Sonnet: A Control-Theoretic Approach for Resource Allocation in Cluster Management

Ruifeng Ma[a], Yufeng Zhan[a], Yuanqing Xia[a,*], Chuge Wu[a], Liwen Yang[a] and Runze Gao[a]

[a] *Beijing Institute of Technology, Beijing, 100081, China*

ARTICLE INFO

Keywords:
dynamic applications
resource allocation
cluster management
model-free adaptive control

ABSTRACT

Cluster users expect to minimize the resource costs while ensuring target performance for different applications. It is particularly difficult to reach such a goal, because the applications are diverse with dynamic load changes, and interference exists between them. In addition, the performance of the applications depends on heterogeneous resources with different costs. However, existing works either use simplistic and generalized heuristics that disregard resource-specific characteristics or need suspending service to get expert knowledge to optimize the resource cost for a brand-new application or runtime, which fails to optimize the resource allocation finely.

In this paper, we propose Sonnet, a control-theoretic approach to perform efficient resource allocation. Sonnet can efficiently optimize the cost of resources while satisfying the SLO by quickly establishing new application performance models through only online profiling and without affecting service. Experiments on Docker Swarm using various open-source benchmarks demonstrate that Sonnet can decrease the SLO violation rate by 91% while reducing resource costs up to 47% compared with the state-of-the-arts.

1. Introduction

Cluster users need to declare how many resources are needed for their applications in the cluster. In the Kubernetes cluster, users set the resource limits for each pod; in public clouds, users need to choose the type of virtual machines (VMs) they will rent. Edge computing can reduce data transmission latency and prevent user data from leaking into the cloud. However, users need to allocate resources more sophisticatedly due to the limited resources on the edge. Besides, latency-sensitive applications executed in clusters need to reach latency service level objectives (SLO). However, it is difficult to allocate appropriate resources to applications. Firstly, the workloads in clusters with inherent dynamic behavior and the interference between co-located workloads can frequently lead to SLO violations. Secondly, tasks in the cluster have different characteristics, and the generalized approach will lead to suboptimal resource allocation. Thirdly, computing resources at the edge are limited, requiring more sophisticated solutions to reduce resource waste. This makes resource allocation extremely difficult, and inefficient resource allocation strategies will lead to resource over-provision or a large number of SLO violations.

Some previous works have been proposed to reduce SLO violations while improving resource utilization. Firstly, some works [1, 2, 3, 4, 5] use simplistic heuristics that disregard specific characteristics. Tasks in the cluster have different characteristics, and the generalized approach will lead to suboptimal resource allocation. Secondly, other studies [6, 7, 8] need to suspend the service to get expert knowledge to optimize the resource allocation. However, the offline profiling will extend the response time. Therefore, we need a repaid method that can dynamically adapt to load changes and achieve fast allocation among various resources with cost-effectiveness.

In this paper, we propose Sonnet, a control-theoretic approach for automated resource allocation that reduce users' costs without degrading the generality and performance of applications. It can allocate resources in a cost-effectively way for a brand new application without offline profiling or prior knowledge. Firstly, Sonnet defines an SLO for each application. With SLO, users only need to specify latency, execution time, and other related metrics that can be measured during application execution. This decouples the users from the underlying resource allocation loop, which is quite complex and beyond the users' capabilities.

Then, Sonnet estimates the amount of resources to achieve target SLO with cost-effectiveness using a modified model-free adaptive control (MFAC) [9] algorithm without taking into consideration the structure of the system itself. MFAC is suitable for multiple-input and multiple-output (MIMO) systems, which adjusts the amount of multiple resources based on the difference between the observed performance and the target performance. Essentially, Sonnet efficiently and swiftly constructs an online mapping model between the target SLO and the necessary multiple resources, all while ensuring uninterrupted regular service. The online model enables Sonnet to adapt to dynamic load changes. Compared to existing methods, on the one hand, MFAC can be applied for MIMO systems and is suitable for multiple resource allocation, while Skynet [3] uses proportional integral derivative (PID) for single-input and single-output (SISO) systems and cannot capture which resource the current application is more sensitive to. Therefore, MFAC can allocate resources with cost-effectiveness. On the other hand, MFAC is an online adaptive approach

*Corresponding author. School of Automation, Beijing Institute of Technology, Beijing, 100081, China.

mrf@bit.edu.cn (R. Ma); yu-feng.zhan@bit.edu.cn (Y. Zhan); xia_yuanqing@bit.edu.cn (Y. Xia); wucg@bit.edu.cn (C. Wu); ytyangliwen@163.com (L. Yang); runze_gao@bit.edu.cn (R. Gao)

and does not require offline profiling, while reinforcement learning (RL) based methods [7] [8] usually need long-time offline data collecting and training. Additionally, in SLO measurement, noise is inevitable. We use a low-pass filter to filter the collected data. It can filter out the high-frequency noise of the measurement and avoid oscillation.

Finally, we implement our approach on Docker Swarm on HUAWEI cloud. We use Linux traffic control to avoid interference caused by network sharing between containers. Experiments have been conducted with several workload combinations to evaluate the effectiveness of our proposed approach. Our approach can adaptively handle different workloads online. Experimental results show that, compared with the state-of-the-arts, Sonnet can decrease SLO violation rate by 91% while reducing resource costs up to 47%. The main contributions of this paper are summarized as follows:

- We design an end-to-end online resource allocation system named Sonnet, which does not require offline profiling or prior expert knowledge.
- We use an MFAC algorithm to build a dynamic mapping model between the target SLO and resources, and optimize the resource costs.
- We implement Sonnet on Docker Swarm, as a custom SLO-based cluster management tool.

The rest of this paper is organized as follows. Section 3 gives the motivation of this paper. In Section 4, we present the design and implementation details of Sonnet. The performance of the proposed approach on various open-source benchmarks is evaluated by comparing it with the state-of-the-arts in Section 5. Section 2 discusses the related works and Section 6 concludes the paper.

2. Related Work

In this section, we discuss related works of resource allocation. We categorize the related work into two main classes: analyze and modeling method and learning based method.

2.1. Analyze and Modeling Method

For online resource estimation, we usually observe some metrics of the container and perform vertical scaling of the container according to these metrics. Skynet [3] applies PID controllers to estimate the resource demands. The application performance metric, such as latency, is observed and set as a control variable. PID controller is an SISO controller and the internal relations between multiple PID controllers cannot be established. Therefore, Skynet is not optimal. Dechouniotis et al. [19] employs admission control for regulation. The offline phase includes identifying the linear parameter-varying (LPV) system and setting operational points, while the online phase includes optimizing the resources. The taxonomy [22] comprised of relevant attributes defining the following two perspectives, i.e., control-theory as an implementation technique as well as cloud elasticity as a target application domain.

Parties [6] and Heracles[17] consider the trade off of High-dimensional resource configurations. But they need to test many times and select the resources that can reach the target performance of the application. The multiple testing method is time-consuming, which significantly reduces the response speed.

Similar to Kubernetes HPA, Autopilot [4] monitors previous resource usage of multiple pods rather than SLO. It adjusts the resource allocation based on the carefully designed rules. However, designing the rules requires professional experience and lots of tricks, which have great limitations and cannot be widely applied. Similar to autopilot, SHOWAR [5] monitors runq latency [23] which can reflect the intensity of CPU contention. However, it also needs carefully designed rules and professional experience.

Gandhi et al. [14] have used Kalman filtering [24] to model the queuing model of cloud application requests. However, the model does not consider the allocation of multiple resources.

Bashir et al. [10] estimate the peak resource consumption of the application and develop an overcommitment strategy based on this peak prediction. However, it faces the risk of inaccurate performance predictions.

Liu et al. [15] propose that the number of threads for an application should be adjusted after autoscaling is applied. They fit model offline. However, they do not consider the high-dimensional optimization of CPU, memory, and network bandwidth resources, and offline optimized models cannot adapt to online fluctuating systems. Similarly, StepConf [16] describes the offline model of replicas and application performance, but in the online environment, due to the contention of underlying hardware resources, the performance of the application will be affected, and the offline model is not accurate. Parslo [13] also uses the offline model and cannot be applied to vertical scaling.

2.2. Learning Based Method

Rossi et al. [8] have used RL for resource allocation. To speed up the training process, they adopted a model-based RL approach, which, however, introduces modeling errors and cannot avoid online disturbances. Similarly, FIRM [7] also applies RL methods. It trains the algorithm on real clusters, but this leads to excessively long training times.

Besides, Yu et al. [11] use Bayesian optimization to fit the performance model online, but this required multiple tests and data collection, which reduced the response time of the system. DeepRest[12] estimates application resource utilization using an neural network based on attention mechanism. But this method cannot guarantee SLO. Horn et al.[18] use machine learning (ML) methods to model the relationship between application performance and resource allocation offline. But the optimality of its scaling algorithm cannot be guaranteed. Sinan[21] fully collects offline data to predict the long-tail delay, and heuristically adjusts resource allocation according to the established ML model.

Table 1
Summary of the available choices of a resource management system.

Work	SLO Guar.	Resrc. Type	High Dim. Resrc. Optim.	Model Online Adapt.	Reduce Noise
Autopilot, 2020 [4]	✗	Multiple	✗	✗	✔
Bashir et al. , 2021 [10]	✗	Multiple	✗	✗	✔
K8s HPA	✗	Multiple	✗	✗	✗
Skynet, 2021 [3]	✔	Multiple	✗	✔	✗
PARTIES, 2019 [6]	✔	Multiple	✗	✗	✗
Microscaler, 2019 [11]	✔	Single	✗	✔	✗
FIRM, 2020 [7]	✔	Multiple	✔	✗	✗
DeepRest, 2022 [12]	✗	Multiple	✔	✗	✗
Rossi et al. , 2019 [8]	✗	Single	✗	✗	✗
Parslo, 2021 [13]	✔	-	✗	✗	✗
SHOWAR, 2021 [5]	✗	Multiple	✗	✗	✔
Gandhi et al. , 2014 [14]	✔	Single	✗	✔	✔
Liu et al., 2022 [15]	✔	Multiple	✔	✗	✗
StepConf, 2022 [16]	✔	Multiple	✔	✗	✗
Heracles, 2015 [17]	✔	Multiple	✗	✗	✗
Horn et al. , 2022 [18]	✔	Multiple	✗	✗	✗
Dechouniotis et al. , 2015 [19]	✔	Multiple	✗	✗	✗
Spatharaki et al. ,2022 [20]	✔	Multiple	✗	✗	✗
Sinan et al. , 2021 [21]	✔	Multiple	✗	✗	✗
Sonnet	✔	Multiple	✔	✔	✔

Spatharaki et al. [20] dynamically redirects incoming requests using a novel AIMD-like task scheduling solution to manage varying workloads. To handle dynamic workloads, a prediction mechanism estimates the number of incoming requests. Additionally, a ML-based application profiling model is introduced to address scalability issues.

Existing approaches either rely on simplistic and general heuristics that ignore resource-specific characteristics or require service suspension to obtain expert knowledge for optimizing resource costs for new applications or runtimes, failing to finely optimize resource allocation.

In this paper, we propose Sonnet, which can efficiently optimize the cost of resources while satisfying the SLO by quickly establishing new application performance models through only online profiling and without affecting service.

3. Motivation

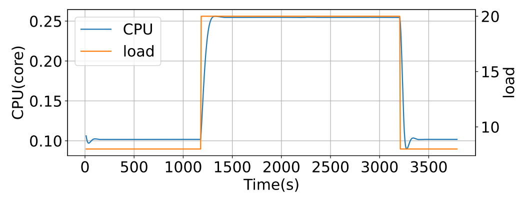
A resource management system has a straight-forward goal: assign the minimum amount of resources for each application to reach a target performance. Therefore, resource managers need to deal with diverse applications, various application loads, high-dimensional resources (e.g., CPU, memory, etc), and inter-applications interference. However, in the existing system, the amount of resources is often manually specified by the users. To avoid SLO violations and out of memory (OOM) errors caused by insufficient resource allocation, operators tend to over-allocate resources, which can lead to higher costs.

To understand the coupling relationship between SLO (e.g., latency) and high-dimensional resources, we have conducted some experiments on widely used application benchmarks (i.e. Memtier Benchmark [25], Apache HTTP server benchmarking tool [26], etc). Our key insights are as follows.

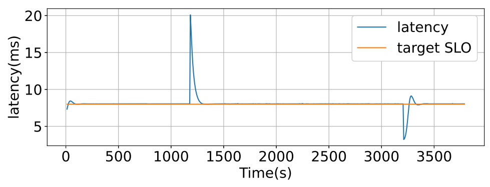
3.1. Load Changes

Typical load of a online application has inherent dynamic behavior. For example, the number of visits to Google is changing all the time. There is an internal relationship between load and resources. We need to adjust resource allocation after load changes.

As shown in Fig. 1, the load of Nginx in an online application changes dynamically. In Fig. 1, the unit of CPU is the number of cores. We define the unit of CPU as the number of cores. Specifically, when we mention 0.2 CPU, it implies that the container is capable of utilizing up to 20% of the time of a single CPU core. The load is the concurrency of Nginx, which means the number of requests at the same time. We change the loads of Nginx by changing the concurrency.

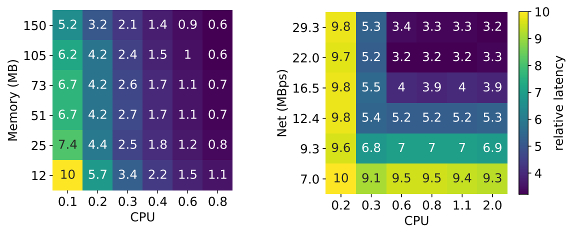


(a) Dynamic CPU and load



(b) The latency of Nginx and SLO

Figure 1: Load changes cause SLO violations in Nginx.



Memory (MB)	0.1	0.2	0.3	0.4	0.6	0.8
150	5.2	3.2	2.1	1.4	0.9	0.6
105	6.2	4.2	2.4	1.5	1	0.6
73	6.7	4.2	2.6	1.7	1.1	0.7
51	6.7	4.2	2.7	1.7	1.1	0.7
25	7.4	4.4	2.5	1.8	1.2	0.8
12	10	5.7	3.4	2.2	1.5	1.1

Net (MBps)	0.2	0.3	0.6	0.8	1.1	2.0
29.3	9.8	5.3	3.4	3.3	3.3	3.2
22.0	9.7	5.2	3.2	3.2	3.2	3.3
16.5	9.8	5.5	4	3.9	4	3.9
12.4	9.8	5.4	5.2	5.2	5.2	5.3
9.3	9.6	6.8	7	7	7	6.9
7.0	10	9.1	9.5	9.5	9.4	9.3

(a) ML (b) Memcached

Figure 2: Applications have different sensitivities to different kinds of resources.

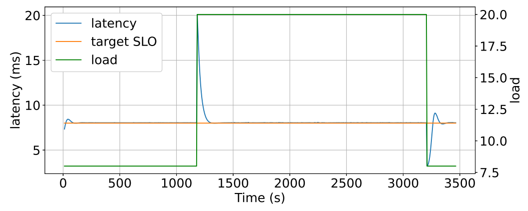
When resource allocation is fixed, latency will increase as the load rises.

We can find that when the load decreases, the CPU resource should be scaled down. When the load increases, the CPU resource should be expanded. For example, at about 1200 seconds, the load of Nginx is increased from 8 to 20, resulting in an increase of latency. Under this circumstance, we need to increase the resources (e.g., CPU) to ensure the achievement of target SLO. But at about 3200 seconds, we need to decrease the resources to reduce the costs. An efficient resource allocation framework should allocate resources to every application proactively, according to the load changes. However, it is difficult or even impossible to model the relationship between load and resources.

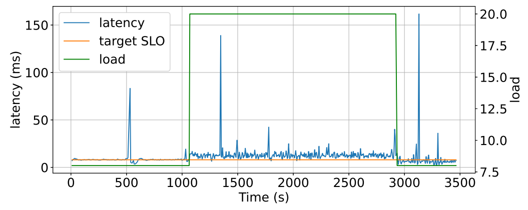
3.2. High-Dimensional Resource Allocation

The performance of an application is influenced by a variety of resources. The sensitivity of applications to CPU and memory is different. The performance bottlenecks of the applications vary among different resources as the load changes. It is difficult to derive the optimal resource allocation strategy.

The workload of ML is Random Forest training. As shown in Fig. 2(a), the sensitivity of ML to CPU and memory are different. For example, the performance of 0.6 CPU and 12 MB memory is approximately equivalent to that of 0.4 CPU and 105 MB memory. When the CPU is reduced by 1.5 times, the memory needs to be expanded by 8.75 times to keep the same performance. As shown in Fig. 2(b), when the network bandwidth is 9.3MBps, increasing the CPU from 0.2 to 0.3 can bring Memcached [27] performance improvement. When we further increase the CPU, the performance of the application does not improve when the network bandwidth remains 9.3MBps. And the performance bottleneck changes from the CPU to the network bandwidth.



(a) Sonnet with network interference isolator



(b) Remove Sonnet network interference isolator

Figure 3: Latency spikes on applications due to low-level resource contention.

3.3. Interference between Applications

Although there are various resource isolation technologies such as container (e.g., Docker [28]). However, multiple containers on the same machine multiplex resources, such as network link, memory, CPU, and others, will have interference for a shared resource. The interference between applications will increase the SLO violations seriously. As shown in Fig. 3, we take network interference as an example. With the same load, if Sonnet considers resource contention, the latency of Nginx remains stable and can be well controlled near the target SLO. However, if the network interference isolator in Sonnet is removed, a large number of SLO violations occur, and the latency is unstable. As shown in Fig. 3, the latency can exceed 150 ms, which is 19 times larger than the target SLO.

As mentioned before, how to allocate the resources with the minimal costs to each application satisfying the target SLO is very difficult. But a well-designed resource allocator can not only ensure the target SLO, but also significantly reduce user costs. Therefore, it is necessary for the resource allocator to dynamically adjust itself according to the changes of the system in real time to find an efficient allocation strategy.

4. Sonnet Design

Sonnet has a straight-forward goal: given a fixed set of resources with predefined prices. Then, assigning the resources with minimal cost to each application which can reach the acceptable performance online. To achieve this goal, we need to solve the optimization problem which establishes the relationship between SLO and resource cost. The control objective can be formulated as

$$\min \left[\left(y^*(k+1) - y(k+1) \right)^2 + \boldsymbol{\varphi} \boldsymbol{u}(k) \right], \qquad (1)$$

where the k refers to time steps. y^* represents the target SLO, and y represents the measured performance, both of which are 1×1 scalars. $\boldsymbol{u}$ denotes the allocation of resources, encompassing CPU, memory, and network bandwidth, represented as a 3×1 vector. $\boldsymbol{\varphi}$ is the cost vector, representing the price associated with each resource, and it is a 1×3 vector. Users can scale the value of $\boldsymbol{\varphi}$ proportionally to balance performance and cost. If performance guarantees are more important, the value of $\boldsymbol{\varphi}$ can be decreased proportionally. If resource cost is more important, the value of $\boldsymbol{\varphi}$ can be increased proportionally. The total resource cost is $\boldsymbol{\varphi}\boldsymbol{u}(k)$, and $(y^*(k+1) - y(k+1))^2$ represents the error between the measured latency and target SLO.

As mentioned in Section 3, resource allocator needs to consider load changes of application, interference between applications, and high-dimensional resource allocation. In this paper, we design the system architecture of Sonnet as shown in Fig. 4. The model estimator and controller are the two core components of Sonnet. With model estimator and controller, we build the modified MFAC algorithm to control the system. Particularly, the model estimator identifies the system model with dynamic linearization [9]. According to the identified model and the custom resource cost function, the controller outputs the volume of resources with minimal cost to reach the target SLO. The detailed design of model estimator and controller will be discussed in the following sections.

In this paper, we implement Sonnet on Docker Swarm. Note that Docker has no option to manage the network bandwidth at present [29]. As mentioned in Section 3.2, the contention of network bandwidth will cause interference. Therefore, we use Linux traffic control [30] to build a network isolator to avoid bandwidth interference. Latency measurement is often mixed with noise, which will seriously affect the performance of Sonnet. In this paper, we use a low pass filter to eliminate measurement noise.

4.1. Model Estimator

In this section, we introduce how to use the dynamic linearization to model an application as a multiple-input and single-output (MISO) system. The dynamic linearization method, as demonstrated in Fig. 5, can rapidly adjust the estimated value of the performance model to accommodate varying resource allocations, approximating the nonlinear system in segmented intervals. Dynamic linearization shows great model identification performance and has been widely used in practical systems, such as unmanned surface vehicles [31], autonomous cars [32], quadrotor aircraft [33], wide-area power systems [34], etc. It provides state estimation for the MFAC algorithm.

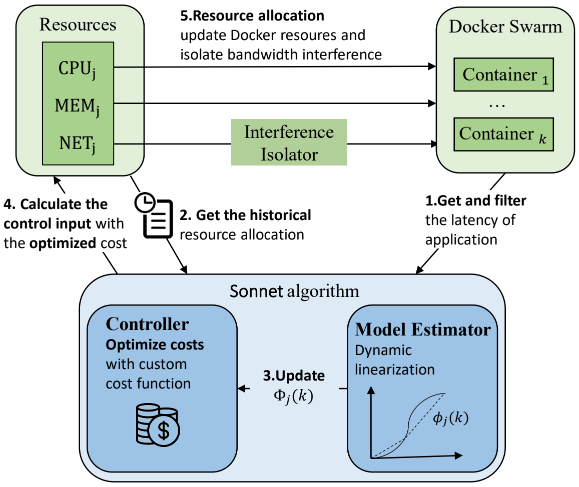

Resources
CPU$_j$
MEM$_j$
NET$_j$
5.Resource allocation
update Docker resoures and isolate bandwidth interference
Interference Isolator
Docker Swarm
Container $_1$
…
Container $_k$
4. Calculate the control input with the optimized cost
2. Get the historical resource allocation
1.Get and filter the latency of application
Sonnet algorithm
Controller
Optimize costs with custom cost function
3.Update $\Phi_j(k)$
Model Estimator
Dynamic linearization
$\phi_j(k)$

Figure 4: The architecture of Sonnet.

Table 2
Main Notations

y^*	the target SLO
y	the measured performance
$\boldsymbol{\varphi}$	the cost vector for the resources
k	the time steps
u	the allocation of resources
$\boldsymbol{f}(\cdot)$	the application system
$\boldsymbol{\Phi}$	the PJM of the application system
n_u, n_y	the orders of input and output
μ	the parameter to penalize changes in PJM
λ	the parameter to penalize changes in $\boldsymbol{u}(k)$
M	the number of points in moving average filter.
y_{out}	the output of the filter.
y_{in}	the input of the filter.
m_1	the concurrency before the load change.
m_2	the concurrency after the load change.
τ	the first-order low-pass filter coefficient.
T	the sampling period of the latency.

As shown in Fig. 4, the resource allocation system is a MISO system. It can be regarded as a discrete-time nonlinear system as

$$y(k+1) = f\left(y(k), \cdots, y(k-n_y), \boldsymbol{u}(k), \cdots, \boldsymbol{u}(k-n_u) \right), \quad (2)$$

where $y(k) \in \mathrm{R}$ represents the output of the system at time k (eg., the service performance of the application). $\boldsymbol{u}(k) \in \mathbf{R}^m$ represents the input of the system (eg., the m-type resources to be allocated, which is a column vector of size m). The n_y and n_u are the orders of input and output, and $\boldsymbol{f}(\cdot)$ is the system function for application.

We assume that $\boldsymbol{f}(\cdot)$ in Eqn. (2) satisfies the generalized Lipschitz condition, which means for $\forall k_1 \neq k_2$, the $k_1, k_2 \geq$

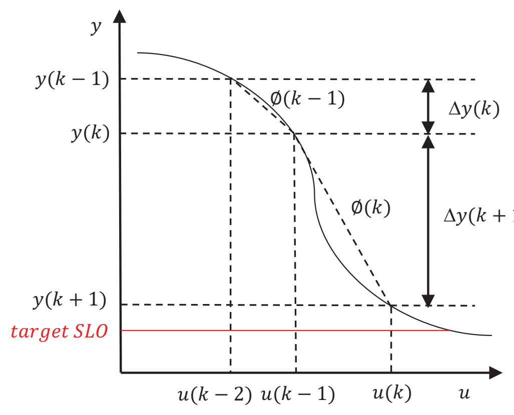


Figure 5: PPD models the relationship between resource allocation and quality of service (latency)

0 and $\boldsymbol{u}(k_1) \neq \boldsymbol{u}(k_2)$, it has

$$\left|y\left(k_1+1\right)-y\left(k_2+1\right)\right| \leqslant b\left\|\boldsymbol{u}\left(k_1\right)-\boldsymbol{u}\left(k_2\right)\right\|, \quad (3)$$

where $y(k_i+1) = \boldsymbol{f}(y(k_i), \ldots, y(k_i-n_y), \boldsymbol{u}(k_i), \ldots, \boldsymbol{u}(k_i-n_u)), i = 1, 2$. b is an arbitrary positive constant that constrains the rate at which the function can change.

When the partial derivatives of $\boldsymbol{f}(\cdot)$ with respect to all variables are continuous, then the system defined in Eqn. (2) can be transformed into a linear system as [9]

$$\Delta y(k+1) = \boldsymbol{\Phi}(k)\Delta\boldsymbol{u}(k) \quad (4)$$

where $\boldsymbol{\Phi}(k) = (\phi_{11}(k), \phi_{12}(k), \phi_{13}(k)) \in \mathbf{R}^{1\times 3}$ is the pseudo jacobian matrix (PJM) of the system in Eqn. (2), which is a 1×3 vector. $\Delta y(k+1)$ is equal to $y(k+1) - y(k)$ and $\Delta\boldsymbol{u}(k)$ is equal to $\boldsymbol{u}(k) - \boldsymbol{u}(k-1)$.

In order to show a clear description of the above method, we take a SISO nonlinear system as an example. For SISO system, PJM can be written as pseudo-partial derivative (PPD) $\phi_c(k) \in$ R. PPD represents the derivative of the nonlinear function $\boldsymbol{f}(\cdot)$ between $\boldsymbol{u}(k-1)$ and $\boldsymbol{u}(k)$. The geometric interpretation of PPD is shown in the Fig. 5. The segmented dotted line represents the dynamic linearized model established with the information of historical resource allocation and performance (e.g., latency). The boundedness of PPD means that the nonlinear function does not undergo abrupt changes, that is, it is a bounded derivative value. This bounded condition also coincides with in the resource allocation system.

To estimate PJM, we minimize the following parameter estimation criterion function [9] as

$$\begin{aligned} P\left(\boldsymbol{\Phi}(k)\right) = &(\Delta y(k) - \boldsymbol{\Phi}(k)\Delta\boldsymbol{u}(k-1))^2 \\ &+ \mu\|\boldsymbol{\Phi}(k) - \hat{\boldsymbol{\Phi}}(k-1)\|^2, \end{aligned} \quad (5)$$

where $\mu > 0$ is a parameter used to penalize excessive changes in PJM. $\Delta y(k)$ and $\Delta\boldsymbol{u}(k-1)$ in Eqn. (5) represent $y(k) - y(k-1)$ and $\boldsymbol{u}(k-1) - \boldsymbol{u}(k-2)$ respectively. In order to find the minimum of Eqn. (5), we set the derivation of $P\left(\boldsymbol{\Phi}(k)\right)$ equal to zero, the estimation of PJM [9] can be obtained by Eqn. (6)

$$\begin{aligned} \hat{\boldsymbol{\Phi}}(k) = &\hat{\boldsymbol{\Phi}}(k-1) \\ &+ \frac{\eta\left(\Delta y(k) - \hat{\boldsymbol{\Phi}}(k-1)\Delta\boldsymbol{u}(k-1)\right)}{\mu + \|\Delta\boldsymbol{u}(k-1)\|^2} \\ &\times \Delta\boldsymbol{u}^{\mathrm{T}}(k-1), \end{aligned} \quad (6)$$

where η is a step length. In this paper, we set $\eta = 1$ and $\mu \in (0, 15]$.

4.2. Controller Design

As mentioned in Eqn. (1), we need to solve the optimization problem to meet the target SLO while saving costs. For a control system, we must ensure that the control input should not change too rapidly, otherwise the performance of the whole control system will be greatly affected, resulting in uncontrollable. The control objective of the original MFAC algorithm is $J(\boldsymbol{u}(k)) = (y^*(k+1) - y(k+1))^2 + \lambda\|\boldsymbol{u}(k) - \boldsymbol{u}(k-1)\|^2$. We add the resource cost $\boldsymbol{\varphi u}$(k) into the control objective for modified MFAC. This aims to find a low cost setting among several different resource configurations that can reach the same target SLO. Therefore, we set the optimization goal as

$$\begin{aligned} J(\boldsymbol{u}(k)) = &\left(y^*(k+1) - y(k+1)\right)^2 \\ &+ \lambda\|\boldsymbol{u}(k) - \boldsymbol{u}(k-1)\|^2 + \boldsymbol{\varphi u}(k), \end{aligned} \quad (7)$$

where the first term of Eqn. (7) is to guarantee target SLO, $\lambda > 0$ is a parameter used to penalize excessive changes in $\boldsymbol{u}(k)$, and the third term is to optimize total cost of resources. This regularization term $\lambda\|\boldsymbol{u}(k) - \boldsymbol{u}(k-1)\|^2$ helps to maintain smooth variations in the control process and prevents excessive oscillations. It can achieve a balance between smoothness and speed of the control response by appropriately choosing λ.

This regularization term $\lambda\|\boldsymbol{u}(k) - \boldsymbol{u}(k-1)\|^2$ can also avoid the shock of the control quantity caused by the excessive change of $\boldsymbol{u}(k)$, and the instability caused by the excessive adjustment of resources in the computing system.

The modified MFAC both tracks the target SLO and optimizes the cost. However, to achieve new control objectives, we need to redesign the control policy. The control objective of the modified MFAC is a convex function. Thus, the objective function is minimized at the point where the derivative is zero. Therefore, we take the derivative of the objective function and set it equal to zero. Setting $\Delta y(k+1) = y(k+1) - y(k)$ and $\Delta\boldsymbol{u}(k) = \boldsymbol{u}(k) - \boldsymbol{u}(k-1)$, Eqn. (7) is equal to

$$\begin{aligned} J(\boldsymbol{u}(k)) = &\left\|y^*(k+1) - y(k) - \Delta y(k+1)\right\|^2 \\ &+ \lambda\|\Delta\boldsymbol{u}(k)\|^2 + \varphi\boldsymbol{u}(\mathrm{k}). \end{aligned} \quad (8)$$

We define the control error as

$$e(k) = y^*(k+1) - y(k), \tag{9}$$

where y^* is the target latency, and $y(k)$ is the latency at time k. Substitute Eqn. (4) into Eqn. (8), then we have

$$\begin{aligned} J(\boldsymbol{u}(k)) &= (e(k) - \boldsymbol{\Phi}(k)\Delta\boldsymbol{u}(k))^T\,(e(k) - \boldsymbol{\Phi}(k)\Delta\boldsymbol{u}(k)) \\ &\quad + \lambda\Delta\boldsymbol{u}(k)^T\Delta\boldsymbol{u}(k) + \boldsymbol{\varphi}\boldsymbol{u}(\mathrm{k}) \\ &= e(k)^2 - 2e(k)\boldsymbol{\Phi}(k)\Delta\boldsymbol{u}(k) + \boldsymbol{\varphi}\boldsymbol{u}(\mathrm{k}) \\ &\quad + \Delta\boldsymbol{u}(k)^T\left(\lambda I + \boldsymbol{\Phi}^T(k)\boldsymbol{\Phi}(k)\right)\Delta\boldsymbol{u}(k). \end{aligned} \tag{10}$$

Since $\frac{\partial J}{\partial\Delta\boldsymbol{u}(k)} = \frac{\partial J}{\partial\boldsymbol{u}(k)}$, the derivation of $J(\boldsymbol{u}_k)$ is

$$\begin{aligned} \frac{\partial J}{\partial\boldsymbol{u}(k)} &= 2\left(\lambda I + \boldsymbol{\Phi}^T(k)\boldsymbol{\Phi}(k)\right)\Delta\boldsymbol{u}(k) \\ &\quad - 2e(k)\boldsymbol{\Phi}^T(k) + \boldsymbol{\varphi}^T. \end{aligned} \tag{11}$$

Minimizing $J(\boldsymbol{u}(k))$ with respect to $\boldsymbol{u}(k)$, we set the derivation of $J(\boldsymbol{u}(k))$ to 0. Since $\hat{\boldsymbol{\Phi}}(k)$ is the estimation value of $\boldsymbol{\Phi}(k)$, then we have

$$\Delta\boldsymbol{u}(k) = \left(\lambda I + \hat{\boldsymbol{\Phi}}^T(k)\hat{\boldsymbol{\Phi}}(k)\right)^{-1}\left(e(k)\hat{\boldsymbol{\Phi}}^T(k) - \frac{\boldsymbol{\varphi}^T}{2}\right). \tag{12}$$

To avoid complex matrix inversion operations and reduce computation time, we can use the matrix inversion lemma, the estimation of $\boldsymbol{u}(k)$ satisfies

$$\hat{\boldsymbol{u}}(k) = \hat{\boldsymbol{u}}(k-1) + \frac{\rho(e(k)\hat{\boldsymbol{\Phi}}^T(k) - \frac{\boldsymbol{\varphi}^T}{2})}{\lambda + \left\|\hat{\boldsymbol{\Phi}}(k)\right\|^2}, \tag{13}$$

where ρ is a step length. In this paper, we set $\rho = 1$ and $\lambda = 15$. Substituting Eqn. (6) into Eqn. (13), the control input of Sonnet has been calculated. Then, Sonnet applies Eqn. (13) to allocate resources for each application. By this method, Sonnet can not only ensure the application to reach the desired SLO, but also minimize the resource costs.

As shown in Fig.1 in Section 3.1, when the load (the concurrency of request) changes, the application latency will change drastically. We want the control error of the latency to be limited and gradually decrease with Sonnet's algorithm. Therefore, we analyze the limit of the latency and the trend of the error.

The results of the analysis are as follows: consider the request concurrency changes from from m_1 to m_2 at time k_{change}. When the entries in $\boldsymbol{\varphi}$ are small enough, for $\forall k > k_{change}$, the latency y_2 satisfies

$$y_2(k) \leq \frac{1+m_2}{1+m_1}y_1, \tag{14}$$

where y_1 is the latency before the sudden change of concurrency. That is, the system has bounded input, bounded output stability for the changes of the concurrency m_2. Besides, the absolute value of control error of the latency gradually decreases until $\rho e(k)\boldsymbol{\Phi}(k)\hat{\boldsymbol{\Phi}}^{\mathrm{T}}(k)/(\lambda+\|\hat{\boldsymbol{\Phi}}(k)\|^2) = \rho\boldsymbol{\Phi}(k)\boldsymbol{\varphi}^{\mathrm{T}}/(2(\lambda + \|\hat{\boldsymbol{\Phi}}(k)\|^2)$.

The analysis is as follows. Substitute $\Delta y(k+1) = \boldsymbol{\Phi}(k)\Delta\boldsymbol{u}(k)$ into Eqn. (9) to get

$$\begin{aligned} e(k+1) &= e(k) - \boldsymbol{\Phi}(k)\Delta\boldsymbol{u}(k) \\ &= e(k) - \frac{\rho\boldsymbol{\Phi}(k)\hat{\boldsymbol{\Phi}}^{\mathrm{T}}(k)}{\lambda + \left\|\hat{\boldsymbol{\Phi}}(k)\right\|^2}e(k) \\ &\quad + \frac{\rho\boldsymbol{\Phi}(k)\boldsymbol{\varphi}^T}{2(\lambda + \left\|\hat{\boldsymbol{\Phi}}(k)\right\|^2)}, \end{aligned} \tag{15}$$

where the dimension of $e(k)$ is 1×1, the dimension of $\boldsymbol{\Phi}(k)$ is 1×3, and the dimension of $\boldsymbol{\varphi}$ is 1×3. When it satisfies $[\rho\boldsymbol{\Phi}(k)\boldsymbol{\varphi}^{\mathrm{T}}/2(\lambda+\|\hat{\boldsymbol{\Phi}}(k)\|^2) - \rho e(k)\boldsymbol{\Phi}(k)\hat{\boldsymbol{\Phi}}^{\mathrm{T}}(k)/(\lambda+\|\hat{\boldsymbol{\Phi}}(k)\|^2)]e(k) < 0$, the error will decrease.

For control systems, the value of $|\boldsymbol{e}(k)|$ at the initial moment of interference is large. If the entries in $\boldsymbol{\varphi}$ are small enough, the error decreases until $\rho e(k)\boldsymbol{\Phi}(k)\hat{\boldsymbol{\Phi}}^{\mathrm{T}}(k)/(\lambda + \|\hat{\boldsymbol{\Phi}}(k)\|^2) = \rho\boldsymbol{\Phi}(k)\boldsymbol{\varphi}^{\mathrm{T}}/(2(\lambda + \|\hat{\boldsymbol{\Phi}}(k)\|^2)$.

The initial error of latency can be calculated by the $M/M/1$ model in the queueing theory. The average latency in the $M/M/1$ model can be calculated by $1/(\theta - \gamma)$, where γ is the arriving rate according to a Poisson process, and θ is the service time. y_1 is the latency before the sudden change of the system. The request arrival rate is the concurrency divided by the average latency, which is m_1/y_1. It has

$$y_1 = \frac{1}{\theta - \frac{m_1}{y_1}}. \tag{16}$$

So $\theta = (1+m_1)/y_1$. y_2 is the latency after the sudden change of the system. The request arrival rate is m_2/y_2. It has

$$y_2 = \frac{1}{\theta - \frac{m_2}{y_2}}. \tag{17}$$

After transformation, we can get

$$y_2 = \frac{1+m_2}{1+m_1}y_1. \tag{18}$$

As indicated by the Eqn. (15), if the entries in $\boldsymbol{\varphi}$ are small enough, the error will gradually decrease until $\rho e(k)\boldsymbol{\Phi}(k)\hat{\boldsymbol{\Phi}}^{\mathrm{T}}(k)/(\lambda + \|\hat{\boldsymbol{\Phi}}(k)\|^2) = \rho\boldsymbol{\Phi}(k)\boldsymbol{\varphi}^{\mathrm{T}}/(2(\lambda + \|\hat{\boldsymbol{\Phi}}(k)\|^2)$. So for $\forall k > k_{change}$, the latency y_2 satisfies Eqn. (14).

4.3. Latency Measurement Filtering

In Sonnet, the controller takes the $e(k) = y^*(k+1) - y(k)$ as the input, then outputs the resource allocation strategy. Obviously, we need to measure the latency $y(k)$. However, due to the bugs in hardware or software, the measurements are always with noise. In control systems, the noise will bring great challenges to the controller design. And if it is not handled well, the control system will oscillate, or even

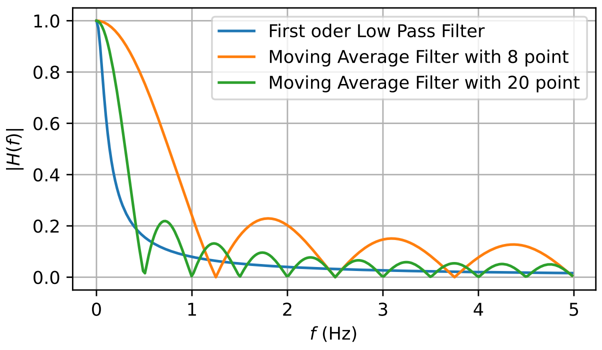


Figure 6: Frequency response of the moving average filter and low pass filter.

Algorithm 1 Sonnet's workflow

```
1:  Set the resource cost vector $\boldsymbol{\varphi}$
2:  Set the target latency SLO $y^*$
3:  Initialize $\boldsymbol{\Phi}(\mathbf{0})$, $\boldsymbol{u}(0)$, $\eta$, $\mu$, $\rho$, $\lambda$
4:  for $k$ in $1, 2, \cdots$, do
5:      Generate the workload and measure the latency $y(k)$
6:      Filter $y$ based on Eqn. (23)
7:      Calculate $\Delta\boldsymbol{u}_k$ and $\Delta y_k$
8:      Estimate system model $\hat{\boldsymbol{\Phi}}(k)$ By Eqn. (6)
9:      Calculate $e = y^*(k+1) - y(k)$
10:     Calculate CPU, memory, and network bandwidth allocation $\hat{\boldsymbol{u}}(k)$ using Eqn. (13)
11:     Use Docker API to update CPU and memory
12:     Use Linux traffic control to isolate net bandwidth
13: end for
```

unstable. Therefore, in order to ensure the performance of Sonnet, we need a filter to eliminate the noise.

As shown in Fig. 6, there are two kinds of widely used filters: moving average filter and first-order low pass filter. The moving average filter is

$$y_{out}(k) = \frac{1}{M} \sum_{j=k-M}^{k} y_{in}(j), \tag{19}$$

where $y_{out}(k)$ is the output of the filter at time k, which is the filtered latency. $y_{in}(\cdot)$ is the input signal, which is the latency measurement. M is the number of points used in the moving average filter.

Due to slow roll-off and poor stopband attenuation, as shown in Fig. 6, the moving average is a very poor low-pass filter. So we take the first-order low pass filter to filter the latency measurement, The filtering strength of high-frequency signals is determined by the filter coefficient τ and the sampling period of the signal. The equation of the first-order low pass filter is

$$\tau \frac{dy_{out}}{dt} + y_{out} = y_{in}, \tag{20}$$

where y_{out} is the output of the filter, which is the filtered latency. y_{in} is the input signal, which is the latency measurement. dy_{out} represents the differential of the output signal, while dt represents the differential of time. Performing Laplace transform [35] on Eqn. (20), we get the transfer function as

$$G(s) = \frac{y_{out}(s)}{y_{in}(s)} = \frac{1}{\tau s + 1}. \tag{21}$$

Transforming the transfer function $G(s)$ from the S domain to the Z domain using the first-order backward difference method, we have

$$G(z) = \frac{Y_{out}(z)}{Y_{in}(z)} = \frac{T}{\tau\left(1 - z^{-1}\right) + T}, \tag{22}$$

Performing an inverse Z-transform on Eqn. (22), we can get

$$y_{out}(k) = \frac{T}{\tau + T} y_{in}(k) + \frac{\tau}{\tau + T} y_{out}(k-1), \tag{23}$$

where $y_{in}(k)$ is the measured latency at time k and $y_{out}(k-1)$ is the filtered result at time $k-1$. In this paper, we set $T/(\tau+T) \in [0.2, 0.5]$. From the experiments, we find that the low-pass filter can effectively eliminate measurement noise.

4.4. Implementation

We implement Sonnet on Docker Swarm cluster. The user presets the cost vector $\boldsymbol{\varphi}$ according to the resources price. As shown in Fig 4, firstly we use open source benchmark tools such as Apache HTTP server [26], Scikit-learn [36], and memtier [25] to generate workloads and measure quality of service (e.g., latency). Latency measurement is always with noise due to bugs in hardware and software. We use a low pass filter introduced in Section 4.3 to eliminate noise.

Secondly, we use dynamic linearization to estimate system model. Model estimator needs the historical control input and system output information. So we collect the data of historical resource allocation and latency. After estimating the PJM $\hat{\boldsymbol{\Phi}}(k)$, then we calculate the resource allocation vector $\hat{\boldsymbol{u}}(k)$. Eqn. (13) shows resource allocation at present related to historical resource allocation $\hat{\boldsymbol{u}}(k-1)$, estimated PJM $\hat{\boldsymbol{\Phi}}(k)$, cost vector $\boldsymbol{\varphi}$, target SLO $y^*(k)$, and measured latency $y(k)$.

Finally, we update the resource allocation according to $\hat{\boldsymbol{u}}(k)$. Currently, a large number of applications are deployed on the same server, and they compete for shared resources (e.g., network bandwidth) from each other. In this paper, for CPU and memory, we use Docker to isolate the resources, and Docker API to update the corresponding resources. Currently, Docker still lacks the ability to limit network bandwidth of containers [29]. As has been discussed in Section 3.2, the lack of network bandwidth isolation will lead to mutual interference between applications. In this paper, we use Linux traffic control [30] to limit the bandwidth of the virtual network card of each Docker container. In this way,

Table 3
Characteristics of the workloads

App	workload	load variation	range
Nginx	ab	concurrency	8-20
ML	-	RF trees	25-70
Memcached	Memtier Benchmark	concurrency	10-25
Redis	Memtier Benchmark	concurrency	10-25

we isolate shared network resources between containers. The workflow of Sonnet is shown in Algorithm 1. In line 1-2, the user presets the resources cost vector $\boldsymbol{\varphi}$ and target SLO y^*. And in line 3, the user gives the initial resource allocation strategy $\boldsymbol{u}(0)$ and initializes η, μ, ρ, λ, and $\boldsymbol{\Phi}(0)$. In line 5-6, the workloads are generated. After the latency been measured, we need to filter the latency. In line 7-9, the resource allocation strategy $\boldsymbol{u(k)}$ is calculated. And in line 11-12, the resources are allocated by Docker and Linux traffic control.

5. Performance Evaluation

To evaluate the performance of the proposed algorithm, we compare it against the existing state-of-the-art approaches. In this section, we first describe the experimental settings, followed by the experimental results and analysis.

5.1. Experimental Settings

We use 4 HUAWEI cloud HECS servers that constitute a Docker Swarm cluster with a single master. The servers are Intel(R) Xeon(R) CPU E5-2680 v4 Processors with: (1) a frequency of 2.40GHz, (2) 8 cores, (3) 32GB of memory. Four popular applications have been used in our experiment to evaluate the performance of Sonnet.

Redis[37]: an in-memory data structure store, used as an in-memory key–value database, cache and message broker. This application is CPU and network bandwidth intensive.

Memcached[27]: a memory-caching system which is used to speed up dynamic database-driven websites by caching data and objects in RAM. This application is CPU and network bandwidth intensive.

Nginx[38]: a web server, load balancer, and HTTP cache. This application is CPU and network bandwidth intensive.

ML: we use scikit-learn[36] to train Random Forest[39]. This application is CPU and memory intensive.

We use *latency* as SLO metric. As shown in Table 3, we use Apache HTTP server benchmarking tool (ab) [26] and Memtier Benchmark [25] as workload generators for Redis, Nginx and Memcached. We run the benchmarking tool for 5 seconds each time and take the average latency as the measurement metric. The object data size for Memtier Benchmark is 32. The ratio of Set: Get is 1:10 for Memcached, and 1:1 for Redis. The ab benchmarking processes [26] constantly issue a uniformly random number of concurrent requests in batches.

Concurrency means the number of requests to perform at the same time. We change the loads of Nginx, Memcached, and Redis by changing the concurrency. When resource allocation is fixed, latency will increase as load rising. For ML application, we change its load by changing the number of Random Forest trees.

We compare our proposed method with two state-of-the-arts and one baseline.

Parties [6]: it is a quality of service aware resource manager that minimizes resource consumption without violating SLO. Parties tests different resource configurations multiple times and then increases or decreases resources of applications.

Skynet [3]: this is an automated and adaptive approach to cloud resource management. Skynet estimates the resources required to achieve the target SLO and uses augmented Ziegler-Nichols [40] PID controllers to allocate the resources.

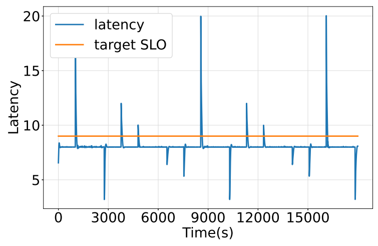
Oracle: we have developed an Oracle resource allocator which is capable of allocating resources with near-perfect accuracy. Since it takes a lot of time to collect data, it is not feasible in practical scenarios. We use it to measure the gap between Sonnet and the best allocation. To achieve this, we employ an offline grid search profiling approach, gathering a total of $20 \times 20 \times 20 \times 5 \times 4 = 160,000$ data points for each application(20 for the CPU, 20 for the memory, 20 for the network bandwidth, 4 for different loads settings. We repeated each configuration 5 times and set the mean value as the evaluation). The points that do not violate the target SLO with the smallest resource cost are set as Oracle.

Finally, we set the load changes as a step signal. The range of load variation is shown in Table 3. As shown in Fig. 7b, the load will complete 14 step changes in 5 hours. The load changes about every 21.5 minutes on average. In the most extreme case, the load will increase or decrease 2.8 times at an instant. For Memcached and Redis, the $T/(RC+T)$ in Section 4.3 is set as 0.5, while for Nginx, it is set as 0.3. And for ML, it is set as 0.2. The periods of performance measurement and resource adaptation are both 7 seconds for Nginx, Redis, and Memcached. This means that every 7 seconds, Sonnet checks the average latency and adjusts resource allocation. Because ML has a longer latency and requires more measurement time, the resource adaptation for ML is up to 60 seconds. Commercial cloud services (i.e., Google Cloud [41]) also support per-second billing. We change the load rapidly and drastically to simulate more extreme application scenarios.

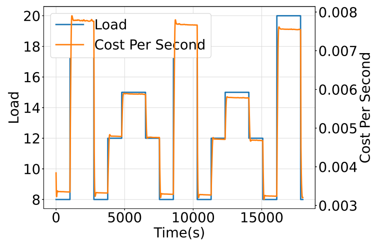
5.2. Convergence Evaluation

To verify that Sonnet can adapt to dynamic workloads, we conducted a 5-hour evaluation on Nginx. As shown in Fig. 7, Sonnet can estimate the system model during load changing. With the estimated model, Sonnet dynamically controls the resource allocation to adapt to load changes.

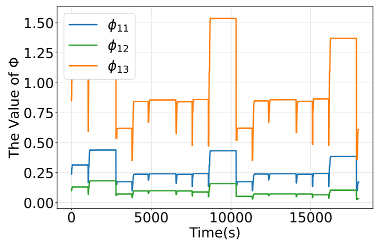
Fig. 7(a) shows the request latency of Nginx. We want to minimize the control objective in Eqn. (1). To avoid excessive cost increases, there are still a small number of SLO violations when the load changes suddenly. But, as



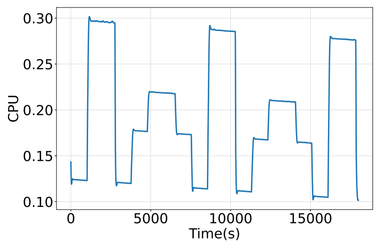
(a) The latency of Nginx



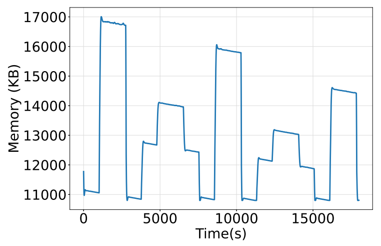
(b) The dynamic load and costs



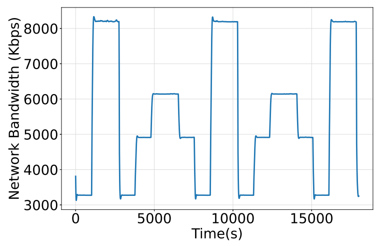
(c) Adaptive Estimation of System State



(d) CPU of Nginx

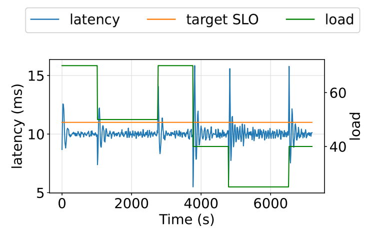


(e) Memory of Nginx

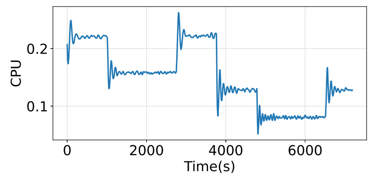


(f) Network Bandwidth of Nginx

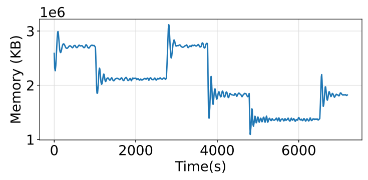
Figure 7: Sonnet adapts to changing load.



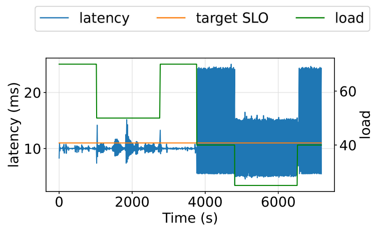
(a) Latency of ML with filter



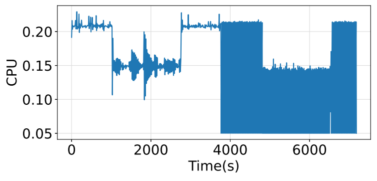
(b) CPU of ML with filter



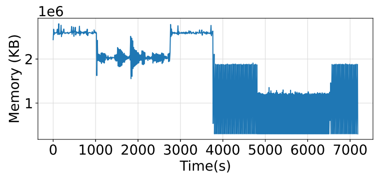
(c) Memory of ML with filter



(d) Latency of ML without filter



(e) CPU of ML without filter

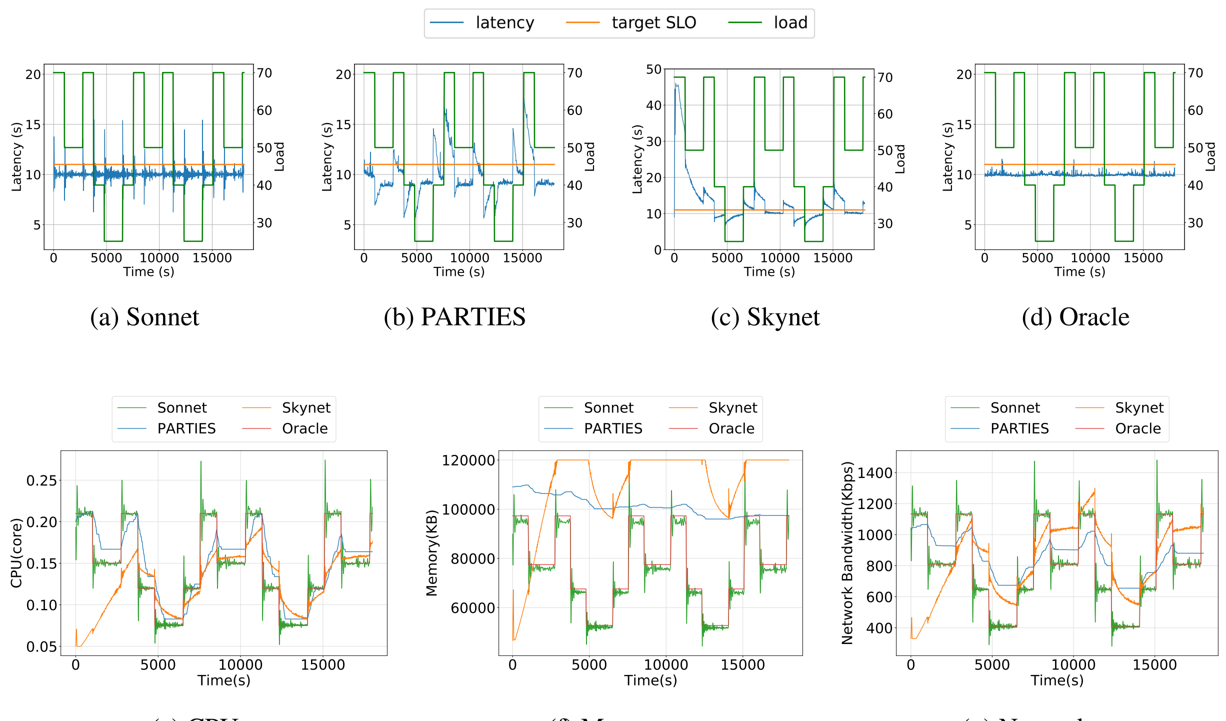


(f) Memory of ML without filter

Figure 8: The low-pass filter can filter out measurement noise for ML applications, thereby avoiding unnecessary resource allocation fluctuations.

Table 4
Performance Metric

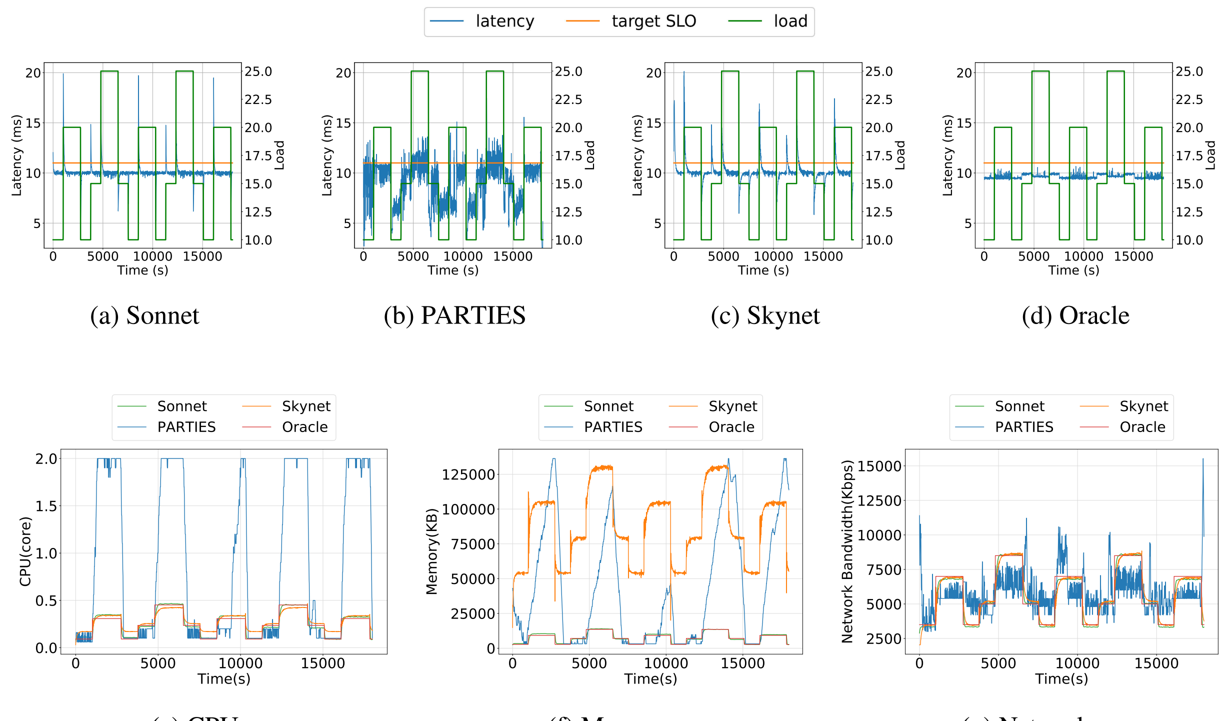
	Total costs				SLO violation rate			
App	Sonnet	Oracle	PARTIES	Skynet	Sonnet	Oracle	PARTIES	Skynet
Memcached	954.95	967.47	957.77	998.31	2.88%	0.19%	17.65%	9.10%
ML	154.36	154.54	170.64	168.02	4.63%	0.19%	20.92%	50.08%
Nginx	992.94	967.09	1874.85	1001.89	1.47%	0.04%	4.04%	7.62%
Redis	1007.84	1028.11	1034.85	1034.79	4.19%	0.0%	14.04%	8.90%



(a) Sonnet (b) PARTIES (c) Skynet (d) Oracle

(e) CPU (f) Memory (g) Network

Figure 9: Latency and resources of ML when varying the load of application.



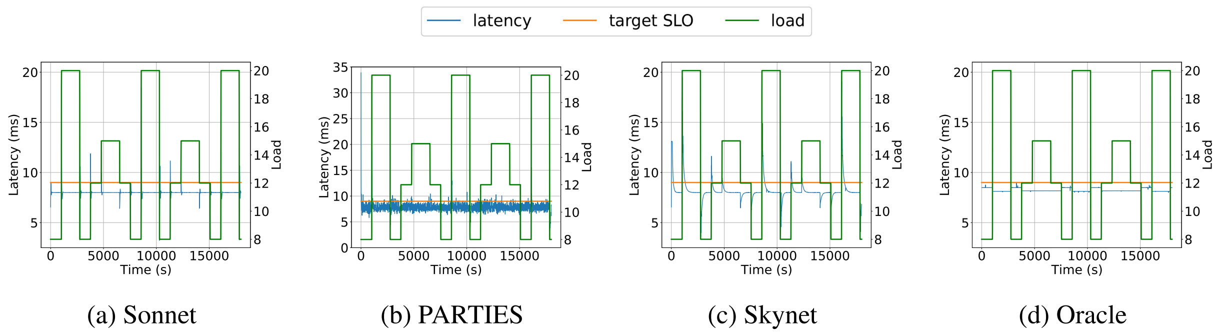
(a) Sonnet (b) PARTIES (c) Skynet (d) Oracle

(e) CPU (f) Memory (g) Network

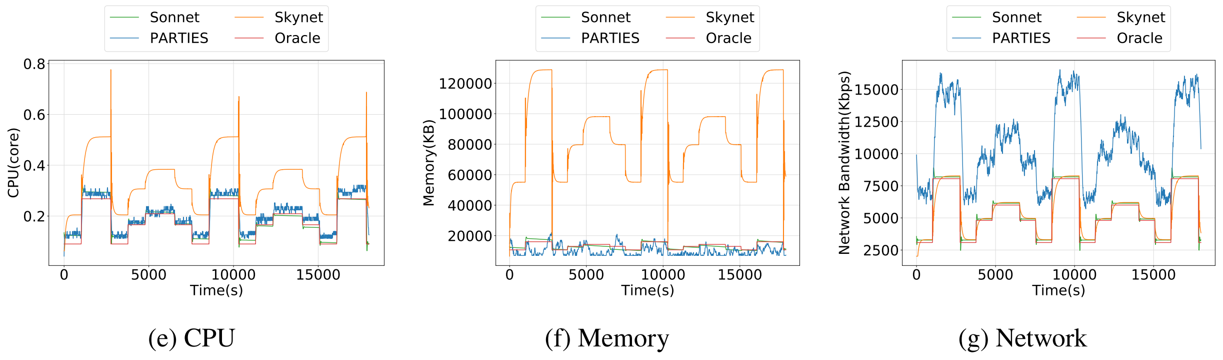
Figure 10: Latency and resources of Redis when varying the load of application.

shown in Fig. 7(a), Sonnet can quickly adjust the resource allocation to pull back the latency to meet the latency SLO.

Fig. 7(c) shows the system model estimation $\hat{\mathbf{\Phi}}$ calculated by Eqn. (2). The estimation varies as load changes. When the load changes, Sonnet can quickly update the estimated value of the system, and then converge to a constant quickly. At around 10500 seconds, the load drops sharply, and so does our estimated system model. At around $16,000$



(a) Sonnet (b) PARTIES (c) Skynet (d) Oracle



(e) CPU (f) Memory (g) Network

Figure 11: Latency and resources of Nginx when varying the load of application.

Table 5
Algorithms Execution Time

Algorithm	Execution time (Seconds)
Sonnet	$3.03 \times 10^{-3} \pm 1.30 \times 10^{-4}$
Skynet	$9.18 \times 10^{-4} \pm 5.76 \times 10^{-5}$
PARTIES	4.67 ± 0.44

seconds, the load suddenly increases, and our estimation can also follow the changes quickly.

5.3. The Effectiveness of Filter

Owing to server hardware and software issues, service performance metrics are often corrupted by noise. Fig. 9(a) shows that we configure the application's resources to an almost optimal constant configuration, the performance of the application is full of fluctuations. We apply a filter to reduce the effect of noise.

Fig. 8 compares the performance of sonnet retaining filter and removing filter under ML application. As shown in Fig. 8(a), at around 2000 seconds, the latency measurement is full of fluctuations. If the low-pass filter is removed, the controller in Section 4-2 will suffer from frequent, unnecessary resource allocation adjustments as illustrated in Fig. 8(e). As shown in Fig. 8(d), at about 2000 seconds, after removing the low-pass filter, the fluctuation of the latency becomes larger. What's more, at about 3800 seconds, due to the drastic change of the load, the resource allocation of the controller also changes drastically. When the filter is removed, as shown in Fig. 8(d), the system becomes unstable.

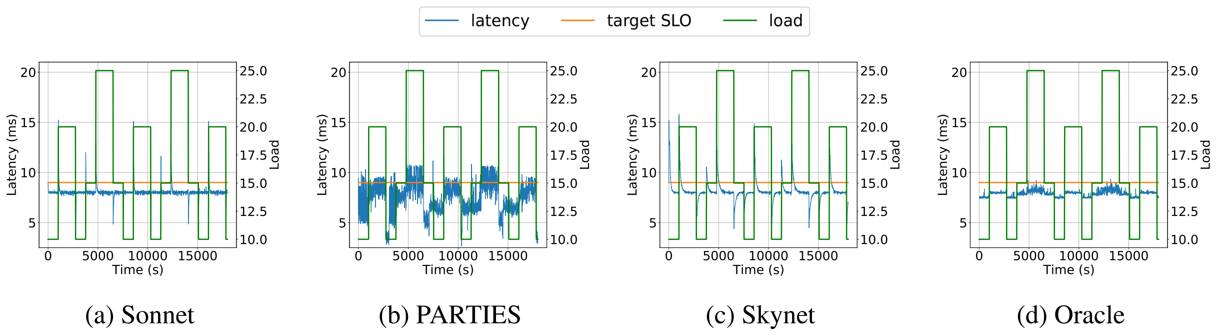
5.4. Experimental Results and Analysis

On Docker Swarm cluster, we compared the performance of Sonnet, PARTIES, Skynet, and Native methods on ML, Redis, Nginx, and Memcached applications. The resource cost vector $\boldsymbol{\varphi}$ in Eqn. (1) is set as [CPU, memory, network bandwidth] = [3.6, 36, 36], which means you need to pay 3.6 to use 1 CPU per hour, 36 to use 1 GB memory per hour and 36 to use 1 Mbps network bandwidth per hour. In the calculation, we convert the resource cost vector to per second. The total cost in Table 4 integrates the resource usage by time and obtains the total cost values.

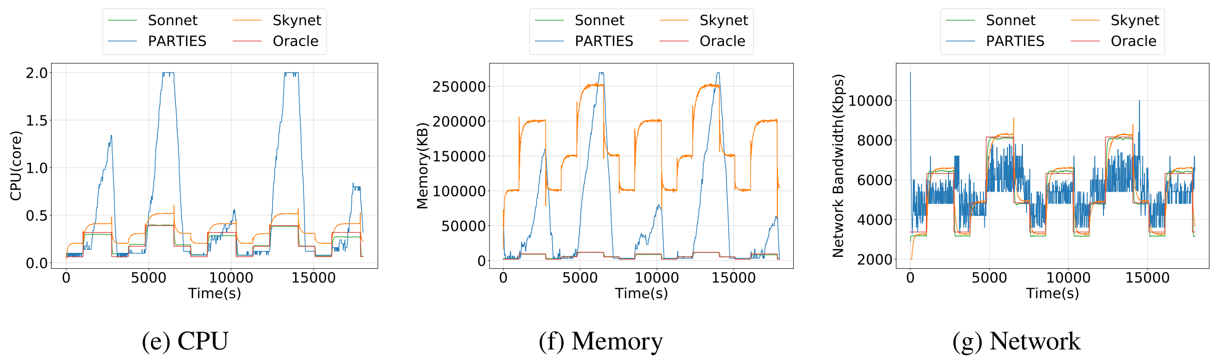
For Redis, ML, and Memcached applications, the μ and λ have been set to 15. And for Nginx, the μ is set as 6 while λ remains at 15. To avoid SLO violations, we set y^* in Eqn. (7) about 10% smaller than the target SLO. This is a trick to reduce SLO violations. As shown in Fig. 9(a), the measured values of most system noise will fluctuate within 10%. Therefore we set y^* in Eqn. (7) about 10% smaller than the target SLO to avoid defaults caused by noise fluctuations. The experimental results are shown in Table 4 and Fig. 9-12.

Sonnet Effectiveness: In the four applications, Sonnet's SLO violation rate is below 5% when the load is changed rapidly and drastically. This means that Sonnet's resource allocation can response quickly to load changes. Compared with stat-of-the-arts, Sonnet can reduce SLO violation rate by 91% at most, and reduce resource cost by 47%.

As shown in Fig. 9(a), Fig. 10(a), Fig. 11(a), and Fig. 12(a), when the load changes suddenly, Sonnet can quickly adjust the resource allocation to let the latency meet the target SLO. Under Sonnet's control, the application's latency remains constant with little jitter. Comparing



(a) Sonnet (b) PARTIES (c) Skynet (d) Oracle



(e) CPU (f) Memory (g) Network

Figure 12: Latency and resources of Memcached when varying the load of application.

Fig. 10(a) and Fig. 10(d), the performance fluctuations of the two are at the same level, and the latency fluctuation of Sonnet is close to that of Oracle.

Fig. 8(a) and 9(a) show that a latency violation occurs when the load drops suddenly. This phenomenon is common in control systems and is known as overshoot. In control theory, overshoot means that when there is a step change in the controlled system or reference signal, the signal will exceed the target value. In compute resource allocation, when the load decreases, the controller may reduce the compute resource allocation too much in a short period of time, resulting in SLO violations. Damm et al. [42] provided the conditions for overshoot occurrence.

In Figs. 9-12, the resource usage consumed by 4 methods is also displayed. In Fig. 12, Sonnet's and Oracled's configurations are the closest for each resource type. It is because Skynet tends to overprovision resources and the PARTIES configuration is full of jitter.

Comparison of Algorithm Execution Time: In general, the algorithm execution times for Sonnet and Skynet can be considered negligible, while PARTIES requires a longer time. As shown in Table 5, Sonnet's algorithm requires about 3 milliseconds of processing time when executed in Python on Intel(R) Xeon(R) CPU E5-2680 v4. Sonnet contains only the Eqn. (6) and Eqn. (13). The calculation involves several matrix multiplications and additions. The vector dimensions are only three-dimensional, so the computation time is extremely short.

Similarly, Skynet [3] requires about 1 millisecond as shown in Table 5. Skynet is equipped with an AZNPID for each resource, and the computation process involves only a few numerical multiplications and additions. PARTIES performs multiple tests on different resource configurations, requiring a few seconds for the resource configuration [6].

Comparison with Oracle: The Oracle resource allocator fits the fully collected offline data, which can maintain the SLO performance stability of the application when the load changes, and save resource costs as much as possible. We compare the performance difference between Oracle and Sonnet to measure the performance of Sonnet.

In Memcached, ML, Nginx, Redis applications, the total resource cost of Sonnet is only 1.31%, 0.12%, 2.67%, 1.97% different from that of Oracle, respectively. This means that the resource allocation of Sonnet is close to optimal. Comparing Fig. 10(a) and Fig. 10(d), the noise fluctuations of Sonnet and Oracle are roughly the same.

Comparison with PARTIES: PARTIES tests different resource configurations many times to find the resources that can improve the performance of the application. PARTIES relies on tricks set by experts, making it difficult to generalize to other scenarios.

For example, as shown in Fig. 9(b), the method of multiple testing slows down the response speed of the controller. If the resource allocation adjustment step size is too long, the oscillation will occur, resulting in a larger SLO violation rate. If the adjustment step size of the resource allocation is set too short, measurement noise causes PARTIES to fail to identify the resources that have the greatest impact on the application, resulting in larger costs. Comparing Sonnet with PARTIES, in Nginx, the total costs of PARTIES is 88% larger than Sonnet, and the SLO violation rate is 175% larger than Sonnet.

Comparison with Skynet: Skynet uses Ziegler-Nichols PID controllers, and it takes time to adjust the three gain parameters of proportional(P), integral(I), and differential(D). In ML applications, due to the rapid and drastic changes in the load, the adaptive changes of Skynet parameters cannot keep up with the load changes. As shown in Fig. 9(c), Skynet is trying to track load changes by adjusting resource allocation. Unfortunately, in ML applications, Skynet is too slow to adjust resource allocation, resulting in load changes again before the target SLO is reached. As shown in Table 4, in the ML application, the SLO violation rate is too high due to the slow adjustment speed of Skynet. Skynet SLO violation rate in the ML application is 9.81 times larger than Sonnet, even exceeds Native which is not scaling containers. In Nginx, Memcached and Redis applications, Skynet's SLO violation rate is larger than Sonnet's due to the same problem. In the Nginx application, Skynet SLO violation rate is 418% larger than Sonnet.

Since the PID controller is a SISO system, Skynet uses a set of PID controllers to control CPU, memory, and network bandwidth, respectively. When the application has SLO violations, the adaptive PID will increase the gain of all the PID controllers proportionally at the same time, thereby increasing the CPU, memory and network bandwidth nearly proportionally at the same time. Therefore, Skynet cannot achieve the optimal allocation of high-dimensional resources as mentioned in Section 3.3. This leads to the need of experts' experience to carefully set bigger initial gains for the PID which controls resources has a higher impact on application performance.

In conclusion, because Skynet has slower adaptive parameter tuning speed and cannot achieve the optimal allocation of high-dimensional resources, in Table 4, Sonnet performs better than Skynet in both SLO violation rate and total costs.

6. Conclusion

This paper is motivated by how to allocate the resources to each application to reach the target SLO with cost-effectiveness. As the application always has dynamic load changes, interference with others, and high-dimensional resources, blindly overprovisioning the resources to each application to ensure the target SLO will lead to a massive waste of resources, resulting in excessive costs. In this paper, we propose to improve the effectiveness of the cluster management by carefully controlling the resource allocation strategy. The unawareness of the system characteristics prompts us to use the control-theoretic approach to design resource allocator. Therefore, we propose Sonnet, an end-to-end online resource allocation system, which does not require offline data collection. The model-free adaptive control algorithm has been used to build a dynamic mapping model between the target SLO and resources. Since Sonnet does not require any prior knowledge, it can be quickly adapted to various applications. The final experiments based on four open-source benchmarks further demonstrate the superiority of our methods as compared with the state-of-the-arts.

In the future, we intend to extend our work in the following ways: (1) With the widespread use of AI applications such as ChatGPT and its stable spread, AI applications require a lot of computing resources. This requires us to finely allocate more types of resources such as GPU and GPU memory. (2) The emergence of technologies such as AWS Step Functions [43] have led to the integration of containerized applications into directed acyclic graph (DAG) workflows. This integration presents new resource allocation challenges, particularly in determining the optimal latency division for individual nodes within the DAG. (3) To improve resource utilization, it has become a new trend for clusters to co-locate placing batch jobs and online services [44]. Batch jobs will cause a performance degradation to online services, which can be regarded as disturbances in control theory, and can be solved by control theory.

References

[1] C. Delimitrou, C. Kozyrakis, Quasar: resource-efficient and qos-aware cluster management, in: R. Balasubramonian, A. Davis, S. V. Adve (Eds.), Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014, ACM, 2014, pp. 127–144. doi:10.1145/2541940.2541941.
URL https://doi.org/10.1145/2541940.2541941

[2] C. Delimitrou, C. Kozyrakis, Paragon: Qos-aware scheduling for heterogeneous datacenters, in: V. Sarkar, R. Bodík (Eds.), Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013, ACM, 2013, pp. 77–88. doi:10.1145/2451116.2451125.
URL https://doi.org/10.1145/2451116.2451125

[3] Y. Sfakianakis, M. Marazakis, A. Bilas, Skynet: Performance-driven resource management for dynamic workloads, in: C. A. Ardagna, C. K. Chang, E. Daminai, R. Ranjan, Z. Wang, R. Ward, J. Zhang, W. Zhang (Eds.), 14th IEEE International Conference on Cloud Computing, CLOUD 2021, Chicago, IL, USA, September 5-10, 2021, IEEE, 2021, pp. 527–539. doi:10.1109/CLOUD53861.2021.00069.
URL https://doi.org/10.1109/CLOUD53861.2021.00069

[4] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, J. Wilkes, Autopilot: workload autoscaling at google, in: A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, M. I. Seltzer (Eds.), EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020, ACM, 2020, pp. 16:1–16:16. doi:10.1145/3342195.3387524.
URL https://doi.org/10.1145/3342195.3387524

[5] A. F. Baarzi, G. Kesidis, SHOWAR: right-sizing and efficient scheduling of microservices, in: C. Curino, G. Koutrika, R. Netravali (Eds.), SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021, ACM, 2021, pp. 427–441. doi:10.1145/3472883.3486999.
URL https://doi.org/10.1145/3472883.3486999

[6] S. Chen, C. Delimitrou, J. F. Martínez, PARTIES: qos-aware resource partitioning for multiple interactive services, in: I. Bahar, M. Herlihy, E. Witchel, A. R. Lebeck (Eds.), Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019, ACM, 2019, pp. 107–120. doi:10.1145/3297858.3304005.
URL https://doi.org/10.1145/3297858.3304005

[7] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, R. K. Iyer, FIRM: an intelligent fine-grained resource management framework for slo-oriented microservices, in: 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event,

November 4-6, 2020, USENIX Association, 2020, pp. 805–825.
URL https://www.usenix.org/conference/osdi20/presentation/qiu

[8] F. Rossi, M. Nardelli, V. Cardellini, Horizontal and vertical scaling of container-based applications using reinforcement learning, in: E. Bertino, C. K. Chang, P. Chen, E. Damiani, M. Goul, K. Oyama (Eds.), 12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019, IEEE, 2019, pp. 329–338. doi:10.1109/CLOUD.2019.00061.
URL https://doi.org/10.1109/CLOUD.2019.00061

[9] Z. Hou, S. Jin, Model free adaptive control, CRC press Boca Raton, FL, 2013.

[10] N. Bashir, N. Deng, K. Rzadca, D. E. Irwin, S. Kodak, R. Jnagal, Take it to the limit: peak prediction-driven resource overcommitment in datacenters, in: A. Barbalace, P. Bhatotia, L. Alvisi, C. Cadar (Eds.), Proc. of ACM EuroSys, pp. 556–573.

[11] G. Yu, P. Chen, Z. Zheng, Microscaler: Automatic scaling for microservices with an online learning approach, in: E. Bertino, C. K. Chang, P. Chen, E. Damiani, M. Goul, K. Oyama (Eds.), 2019 IEEE International Conference on Web Services, ICWS 2019, Milan, Italy, July 8-13, 2019, IEEE, 2019, pp. 68–75. doi:10.1109/ICWS.2019.00023.
URL https://doi.org/10.1109/ICWS.2019.00023

[12] K. H. Chow, U. Deshpande, S. Seshadri, L. Liu, Deeprest: deep resource estimation for interactive microservices, in: Y. Bromberg, A. Kermarrec, C. Kozyrakis (Eds.), EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022, ACM, 2022, pp. 181–198. doi:10.1145/3492321.3519564.
URL https://doi.org/10.1145/3492321.3519564

[13] A. Mirhosseini, S. Elnikety, T. F. Wenisch, Parslo: A gradient descent-based approach for near-optimal partial SLO allotment in microservices, in: C. Curino, G. Koutrika, R. Netravali (Eds.), SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021, ACM, 2021, pp. 442–457. doi:10.1145/3472883.3486985.
URL https://doi.org/10.1145/3472883.3486985

[14] A. Gandhi, P. Dube, A. A. Karve, A. Kochut, L. Zhang, Adaptive, model-driven autoscaling for cloud applications, in: X. Zhu, G. Casale, X. Gu (Eds.), 11th International Conference on Autonomic Computing, ICAC '14, Philadelphia, PA, USA, June 18-20, 2014, USENIX Association, 2014, pp. 57–64.
URL https://www.usenix.org/conference/icac14/technical-sessions/presentation/gandhi

[15] J. Liu, S. Zhang, Q. Wang, J. Wei, Coordinating fast concurrency adapting with autoscaling for slo-oriented web applications, IEEE Trans. Parallel Distributed Syst. 33 (12) (2022) 3349–3362.

[16] Z. Wen, Y. Wang, F. Liu, Stepconf: Slo-aware dynamic resource configuration for serverless function workflows, in: IEEE INFOCOM 2022 - IEEE Conference on Computer Communications, London, United Kingdom, May 2-5, 2022, IEEE, 2022, pp. 1868–1877. doi:10.1109/INFOCOM48880.2022.9796962.
URL https://doi.org/10.1109/INFOCOM48880.2022.9796962

[17] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, C. Kozyrakis, Heracles: improving resource efficiency at scale, in: D. T. Marr, D. H. Albonesi (Eds.), Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015, ACM, 2015, pp. 450–462. doi:10.1145/2749469.2749475.
URL https://doi.org/10.1145/2749469.2749475

[18] A. Horn, H. M. Fard, F. Wolf, Multi-objective hybrid autoscaling of microservices in kubernetes clusters, in: J. Cano, P. Trinder (Eds.), Euro-Par 2022: Parallel Processing - 28th International Conference on Parallel and Distributed Computing, Glasgow, UK, August 22-26, 2022, Proceedings, Vol. 13440 of Lecture Notes in Computer Science, Springer, 2022, pp. 233–250. doi:10.1007/978-3-031-12597-3_15.
URL https://doi.org/10.1007/978-3-031-12597-3_15

[19] D. Dechouniotis, N. Leontiou, N. Athanasopoulos, A. Christakidis, S. Denazis, A control-theoretic approach towards joint admission control and resource allocation of cloud computing services, International Journal of Network Management 25 (3) (2015) 159–180.

[20] D. Spatharakis, I. Dimolitsas, E. Vlahakis, D. Dechouniotis, N. Athanasopoulos, S. Papavassiliou, Distributed resource autoscaling in kubernetes edge clusters, in: 2022 18th International Conference on Network and Service Management (CNSM), 2022, pp. 163–169. doi:10.23919/CNSM55787.2022.9965056.

[21] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, C. Delimitrou, Sinan: Ml-based and qos-aware resource management for cloud microservices, in: T. Sherwood, E. D. Berger, C. Kozyrakis (Eds.), ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021, ACM, 2021, pp. 167–181. doi:10.1145/3445814.3446693.
URL https://doi.org/10.1145/3445814.3446693

[22] A. Ullah, J. Li, Y. Shen, A. Hussain, A control theoretical view of cloud elasticity: taxonomy, survey and challenges, Cluster Computing 21 (2018) 1735–1764.

[23] B. Gregg, Linux bcc/bpf run queue (scheduler) latency, https://www.brendangregg.com/blog/2016-10-08/linux-bcc-runqlat.html, accessed July. 9, 2022.

[24] R. J. Meinhold, N. D. Singpurwalla, Understanding the kalman filter, The American Statistician 37 (2) (1983) 123–127.

[25] memtier_benchmark: A high-throughput benchmarking tool for redis & memcached, https://redis.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/, accessed July.10, 2022.

[26] ab apache http server benchmarking tool, https://httpd.apache.org/docs/2.4/programs/ab.html, accessed July.10, 2022.

[27] memcached - a distributed memory object caching system., https://memcached.org/, accessed July.13, 2022.

[28] D. Merkel, et al., Docker: lightweight linux containers for consistent development and deployment, Linux journal 239 (2) (2014).

[29] Z. WANG, C. YANG, Bandwidth control mechanism for docker container network based on traffic control, Journal of Computer Applications 39 (12) (2019) 3628–3632.

[30] M. A. Brown, Traic control howto, http://linux-ip.net/articles/Traffic-Control-HOWTO/, accessed July.11, 2022.

[31] Y. Liao, Q. Jiang, T. Du, W. Jiang, Redefined output model-free adaptive control method and unmanned surface vehicle heading control, IEEE Journal of Oceanic Engineering 45 (3) (2019) 714–723.

[32] S. Liu, Z. Hou, T. Tian, Z. Deng, Z. Li, A novel dual successive projection-based model-free adaptive control method and application to an autonomous car, IEEE Transactions on Neural Networks and Learning Systems 30 (11) (2019) 3444–3457.

[33] S. Liu, Z. Hou, X. Zhang, H. Ji, Model-free adaptive control method for a class of unknown mimo systems with measurement noise and application to quadrotor aircraft, IET Control Theory & Applications 14 (15) (2020) 2084–2096.

[34] C. Lu, Y. Zhao, K. Men, L. Tu, Y. Han, Wide-area power system stabiliser based on model-free adaptive control, IET Control Theory & Applications 9 (13) (2015) 1996–2007.

[35] P. A. Lynn, The laplace transform and the z-transform, in: Electronic Signals and Systems, Springer, 1986, pp. 225–272.

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

[37] Redis, https://redis.io/, accessed July.13, 2022.

[38] Nginx: Advanced load balancer, web server, & reverse proxy, https://www.nginx.com/, accessed July.13, 2022.

[39] L. Breiman, Random forests, Machine learning 45 (1) (2001) 5–32.

[40] C. C. Hang, K. J. Åström, W. K. Ho, Refinements of the ziegler–nichols tuning formula, in: IEE Proceedings D: Control Theory and Applications, Vol. 138, 1991, pp. 111–118.

[41] Google, Extending per second billing in google cloud, https://cloud.google.com/blog/products/gcp/extending-per-second-billing-in-google, accessed October. 9, 2023.

[42] T. Damm, L. N. Muhirwa, Zero crossings, overshoot and initial undershoot in the step and impulse responses of linear systems, IEEE Transactions on Automatic Control 59 (7) (2014) 1925–1929. doi:10.1109/TAC.2013.2294616.

[43] AWS, Aws step functions, https://aws.amazon.com/step-functions/, accessed September. 9, 2023.

[44] Q. Liu, Z. Yu, The elasticity and plasticity in semi-containerized co-locating cloud workload: a view from alibaba trace, in: Proceedings of the ACM Symposium on Cloud Computing, 2018, pp. 347–360.