants enjoy a more responsive system and providers benefit from a more efficient cluster utilization. We materialize our solution as a system working along side Zoe Analytics which is a core block of the IOStack toolkit.
Keywords	scheduling, core, elastic, analytics applications, distributed

History of changes

Version	Date	Author	Summary of changes
1.0	04-10-2017	Pace Francesco	First version
1.1	06-10-2017	Daniele Venzano	Write Zoe Analytics sections
1.2	30-10-2017	Pace Francesco	Write Research sections

Table of Contents

Executive summary

This deliverable presents the benefits that can be achieved when low-level performance information is used in a feedback loop to drive scheduling and deployment systems. Nowadays, data-centers are largely under-utilized because resource allocation is based on reservation mechanisms which ignore actual resource utilization. Indeed, while the amount of resources used by applications is not constant throughout their execution, it is common practice to provision (and reserve) resources for peak demand, which may occur only for a small portion of the application life-time. Our approach, which monitors resource utilization and relies on resource demand forecasting, is able to reduce the turnaround time by more than one order of magnitude, while minimizing application failures. We materialize our solution as a system working along side Zoe Analytics, a core component of the IOStack toolkit.

In the first part of the document we describe how the project Zoe Analytics has advanced, the implementation improvements, the interest it generated in several well-known companies and the integration with other core components of the IOStack toolkit.

In the second part of the deliverable we present a dynamic approach to scheduling analytics applications and its implementation in Zoe. Optimization objectives, algorithms and scheduling policies are explained, together with the evaluation, performed with simulations and an implementation based on Zoe.

Part I
Zoe Analytics

1 Zoe Analytics introduction

Zoe Analytics solves the problem of easily, quickly and reliably deploying complex, distributed analytic applications on clusters of physical or virtual machines.

It does so by implementing a thin layer on top of an existing orchestration back-end that offers easy user access to available applications and resources. Zoe applies advanced scheduling concepts to manage the queue of incoming requests, with the goal of minimizing turnaround times.[1]

Zoe manages applications. An example of a Zoe application is a Jupyter Notebook, with the Python kernel, the PySpark library, an Apache Spark master and several Apache Spark workers. This is a full distributed analytic application that requires extensive systems knowledge to set up correctly by hand. Zoe not only automates the deployment of this ZApp (Zoe application), but also assigns resources automatically and dynamically, based on up-to-date monitoring information. Since Spark workers are elastic (at least one is needed to make progress, if more are available progress will be faster) Zoe can regulate the number of running workers based on overall cluster usage metrics. In II we describe the research on scheduling preemption and on dynamically regulating resources for each ZApp service.

ZApps are defined using a JSON document and some meta-data. Creating new ZApps requires knowledge of how the back-end works and its image format. For this reason the creation of ZApps is usually limited to power users and administrators.

2 Summary of progress

Since Zoe was last described in deliverable 5.2, the project has made good progress in numerous areas. Below we touch on the most important points, for a more complete picture, check also the change log available in the Zoe repository on GitHub and the Zoe website.

In 2017 KPMG[2] invested in Zoe and actively participated to the development with one developer and several internships. Their contribution has focused on Kubernetes support and on running automated tests on the code base.

Air France/KLM also continues to use Zoe, contributing useful feedback and bug reports. Zoe has been deployed in all three data centers and is preferred over competing solutions from IBM and others.

2.1 Trademark and logo

The Zoe project has gathered a lot of interest from numerous parties, interested users and competitors alike. EUR often receives queries and collaboration offers from start-ups and the Zoe website is visited almost daily by Google, Amazon and Microsoft employees.

To secure the project identity EUR decided to register a trademark on the name and logo. After research for conflicting registrations, EUR decided to register the name "Zoe Analytics" and the logo visible in figure 1.

At the time of this writing the Eurecom legal department is in charge of the procedure to register the trademark in France. Later the registration will be extend to the EU and other countries, as needed.

2.2 Kubernetes support

Google Kubernetes (`https://kubernetes.io/`) is one of the major players in the container orchestration space. The project is open source and takes its roots from Borg and the long experience of Google in the virtualization business.

[1]The time interval between a request submission and its completion. See deliverable 5.2 for more details.

[2]KPMG DE (`https://home.kpmg.com/de/de/home.html`) is one of the leading providers of audit, tax and advisory services in Germany, employing more than 10.200 people.

Figure 1: The Zoe logo

KPMG has taken the decision to use Kubernetes internally for a number of services and web applications. Since Zoe has been built from the start to share its back-end with other applications, the obvious step was to have Zoe talk to Kubernetes. This way a single cluster, with a single pool of resources, can be used concurrently, by web services, batch analytic jobs and data scientists, without any system administration overhead.

Starting from the 2017.03 release, in March 2017, Zoe supports Kubernetes as a back-end. In the 2017.06 (June) release a big refactoring and clean-up effort was concluded to create a real pluggable back-end system. A well defined interface sits between Zoe and the back-end library, paving the way to support other orchestrators. Even non container-based ones, like OpenStack, are possible. ZApp descriptions have been cleaned-up as well, removing Docker specific entries.

There are other benefits in supporting Kubernetes. While setting up Kubernetes is not easy, ready-made deployments are available on all major cloud platforms, making it possible to have one-click Zoe deployments in the cloud, for example.

Moreover, having a generic back-end API, makes Zoe future-proof and ready to work with new orchestration systems with minimal changes.

2.3 Evolution of the user interface

Initially Zoe was used by experienced users who do not mind using command-line tools. When the user base started growing, the need for a more advanced web interface become more and more evident.

One of the most requested features by users was to have a list of applications to chose from. This request, apparently simple, becomes much more complex once the realities of Zoe deployments are taken into account.

To update the user web interface we needed to define a generic ZApp package, that describes the following items in a generic and portable way:

- parameters: most ZApps have parameters to customize the way they run. Some examples are resource limits (CPU, memory), software versions, locations of external services, etc. Each parameter should come at least with a type, some validation mechanism, a brief documentation and a way to apply the value to the ZApp.
- documentation: how use the ZApp, what libraries and software versions it contains
- permissions: administrators want the ability to decide who can access a certain application
- logo: to make the application list attractive, each ZApp should have a graphical logo
- software image: each container back-end has its own way of managing system images. Swarm and Kubernetes use the Docker image format, that can be built by Docker tools using a Dockerfile
- other meta-data: author name and email, a web URL to check for updates

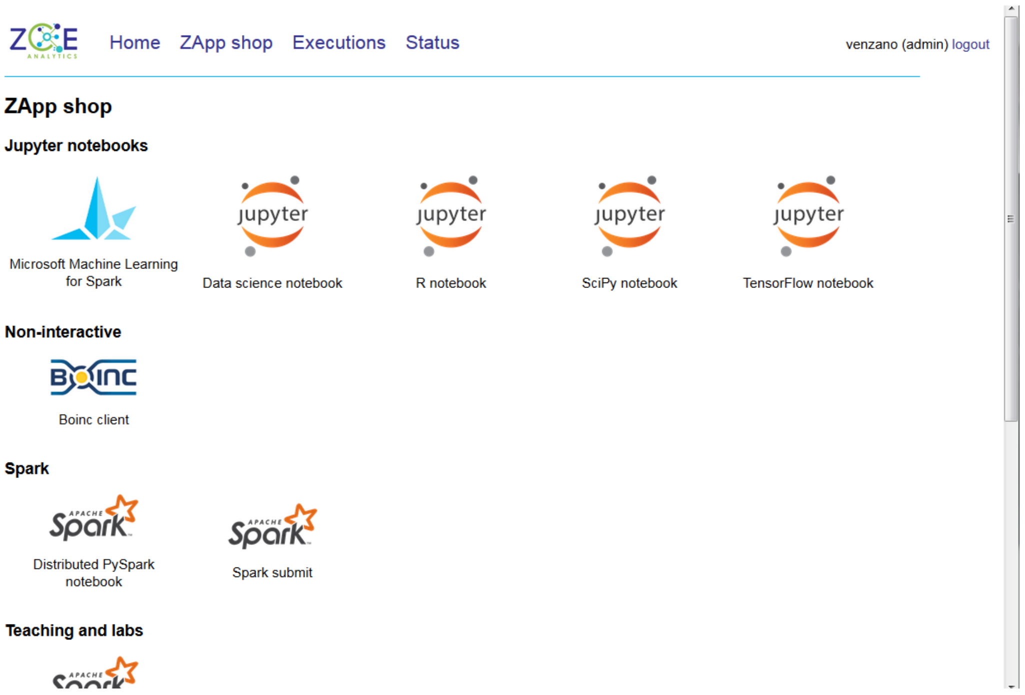


Figure 2: New web interface: the ZApp shop at Eurecom.

With this information it is possible to build in Zoe a generic "app shop", that users can access to list ZApps, read information about each one and start new executions. We made available a number of ZApp packages via GIT and in the source code Zoe distribution, so that Zoe deployers can start with a rich selection of pre-built ZApps.

Together with the "app shop" development, we rebuilt the web interface of Zoe, to have a better usability and provide useful information at a glance. Colors and fonts have also been changed to match the new logo.

Figures 2 and 3 show two screen-shots of the new web interface and the "app shop".

2.4 Industrialization, testing and CI

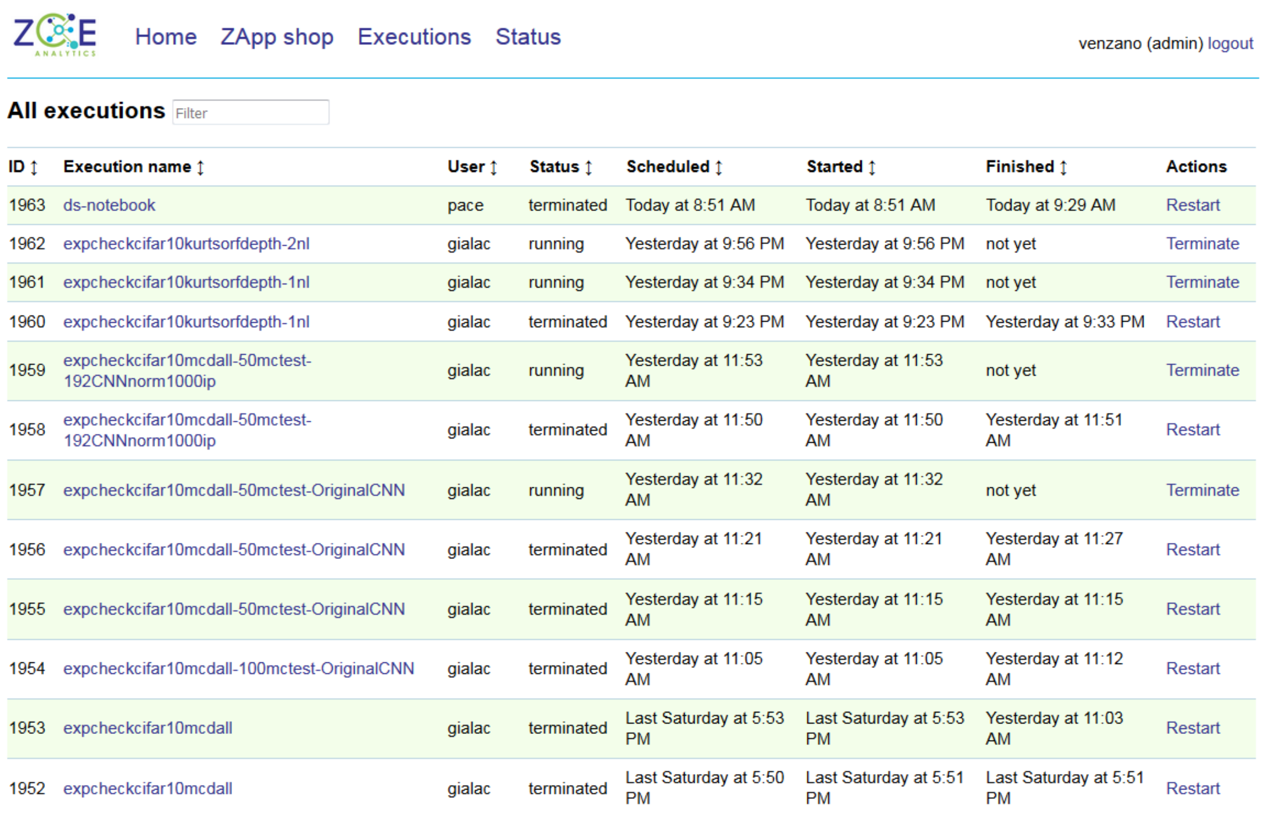
An important contribution of KPMG activities has been to define an environment to automatically test Zoe via a continuous integration (CI) pipeline. While KPMG decided to base their CI pipeline on Jenkins (`https://jenkins.io/`), we decided to implement it with the CI features of GitLab (`https://about.gitlab.com/`).

Of particular interest to an Open Source project like Zoe, GitLab requires the CI pipeline to be defined in a text file together with the source code. The advantage is that it is much easier to describe document and replicate the same pipeline elsewhere, for other contributors. The same feature is also available in Jenkins, but it is not yet fully supported.

Figure 4 visualizes the stages that compose the test pipeline. Each stage is run in a clean Docker container, that is created just for the test, containing the source doe version to test and the necessary library requirements.

The first stage is for static testing. Python is a dynamic language and Python code does no go through a compilation phase. Numerous tools have been developed to scan the code looking for syntax and type errors.

- docs-test: Zoe documentation is written with Sphinx. Here we test that the documentation is well formatted and that all internal references are consistent
- pylint: pylint is a static testing tool for Python code. For Zoe, we decided to set the bar very high and to pass the test, no errors or warning of any kind are accepted.



ID ↕	Execution name ↕	User ↕	Status ↕	Scheduled ↕	Started ↕	Finished ↕	Actions
1963	ds-notebook	pace	terminated	Today at 8:51 AM	Today at 8:51 AM	Today at 9:29 AM	Restart
1962	expcheckcifar10kurtsorfdepth-2nl	gialac	running	Yesterday at 9:56 PM	Yesterday at 9:56 PM	not yet	Terminate
1961	expcheckcifar10kurtsorfdepth-1nl	gialac	running	Yesterday at 9:34 PM	Yesterday at 9:34 PM	not yet	Terminate
1960	expcheckcifar10kurtsorfdepth-1nl	gialac	terminated	Yesterday at 9:23 PM	Yesterday at 9:23 PM	Yesterday at 9:33 PM	Restart
1959	expcheckcifar10mcdall-50mctest-192CNNnorm1000ip	gialac	running	Yesterday at 11:53 AM	Yesterday at 11:53 AM	not yet	Terminate
1958	expcheckcifar10mcdall-50mctest-192CNNnorm1000ip	gialac	terminated	Yesterday at 11:50 AM	Yesterday at 11:50 AM	Yesterday at 11:51 AM	Restart
1957	expcheckcifar10mcdall-50mctest-OriginalCNN	gialac	running	Yesterday at 11:32 AM	Yesterday at 11:32 AM	not yet	Terminate
1956	expcheckcifar10mcdall-50mctest-OriginalCNN	gialac	terminated	Yesterday at 11:21 AM	Yesterday at 11:21 AM	Yesterday at 11:27 AM	Restart
1955	expcheckcifar10mcdall-50mctest-OriginalCNN	gialac	terminated	Yesterday at 11:15 AM	Yesterday at 11:15 AM	Yesterday at 11:15 AM	Restart
1954	expcheckcifar10mcdall-100mctest-OriginalCNN	gialac	terminated	Yesterday at 11:05 AM	Yesterday at 11:05 AM	Yesterday at 11:12 AM	Restart
1953	expcheckcifar10mcdall	gialac	terminated	Last Saturday at 5:53 PM	Last Saturday at 5:53 PM	Yesterday at 11:03 AM	Restart
1952	expcheckcifar10mcdall	gialac	terminated	Last Saturday at 5:50 PM	Last Saturday at 5:51 PM	Last Saturday at 5:51 PM	Restart

Figure 3: New web interface: the execution list can be searched and is easier to read.

- unittests: a small part of the code base (9%) is covered by unit tests. Part of the road-map and future work is to extend the coverage to more areas of Zoe.

The second stage runs an integration test. All Zoe components are started in a clean environment and the most important API endpoints are exercised automatically. This helps to catch incompatibilities with new external library versions, since the clean environment is re-installed each time according to the official installation procedure, by downloading dependencies from the Internet.

The third stage build new Docker images containing the new version of Zoe. If the pipeline reaches this stage, the source code has passed all tests.

The deploy stage deploys Zoe in a staging environment, where it can be tested by hand:

- docs: build and pushes the documentation to a web server accessible internally, so that the latest documentation can be accessed by the developers
- mirror-github: mirrors the GitLab repository to GitHub, for better project visibility
- zoe: deploys Zoe on a virtual machine, so that it can be tested by hand

Finally, the clean-up stage removes Docker images from previous builds that are no longer needed.

KPMG has added also SonarQube (`https://www.sonarqube.org/`) to their testing pipeline. It is a product that tracks code quality over time, by running a set of tests for each commit and generating an overall score. Several languages are supported: for Python the tests are performed by pylint.

3 Integration with Crystal

The IOStack project proposes an integrated vision from the top, the analytic tools users, to the bottom, the storage services. Zoe is near the top of this stack and needs to communicate with the lower layers.

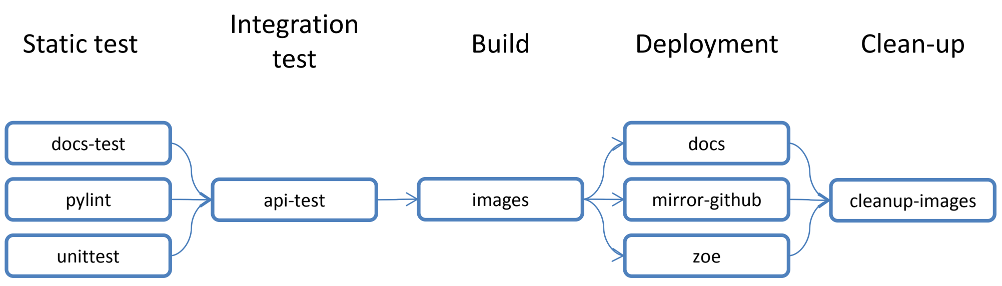


Figure 4: The CI pipeline implemented with GitLab at Eurecom. Each push to the master branch of the public repository goes through all stages.

The ZApp description of Zoe has been augmented with new fields that help Crystal in identifying the correct policy to apply at the storage that will be involved.

Whenever a new user starts a ZApp, thereby creating a new execution, Zoe sends via RabbitMQ a message containing the following information:

- user / tenant ID
- high-level policy to apply (gold, silver or bronze, for example)
- which storage system is going to be used by the execution

Other information that Zoe can provide include:

- the position of executions in the Zoe scheduling queue, to give Crystal a forewarning of when the execution is going to start
- more specific information about which parts of the storage are going to be used (specific files and objects)

At this time, testing to understand the impact and usefulness of each piece of information is undergoing.

For more information about Crystal, refer to deliverable 2.4.

4 Status of the open source project

Zoe Analytics has been an open source project since the beginning. In 2017 the project has started seeing external contributions from KPMG, that pushed for important new feature developed entirely with their own resources.

Air France/KLM has a long history of "invisible" collaboration with the project. Because of very strict internal policies, employees are not permitted to contribute directly to open source projects. A process to change these policies has been started. Currently we receive feedback on Zoe via email and via periodic meetings at Eurecom and in Air France Sophia Antipolis. Zoe is known at the highest levels of the company and it has been chosen over other solutions from IBM and Microsoft. Air France data scientists are enthusiastic about the system and the way it lets them work from day to day.

We are also establishing internal rules that all contributors wishing to participate need to adhere to:

- Fixed release schedule, one release every three months in March, June, September and December
- A mailing for internal communication

- Use of the public GitHub issue tracker to coordinate all bug reports and feature requests
- Tests that need to pass for contributions to be accepted and merged
- Coding style and code quality metrics
- A public road map

A possibility we are investigating is pushing Zoe Analytics to become an Apache project, under the umbrella of the Apache Software Foundation[3].

5 Planned releases and future road map

At the time of this writing, the 2017.09 release has been published, containing mainly the ZApp shop and the new web interface.

During the 2017.12 release cycle, we will concentrate our efforts on:

- Affinity and anti-affinity: when deploying services, Zoe should be able to instruct the back-end to run them near each other (on the same host, affinity) or far (on different hosts, anti-affinity). We have measured that the placement of services impact has crucial implications for performance, for example for TensorFlow applications.
- Alternative back-end: Zoe, historically, has supported Docker Swarm. Docker is in the process of deprecating it, so we need to find a valid alternative and develop a Zoe back-end for it.

We have also research activities, in progress, that, once complete, will be transformed into production code and released as part of the open source project. They include the preemption and dynamic allocation work that is presented in the next part of this document.

[3]`http://www.apache.org/foundation/`

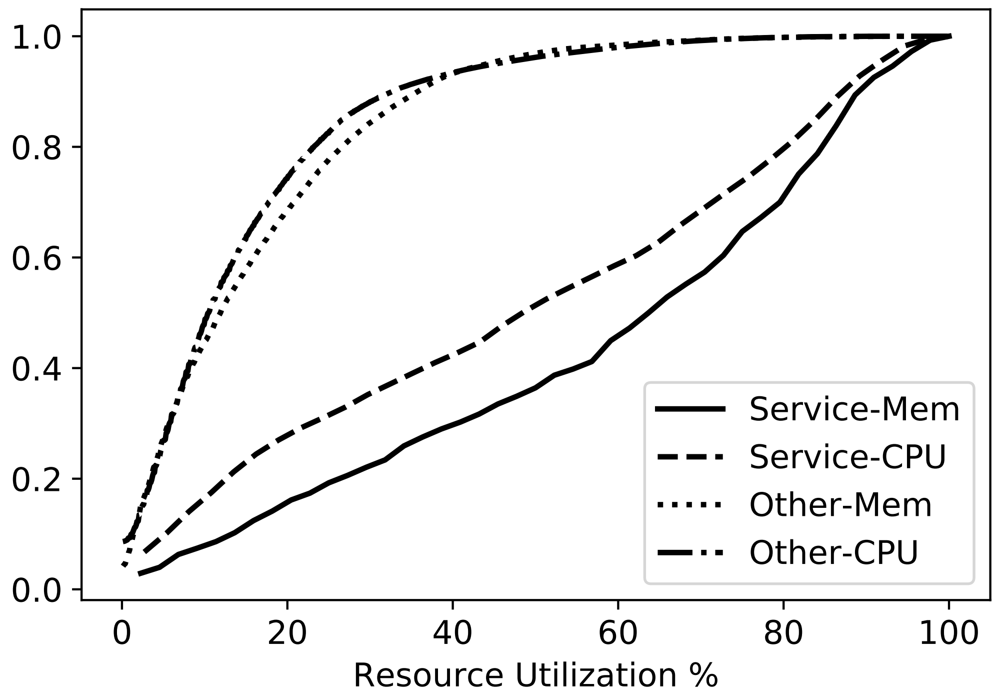


Figure 5: Resource Utilization from Google cluster [11, 10].

Part II
Dynamic resource allocation scheduler

6 Introduction

Data-center efficiency is a key issue that has attracted a vast amount of research [1, 2, 3, 4, 5, 6, 7]. Recently, the cloud computing paradigm, both in its public and private forms, fueled the proliferation of a wide array of resource management mechanisms [4, 5, 8, 9] aiming at an efficient operating point, where cluster resources are fully utilized. Despite such efforts, data-center resources go often under utilized, as shown in many recent traces from large-scale production deployments [10, 11]. Figure 5 illustrates resource utilization in a operational cluster at Google, for a mixed workload of production services and batch applications; in most cases ($\sim$ 80%) resource utilization is less than the 40% of resources allocated for batch applications, and less than the 80% for service applications.

Current approaches that address efficiency requirements fall in two broad categories. There are those that aim at steering tenants' behavior through the design of carefully engineered incentive mechanisms, which are largely adopted by public cloud providers [1]. In this case, tenants are endowed with the task of optimizing their cost to operate their applications, whereas providers operate on prices to steer the allocation of idle resources. Alternatively, other approaches opt to operate at the system level, and propose mechanisms that gauge resource allocation based on information on resource reservations [12, 3, 9, 4, 5, 8, 13].

In general, the ultimate goal of current approaches is to render the concept of resource reservation obsolete, and either let tenants reason in terms of value and cost [1], or let the system determine how to avoid wasting precious and costly resources, especially when the latter are scarce and entail application queuing in the scheduler.

Although the concepts we present in this work are widely applicable to a general concept of cloud applications – which include for example long-running, latency sensitive production services – we apply them to the exponentially growing body of data-intensive applications that use distributed frameworks such as Apache Spark [14], Google TensorFlow [15], and similar, to accomplish tasks such as data transformation and cleaning, large-scale machine learning, and scientific applications.

6.1 Reservation centric resource allocation

In most private or public cloud systems, users gain access to computing resources by specifying the amount required to run their application, in the form of a reservation request. Upon receiving a request, the cluster **scheduler** decides which application to serve based on the scheduling policy the provider implements (e.g.; First-In-First-Out (FIFO), Shortest Job First (SJF)). Cluster schedulers operate according to several variants of objective functions, the most common being the *(i)* average turnaround time and *(ii)* makespan. The first metric accounts for the average time requests spend in the system (queuing and execution times). The second metric considers the time required by the system to complete the aggregate workload of the requests received. Optimizing for such objectives translates in high system responsiveness, which is truly desirable from the tenant perspective.

Cluster schedulers use a resource management mechanism that is in charge of resource provisioning and management. Given a **resource request**, the resource manager determines its admission in the cluster based on its **reservation** information.[4] An admitted request triggers a **resource allocation** procedure, which eventually [5] concludes with reserved resources being exclusively allocated to the request.

In most system implementations, the concept of reservation and allocation coincide, although neither is representative of the true **resource utilization** a request might induce on the system. In fact, resource utilization is generally not constant throughout a request lifetime, and fluctuates according to application behavior [16].

The main consequence for current cloud environments is that reservation requests are **engineered to cope with peak resource demands** of an application. This is a key factor that induces poor system utilization, and ultimately, efficiency. This condition is exacerbated by coarse-grained reservation specifications, which is a common practice in public cloud providers: instance flavors exhibit discrete gaps in terms of resource units. In fact, picking the right configuration for cloud applications (and in particular for the "big data" applications we consider in this work) is a daunting task [17], which requires sophisticated optimization mechanisms going beyond human tuning abilities. As a simple example, consider a provider offering two flavors of Virtual Machine (VM) (or Containers), with 32GB or 64GB of main memory. An application requiring e.g. 33GB of memory will necessarily need a reservation request for the latter flavor, resulting in almost 50% of resource slack.

Thus, mechanisms to reduce **resource slack**, which is defined as the difference between resource allocation and resource utilization, are truly needed, for they can prevent clusters from denying admission for new requests which would queue up, while spare capacity goes unused.

6.2 Problem Statement

In this work, we focus on distributed analytics applications and consider systems such as [18, 19], which allow to treat distributed data processing frameworks [14, 15] as an individual collection of resource requests. Indeed, such frameworks are composed by several **components**, that are characterized by either a **core** or **elastic** nature. Core components are compulsory for a framework to produce useful work; elastic components, instead, optionally contribute to a job, e.g. by decreasing its runtime. Consider, for example, a framework such as Apache Spark. To produce work, it needs some core components: a controller, a master, and one worker. Any additional worker is an elastic component. An application with core components only is called **rigid**, whereas applications with a mix of core and elastic components is called **elastic**. Note that two components, albeit running replicas of the same function but on different data, generally exhibit different resource utilization patterns.

With this context in mind, in this work we study the problem of cluster efficiency by minimizing the resource slack.[5] Recall that slack arises because of reservation centric resource management,

[4]In our prose, we neglect several important technical details that are however irrelevant to our point, such as quota management, security aspects, and concurrency control, to name a few.

[5]By abuse of notation, in this context, the term minimize does not involve the theoretical task of a formal proof.

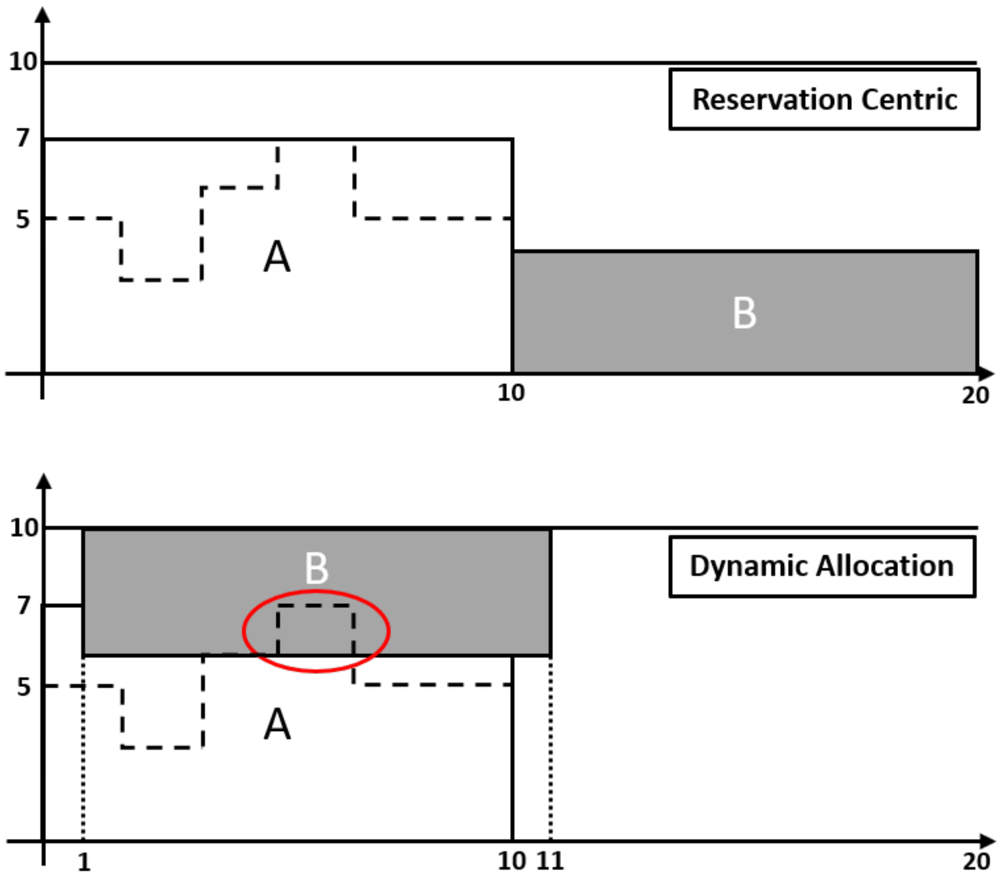


Figure 6: Illustrative examples of request scheduling: (top) without and (bottom) with resource reclamation.

which matches allocation to reservation. In abstract terms, we approach the problem by studying ways of modulating resource allocation to follow utilization as closely as possible. A simple toy example nicely illustrates the intricacies of the problem.

Toy Example. Consider the simple, one dimensional packing problem we illustrate in Fig. 6. We assume a cluster with a finite capacity of 10 resource units (RAM, in this case), which receives two application requests: *A* and *B*, which both arrive at the same time and have an execution time of 10 seconds. *A* is over-provisioned: it reserves 7 units, but its utilization fluctuates. Instead, *B* is well-calibrated and reserves and uses 4 units.

We assume a simple FIFO policy which breaks ties randomly. In this situation, current reservation centric approaches would admit a single application. In Fig. 6 (top) we illustrate the case in which application A is scheduled first, whereas application B sits in the scheduler queue. Indeed, reserved resources are locked by A and cannot be granted to B.

Instead, as shown in Fig. 6 (bottom), by mapping resource allocation to real utilization, thus using resource allocation and not reservation to admit new requests, B can be scheduled and produce work. This contributes to both shorter average turnaround times and makespan. However, resource utilization dynamics introduce an issue: by following too closely utilization, sudden spikes could wreak havoc in the system, inducing the underlying operating system to manage "self-inflicted" memory shortages in an application-agnostic manner [4]. Due to the seemingly unique ways in which utilization can vary, matching resource allocation to utilization for each application can only be approximate, and approximation errors will inevitably induce application failures.

Careful engineering would suggest to introduce a buffer that will act as "safe-guard" to the approximation, such that resources are allocated in a way to absorb unexpected demand. This last consideration suggests that the problem we study revolves around two key aspects:

1. *Resource allocation should follow utilization dynamics, to minimize slack.*
2. *Resource allocation should allow for unexpected demand peaks, to prevent application failures.*

Unfortunately, these two aspects are at odds: the larger the tolerance of the system to approximation errors, the smaller the gains in terms of system efficiency.

Contributions. In this work we present our design of a data-driven scheduling mechanism that improves cluster utilization, thus decreasing the average turnaround time, while preventing application failures due to resource contention. Our approach monitors resource utilization and relies on sophisticated resource demand forecasting to modulate allocated resources such as they approximate utilization patterns well.

Our experiments, that we conduct on a system simulator as well as a system implementation using real-life data-center traces, indicate substantial gains over existing alternatives: our approach contributes to more efficient and responsive clusters, while carefully controlling the number of application failures due to the approximate nature of our control approach. In summary, the contributions we present in this work are as follows:

- We present the system design for a dynamic resource allocation mechanism, which can be generally applied to existing cluster management frameworks. In this work, we target a specific family of analytic application schedulers, and materialize our ideas for such schedulers.
- We introduce a novel component for accurate forecasting of resource utilization, featuring a probabilistic treatment that allows quantification of uncertainty. Confidence information is used to steer system parameters to safeguard against unexpected resource demand peaks.
- We perform an extensive simulation campaign using publicly available production traces from Google data-centers. We compare our approach to that of Borg, and discuss about the trade-off that an optimistic vs. a pessimistic approach to application preemption entails.
- We present the design of a system implementation of our idea, that we use in an academic cluster serving students and researchers. Our preliminary results indicate substantial improvements in terms of efficiency, which translate in a system capable of ingesting a heavier workload with the same number of machines.

The remainder is organized as follows. In Section 7 we review related works, while in Section 8 we present our system design. We validate our ideas using a simulation campaign in Section 9, present our system implementation in Section 10 and its evaluation in Section 11.

7 Related Work

Dynamic resource allocation has been approached in many different ways in the literature [4, 20, 21, 6, 7, 22, 23, 12, 24, 3, 25, 26, 27, 1, 2].

The authors in [20, 21] present a solution called KOALA-F that is based on a feedback control loop that requires every *framework* running inside the cluster to periodically send information to the scheduler. Every framework must be enhanced so that it can talk directly to KOALA-F, in order to transmit information about their metrics in form of color: red, yellow and green. KOALA-F will allocate or deallocate resources to that specific framework based on that color. Red means that the framework is struggling due to lack of resources, yellow is the good state, while green means that there are more resources than needed. In our work we operate on the application rather than on the framework level. In addition, our solution does not require such instrumentation; we are completely agnostic to the application that is running and we use general metrics in order to dynamically reallocate the resources of the running applications.

The authors in [6] introduce a type of scheduler that is reservation-based. They propose a reservation definition language (RDL) that allows users to declaratively reserve access to cluster resources. They formalize the planning of current and future cluster resources as a Mixed-Integer Linear Programming (MILP) problem and they integrate their work in YARN [28]. In our work, we avoid delegating this task to users by asking them to specify such information; most of the time the users themselves have no idea of how their applications will behave.

Some other works like [1, 26] propose to address the problem with economics principles. In particular, in [26] the authors build a pricing model that enables infrastructure providers to incentivize their tenants to use graceful degradation, a self-adaptation technique originally designed for constructing robust services that survive resource shortages. The authors in [1], present a framework for scheduling and pricing cloud resources, aimed at increasing the efficiency of cloud resources usage by allocating resources according to economic principles. However, they use an overbooking mechanisms to achieve their goals, which is a solution prone to application failures when the resources are not sufficient.

Finally, works like [3, 25, 7, 22, 23, 12, 24], focus either on resource placement or on meeting Service Level Objective (SLO) requirements. In the first case they relate the problem to a packing problem and try to optimize it, while in the second case they leverage the elasticity of some frameworks and they give more resources to applications that are falling behind on their SLO.

Albeit all these works are valid and propose their own vision of the problem, they all share one element: they focus on "time sharable" resources, like the CPU, rather than resources that are "finite" like Memory[6]. In particular, the outcome of a process that does not have enough CPUs will be a slower runtime, while in the case where the amount of Memory is insufficient, the application will fail.

The only work that addresses "finite" resources is [4], where Borg is introduced: a large-scale cluster management system used in Google. Among other features, Borg features a resource reclamation system that seizes unused resources and offer them to other applications. Despite the lack of details of their design and implementation, the authors study the impact of wrong memory reallocation on running tasks, which causes resource contention: the OS enters a special state to kill processes that are Out Of Memory (OOM). The authors present different levels of "rigidity" for their reclamation system (baseline, medium and aggressive) and show both the benefit and the number of OOMs events for each of them. They conclude by stating that they accept the trade-off obtained by the medium setting. In our work, we seek to gain control over the OS and minimize such events while maximizing the resource utilization.

8 System Design

For the purpose of clarifying our approach, we present our design as an instance working in conjunction with an application scheduler such as [18, 19]. However, it is straightforward to apply our approach to other cluster management back-ends such as Docker Swarm [29] and Kubernetes [30], or alternative schedulers [31]. Figure 7 outlines our design, which consists of four modules which operate on a given cluster management back-end: an application scheduler, a resource monitor, a forecasting component and an actuator.

A bird's view on the operation of our system is as follows. Application execution requests take the form of resource reservations, which are submitted to the application scheduler. The application scheduler admits the request based on reservation information alone, and instructs the back-end to provision and allocate the necessary resources. The resource monitor collects information about both allocated and used resources, which are fed to the system state and the forecasting component respectively. The actuator module gauges resource allocation to match predicted utilization patterns, and is responsible for the preemption of running applications in case of sudden peaks in resource demand. The modified resource allocation is reflected in the system state, which triggers new scheduling de-

[6]On the one hand, a resource is considered "time sharable" when the Operating System (OS) is able to use time sharing for scheduling it, and thus it does not impose limits on its availability. On the other hand, "finite" resources are those that cannot be sliced in time and thus cannot be effectively shared by multiple processes.

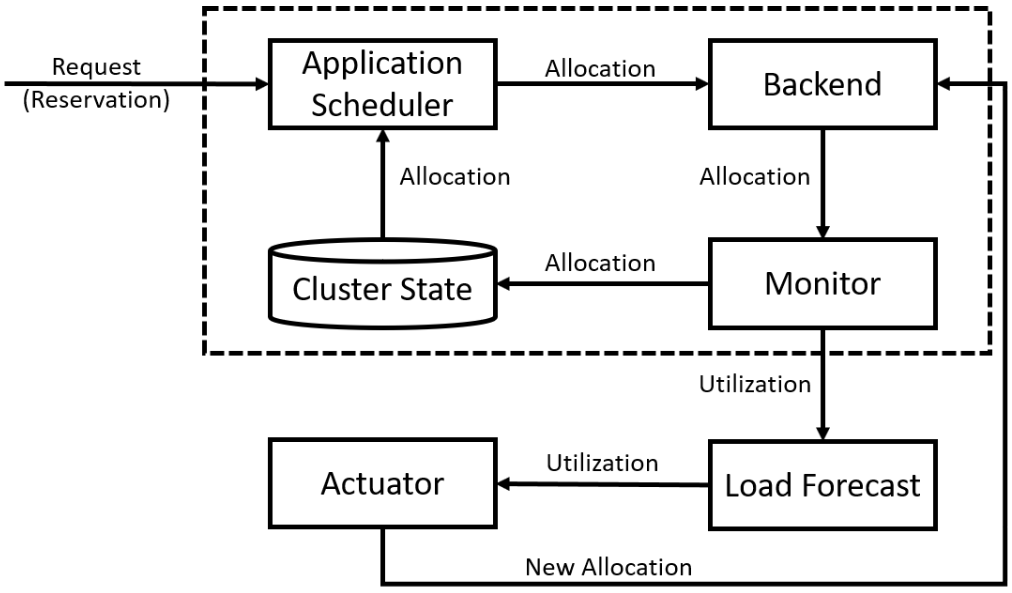


Figure 7: System design.

cisions. Next, we describe in more detail the components that materialize our ideas.

Resource monitor. It collects information about resource allocation and utilization from every component of every running application. This happens at regular time intervals: higher frequencies provide more accurate views, but generate more data. Our goal is to minimize intrusiveness by being application agnostic: for this reason we do not instrument applications (as done for example in [20]), but take standard metrics (CPU, memory, etc) as they are seen by the OS.

Forecast module. Its task is to anticipate the resource utilization of every application component. We advocate for a Bayesian approach to predict resource utilization and quantify the uncertainty of these predictions. A more detailed exposition of the Machine Learning (ML) methodology we employ can be found in Section 8.1.

Independently of the prediction methodology, we identify an important parameter that defines the "lookahead" of the prediction, which we call the prediction window ω. Naturally, a large ω induces less confidence in future utilization predictions. However, a small ω could potentially mask a future demand peak. To mitigate such effects, the predictor returns the maximum value of resource utilization in the given future time window ω.

Actuator module. This module uses utilization forecasts to adjust the resources allocated to every component of a running application. We anticipate prediction errors, thus we compensate using a "safe-guard" buffer of size β to artificially increase (that is, to force over estimation) predicted peak resource utilization. The buffer size β is a function of the uncertainty quantified by the forecasting module.

Additionally, the actuator is in charge of preemption. Preemption policies can either be optimistic [5, 4] or strict (pessimistic). We advocate for a strict policy, to avoid delegating application preemption to the OS, which manages resource shortage (such as OOM problems) in an application agnostic way. A detailed exposition of the preemption policy can be found in Section 8.2.

8.1 Time-Series Predictors

Recently, machine learning inspired scheduling methodologies have gained popularity in the field of cloud computing. An approach that relies on the clustering of historical data to increase resource utilization has appeared in [32]. The Ernest [33] framework makes use of a parametric model to predict the running time of a given job on a specified hardware configuration. Predicting resource demand is also a key aspect of the ERA framework [1]. In our work, we pay particular attention to load forecasting: we employ a flexible non-parametric methodology that makes no modeling assumptions (e.g. linearity) and that offers uncertainty quantification.

We have seen that the forecast module is responsible for making predictions about the future resource utilization. Regarding a single application, we assume data is available in the form of a time series that reflects the memory usage of its components across time. We seek to discover patterns of memory usage that allow reasoning about our expectations regarding the future state of the system utilization.

We have opted not to follow traditional time series analysis approaches [34], as it is of great importance to quantify the level of uncertainty associated with each prediction. Predictive errors are unavoidable, thus predictive confidence can be used as a guide to adjust the degree of adaptiveness to the anticipated workload. Intuitively, a prediction with low confidence implies that the actuator module should be more conservative regarding changes in resource allocation.

For this reason we use a scheme that relies on Gaussian Process (GP) regression [35], which is a Bayesian non-parametric regression method with many attractive features. Bayesian approaches control model complexity and thus avoid problems such as over-fitting (i.e. constructing overly complicated models that generalize poorly) [36]. Moreover, GPs offer a sensible framework for tuning their hyper parameters, through evidence maximization, that does not require cross-validation approaches which are typically more expensive and unpractical in the context of our work. Finally, the output of a GP regression model is a predictive distribution, rather than a single prediction, which allows reasoning about uncertainty.

More formally, a GP is a collection of random variables, any finite subset of which follows a multivariate Gaussian distribution. A GP is essentially a generalization of a Gaussian distribution in the space of functions, and it is uniquely characterized by a mean function $\mu(x)$ and a covariance function $k(x, x')$. In general, the task of regression is to infer a latent function f given a set of observations $\mathbf{X}$ and $\mathbf{y}$, where $\mathbf{X}$ is a matrix containing the observation inputs and $\mathbf{y}$ the corresponding vector of outputs. In GP regression in particular, this is achieved by considering a prior GP distribution over the space of functions, and then by evaluating the posterior function distribution in light of the observed data $\mathbf{X}$ and $\mathbf{y}$, as specified by the Bayes rule. It is common practice to consider a zero mean function for the prior, i.e. $\mu(x) = 0$, as it is sensible in most cases to subtract the mean from the input data. The prior covariance value between inputs x and x' is determined by the kernel $k(x, x')$. Under the assumption of a Gaussian likelihood, that is for any input-output pair we have: $y \sim N(f(x), \sigma^2)$, the posterior is also a GP whose mean and covariance can be calculated analytically. For the mean and covariance function of the posterior GP given a zero-mean prior we have:

$$\mathrm{E}[f(x) \mid \mathbf{X}] = k(x, \mathbf{X})(k(\mathbf{X}, \mathbf{X}) + \sigma^2)^{-1}\mathbf{y} \tag{1}$$

$$\begin{aligned}\mathrm{Var}[f(x) \mid \mathbf{X}] = {} & k(x, x') \\ & - k(x, \mathbf{X})(k(\mathbf{X}, \mathbf{X}) + \sigma^2)^{-1}k(\mathbf{X}, x)\end{aligned} \tag{2}$$

The predicted value at a new point will be the expectation under the posterior distribution, and the posterior variance quantifies the uncertainty about the prediction.

Different choices for the prior covariance function will result in different predictive distributions. In general, the optimal choice for the kernel is problem-dependent; it is the responsibility of the modeler to use a kernel that respects the peculiarities of the data. For example, the squared-exponential kernel is a popular choice in many contexts where a smooth function is desired, as it assigns non-zero prior probability to smooth functions only. In this work, we aspire to capture functions that are not necessarily differentiable, therefore the exponential kernel is a more reasonable choice:

$$k(x, x') = \alpha \exp\left(-\frac{|x - x'|}{l}\right) \tag{3}$$

where α and l are the amplitude and length-scale hyper-parameters of the kernel. We shall adopt the common strategy to optimize α and l by means of evidence maximization.

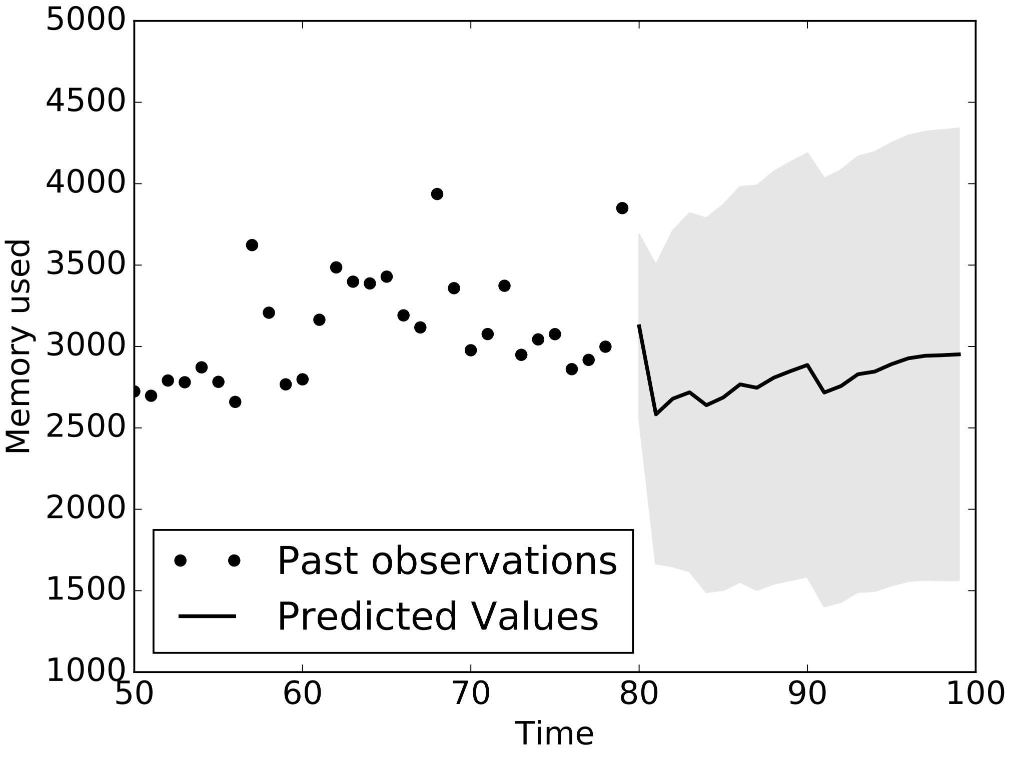


Figure 8: Predicted workload for a sample time series.

However, we are not ready to apply GP regression just yet; the time series data in their original form are not well-suited for this kind of treatment. A GP model transfers information across points that are considered similar, as this is reflected in the choice of the covariance function. If we assume that the inputs $\mathbf{X}$ solely consist of the recorded times, then similarity is only a matter of temporal locality, which is not optimal practice if the aim is to predict sudden changes of behavior throughout the course of a time series.

Hence, we resort to the definition of a kernel that relies on the observation history. It is implicitly assumed that if two sequences of observations are similar, then they must have been caused by the same "hidden" background processes; it is reasonable then to extrapolate and predict that the future observations will be similar as well. Such a history-dependent kernel can be easily constructed by transforming the data in an appropriate way. Consider a history window of size h, the training instances will be vectors of the form:

$$\tilde{\mathbf{x}}_n = [x_n, y_{n-h}, \ldots, y_{n-1}]^\top$$

where x_n is the n-th recorded time. Therefore, the history-dependent kernel is implemented by applying a typical exponential kernel on the transformed inputs:

$$k_h(x, x') = k(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}') \tag{4}$$

Two different inputs x and x' will be similar if they have a similar history, or equivalently, if the h preceded inputs have similar outputs. Note that we have kept the recorded times x_n along with the history, thus we do not completely ignore locality in the original input space.

From a practical perspective, the load forecast component performs the following steps every time new data is available from the monitor module:

1. New resource utilization data is appended to the collection of observations $\mathbf{X}, \mathbf{y}$.

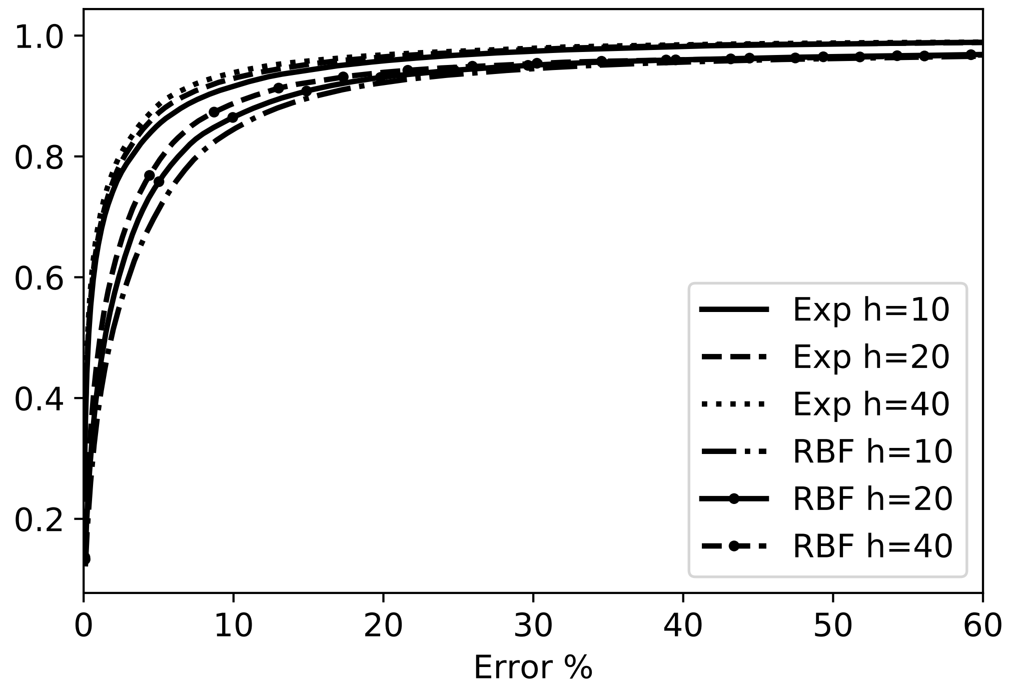


Figure 9: Error distribution of predicted workload for a collection of time series in our academic cluster.

2. Using a history-dependent kernel $k_h(x, x')$, Equations (1) and (2) are used to make predictions for a window ω based on observations $\mathbf{X}, \mathbf{y}$.

An example of regression with a history-dependent kernel can be seen in Figure 8. The black dots represent the past utilization observations up to time 80; at this point GP regression is employed to produce estimations of the utilization for the next 20 time units. The shaded areas represents the 95% confidence interval for the predicted values.

We have to note that the regression step can be computationally expensive; Equations (1) and (2) involve a matrix inversion (for $k(\mathbf{X}, \mathbf{X}) + \sigma^2$), which is an operation of cubic complexity, and the set of observations $\mathbf{X}, \mathbf{y}$ will grow indefinitely during the lifetime of the system. There is a plethora of methodologies on sparse GPs in the literature [37, 38, 39, 40], that can be used to reduce the complexity of regression. In terms of this work, we adopt the simple solution of restricting the history $\mathbf{X}, \mathbf{y}$ to the N latest observations, thus keeping the model relatively small.

We have prioritized on simplicity because the system will have to keep many predictive models concurrently: we have a regression model for each component of every running application in the system. In the experiment summarized in Figure 9, we see that our design choices are enough to capture recent trends in data. We have applied our scheme on a dataset consisting of approximately 6000 time series that monitor the memory usage for applications in our academic cluster. Figure 9 summarizes the empirical distribution function for the predictive errors observed across the entire dataset. We have experimented with different values for the size h of the history window; as seen in Figure 9, increasing the value of h results in smaller prediction errors.

Also, for the implementation of the history-dependent kernel as described in (4), we have experimented both with the exponential and the squared-exponential (also known as RBF in the literature) functions. Figure 9 implies that the exponential implementation outperforms the RBF choice in terms of prediction error. This result is in line with our expectations, as the time series in question are typically not smooth. For the experiments of Section 9 and Section 10, we consider the exponential implementation of the history-dependent kernel only.

8.2 Addressing Resource Conflicts

We now delve into the details of the actuator module, and focus on its approach to application preemption. Recall that the actuator varies allocated resources as a function of predicted utilization: upon a spike, it needs to redeem resources from running applications and dedicate them to those experiencing a peak demand, for otherwise such applications are doomed to fail. Thus, the goal of the preemption policy is to decide how to redistribute resources, by operating on running applications and their components. Such a policy can optionally account for application priorities, as dictated by the application scheduler. Note that, irrespectively of the chosen preemption policy, a failed application is resubmitted to the application scheduler, making sure it enters the scheduling queue in a position commensurate to its original priority.

Algorithm 1: Host-level resource allocation and pessimistic preemption policy.

```
Data: $\mathcal{J}_{\mathcal{H}} \leftarrow$ jobs on host $\mathcal{H}$
1  $J \leftarrow$ SORT($policy$, $\mathcal{J}_{\mathcal{H}}$)
2  $resFree \leftarrow host.resources$
3  foreach $req \in J$ do
4      $C \leftarrow req.CoreCpts$
5      $tmpResFree \leftarrow resFree - \sum_{j \in C} C_j.futureRes$
6      if $tmpResFree < 0$ then
7          INSERT($req$, $K$)
8      else
9          $resFree \leftarrow tmpResFree$
10         $E \leftarrow$ SORT($timeAlive$, $req.ElasticCpts$)
11         forall the $e \in E$ do
12             if $resFree - e.futureRes \leq 0$ then
13                 INSERT($e$, $K_E$)
14             else
15                 $resFree \leftarrow resFree - e.futureRes$
16 foreach $req \in K$ do
17     PREEMPTREQ($req$)
18 foreach $e \in K_E$ do
19     PREEMPCOMPONENT($e$)
20 foreach $c \in \bigcup_{j \in \mathcal{J}}(req.CoreCpts \cup req.ElasticCpts)$ do
21     RESIZECOMPONENT($c$)
```

Recent works (for example [4]) advocate for an optimistic preemption policy, which is reminiscent of optimistic concurrency control [5]: resources are redeemed without taking explicit actions to manage the consequences of resource redistribution. Either explicit (and often manually set) priorities determine the fate of running applications, or the task is left to the OS.

Here, we present an alternative preemption policy, which we call pessimistic. Our goal is to control which application should be partially or fully preempted, while minimizing the amount of work that is wasted by preemption.[7]

Algorithm 1 present the details of our resource allocation and pessimistic preemption policy: the algorithm operates at the host, rather than at the cluster level. The algorithm starts by sorting (line 1) the applications running on each host according to the application scheduler policy (e.g.; FIFO, SJF, etc.). Then it simulates (lines 2-15) an allocation by trying to maximize the resource utilization while minimizing the number of running applications. In particular, it first allocates the core (line 5)

[7]We consider preemption primitives such as a `kill` operation, which inevitably wastes work. Component or application suspension and migration are outside the scope of this work. Alternatively, it would be interesting to consider techniques such as [41], which would allow a graceful management of memory pressure.

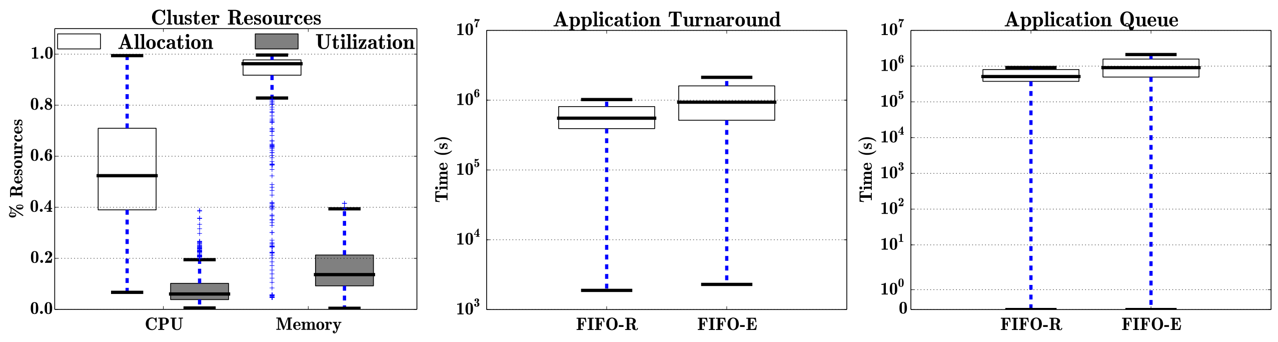


Figure 10: Results obtained from a system with no dynamic reallocation, called baseline.

components and then all elastic components[8] that fit in the host (lines 11-15). If there is free space left, it moves to the next application in line and repeats. The algorithm continues the loop at line 3 until the host is full. Resource allocation is determined, and we can turn our attention to preemption. Core components that no longer fit a host entail full application preemption (line 16). Also elastic components can be preempted (line 18), inducing only a partial application preemption. Finally, the algorithm resizes (line 20) the components according to the simulated system, following the computed allocations and preemption.

Since application progress can be lost, our algorithm allocates the core components of an application, then moves to the elastic components by giving more priority to the ones that have been living in the cluster for a longer time (line 10). Components recently scheduled are the best candidates for preemption, because they have likely produced less work.

9 Numerical Evaluation

9.1 Methodology

We evaluate our algorithm using an event-based, trace-driven discrete simulator which was developed to study the scheduler Omega [5], and was later extended in [18] to study application schedulers. We have made additional extensions[9] to support the concepts of this work.

In particular we have implemented two alternatives for time series prediction. We first consider an ideal setup with an oracle with perfect information about future workload: this allows to determine an upper bound of the performance gains achieved by our approach. We then use GP regression described in Section 8.1, with the intention to investigate the impact of prediction errors on system performance.

In this work, we are mainly interested in focusing on memory resources, which are much harder to manipulate than computational (CPU) resources. We use publicly available traces [11, 42, 10, 43], and generate a workload by sampling from the empirical distributions computed from such traces. We simulate both rigid (e.g. TensorFlow) and elastic (e.g. Apache Spark) variants of batch applications, which use the label **R** and **E**, respectively. Applications are assigned a number of components ranging from a few to tens of thousands. The resource requirements (in terms of memory) of application components follow that of the input traces, ranging from a few MB of memory to a few dozens of GB, and up to 6 cores. Application runtime is generated according to the input traces, and ranges from a few dozens of seconds to several weeks (of simulated time). Application inter-arrival times are drawn from the empirical distributions of the input traces, and exhibit a bi-modal distribution with fast-paced bursts, as well as longer intervals between application submissions.

We simulate a cluster consisting of 100 homogeneous machines, each with 32 cores and 128GB of memory. All results shown here include 10 simulation runs, for a total of roughly 3 months of simulation time for each run.

[8]In the case the application scheduler does not support the distinction between core and elastic components, all components are treated as core.

[9]`https://github.com/DistributedSystemsGroup/cluster-scheduler-simulator`

The metrics we use to analyze the results include: *(i)* application **turnaround**, which allows reasoning about the scheduling objective function, *(ii)* **resource allocation**, measured as the percentage of CPU and memory the scheduler allocates to each application, *(iii)* **resource utilization**, measured as the percentage of CPU and memory that each application uses and *(iv)* application **failures**, which give us information about the aggressiveness of our approach. Additionally, we show **queuing times**, which are an important factor contributing to the turnaround time.

9.2 Simulation results

In this section we first show the baseline performance achieved by a reservation centric application scheduler. Then we move to the evaluation of our dynamic resource allocation mechanism: we first show ideal results obtained with an oracle, and compare them to a full-fledged mechanism that uses GP regression. In doing so, we also compare optimistic and pessimistic preemption policies.

Baseline. A reservation centric approach achieves the performance reported in Figure 10. On the left we show the difference in resource allocation compared to utilization of CPU and memory. This result reiterates on the low efficiency of reservation centric approaches: the median resource slack is large (40% for the CPU and 90% for the Memory). Untapped resources could be used to decrease queuing and consequently turnaround times, which are shown in Figure 10.

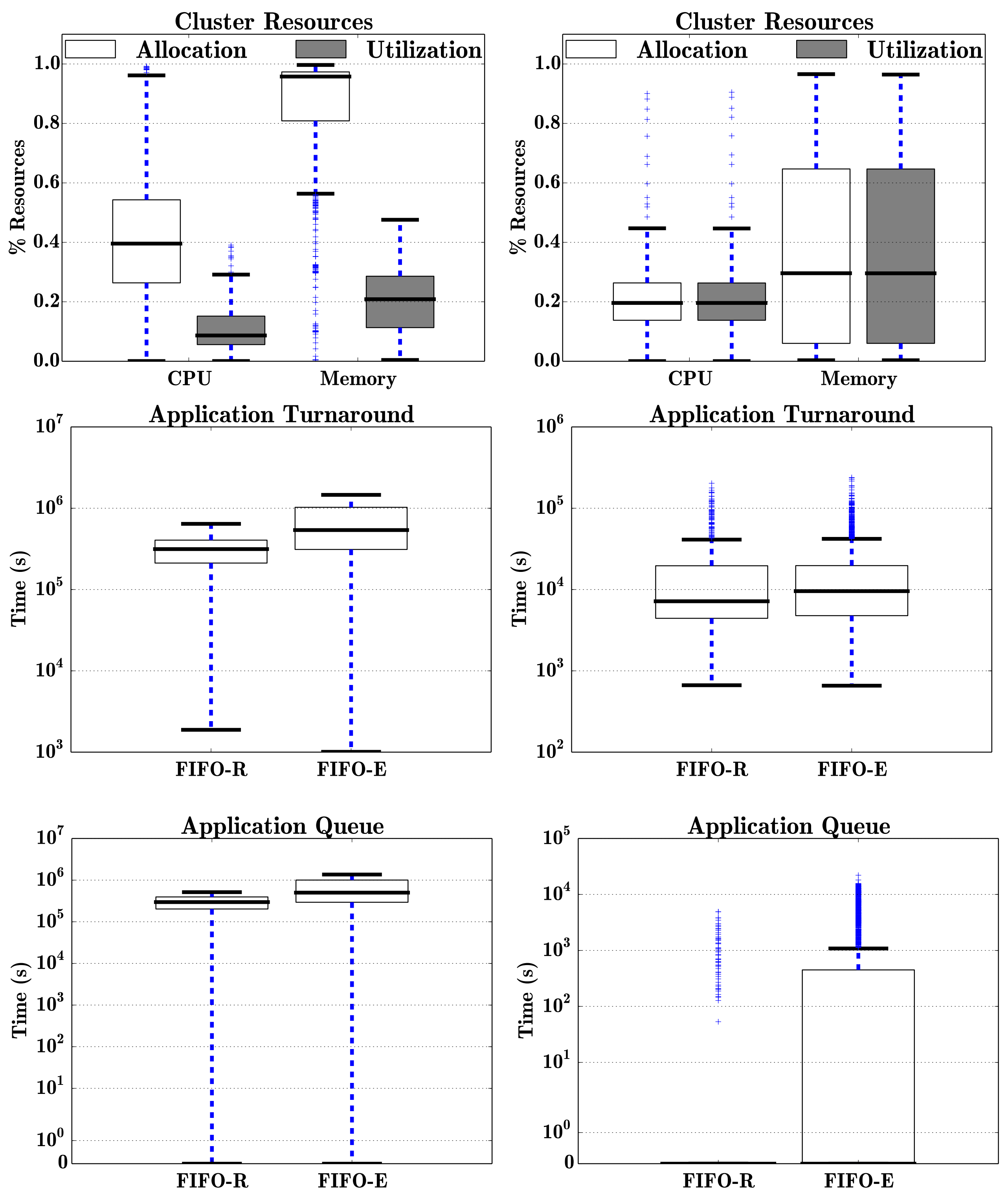
Oracle-based dynamic allocation. Next, we gloss over prediction errors induced by a real statistical model and consider an ideal scenario from the load forecasting point of view. Ultimately, our goal is to discern virtues and drawbacks of different preemption policies. Results are summarized in Figure 11: each row corresponds to cluster resources, application turnaround and queue times, whereas each column correspond to the optimistic or pessimistic preemption policy. Before proceeding further, we clarify that our simulator implements the concept of work lost when an application component is killed or crashes.

Overall, independently of the preemption policy, our results indicate that an ideal dynamic resource allocation mechanism brings substantial benefits in terms of all metrics we consider. Cluster efficiency improves because resource slack, computed as the difference between allocated and used resources, drastically shrinks as shown in Figure 11 (first row), and in comparison to Figure 10 (left). Then, turnaround times are notably smaller as shown in Figure 11 (second row), and in comparison to Figure 10 (left). Indeed, queuing times decrease, as shown in Figure 11 (third row), because the system can quickly ingest new applications, since resource allocation follows utilization more closely.

Figure 11 can now be used to compare optimistic versus pessimistic eviction policies, in absence of prediction errors. While both approaches improve over the baseline, the pessimistic policy we introduce in this work is consistently superior to the optimistic in all respects. As shown in Figure 11 (first row), the pessimistic policy induces our dynamic allocation mechanism to follow very closely application resource utilization: in this case, resource slack becomes negligibly small. This result explains why turnaround times, Figure 11 (second row), are orders of magnitude smaller with the pessimistic policy: by freeing up resources, the application scheduler is amened to trigger new executions, and consequently queuing times in Figure 11 (third row) are very small for a vast majority of applications. Furthermore, we compute the number of application failures: in case of the optimistic policy we record 36.63% application failures, whereas with the pessimistic policy no application fails. Indeed, with the optimistic policy, when two applications compete for resources and there are none left, the system will let one of the two fail. Instead, the pessimistic policy avoid failures through partial preemption, by freeing elastic resources first.

To summarize, this first set of results indicate that dynamic resource allocation is a viable approach in principle, and provide an upper bound to which a system with a real statistical modeling approach should tend to. Furthermore, our results indicate that, when factoring out intricate considerations about prediction errors, a strict preemption policy is superior to an optimistic one, because results in no application failures.

GP-based dynamic allocation. Next, we study the system behavior when using GP regression to



(a) Optimistic

(b) Pessimistic

Figure 11: Comparison between an optimistic vs pessimistic approach over different metrics using an oracle.

predict future resource utilization. As anticipated in Section 8, statistical models are prone to prediction errors, which we address using a "safe-guard" buffer β. A key feature of our approach is that β is a function of the uncertainty information produced by our Bayesian approach to regression. In practice, when the predictor outputs a future (peak) resource utilization, we adjust the value by

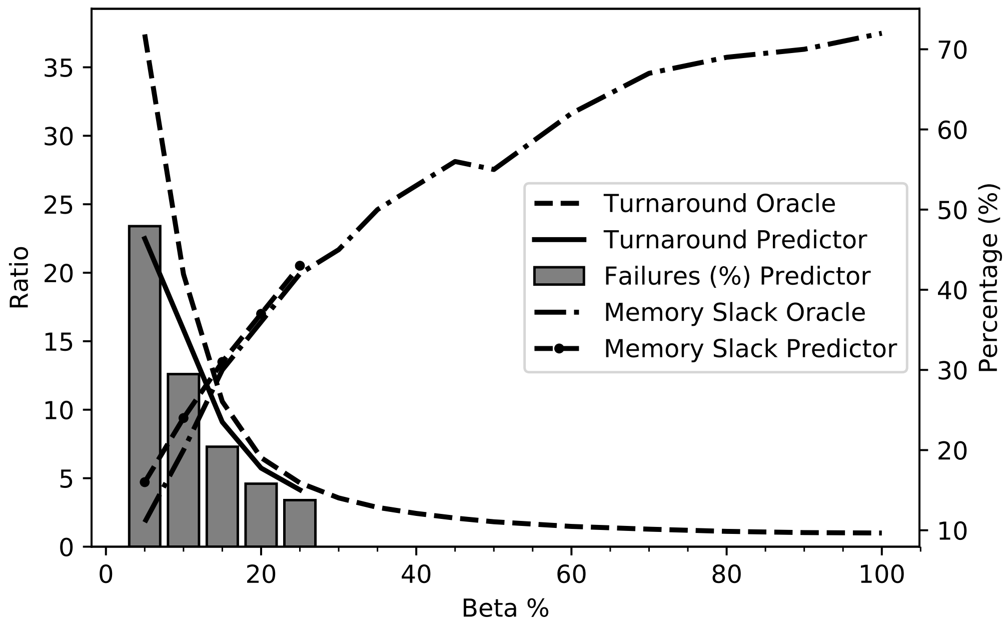


Figure 12: Impact of the buffer parameter β.

adding the buffer β.

Since β is an important parameter, we first focus on a detailed analysis of its role. Figure 12 reads as follows. On the x-axis we have β, which is the parameter we study. The first y-axis reports the ratio (higher values are preferred) between the average turnaround obtained by a dynamic mechanism vs. the baseline: we show both the upper bound obtained with an oracle, and the ratio obtained with GP regression, as a function of β. The second y-axis reports the memory resource slack computed as the difference between median resource allocation and utilization, both for an ideal system and for a GP-based system, as a percentage (lower values are preferred). Finally, we also report an histogram accounting for the percentage (lower values are preferred) of application failures when using a real system. This figure is also useful to overview the tension that exists between low values of slack, indicative of system efficiency, and a small number of application failures, as discussed in the introduction.

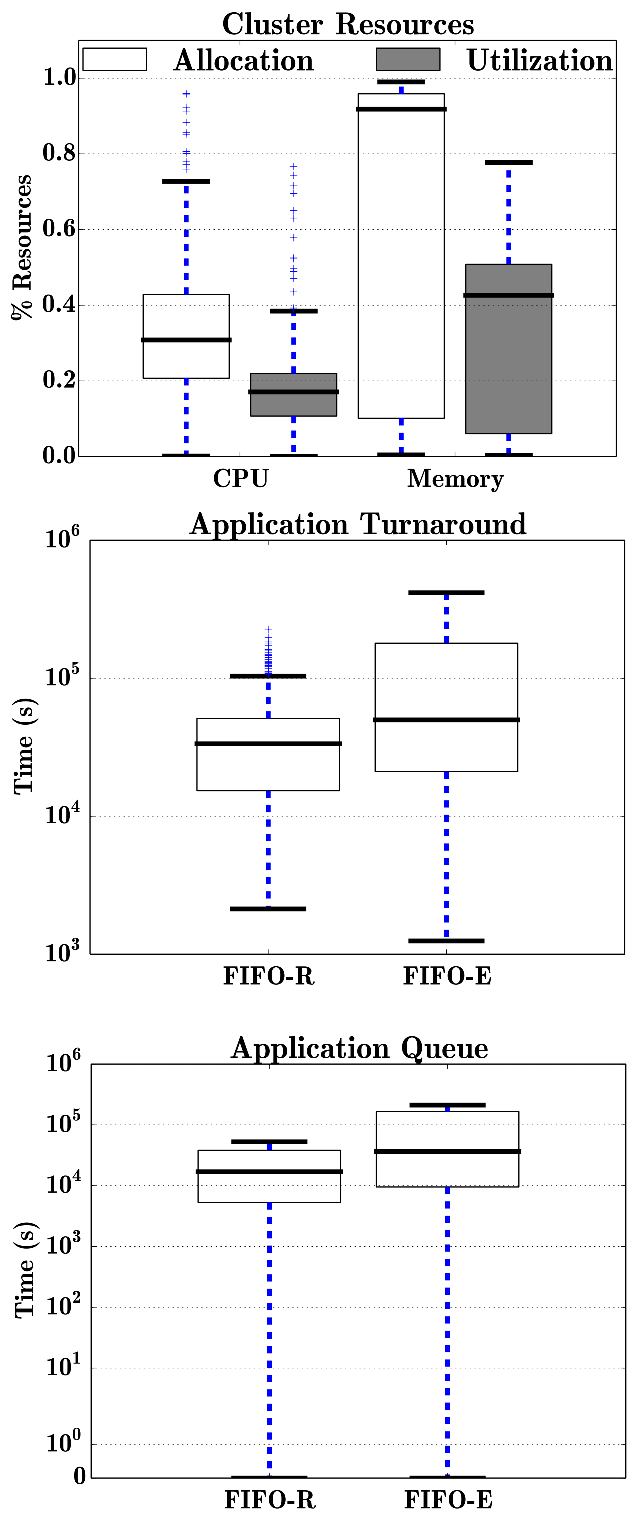
Figure 12 indicates that as β increases, the benefits in terms of improved turnaround due to dynamic allocation fade away. Indeed, aggressive "safe-guard" buffer sizes correspond to gradually disabling the dynamic allocation: $\beta = 100\%$ corresponds to the baseline system, and obtains a turnaround ratio of 1. As expected, as β increases, the resource slack also increases: indeed, resource allocation becomes more conservative, and matches, in the limit, resource reservations.

It is important to observe the behavior of application failures as a function of β: Figure 12 reports results for GP regression in conjunction with a pessimistic preemption policy. As β increases, the failure rate decreases, because dynamic reallocation is more conservative: irrespectively of prediction errors, a large "safe-guard" induces the system to prevent new applications to be admitted in the system, which in turn decreases opportunities of modulating allocation. Although failed application are resubmitted in the system, a good choice of β should prefer small failure rates.

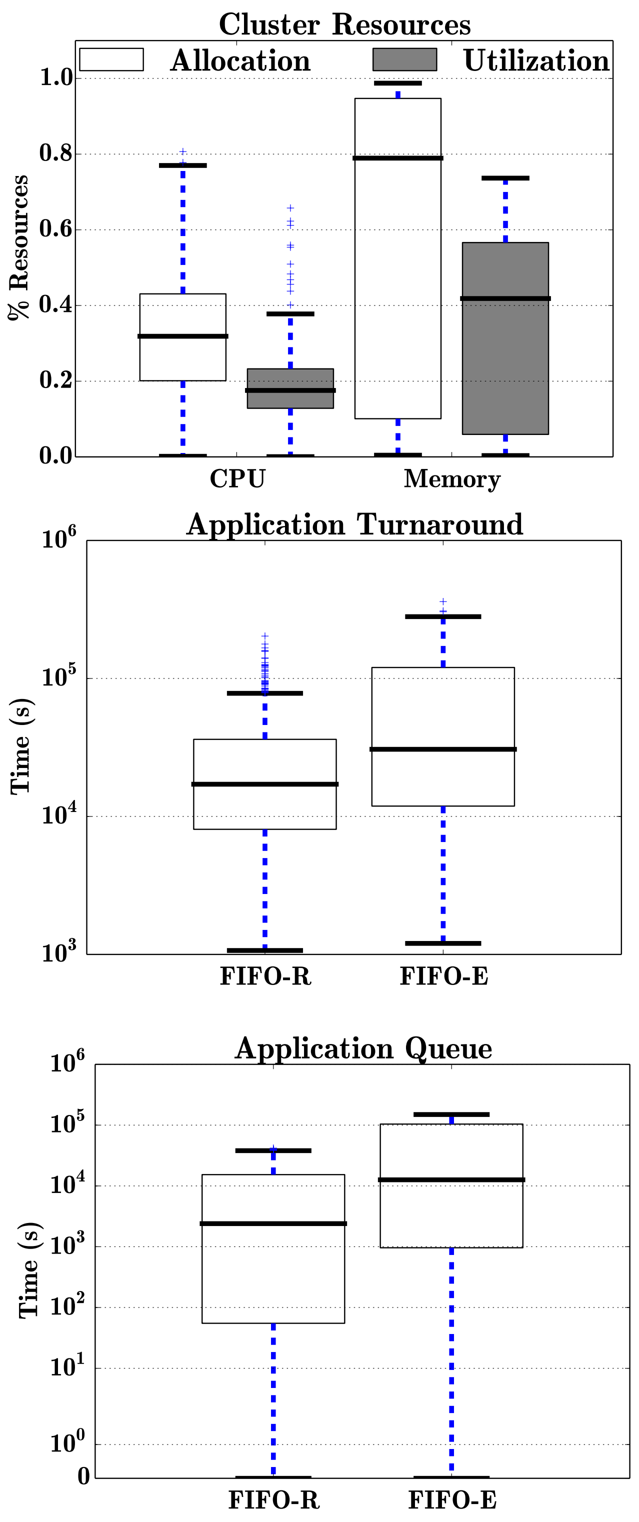
Given our experimental settings, our analysis indicates $\beta = 10\%$ to be a reasonable value that we use, as base[10], throughout the remainder of this section.

We are now ready to study the main metrics we are interested in, and compare –with a real statistical model – optimistic and pessimistic preemption policies, using Figure 13. We also compare results to the ideal case, reported in Figure 11. As a general observation, we see that prediction

[10]This base value is adjusted according to the confidence output by the predict: base value + 3×standard deviation.



(a) Optimistic



(b) Pessimistic

Figure 13: Comparison between an optimistic vs pessimistic approach over different metrics using an GP regression.

errors, and the parameter β play an important role, and result in slightly worse results. Figure 13 (first row) shows that resource slack cannot reach the ideal value of 0%. Furthermore, the difference between a pessimistic and an optimistic preemption policy is less marked than in the ideal case. Indeed, the "safe-guard" buffer has a positive effect for the optimistic policy, which achieves a similar

	Gain - β=10%	**Gain - β=5%**
95-th %	29%	38%
50-th %	30%	80%

Table 9.2a: Analysis of failed applications mistreatment. The table shows the gains in terms of turnaround time, when using a pessimistic approach instead of an optimistic.

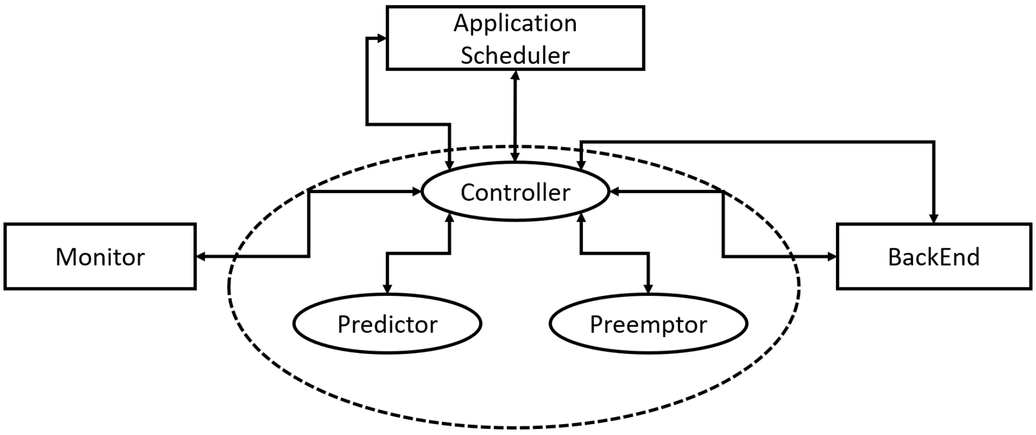


Figure 14: Implementation block diagram.

median slack to the ideal case. The pessimistic policy instead obtains a larger median slack than in the ideal case, and this is due to estimation errors and the strict policy to allocate and deallocate resources. Nevertheless, the pessimistic approach maintains the edge on all metrics, which are visible in Figure 13: median values of resource slack, turnaround and queuing times are in favor of the pessimistic policy. The resource slack is 10% lower while turnaround and queue time are orders of magnitude better than the reservation centric baseline.

We conclude our analysis by focusing on application failures: our goal is to quantify the mistreatment induced by preemption policies in terms of turnaround ratio. Table 9.2a reports the median and 95-th percentile gain of the turnaround ratios computed between the baseline system (in which no application fails) and the dynamic mechanism with the two preemption variants.[11] In general, we see a competitive advantage of the pessimistic policy, which becomes more prominent as a function of β. Focusing on the column for $\beta = 10\%$, the median turnaround of failed application handled by a pessimistic policy is 30% better than that obtained with an optimistic one; the gain for the 95-th percentile of failed application is 29%. When $\beta = 5\%$, the dynamic allocation mechanism follows more closely the real resource utilization, because the "safe-guard" buffer is smaller. In this case, application failures are more prominent (see also Figure 12), and the gap between pessimistic and optimistic policies widens, reaching up to 80% median and 38% 95-th percentile gains.

To summarize, our simulation results indicate that the approach we present in this work achieves substantial gains with respect to a baseline. The GP regression model we advocate for achieves low error rates, and its quantification of uncertainty is beneficial to adjust the value of the "safe-guard" buffer: indeed, results are close to an ideal setup that uses an oracle to predict future utilization patterns.

10 System Implementation

We materialize the ideas presented in this work with a system implementation of the dynamic allocation mechanism, which we cast for the application scheduler [18, 44] we recently adopted to manage our workload. We stress again that we focus on managing memory resources in particular.

Figure 14 shows the block diagram of such an implementation that is composed by three blocks:

[11]We compute two turnaround ratios, between the baseline and each variant. Then we report the difference, in percentage, between the two.

controller, predictor and preemptor. We assume the other modules (application scheduler, monitor and back-end) to be part of the cluster management system as discussed in Section 8. Our implementation consist of few hundreds lines of Python code, because of simplicity and the availability of machine learning libraries required for the predictor module.

Note that scalability can be an issue for a centralized system due to the computational complexity of the predictor. However, a distributed solution can be easily engineered; since updating the resource allocation is a task that must be done at the host level, all of our blocks (controller, predictor and preemptor) can be deployed on each host.

We use Docker [8] as the back-end and we have investigated how to resize its containers (components). There are two values that Docker uses to check for Memory limits: a hard and a soft limit. When the hard limit is surpassed, the container is killed. Instead, when the soft limit is reached, the OS tries to release some resources first. We opt to modulate resource allocation by adjusting the soft limit value since the application scheduler we use takes decisions based on that value. However, modifying the hard limit is a viable option as well. Changing the containers allocation is important for the reasons briefly discussed in Section 8. In particular, we rely on the OS low level mechanisms to notify the processes running in the container that they have to free some of their resources. This practice is compatible with frameworks such as the Java Garbage Collector (GC) that attempts to release allocated but unused memory space. Note that our technique is compatible with approaches such as [27], which trade performance for a smaller memory footprint of applications.

Recall that our approach feeds the forecast module with data from the monitoring component at regular time intervals. Frequent updates ultimately result in better system efficiency, as the predictor operate on a high-fidelity view of resource utilization in the cluster. However, this might impose a high toll in terms of monitoring scalability. On the other hand, infrequent updates improve scalability at the expense of lower system efficiency and responsiveness. In our system, we collect resource utilization information every minute, which is in line with what done in [4].

We now provide additional details of our system:

- **Predictor.** It implements the GP regression model described in Section 8.1 that predicts the resource utilization of all application components running in the cluster using a history size of 10 points. We implement GP regression using the know library GPy [45]. Clearly, alternative machine learning algorithms that can quantify predictive uncertainty can also be used.

- **Preemptor.** It materializes alg. 1 by performing, for each host, a simulation of the resource to be allocated to application components given the information obtained from the *predictor*. This component identifies the best allocation that prevents application failures, using a pessimistic policy.

- **Controller.** It is responsible for coordinating all system modules. It is in charge of calculating the allocation for a specific application component given the predicted value and variance obtained from the *predictor*. As discussed previously, the buffer β is set to compensate for prediction uncertainty. The final task of the controller is that of issuing commands to preempt (kill, in our system) an entire application, or individual components thereof, and to resize the allocation, as computed by *preemptor* module. It is important to point out that the controller adapts resource allocations only after enough historical data points are available for the *predictor*: we call this a **grace period**. In our experiments, resource allocation matches resource reservations for the first 10 minutes of an application lifetime.

11 Experimental Evaluation

We have deployed the system implementation in our cluster, which we operate using [44, 18]. Our goal is to perform a comparative analysis between our dynamic allocation mechanism and a baseline, reservation centric approach. In our experiments, we consider exactly the same workload trace on both systems which takes approximately 24 hours from the first submission to the completion of the last application.

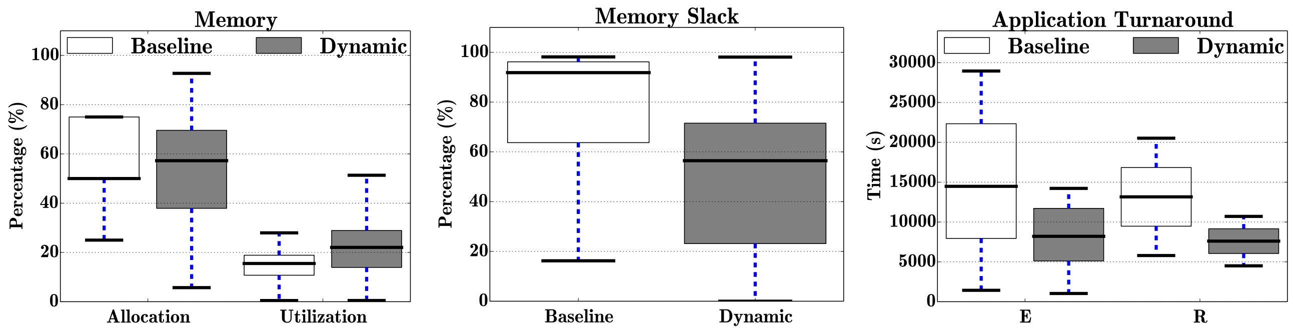


Figure 15: Comparison of memory, memory slack and turnaround time distributions using the FIFO discipline. Gray boxes correspond to the dynamic system. *E* stands for elastic and *R* stands for rigid applications.

Workload. We use two representative application templates including: 1) an elastic application using the Apache Spark framework; 2) a rigid application using the TensorFlow framework. Following the trend of the traces used in Section 9, we set our workload to include 60% of elastic and 40% of rigid applications, for a total of 100 applications. Application inter-arrival times follow a Gaussian distribution with parameters $\mu = 120$ sec, and $\sigma = 40$ sec, which is compatible with what we observe in our cluster. Regarding the elastic application templates, we consider three use cases. First we consider an application that induces a random-forest regression model to predict flight delays, using publicly available data from the US DoT.[12] Second we consider a music recommender system based on the alternating least squares algorithm, using publicly available data from Last.fm[13]. Third we consider an Extract, Transform and Load (ETL) application. All applications have 3 different flavors: while they all have 3 core components, the number of elastic components varies depending on the flavor. In terms of RAM, all flavors have different reservation values that span from 8GB to 32GB. Instead, using the rigid application template, we train a deep GP model [46], and use a single-node TensorFlow program, requiring 1 worker with 8-16-32GB of RAM depending on the flavor.
Experimental setup. We run our experiment on a isolated platform (which we use as testbed for non-production systems) with ten servers, each with a 8-core CPU running at 2.40GHz, 64GB of memory, 1Gbps Ethernet network fabric and two 1TB hard drives. The servers use Ubuntu 14.04 and Docker 17.09.0. Docker images for the applications are preloaded on each machine to prevent container startup delays and network congestion.
Summary of results. Using the FIFO scheduling policy, we compare the two systems; baseline and dynamic. Overall, the dynamic system is largely more efficient and responsive.

We measure substantial improvements in terms of resource allocation, as illustrated in Figure 15 (left): indeed our system can afford to ingest more applications, that would otherwise wait to be served. Figure 15 (center) illustrates resource slack, which is roughly 40% lower with our dynamic allocation mechanism.

As a consequence, applications spend less time in the application scheduler queue and have short turnaround times, as shown in Figure 15 (right). The median turnaround times are $\sim$ 50% shorter, for both elastic and rigid applications. Note also that the tails of the distributions are in favor of our approach. Finally, we report that no application, nor component failed when using our dynamic allocation mechanism, configured with the pessimistic preemption policy.

[12]http://stat-computing.org/dataexpo/2009/the-data.html
[13]http://www-etud.iro.umontreal.ca/~bergstrj/audioscrobbler_data.html

Part III
Conclusions

The emergence of "the data-center as a computer" paradigm has led to unprecedented advances in cluster management frameworks, that aim at exposing distributed cluster resources as high-level primitives to a variety of business-critical and scientific applications. However, the current resource reservation model, based on estimated peak usage, causes over-provisioning and hinders an efficient use of cluster resources.

The problem of under utilization has been addressed by several approaches. For example, the design of economic incentives to steer system operation has led to the development of complex resource markets, e.g. AWS Spot instances, which calls for the design of failure tolerant applications, due to the ephemeral nature of the resources they are offered.

The work presented in this deliverable completes tasks 5.2 (system deployment strategies, Zoe Analytics) and 5.3 (System deployment tools). We use the monitoring information coming from task 5.1 (system performance monitoring) and the deployment system from task 5.2 to develop a software module that implements strategies for optimizing *analytics as a service* deployments.

We presented Zoe Analytics, a project started in IOStack that is becoming successful in its own right, and a software module that cooperates with it to dynamically adjust resources allocated to an application, so that clusters can be used more efficiently, leading to immediate savings in terms of hardware and energy.

The design of the resource allocation module features a method to build a statistical model to forecast resource utilization, and a preemption policy that reallocates system resources while minimizing failures. Both are contributions that have been implemented in Zoe, but are generic and can be applied to other systems.

We have validated our mechanism numerically and with an experimental campaign based on the Zoe implementation. The numerical simulations shed light on the key role played by our ability to model and use prediction uncertainty, and by the use of strict preemption vs. optimistic concurrency control. The real system implementation experiments indicate notably improved system efficiency, which translates in better responsiveness.

References

[1] M. Babaioff, Y. Mansour, N. Nisan, G. Noti, C. Curino, N. Ganapathy, I. Menache, O. Reingold, M. Tennenholtz, and E. Timnat, "Era: A framework for economic resource allocation for the cloud," in Proceedings of the 26th International Conference on World Wide Web Companion, pp. 635–642, International World Wide Web Conferences Steering Committee, 2017.

[2] Y. Yang, G.-W. Kim, W. W. Song, Y. Lee, A. Chung, Z. Qian, B. Cho, and B.-G. Chun, "Pado: A data processing engine for harnessing transient resources in datacenters," in Proceedings of the Twelfth European Conference on Computer Systems, pp. 575–588, ACM, 2017.

[3] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao, "Efficient queue management for cluster scheduling," in Proceedings of the Eleventh European Conference on Computer Systems, p. 36, ACM, 2016.

[4] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in Proceedings of the Tenth European Conference on Computer Systems, p. 18, ACM, 2015.

[5] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in Proceedings of the 8th ACM European Conference on Computer Systems, pp. 351–364, ACM, 2013.

[6] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, "Reservation-based scheduling: If you're late don't blame us!," in Proceedings of the ACM Symposium on Cloud Computing, pp. 1–14, ACM, 2014.

[7] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica, "True elasticity in multi-tenant data-intensive compute clusters," in Proceedings of the Third ACM Symposium on Cloud Computing, p. 24, ACM, 2012.

[8] Docker, "Docker." http://www.docker.com/, 2017.

[9] B. Hindman et al., "Mesos: A platform for fine-grained resource sharing in the data center," in Proc. of the USENIX NSDI 2011, NSDI'11, (Berkeley, CA, USA), pp. 295–308, USENIX Association, 2011.

[10] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in Proceedings of the Third ACM Symposium on Cloud Computing, p. 7, ACM, 2012.

[11] J. Wilkes, "More Google cluster data." Google research blog, Nov. 2011. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[12] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in NSDI, vol. 11, pp. 24–24, 2011.

[13] A. W. S. (AWS), "Elastic map reduce (emr)." https://aws.amazon.com/emr/, 2017.

[14] Apache, "Spark." http://spark.apache.org/, 2017.

[15] Google, "Tensorflow." https://www.tensorflow.org/, 2017.

[16] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda, "Tr-spark: Transient computing for big data analytics," in Proceedings of the Seventh ACM Symposium on Cloud Computing, pp. 484–496, ACM, 2016.

[17] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in NSDI, pp. 469–482, 2017.

[18] F. Pace, D. Venzano, D. Carra, and P. Michiardi, "Flexible scheduling of distributed analytic applications," in CCGRID 2017, 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, May 14-17, 2017, Madrid, Spain, (Madrid, SPAIN), 05 2017.

[19] Apache, "Aurora." http://aurora.apache.org/, 2017.

[20] A. Kuzmanovska, R. H. Mak, and D. Epema, "Koala-f: A resource manager for scheduling frameworks in clusters," in Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on, pp. 80–89, IEEE, 2016.

[21] A. Kuzmanovska, R. H. Mak, and D. Epema, "Dynamically scheduling a component-based framework in clusters," in Workshop on Job Scheduling Strategies for Parallel Processing, pp. 129–146, Springer, 2014.

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

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

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

[25] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: improving resource efficiency at scale," in ACM SIGARCH Computer Architecture News, vol. 43, pp. 450–462, ACM, 2015.

[26] M. Shahrad, C. Klein, L. Zheng, M. Chiang, E. Elmroth, and D. Wentzlaf, "Incentivizing self-capping to increase cloud utilization," in ACM Symposium on Cloud Computing 2017 (SoCC'17), Association for Computing Machinery (ACM), 2017.

[27] W. U. Hassan and W. Zwaenepoel, "Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling," in USENIX Annual Technical Conference (USENIX ATC 17), 2017.

[28] V. K. Vavilapalli et al., "Apache hadoop yarn: Yet another resource negotiator," in Proc. of the ACM SoCC 2013, p. 5, ACM, 2013.

[29] Docker, "Swarm." https://docs.docker.com/swarm/, 2015.

[30] Google, "Kubernetes." http://kubernetes.io/, 2017.

[31] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das, "Phoenix: A constraint-aware scheduler for heterogeneous datacenters," in Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on, pp. 977–987, IEEE, 2017.

[32] Y. Zhang, G. Prekas, G. M. Fumarola, M. Fontoura, Í. Goiri, and R. Bianchini, "History-based harvesting of spare cycles and storage in large-scale datacenters," in Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, (Berkeley, CA, USA), pp. 755–770, USENIX Association, 2016.

[33] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in NSDI, pp. 363–378, 2016.

[34] G. Box, G. Jenkins, and G. Reinsel, Time Series Analysis, Forecasting and Control. Wiley Series in Probability and Statistics, Wiley, 2008.

[35] C. E. Rasmussen and C. K. I. Williams, Gaussian Processes for Machine Learning. MIT Press, 2006.

[36] D. J. C. MacKay, Information Theory, Inference & Learning Algorithms. Cambridge University Press, 2003.

[37] E. Snelson and Z. Ghahramani, "Sparse gaussian processes using pseudo-inputs," in Proceedings of the 18th International Conference on Neural Information Processing Systems, NIPS'05, (Cambridge, MA, USA), pp. 1257–1264, MIT Press, 2005.

[38] J. Quiñonero Candela and C. E. Rasmussen, "A unifying view of sparse approximate gaussian process regression," J. Mach. Learn. Res., vol. 6, pp. 1939–1959, Dec. 2005.

[39] A. Rahimi and B. Recht, "Random features for large-scale kernel machines," in NIPS, 2007.

[40] K. Chalupka, C. K. I. Williams, and I. Murray, "A framework for evaluating approximation methods for gaussian process regression," J. Mach. Learn. Res., vol. 14, pp. 333–350, Feb. 2013.

[41] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in NSDI, pp. 649–667, 2017.

[42] C. Reiss et al., "Google cluster-usage traces: format + schema," technical report, Google Inc., Nov. 2011.

[43] Google, "Google Public Traces." https://github.com/google/cluster-data, 2011.

[44] Eurecom, "Zoe-analytics." http://zoe-analytics.eu/, 2015.

[45] Sheffield, "Gpy." https://sheffieldml.github.io/GPy/, 2017.

[46] K. Cutajar, E. Bonilla, P. Michiardi, and M. Filippone, "Random feature expansions for deep Gaussian processes," in ICML 2017, 34th International Conference on Machine Learning, 6-11 August 2017, Sydney, Australia, (Sydney, AUSTRALIA), 08 2017.