Dynamic Resource Shaping for Compute Clusters

Francesco Pace
Eurecom
Sophia-Antipolis, France
francesco.pace@eurecom.fr

Dimitrios Milios
Eurecom
Sophia-Antipolis, France
dimitrios.milios@eurecom.fr

Damiano Carra
Università di Verona
Verona, Italy
damiano.carra@univr.it

Pietro Michiardi
Eurecom
Sophia-Antipolis, France
pietro.michiardi@eurecom.fr

***Abstract*—Nowadays, data-centers are largely under-utilized because resource allocation is based on reservation mechanisms which ignore actual resource utilization. Indeed, it is common to reserve resources for peak demand, which may occur only for a small portion of the application life time. As a consequence, cluster resources often go under-utilized.**

In this work, we propose a mechanism that improves compute cluster utilization and their responsiveness, while preventing application failures due to contention in accessing finite resources such as RAM. Our method monitors resource utilization and employs a data-driven approach to resource demand forecasting, featuring quantification of uncertainty in the predictions. Using demand forecast and its confidence, our mechanism modulates cluster resources assigned to running applications, and reduces the turnaround time by more than one order of magnitude while keeping application failures under control. Thus, tenants enjoy a responsive system and providers benefit from an efficient cluster utilization.

I. Introduction

Data-center efficiency is a subject that attracted a vast amount of research [1]–[7]. Recently, the cloud computing paradigm, both in its public and private forms, fueled the proliferation of a wide array of resource management tools [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 recent traces from large-scale production clusters [10], [11]: in most cases ($\sim$ 80%) resource utilization is less than 40% or 80% of the allocated resources depending on application types.

Current approaches that address efficiency requirements fall in two broad categories. The first involves methodologies that steer tenants' behavior through the design of incentive mechanisms; tenants are endowed with the task of optimizing their cost to operate their applications, whereas providers operate on prices to regulate the allocation of idle resources. Such approaches are largely adopted by public cloud providers [1]. The second category concerns approaches that operate at the system level, and propose mechanisms that allocate resources based on tenants' reservations [3]–[5], [8], [9], [12], [13]. Essentially, existing approaches either let tenants reason in terms of value and costs [1], or let the system determine how to avoid wasting scarce and costly resources.

In this paper, we discuss a methodology that belongs to the second category: we present a mechanism that dynamically adjusts resources allocated to running applications according to their expected utilization, as opposed to a static allocation based on tenants' reservations. In the context we consider, we define as *applications* the use of distributed frameworks such as Apache Spark [14] and Google TensorFlow [15] that use different components to produce work.

In general, cluster schedulers provision and manage resources as follows: given a **resource request**, the resource manager determines its admission in the cluster based on **reservation** information.[1] An admitted request triggers a **resource allocation** procedure, which concludes with reserved resources being allocated to the request [5]. 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 time, but 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, which is one key factor that induces poor system utilization, and ultimately, negatively impacts system efficiency. This is exacerbated by coarse-grained reservation specifications: 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 paper) is a daunting task [17], which requires sophisticated optimization mechanisms going beyond human tuning abilities.

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

Problem Statement. We study the problem of cluster efficiency by reducing the resource slack induced by reservation-centric application schedulers, which match allocation to reservation. To do so, we introduce a new mechanism that **predicts** the resource utilization and adjusts the resource allocation accordingly. The main challenge to face is that prediction errors may have problematic consequences, since sudden spikes could wreak havoc the system [4]. When dealing with finite resources such as RAM, in fact, not providing the correct amount of resources leads to application failures. Careful engineering would suggest to introduce a buffer that

[1] 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.

will act as "safe-guard" to prediction errors. This results in a **trade-off**, since on the one hand the safe-guard buffer should be small to minimize slack, while on the other hand it should be sufficiently large to prevent application failures.

In our approach, we leverage on three key ideas: prediction confidence, application elasticity and controlled failures. In the prediction process, we argue for models that provide additional information about the **confidence** of the prediction. We use such information to dynamically adapt the safe-guard buffer that should prevent application failures. In addition, application frameworks are composed by several elements that are characterized by either a **core** or **elastic** nature [4], [18]. Core components are compulsory for a framework to produce useful work (e.g, Apache Spark requires a controller, a master, and one worker); additional elastic components can contribute to a job, e.g. by decreasing its runtime. An application that features only core components is called **rigid**, whereas applications with a mix of core and elastic components are called **elastic**. If the resource demand is higher than the available resources, we intervene (when possible) on elastic components to avoid application failures. As a last step, should the previous two mechanisms not be sufficient to provide enough resources, we explicitly decide which application should fail so that to minimize the amount of wasted work.

Contributions. In this paper we present our design of a data-driven resource shaping 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 **online forecasting** of resource demand to modulate allocated resources such as they approximate utilization patterns well. In summary, the contributions we present in this work are as follows:

- We present a new mechanism that dynamically adjusts resources allocated to applications by an existing scheduler. In this work, we target a specific family of application schedulers, and materialize our ideas for such systems.
- We compare parametric and non-parametric machine learning methodologies for the forecasting of resource utilization. In particular, we focus on quantification of uncertainty, which 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 datacenters, and discuss about the trade-off that an optimistic vs. a pessimistic approach to application preemption entails. We also present a full-fledged implementation of our mechanism, that we use in an academic compute cluster serving hundreds of students and researchers. Our 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 of the paper is organized as follows. In Section II we review the related literature. In Section III we present our system design, and we validate our ideas using a simulation campaign in Section IV. We present our prototype implementation in Section V and its evaluation in Section V-A. Finally we conclude in Section VI.

II. Related Work

Resource allocation has been approached in many different ways in the literature [1]–[4], [6], [7], [12], [19]–[27].

The authors in [19], [20] use feedback control loop which requires every *framework* to periodically send application-specific information to the scheduler, which is used to steer resource allocation. In contrast, our approach does not require such instrumentation, as it is application agnostic.

The authors in [6] introduce a reservation-based scheduler and 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 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; generally, users have no knowledge of how their applications will behave.

The authors in [27] develop a feedback control loop for virtual machines, using a simple regression model to forecast future allocation. They show that it is possible to reduce the CPU resource slack, but they do not address memory and the consequences that under-provisioning such resource has on applications, as we do in our work.

The authors in [29] adopt a distributed scheduling architecture, whereby each scheduler aims at minimizing task completion time by careful placement strategies that use estimates of task runtime and their resource utilization. Contrary to our work, they use over-provisioning of resources and they tackle conflicts in an optimistic-manner. Our approach cooperates with an existing scheduler, instead of replacing it, and does not use task runtime to gauge cluster resources.

Some other works [1], [25] propose to address the problem with economics principles. In particular, in [25] 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 achieve that by allocating more capacity than what is physically available, i.e., over-provisioning, which is a solution prone to uncontrolled failures[2] when utilization exceeds available resources.

Finally, works such as [3], [7], [12], [21]–[24], [30], [31], focus either on resource placement or on meeting Service Level Objective (SLO). In the first case they relate to a packing problem and try to optimize it; Karanasos et al [30] suggest to dynamically re-balance the load across hosts if the packing

[2] The Operating System (OS) kills processes due to Out Of Memory (OOM) following its own algorithm.

performed at a certain time leads to uneven loaded hosts. In the second case they leverage the elasticity of some frameworks and they increase resources for applications that are falling behind on their SLO. Our work is orthogonal to such methods and can leverage them to improve the system performance.

The authors in [32] propose task scheduling and data placement techniques that rely on historical resource utilization. Specifically, they process the history of CPU utilizations using the Fast Fourier Transform. Using the k-means algorithm, they cluster patterns in three categories: periodic, constant and unpredictable. They exploit the patterns of periodic and constant categories to improve the quality of task scheduling.

Albeit all these works are valid and propose their own vision of the problem, they share one element: although some of them address a multi-dimensional packing problem for provisioning resources to applications, when it comes to reclaiming resources granted to applications they mostly focus on "time sharable" resources, like the CPU, rather than "finite" resources like memory. As a consequence, such methods are limited to improve system efficiency from the perspective of CPU utilization.

An example of prior work that modulates "finite" resources is Borg [4]. Borg features a resource reclamation system that seizes unused resources and offers them to other applications. 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 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 accepting the trade-off obtained by the medium setting. Instead, we present a dynamic allocation system that relies on online resource forecasting, with accurate quantification of uncertainty. In addition, we seek to gain control over the OS and minimize application failures events while maximizing the resource utilization.

What sets apart our approach from previous work is as follows. We use on-line forecasting with quantification of uncertainty to steer system behavior. We explicitly take into account finite resources which, if handled improperly, can lead to failures. Additionally, we operate on low-level UNIX processes, and take control over the OS for shaping the resources allocated to applications.

III. System Design

Figure 1 illustrates the architecture we assume in our work. The *backend* module is an instance of a cluster management system, such as Docker [33] or Kubernetes [34]. Additionally, we assume the presence of an *application scheduler* such as [18], which reads the compute cluster state from a dedicated database component. The monitoring component populates the cluster state database with allocation and utilization measurements taken from the backend, for *every* component of *every* running application. To minimize intrusiveness, this component uses standard metrics (CPU, memory, etc) as they are seen by the OS. In this Section, we focus on the two additional components we present in this paper:

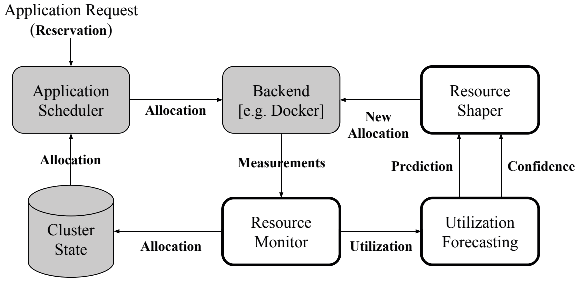


Fig. 1. System overview: shaded boxes represent existing components, white boxes indicate new components presented in this work.

Utilization forecasting module. The goal of this module is to anticipate the resource utilization of every application component. We study both parametric and non-parametric modeling approaches to predict resource utilization, with emphasis on the quantification of the uncertainty associated to these predictions. A more detailed exposition of the methodology we employ can be found in Section III-A.

Resource shaper module. This module uses utilization forecasts to adjust the resources allocated to every component of running applications. 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. A more detailed exposition of β can be found in Section III-B. Additionally, the resource shaper is in charge of application preemption. Preemption policies can either be optimistic [4], [5] or strict (pessimistic). We advocate for a strict policy, to avoid delegating application preemption to the OS, which manages resource shortage (such as OOM) in an application agnostic and "unpredictable" way. A detailed exposition of the preemption policy can be found in Section III-B.

A. Utilization Forecasting Module

The forecasting module is responsible for making predictions about future resource utilization, for each application component. For a given application, we forecast *both* CPU and memory utilization using measurement time series that reflects resource usage across time. We seek to discover patterns of resource usage that allow reasoning about our expectations on the future state of the system utilization.

We advocate for the need to **quantify the level of uncertainty** associated with each prediction: predictive errors may have serious impact on "finite" resources (i.e. memory), as they can cause application failures. Although errors are unavoidable to a certain extent, predictive confidence can be used to adjust the degree of adaptiveness to the anticipated workload: intuitively, a prediction with low confidence implies that the resource shaper should be conservative regarding changes in resource allocation.

In this work we compare the traditional *parametric* Autoregressive Integrated Moving Average (ARIMA) model to an alternative *non-parametric* model that offers a principled quantification of uncertainty.[3] On the one hand, we use state-of-the-art ARIMA implementations that automatically tune hyper parameters and that provide a method to compute confidence levels associated to predicted values [36]. On the other hand, we model resource utilization using GP regression [37], 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 [38]. 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 in a principled way.

1) Time-series Prediction with ARIMA: ARIMA [36] is often considered as the "go-to method" for time series forecasting: it is a generalization of the Autoregressive Moving Average (ARMA) model to cope with non-stationary time series data, which appear frequently in real-life applications such as the one we consider in this paper. Considering observation y_t at time t, the ARMA(p',q) model is described as follows:

$$y_t - \alpha_1 y_{t-1} - ... - \alpha_{p'} y_{t-p'} = \epsilon_t + \theta_1 \epsilon_{t-1} + ... + \theta_1 \epsilon_{t-q} \quad (1)$$

where α are the parameters of the autoregressive part of the model, θ are the parameters of the moving average part and the ϵ are error terms, and $p', q \geq 1 \in \mathbb{N}$ refer to the order of the autoregressive and moving average parts of the model.

The idea of ARIMA is that current values of a time series can be obtained by a linear combination of its past values, using finite differencing to produce stationary data. Formally, the ARIMA(p,d,q) model using lag polynomials is:

$$(1 - \sum_{i=1}^{p} \phi_i L^i)(1-L)^d y_t = \delta + (1 + \sum_{i=1}^{q} \theta_i L^i)\epsilon_t \quad (2)$$

where $p = p' - d$, δ is a constant and L is defined as the lag or back-shift operator. d is an integer greater than or equal to zero and refers to the order of the integrated parts of the model and controls the level of differencing. Generally $d = 1$ is enough in most cases.

In this work, we perform model selection – i.e., searching through combinations of order parameters to pick the set that optimizes model fit criteria – using the Akaike information criteria, a method that is widely available in most ARIMA implementations. Note that parameter optimization is an operation that needs to be performed multiple times during a forecasting period, to adapt to time series dynamics.

Finally, most ARIMA implementations output confidence intervals associated with the selected model parameters [36]. We note that confidence intervals should not be confused with prediction intervals: the former are associated to the probability of the true model parameters to be within the confidence interval, whereas the latter are associated to the likely range of future values output by the model. As discussed in the literature [36], confidence intervals for the mean are generally much narrower than prediction intervals. This has a direct consequence in the context of our work, which revolves around the idea of using predictive confidence to steer system behavior: for this reason, in the next section, we develop a Bayesian approach to time series modeling that features a principled approach to compute predictive confidence.

2) Time-series Prediction with GPs: In the GP literature, time series are treated as state space models, which are generalizations of auto-regressive models [39], [40]. Considering state x_t and observation y_t at time t, a state space model is described as follows:

$$\begin{aligned} x_{t+1} &= f(x_t) + \epsilon_t \\ y_t &= g(x_t) + v_t \end{aligned} \quad (3)$$

where $f(x_t)$ is the state transition function and ϵ_t is the process noise, which follows a normal distribution. The state x_t may not be observed directly; an observation y_t is given as a function of the state $g(x_t)$, which is additionally corrupted by observation noise v_t. According to Equation (3), a time series is modeled as a non-linear Markovian dynamical system. The Markov property implies that the current state x_t is *conditionally* independent from past states $\{x_\tau : \tau < t-1\}$, given the previous state x_{t-1}. The same is not true for the observations however. Thus, given a collection of noisy observations $\{y_\tau : \tau \leq t\}$, the goal for time series prediction is to infer the future state x_{t+1}. This requires learning the functions f and g, which involves placing a GP prior over f and g. However, the posterior over a non-linear dynamical system is not Gaussian, thus several approximation methods have been proposed in the literature [41], [42].

In the context of recording resource utilization, we can make some simplifying assumptions. It is reasonable to assume that an observation y_t matches the state x_t. Of course, we have to acknowledge that resource utilization constantly fluctuates; these fluctuations however can be sufficiently explained by the noise term ϵ_t, which now accounts for both the process and the observation noise. We shall additionally make the dependency on past states explicit; for a history window of size h, we consider the following state-space model:

$$y_t = f(y_{t-1}, \ldots, y_{t-h}) + \epsilon_t \quad (4)$$

To make predictions, we shall learn the transition function f by means of standard GP regression. From Equation (4), the transition function depends on the history explicitly. In this way, we avoid the additional costs of approximating the true posterior of a non-linear dynamical system.

A GP model transfers information across points that are considered similar, as this is reflected in the choice of kernel

[3] We are aware of alternative approaches, e.g. [35], but none of them offer a principled quantification of uncertainty as Gaussian Process (GP) model do.

$k(x, x')$, which determines the prior covariance between inputs x and x'. 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.

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 utilization patterns expressed as vectors of the form:

$$\tilde{\mathbf{x}}_t = [x_t, y_{t-h}, \ldots, y_{t-1}]^\top \tag{5}$$

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

$$k_h(x, x') = k(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}') \tag{6}$$

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

3) How Online Forecasting Works: In practice, *our forecasting component operates in an online manner*. As long as new data is available, the predictive model will be trained and subsequently queried about the future workload. Depending on the modeling methodology, our approach is as follows.

Using the ARIMA model. The online training and prediction process that uses ARIMA operates by appending the new resource utilization data to the collection of observations gathered so far. ARIMA hyper-parameters are optimized using well-known, albeit computationally expensive, methods [43], [44].

Using the GP model. The online training and prediction process that uses GP regression operates as follows:

1) New resource utilization data is appended to the collection of observations $\mathbf{X}, \mathbf{y}$. The rows of $\mathbf{X}$ are patterns as defined in Equation (5).
2) Using a history-dependent kernel $k_h(x, x')$, Equations (7) and (8) are used to make predictions based on observations $\mathbf{X}, \mathbf{y}$.

Under the assumption of a zero-mean prior and 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 as follows:

$$\mathrm{E}[f(x) \mid \mathbf{X}] = k_h(x, \mathbf{X})(k_h(\mathbf{X}, \mathbf{X}) + \sigma^2)^{-1}\mathbf{y} \tag{7}$$

$$\begin{aligned}\mathrm{Var}[f(x) \mid \mathbf{X}] &= k_h(x, x') \\ &- k_h(x, \mathbf{X})(k_h(\mathbf{X}, \mathbf{X}) + \sigma^2)^{-1} k_h(\mathbf{X}, x)\end{aligned} \tag{8}$$

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.

The regression step can be computationally expensive. Equations (7) and (8) involve a matrix inversion (for $k(\mathbf{X}, \mathbf{X}) + \sigma^2$), which is an operation of cubic complexity. Moreover, the set of observations $\mathbf{X}, \mathbf{y}$ will grow indefinitely during the lifetime of the system. While there is a plethora of methodologies on sparse GPs in the literature [45]–[48], that can be used to reduce the complexity of regression, in this work we adopt the simple solution of restricting the dataset $\mathbf{X}, \mathbf{y}$ to the N latest observations, thus keeping the model tractable. Note that N is the number of patterns used; it should not be confused with h, which is the size of each pattern.

Discussion. We have performed a preliminary study on our modeling approaches, using a dataset consisting of approximately 6000 time series that monitor the memory usage of applications in our academic cluster. Overall, our results indicate that ARIMA exhibit lower error rates than a GP model. However, crucially, the confidence intervals associated with ARIMA are excessively small, even smaller than the measured prediction error rate. Instead, the GP model provides an accurate quantification of prediction uncertainty. In other words, ARIMA's confidence intervals can reveal harmful when used as the only information to reason about uncertainty, because they could lead to over-confident decisions. This is corroborated by our experimental results.

B. Resource Shaper Module

We use the resource shaper module to adjust resource allocated to an application and its components as a function of predicted utilization, and its confidence. When resource are underutilized, the resource shaper "redeems" the excess capacity such that the application scheduler can dequeue idle applications. On the contrary, upon a utilization spike, the resource shaper 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* we associate to the resource shaper 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. 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.

Recent works [4] study 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. We consider preemption primitives such as the `kill` operation, which inevitably waste work. Component or application suspension [49] and migration are outside the scope of this work.

Algorithm 1 presents the pessimistic preemption policy, which is triggered by the output of the forecasting module. Given the current cluster state, and the resource utilization forecasts, the algorithm computes a new resource allocation for each running application, which is then imposed on the cluster by operating directly on application components through low-level preemption primitives.

The algorithm starts by initializing (lines 1-5) the variables that holds the information about the allocated resources. Then it sorts (line 6) running applications according to the application scheduler policy (e.g.; First-In-First-Out (FIFO), that is, arrival times), and it computes (lines 7-33) an allocation by trying to maximize the resource allocation while minimizing the number of running applications. In particular, it first allocates the core (lines 8-19) components and then all elastic components[4] that fit in the host (lines 23-33). The algorithm continues until all running applications are processed.

Resource allocation is determined, and we can turn our attention to preemption. Core components that no longer fit a host entail *full* application preemption (lines 34-36). Also elastic components can be preempted (lines 37-38), inducing only a partial application preemption. In addition, in case of elastics components, we can experience partial or entire loss of the work done by the preempted component. For this reason, our algorithm allocates the core components of an application, then moves to the elastic components by giving priority to the ones that have been living in the cluster for a longer time (line 25). Components recently scheduled are suitable candidates for preemption, because they have likely produced less useful work. Finally, the algorithm resizes (lines 39-41) the components according to the computed allocations. Our algorithm currently supports CPU and Memory, but it can be extended to other types of resource as well.

Safe-guard buffer. We are now ready to define the "safe-guard" buffer. The buffer size β is a function of the uncertainty quantified by the forecasting module $\beta = K_1 R_{A_i} + K_2 V_{A_i}$, where R_{A_i} is the initial resource **request** for application A_i, and V_{A_i} is the estimated variance of the prediction, as these are given by the forecasting module (ARIMA or GP). β involves a constant term $K_1 R_{A_i}$ and a dynamic term $K_2 V_{A_i}$. The constant term can be thought of as a *minimum resource allocation* that is granted to application A_i [4]. The dynamic term uses the confidence (expressed as variance V_{A_i}) given by the predictor to adjust β accordingly: it thus changes during an application lifetime. In Section IV, we study how different values of K_1 and K_2 affect the performance of our method.

Algorithm 1: Overview of the pessimistic preemption policy implemented by the resource shaper module.

```
Data: $\mathcal{H} \leftarrow$ Hosts, $\mathcal{A} \leftarrow$ Running Applications
1  $cpusFree \leftarrow Array(\mathcal{H})$
2  $memFree \leftarrow Array(\mathcal{H})$
3  foreach $host \in \mathcal{H}$ do
4      $cpusFree[host] \leftarrow host.totalCpus$
5      $memFree[host] \leftarrow host.totalMem$
6  $\mathcal{J} \leftarrow$ SORT($schedulingPolicy$, $\mathcal{A}$)
7  foreach $req \in \mathcal{J}$ do
8      $cpus \leftarrow cpusFree$
9      $mem \leftarrow memFree$
10     $remove \leftarrow False$
11     foreach $c \in req.CoreCpts$ do
12         $cpus[c.host] \leftarrow cpus[c.host] - c.futureCpus - \beta$
13         if $cpus[c.host] < 0$ then
14             $remove \leftarrow True$
15             break
16         $mem[c.host] \leftarrow mem[c.host] - c.futureMem - \beta$
17         if $mem[c.host] < 0$ then
18             $remove \leftarrow True$
19             break
20     if $remove$ then
21         INSERT($req$, $\mathcal{K}$)
22     else
23         $cpusFree \leftarrow cpus$
24         $memFree \leftarrow mem$
25         $E \leftarrow$ SORT($timeAlive$, $req.ElasticCpts$)
26         foreach $e \in E$ do
27             $cpus \leftarrow cpusFree[e.host] - e.futureCpus - \beta$
28             $mem \leftarrow memFree[e.host] - e.futureMem - \beta$
29             if $cpus \leq 0$ or $mem \leq 0$ then
30                 INSERT($e$, $\mathcal{K}_{\mathcal{E}}$)
31             else
32                 $cpusFree[r.host] \leftarrow cpus$
33                 $memFree[r.host] \leftarrow mem$
34 foreach $req \in \mathcal{K}$ do
35     foreach $c \in (req.CoreCpts \cup req.ElasticCpts)$ do
36         PREEMPCOMPONENT($c$)
37 foreach $e \in \mathcal{K}_{\mathcal{E}}$ do
38     PREEMPCOMPONENT($e$)
39 foreach $req \in \mathcal{J} \setminus \mathcal{K}$ do
40     foreach $c \in (req.CoreCpts \cup req.ElasticCpts)$ do
41         RESIZECOMPONENT($c$)
```

IV. Simulation-based Evaluation

A. Methodology

We evaluate our mechanism using an event-based, trace-driven discrete simulator which was developed to study the scheduler Omega [5]. We have made additional extensions[5] to support the concepts of this work.

We use publicly available traces [10], [11], [50], [51], and generate a workload by sampling from the empirical distributions computed from such traces. Our workload is composed by 150,000 batch applications, both *rigid* (e.g. TensorFlow) and *elastic* (e.g. Apache Spark) variants. Applications are assigned a number of components ranging from a few to tens of thousands. The resource requirements of application components follow that of the input traces, ranging from a

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

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

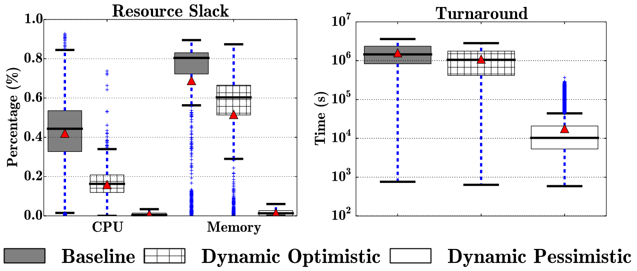


Fig. 2. Boxplots comparing baseline vs optimistic vs pessimistic approaches over different metrics, using an oracle in place of the prediction module. The red triangle is the mean.

few MB of memory to a few dozens of GB, and up to 6 CPU cores. Application *runtime* is generated according to the input traces, and ranges from a few dozens of seconds to several weeks (of simulated time). Inter-arrival times are drawn from the empirical distributions of the input traces, and exhibit a bi-modal distribution with fast-paced bursts, and longer intervals between application submissions.

We simulate a cluster consisting of 250 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.

The metrics we use to analyze the results include: application **turnaround**, which allows reasoning about system responsiveness, **resource slack**, measured as the difference of percentage of CPU and memory the scheduler allocates to each application compared to the percentage actually used by the application and **application failures**, which give us information about the aggressiveness of our approach.

As anticipated in Section III, statistical models are prone to prediction errors, which we address using the buffer β, which is a function of the uncertainty produced by the model. In our experiments we demonstrate the effect of the buffer parameters ($\beta = f(K_1, K_2)$) on the average turnaround, memory slack and application failures. The parameter K_1 controls the fraction of the resource request that is guaranteed to an application. The parameter $K_2 \in [0, 1, 2, 3]$ – in our experiments – is a multiplicative constant applied to the predictive variance output by a model. For such experiments, we report (in Figure 3a and Figure 3b) the ratio of the turnaround of our approach divided by the baseline.

B. Results

We first consider an ideal setup with an oracle having perfect information about future workload: this allows to determine an upper bound on the performance gains achieved by our approach. Then, we compare ARIMA and GP models, to study the impact of prediction errors on system performance.

Baseline. We compare our approach to a simple, but largely used baseline, the reservation centric approach first used by [5] (similar to Mesos and Yarn). This approach statically allocates the full amount of requested resource by an application. The baseline achieves the performance reported in Figure 2.

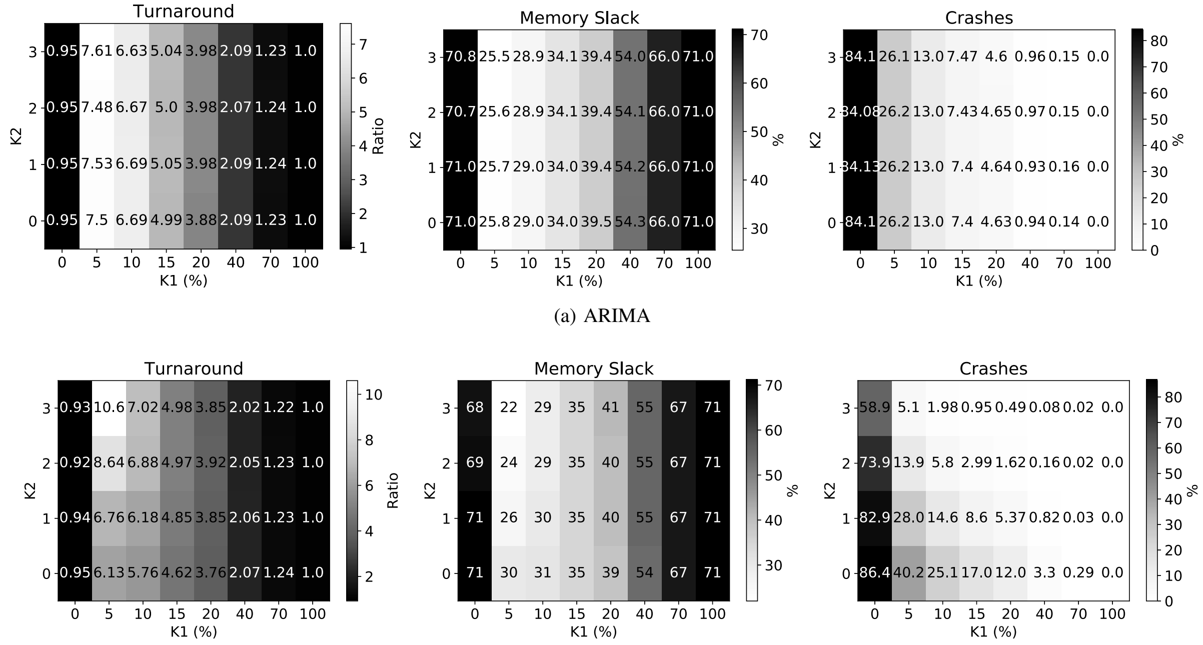
Oracle-based resource shaping. We gloss over prediction errors induced by a real statistical model and consider an ideal scenario from the forecasting point of view. Ultimately, our goal is to discern virtues and drawbacks of different preemption policies. Results are summarized in Figure 2: the plots correspond to resources slack and application turnaround, whereas each box correspond to the baseline and our resource shaping approach, with an optimistic (as originally implemented in the Omega simulator [5]) and our pessimistic preemption policy. Note that our simulator implements the concept of work lost upon failures and preemption.

Overall our results indicate that resource shaping brings substantial benefits in terms of all metrics we consider, in the absence of prediction errors. Cluster efficiency improves because resource slack, computed as the difference between allocated and used resources, drastically shrinks as shown in Figure 2 (left) compared to the baseline. Similarly, turnaround times are notably smaller as shown in Figure 2 (right) in comparison to the baseline. Indeed, the system can ingest new applications more quickly, because resources are better used.

Figure 2 can now be used to compare preemption policies, in the absence of prediction errors. While both approaches improve over the baseline, the pessimistic policy we introduce in this work outperforms the optimistic policy. As shown in Figure 2 (left), our policy induces the resource shaper to follow very closely application resource utilization: in this case, resource slack becomes negligibly small. This result explains why turnaround times, Figure 2 (right), are almost two orders of magnitude smaller with our policy: by freeing up resources, the application scheduler is amened to trigger new executions, thus queuing times shirk. Furthermore, we compute the number of application failures: in case of the optimistic policy we record 37.67% application failures, whereas with our policy no application fails because it avoids failures through partial preemption, by freeing elastic resources first.

ARIMA-based resource shaping. We focus on Figure 3a and slice it by row. When $K_2 = 0$ we omit uncertainty information and only consider the effects of a minimum guaranteed resource allocation. Even with just $K_1 = 5\%$, our approach achieves 7.5x average improvements in terms of application turnaround, while resource slack is only 30% in average. However, the number of crashed application is high: roughly 26% of applications experience a failure in average. The situation improves only for large values of K_1. When $K_1 = 100\%$, our method grants all requested resources: here no application fails, but turnaround times and slack exhibit no improvements on the baseline.

We note that the absence of a static term (i.e. $K_1 = 0\%$) results in a turnaround that is very close to the baseline regardless of K_2, due to the high number of applications failures which also lead to an high memory slack. This is a consequence of the occasional high confidence of the predictor in cases where a sudden change in the usage behavior occurs. It is necessary to maintain a static component to accommodate unexpected variations.



(a) ARIMA

(b) GP

Fig. 3. Heat maps showing the effect of K_1 and K_2, which compose β, on different metrics when using ARIMA and GP. Bright cells are better.

Finally, we focus on $K_1 = 5\%$: the minimum resource allocation is small, and we absorb prediction errors and fluctuations using uncertainty information. However, as K_2 increases, all metrics remain similar: the uncertainty produced by the ARIMA model is not sufficiently accurate to compensate forecasting errors.

GP-based resource shaping. Next, in Figure 3b we can see that while the GP model gives slightly worst results when ignoring uncertainty information ($K_2 = 0$) compared to ARIMA – because its error rates are slightly higher – as K_2 increases, all metrics improve: average turnaround ratios increase up to 10.6x, average slack is reduced to a 22% in average, while application failures quickly decrease. This is due to the accurate quantification of prediction uncertainty, which allows to follow more closely real utilization patterns.

Overall, the best performance is achieved when the system is flexible regarding the size of the buffer, i.e., a high value for its dynamic and a small value for its static components.

V. System Implementation

We materialize the ideas presented in this paper with a full-fledged, python-based, implementation of our mechanism, following the system design presented in Section III, and depicted in Figure 1. In our implementation, the resource shaper modulates both CPU and memory resources.

Back-end. We use Docker [8] as the back-end, which exposes several methods to redeem resources (CPU and memory) intially allocated to a container. We treat memory with care: upon a sudden spike in the memory utilization, containers might be killed by the OS, that checks memory limits. In our work we use soft-limits: in this case, the OS notifies the processes running in the container to free some of their resources to reduce memory pressure. 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 [26], which trade performance for a smaller memory footprint.

Monitoring module. It feeds the utilization forecast module with data at regular time intervals. Frequent updates ultimately result in better system efficiency, as the predictor operates 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 implementation, we collect resource utilization information every minute, which is in the same orders of what is used in [4].

Forecasting module. It implements the two models we discuss in Section III-A. For the ARIMA model we use the well-known `StatsModel` [52] library, which features an efficient implementation of the ARIMA model and its automatic parameter tuning through the Pyramid wrapper [53]. For the GP model we use the well-known library `GPy` [54]. Both models consider a small history of the ten past observations for training, to keep computational complexity under control.

Resource shaping module. It implements resource allocation, preemption and resizing, as outlined in Algorithm 1. It is important to point out that the resource shaper adapts resource

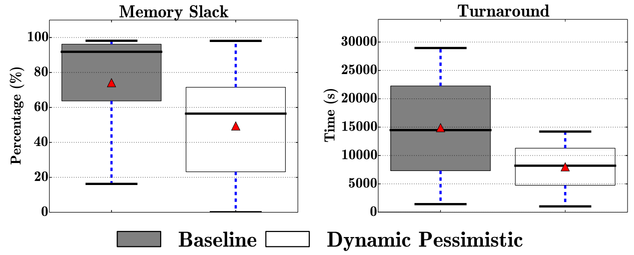


Fig. 4. Boxplots comparing baseline vs pessimistic dynamic approach over memory slack and turnaround time distributions using GP-based resource shaping. The red triangle is the mean.

allocations only after enough historical data points are available for the forecasting module: we call this a **grace period**, and set it to 10 minutes in our experiments.

The resource shaper uses the mechanisms exposed by Docker (as discussed above) to adjust application resources, and to eventually preempt components or entire applications. This module computes a new resource allocation for all running application in the system, based on the predicted value and variance obtained from the forecasting module. The buffer β is set to compensate for prediction uncertainty, using the parameters that we obtain through simulations, that is $K_1 = 5\%$ and $K_2 = 3$.

A. *Experimental Evaluation*

We have deployed the mechanism presented in this paper in our cluster (which we operate using [55]) to perform a comparative analysis between dynamic resource shaping and a reservation centric baseline, as done in Section IV.

Workload. In our experiments, we use a trace that we synthesize based on an analysis of our platform logs. The trace takes approximately 24 hours from the first submission to the completion of the last application. In our academic platform, users submit two kinds of applications: Apache Spark jobs (representative of elastic frameworks) and TensorFlow applications (representative of rigid frameworks).

Our trace includes 60% of elastic and 40% of rigid applications for a total of 100 applications. Application interarrival times follow a Gaussian distribution with parameters $\mu = 120$ sec, and $\sigma = 40$ sec, which is compatible with what we observe in our cluster logs.

An application to submit is chosen among two Spark jobs that implement a recommender algorithm using alternating least squares and a random-forest regression model respectively, a TensorFlow application that implements an approximate deep GP model [56] and an Extract, Transform and Load (ETL) application that uses SparkSQL. 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. The TensorFlow application, instead, requests 1 worker and 8-16-32GB of RAM depending on the flavor.

Experimental setup. We run our experiment on our academic platform with 20 server-grade machines, using Ubuntu 14.04 and Docker 17.09.0. Docker images for the applications are preloaded on each machine to prevent startup delays and network congestion.

Summary of results. We compare our dynamic resource shaper mechanism with our pessimistic preemption policy to the typical reservation centric baseline, which simply allocates all requested resources and never modifies such allocation. Overall, the our approach largely outperforms the baseline. We measure substantial improvements in terms of resource allocation: indeed our system can afford to ingest more applications, that would otherwise wait to be served. Figure 4 (left) illustrates resource slack, which is roughly 40% lower with our resource shaping mechanism. As a consequence, applications spend less time in the scheduler queue and have short turnaround times, as shown in Figure 4 (right). The median turnaround times are $\sim 50\%$ shorter. Note also that the tails of the distributions are in favor of our approach. Finally, we report that no application, nor component failed due to resource shaping.

VI. 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 to a variety of business-critical and scientific applications. However, the current resource reservation model hinders an efficient use of cluster resources. Resource utilization dynamics induce over-provisioning, which is one of the main culprit of poor efficiency. The problem of underutilization 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 call for the design failure tolerant applications, due to the ephemeral nature of the resources they are offered.

In this work, we presented a mechanism that cooperates with a scheduler to dynamically adjust resources allocated to an application, so that they closely match those they actually use throughout their lifecycle. Our design featured: a method to build a statistical model to forecast resource utilization, and a preemption policy that reallocates system resources while minimizing failures.

We have validated our mechanism with an thorough experimental campaign, both in simulation and using a real implementation. Our simulations shed lights on the key role played by the ability to model and use prediction uncertainty, and by the a strict preemption policy to manage concurrency issues. We implemented a system prototype of our dynamic allocation mechanism and deployed it in our academic platform, where we executed a real workload. Results indicate notably improved system efficiency, which translates in better system responsiveness.

References

[1] M. Babaioff *et al.*, "Era: A framework for economic resource allocation for the cloud," in *Proceedings of the 26th International Conference on World Wide Web Companion*. International World Wide Web Conferences Steering Committee, 2017, pp. 635–642.

[2] Y. Yang *et al.*, "Pado: A data processing engine for harnessing transient resources in datacenters," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 575–588.

[3] J. Rasley *et al.*, "Efficient queue management for cluster scheduling," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 36.

[4] A. Verma *et al.*, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 18.

[5] M. Schwarzkopf *et al.*, "Omega: flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 351–364.

[6] C. Curino *et al.*, "Reservation-based scheduling: If you're late don't blame us!" in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–14.

[7] G. Ananthanarayanan *et al.*, "True elasticity in multi-tenant data-intensive compute clusters," in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 24.

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

[9] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. of the USENIX NSDI 2011*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488

[10] C. Reiss *et al.*, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 7.

[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 *et al.*, "Dominant resource fairness: Fair allocation of multiple resource types," in *NSDI*, vol. 11, no. 2011, 2011, pp. 24–24.

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

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

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

[16] Y. Yan *et al.*, "Tr-spark: Transient computing for big data analytics," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 2016, pp. 484–496.

[17] O. Alipourfard *et al.*, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *NSDI*, 2017, pp. 469–482.

[18] F. Pace *et al.*, "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] 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*. IEEE, 2016, pp. 80–89.

[20] ——, "Dynamically scheduling a component-based framework in clusters," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2014, pp. 129–146.

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

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

[23] R. Grandl *et al.*, "Multi-resource packing for cluster schedulers," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 455–466.

[24] D. Lo *et al.*, "Heracles: improving resource efficiency at scale," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 450–462.

[25] M. Shahrad *et al.*, "Incentivizing self-capping to increase cloud utilization," in *ACM Symposium on Cloud Computing 2017 (SoCC'17)*. Association for Computing Machinery (ACM), 2017.

[26] 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.

[27] P. Padala *et al.*, "Adaptive control of virtualized resources in utility computing environments," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 289–302.

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

[29] E. Boutin *et al.*, "Apollo: Scalable and coordinated scheduling for cloud-scale computing." in *OSDI*, vol. 14, 2014, pp. 285–300.

[30] K. Karanasos *et al.*, "Mercury: Hybrid centralized and distributed scheduling in large shared clusters." in *USENIX Annual Technical Conference*, 2015, pp. 485–497.

[31] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM, 2016, pp. 50–56.

[32] Y. Zhang *et al.*, "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*, no. EPFL-CONF-224446, 2016, pp. 755–770.

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

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

[35] H. Nguyen *et al.*, "AGILE: Elastic distributed resource scaling for infrastructure-as-a-service," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX, 2013.

[36] P. Brockwell *et al.*, *Introduction to Time Series and Forecasting, Second Edition*. Springer, 2002.

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

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

[39] A. McHutchon, "Nonlinear Modelling and Control using Gaussian Processes," Ph.D. dissertation, University of Cambridge, 2015.

[40] R. Frigola-Alcalde, "Bayesian Time Series Learning with Gaussian Processes," Ph.D. dissertation, University of Cambridge, 2015.

[41] R. Frigola, Y. Chen, and C. E. Rasmussen, "Variational Gaussian Process State-Space Models," in *Advances in Neural Information Processing Systems*. MIT Press, 2007.

[42] A. Svensson, A. Solin, S. Särkkä, and T. Schön, "Computationally Efficient Bayesian Learning of Gaussian Process State Space Models," in *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, vol. 51. PMLR, 2016, pp. 213–221.

[43] Pyramid, "Auto-arima," http://pyramid-arima.readthedocs.io.

[44] R. Documentation, "Auto-arima," https://www.rdocumentation.org/.

[45] E. Snelson and Z. Ghahramani, "Sparse gaussian processes using pseudo-inputs," in *Proceedings of the 18th International Conference on Neural Information Processing Systems*, ser. NIPS. Cambridge, MA, USA: MIT Press, 2005, pp. 1257–1264. [Online]. Available: http://dl.acm.org/citation.cfm?id=2976248.2976406

[46] 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.

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

[48] 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, no. 1, pp. 333–350, Feb. 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2502581.2502592

[49] M. Pastorelli, M. Dell'Amico, and P. Michiardi, "Os-assisted task preemption for hadoop," in *Distributed Computing Systems Workshops (ICDCSW), 2014 IEEE 34th International Conference on*. IEEE, 2014.

[50] C. Reiss *et al.*, "Google cluster-usage traces: format + schema," Google Inc., Technical Report, Nov. 2011.

[51] Google, "Google traces," https://github.com/google/cluster-data.

[52] S. Seabold and J. Perktold, "Statsmodels: Econometric and statistical modeling with python," in *9th Python in Science Conference*, 2010.

[53] GitHub, "Pyramid," https://github.com/tgsmith61591/pyramid.

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

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

[56] 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.